

DISTRIBUTED RECONFIGURATION AND FAULT DIAGNOSIS IN CELLULAR PROCESSING ARRAYS

by

Shannon Edward Lawson

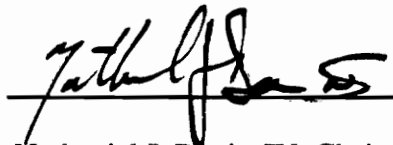
Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

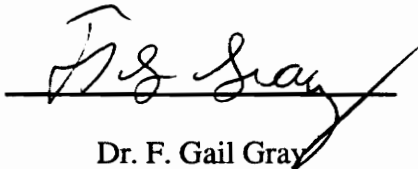
in

Electrical Engineering

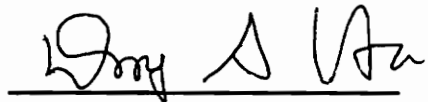
APPROVED:



Dr. Nathaniel J. Davis, IV, Chairman



Dr. F. Gail Gray



Dr. Dong S. Ha

January, 1993

Blacksburg, Virginia

C.2

LD
5655
V855
1993
L397
C.2

DISTRIBUTED RECONFIGURATION AND FAULT DIAGNOSIS IN CELLULAR PROCESSING ARRAYS

by

Shannon Edward Lawson

Committee Chairman: Dr. Nathaniel J. Davis, IV

Electrical Engineering

ABSTRACT

An overview of an existing hierarchical reconfiguration scheme for a fault-tolerant two-dimensional cellular architecture is presented, wherein an array of finite state machine cells controls the processing and switching elements. This allows the array to either reconfigure in the presence of faults, or to perform different processing functions. Since the control mechanism is distributed, the system is not subject to single-point "hard core" failures, as in the case of a global control mechanism. Unlike other fault-tolerant systems, the proposed method does not assume the existence of components which never fail.

The processing elements in the array are logically connected in a mesh pattern, and are provided with additional physical connections to other cells. A local reconfiguration scheme allows faulty cells to be bypassed via these additional connections, so that the logical mesh can be restored. This technique allows the array to quickly reconfigure in the presence of up to triple faults.

When local reconfiguration fails, the array can still reconfigure by using a global reconfiguration scheme, in which the functional part of the array relocates itself to a fault-free area. The process is "global" in the sense that the entire functional part of the array is involved in the process, but the mechanism to accomplish this is still distributed in nature.

With the framework of the system established, the results of this research are presented. The hardware complexities of the existing global reconfiguration scheme are analyzed, and compared with the complexities of previous work in this area. A distributed diagnosis algorithm is also developed, which works in conjunction with the local reconfiguration mechanism to quickly detect and isolate faults in the array. Using these results, the foundations are laid for a totally self-checking implementation of the control cells, which allows online concurrent fault detection in the array.

ACKNOWLEDGMENTS

The author wishes to thank the committee chairman, Dr. Nathaniel J. Davis, IV, for his guidance and support over the last 2 years. The author also wishes to thank the other committee members, Dr. F. Gail Gray and Dr. Dong S. Ha, for their instruction and assistance in preparing this thesis.

Special thanks go to Mrs. Virginia D. McWhorter, Assistant Department Head of the Bradley Department of Electrical Engineering, Dr. Joseph G. Tront, Assistant Dean in the College of Engineering, and Mr. Hugh W. Munson, Instructor in the General Engineering Department, for their faith in my potential.

Most of all, thanks go to my wife, Debra Ann Patterson Lawson, and my parents, Mr. W. E. Lawson and Mrs. Carolyn K. Lawson, for their love, patience, and understanding. This work could not have been done without you.

TABLE OF CONTENTS

- 1. Introduction..... 1**
 - 1.1 Statement of Purpose..... 3
 - 1.2 Thesis Overview 4

- 2. Background 6**
 - 2.1 Systolic Arrays 6
 - 2.2 Motivation for Reconfiguration..... 9
 - 2.2.1 Yield Enhancement..... 9
 - 2.2.1.1 Fault Tolerant Testable Scan Design..... 10
 - 2.2.1.2 The Kuo-Fuchs Repair-Most Strategy 16
 - 2.2.2 Runtime Fault Tolerance..... 21
 - 2.2.2.1 The Diogenes Approach 21
 - 2.2.2.2 Interstitial Redundancy 25
 - 2.2.2.3 Fault-Stealing Strategies..... 25
 - 2.2.3 Array Polymorphism..... 30
 - 2.2.3.1 The Configurable, Highly Parallel (CHiP) Computer 30
 - 2.2.3.2 Hierarchical Reconfiguration 32
 - 2.3 Global Reconfiguration..... 35
 - 2.3.1 Martin's Method 38
 - 2.3.2 Kumar's Method 38
 - 2.3.3 Brighton's Method..... 38
 - 2.3.3.1 Definitions of Terms Used..... 39
 - 2.3.3.2 Pattern Growth Example: The Banyan Network 40

2.3.3.3	Pattern Growth Rules	48
2.4	Local Reconfiguration	50
2.4.1	The Processing Element.....	50
2.4.2	The Fault Register	52
2.4.3	Fault Coverage	54
2.5	Summary	54
3.	Reconfiguration Complexity Analysis	57
3.1	Explanation of Symbols Used	57
3.2	Local Memory Complexity	58
3.2.1	Martin	58
3.2.2	Kumar	59
3.2.3	Brighton	63
3.3	Interconnection Complexity	64
3.3.1	Martin	64
3.3.2	Kumar	65
3.3.3	Brighton	65
3.4	Transmitted Data Complexity	66
3.4.1	Martin	66
3.4.2	Kumar	67
3.4.3	Brighton	67
3.5	Complexity Analysis Results	68
3.6	Conclusions	68

4. Distributed Fault Diagnosis	72
4.1 Motivation for Distributed Diagnosis	73
4.2 The Cell Neighborhood.....	77
4.3 The Fault Register	88
4.4 Fatal Fault Patterns	92
4.4.1 The L-fault	94
4.4.2 The V-fault and the /-fault	98
4.4.3 Other Problematic Triple Faults	98
4.5 Distributed Diagnosis Simulator.....	105
4.5.1 Single Fault Diagnosis	108
4.5.2 Double Fault Diagnosis.....	108
4.5.3 Triple Fault Diagnosis.....	112
4.5.4 Quadruple Fault Diagnosis.....	112
4.5.5 Diagnosis of 5 or More Faults.....	128
4.6 Reconfiguration Using Distributed Diagnosis	128
4.7 Reliability Analysis.....	145
4.8 Results.....	147
4.9 Conclusions	150
 5. Self-Checking Circuits.....	 153
5.1 Properties of Self-Checking Circuits	154
5.1.1 Totally Self-Checking (TSC) Circuits.....	154
5.1.2 Strongly Fault-Secure (SFS) Circuits	157
5.1.3 Strongly Code-Disjoint (SCD) Circuits	158

5.2	Coding Theory for Self-Checking Circuits.....	159
5.2.1	Berger Codes.....	161
5.2.2	Bose-Lin Codes.....	161
5.2.3	The Two-Rail Code.....	162
5.3	A Self-Checking Control Cell Architecture	162
5.3.1	The Incoming Data Error-Trap Circuitry.....	168
5.3.2	The Bidirectional I/O Channels	172
5.3.3	The Registers and Buffers.....	175
5.3.4	The RFSM.....	182
5.3.4.1	Sequentially Self-Checking (SeSC) Circuits	184
5.3.4.2	Strongly Language-Disjoint (SLD) Checkers	189
5.3.5	The Code Conversion Module	197
5.3.6	The TSC Checker for the Code Conversion Module	197
5.3.7	The Internal Fault Indicators.....	198
5.4	Results.....	198
5.5	Conclusions	202
6.	Conclusions.....	205
6.1	Summary of Accomplishments	205
6.2	Future Research Opportunities.....	208

Bibliography 209

Appendix A. Simulation Source Code 215

 A.1 Array.c 215

 A.2 Faultmap.c..... 239

Vita 247

LIST OF FIGURES

Figure 2.1	Example systolic array for performing linear convolution	8
Figure 2.2	A proposed augmented PE for yield enhancement	11
Figure 2.3	Possible settings for augmented PE switch SW	12
Figure 2.4	Simplified diagram of vertical input paths for augmented PE	14
Figure 2.5	Successfully reconfigured systolic matrix multiplier array	15
Figure 2.6	Repair-most strategy example with 2 spare rows and 2 spare columns	17
Figure 2.7	Successful reconfiguration of Figure 2.6 using repair-most strategy	19
Figure 2.8	Unsuccessful reconfiguration of Figure 2.6 using repair-most strategy	20
Figure 2.9	Hardware stack structure for the Diogenes approach	22
Figure 2.10	Hardware queue structure for the Diogenes approach	23
Figure 2.11	Example of interstitial redundancy	26
Figure 2.12	Example of direct reconfiguration (DR)	28
Figure 2.13	Example of complex fault stealing (CFS)	29
Figure 2.14	Pseudocode algorithm for complex fault stealing	31
Figure 2.15	CHiP architecture configured as a mesh array	33
Figure 2.16	CHiP architecture of Figure 2.15 configured as a binary tree	34
Figure 2.17	An array of PEs divided into control and computational planes	36
Figure 2.18	Banyan network used in pattern growth example	41
Figure 2.19	Control cell state assignments for the Banyan Network	42
Figure 2.20	XR, YR, and LSR contents during pattern growth for $t < 4$	44
Figure 2.21	XR, YR, and LSR contents at $t=9$	46
Figure 2.22	Final control pattern for the Banyan network example, $t > 9$	47

Figure 2.23	Cell interconnection pattern used in White's Algorithm 4-12.....	51
Figure 2.24	End result of local reconfiguration in response to a double fault	53
Figure 2.25	Fatal L-fault pattern example prior to local reconfiguration attempt.....	55
Figure 3.1	A diamond pattern of cells in a processor array	61
Figure 4.1	The zero fault latency model	74
Figure 4.2	The link fault cannot be detected by any neighbor of the PE other than N..	76
Figure 4.3	The von Neumann cell neighborhood has a scope of 5.....	78
Figure 4.4	The Moore cell neighborhood has a scope of 9.....	79
Figure 4.5	The White cell neighborhood has a scope of 13.....	80
Figure 4.6	The possible choices for a NORTH logical neighbor	82
Figure 4.7	The possible choices for an EAST logical neighbor	83
Figure 4.8	The possible choices for a SOUTH logical neighbor.....	84
Figure 4.9	The possible choices for a WEST logical neighbor	85
Figure 4.10	The Lawson cell neighborhood has a scope of 9.....	86
Figure 4.11	Setting the fault register bits.....	90
Figure 4.12	Distributed diagnosis diagram using a White neighborhood for the interconnection network, and a von Neumann neighborhood for passing the fault data	91
Figure 4.13	Distributed diagnosis diagram using a White neighborhood for both the interconnection network and for passing the fault data	93
Figure 4.14	Unsuccessful L-fault detection using a von Neumann neighborhood.....	95
Figure 4.15	Additional fault register bits needed to detect L-faults.....	96
Figure 4.16	Example of successful L-fault detection using the 20-bit fault register	97
Figure 4.17	Unsuccessful V-fault detection using a von Neumann neighborhood.....	99
Figure 4.18	Unsuccessful /-fault detection using a von Neumann neighborhood	100

Figure 4.19	Additional fault register bits needed to detect all triple faults	101
Figure 4.20	Successful detection of a V-fault using the 24-bit fault register.....	102
Figure 4.21	Successful detection of a /-fault using the 24-bit fault register	103
Figure 4.22	Triple fault which can be treated as a single fault and a double fault.....	104
Figure 4.23	Single fault detection	109
Figure 4.24	Double fault detection (Case 1).....	110
Figure 4.25	Double fault detection (Case 2).....	111
Figure 4.26	Tightly-clustered triple fault pattern (Case 1)	113
Figure 4.27	Tightly-clustered triple fault pattern (Case 2)	114
Figure 4.28	Tightly-clustered triple fault pattern (Case 3)	115
Figure 4.29	Tightly-clustered triple fault pattern (Case 4)	116
Figure 4.30	Tightly-clustered triple fault pattern (Case 5)	117
Figure 4.31	Fatal quadruple fault pattern (Case 1)	118
Figure 4.32	Fatal quadruple fault pattern (Case 2)	119
Figure 4.33	Fatal quadruple fault pattern (Case 3)	120
Figure 4.34	Fatal quadruple fault pattern (Case 4)	121
Figure 4.35	Fatal quadruple fault pattern (Case 5)	122
Figure 4.36	Fatal quadruple fault pattern (Case 6)	123
Figure 4.37	Fatal quadruple fault pattern (Case 7)	124
Figure 4.38	Fatal quadruple fault pattern (Case 8)	125
Figure 4.39	Fatal quadruple fault pattern using current local reconfiguration hardware (Case 1).....	126
Figure 4.40	Fatal quadruple fault pattern using current local reconfiguration hardware (Case 2).....	127
Figure 4.41	Properly detected, tightly clustered quadruple fault pattern (Case 1).....	129

Figure 4.42	Properly detected, tightly clustered quadruple fault pattern (Case 2).....	130
Figure 4.43	Properly detected, tightly clustered quadruple fault pattern (Case 3).....	131
Figure 4.44	Properly detected, tightly clustered quadruple fault pattern (Case 4).....	132
Figure 4.45	Properly detected, tightly clustered quadruple fault pattern (Case 5).....	133
Figure 4.46	Properly detected, tightly clustered quadruple fault pattern (Case 6).....	134
Figure 4.47	Properly detected, tightly clustered quadruple fault pattern (Case 7).....	135
Figure 4.48	Properly detected, tightly clustered quadruple fault pattern (Case 8).....	136
Figure 4.49	Properly detected, tightly clustered quadruple fault pattern (Case 9).....	137
Figure 4.50	Properly detected, tightly clustered quadruple fault pattern (Case 10).....	138
Figure 4.51	Properly detected, tightly clustered quadruple fault pattern (Case 11).....	139
Figure 4.52	Properly detected, tightly clustered quadruple fault pattern (Case 12).....	140
Figure 4.53	Properly detected, tightly clustered quadruple fault pattern (Case 13).....	141
Figure 4.54	Properly detected, tightly clustered quadruple fault pattern (Case 14).....	142
Figure 4.55	Properly detected, tightly clustered quadruple fault pattern (Case 15).....	143
Figure 4.56	Properly detected, tightly clustered quadruple fault pattern (Case 16).....	144
Figure 5.1	Basic self-checking circuit.....	155
Figure 5.2	TSC checkers for the two-rail code.....	163
Figure 5.3	Control cell interconnections for two-rail data transfers.....	165
Figure 5.4	Block diagram of the self-checking control cell architecture	167
Figure 5.5	The two-rail error-trap circuit	169
Figure 5.6	Block diagram of the bidirectional I/O channels for the control cell	173
Figure 5.7	The registers and buffers for the control cell.....	176
Figure 5.8	The SFS register bit-slice	181
Figure 5.9	Block diagram of a self-checking sequential system.....	183
Figure 5.10	Sequentially self-checking machines made from SFS circuits	188

Figure 5.11	Block diagram of a proposed SLD checker	193
Figure 5.12	Improved SLD checker design which eliminates SCD checkers	196

LIST OF TABLES

Table 3.1 Complexity of Global Reconfiguration Overhead 69

Table 4.1 Maximum fault latency times for up to quadruple faults..... 148

Table 5.1 Results of single stuck-at faults in the XNOR gate 170

Table 5.2 Results of single stuck-at faults in the XOR gate 171

1. INTRODUCTION

The advent of Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI) devices with high component densities makes possible the fabrication of powerful parallel processing multinode architectures on a single silicon substrate. These systems are often discussed in the context of signal processing applications, where the iterative nature of the computations involved is well suited to the regularity of layout desired in VLSI/WSI implementations [Kun82].

However, array processors are often special purpose in nature. This creates the need for many specialized architectures, each inflexible and uniquely suited to a particular task [Sny82]. Integrated circuits are also inherently difficult to repair; the failure of even a single transistor out of the millions available in the system can result in catastrophic failure. Moreover, such faults can manifest themselves either during production, or later in a runtime environment. Fortunately, all of these difficulties can be successfully addressed to some degree through the concept of reconfigurable architectures [Whi91].

Many strategies have been developed to reconfigure VLSI and WSI processor arrays, including [Sny82], [KiR89], [KuF87], [Ros83], and others. These systems employ redundant hardware in order to achieve the desired level of fault tolerance and reconfigurability. A control mechanism is responsible for managing redundant resources, as well as configuring processing and switching elements. Some reconfiguration schemes use a centralized, or global, control mechanism, such as in [ChF89], [SaS86], and [Sny82], whereas others utilize a distributed control strategy, including [Bri87], [GoG84], [GrW89], [Hos89], [Kum84], [MaG80], [Mar80], and [Whi91].

The use of a central control mechanism introduces the possibility of a single-point, or "hard core," failure in an otherwise fault-tolerant system. If the controller itself fails, the entire system fails. The issue of controller reliability is sometimes circumvented by either explicitly or implicitly assuming the existence of an external fault-free control mechanism [Ros83], [KiR89]. Such assumptions may be valid in a manufacturing environment, where multiple redundant controllers may exist, and where fault diagnosis and reconfiguration are less time-critical than in a runtime environment. However, these luxuries are not generally available outside a manufacturing context.

In [Hos89], a distributed fault diagnosis strategy is presented which does not require a central reconfiguration control mechanism. Unfortunately, the author makes the assumption that interprocessor communications links never fail, based on their relative simplicity and small layout requirements as compared to the processing elements. Research presented in [Mar80], [Kum84], [Bri87], [Whi91], and elsewhere, has focused on developing distributed reconfiguration strategies which do not rely on failure-proof components. A common theme throughout this work is an array containing redundant processing elements, only a subset of which are used at runtime. The set of all processing elements is called the *physical array*, and the structure needed for processing is called the *logical array*. The logical array may require a rectangular mesh, binary tree, or any other processor interconnection scheme. It is the task of the reconfiguration control mechanism to map the logical array into a functioning part of the physical array.

Through [Mar80], [Kum84], and [Bri87], a *global reconfiguration* scheme evolves. In response to a fault, the logical array is relocated to a fault-free area of the physical array. The same technique also provides array polymorphism, allowing a single

array to reconfigure itself for use in various specialized processing tasks. The method presented in [Bri87] has distinct advantages over those given in both [Mar80] and [Kum84], with respect to the amount of required hardware and layout area, and is therefore the most desirable approach of the three. When exclusively employed in response to faults, the global reconfiguration process is intrinsically costly in terms of the hardware and time overhead required to relocate the logical array [Whi91]. Thus, it is not necessarily suitable for use in all fault reconfiguration scenarios.

In [Whi91], a *local reconfiguration* strategy is presented, in which the tasks assigned to faulty processing elements are passed to neighboring elements, and spare elements are shifted in to satisfy the logical array size requirements. This technique is reminiscent of the "fault stealing" strategy presented in [SaS86], and allows the array to tolerate a small number of faults within the logical portion of the array. This approach provides relatively quick response to faults as they occur, as compared to the global reconfiguration approach. The local configuration strategy is 2-fault-tolerant, and also provides coverage for most triple-fault configurations [Whi91].

1.1 STATEMENT OF PURPOSE

The purposes of the research presented herein are threefold. The first purpose is to show the efficiency of the global reconfiguration scheme presented in [Bri87], as compared to [Mar80] and [Kum84], with respect to the size of local memory required by each processing element, the number of interconnection wires needed to provide a given level of fault tolerance, and the size of the data packets passed between neighbors to support each of the given schemes. The second purpose is to provide a distributed

mechanism for alerting processing elements to the existence of faulty neighboring processors, and consequently to provide coverage for all triple-fault patterns. The third purpose is to provide a concurrent fault detection mechanism, thus reducing fault latency as compared to periodic off-line testing schemes.

1.2 THESIS OVERVIEW

Chapter 2 provides a background on systolic array processors and the desirability of reconfigurable architectures. Examples which address each of the different reconfiguration scenarios are presented. Overviews of the global and local reconfiguration schemes discussed in [Bri87] and [Whi91], respectively, are included as well. The information provided in this chapter gives the reader the background necessary to understand the new results presented in the subsequent chapters.

Chapter 3 derives the relative hardware complexities of [Mar80], and [Kum84], and compares them to those of [Bri87]. It is shown that the latter scheme results in a reduced local memory for an array element, as compared to either of the other two methods. The method in [Bri87] also offers a small, fixed interconnect cost, independent of the degree of fault tolerance desired, as opposed to the results given in [Mar80]. Finally, the method given in [Bri87] requires smaller data packets than the method given in [Kum84].

Results presented in [Mar80], [Kum84], [Bri87], and [Whi91], assume the existence of some unspecified mechanism to detect faults in the array. Further, the assumption is made that all neighbors of a faulty cell become aware of that fault

simultaneously. Chapter 4 addresses these assumptions in the form of a distributed algorithm for fault diagnosis and isolation. It is shown that it is not necessary for all neighbors to know of the existence of a faulty cell simultaneously. Further, the time required for fault information to be disseminated to all neighbors of the faulty cell has an upper bound which depends on the interconnect scheme used to pass the location of the faulty cell. Another consequence of this analysis, performed in conjunction with Joseph Wegner, is a relatively inexpensive means to improve the reliability of the local reconfiguration scheme given in [Whi91] so that it tolerates all triple faults.

One of the major goals of this effort is the inclusion of concurrent, or on-line, fault detection. This approach reduces fault latency, as compared to off-line testing, as faults are detected as they occur. As a possible solution to problems associated with concurrent testing, Chapter 5 discusses the design of totally self-checking (TSC) finite state automata, and relates this information to the diagnosis and isolation algorithm presented in Chapter 4. Specifically, it is shown that all faults in the assumed fault set are correctly diagnosed, including link failures between processing elements.

Chapter 6 summarizes the results of this research, and discusses open questions for future efforts. In particular, it is shown that with the inclusion of triple-fault-tolerance in the local reconfiguration scheme, the addition of concurrent fault detection, and the distributed fault reporting mechanism, the hierarchical reconfiguration strategy is made more robust.

2. BACKGROUND

This chapter provides a foundation for the results presented in subsequent chapters. The basic concepts of systolic arrays are introduced, and the suitability of these arrays for implementation in VLSI and WSI technologies is argued. However, such implementations present certain difficulties, thus motivating the subsequent discussion of reconfiguration as a means to alleviate these problems. The relative merits and deficiencies of global and distributed reconfiguration control mechanisms are examined, with the focus being on a fault-tolerant computing environment. Finally, the state of the previously developed hierarchical reconfiguration scheme is presented, which incorporates both the local and global reconfiguration schemes of [Whi91] and [Bri87], respectively.

2.1 SYSTOLIC ARRAYS

Algorithms can generally be categorized as either compute-bound or I/O-bound processes. A compute-bound process is one in which the number of operations performed in the algorithm is greater than the number of distinct data objects manipulated during processing. An I/O-bound process, on the other hand, involves a relatively small number of operations performed on a relatively large number of data objects. Unfortunately, compute-bound processes can become I/O-bound when executed in a classical von Neumann architecture [Kun82]. This can occur when many of the same data objects are retrieved from memory several times during processing, so that the total number of I/O operations is on the same order as the number of processing operations. Thus, the general purpose von Neumann architecture is unsuitable for such applications.

Systolic arrays embody a class of special purpose architectures which eliminate the von Neumann memory bottleneck by using a single access of a given data object to perform parallel computations. Systolic arrays are so named by the analogy that data circulates through the array as blood does through the circulatory system.

An example systolic array which performs convolution is given in [Kun82], and reproduced as Figure 2.1. In the figure, x_i represents the i^{th} input to the system, w_i is the i^{th} coefficient of the system's unit pulse response, and y_i is the i^{th} system output. In this particular implementation, w_i remains fixed at cell i , and the output y_i propagates to cell $i+1$. Note that the system accesses each input x_i only once, as opposed to the multiple accesses required by a sequential system. The same technique can be applied to many other signal processing and matrix algorithms, including 1-D and 2-D Fast Fourier Transforms, IIR filter implementation, matrix multiplication, and matrix LU decomposition [Sny82].

It is important to realize that each cell in the array is essentially the same as any other, with the possible exception in the example of the w_i coefficients. If these coefficients are loaded into the cells at runtime, rather than hardwired, then all cells are identical. This structural regularity is well-suited to VLSI/WSI implementation [Kun82].

Systolic arrays typically arise in the context of a larger system, as is assumed here. In such systems, a host computer applies input data to the array as needed. The array in turn provides the host with the resulting output data, which may then be used in computations better suited to general purpose architectures [Kun82].

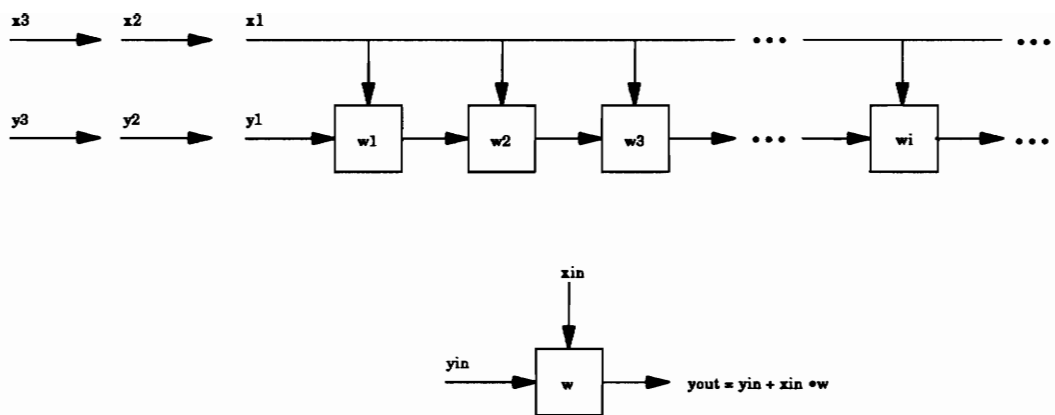


Figure 2.1 - Example systolic array for performing linear convolution.

2.2 MOTIVATION FOR RECONFIGURATION

VLSI/WSI array processors can, unfortunately, develop faults during production or at runtime for a number of reasons [GrW89]. It is therefore desirable to introduce some measure of fault-tolerance into the end-product which allows it to perform to the desired specifications, even in the presence of a predetermined number of faulty elements. Systolic arrays can also be designed to perform a variety of processing tasks, thus reducing the need for separate, special-purpose architectures [Sny82], [GrW89]. Such arrays may need to be reconfigured, either during production or at runtime, to select the desired array functionality.

2.2.1 YIELD ENHANCEMENT

Deficiencies in current manufacturing processes can result in low product yields, even for single faults [GrW89]. New techniques such as X-ray lithography may show promise as a means of allowing technicians to directly effect repairs on a faulty integrated circuit, but this technology is not sufficiently developed to employ in practical applications. Rather than discard faulty chips, it is often desirable to introduce sufficient redundancy so that a required minimum number of working array elements per chip is very likely to be available. Such goals are achieved through reconfiguration for yield enhancement, in which faulty elements are identified and disabled at production time, and are replaced by spares. Yield enhancement schemes are often referred to as static reconfiguration, since reconfiguration is performed only once at production time, and the changes are permanent. Static reconfiguration is generally not a time-critical operation, since full functionality is only needed at runtime; thus, sophisticated fault detection and

location schemes may be employed to improve the process yield. Examples of reconfiguration techniques which are suitable for yield enhancement include Fault Tolerant Testable Scan Design [KiR89], and the Kuo-Fuchs Strategy [KuF87].

2.2.1.1 FAULT TOLERANT TESTABLE SCAN DESIGN

In [KiR89], Kim and Reddy propose the augmented PE reproduced in Figure 2.2 as a means of yield enhancement. It accepts data on both horizontal and vertical data paths, and contains several components to facilitate testing and reconfiguration.

The horizontal data input register, HR, accepts data on the horizontal input path for processing. Similarly, the vertical data input register, VR, accepts data on the vertical input path. A control flip-flop, CS, configures a switch, SW, which determines the input to VR. The three possible switch configurations are shown in Figure 2.3, and are reproduced from [KiR89]. The bypass register, BR, is used to route incoming horizontal data around a faulty PE. Hence, the assumption is made that a faulty PE can still perform this buffering operation. A delay register, DR, can be used to buffer vertical input data by one clock cycle to prevent skew between the horizontal and vertical data. This feature is used when the horizontal input data comes from a bypassed PE. Multiplexer MUX2 selects the contents of either VR or DR as vertical input data to the Computational Unit, CU, as determined by the CD control flip-flop. The CU performs the necessary computation on the horizontal and vertical input data, and stores the results in the computational result register, CR. Multiplexer MUX1 selects the output data path, which is determined by the 2-bit control register CM. When $CM = 00$, the horizontal input data

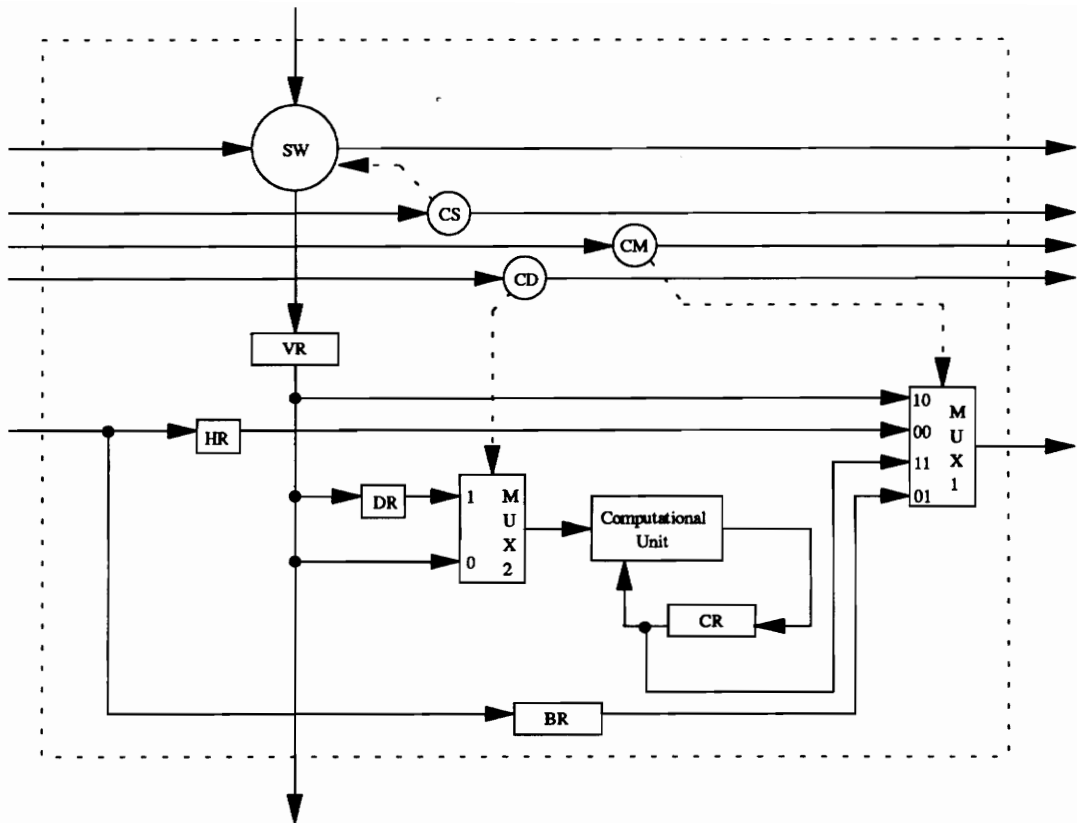


Figure 2.2 - A proposed augmented PE for yield enhancement.

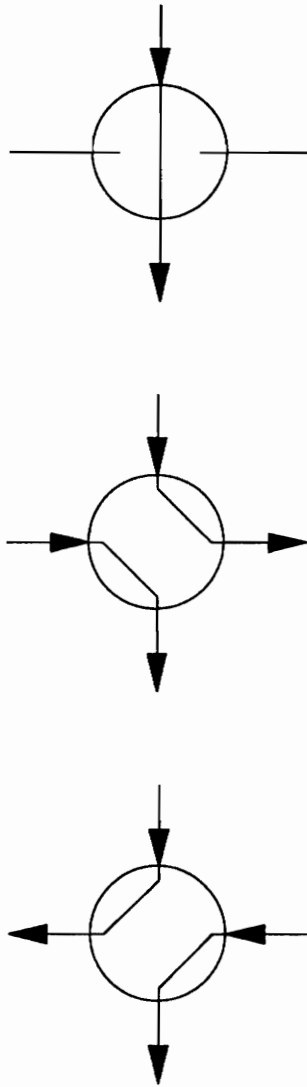


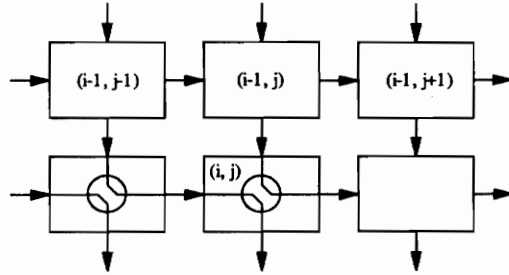
Figure 2.3 - Possible settings for augmented PE switch SW.

held in HR appears at the output. When $CM = 01$, the horizontal input data held in BR appears at the output. When $CM = 10$, the vertical input data held in VR appears at the output. Finally, when $CM = 11$, the computational result in CR appears at the output.

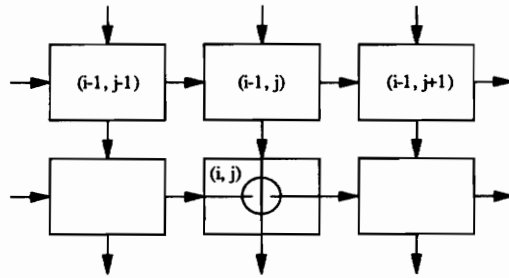
As indicated by the simplified PE interconnection patterns shown in Figure 2.4, a PE at location (i, j) can receive vertical input data through SW from the PEs at $(i-1, j-1)$, $(i-1, j)$, or $(i-1, j+1)$. This provides flexibility in selecting an alternate vertical data input path when faulty PEs are bypassed. Testing is accomplished in four steps:

1. Test data are scanned into the control flip-flops and the test responses are scanned out. By noting faults in the test responses, it is possible to determine which PEs should be bypassed.
2. More test data is scanned in along the horizontal data path, and the test responses are scanned out. Since the control flip-flops have already been tested, any faults are due to the horizontal data path. Faulty PEs can then be located and bypassed.
3. Test data is scanned in on the vertical path, and MUX1 is used to scan out responses along the fault-free horizontal path found in step 2. Any faulty PEs in the vertical path are detected, and so can be bypassed.
4. Test data is scanned in on the fault-free horizontal and vertical paths found in step 3, and passed into the computational unit. Results in the CR are scanned out on the horizontal path. Any PEs with faulty computational results can be detected and bypassed.

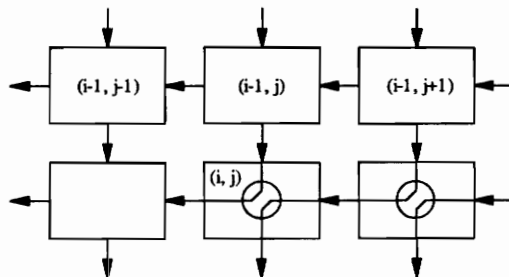
The testing and reconfiguration process is managed by an external global control mechanism, which injects test data, evaluates test responses, and sets control flip-flops. Figure 2.5, reproduced from an example in [Kun82], shows a reconfigured systolic array which multiplies two 3-by-3 matrices. Zero values in the data stream insure that



a) Cell (i,j) receives vertical input from cell $(i-1, j-1)$.



b) Cell (i,j) receives vertical input from cell $(i-1, j)$.



c) Cell (i,j) receives vertical input from cell $(i-1, j+1)$.

Figure 2.4 - Simplified diagram of vertical input paths for augmented PE.

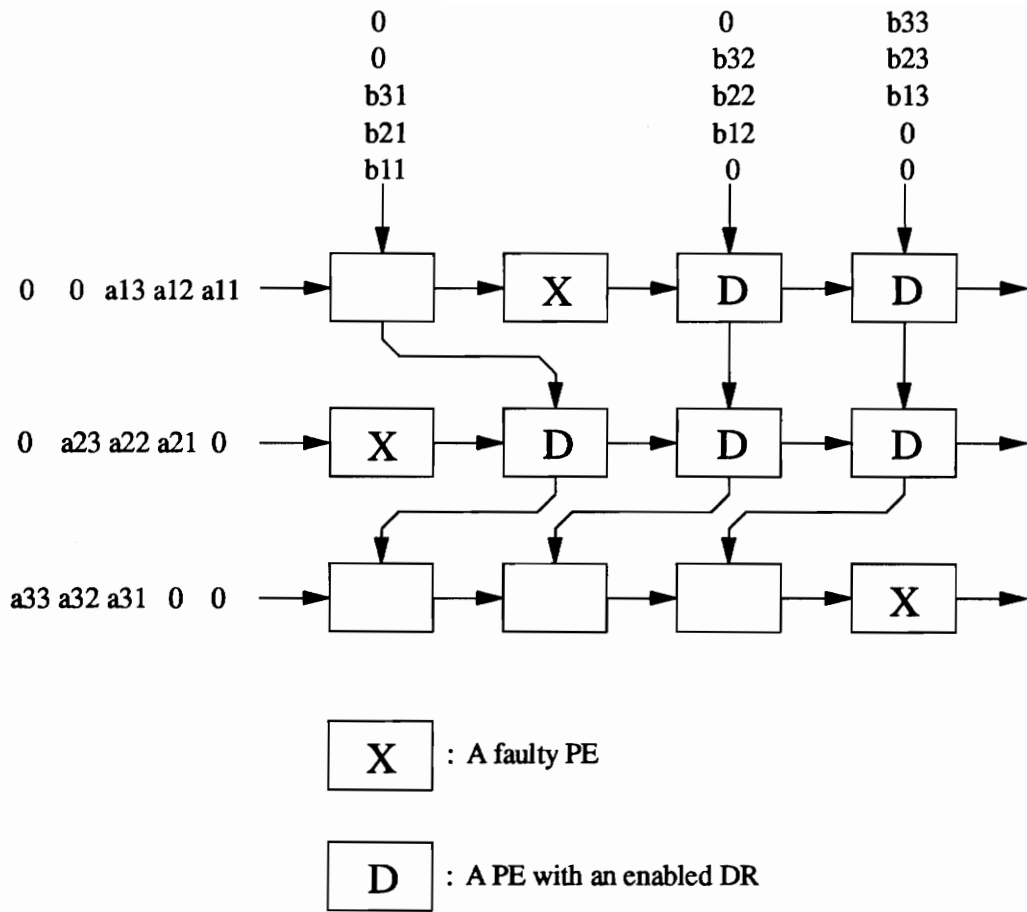


Figure 2.5 - Successfully reconfigured systolic matrix multiplier array.

horizontal and vertical data items are correctly synchronized with one another during the computation. Faulty PEs are indicated by the symbol X, and fault-free PEs using DR to provide the proper delay on the horizontal data path are denoted by the symbol D. The scheme is processor-switched, meaning that individual processors are switched into the logical array to replace faulty PEs. The technique employs local redundancy, since a PE can only substitute for its left or right neighbor, rather than for an arbitrary faulty PE [ChF90]. Since this technique tolerates at most one fault per row, it may be unsuitable for yield enhancement when clustered faults are present.

2.2.1.2 THE KUO-FUCHS REPAIR-MOST STRATEGY

Set switching, a strategy in which banks of PEs are switched in to replace those containing faulty PEs, may be used to accommodate the clustered fault model [ChF90]. The Kuo-Fuchs strategy substitutes a working row or column of array elements for a row or column containing one or more faulty PEs. In this scheme, the system is modeled as a bipartite graph, as shown in Figure 2.6 [Joh89], [ChF90]. One column of nodes in the graph represents rows containing faulty cells, and the other represents columns containing faulty cells. An edge in the graph represents a faulty PE at location (i, j) . Thus the edge is incident to the node for row i and the node for column j . A faulty PE is said to be *covered* if a spare PE is available to replace it. The basic repair-most strategy is to replace rows and columns with the most faults first, thus assuming that the remaining faults may be easily covered. The algorithm for replacing faulty rows and columns is as follows:

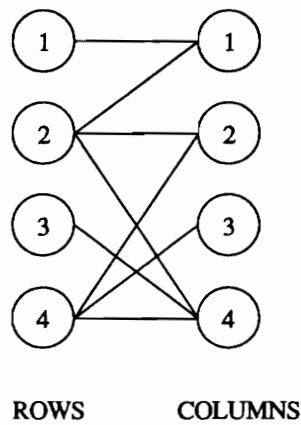
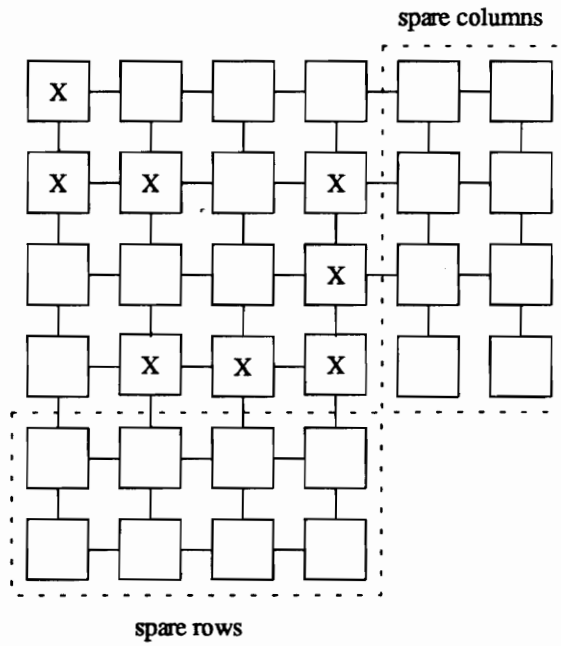


Figure 2.6 - Repair-most strategy example with 2 spare rows and 2 spare columns.

1. Select a node with the maximum number of incident edges.
2. Replace the row/column of PEs represented by the node.
3. Remove the node and its incident edges from the graph.

These three steps are repeated until all faults are covered by spare rows or spare columns, if possible. If two or more nodes have the same number of incident edges, some arbitrary means may be used to select one of those nodes. Note, however, that the order in which nodes are deleted can influence the outcome of the repair-most strategy. Figure 2.7 represents a successful replacement strategy of the bipartite graph in Figure 2.6. If not all of the faulty PEs are covered after all spare rows and columns are used, the algorithm fails. An example of this failure is given in Figure 2.8, using the same initial configuration of Figure 2.6 with a different ordering of row and column replacements than is used in Figure 2.7. After both spare rows and both spare columns are used, the faulty PE located at (4, 3) is still not covered. In the general case, optimal solution of the Kuo-Fuchs graph-theoretical representation is an NP-complete problem [KuF87], similar to the well-known "traveling salesman" problem. However, through the use of heuristics, such as a branch-and-bound approach, it is often possible to produce optimal or near-optimal results in a reasonable time.

Since the algorithm requires information concerning the locations of all faults in the array, it is well-suited to a global control scheme. The global control mechanism can perform the algorithm, and make the necessary replacements. Implementations can be locally redundant, wherein spare sets can replace only certain faulty sets. This reduces the amount of switching hardware as compared to a global redundancy scheme, but can result in the underutilization of spares, due to the clustered nature of the faults. However, a

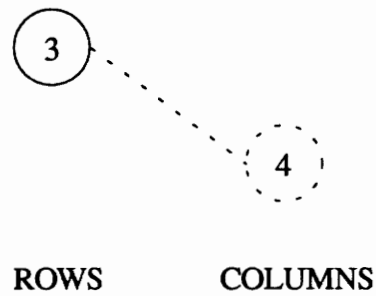
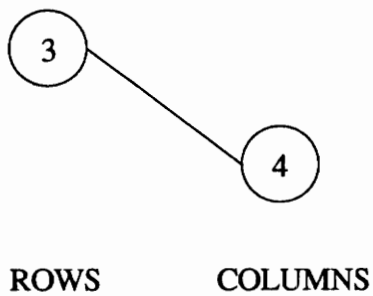
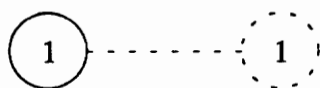
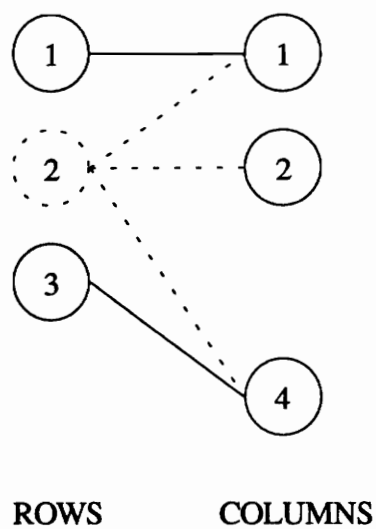
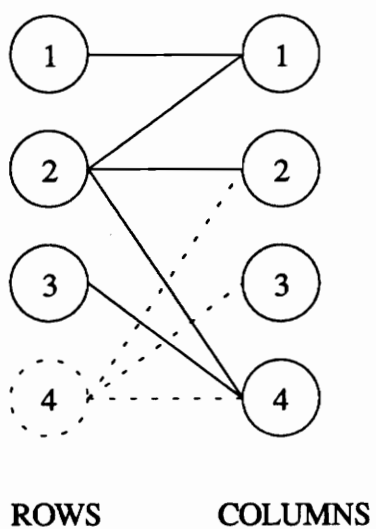


Figure 2.7 - Successful reconfiguration of Figure 2.6 using repair-most strategy.

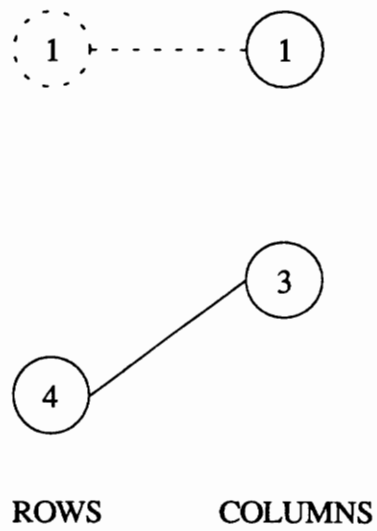
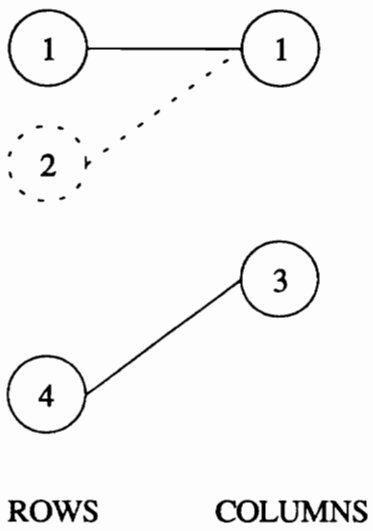
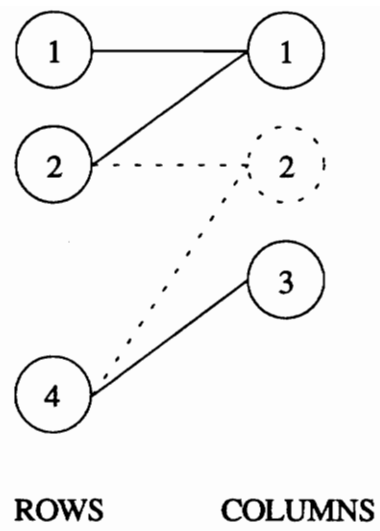
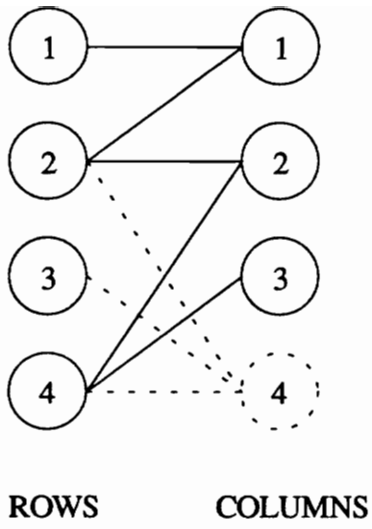


Figure 2.8 - Unsuccessful reconfiguration of Figure 2.6 using repair-most strategy

global redundancy scheme would allow spare sets to replace any faulty sets, thus providing better utilization of spares [ChF90].

2.2.2 RUNTIME FAULT TOLERANCE

Dynamic reconfiguration for reliability enhancement considers faults encountered in a runtime environment, as opposed to static reconfiguration for yield enhancement. High reliability systems must not fail on the basis of a single runtime fault [GrW89]. Though these faults can be either transient or permanent in nature, most runtime faults tend to be transient and randomly distributed within the array [Joh89]. An effective reconfiguration scheme must be capable of dealing with both transient and permanent faults in an efficient manner, since long delays are intolerable.

2.2.2.1 THE DIOGENES APPROACH

Rosenberg introduced an approach for the design of fault tolerant arrays, which linearizes the array via a set of communications links [Ros83]. These links traverse the entire structure, so that any given PE can be connected to any other PE. The cell interconnection paths are assumed to be fault-free, which the author attempts to justify by citing the relative simplicity of the links as compared to the PEs. In his paper, Rosenberg introduces hardware stacks and queues, reproduced in Figures 2.9 and 2.10, respectively.

The stack structure of Figure 2.9 allows a PE to PUSH a request for a connection to a following PE, and also allows it to connect to a preceding PE by asserting the POP line. Logically, PEs are not linearly connected, but can instead skip physically adjacent

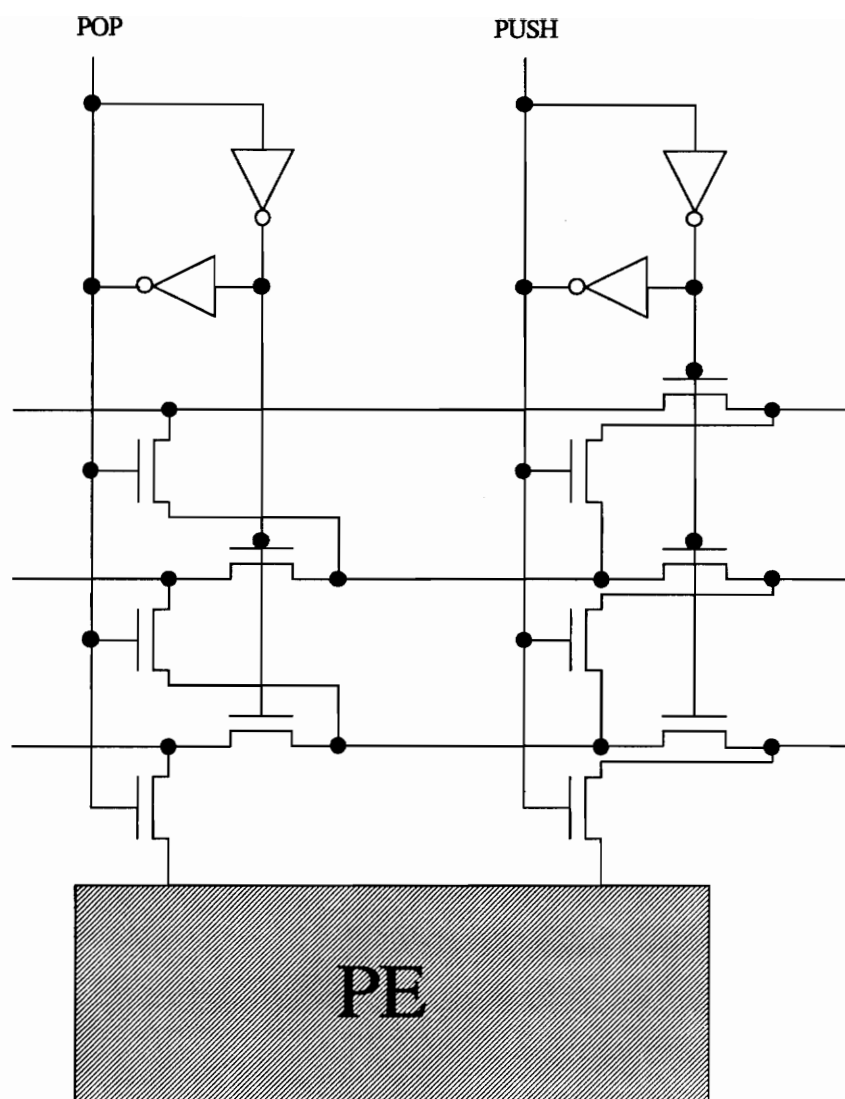


Figure 2.9 - Hardware stack structure for the Diogenes approach.

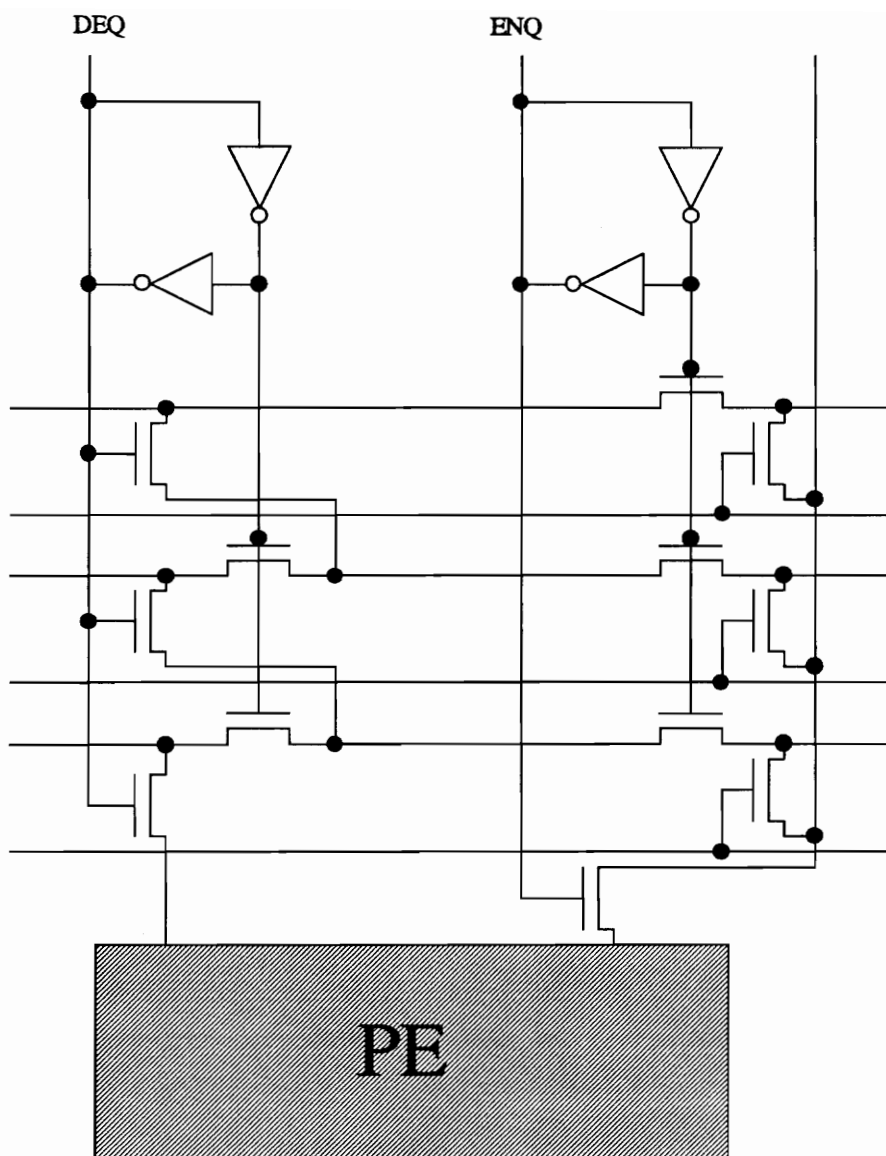


Figure 2.10 - Hardware queue structure for Diogenes approach.

PEs to connect to logically adjacent PEs. By augmenting this structure, it is possible to create other interconnection schemes, such as binary trees [Ros83].

The queue structure of Figure 2.10 allows a PE to ENQ (enqueue) a request for a connection to a following PE, and also allows it to connect to a preceding PE by asserting the DEQ (dequeue) line. If a one-element queue is used, a linear array can be easily constructed. However, in a larger queue, PEs can again skip physically adjacent PEs to connect to logically adjacent PEs. This structure can be combined with the stack structure to create pyramid and rectangular mesh interconnections [Ros83].

Modifications to the Diogenes approach proposed in [BeB92] describe a strategy in which spare rows and columns replace rows and columns with insufficient numbers of working PEs. This results in a lower interconnection overhead, since only the PEs in a given row or column must be linearized, rather than linearizing the entire array. In addition, the modified approach does not assume the existence of fault-free communications links. Unlike the originally proposed design, however, the modified Diogenes approach can not achieve 100% utilization of spares.

A major drawback of both approaches is in the potential for long cell interconnections in the presence of a large number of faulty consecutive nodes. The propagation delays thus incurred can severely limit the array performance. Both approaches depend on a global control mechanism to provide test data and interpret test results in order to locate faulty PEs. This strategy is best suited to small arrays consisting of complex PEs [ChF90].

2.2.2.2 INTERSTITIAL REDUNDANCY

This method uses a processor-switched local redundancy scheme. One possible configuration from [Sin88] is reproduced in Figure 2.11, in which a regularly connected mesh is augmented with spare PEs. As shown, each spare node can replace one of a fixed set of other nodes. As opposed to the Diogenes approach, interstitial redundancy has the advantage of reduced cell interconnection cost in terms of area, reliability, and data propagation delays. Such a scheme could be employed using a distributed reconfiguration algorithm, in which the cells in a given set are self-checking or neighbor-checking. Spares can then be activated to replace the faulty cells. A global reconfiguration mechanism can also be used to determine the locations of faulty cells, and to activate spares accordingly. Interstitial redundancy can be used with many other interconnection patterns. Fault coverage can be improved by adding more interstitial redundancy, at the expense of greater layout area overhead and underutilization of spares.

2.2.2.3 FAULT-STEALING STRATEGIES

These methods, described in [SaS86], allow a faulty cell to "steal" a neighbor to perform its duties. However, this neighbor must in turn "steal" from another neighbor, and so on. The process continues until a spare is found to fill the gap initially created by the faulty PE. This approach is processor-switched and locally redundant. It is well-suited to a distributed environment.

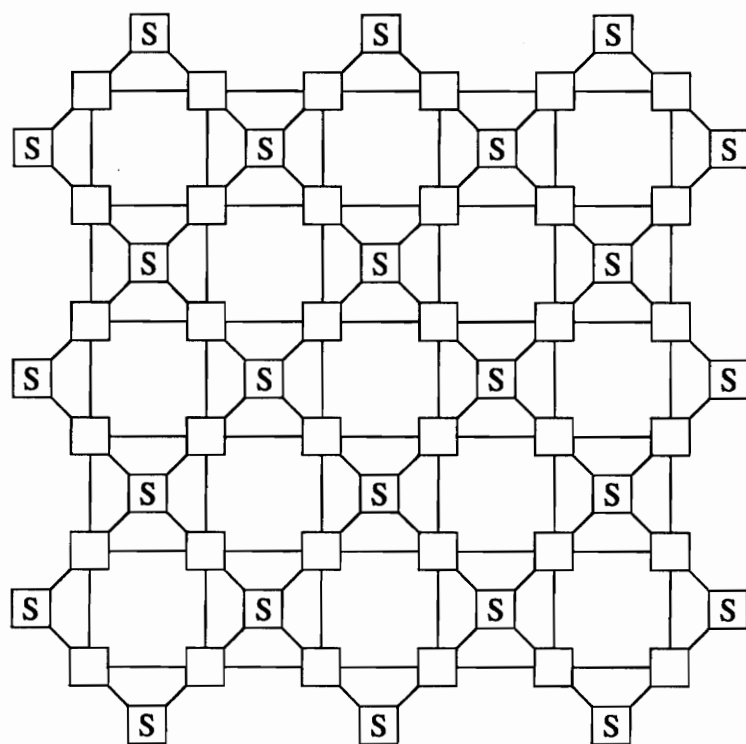


Figure 2.11 - Example of interstitial redundancy.

Direct reconfiguration (DR) forms the basis for the fault-stealing strategies. An example of the DR approach from [ChF90] is reproduced as Figure 2.12. The physical coordinates of PEs are given in parentheses, and their logical coordinates are given in square brackets. In Figure 2.12, for example, the cell with physical coordinates (2, 3) has logical coordinates [1, 2]. The logical coordinate [0, 0] indicates that a PE is not used in the final configuration. In DR, each column is scanned from the bottom up. The leftmost fault in each row is flagged as a "vertical fault," and the remaining faults in the row are "horizontal faults." Vertical faults steal downward, and horizontal faults steal to the right. Note that it is possible for a fault to steal both downward and to the right, as with element [1, 2] in the example. The DR algorithm fails if a given row has two or more horizontal faults. The interconnection costs are modest, the algorithm is efficient, and high utilization of spares is possible. However, DR is unable to reconfigure in the presence of multiple horizontal faults per row.

Fixed-stealing increases array survivability by allowing the rightmost horizontal fault in a row to steal from the same row (as in DR), but the remaining horizontal faults steal vertically up from an adjacent row. To tolerate N faults in a row, it is necessary to allow stealing from up to N fault-free rows previous to the row containing horizontal faults. Variable stealing works in a similar fashion, except it selects as the horizontal fault the cell which cannot steal from a previous row [SaS86].

Complex fault stealing (CFS) allows more flexible vertical stealing, as shown in an example from [ChF90] given here as Figure 2.13. Note the additional fault at (2, 3) can be accommodated by CFS, but would fail under direct reconfiguration, since there are two horizontal faults in the first row. The vertical connection paths are omitted to improve the

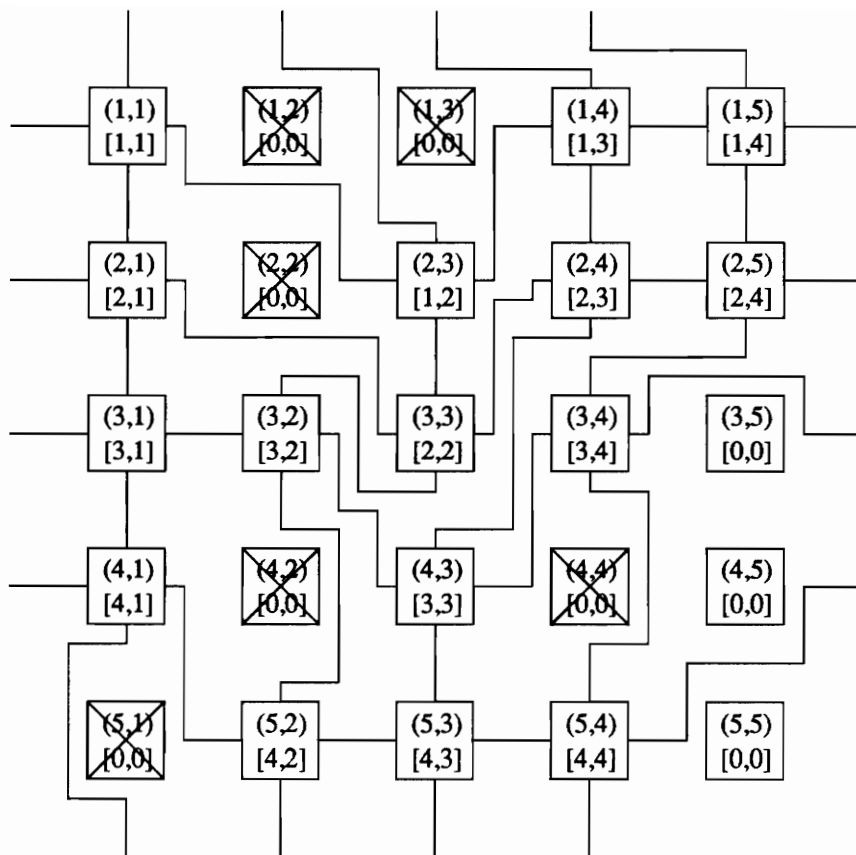


Figure 2.12 - Example of direct reconfiguration (DR).

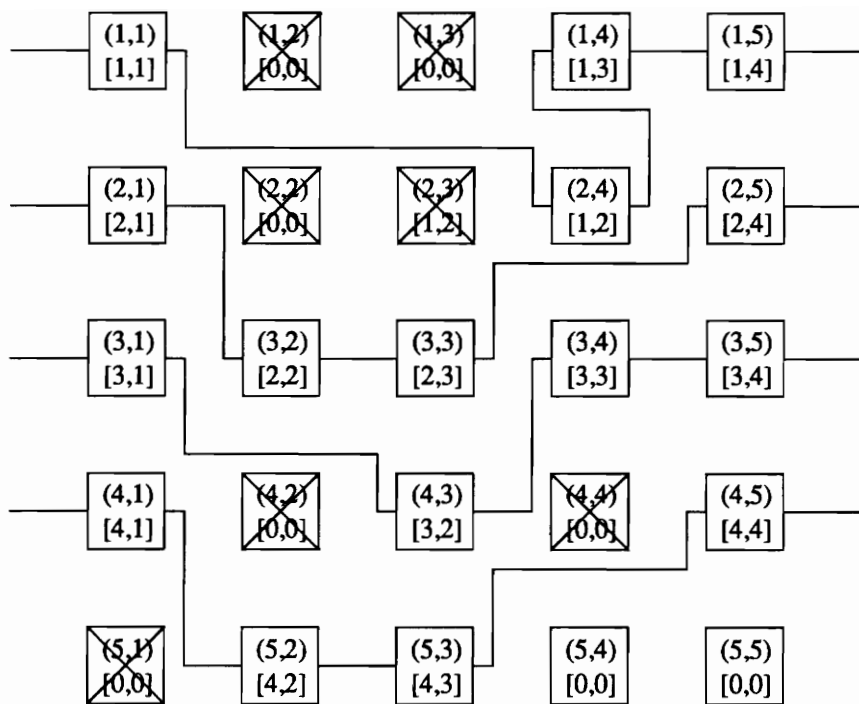


Figure 2.13 - Example of complex fault stealing (CFS).

readability of the figure. The pseudocode algorithm for CFS is given in Figure 2.14 [ChF90]. CFS offers a higher probability of reconfiguration at the expense of increased cell interconnection complexity, and a slightly more complex algorithm than other fault-stealing methods. The cells are assumed to be self-testing, which requires additional cell overhead. However, the authors make no assumptions regarding the types of faults which occur. Rather, it is up to the designer of a fault-stealing system to determine an appropriate fault model for a particular environment. Reconfiguration can be accomplished quickly, which is important in the case of runtime faults.

2.2.3 ARRAY POLYMORPHISM

Reconfiguration does not necessarily require that a fault exists in a processor array. Processor array architectures are often "special purpose," in that the functionality of the array is specifically designed for a particular application [GrW89]. This implies that different applications require different array architectures. This can be very expensive for the manufacturer, as well as for the users of such architectures. Reconfiguration techniques for array polymorphism provide a means of altering the array functionality in terms of the cell interconnections and the processing functions to be performed, as in [Sny82], [GrW89]. This allows a single processor array to perform various computational tasks.

2.2.3.1 THE CONFIGURABLE, HIGHLY PARALLEL (CHIP) COMPUTER

The CHiP computer, introduced by Lawrence Snyder in [Sny82], takes advantage of the array polymorphism concept. It consists of a grid composed of PEs and switching

```

horizontal_steal = FALSE;
for (every faulty or unavailable cell (i, j) in a row)
{
  if (not_rightmost(i, j) || (horizontal_steal == TRUE))
  {
    if (fault_free(i+1, j))
      steal(i+1, j);
    else if (fault_free(i+1, j+1))
      steal(i+1, j+1);
    else if (not_rightmost(i, j))
    {
      steal(i, j+1);
      horizontal_steal = TRUE;
    }
  }
  else
  {
    steal(i, j+1);
    horizontal_steal = TRUE;
  }
}

```

Figure 2.14 - Pseudocode algorithm for complex fault stealing.

elements, whose functions are determined by a global control mechanism. This controller is responsible for configuring the functionality of the PEs and the switch interconnection patterns. By sending commands to the controller, the grid can be configured according to the needs of a particular application.

Examples of the CHiP architecture are given in Figures 2.15 and 2.16, which are taken from [Sny82]. Figure 2.15 shows a typical rectangular mesh pattern, where the square elements represent PEs, and the circular elements represent switches. The same grid can be configured as a binary tree, as shown in Figure 2.16, by simply changing the interconnection pattern. This reconfiguration property can also be used as a measure of fault tolerance. When faults are detected by the global controller, the switch lattice can be configured around the faulty area. Thus, the CHiP architecture is essentially processor-switched and locally redundant. By adding set-switched, multiple, independent CHiP grid structures on a die or wafer, the flexibility of reconfiguration increases. A single grid can be selected for application suitability using set switching, after which the desired interconnection pattern and fault bypassing can be accomplished through processor switching. The switching functions involved in this approach can be complex, due to the requirements of various interconnection schemes [ChF90]. This leads to reduced switch reliability and increased layout area.

2.2.3.2 HIERARCHICAL RECONFIGURATION

Although the previous examples describe several techniques to address one or perhaps two of the three types of reconfiguration, none offers a comprehensive approach to deal with reconfiguration for yield enhancement, runtime fault tolerance, and array

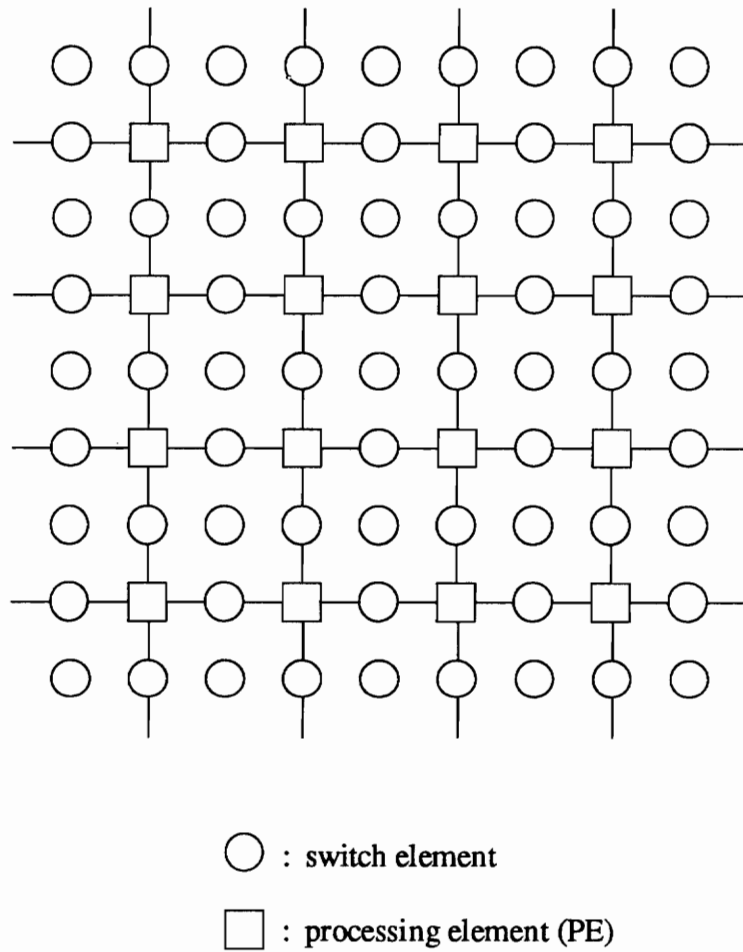


Figure 2.15 - CHiP architecture configured as a mesh array.

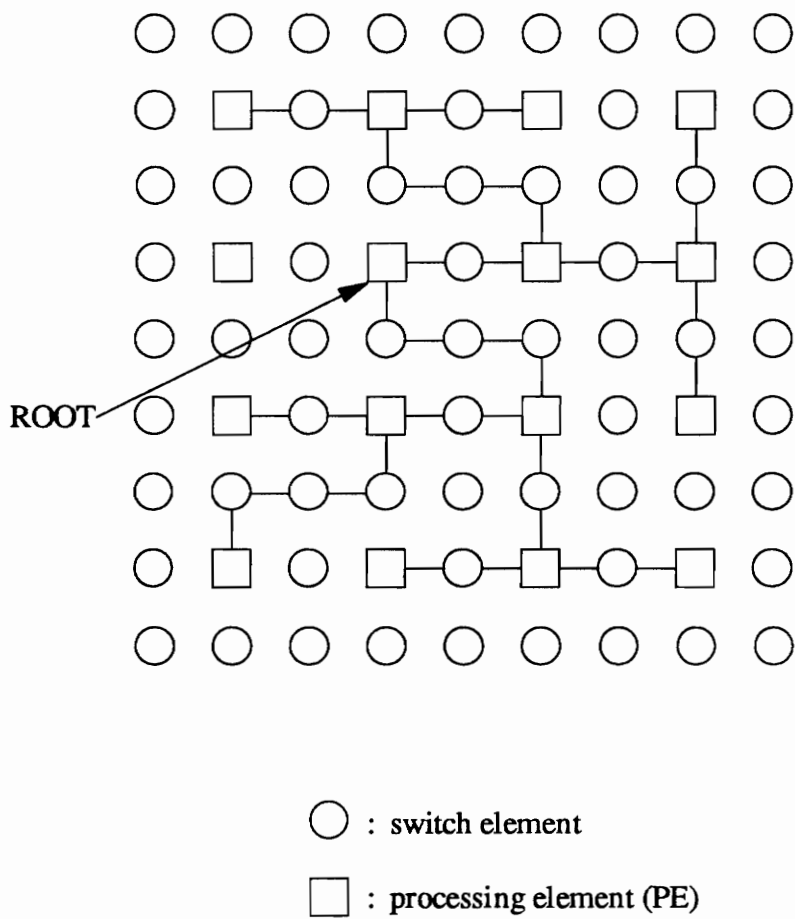


Figure 2.16 - CHiP architecture of Figure 2.15 configured as a binary tree.

polymorphism. Perhaps a hierarchical reconfiguration strategy can provide the flexibility needed to respond to all three reconfiguration scenarios. Such a hierarchical system, described in [Mar80], [GoG84], [Kum84], [Bri87], and [Whi91], addresses all three issues. The results presented in this thesis are a continuation of that effort.

The system is hierarchical in the sense that it supports a two-tiered reconfiguration scheme. A local reconfiguration scheme allocates spare rows and columns of processing elements, along with the requisite number of elements needed to support the logical array. The spare PEs in the additional rows and columns are then used to bypass faulty PEs in the logical array. Should the local reconfiguration mechanism fail to tolerate a given set of faults, a global redundancy scheme then allows the logical array to remap itself into a fault-free portion of the physical array. This approach provides the array with substantial flexibility in tolerating faults. The hierarchical reconfiguration scheme is presented in the following two sections. Since the global scheme is the original approach used for all three reconfiguration scenarios, it is presented first.

2.3 GLOBAL RECONFIGURATION

A processor array can be visualized in terms of a "computational plane" and a "control plane," as depicted in Figure 2.17 [GoG84]. Without loss of generality, the control and computational planes may be physically distinct, or may only be separated conceptually. The physical separation of the two planes could be implemented in a system such as the Hughes 3-D computer in [LiE89]. The Hughes system architecture consists of a set of stacked wafers, each of which contains a different set of functionally compatible modules. For example, customized control and computational modules could each be

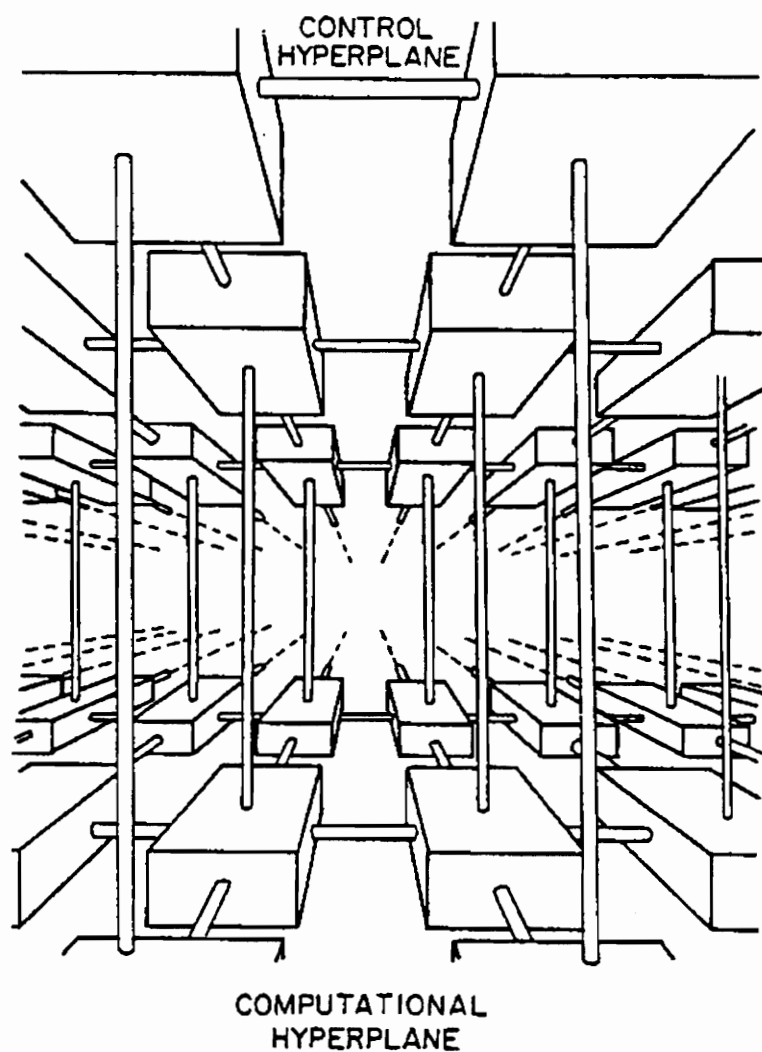


Figure 2.17 - An array of PEs divided into control and computational planes.

designed on separate wafers for specific applications, and then integrated with standardized memory and I/O wafers. Physical separation also simplifies the problem of routing cell interconnections, since the control and computational routing would otherwise need to be coplanar. The computational plane may consist of processing elements which employ internal switching, or may contain both PEs and external switching elements if internal switching is not used. Since computational elements are allowed to be arbitrarily complex in design and functionality, they will not be considered here.

Control plane cells are responsible for configuring the computational elements to which they are linked, so that various computational functions and interconnection schemes can be realized. The control plane is an array of identical, regularly connected cellular finite state machines (FSMs). The FSMs are assumed to be Moore machines, whose outputs are dependent only on their present states. A cell's *neighborhood* is defined as the set of cells to which it is logically connected, and includes the cell itself. The number of cells in the neighborhood is called the *neighborhood scope*. Two-dimensional Moore or von Neumann neighborhoods are typically used. The method presented here employs a two-dimensional von Neumann neighborhood, in which each cell in the array is connected to its North, East, South, and West neighbors. The Moore neighborhood augments this scheme by adding connections to the Northeast, Southeast, Southwest, and Northwest neighbors. Either neighborhood may be used, as long as each cell is included in its own neighborhood in order to ensure the stability of the array [ThW77]. Note that the scope of the von Neumann neighborhood is 5, and the scope of the Moore neighborhood is 9.

2.3.1 MARTIN'S METHOD

Previous research conducted at Virginia Tech focuses on distributed reconfiguration of cellular arrays. In [Mar80], Martin presents a method of producing a self-diagnosing control mechanism which can relocate itself to a fault-free portion of a processor array. However, little consideration is given to the problem of initializing the array to the desired functional pattern. The neighborhood scope used in Martin's method is also prohibitively large, and grows linearly with the number of faults to be tolerated. Thus, a substantial number of cell interconnections is required. The concept is developed for linear arrays, but the extensions of the method to multidimensional arrays depend heavily on the properties developed for the linear case, and so are of limited usefulness.

2.3.2 KUMAR'S METHOD

Kumar developed an improved distributed control method which was more amenable to multidimensional arrays in [Kum84]. This method also developed a strategy for initializing the control mechanism, and provided better fault isolation and reconfiguration capabilities than Martin's method. As it employed a fixed neighborhood independent of the number of faults to be tolerated, the number of cell interconnections was substantially reduced in comparison to Martin's method.

2.3.3 BRIGHTON'S METHOD

Brighton's method, presented in [Bri87], is an outgrowth of the cellular automata research conducted by Martin and Kumar. The size of the local memory required to store

the next-state lookup table for the FSMs is reduced compared to the methods developed by both Martin and Kumar. Like Kumar's method, the neighborhood scope is independent of the desired level of fault tolerance, and employs fewer processor interconnections than Martin's method. The fault isolation and reconfiguration strategies described in [Kum84] are incorporated into Brighton's method with little or no alteration. The basic differences between Brighton's and Kumar's methods are primarily concerned with initializing the control mechanism. The significance of the improvements resulting from Brighton's method are discussed in the complexity analysis of Chapter 3.

2.3.3.1 DEFINITIONS OF TERMS USED

The states of the control cells in the array constitute a control pattern, and the process of establishing this control pattern in the array is called *pattern growth*. When faults in the array are detected, the distributed control mechanism can recreate the correct pattern in a fault-free area of the array, so that a proper operating state is restored.

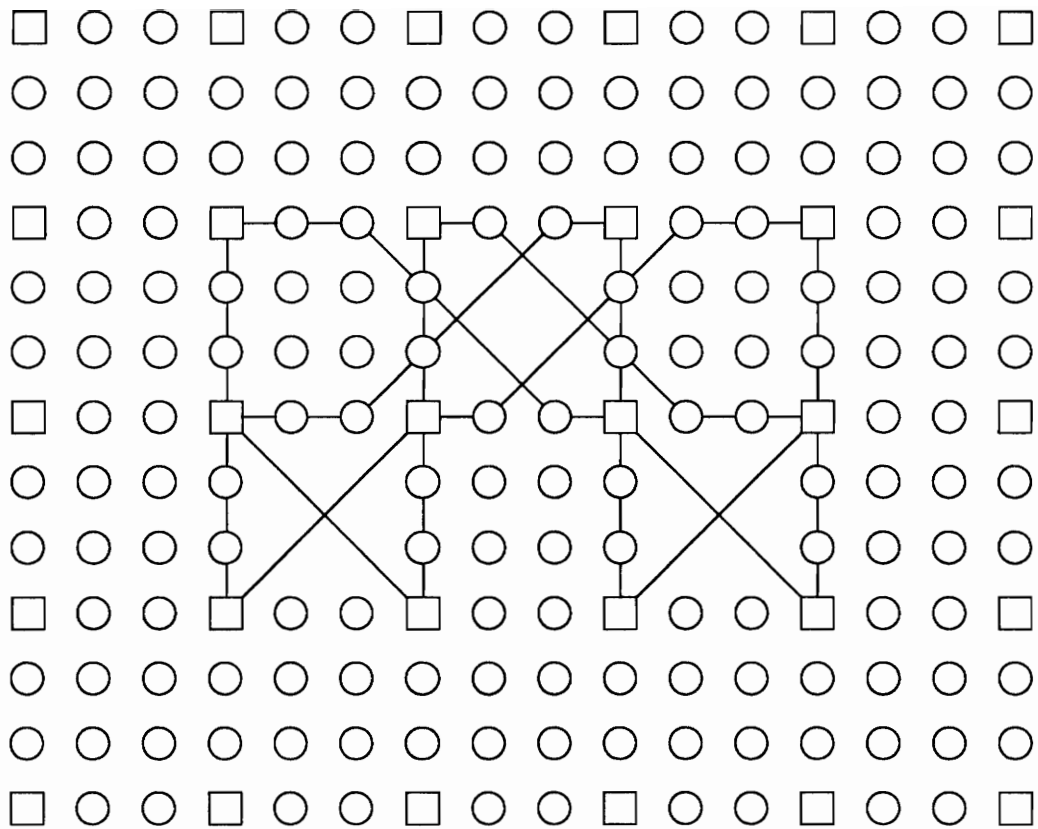
In order to discuss the initialization procedure, it is important to distinguish between the state of a single control cell, the set of states for all cells in the control plane at a given time, and the desired array functionality. The *local state* (LS) of a single control cell describes the cell's interaction with other control cells, as well as with its assigned computational cell. The set of states for all cells in the control plane is referred to as the *control pattern*. A "seed" state is externally injected into the array, through which it migrates until a sufficiently large fault-free area in which to "grow" the desired control pattern is found. This is determined in essentially the same manner as in [Kum84]. Different seed states allow different patterns to be grown according to the chosen array

function (e.g., Fast-Fourier transform, band-matrix multiplication, etc.). These functions are called *global states* (GS). At each discrete time step, the control pattern grows concentrically from the seed cell until the desired pattern size is reached. At the following time step, called "*bloomtime*," each cell in the control pattern uses the GS value and its position in the final pattern as indices into a lookup table stored in local memory. The present state (PS) of a cell is defined by the combination of its LS and GS values. Pattern growth, as described above, is the process of producing the final control pattern from the seed cell corresponding to the desired GS.

2.3.3.2 PATTERN GROWTH EXAMPLE: THE BANYAN NETWORK

The Banyan network shown in Figure 2.18 is used to illustrate the pattern growth mechanism. The square symbols represent processing elements, and the circular symbols denote switches. Although the control cells use a von Neumann neighborhood, the PEs and switches in this example are connected in a Moore neighborhood. Figure 2.19 shows the connection patterns to be assigned to the switches, and the value of LS for the corresponding control elements. All control cells associated with PEs are assigned the value of $LS = 15$ (not shown), since each PE performs the same task. However, this need not be the case, in general.

Each control cell contains the following registers to support pattern growth: LSR, GSR, XR, YR, and TR. Also, a pattern growth flag (PGF) and an OK_PG flag are used. The LSR, or local state register, simply holds the present LS for the cell. Similarly, the global state register (GSR) contains the desired GS. For the example, the GSR value indicates that the Banyan network function is to be used. XR and YR denote the relative



□ : processing element (PE)

○ : switch element

Figure 2.18 - Banyan network used in pattern growth example.

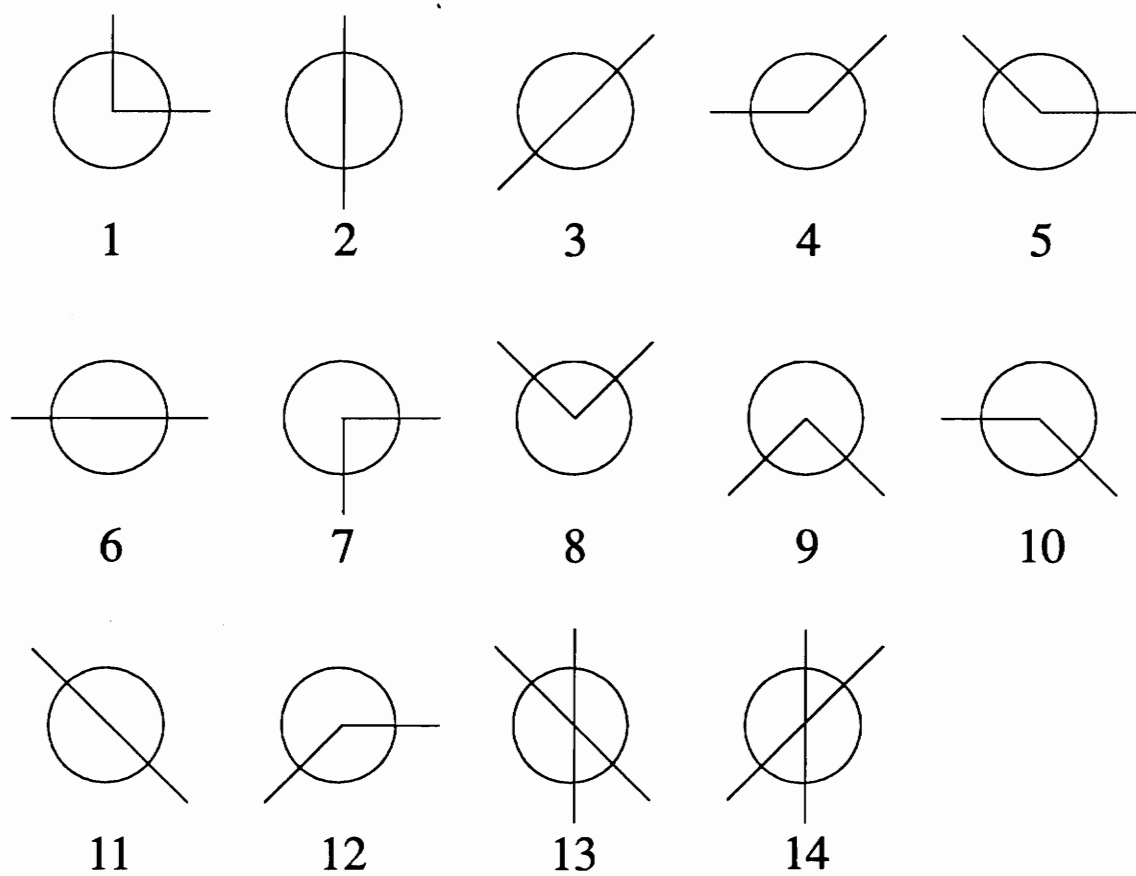


Figure 2.19 - Control cell state assignments for the Banyan Network.

x- and y- coordinates, respectively, of the cell with respect to the final global control pattern's lower left corner, which is given coordinates (1, 1). The time register (TR) is a local counter used by the cell to determine when bloomtime has arrived, so that the final pattern can be constructed. The PGF, when asserted, indicates that a given cell is participating in pattern growth. The OK_PG flag, when asserted, indicates that the seed is in an appropriate area in which to grow the pattern.

Figure 2.20 shows the contents of the XR, YR, and LSR registers for the first three time steps of pattern growth for the Banyan network. From the seed's GSR value, a cell can determine the XR and YR values for the seed cell. These values are selected to be near the center of the final pattern in order to minimize the time required for pattern growth. As shown in the figure for time step $t = 1$, the seed cell has a LS during pattern growth denoted by G_c , and has relative x-y coordinates (5, 4). The maximum pattern dimensions and the bloomtime are also a function of the GSR value. At each time step prior to bloomtime, cells in the von Neumann neighborhood of a given cell assume one of the following states: G_x , G_y , G , or the quiescent state, denoted by 0. Cells on the same row or column as the seed cell assume the G_x or G_y states, respectively. This is shown in Figure 2.20 for $t = 2$ and $t = 3$. All other cells assume the G state, as shown in the figure for time step $t = 3$. The 0-state is used to denote cells which are not currently participating in pattern growth, as well as those which lie outside the final pattern. All non-faulty cells in the array are initialized to the 0-state prior to seeding. Although not explicitly shown in the figure, all cells which are not in the G_c , G_x , G_y , or G states are actually in the 0-state.

<u>Time</u>	<u>XR, YR</u>	<u>LSR</u>
1	5,4	Gc
2	5,5	Gy
	4,4 5,4 6,4	Gx Gc Gx
	5,3	Gy
3	5,6	Gy
	4,5 5,5 6,5	G Gy G
	3,4 4,4 5,4 6,4 7,4	Gx Gx Gc Gx Gx
	4,3 5,3 6,3	G Gy G
	5,2	Gy

Figure 2.20 - XR, YR, and LSR contents during pattern growth for $t < 4$

As the pattern grows, cells pass the information in their GSR, XR, and YR registers to their neighbors. Cells pass XR values to their east and west neighbors, and YR values to their north and south neighbors. Thus, a cell determines its own position from the XR value passed to it by an east or west neighbor, and the YR value passed by a north or south neighbor. Figure 2.20 shows that for time steps $t = 2$ and $t = 3$, cells in the Gy state are in the same column as the seed cell. Although these cells do not explicitly receive an XR value, it is the same as the XR value of the seed cell, and can be locally determined from the value of the GSR. Similarly, cells in the Gx state are in the same row as the seed cell and can therefore determine the missing YR value. Brighton demonstrates in [Bri87] that cells can distinguish this case from the case where a faulty neighbor is not sending the coordinate. Since each time step during pattern growth brings the array closer to bloomtime, TR values are initialized according to the cell's Manhattan distance from the seed cell; thus, all cells reach bloomtime simultaneously.

The control cell register values for time step $t = 9$ are given in Figure 2.21. This is one step prior to bloomtime for the Banyan network. The size and shape of the final pattern are defined, and the array is prepared for the final mapping. Figure 2.22 shows the final control pattern for the Banyan network at bloomtime and beyond. Cells which lie within the pattern use the values in the GSR, XR, and YR to look up their final states in the state-transformation lookup table contained in local memory.

XR, YR									
1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7	10,7
1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6	10,6
1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	10,5
1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4	10,4
1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	10,3
1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	10,2
1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1	10,1

LSR									
G	G	G	G	Gy	G	G	G	G	G
G	G	G	G	Gy	G	G	G	G	G
G	G	G	G	Gy	G	G	G	G	G
Gx	Gx	Gx	Gx	Gc	Gx	Gx	Gx	Gx	Gx
G	G	G	G	Gy	G	G	G	G	G
G	G	G	G	Gy	G	G	G	G	G
G	G	G	G	Gy	G	G	G	G	G

Figure 2.21 - XR, YR, and LSR contents at t = 9.

0	0	0	0	0	0	0	0	0	0	0	0
0	15	6	10	15	10	12	15	12	6	15	0
0	2	0	0	13	3	11	14	0	0	2	0
0	2	0	0	14	11	3	13	0	0	2	0
0	15	6	4	15	4	5	15	5	6	15	0
0	2	11	3	2	0	0	2	11	3	2	0
0	2	3	11	2	0	0	2	3	11	2	0
0	15	0	0	15	0	0	15	0	0	15	0
0	0	0	0	0	0	0	0	0	0	0	0

Figure 2.22 - Final control pattern for the Banyan network example, $t > 9$.

2.3.3.3 PATTERN GROWTH RULES

The following three rules summarize the pattern growth process:

Rule 1: The seed corresponding to the desired GS migrates until it finds a fault-free area of adequate size in which to grow the pattern. The OK_PG flag is asserted, and the seed cell initiates pattern growth by asserting the PGF. Next, the cell sets $XR = \text{cenx}(\text{GSR})$, and $YR = \text{ceny}(\text{GSR})$, where cenx and ceny are functions which return the coordinates of the seed cell relative to the lower left corner of the pattern, as specified by the GSR. The cell then sets $TR = 0$ and $LSR = Gc$.

Rule 2: A cell which is not currently participating in the pattern growth process ($\text{PGF} = \text{false}$), and has $LSR = 0$ should check the following:

1. the cell has no faulty neighbors,
2. at least one neighbor has $LSR = Gc$, Gx , Gy , or G ,
3. all neighbors have valid values for GSR, and
4. the cell's position lies within the pattern.

Cells failing conditions 1 or 3 are mapped to a special quarantine state to isolate the faulty cell(s). Cells failing conditions 2 or 4 remain in the 0-state, since they are not part of the pattern.

If all four conditions are met, the cell asserts the PGF, and loads the XR and YR registers from information passed by its neighbors. The cell's time register is initialized to its Manhattan distance from the seed cell:

$$TR = \text{abs}(XR - \text{cenx}(\text{GSR})) + \text{abs}(YR - \text{ceny}(\text{GSR})).$$

The cell then assigns a value to the LSR according to its position:

```

if (XR == cenx(GSR))
    LSR = Gx;
else if (YR == ceny(GSR))
    LSR = Gy;
else
    LSR = G;

```

The Gc, Gx, Gy, and G local states are referred to as *pattern growth states*.

Cells remain in these states until bloomtime.

Rule 3: At each time step after the PGF is asserted, the cell increments the TR until bloomtime is reached. At that time, the cell uses its GSR, XR, and YR registers as indices into the final state lookup table, and assigns the returned value to the LSR. The PGF is then cleared, prohibiting further pattern growth.

Normally, cells do not change state after bloomtime. The only exceptions to this rule occur during reconfiguration. Although the details of fault isolation and reconfiguration are not given here, the technique is essentially the same as in [Kum84], wherein cells which detect faulty neighbors assume a special quarantine state, Q. This effectively isolates faulty cells from the rest of the array. Kumar shows that if faulty or quarantine cells are part of the current control pattern, a distributed technique can be used to clear the pattern in the remaining fault-free regions, and to choose a single quarantine cell to internally reseed the array [Kum84]. No conflict arises if faults occur during the

pattern growth process, as the faulty pattern is immediately cleared, and a new reconfiguration cycle is initiated according to [Kum84, Bri87]. Kumar's method uses a special clear state (Z) to clear the array to the 0-state in the presence of faults which does not reset faulty or quarantine cells. The Z-state can be injected into the array to clear out the current pattern, allowing the array to be externally reseeded. If either internal or external seeding occurs, the seed migrates to an appropriate area, and a new pattern growth cycle begins.

2.4 LOCAL RECONFIGURATION

The global reconfiguration strategy described above is expensive when applied as a general response to faults in the array. It is not difficult to imagine a scenario in which a single fault triggers the global reconfiguration mechanism, and the pattern is relocated within the array, only to have another single fault reinitiate the sequence. A better approach might be to employ a local reconfiguration scheme in response to small numbers of faults. Thus the expense of global reconfiguration only comes into play when local reconfiguration fails. Such a hierarchical system is developed in [Whi91].

2.4.1 THE PROCESSING ELEMENT

Although White considers many different local reconfiguration strategies, Algorithm 4-12, given in Chapter 5 of [Whi91], produces the most promising results. Figure 2.23 shows that in this algorithm, each PE is physically connected to 12 of its neighbors: North, South, East, West, Northeast, Northwest, Southeast, Southwest, Far North, Far South, Far East, and Far West. To alleviate confusion, the names of physical

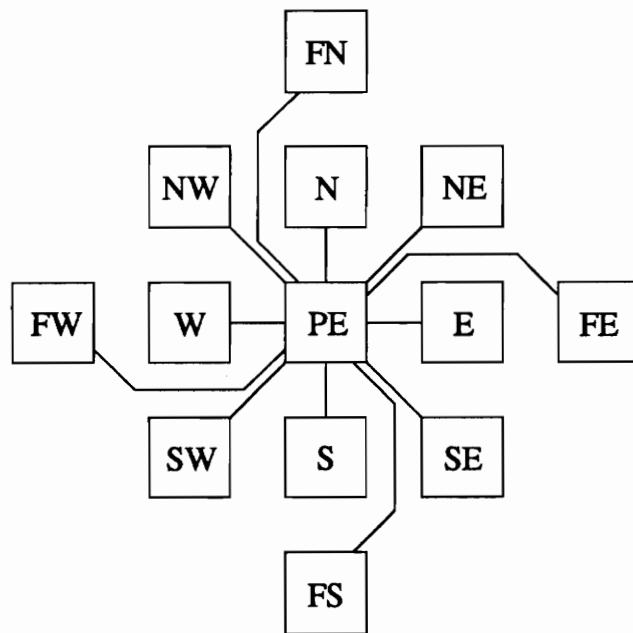


Figure 2.23 - Cell interconnection pattern used in White's Algorithm 4-12.

neighbors are distinguished by initial uppercase letters, and the names of logical neighbors are given completely in uppercase. Thus, the North physical neighbor of a given cell may also be that cell's NORTH logical neighbor. Each PE uses only 4 of the 12 available physical connections at a given time, corresponding to the logical NORTH, SOUTH, EAST, and WEST neighbors. This results in a rectangular mesh pattern for the logical array. As faults occur in the array, PEs may choose different physical connections to maintain the logical array. An example of the results of the local reconfiguration algorithm, given in Figure 2.24, shows that PEs simply disconnect themselves from faulty neighbors, and use connections to other neighbors to maintain the logical rectangular mesh pattern. This approach is reminiscent of the fault-stealing strategies outlined in Section 2.2.2.3.

2.4.2 THE FAULT REGISTER

Since reconfiguration is assumed to occur in a distributed fashion, some means of notifying PEs of faulty neighbors without relying on a centralized controller is needed. In [Whi91], each cell contains a *fault register*, whose bits indicate the presence of faulty cells in the neighborhood of the given cell. This register also contains information pertaining to bypassed faults in the same or adjacent rows as the given cell. Each PE regularly passes the contents of its fault register to its neighbors during operation, by which other neighbors become aware of faulty PEs. More information on the fault register is given in the fault diagnosis analysis in Chapter 4.

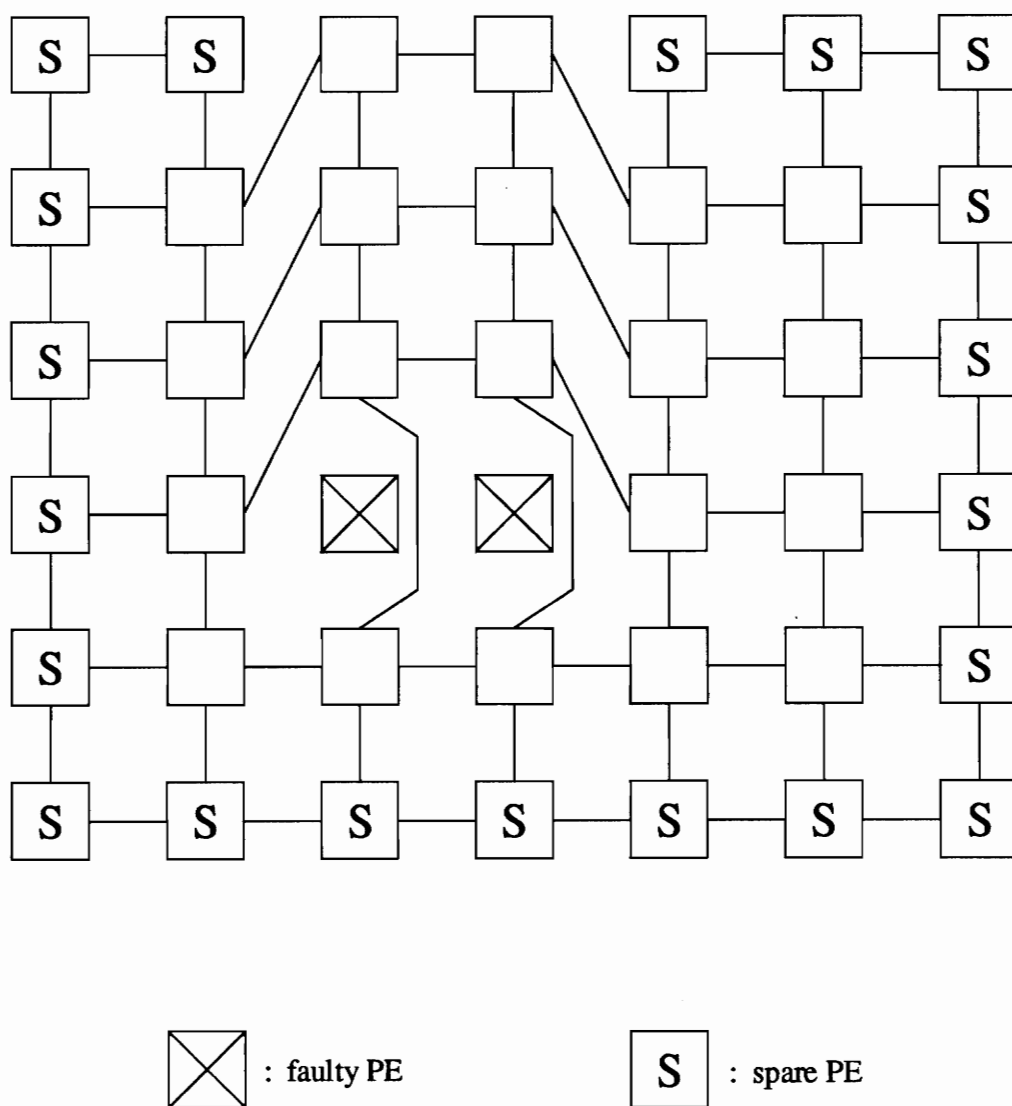


Figure 2.24 - End result of local reconfiguration in response to a double fault.

2.4.3 FAULT COVERAGE

Of the algorithms presented in [Whi91], Algorithm 4-12 offers the best balance between hardware overhead and fault coverage capability. The algorithm is 2-fault-tolerant, as it covers all double faults. For an 8-by-8 logical array with 2 spare rows and 2 spare columns, the resulting 10-by-10 physical array covers 99.7% of the triple-fault patterns as well [Whi91]. In fact, the only triple faults not tolerated in any array are those in which one of the faulty elements is adjacent in the same row as one of the other faulty elements, and is also adjacent in the same column as the remaining faulty element. These faults are termed *L-faults*, due to the resemblance of the fault pattern to the capital letter "L." An example L-fault pattern prior to local reconfiguration is given in Figure 2.25. In Chapter 4, an extension to White's fault register is examined which allows 100% coverage of triple-faults, thus making the array 3-fault-tolerant.

2.5 SUMMARY

This chapter provides the context for the remainder of the thesis. Systolic arrays are introduced as a means to reduce bandwidth requirements in compute-bound processes by performing all calculations which use a common data item in parallel. The regular structure of systolic arrays makes them suitable for VLSI/WSI implementation, but the resulting systems are often fault intolerant and highly application specific. Thus, it is desirable to make such systems reconfigurable to enhance yield, to dynamically tolerate faults, and to provide array polymorphism. Several examples of reconfigurable processing arrays are presented, but none are applicable to all three reconfiguration scenarios, with the exception of the proposed hierarchical reconfiguration strategy. This hierarchical

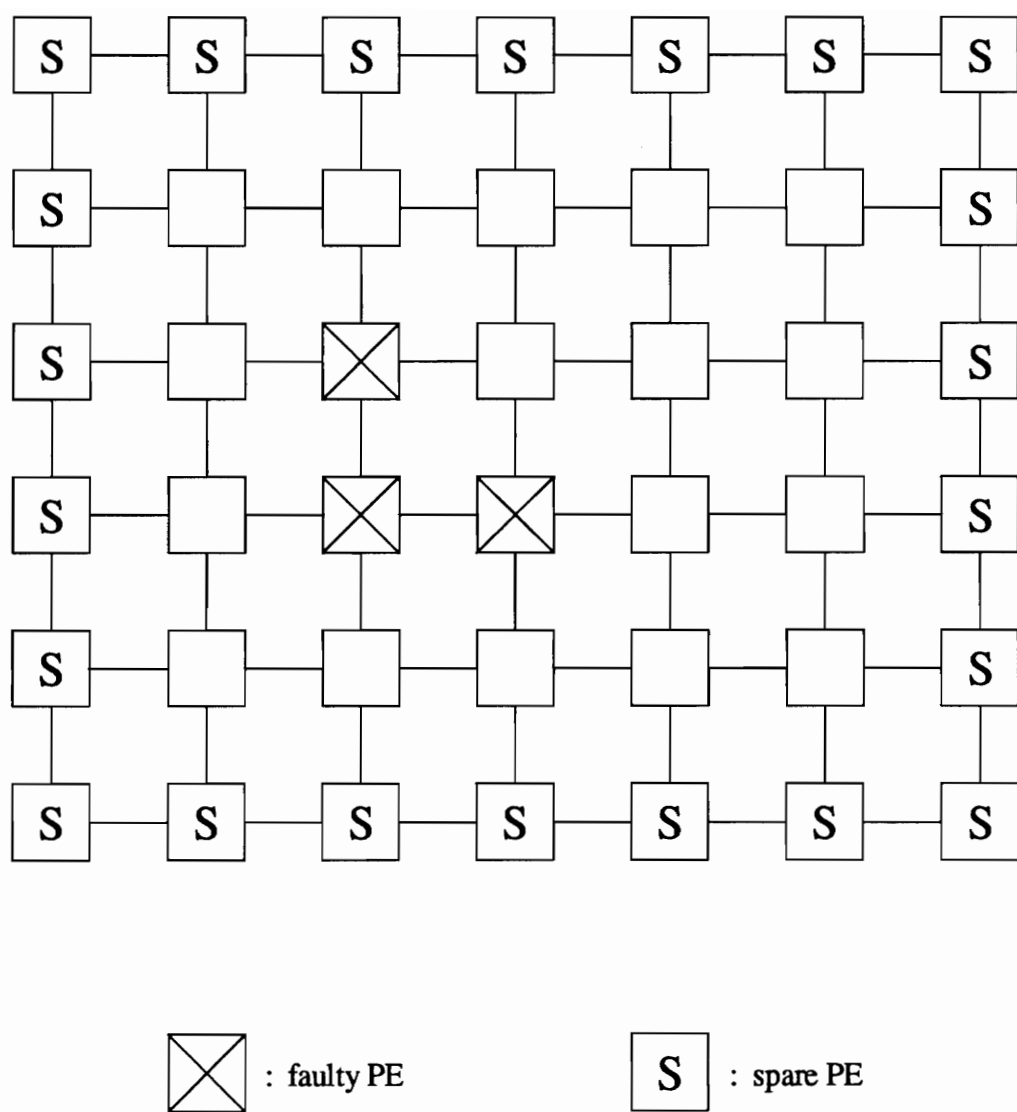


Figure 2.25 - Fatal L-fault pattern example prior to local reconfiguration attempt.

scheme incorporates a local reconfiguration mechanism as a primary approach to fault tolerance, and global reconfiguration algorithm to provide array polymorphism as well as a secondary means of achieving fault tolerance.

The local reconfiguration scheme presented in [Whi91] allows the array to bypass faulty cells in a local sense by allocating spare rows and columns of processing elements. This enhancement provides the array with more flexibility in tolerating faults. All double-faults are tolerated, as well as most of the triple-fault patterns, with the exception of the L-fault patterns.

If the local reconfiguration scheme fails, the global reconfiguration strategy scheme presented in [Bri87] takes effect. This scheme allows distributed control patterns to be grown in a cellular array, regardless of the number of faulty cells present, provided that an ample fault-free area exists. Reconfiguration can be initiated either in response to the presence of faults, or to provide array polymorphism. The method is not subject to hard core failures. In addition, it does not require that control cells be excessively reliable, and it does not assume the existence of components which never fail.

The following chapters present new material which addresses some of the questions concerning previous system development. These issues include the efficiency of the current global reconfiguration scheme, distributed diagnosis of faults in the array, complete triple-fault coverage, and concurrent online fault detection.

3. RECONFIGURATION COMPLEXITY ANALYSIS

This chapter addresses questions raised concerning the efficiency of the global reconfiguration scheme presented in [Bri87] as compared to those given in [Mar80] and [Kum84]. For each method being considered, this chapter compares the upper bounds on the size complexities of various parameters which characterize the state transformation function. These parameters include the size of a cell's local memory, the number of cell interconnection wires, and the size in bits of the data to be passed between processing elements, as required by the state transformation function. It is assumed that data sent from a cell to a given neighbor is transmitted serially over a single wire. This reduces the routing complexity which would otherwise be involved in the circuit layout if data is transmitted in parallel.

3.1 EXPLANATION OF SYMBOLS USED

The total number of local states required by a given method is denoted by n_L , the symbol for the number of global states is n_G , and the maximum control pattern dimension is represented by D . The neighborhood scope defined in Chapter 2 is indicated by n . To simplify the comparison, it is assumed that all global patterns are the same size, that a given method always uses the same neighborhood, and that the same patterns are grown using each of the three methods. Note that Martin's method is restricted to square patterns with sides of length $D = k^2$, for some integer k . Therefore, all three methods are analyzed for these cases of square pattern growth. The values of n_L and n are dependent upon the method used. However, both n_G and D are independent of the method used. Hence, complexity analysis results are given in terms of those parameters.

3.2 LOCAL MEMORY COMPLEXITY

This section compares the size of the local memory needed by a cell to implement the next-state lookup table. State information received from a cell's neighbors, as well as the cell's own current state, determine the cell's transition to a new state. Although the next state function could be implemented as discrete logic for a given array, the lookup table approach provides a consistent methodology for any array.

3.2.1 MARTIN

Martin's method requires a unique neighborhood configuration for each cell participating in pattern growth, as well as for each cell in the final pattern [Mar80]. If one global pattern with n_L local states is grown with n neighbors for each cell, there are n_L^n possible neighborhoods for each cell. Hence, the memory size complexity of the lookup table for Martin's method is given by

$$M = O(n_G \cdot n_L^n). \quad (3.1)$$

From [Mar80], the dimension, D , of the square pattern must be the square of an integer, k . Martin's method requires k local states to provide unique neighborhoods required for each cell in the pattern. That is, $D = k^2$, and $k = n_L$. Thus, the relationship between the number of local states and the maximum pattern dimension, D , is fixed, and is given by

$$n_L = D^{0.5}. \quad (3.2)$$

The next-state mapping function requires knowledge of the states of each of a cell's neighbors, as well as the cell's own state. Therefore, the neighborhood scope for

Martin's method must be determined. The minimum neighborhood scope, n , required to reconfigure in the presence of single faults for [Mar80] is given by

$$n = 12j + 3, \quad (3.3)$$

for some integer $j > 1$. Martin shows that the neighborhood scope increases linearly with the number of faults to be tolerated. By letting $j = 2$ in Equation 3.3, we see that Martin's method has a *minimum* neighborhood scope given by

$$n = 27. \quad (3.4)$$

By substituting Equations 3.2 and 3.4 into Equation 3.1, we find that Martin's method has a *minimum* memory size complexity of

$$M = O(n_G \cdot D^{13.5}). \quad (3.5)$$

3.2.2 KUMAR

Like Martin's method, Kumar's method also requires a unique neighborhood configuration for every cell in the control pattern during pattern growth. This constraint is lifted for the final pattern mapping at bloomtime. A set of intermediate pattern growth states is required for every time step prior to bloomtime, so that the unique neighborhood condition is satisfied. Since patterns in Kumar's method grow in a diamond shape due to the use of a von Neumann neighborhood, a diamond of sufficient size must be grown so that it circumscribes a square area of the desired dimension, D . The following definitions and theorems provide insight into the relationship between the size of a diamond, the size of the inscribed square, the time needed to establish the diamond, and the number of distinct neighborhoods at each time step of Kumar's pattern growth process.

Definition 3.1:

A *diamond* of girth G , where G is odd, consists of consecutive rows of lengths $1, 3, 5, \dots, (G-2), G, (G-2), \dots, 5, 3, 1$, where the middle cell of each row is in the same column as that of the other rows. Note that the girth of the diamond is the length of the longest row (or column) in the diamond. Figure 1 shows the general form for a diamond. Note that a valid diamond of girth G exists if and only if G is odd, since each row (or column) is of odd length.

From [Kum84], a diamond of girth G , where G is odd, contains rectangles of dimension $1 \times G, 3 \times (G-2), \dots, (2m-1) \times (G-2(m-1)), \dots, (G-2) \times 3, G \times 1$. Note that $1 \leq m \leq (G+1)/2$. This is obtained by observing that a rectangle has minimum dimension 1. The general expression for the dimension of a rectangle is $(2m-1) \times (G-2(m-1))$, which defines the limits on m . Since $(2m-1) \geq 1$, we have $m \geq 1$. From $G-2(m-1) \geq 1$, we see that $G-2m+2 \geq 1$, or $m \leq (G+1)/2$.

Theorem 3.1:

The girth, G , of a diamond is related to the length of a side, D , of the largest square which can be inscribed in it by $G = 4\lfloor D/2 \rfloor + 1$.

Proof:

If D is odd, then we can find the relationship between D and G from the general expression for the dimension of a rectangle. A square is simply a rectangle of dimension $D \times D$. From $D = 2m-1$, we see that $m = (D+1)/2$. By substituting this result into $D = G-2(m-1)$, we find that $D = G-2((D+1)/2-1)$, or $G = 2D-1$. Since D is odd, $2\lfloor D/2 \rfloor + 1 = D$, so $G = 2(2\lfloor D/2 \rfloor + 1) - 1 = 4\lfloor D/2 \rfloor + 1$. If D is even, we can embed the

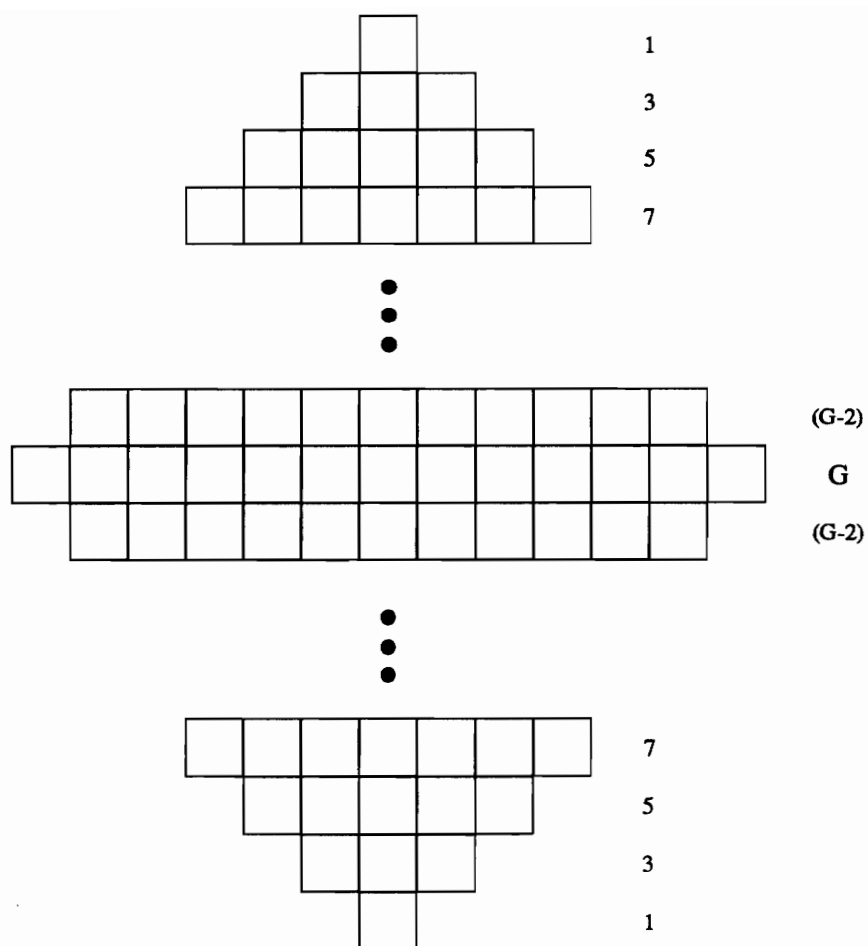


Figure 3.1 - A diamond pattern of cells in a processor array.

square in a larger square with sides of length $(D + 1)$. Note that we must do this, since the rectangles contained within the diamond always have odd values for dimensions. From $D + 1 = 2m - 1$, we see that $m = (D + 2)/2$. By substituting this result into $D + 1 = G - 2(m - 1)$, we find that $D + 1 = G - 2((D + 2)/2 - 1)$, or $G = 2D + 1$. Since D is even, $2\lfloor D/2 \rfloor = D$, so $G = 4\lfloor D/2 \rfloor + 1$.

Theorem 3.2:

The number of time steps, t , required to grow a diamond of girth G , where G is odd, is given by $t = (G + 1)/2$.

Proof by Induction:

Trivial Case:

A single seed cell at $t = 1$ is a diamond of girth $G = 1$.

Assumption:

The number of time steps, t , required to grow a diamond of girth $G = g$, where g is odd, is given by $t = (g + 1)/2$.

Induction Step:

Show that the number of time steps, t , required to grow a diamond of girth $G = (g + 2)$, where $(g + 2)$ is odd, is given by $t = (g + 3)/2$.

Since g is odd, $(g + 2)$ is also odd, so a valid diamond of girth $(g + 2)$ exists, by Definition 3.1. In Kumar's pattern growth process, a valid diamond of girth $(g + 2)$ is constructed from a valid diamond of girth g by adding one cell to each end of each row, and by adding one cell to the top and bottom of the diamond. Thus the longest row

(column) is of length $(g + 2)$, as expected. Note that only one additional time step is required to accomplish this result. Thus, $t = ((g + 1)/2) + 1 = (g + 3)/2$.

Each of the intermediate neighborhoods requires a separate entry in the local lookup table. The number of unique neighborhoods, n_N , in the diamond pattern at time t is the same as the number of cells in the pattern at the next time step [Bri87], and is given by

$$n_N = (t+1)^2 + t^2 = (G+1)/2 + G + 2 \quad (3.6)$$

Since one memory word must be allocated for each neighborhood at each step of the pattern growth process, the total number of memory words required is approximately given by summing Equation 3.6 as the diamond pattern begins as a single seed, and grows to its final size. An average value of $G = 2D$ is used to simplify the calculations. Neglecting the constant values in Theorem 3.1 has no effect on the final expression for the memory complexity function.

$$n_L = \sum_{G=1}^{2D} [(G+1)^2/2 + G + 2] \quad (3.7)$$

$$n_L = (4D^3 + 17D^2 + 15D)/3 \quad (3.8)$$

By Equation 3.8, Kumar's method has a memory size complexity of $O(D^3)$ for a single global function, and hence a total memory complexity of

$$M = O(n_G \cdot D^3) \quad (3.9)$$

3.2.3 BRIGHTON

During pattern growth, Brighton's method requires only the single lookup which is performed at bloomtime. For a single global pattern, this lookup is dependent only on the

contents of the XR and YR registers, each of which is restricted to a maximum value of D . A single global pattern has a maximum of $n_L = D^2$ cells, which gives a total memory size complexity of

$$M = O(n_G \cdot D^2) \quad (3.10)$$

3.3 INTERCONNECTION COMPLEXITY

This section of the complexity analysis focuses on the number of wires used to connect a cell to its neighbors. Clearly, minimizing the number of such wires leads to a minimal layout complexity. Data is transmitted in a serial fashion, so that only a single wire is needed for two neighboring cells to exchange information. A multi-wire bus could also be used, reducing transmission time at the expense of increased layout area.

3.3.1 MARTIN

In order for a cell to perform the next-state function, it must know the state information for each of its neighbors. The minimum number of cell interconnection wires required by Martin's method to communicate this information is one less than the number of neighbors, since a cell needs no wires to communicate with itself. The resulting expression for the minimum number of wires, using the minimum neighborhood size from Equation 3.4, is given by

$$W = (n - 1) = 26 \quad (3.11)$$

since a cell needs no external routing to determine its own state value. Since the neighborhood scope increases linearly with the number of faults, as mentioned above, so too does the number of cell interconnection wires.

3.3.2 KUMAR

A von Neumann neighborhood is used in [Kum84], which gives

$$n = 5 \tag{3.12}$$

Thus, the number of interconnect wires per cell is given by

$$W = (n - 1) = 4 \tag{3.13}$$

Note that the neighborhood scope, and hence the number of interconnection wires, remains constant, regardless of the number of faults to be tolerated.

3.3.3 BRIGHTON

Since Brighton's method uses a von Neumann neighborhood, the number of cell interconnection wires is the same as in Kumar's method. The result is therefore the same as for Kumar's method, and is given in Equation 3.13 above.

3.4 TRANSMITTED DATA COMPLEXITY

This section of the complexity analysis compares the size in bits of the data to be transmitted between neighboring cells. Since this data is transmitted serially, the size of the data packets is directly related to the amount of time required to send them. As in the previous section, a multi-wire bus implementation effectively results in a constant data transmission time over the three methods under consideration, at the expense of increased layout area.

3.4.1 MARTIN

The number of bits to be transmitted is determined by the encoding of a cell's state value, which includes both the global and local state values. Thus, the number of bits to be transmitted is given by

$$b = \log_2(n_G + \log_2(n_L)) \quad (3.14)$$

By substituting Equation 3.2 into Equation 3.14, we see that the total number of transmitted bits is given by

$$b = \log_2(n_G) + \log_2(D^{0.5}) \quad (3.15)$$

3.4.2 KUMAR

Again, the number of bits which must be sent is equal to the size of the global and local state values, as shown in Equation 3.14. By substituting Equation 3.8 into Equation 3.14, we see that the total number of transmitted bits is given by

$$b = \log_2(n_G) + \log_2((4D^3 + 17D^2 + 15D)/3) \quad (3.16)$$

3.4.3 BRIGHTON

As in the previous discussions, the number of bits which must be sent to each neighbor is dependent on the size of the global and local state values. Recall from Section 2.3.3.2 that a cell receives one position value from an east or west neighbor, and the other position value from a north or south neighbor. Thus, in addition to the global and local state information, the data passed between neighboring cells must also include one of the position register values, XR or YR. Thus, the total number of bits is given by

$$b = \log_2(n_G) + \log_2(n_L) + \log_2(PR) \quad (3.17)$$

where PR represents the maximum position register value, which is D . Substituting this value and $n_L = D^2$ into Equation 3.17 gives the total number of bits as

$$b = \log_2(n_G) + \log_2(D^2) + \log_2(D) \quad (3.18a)$$

$$b = \log_2(n_G) + \log_2(D^3) \quad (3.18b)$$

3.5 COMPLEXITY ANALYSIS RESULTS

Table 3.1 summarizes the results of the global reconfiguration complexity analysis. The first column of the table demonstrates a considerable improvement in the memory size complexity of Brighton's method over that found for Martin's and Kumar's methods, respectively. The second column of Table 3.1 shows that the interconnection complexities of both Kumar's and Brighton's methods are dramatically improved as compared to Martin's method. Finally, the third column of the table indicates that more information bits must be transmitted by Kumar's method as compared to Martin's method, but that the number of information bits required by Brighton's method lies between these two extremes. The additional time required by Brighton's method to transmit the greater number of bits, as compared to Martin's method, is offset by the greatly reduced number of cell interconnection wires, as well as the significantly decreased memory size complexity.

3.6 CONCLUSIONS

This chapter derives the relative complexities of the global reconfiguration schemes presented in [Mar80], [Kum84], and [Bri87], with respect to the size of local memory, the number of cell interconnections, and the number of bits transmitted between neighboring cells. It is appropriate at this point to briefly mention the limitations of the comparisons performed in this chapter.

Recall that Martin's method is restricted to the growth of square patterns with sides of length $D = k^2$, where k is some positive integer. Thus, any desired pattern must

Table 3.1 - Complexity of Global Reconfiguration Overhead

Method	Memory Requirement	Interprocessor Wires	Data Bits Transferred
Martin	$O(n_G \cdot D^{13.5})$	26	$\log_2(n_G) + \log_2(D^{0.5})$
Kumar	$O(n_G \cdot D^3)$	4	$\log_2(n_G) + \log_2((4D^3 + 17D^2 + 15D)/3)$
Brighton	$O(n_G \cdot D^2)$	4	$\log_2(n_G) + \log_2(D^3)$

Note: Martin's method assumes single-fault tolerance only

fit within such a square, resulting in potentially poor cell utilization in the general case, since the required number of available cells may be much greater than the number of cells actually used. Kumar's method requires that the desired pattern be inscribed within a diamond of the appropriate girth. In Brighton's method, the final pattern must fit within a rectangle of the appropriate size. Clearly, Kumar's and Brighton's methods must be heavily restricted in order to fairly compare them with Martin's method.

If Kumar's and Brighton's methods are compared directly, however, the worst-case complexity costs for both methods are of the same order as presented in the above sections. This can be seen if the maximum pattern dimension parameter, D , is defined to be the length of the longest side of a rectangular pattern. In a worst-case scenario, the pattern would be square, with sides of length D . Obviously, the number of interconnection wires does not change in a direct comparison. Thus, the same analysis applies as given in Sections 3.2 through 3.4.

Brighton's method has modest local memory requirements to implement the local transformation function, as compared to both [Mar80] and [Kum84]. Brighton's method also requires fewer information bits to be transmitted between neighbors, as compared to Kumar's method. Martin's method requires fewer information bits than Brighton's method. However, the reduction in the amount of transmitted data comes at the expense of a large number of cell interconnections, which increases linearly with the number of faults to be tolerated. In contrast, both Brighton's and Kumar's methods employ a small, fixed neighborhood scope which is independent of the number of faults to be tolerated.

The reduced size of the next-state lookup table, as well as relatively small number of cell interconnection wires, makes Brighton's scheme a more attractive alternative for global reconfiguration as compared to the techniques developed by either Martin or Kumar. Therefore, the results presented in subsequent chapters of this thesis assume Brighton's approach is used as the basis for the global reconfiguration algorithm.

4. DISTRIBUTED FAULT DIAGNOSIS

This chapter discusses a distributed approach for pinpointing faulty PEs in a cellular array. Although the reconfiguration technique in [Dis89] addresses link failures, the governing algorithm relies on global fault knowledge in the array. The approach presented in this chapter eliminates the need for global knowledge of the locations of faulty PEs, and is easily integrated into the existing fault-tolerant cellular architecture. Cells in the control plane are assumed to be self-checking. This assumption guarantees that *fault latency*, the elapsed time between the occurrence of a fault and its detection, can be minimized. The means for making self-checking cells are examined in Chapter 5. As before, the cells in the computational plane are not considered here, given the potential functional diversity of those components.

In this distributed diagnosis scheme, fault latency is defined as the elapsed time between the occurrence of a fault and its detection by all cells which need to be aware of the fault. When this detection is accomplished, the fault is said to be *properly detected*. A fault which is not properly detected is said to be *improperly detected*, and may not result in a correct local reconfiguration. This occurs when cells which need to be aware of the fault are prohibited from learning of its existence, due to limitations in the hardware or software which manages the array.

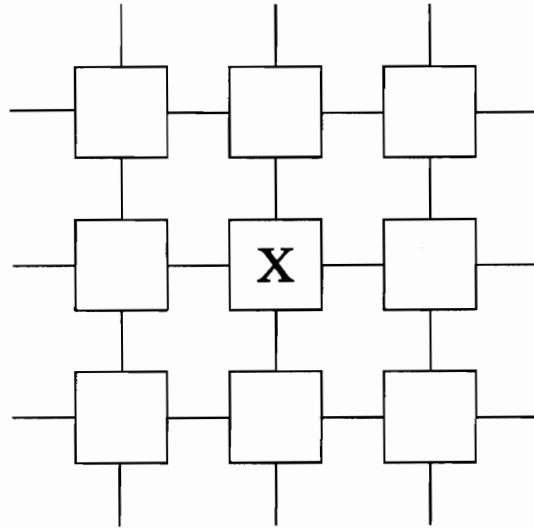
With these definitions in mind, several cell interconnection schemes, or neighborhoods, are examined with respect to fault latency. The neighborhood determines the time needed to propagate the fault data to other cells in the array. Each cell contains a *fault register*, which contains the relative locations of faulty cells within a given area

surrounding the cell. This area is known as the *region of fault awareness*, and this chapter examines the impact of this area on proper fault detection in detail. An explanation of the contents of the fault register, and how its contents are determined is also included.

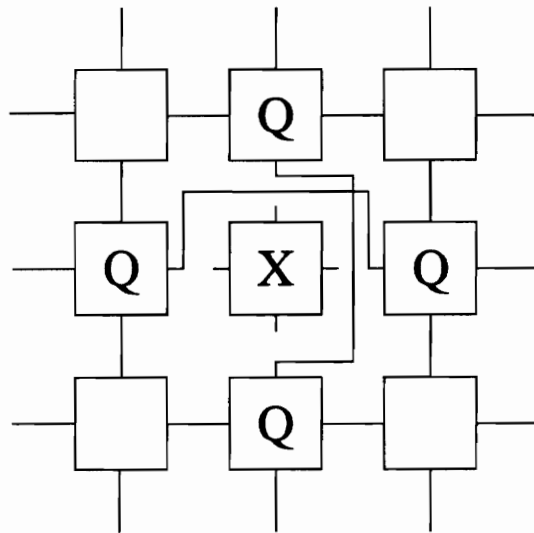
Finally, a simulation program is discussed which unifies the concepts of the cell neighborhood, fault latency, and the fault register. Worst-case simulation results indicate the detection of all triple faults is possible when a von Neumann neighborhood is used to pass the fault data, as well as the detection of all quadruple faults when certain other neighborhoods are used. The simulation results also provide the maximum fault latency for successful detection, with respect to the neighborhood used.

4.1 MOTIVATION FOR DISTRIBUTED DIAGNOSIS

The previously developed fault-tolerant cellular architecture assumes that as faults occur in the array, all PEs adjacent to faulty PEs enter a special quarantine state, thus isolating the faulty regions from the fault-free regions. An example of this behavior is given in Figure 4.1. In this case, adjacency is defined in terms of the von Neumann neighborhood, which consists of the PEs to the north, south, east, and west of the faulty PE. In order to distinguish directions from the names of neighboring cells, the former are given in lower case, and the latter begin with uppercase letters. Thus, the neighbors in the von Neumann neighborhood are, in clockwise order: North (N), East (E), South (S), and West (W). The faulty PE, denoted by the symbol **X**, is isolated from the fault-free portion of the array by the four quarantine PEs, denoted by the symbol **Q**. The figure implies the faulty PE is immediately and simultaneously detected by all four neighbors. This further implies that the fault latency time is zero.



a) A fault occurs...



b) ... and is immediately detected!

Figure 4.1 - The zero fault latency model.

One possible way to achieve zero fault latency is through the concept of self-checking circuits. Such circuits allow faults to be detected at the output as they occur. This eliminates the need for a periodic off-line testing cycle, during which all processing is suspended. Off-line testing is undesirable in this context, since fault latency is directly related to the interval between testing cycles. Self-checking circuits are discussed in Chapter 5. For the purposes of this discussion, the assumption is made that the cells in the control plane are self-checking.

Even with the assumption of self-checking circuitry, however, there is no way to guarantee simultaneous fault detection by all neighbors, as is assumed in Figure 4.1. Consider the example in Figure 4.2, in which the center cell is connected to each of its neighbors via bidirectional communications links. If a failure develops in the link to the north, as shown in the figure, the North neighbor, N, becomes aware of that fault via the self-checking assumption. However, the other three neighbors have no way to detect the fault directly; they must be informed of the fault by N. Other cells besides the four nearest neighbors may also need to be aware of the faulty cell in order to successfully reconfigure the array. As previously discussed, a global control mechanism limits the fault-tolerance of the array. Therefore, some distributed mechanism is needed for carrying information concerning the locations of faulty cells from the detecting neighbors to other cells in the array.

Obviously, some fault latency is incurred in allowing the information to spread from the point of origin to all fault-free cells which need to be aware of faults in the array. Furthermore, since not all the fault-free cells receive fault information simultaneously, an upper bound on the time required to distribute the data is needed. This bound, denoted

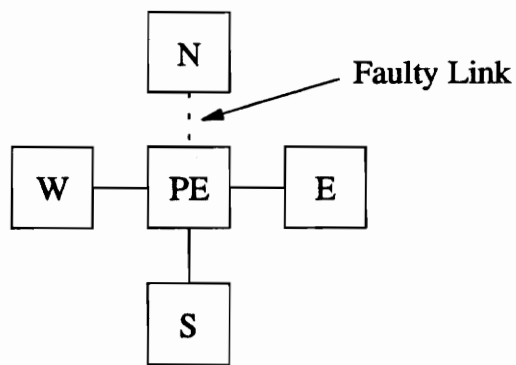


Figure 4.2 - The link fault cannot be detected by any neighbor of the PE other than N.

t_{MAX} , is determined by a number of factors, including the neighborhood used to distribute the fault information, the size and shape of the area of cells surrounding a faulty cell which need to be aware of the fault, and the maximum number of faults to be tolerated. Once t_{MAX} is determined, a cell which becomes aware of a fault need only wait t_{MAX} time steps before attempting to reconfigure the array. This waiting period guarantees that for any tolerable fault pattern, all cells needing to be aware of the fault are notified within t_{MAX} time steps. The following sections of this chapter investigate the interrelationships of these factors, and their impact on t_{MAX} .

4.2 THE CELL NEIGHBORHOOD

As defined in Chapter 2, a cell's neighborhood consists of all cells to which the cell is connected, as well as the cell itself. The number of cells in the neighborhood defines the neighborhood scope. The von Neumann neighborhood shown in Figure 4.3 has a scope of 5. The Moore neighborhood of Figure 4.4, and the White neighborhood of Figure 4.5 have scopes of 9 and 13, respectively. Note that the Moore neighborhood simply augments the von Neumann neighborhood by adding the diagonal neighbors: Northeast (NE), Southeast (SE), Southwest (SW), and Northwest (NW). The White neighborhood extends the Moore neighborhood to include the following neighbors: Far North (FN), Far East (FE), Far South (FS), and Far West (FW). Thus, a von Neumann neighborhood requires less hardware to support the necessary cell interconnections than either the Moore or White neighborhoods, due to its reduced scope. This hardware can take the form of the interconnection links, as well as buffers, multiplexers, and other components needed to manage the additional transfer of information between cells.

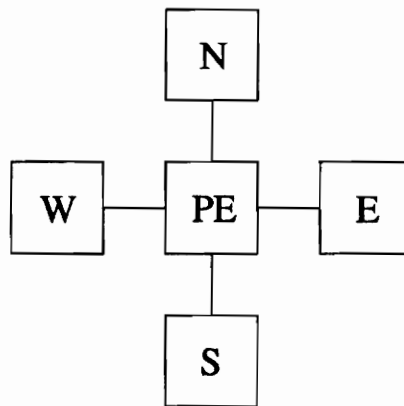


Figure 4.3 - The von Neumann cell neighborhood has a scope of 5.

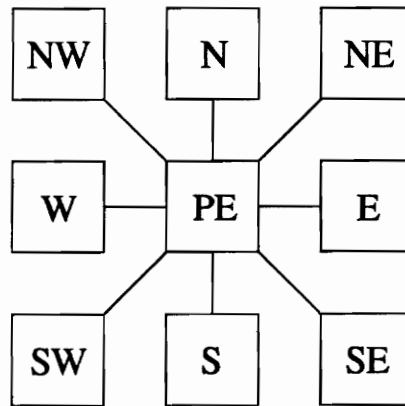


Figure 4.4 - The Moore cell neighborhood has a scope of 9.

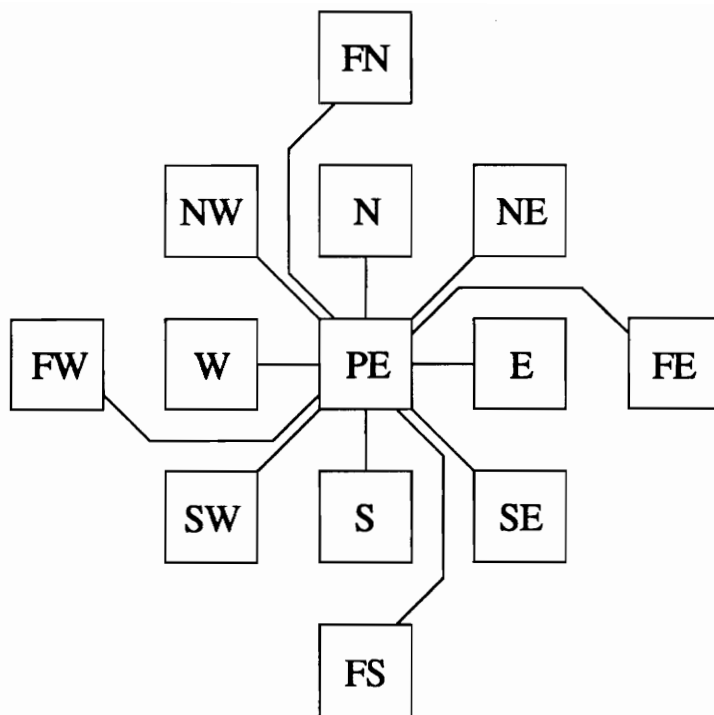


Figure 4.5 - The White cell neighborhood has a scope of 13.

The White neighborhood derives its name from the researcher who created the local reconfiguration scheme of [Whi91], as described in Chapter 2. In this scheme, a distinction is made between the physical neighborhood and the logical neighborhood. Physically, all cells in the array are connected as in Figure 4.5. Logically, however, the array is viewed as a rectangular mesh of cells connected in the von Neumann fashion of Figure 4.3. During local reconfiguration, internal multiplexers are set which map the physically available connections to the proper logical connections, so that the logical mesh is maintained. As shown in Figures 4.6 through 4.9, a cell can choose from one of four cells in the physical neighborhood to serve as a particular logical neighbor. Since there are only 12 physical neighbors, the diagonal physical neighbors can serve as one of two logical neighbors. For example, the southeast neighbor can either be viewed as the EAST logical neighbor, as shown in Figure 4.7, or as the SOUTH logical neighbor, as in Figure 4.8.

Figure 4.10 introduces another neighborhood variation. It adds the Far North (FN), Far South (FS), Far East (FE), and Far West (FW) neighbors to the von Neumann neighborhood. This particular configuration is referred to here as the *Lawson neighborhood*. It will be shown that this neighborhood offers many of the best qualities found in the other neighborhoods which are particularly useful for the distributed fault diagnosis algorithms developed here. Specifically, the fault latency incurred when a Lawson neighborhood is used to pass the fault data is significantly lower than when a von Neumann neighborhood is used. It is important to note that even though a von Neumann neighborhood is currently used to pass the fault data, much of the hardware needed to pass the fault data via a Moore, Lawson, or White neighborhood is already present in the White interconnection network assumed for local reconfiguration in [Whi91]. It is shown that although some additional hardware is required to support the Lawson neighborhood

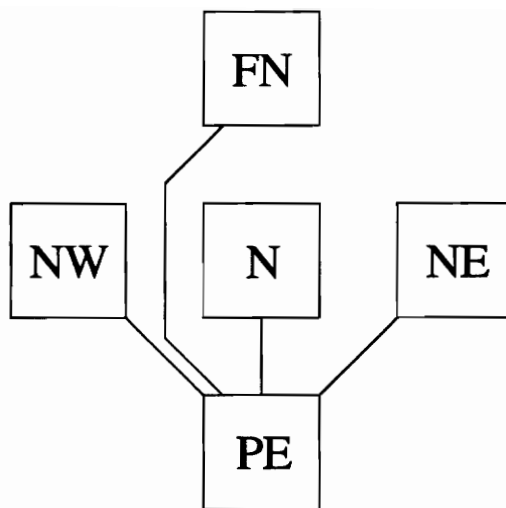


Figure 4.6 - The possible choices for a NORTH logical neighbor.

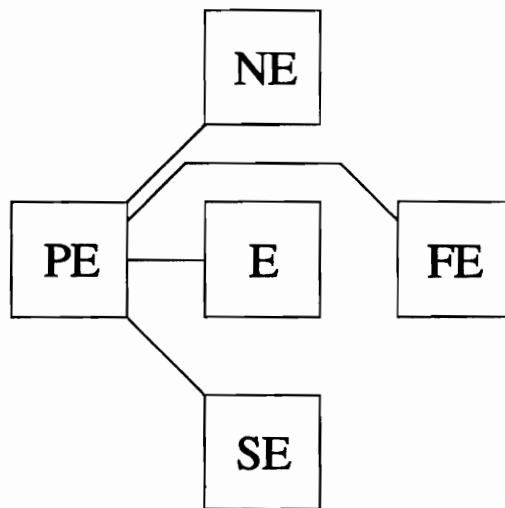


Figure 4.7 - The possible choices for an EAST logical neighbor.

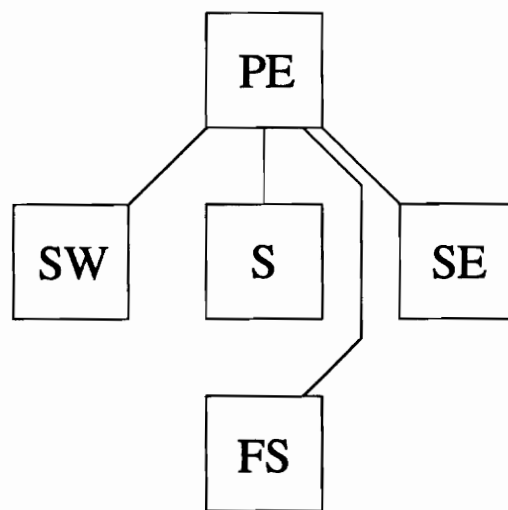


Figure 4.8 - The possible choices for a SOUTH logical neighbor.

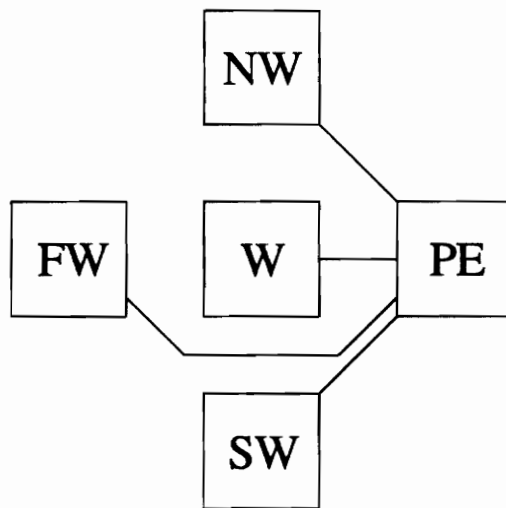


Figure 4.9 - The possible choices for a WEST logical neighbor.

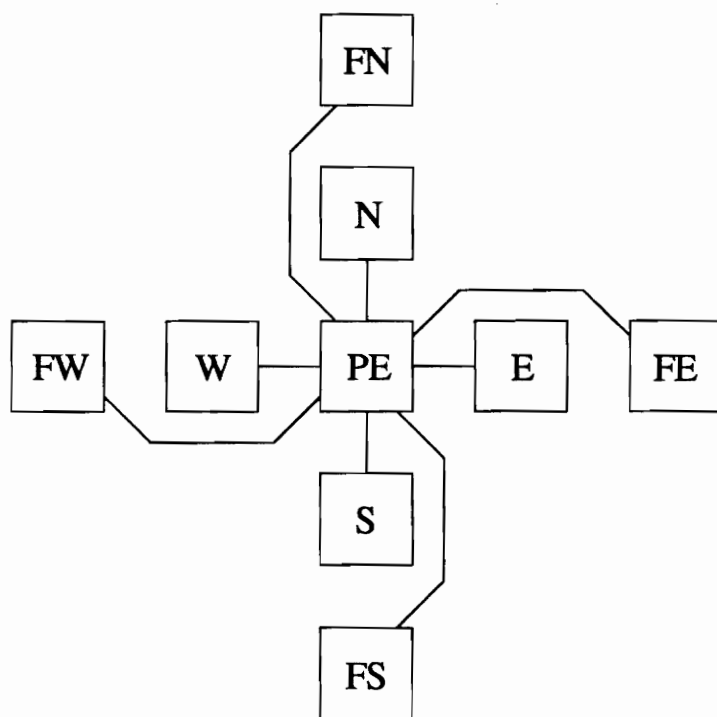


Figure 10 - The Lawson cell neighborhood has a scope of 9.

as a fault detection mechanism, the fault latency time is considerably lower than that of the Moore neighborhood, which has the same scope, and hence the same hardware overhead requirements, as the Lawson neighborhood. Furthermore, the substantially higher hardware investment required to support the White neighborhood for passing fault data does not result in a significant reduction in the fault latency time over the Lawson neighborhood.

The neighborhood scope directly relates to the amount of hardware needed in each cell to support the interconnections required by the neighborhood. This hardware consists of not only the requisite communications links, but also includes circuitry internal to the cell, such as buffers, multiplexers, and combinational logic. The White neighborhood developed for the local reconfiguration algorithm assumes four multiplexers and four buffers for passing fault data between neighbors [Whi91]. Hence, the communications links, multiplexers, and some of the data buffers and combinational logic are already present in the current architecture. However, since a logical von Neumann neighborhood is currently used for passing data between processors, additional hardware is required if a different neighborhood is used for passing the fault data. None of the neighborhoods examined in this chapter require additional communications links, so the only hardware cost is the additional buffering and combinational logic needed. It is desirable to minimize the amount of additional hardware needed, so that the layout complexity and power consumption of the cell are conserved.

4.3 THE FAULT REGISTER

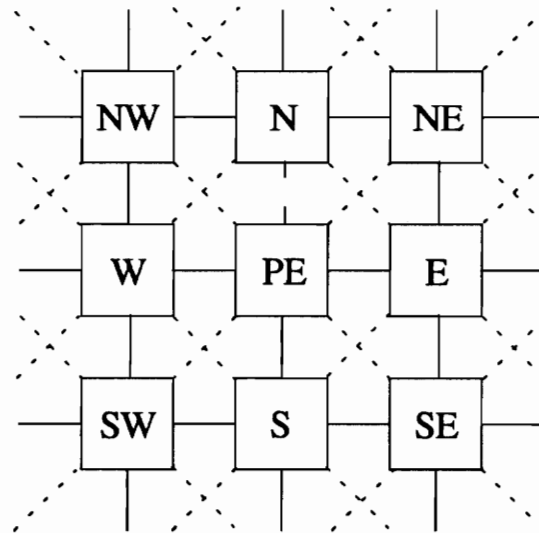
Each cell in the control plane maintains the contents of a *fault register*. This register is simply a set of bit-flags which indicate the faulty or fault-free status of cells in the area surrounding the given cell. Assuming, for example, that the fault register contains flags which correspond to the White neighborhood of Figure 4.5, then the fault register is 12 bits wide. The region defined by the fault register flags is called the *region of fault awareness*, or simply the *region of awareness*. Note that an exclusive and reciprocal relationship exists between any given cell and those cells in its region of awareness. In other words, any given cell is included in the region of awareness of each cell in its own region of awareness, and in no other such region. The region can be expanded to encompass as much area as desired by simply adding the appropriate flags to the fault register.

This chapter initially considers a fault register configuration based on the White neighborhood for distributed diagnosis, since the same configuration defines the physical neighborhood. That is, all cells which are physically connected to a faulty cell should be made aware of the fault. This approach will eventually enable all cells physically connected to a faulty cell to determine the necessary connection patterns for successful reconfiguration. Since there are twelve neighbors in the White neighborhood, the fault register contains twelve bits (one for each neighbor). In an attempt to conserve hardware overhead, a von Neumann neighborhood is assumed for passing the fault data, unless otherwise noted. Modifications will be made as needed over the course of the analysis to accommodate problematic fault configurations.

A cell sets its own fault register flags based on fault register data received from other cells. Figure 4.11 contains an example of this procedure. Suppose a faulty cell, denoted by **X**, is detected by its North neighbor, **N**. Further suppose that the Northwest neighbor of **X**, **NW**, receives fault data from its own East neighbor, **N**, and that in turn, **N** contains a flag indicating the fault in its South neighbor, **X**. If **NW** has a flag in its own fault register corresponding to its Southeast neighbor, **X**, then **NW** can deduce the fault in **X** by examining the South neighbor flag in the fault register data passed to it by **N**.

This deductive process continues until all cells in the region of awareness of **X** are notified of the fault. For example, the West neighbor of **X**, **W**, can be told of the fault at **X** by either **N** or **NW** if a White (or Moore) neighborhood is used to pass the fault data. Since there is a direct diagonal connection between **W** and **N**, both **W** and **NW** learn of the fault at **X** at the same time. If the diagonal connections are not enabled, such as if a von Neumann neighborhood is used to pass the fault data, **W** must learn of the fault at **X** from **NW**, who in turn learns of it from **N** as shown in Figure 4.11.

Figure 4.12 provides a complete example of single fault diagnosis. The faulty cell, **X**, is detected at $t = 0$ by its North neighbor, and is indicated by the numeral 0 at that position. The fault register is assumed to contain bits corresponding to a White neighborhood, as indicated by the region of awareness shown in the figure, since it is desirable to notify all cells physically connected to **X** to be aware of the fault. A von Neumann neighborhood is assumed for the purpose of transmitting the fault data. Thus at $t = 1$, the North, East, and West neighbors of the detecting cell become aware of the fault at **X**, as indicated by the numeral 1 in the appropriate positions. Note that at $t = 5$, the Far South neighbor of **X** is the last cell in the region of awareness to be made aware of the



S

...	1	...
-----	---	-----

North Fault Register

SE

...	1	...
-----	---	-----

Northwest Fault Register

E

...	1	...
-----	---	-----

West Fault Register

W can directly learn about the fault at **X** from **N** if the diagonal connections are enabled. Otherwise, **W** learns about the fault at **X** from **NW**, which first learns about it from **N**.

Figure 4.11 - Setting the fault register bits.

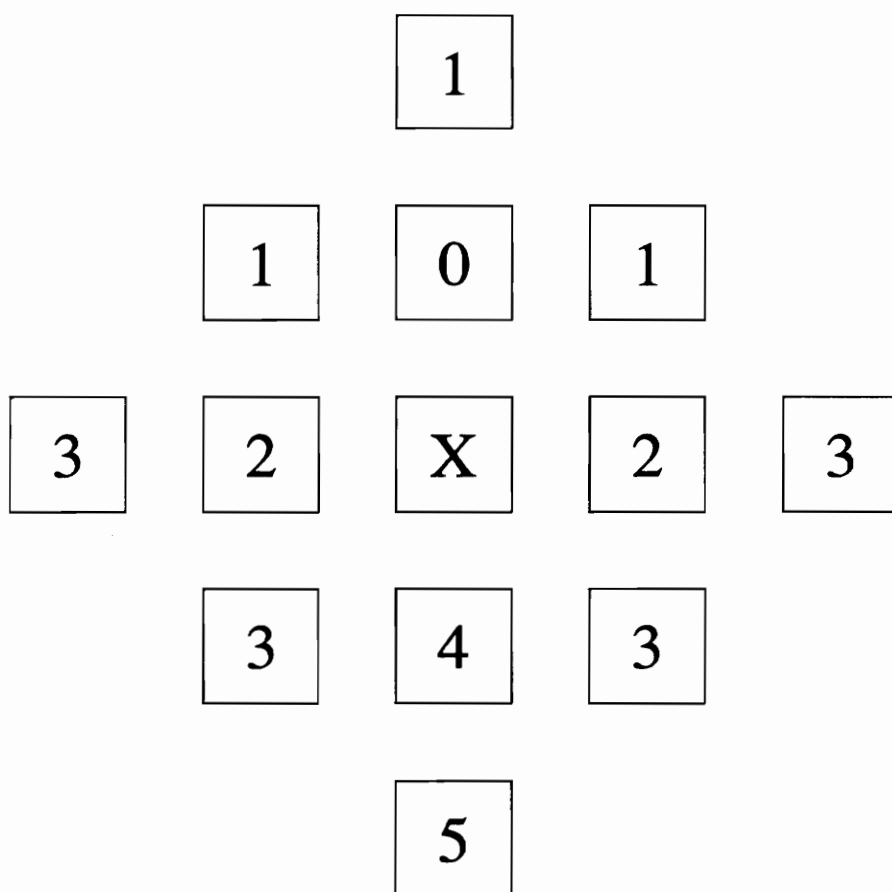


Figure 4.12 - Distributed diagnosis diagram using a White neighborhood for the interconnection network, and a von Neumann neighborhood for passing the fault data.

fault. Figure 4.13 shows the same example when a White neighborhood is used to distribute the fault data. Note that all cells in the region of awareness of X are notified of the fault on or before $t = 2$. Thus, the fault latency incurred from the use of a White neighborhood to distribute the fault data is much less than that required when a von Neumann neighborhood is used.

4.4 FATAL FAULT PATTERNS

Since the region of awareness is finite, all cells that can be made aware of a particular fault will become so within some finite time. The maximum time needed to accomplish this is given by the fault latency parameter, t_{MAX} , mentioned above. Intuitively, more complex fault patterns and larger regions of awareness increase the upper bound on t_{MAX} . The examples in Figures 4.12 and 4.13 illustrate the influence of the fault data distribution neighborhood on the fault latency parameter, since data may need to pass through intermediate neighbors to reach a particular cell.

It is important to recognize that some fault patterns may prevent fault data from reaching particular cells within the region of awareness. This occurs when the fault data distribution neighborhood is inadequate for bypassing a given fault pattern. For the von Neumann fault data distribution neighborhood, this is equivalent to finding fault patterns which divide the region of awareness into two or more disjoint regions. Other neighborhood configurations have similar limitations, since any sufficiently large fault pattern can be shown to be improperly detected. Such improperly detected faults are called *fatal faults*, since they are not tolerated by the existing array.

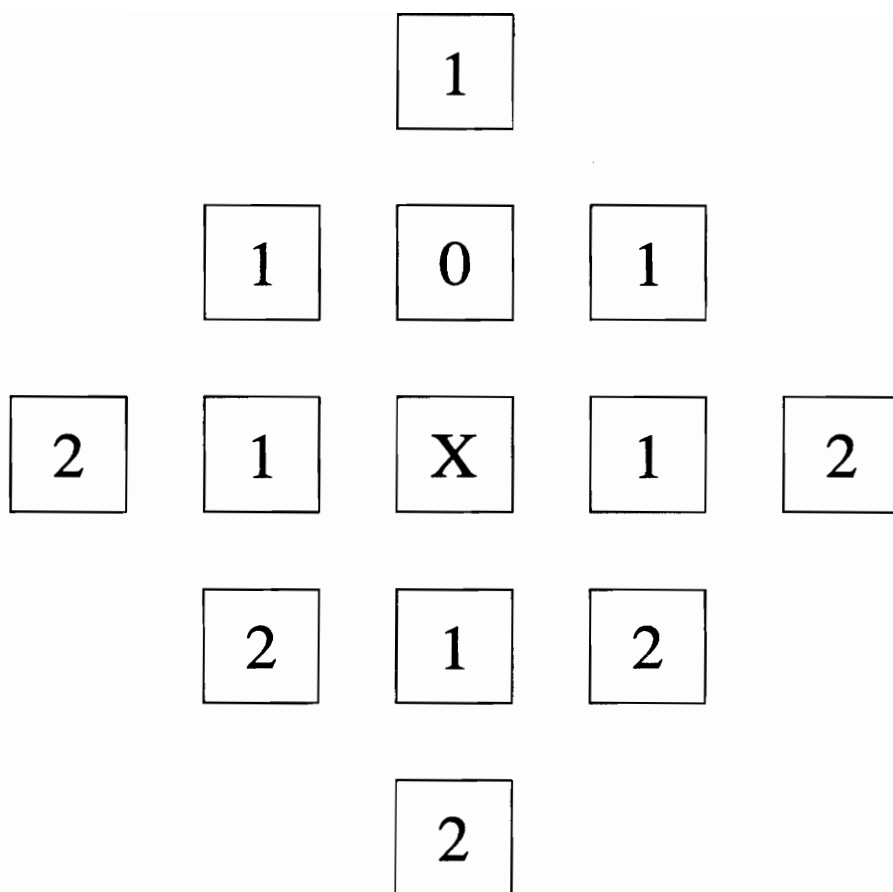
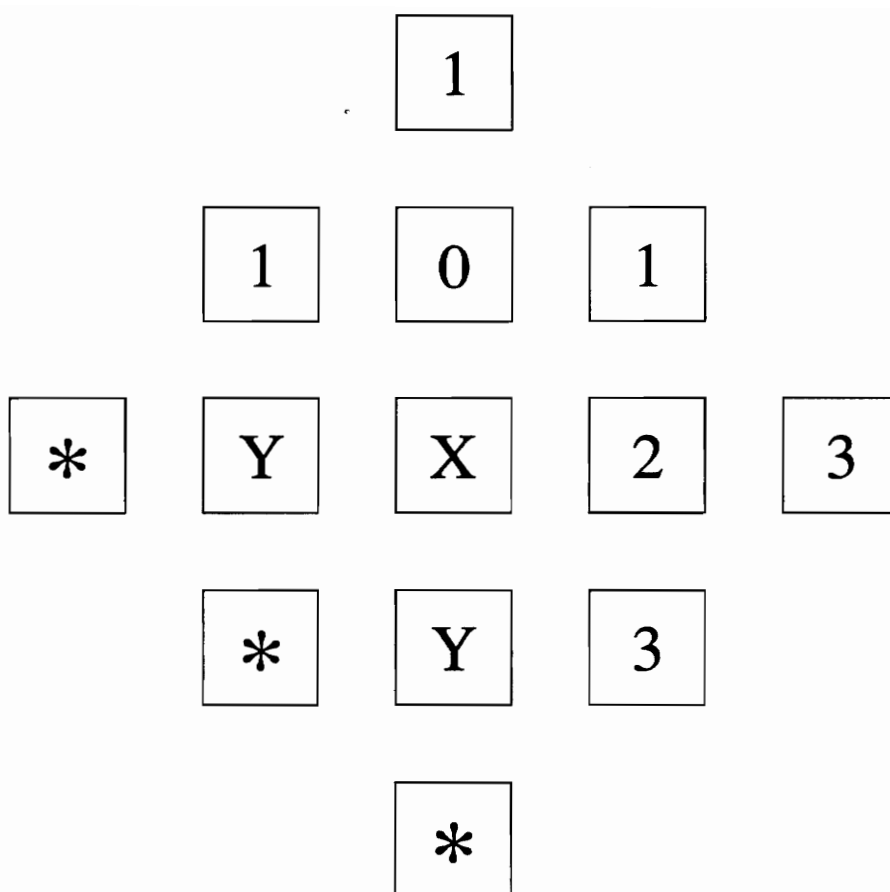


Figure 4.13 - Distributed diagnosis diagram using a White neighborhood for both the interconnection network and for passing the fault data.

4.4.1 THE L-FAULT

An example of a fatal fault is given in the triple fault pattern of Figure 4.14. Recall that this particular fault distribution is referred to as an L-fault pattern. The cell marked **X** defines the region of awareness for the example. The cells marked **Y** represent faulty cells within the region of awareness for **X**; these cells have their own regions of awareness, which are not relevant to this example. A von Neumann neighborhood is to be used for passing the fault data. Suppose that **X** fails at $t = 0$. Note that after $t = 3$, the cells marked ***** are still unaware of the fault at **X**. There is no way for the fault data to reach these cells, due to the faulty cells marked **Y**.

Figure 4.15 illustrates one solution to this particular problem. By expanding the region of awareness beyond the White neighborhood which was initially assumed, all L-fault patterns can be properly detected. The eight neighbors added to the region of awareness are, in clockwise order: North-Northeast (NNE), East-Northeast (ENE), East-Southeast (ESE), South-Southeast (SSE), South-Southwest (SSW), West-Southwest (WSW), West-Northwest (WNW), and North-Northwest (NNW). This modification corresponds to adding eight bits to the original twelve bits in the fault register, as well as the combinational logic needed to properly set these bits. The example of Figure 4.14 is repeated in Figure 4.16 to include the expanded fault register. Note that all the physical neighbors of **X** are aware of its faulty status on or before $t = 5$, and that all cells in the region of awareness of **X** are aware of the same on or before $t = 6$. This does not guarantee the same results for the faults marked **Y**. In order to completely diagnose the L-fault, additional time may be needed beyond $t = 6$. This depends on how each of the



Cells marked * are unable to learn about the fault at X, due to the faulty cells marked Y.

Figure 4.14 - Unsuccessful L-fault detection using a von Neumann neighborhood.

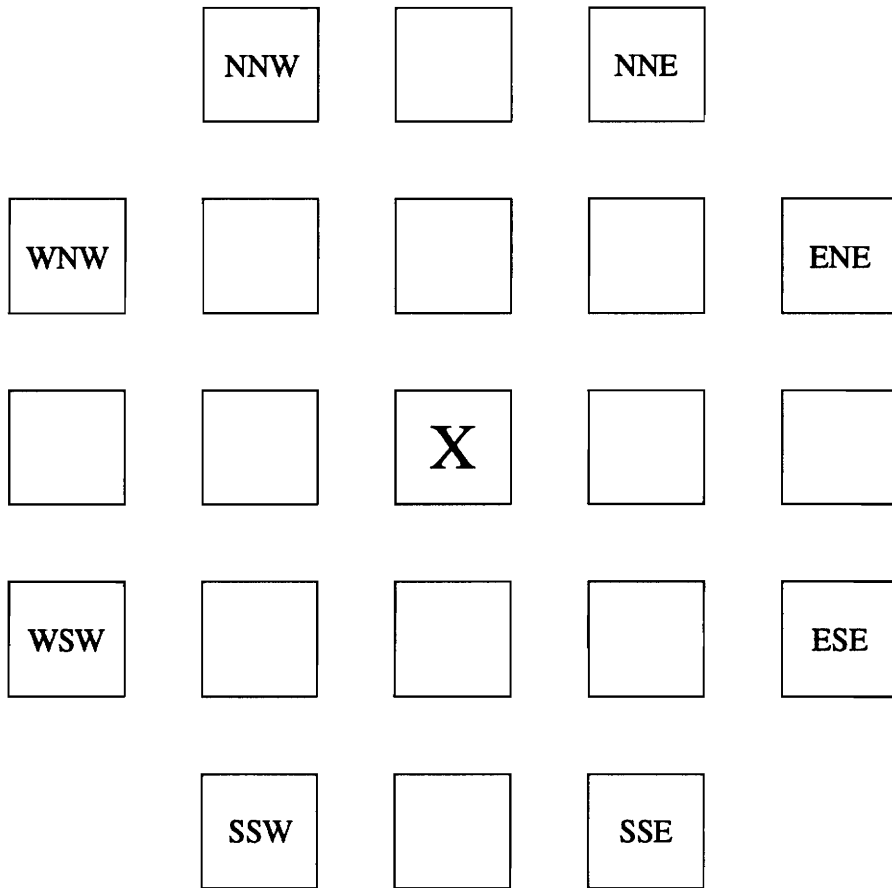


Figure 4.15 - Additional fault register bits needed to detect L-faults.

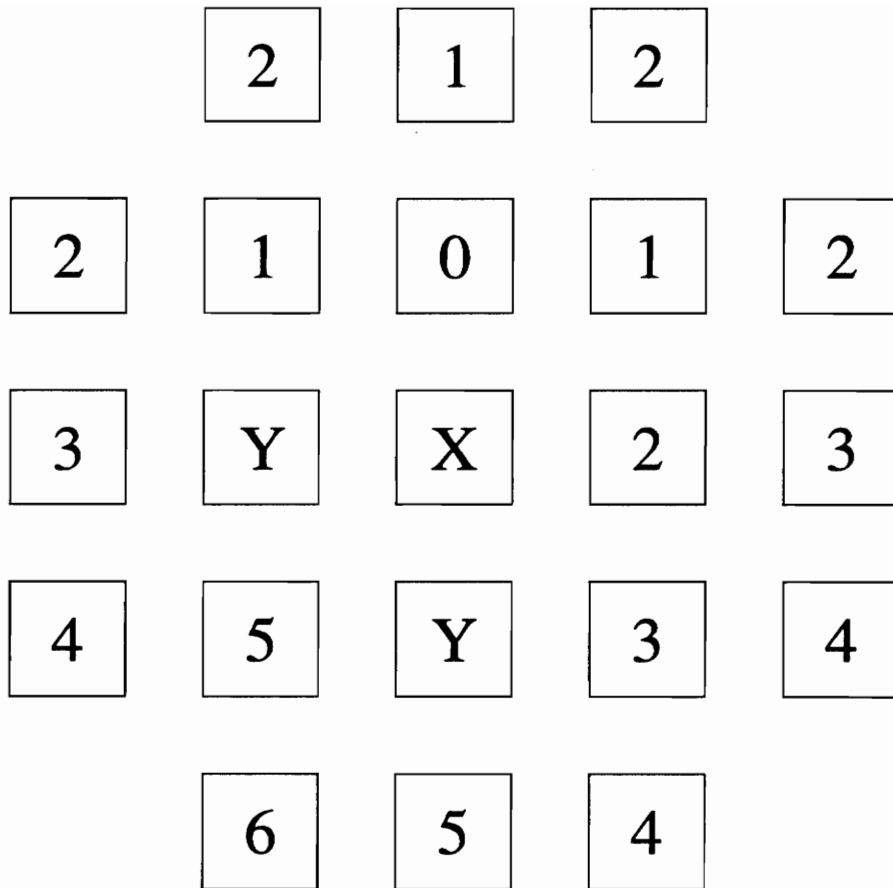


Figure 4.16 - Example of successful L-fault detection using the 20-bit fault register.

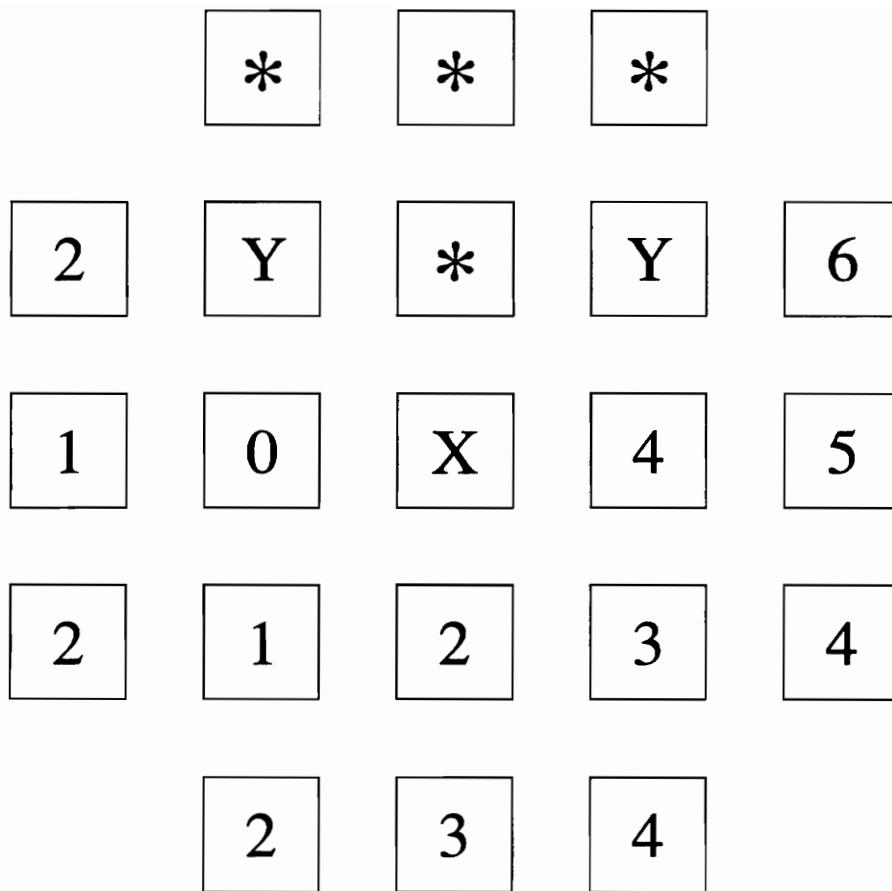
faults is initially detected, and which neighborhood is used to distribute the fault data. These issues are explored in more detail in section 4.5.

4.4.2 THE V-FAULT AND THE /-FAULT

Even with the expansion of the fault register as described above, two other triple faults can prove fatal. Figure 4.17 illustrates the *V-fault*, and Figure 4.18 gives an example of the */-fault* ("slash fault"). Both faults can be accommodated by again expanding the region of awareness to include the four additional cells indicated in Figure 4.19. Thus, four bits are added to the fault register, bringing the total number of fault register bits to twenty-four. The new bits correspond to the far diagonal neighbors: Far Northeast (FNE), Far Southeast (FSE), Far Southwest (FSW), and Far Northwest (FNW). Figures 4.20 and 4.21 show that both the V-fault and the /-fault are properly detected when the far diagonal neighbors are added.

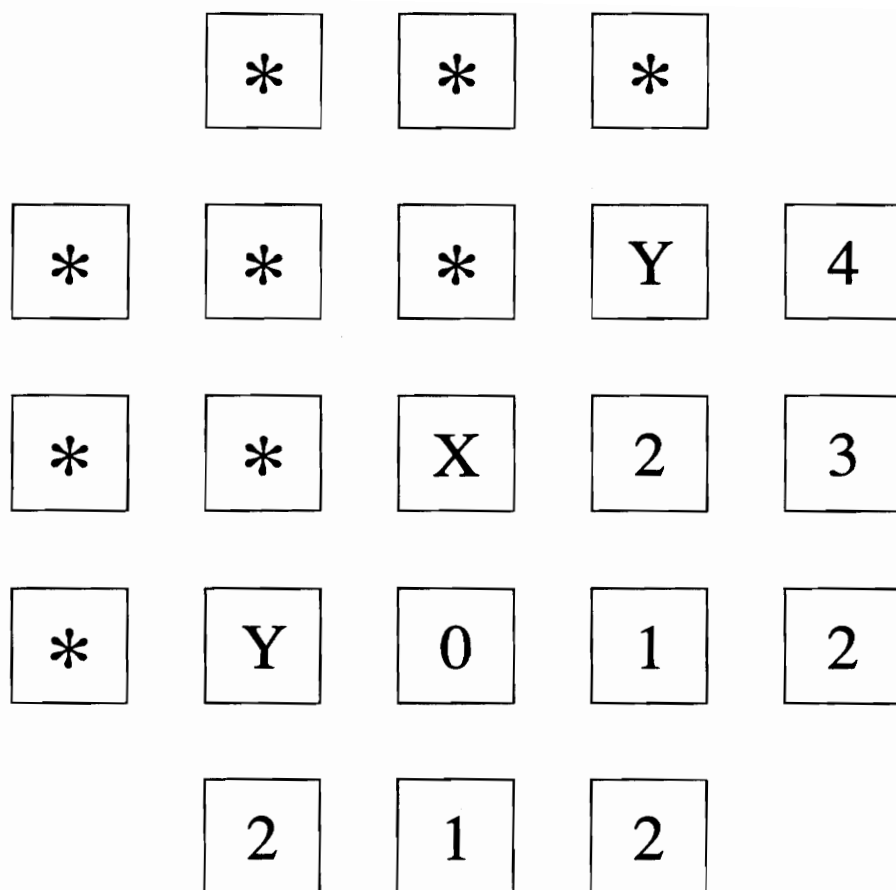
4.4.3 OTHER PROBLEMATIC TRIPLE FAULTS

Figure 4.22 shows that with the addition of the last four bits to the fault register, another triple fault can divide the region of awareness into disjoint regions. Thus, if a von Neumann neighborhood is used to pass data, the cell at * cannot be made aware of the fault at X. This type of situation is unavoidable, in general, regardless of how large the region of awareness is allowed to grow. However, the isolated cell, *, is not a physical neighbor of X. In other words, * can do nothing to directly influence the local reconfiguration needed to bypass X. In this case, the fault can actually be treated as a



Cells marked * are unable to learn about the fault at X, due to the faulty cells marked Y.

Figure 4.17 - Unsuccessful V-fault detection using a von Neumann neighborhood.



Cells marked * are unable to learn about the fault at X, due to the faulty cells marked Y.

Figure 4.18 - Unsuccessful /-fault detection using a von Neumann neighborhood.

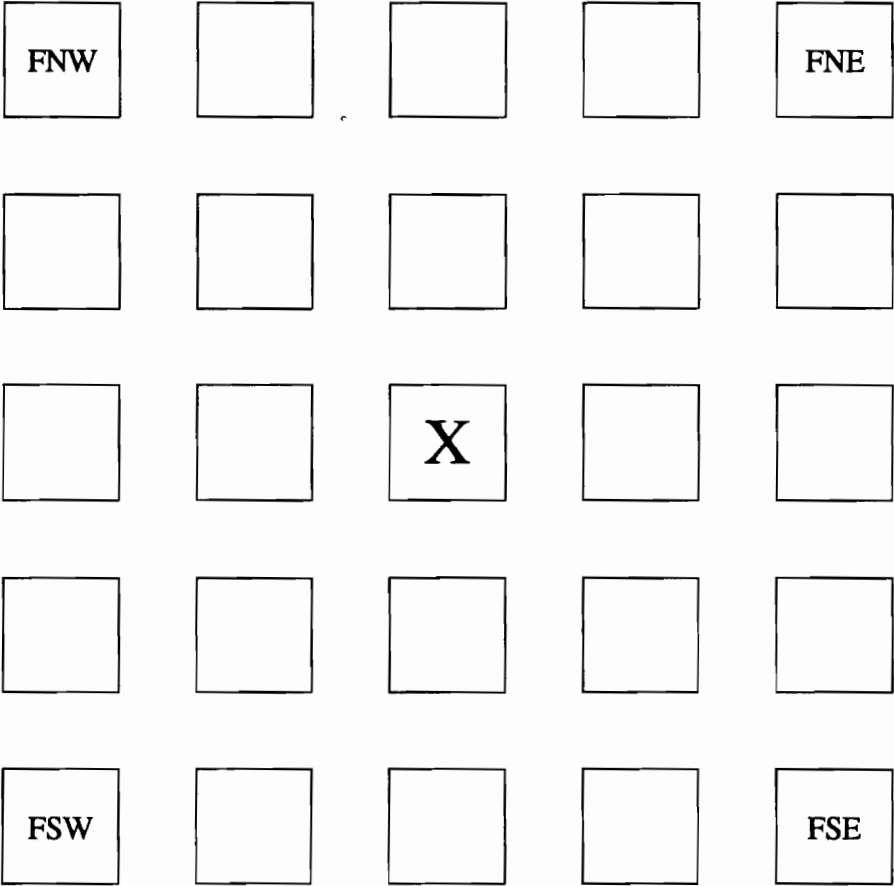


Figure 4.19 - Additional fault register bits needed to detect all triple faults.

3	4	5	6	7
2	Y	6	Y	6
1	0	X	4	5
2	1	2	3	4
3	2	3	4	5

Figure 4.20 - Successful detection of a V-fault using the 24-bit fault register.

7	8	7	6	5
6	7	8	Y	4
5	6	X	2	3
4	Y	0	1	2
3	2	1	2	3

Figure 4.21 - Successful detection of a /-fault using the 24-bit fault register.

5	4	3	2	3
4	3	2	1	2
5	4	X	0	1
4	3	2	1	Y
5	4	3	Y	*

Figure 4.22 - Triple fault which can be treated as a single fault and a double fault.

compound fault, consisting of the single fault at X, and the double fault indicated by the cells marked Y. Thus, * does not need to be aware of the fault at X at all. Fortunately, no other fatal triple faults exist. Therefore, by increasing the number of bits in the fault register from twelve to twenty-four, all triple faults can be properly detected.

4.5 DISTRIBUTED DIAGNOSIS SIMULATOR

With the fault register configuration defined, it is now possible to determine the fault latency parameter, t_{MAX} , for the fault patterns of interest. A software simulation of the distributed diagnosis algorithm is presented for determining t_{MAX} . The following discussion of the algorithm is general in nature, but serves to illustrate the basic concepts involved in performing the simulation. The C source code for the simulator is provided as Appendix A.

The calling syntax of the simulator is as follows:

```
array [v|ml|lw] filespec
```

where [v|ml|lw] indicates the neighborhood to be used to pass the fault data (von Neumann, Moore, Lawson, or White, respectively), and filespec is the specification for a text file which contains the locations of faults in the array. This fault specification file consists of one line of text per faulty cell, and two space-delimited numerals per line. The numerals indicate the row and column of the corresponding fault, respectively. The simulator uses a 10-by-10 physical array to investigate the distributed diagnosis problem, with row 0 being the topmost row, and column 0 being the leftmost column. Once the

faults are placed in the array, the simulation is ready to begin, and the simulator makes some assumptions to guarantee a worst-case scenario.

It is assumed that each faulty cell is detected by only one neighbor, rather than a general cell failure in which the faulty cell is simultaneously detected by two or more neighbors. This single detection model corresponds to a single link failure, which represents the most challenging fault detection problem. Since an upper bound on t_{MAX} is required, this worst-case assumption provides the desired results.

It is further assumed that faulty cells do not send valid fault data to any neighbors. Note that in the case of bidirectional link faults, it is entirely possible for a cell which is otherwise fault-free to detect the faulty link as if the adjoining neighbor is faulty. It is therefore perfectly reasonable in this case to assume the cell can tell its other neighbors of the problem. Similarly, the cell on the other end of the same link sees the first cell as faulty, and likewise broadcasts this information to its own neighbors. Hence, a bidirectional link fault appears as a double fault to the cells in the appropriate regions of awareness. Nevertheless, the simulator assumes faulty cells do nothing to improve the distribution of fault data. In practice, multiple detection and partly functional circuits can occur, and in any case would only decrease the amount of time needed to distribute the necessary data.

Each fault pattern is simulated once for each possible combination of detecting cells. This insures that the worst-case time needed to distribute the fault data is determined. At each step of the simulation, each cell examines the contents of the fault registers of its fault-free neighbors, and updates the appropriate bits in its own fault

register accordingly. This process is accomplished via a simple table lookup scheme, which could be implemented in hardware using combinational logic, or by using a memory table similar to that used for the pattern growth process. The simulation terminates when all fault-free physical neighbors of faulty cells are aware of the faults, or when the state of the array does not change from one iteration to the next. The latter condition indicates a failure to properly diagnose the fault. At the end of the simulation run, the number of simulations is given, along with the maximum number of iterations needed to distribute the fault data to all physical neighbors of the faulty cells.

Note that the time needed to inform all cells in the region of awareness of a faulty cell is sometimes greater than the time needed to inform only the physical neighbors of the faulty cell. However, the difference is at most two time steps. Consider the case in which all of the physical neighbors of a faulty cell X are aware of the fault at time $t = 0$. If a von Neumann neighborhood is used to pass the fault data, the region of awareness defined by the 24-bit fault register is covered at time $t = 2$. Under the worst-case simulation cases, the region of awareness is covered in at most one time step after all of the physical neighbors are covered. This is due to the fact that during the time needed to inform the physical neighbors of the fault, some or all of the additional cells in the region of awareness are informed as well. Waiting the additional time step can be advantageous in a local reconfiguration scenario. Even though cells which are not physical neighbors of a faulty cell do not directly take part in locally bypassing a faulty cell, they could conceivably use the knowledge of the fault to change their own interconnections so that a spare could take part in the reconfiguration process. Thus, the given value of t_{MAX} will include the additional time step, if needed.

4.5.1 SINGLE FAULT DIAGNOSIS

Although there is really only one single fault case, the simulator treats each detection possibility as a separate case. Figure 4.23 shows the region of awareness surrounding the faulty cell, X. Since the faulty cell has four neighbors, each of which is the potential detecting cell, four simulations are performed. Also given in the figure are the maximum times needed to fully distribute the fault data. When two numbers are given, the first indicates the time needed to notify all of the physical neighbors of X, and the second indicates the time needed to notify all cells in the region of awareness.

4.5.2 DOUBLE FAULT DIAGNOSIS

There are 2 double faults of interest, as shown in Figures 4.24 and 4.25. The faulty cell marked X defines the area of awareness shown. Other faulty cells are marked Y. Although the von Neumann case is shown here, all four neighborhoods are simulated. Both of these patterns are tightly clustered, in that the minimum radial distance from any given fault to each of the other faults in the pattern is equal to one. Other double fault patterns are possible, but once any faulty cell achieves a minimum radial distance of two or more from the other faults in a pattern, the maximum time needed to properly diagnose the fault is no greater than the maximum time needed to diagnose the tightly clustered fault patterns. This is verified via exhaustive simulation of up to quadruple faults.

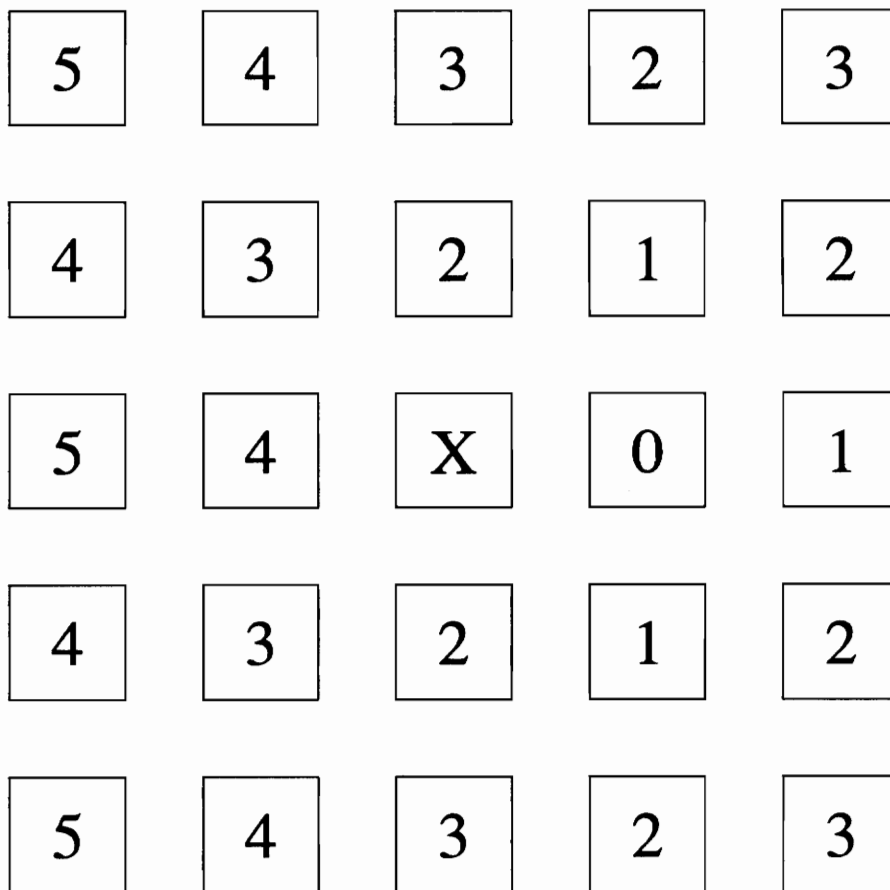


Figure 4.23 - Single fault detection using a von Neumann neighborhood.

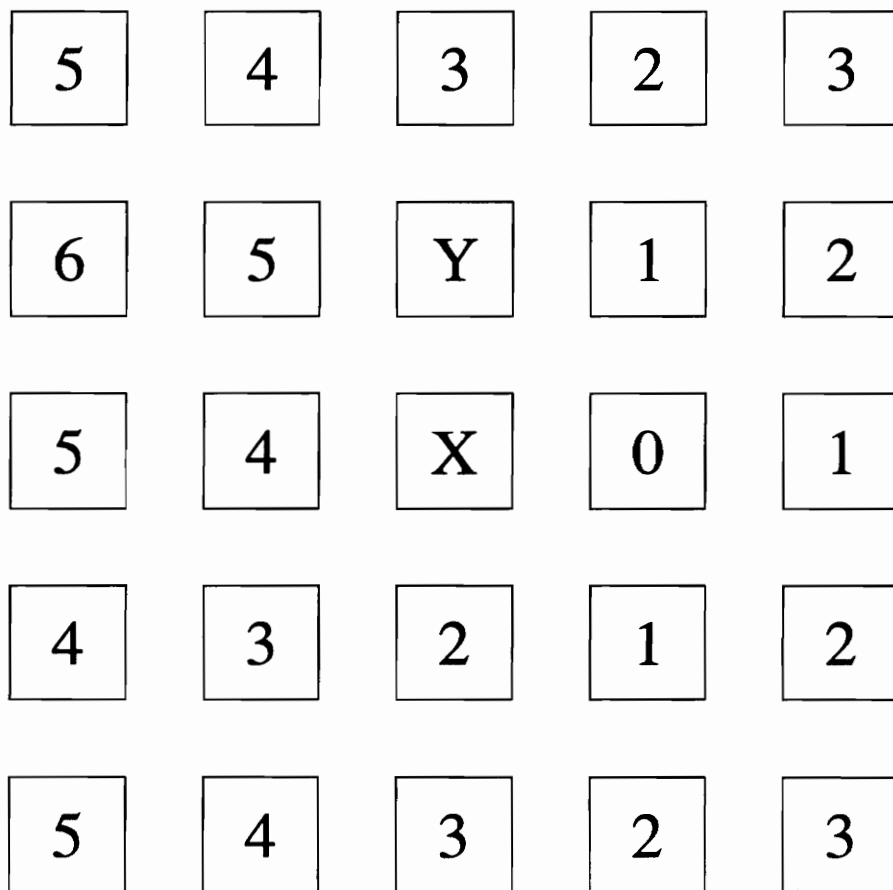


Figure 4.24 - Double fault detection using a von Neumann neighborhood.

(Case 1)

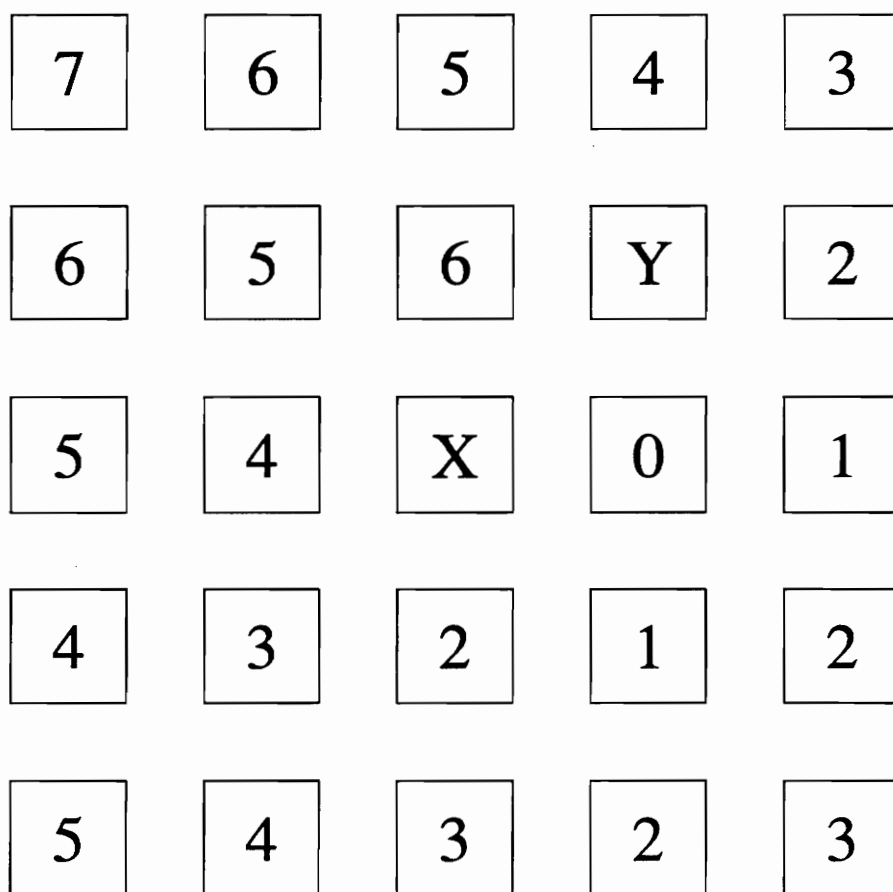


Figure 4.25 - Double fault detection using a von Neumann neighborhood.

(Case 2)

4.5.3 TRIPLE FAULT DIAGNOSIS

All triple fault patterns have been simulated. The only triple fault patterns presented here are the tightly clustered cases, as given in Figures 4.26 through 4.30. These and subsequent fault pattern diagrams indicate only the faulty cells, and omit the fault detection times given in the previous diagrams. Maximum fault detection times are summarized at the end of the chapter for each of the four neighborhoods.

4.5.4 QUADRUPLE FAULT DIAGNOSIS

The von Neumann neighborhood is unable to properly detect eight quadruple fault patterns, which are given in Figures 4.31 through 4.38. Each of these fatal faults divides the region of awareness into two or more disjoint regions. Note that expanding the region of awareness has no effect on diagnosing the fault of Figure 4.31, since there is no means for passing fault data into or out of the cell in the interior region defined by the faulty cells.

The Moore neighborhood is incapable of properly detecting the fault of Figure 4.38. Joseph Wegner, a researcher at Virginia Tech who is currently investigating the local reconfiguration algorithm, has determined that the fault pattern of Figure 4.38 cannot be successfully handled by the local reconfiguration hardware, even if proper detection is possible. In other words, no means currently exist to successfully bypass this particular fault pattern with the existing physical neighborhood. There are 2 additional quadruple faults which cannot be tolerated by the current local reconfiguration hardware. These patterns are given as Figures 4.39 and 4.40.

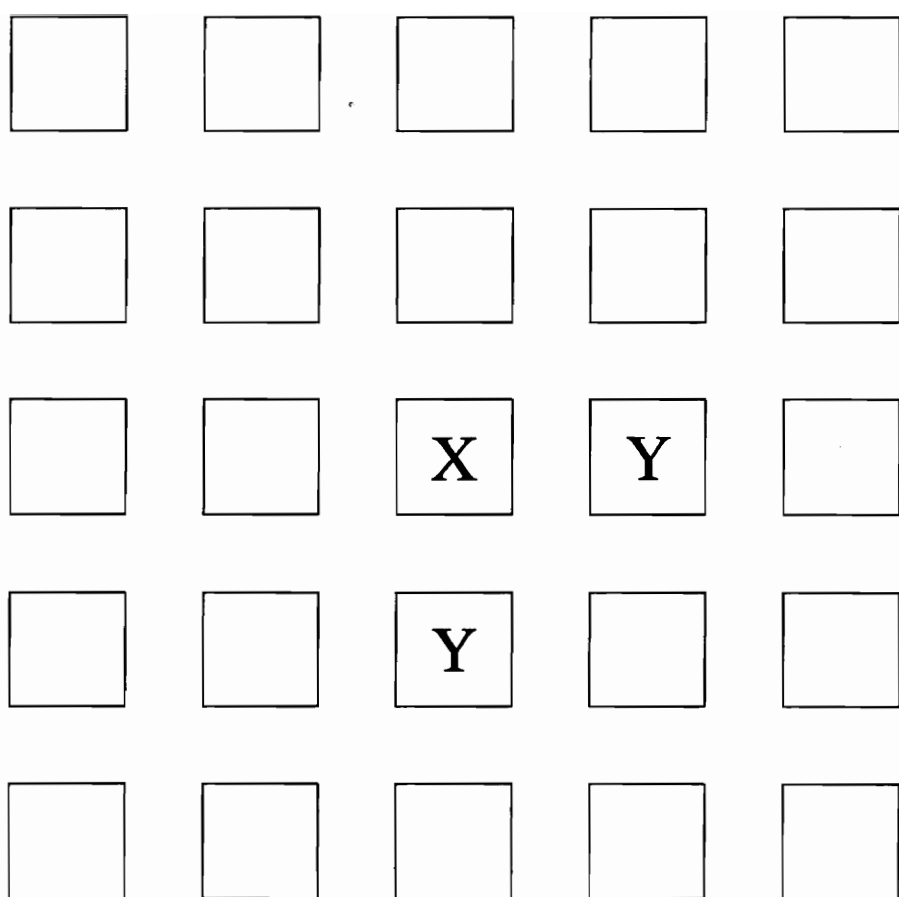


Figure 4.26 - Tightly-clustered triple fault pattern.

(Case 1)

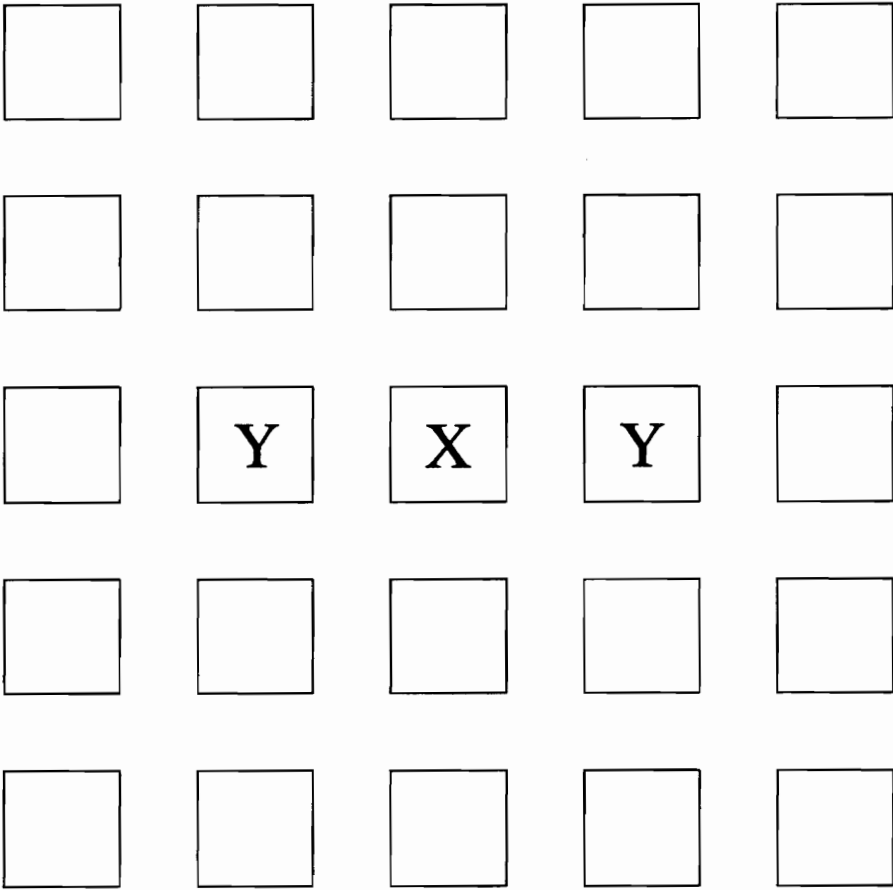


Figure 4.27 - Tightly-clustered triple fault pattern.
(Case 2)

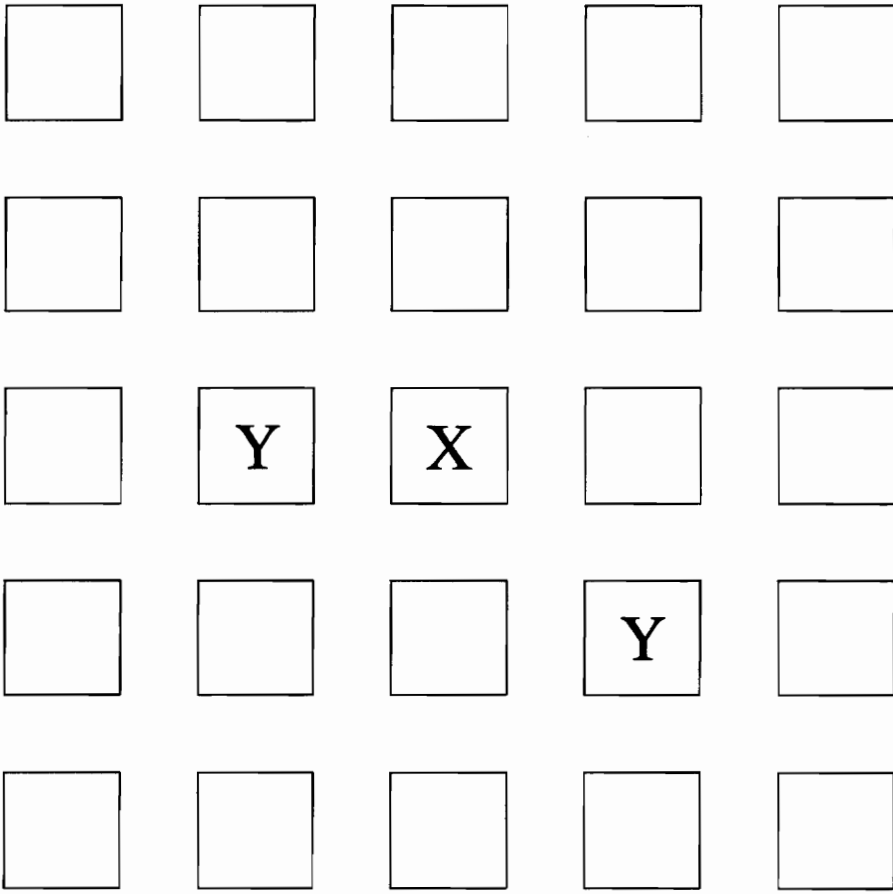


Figure 4.28 - Tightly-clustered triple fault pattern.
(Case 3)

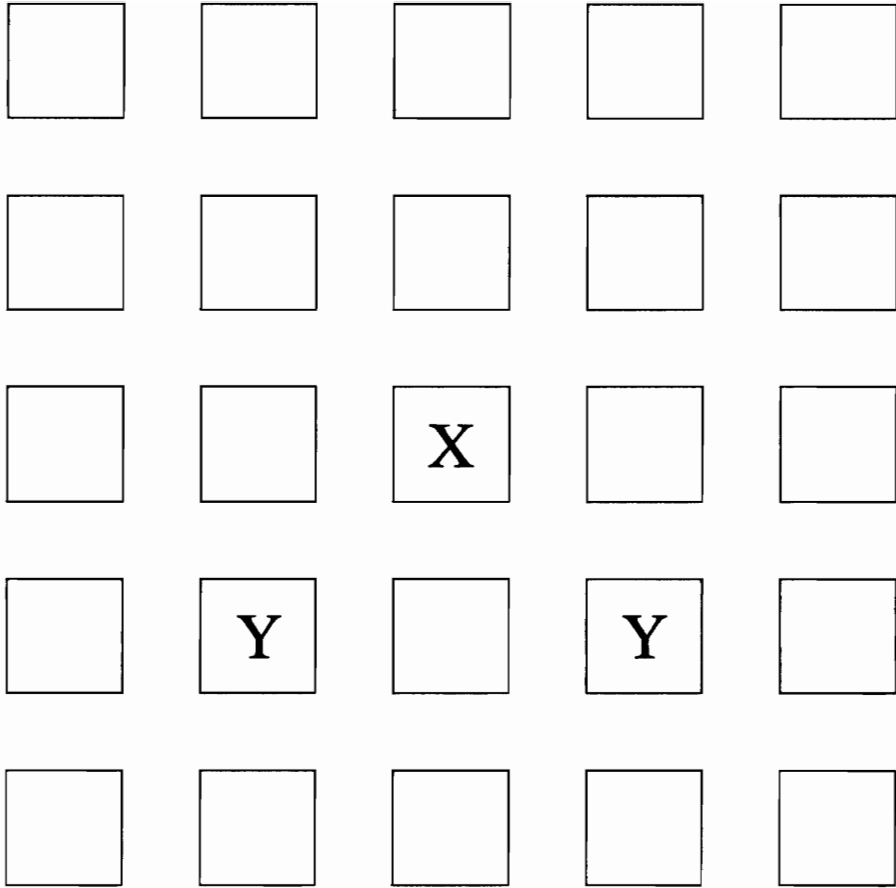


Figure 4.29 - Tightly-clustered triple fault pattern.
(Case 4)

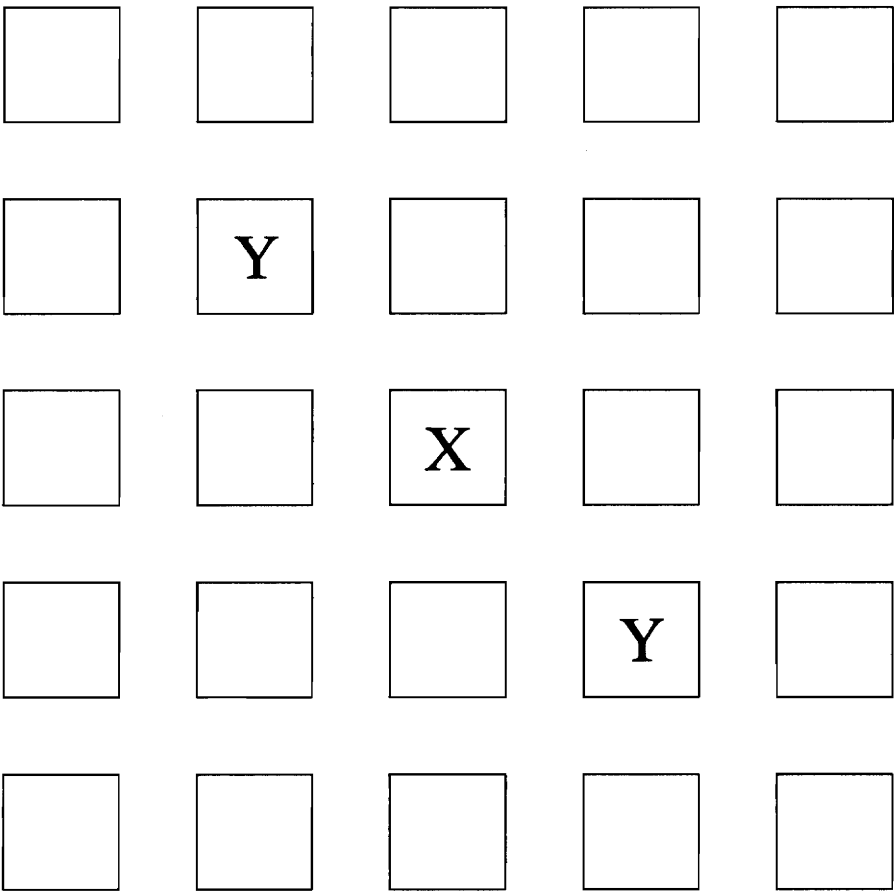


Figure 4.30 - Tightly-clustered triple fault pattern.
(Case 5)

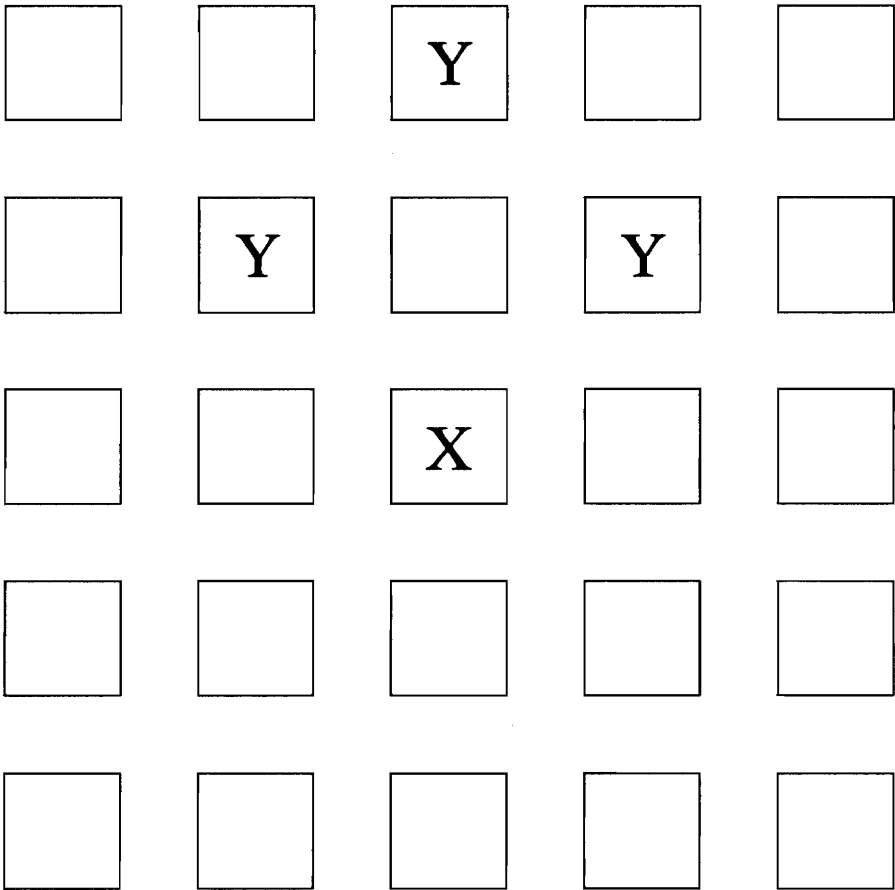


Figure 4.31 - Fatal quadruple fault pattern using a von Neumann neighborhood.
(Case 1)

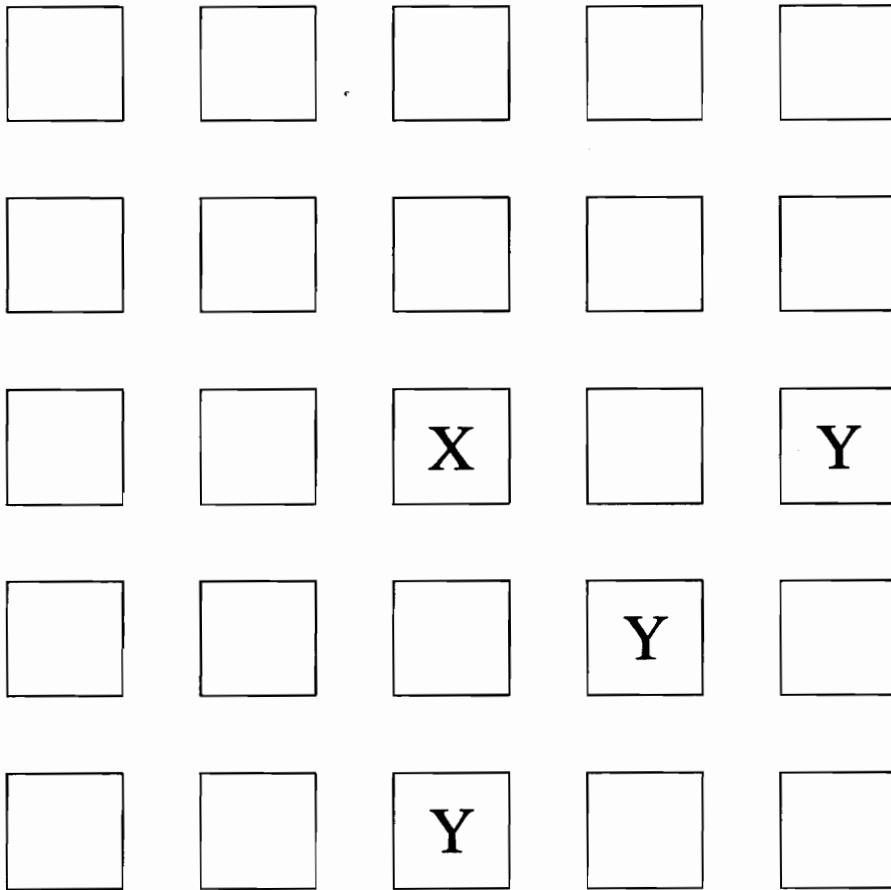


Figure 4.32 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 2)

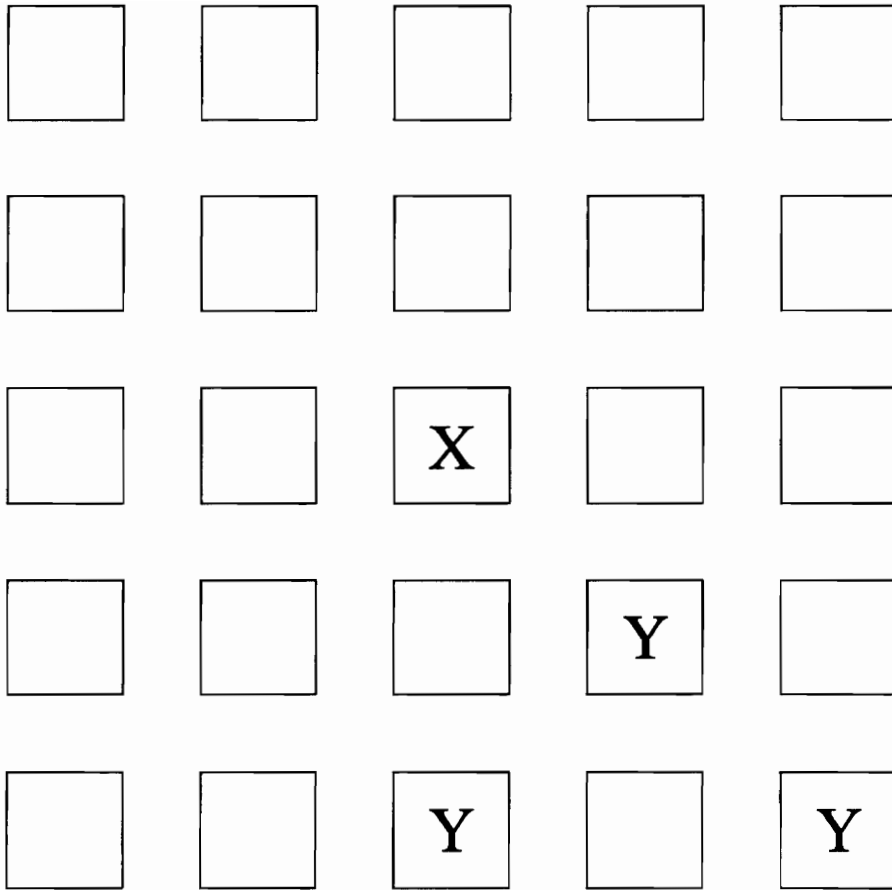


Figure 4.33 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 3)

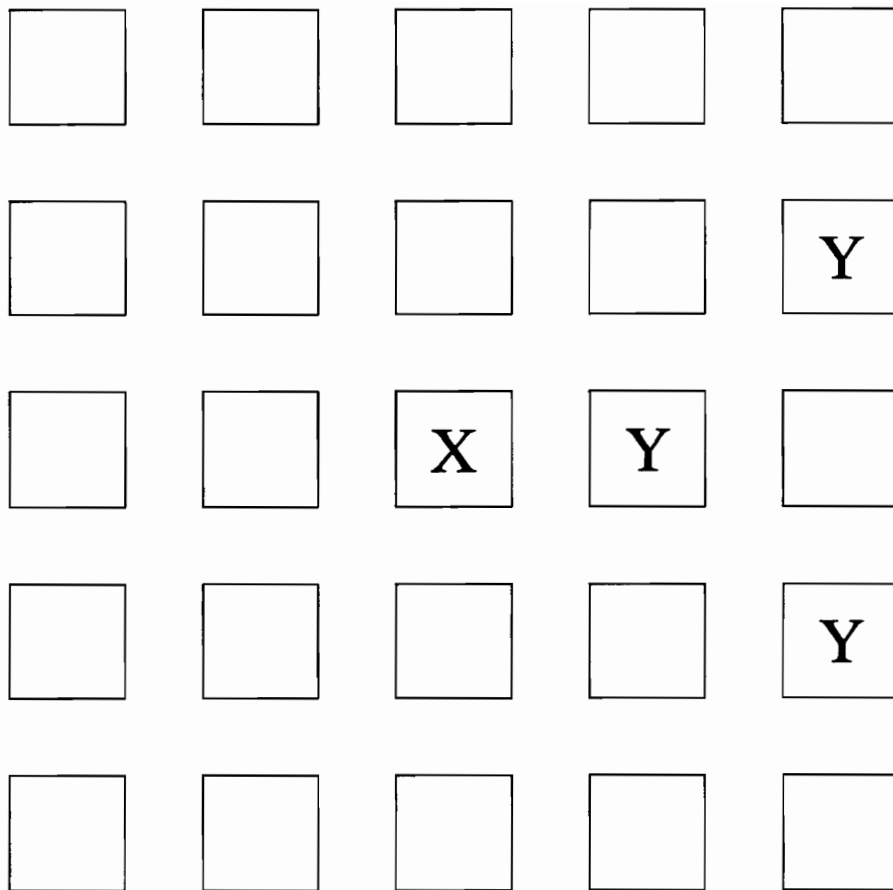


Figure 4.34 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 4)

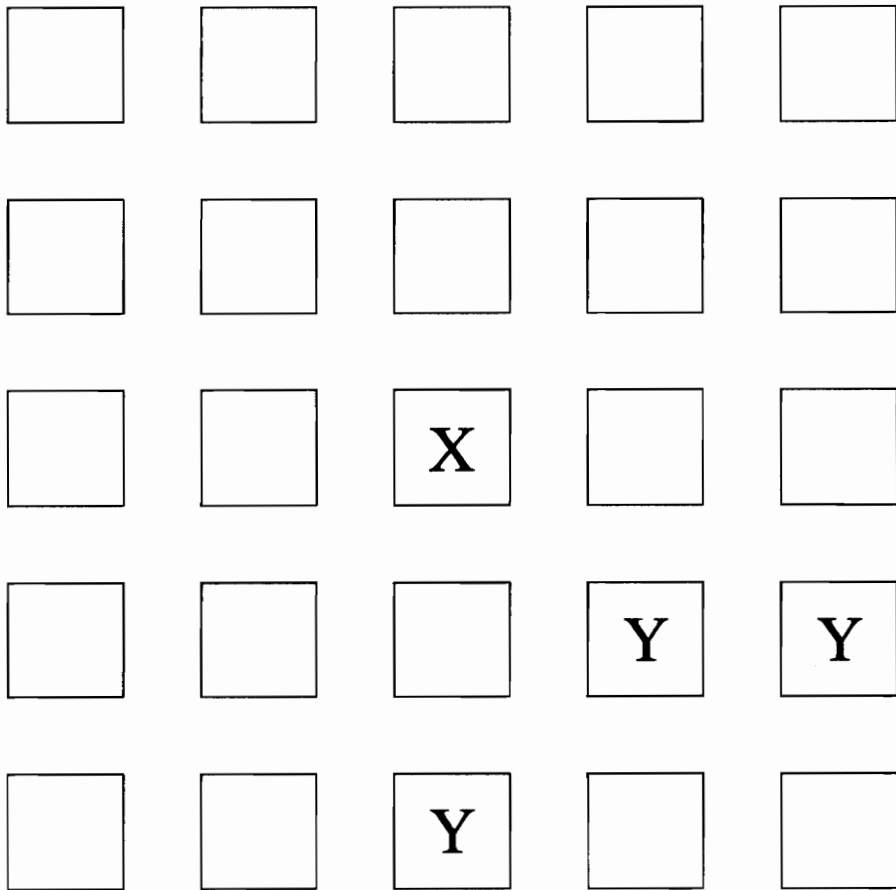


Figure 4.35 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 5)

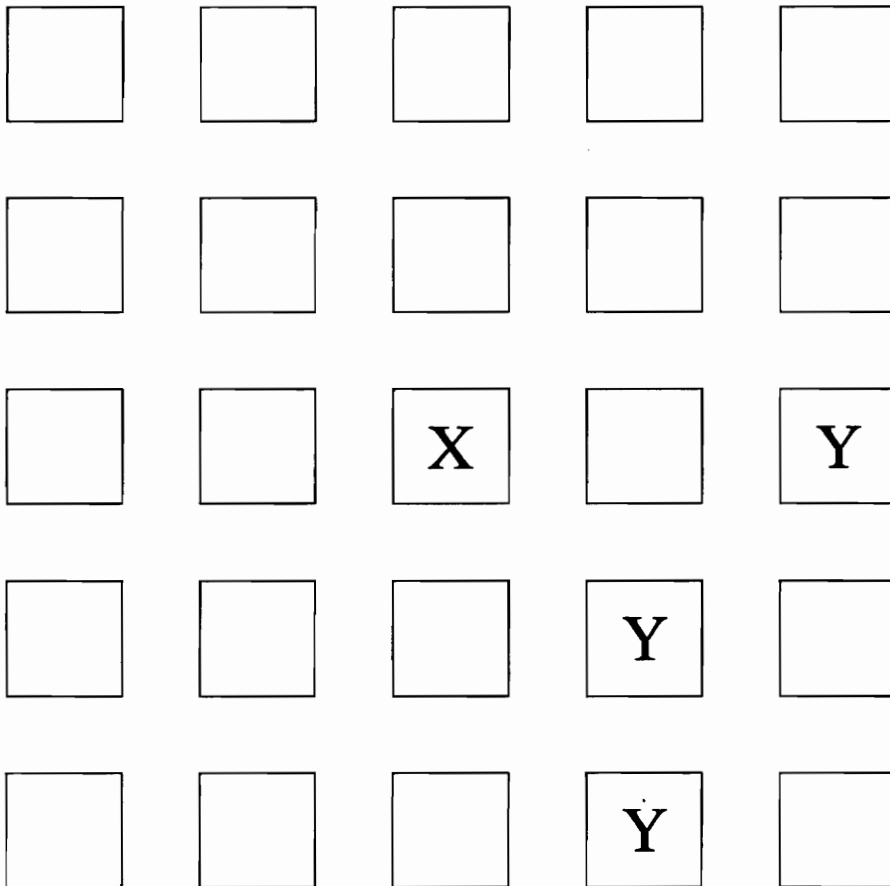


Figure 4.36 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 6)

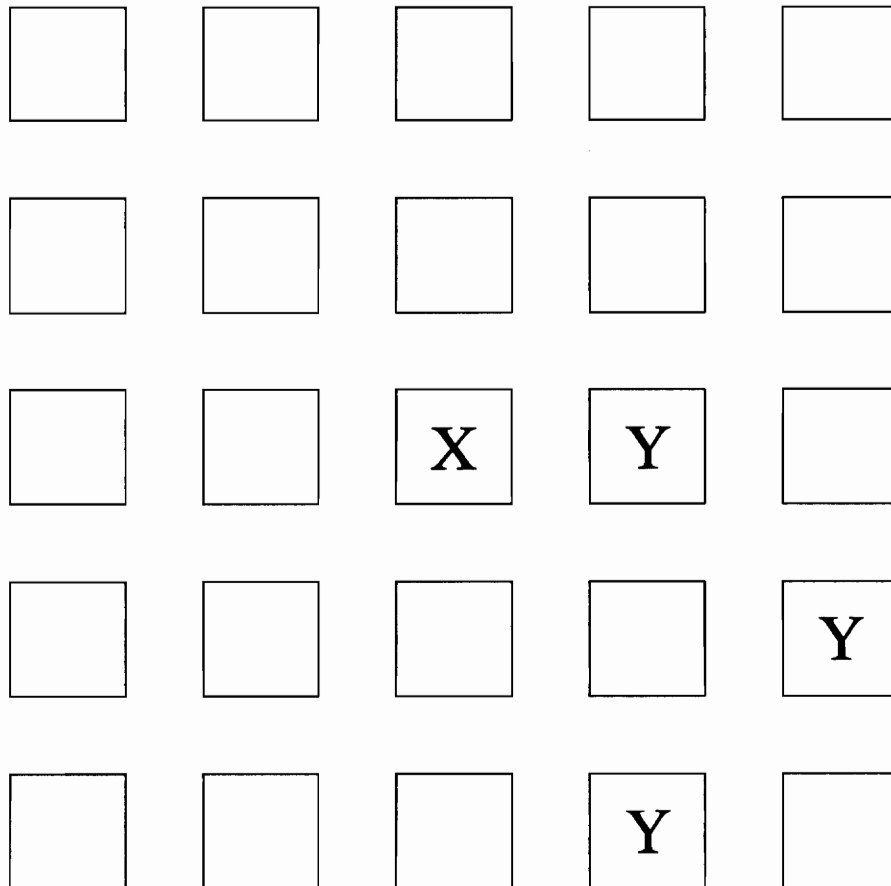


Figure 4.37 - Fatal quadruple fault pattern using a von Neumann neighborhood.

(Case 7)

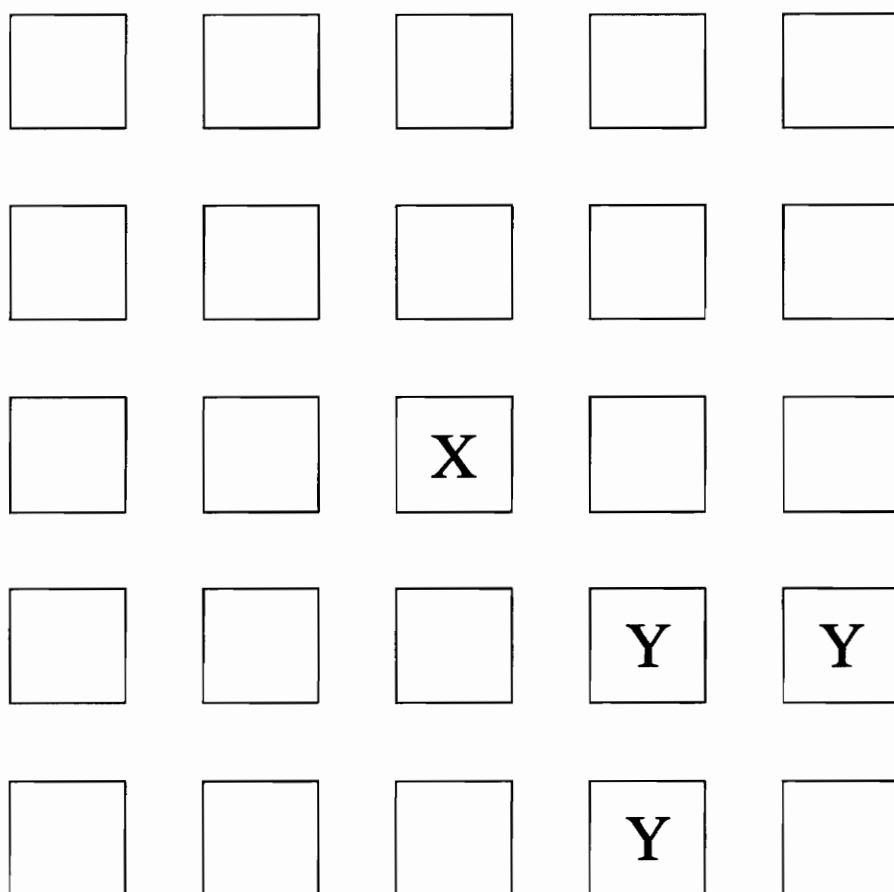


Figure 4.38 - Fatal quadruple fault pattern using von Neumann or Moore neighborhoods.

(Case 8)

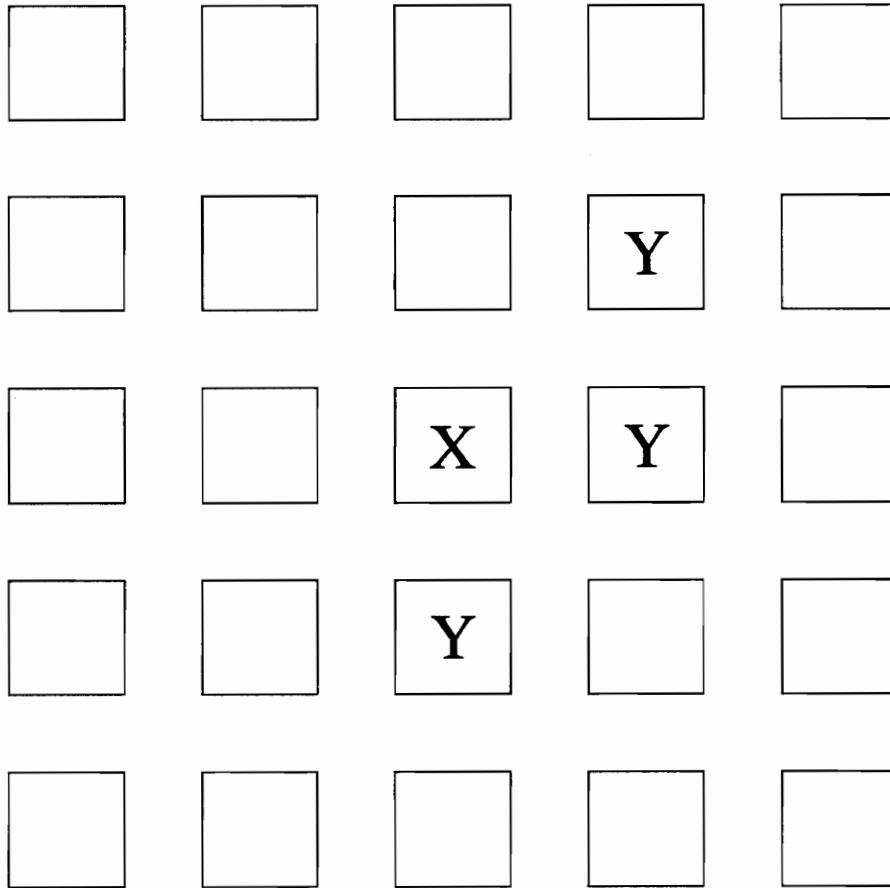


Figure 4.39 - Fatal quadruple fault pattern using current local reconfiguration hardware.

(Case 1)

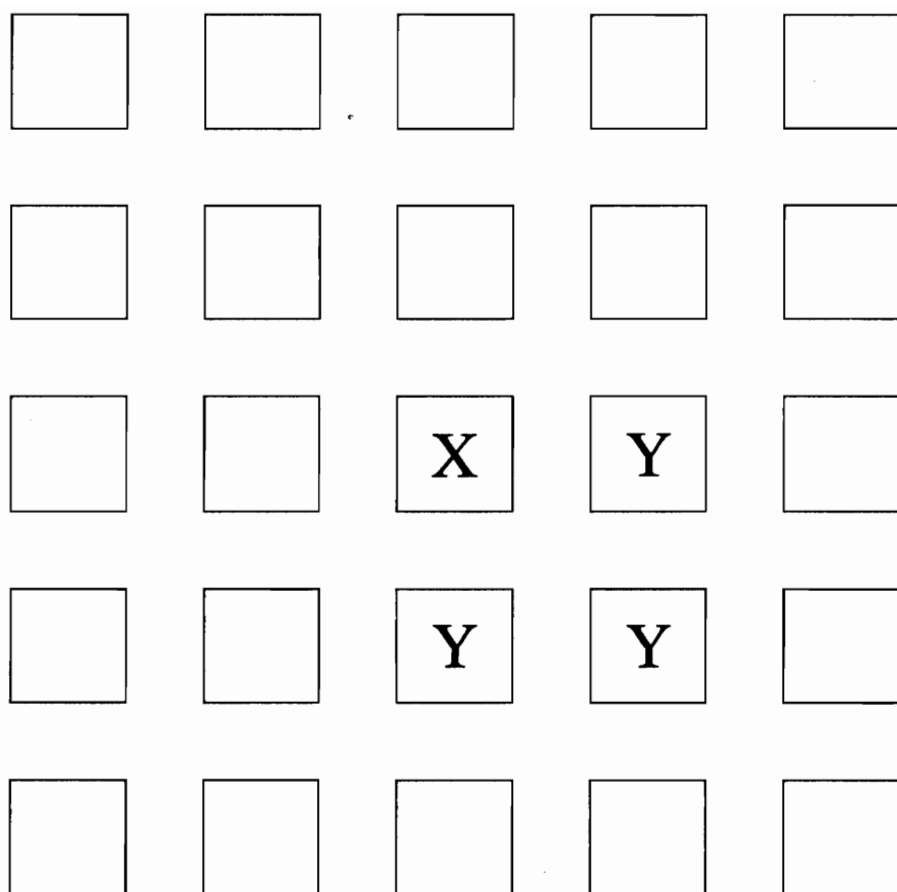


Figure 4.40 - Fatal quadruple fault pattern using current local reconfiguration hardware.

(Case 2)

After eliminating the above quadruple fault patterns, there are 16 remaining tightly clustered quadruple fault patterns, as shown in Figures 4.41 through 4.56. It should be noted that both the Lawson and White neighborhoods can properly detect all quadruple faults, including those not currently tolerated by the local reconfiguration scheme.

4.5.5 DIAGNOSIS OF 5 OR MORE FAULTS

Many higher order faults can be viewed as compound faults of up to quadruple faults, for the purposes of fault detection, if the simpler fault patterns are sufficiently separated from one another. The above results still hold in those cases. However, many higher order faults are fatal, and can prevent proper detection by prohibiting the passing of fault data to the necessary cells. These problems can be addressed by adding more bits to the fault register, and thus expanding the region of awareness, or by using an appropriate neighborhood, or a combination of these techniques. As for the lower order cases, the more tightly clustered faults will tend to determine the time needed for proper detection.

4.6 RECONFIGURATION USING DISTRIBUTED DIAGNOSIS

Once t_{MAX} is determined, it is possible to suggest a means to incorporate the distributed diagnosis algorithm into the existing local reconfiguration scheme. If a cell becomes aware of a new fault, it must wait t_{MAX} time steps before reconfiguring its communications links to bypass the fault. If multiple faults occur simultaneously in the cell's region of awareness, waiting t_{MAX} time steps insures that all quadruple faults are properly detected, provided that a Lawson or a White neighborhood is used for passing the fault information. Note that in a worst-case scenario, the last cell becomes aware of

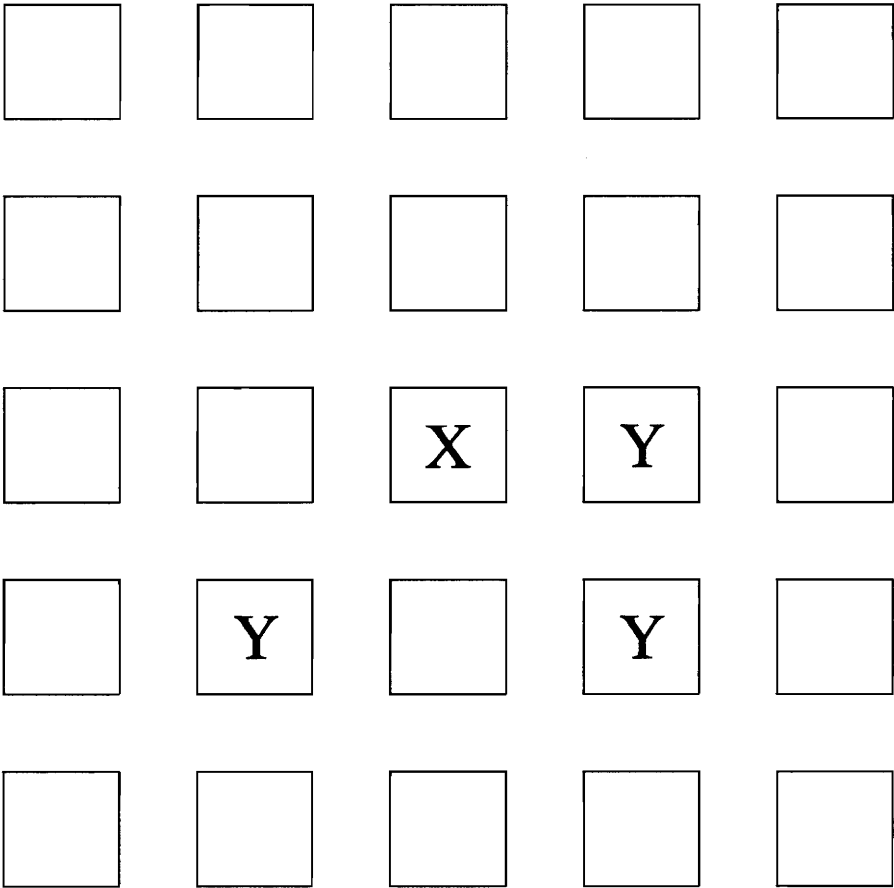


Figure 4.41 - Properly detected, tightly clustered quadruple fault pattern.
(Case 1)

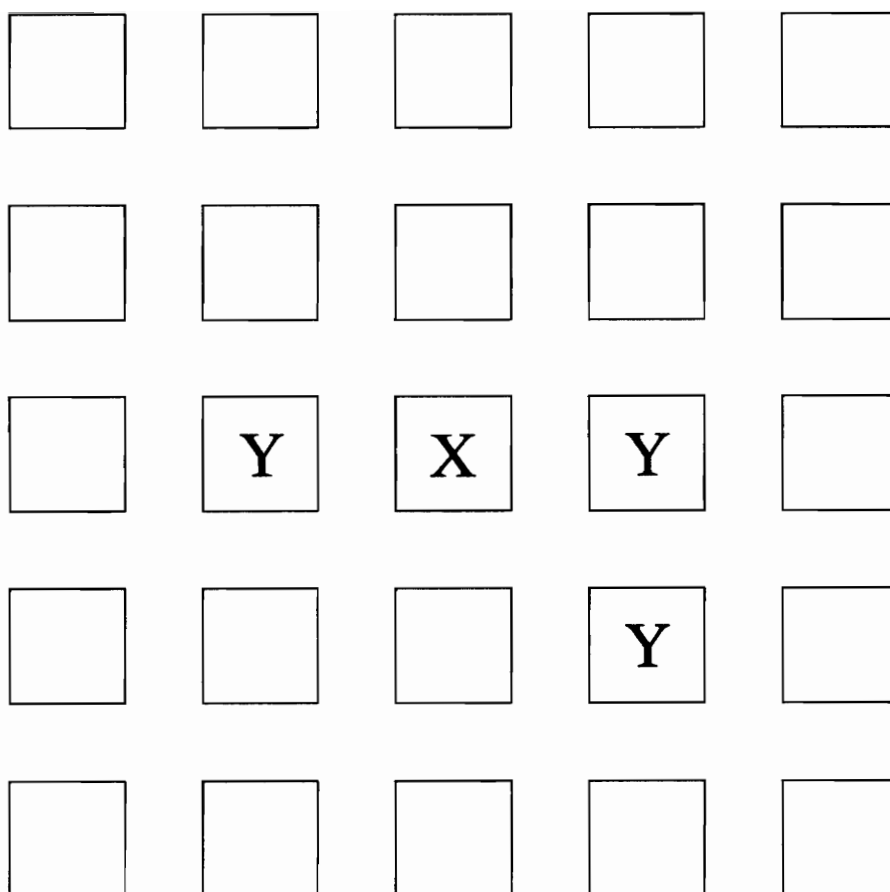


Figure 4.42 - Properly detected, tightly clustered quadruple fault pattern.

(Case 2)

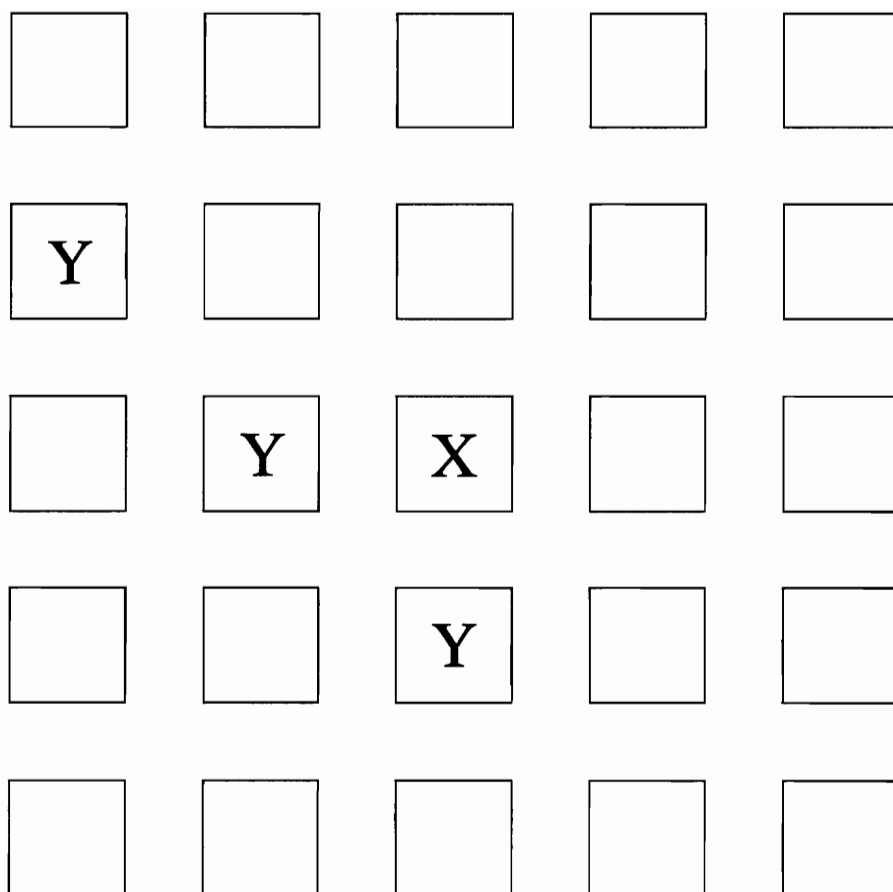


Figure 4.43 - Properly detected, tightly clustered quadruple fault pattern.

(Case 3)

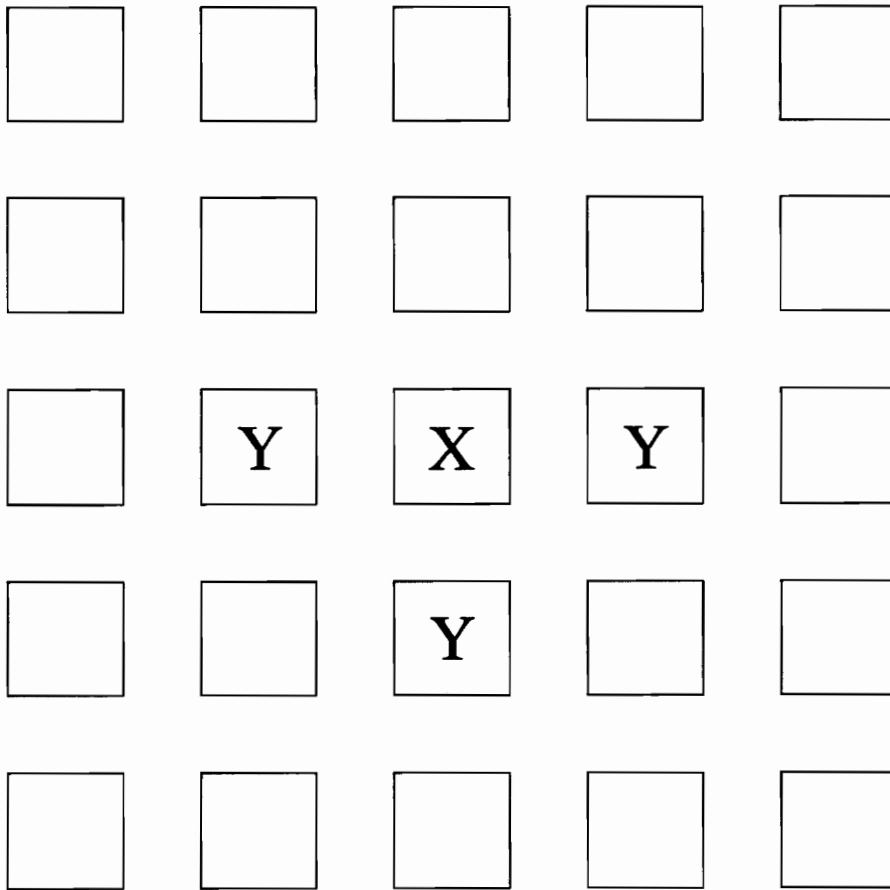


Figure 4.44 - Properly detected, tightly clustered quadruple fault pattern.
(Case 4)

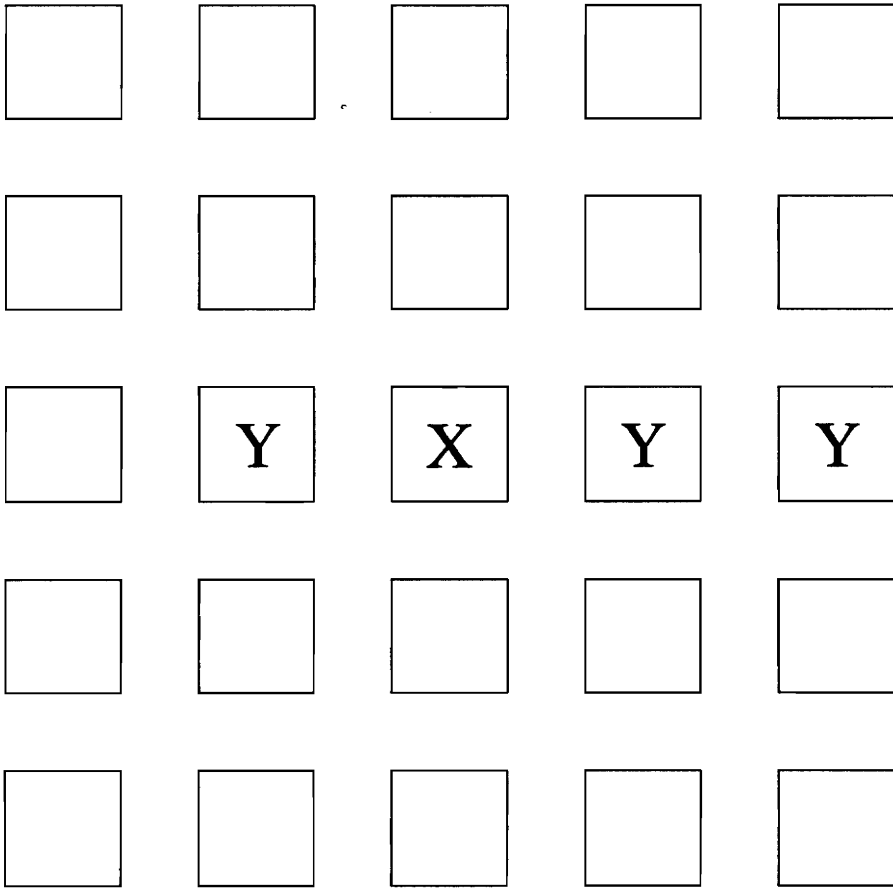


Figure 4.45 - Properly detected, tightly clustered quadruple fault pattern.
(Case 5)

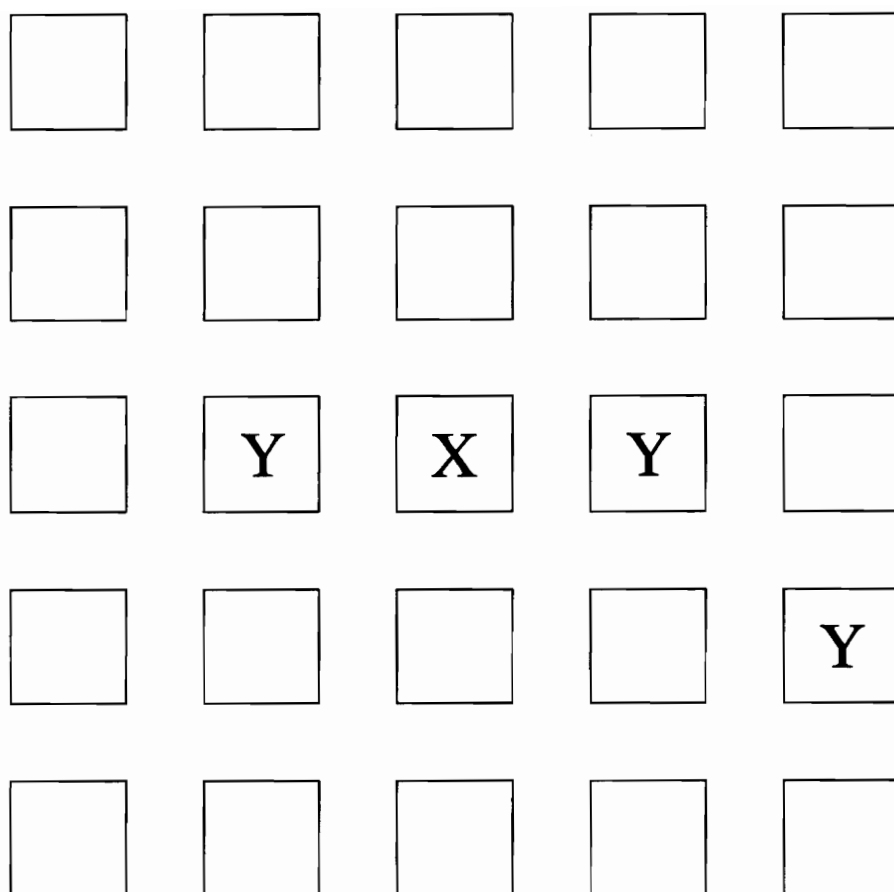


Figure 4.46 - Properly detected, tightly clustered quadruple fault pattern.

(Case 6)

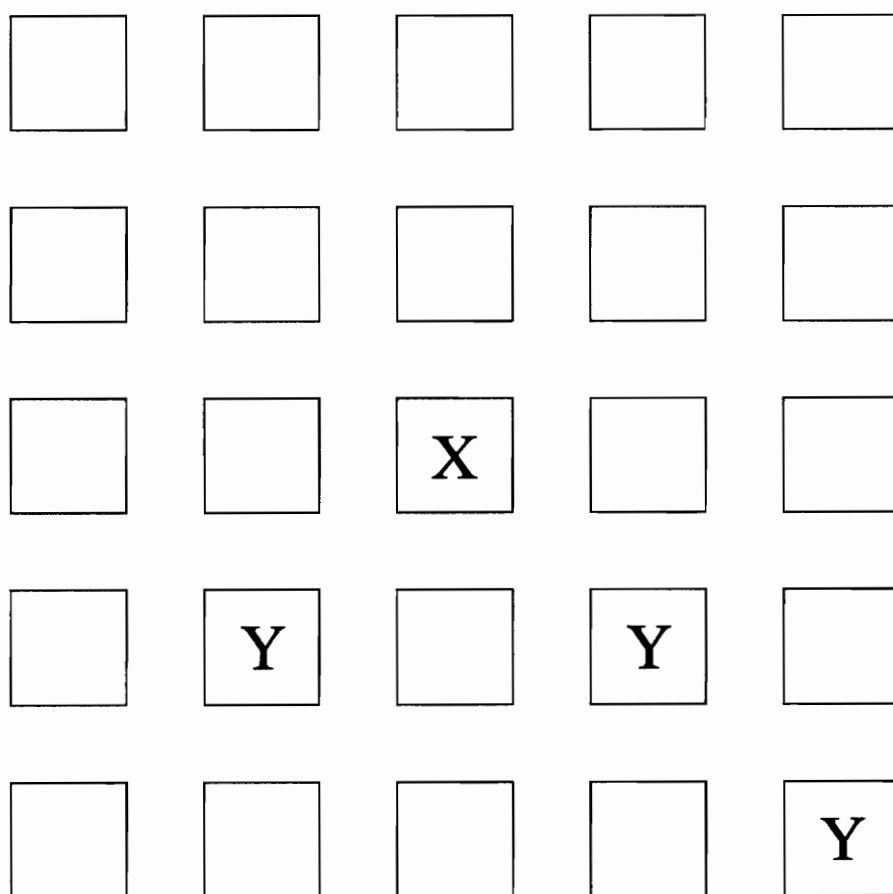


Figure 4.47 - Properly detected, tightly clustered quadruple fault pattern.
(Case 7)

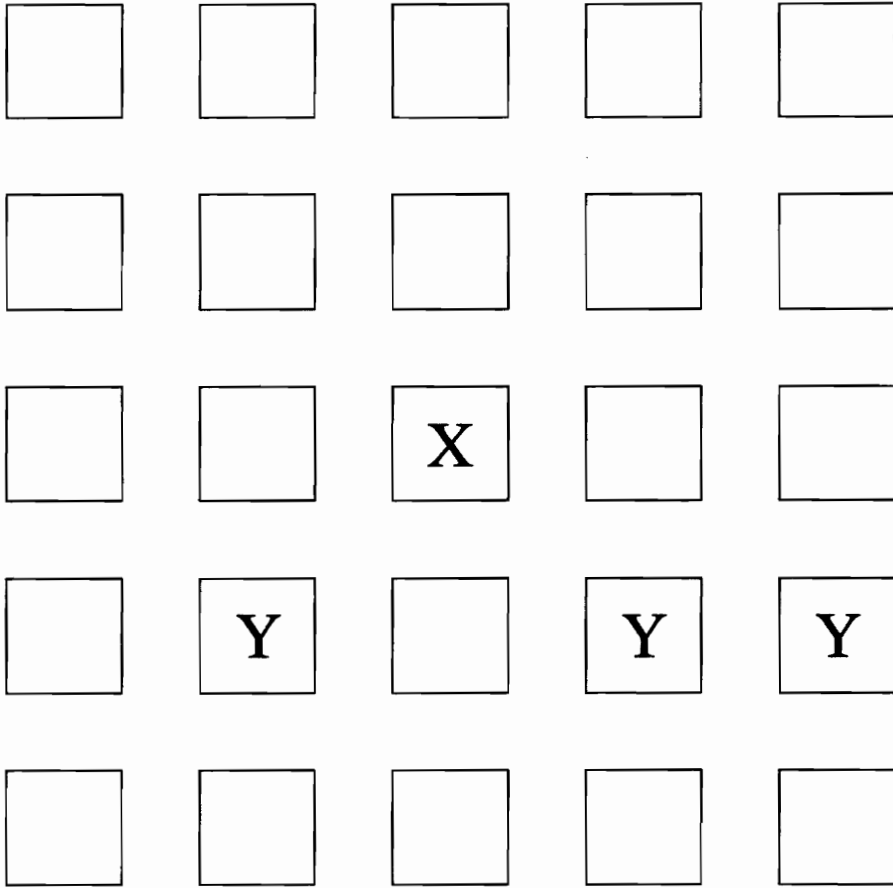


Figure 4.48 - Properly detected, tightly clustered quadruple fault pattern.
(Case 8)

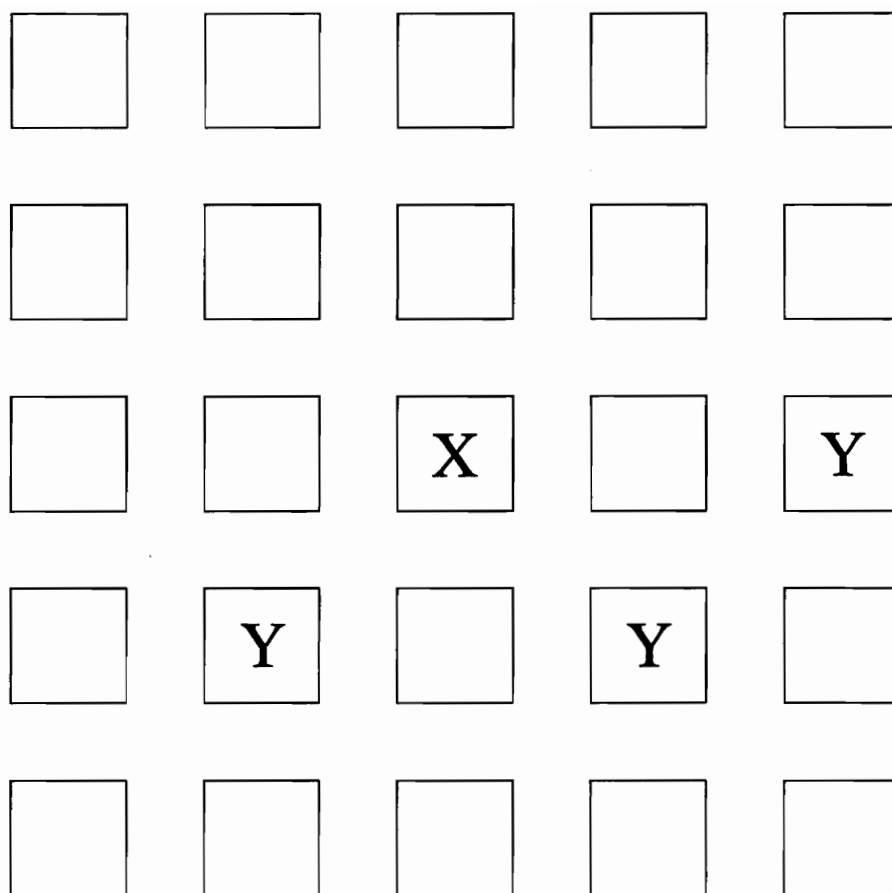


Figure 4.49 - Properly detected, tightly clustered quadruple fault pattern.

(Case 9)

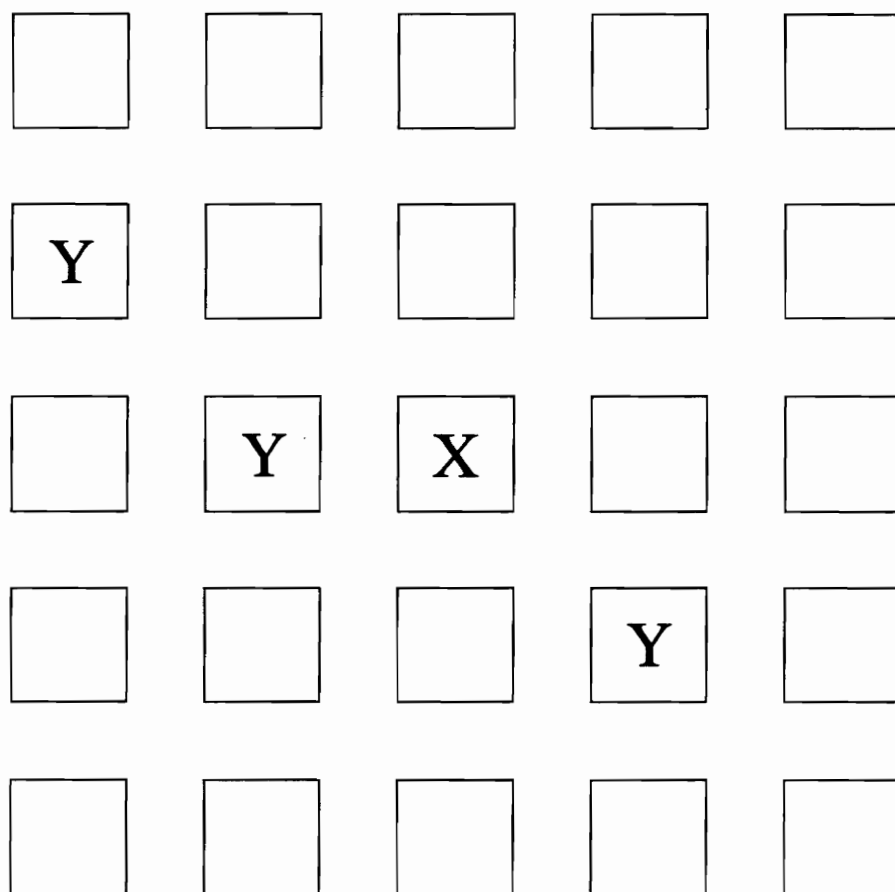


Figure 4.50 - Properly detected, tightly clustered quadruple fault pattern.

(Case 10)

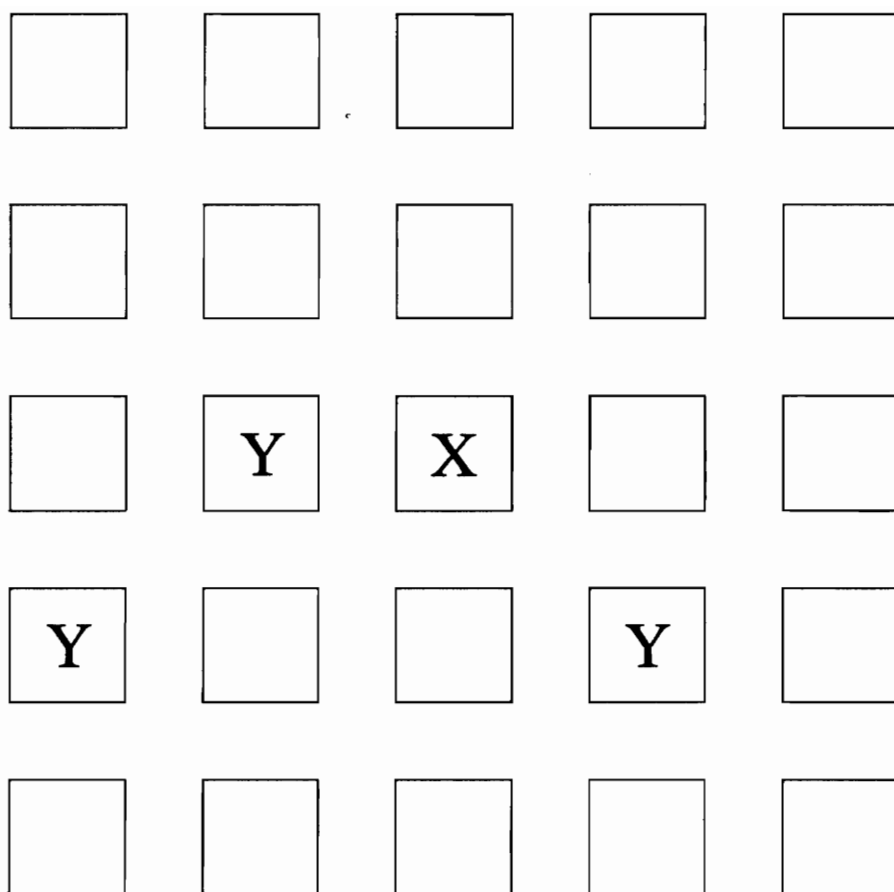


Figure 4.51 - Properly detected, tightly clustered quadruple fault pattern.

(Case 11)

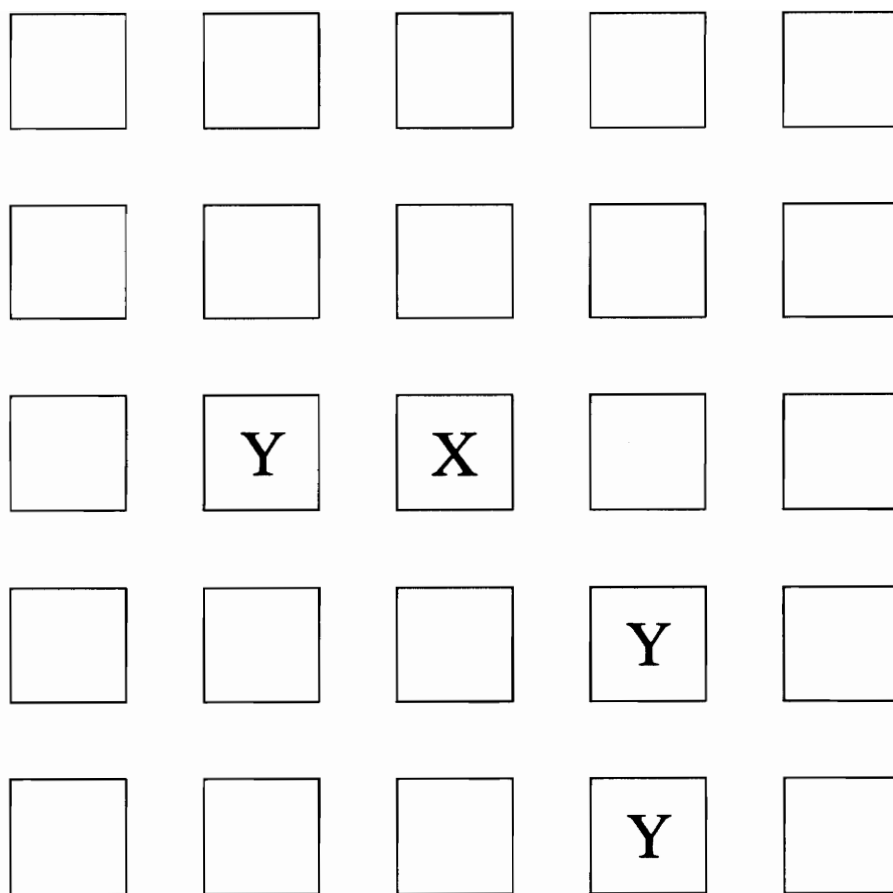


Figure 4.52 - Properly detected, tightly clustered quadruple fault pattern.

(Case 12)

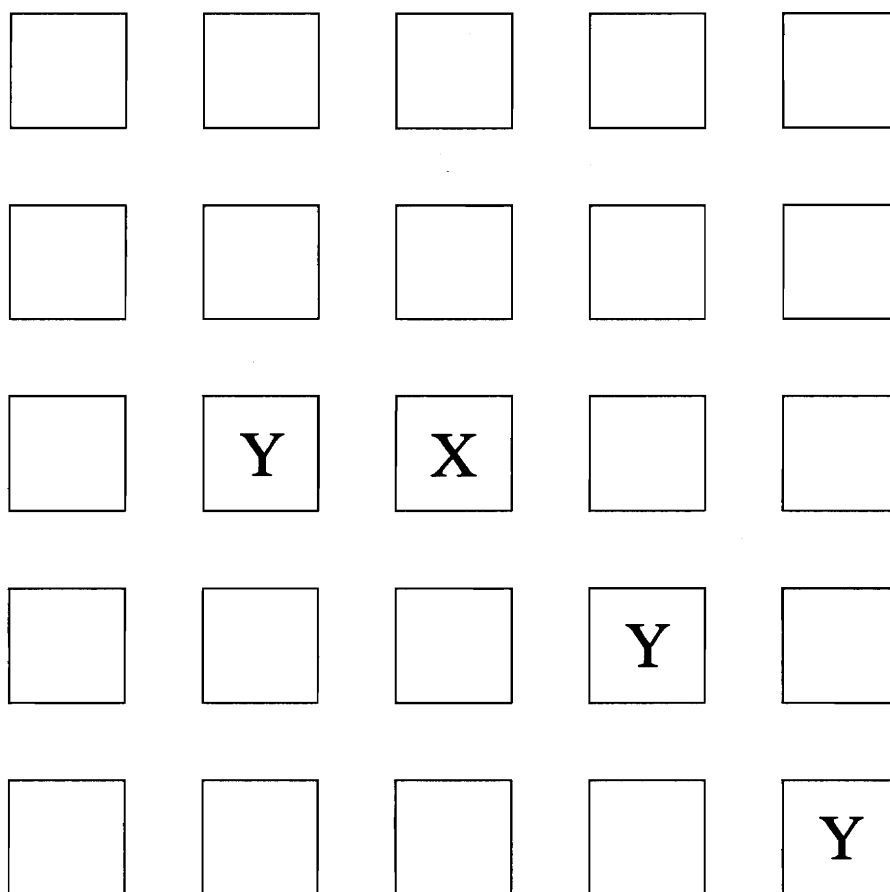


Figure 4.53 - Properly detected, tightly clustered quadruple fault pattern.
(Case 13)

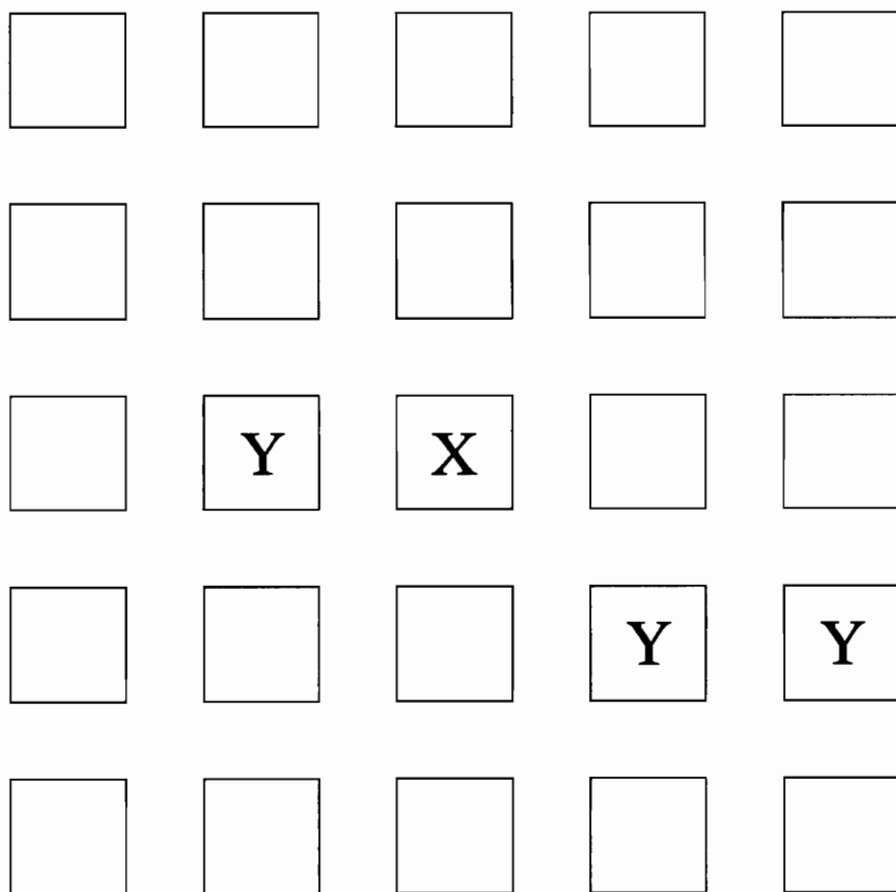


Figure 4.54 - Properly detected, tightly clustered quadruple fault pattern.

(Case 14)

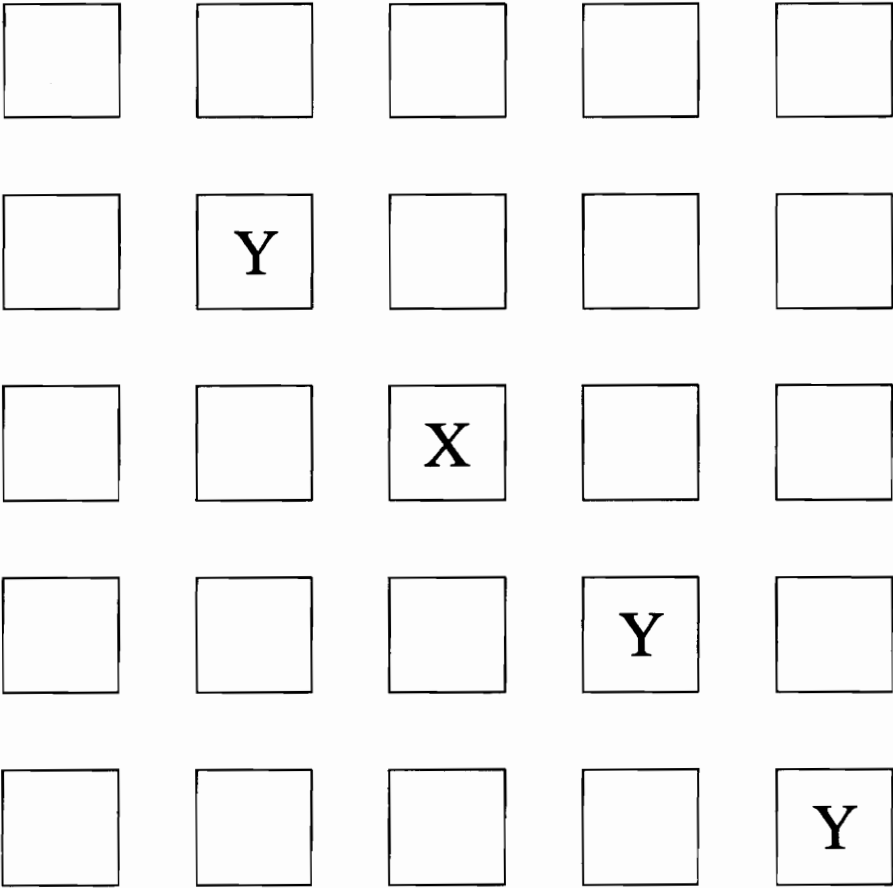


Figure 4.55 - Properly detected, tightly clustered quadruple fault pattern.
(Case 15)

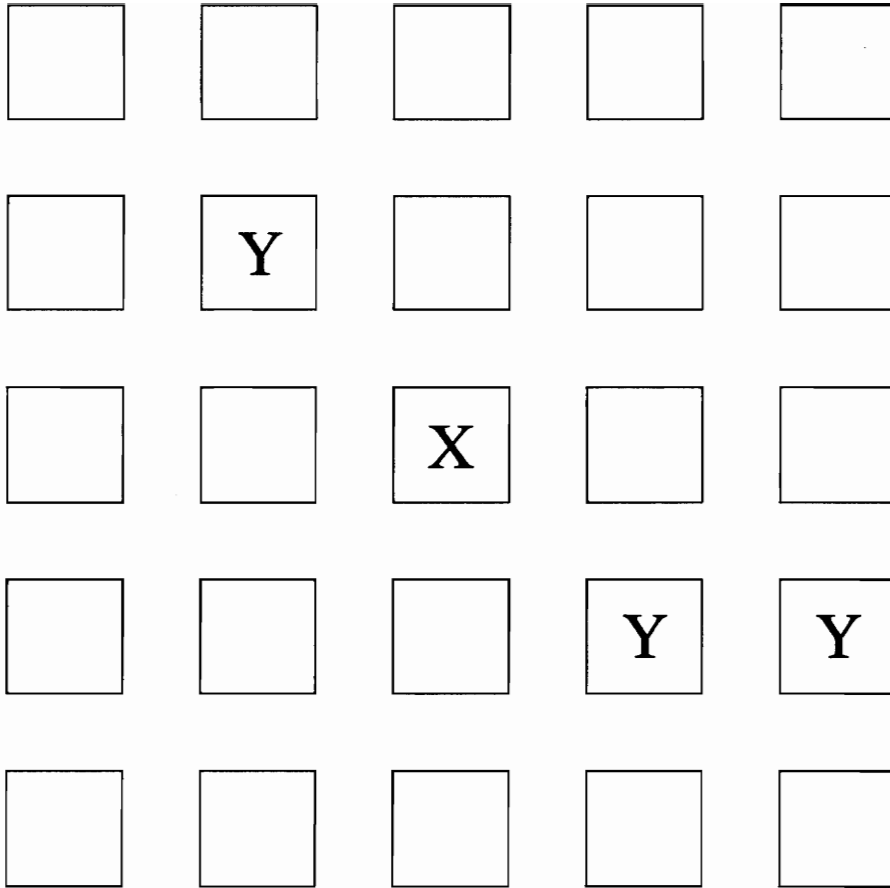


Figure 4.56 - Properly detected, tightly clustered quadruple fault pattern.

(Case 16)

one or more faults at time $t = t_{MAX}$. After waiting t_{MAX} time steps, at time $t = 2t_{MAX}$, this cell is prepared for local reconfiguration. Note that it is assumed that no additional faults occur during this period, since the value for t_{MAX} would otherwise have no applicability. In other words, $MTBF \gg 2t_{MAX}$ must be true for this scheme to function as intended, where MTBF is the mean time between failures of the system.

4.7 RELIABILITY ANALYSIS

The assumption that $MTBF \gg 2t_{MAX}$ needs to be justified. Also, it is desirable to show the improvement in the reliability of the reconfigurable system over a system with no reconfiguration capability. First consider an M -by- N array of processing elements with no reconfiguration capability, where each of the PEs is assumed to have a failure rate of $\lambda = 10^{-6}$ failures per hour. Since the array cannot reconfigure, the reliability of the system is given by

$$R(t) = e^{-MN\lambda t}. \quad (4.1)$$

For $M = N = 10$, the reliability of the system is

$$R(t) = e^{-100\lambda t}. \quad (4.2)$$

The mean time between failures of the system is defined as

$$MTBF = MTTF + MTTR, \quad (4.3)$$

where MTTF is the mean time to failure of the system, given that the system is operational at time $t = 0$, and MTTR is the mean time to repair the system after a fault occurs. Since the system is non-repairing, we have

$$MTBF = MTTF. \quad (4.4)$$

The mean time to failure of the system can be found by integrating the reliability function for all $t \geq 0$. Integrating Equation 4.1 over this interval yields

$$MTTF = 1/(MN\lambda). \quad (4.5)$$

For $M = N = 10$, the mean time to failure is

$$MTTF = 10^4 \text{ hours}. \quad (4.6)$$

Now consider the locally reconfigurable system, which currently has the goal of tolerating all triple faults. Even though the system can tolerate many of the higher order fault patterns, a lower bound on the reliability of the system is the same as if the system could not tolerate any fault pattern more complex than a triple fault:

$$R(t) = \sum_{i=0}^3 \binom{MN}{i} e^{-(MN-i)\lambda t} (1 - e^{-MN\lambda t})^i. \quad (4.7)$$

The summation yields

$$\begin{aligned} R(t) = & \binom{MN}{3} e^{-(MN-3)\lambda t} + \left[\binom{MN}{2} - 3 \binom{MN}{3} \right] e^{-(MN-2)\lambda t} \\ & + \left[MN - 2 \binom{MN}{2} + 3 \binom{MN}{3} \right] e^{-(MN-1)\lambda t} + \left[1 - MN + 2 \binom{MN}{2} - \binom{MN}{3} \right] e^{-MN\lambda t} \end{aligned} \quad (4.8)$$

which for $M = N = 10$ gives

$$R(t) = 161700 e^{-97\lambda t} - 480150 e^{-98\lambda t} + 475300 e^{-99\lambda t} - 156849 e^{-100\lambda t}. \quad (4.9)$$

It can be shown that the reliability expression for the reconfigurable array in Equation 4.9 is greater than the expression of Equation 4.2, which represents the reliability

It can be shown that the reliability expression for the reconfigurable array in Equation 4.9 is greater than the expression of Equation 4.2, which represents the reliability of the same array without the capability for local reconfiguration. It is perhaps more meaningful to examine the improvement in the MTTF of the reconfigurable system over that of the system without reconfiguration capability. Given Equation 4.3, it is reasonable to expect that the mean time to repair the system should be small with respect to the mean time to failure. Otherwise, the system would spend a considerable amount of time in the repair mode. If the MTTR is assumed to be negligible with respect to the MTTF, it follows that Equation 4.4 also holds for the reconfigurable system. Performing the integration of the reliability function of Equation 4.9 for all $t \geq 0$ yields

$$MTTF = 40614 \text{ hours}, \quad (4.10)$$

which is more than a fourfold improvement over the MTTF of the non-reconfigurable system. The assumption that the MTTR is negligible with respect to the MTTF is validated by Equation 4.10, since the total time for local reconfiguration is assumed to be on the order of seconds or fractions thereof. If this were not the case, the reconfigurable array would have little or no practical application. Equation 4.10 also validates the assumption that $MTBF \gg 2t_{MAX}$, since the time required for distributed diagnosis is less than that needed to perform local reconfiguration.

4.8 RESULTS

Table 4.1 summarizes the fault simulation results for up to quadruple faults. Note that for quadruple faults, the figures given for the von Neumann and Moore neighborhoods represent the maximum times needed to properly diagnose the fault

Table 4.1 - Maximum fault latency times for up to quadruple faults.

Number of Faults	Fault Data Distribution Neighborhood	t_{MAX}
1	von Neumann	5
	Moore	3
	Lawson	3
	White	2
2	von Neumann	7
	Moore	4
	Lawson	4
	White	3
3	von Neumann	10
	Moore	4
	Lawson	4
	White	3
4	von Neumann	12*
	Moore	6*
	Lawson	4
	White	3

*The indicated fault latency times do not consider fatal quadruple faults when these neighborhoods are used to distribute the fault data.

patterns which each neighborhood can successfully accommodate. The value for t_{MAX} is chosen to be that given for the quadruple fault case, since any simpler fault patterns are covered as well. This approach accommodates all triple faults, while allowing future research to investigate a local reconfiguration algorithm for quadruple fault patterns.

The von Neumann neighborhood requires less hardware overhead than any of the other neighborhoods shown, since only 4 fault buffers are needed for the combinational logic used to set the fault register flags. However, this neighborhood offers the poorest value for the fault latency parameter, with $t_{MAX} = 12$. The equal scope of the Moore and Lawson neighborhoods implies an equal amount of internal hardware for each cell. These neighborhoods require 8 fault buffers and the associated combinational logic. However, the Lawson neighborhood provides a better value for the fault latency parameter, $t_{MAX} = 4$, as opposed to the Moore neighborhood, with a value of $t_{MAX} = 6$. The White neighborhood has the largest scope, and hence the greatest hardware requirement, of any of the other neighborhoods. A total of 12 buffers and the accompanying increase in combinational logic is needed. This approach gives only a marginal improvement in fault latency over the Lawson neighborhood, with $t_{MAX} = 3$. These results suggest that the Lawson neighborhood offers a good compromise between the conflicting goals of minimizing both fault latency and the additional hardware needed to support an expanded fault detection network, while providing a significant improvement in reliability over an array lacking the ability to reconfigure.

4.9 CONCLUSIONS

This chapter presents a distributed scheme for fault diagnosis. An approach of this type is needed, since it is unrealistic to assume that a faulty cell is instantaneously and simultaneously detected by all of its neighbors, or that global knowledge of a fault exists within the array. Instead, the fault register concept of [Whi91] is expanded to accommodate the distributed diagnosis scheme. The region of awareness concept is introduced to relate the fault register contents to a geometrical view of the fault data passing scheme. Since a major goal of the current local reconfiguration research is to provide complete coverage of all triple fault patterns, the fault register is now capable of properly detecting all such fault patterns.

Four different neighborhoods for passing the fault data are provided in the software simulation: von Neumann, Moore, Lawson, and White. The simulation results show that the method presented here provides proper diagnosis of up to triple faults, when a von Neumann scheme is used to pass the fault data within the array. When a Lawson or White neighborhood is used instead, the same approach also allows all quadruple faults to be properly diagnosed. Furthermore, the only quadruple fault pattern not properly diagnosed by the Moore neighborhood can not be tolerated by the present local reconfiguration hardware, and so is of little importance at the present time.

The simulation results also determine the upper bound on the fault latency parameter, t_{MAX} , for up to quadruple faults. The value of t_{MAX} depends on the neighborhood used to pass the fault register data, which in turn is directly related to the amount of hardware needed to support that neighborhood. However, some of the

required hardware is already included by the current local reconfiguration scheme, since each cell is physically connected to the 12 neighbors in a White neighborhood. The present hardware fully supports the logical von Neumann neighborhood used for local reconfiguration. The only additional hardware needed to support a larger neighborhood consists of a fault register buffer for each additional neighbor, and the combinational logic needed to set the appropriate bits in the fault register. The increase in hardware complexity needed to accommodate the reconfigurable architecture does degrade the reliability of a single cell somewhat. However, the reliability calculations of Section 4.6 indicate a substantial increase in the reliability of the fault-tolerant array with respect to an array without reconfiguration capability.

By simply adding four fault data buffers and the requisite combinational logic, the Lawson neighborhood offers diagnosis of all quadruple faults within $t_{MAX} = 4$. This value is one-third of the value $t_{MAX} = 12$ provided by the von Neumann neighborhood, and two-thirds of the value $t_{MAX} = 6$ provided by the Moore neighborhood, with the same hardware cost as the latter. The White neighborhood requires 8 additional fault data buffers and associated combinational logic over the von Neumann neighborhood. However, with twice the number of additional buffers as the Moore and Lawson neighborhoods, the White neighborhood provides only marginal improvement over the latter in fault latency, with $t_{MAX} = 3$. Therefore, the Lawson neighborhood appears to be the best overall solution to the distributed diagnosis problem, in terms of fault coverage, reduced fault latency, and hardware complexity.

The results of this chapter are being used by Joseph Wegner to develop an improved local reconfiguration algorithm, which uses a slightly modified version of the existing local reconfiguration mechanism to accommodate all triple faults. Since the diagnosis algorithm makes it possible to identify all such faults, the system can be made 3-fault tolerant. As noted in Chapter 2, this is a significant improvement over the existing approach [Whi91], which is unable to perform successful local reconfiguration in the presence of L-faults.

5. SELF-CHECKING CIRCUITS

This chapter investigates available techniques for implementing control cells in the array, such that the self-checking assumptions made in Chapter 4 are satisfied. That is, faulty control cells are recognized as such by at least one neighbor as soon as an erroneous output occurs, whereupon the distributed fault detection mechanism notifies other cells of the faults. The theory of self-checking circuits is both diverse and complex. Therefore, rather than presenting a comprehensive treatment of the theory of self-checking circuits, this chapter first presents the basic definitions and concepts, and then focuses on some techniques found in the literature which appear applicable to the goals of this research.

It is not currently practical to attempt a complete self-checking implementation of the control cell. This is because the global and local reconfiguration mechanisms cannot be integrated until the improved local reconfiguration algorithm currently under development by Joseph Wegner is completed (see Chapter 4). The Reconfiguration Finite State Machine, which is the heart of the control cell architecture, can be made self-checking after its behavior is determined by the integration of the two reconfiguration algorithms [JaC88]. Therefore, the general theory and structure for a self-checking implementation of the control cell is presented, and particular attention is given to methods needed to transfer data between cells. It is shown that the system satisfies the self-checking assumptions made in the distributed diagnosis algorithm of Chapter 4.

5.1 PROPERTIES OF SELF-CHECKING CIRCUITS

The theory of self-checking circuits is formalized in [AnM73]. The block diagram of Figure 5.1 shows the original self-checking circuit concept. It consists of a functional circuit, and a checker which monitors the output of the functional circuit, and produces an error indication when an erroneous output is detected. The inputs and outputs of the functional circuit are encoded to guard against errors, and the checker simply monitors the output lines of the functional circuit to insure that properly encoded outputs are produced.

5.1.1 TOTALLY SELF-CHECKING (TSC) CIRCUITS

The checker of Figure 5.1 monitors the encoded output lines of the functional circuit, and produces an error indication on its outputs when an erroneous output, or *non-codeword*, is detected. However, the possibility exists that the checker is faulty. It is not practical to add a second checker to check the first checker, as this leads to an infinitely regressive solution as more and more checkers are added to the system. Instead, the checker is designed to be *self-checking*. That is, faults in the checker can produce an error indication on the checker outputs, just as if the checker detects a fault in the functional circuit. Some definitions are needed to further characterize self-checking circuits.

Definition 5.1:

If a fault f causes a functional circuit to produce a codeword output other than the expected codeword output for a given codeword input, the output codeword is said to be an *incorrect codeword*.

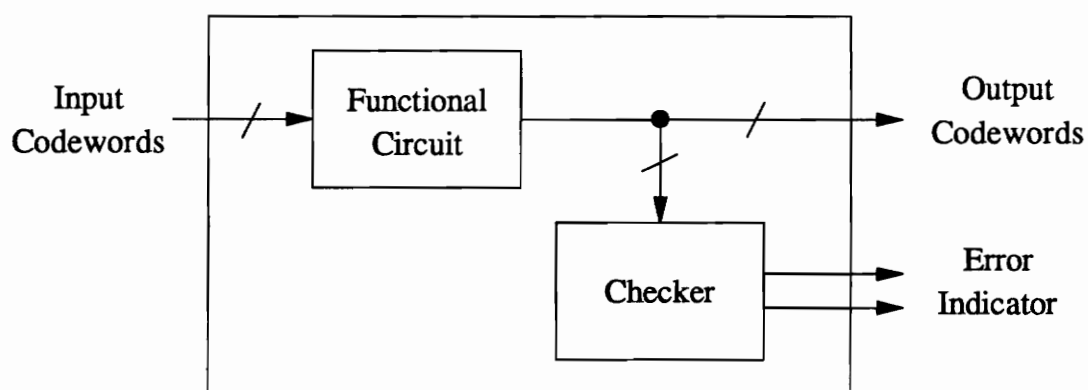


Figure 5.1 - Basic self-checking circuit.

Definition 5.2:

A circuit is said to be *fault-secure* with respect to a set of faults F if, for any fault $f \in F$, the circuit never produces an incorrect codeword output for a codeword input [JhK90]. That is, either the correct codeword or a non-codeword output is produced by the network in response to any given codeword input. The set F is called the *fault set*, and contains all the faults for which a given circuit property, such as fault-secureness, holds.

Definition 5.3:

A circuit is *self-testing* with respect to a fault set F if, for any fault $f \in F$, at least one codeword input results in a non-codeword output during normal operation of the circuit [JhK90].

Definition 5.3 implies that a faulty self-testing circuit eventually receives one or more input codewords which cause the fault to be detected as a non-codeword output. Suppose for any given fault $f \in F$, there is a set of input codewords, I , such that a given circuit produces non-codeword outputs in the presence of f only when codewords in I are applied to the input. If no member of I is applied to the circuit during the course of normal operation, the circuit is not self-testing with respect to f .

Definition 5.4:

A circuit is *code-disjoint* if any codeword input produces a codeword output, and any non-codeword input produces a non-codeword output [JhK90].

Definition 5.5:

A circuit is *totally self-checking* (TSC) with respect to a fault set F if it is both fault-secure *and* self-checking. A *TSC checker* is both TSC *and* code-disjoint [JhK90].

Definition 5.5 implies that any non-codeword on the checker outputs signifies an error in either the checker or the functional circuit. The following assumptions are made for faults in TSC circuits [JhK90]:

- 1) Faults occur sequentially.
- 2) All of the codeword tests required to detect a fault are applied so that the fault is detected before the next fault occurs.

Under these fault assumptions, a TSC circuit always produces a non-codeword as its first erroneous output. A circuit is said to satisfy the *TSC goal* if it produces a non-codeword as its first erroneous output. This allows the fault to be detected as soon as its output response differs from expected behavior. The TSC goal must be met in order to validate the distributed diagnosis algorithm presented in Chapter 4. Designing TSC systems is extremely complex, in general. However, systems can be designed which meet the TSC goal under the same fault assumptions as for TSC circuits, without necessarily being TSC themselves [SmM78]. This is accomplished by constructing the system from subsystem blocks which meet the TSC goal.

5.1.2 STRONGLY FAULT-SECURE (SFS) CIRCUITS

Strongly fault-secure (SFS) circuits comprise a class of functional circuits which meet the TSC goal [SmM78]. Suppose a circuit is fault-secure but not self-testing. Then

there exists a fault $f_1 \in F$ such that the circuit never produces an erroneous output for codeword inputs. If another fault $f_2 \in F$ occurs, it is possible that the *fault sequence* $(f_1 \cup f_2) \notin F$ allows the network to produce incorrect codeword outputs. Such fault sequences may contain any number of faults from F , which may occur in any order. Clearly, it would be advantageous to insure the fault-secure property in the presence of these fault sequences. A general class of such circuits which meet the TSC goal can now be defined.

Definition 5.6:

A circuit is *strongly fault-secure* with respect to a set of faults F if, for any fault $f \in F$, either

- a) the circuit is self-testing and fault-secure, or
- b) the circuit is fault-secure, and if another fault from F occurs in the circuit, then either a) or b) is true for the fault sequence [JhK90].

Since SFS circuits are fault-secure, even in the presence of undetectable fault sequences, the first erroneous output due to a fault sequence is guaranteed to be a non-codeword. Thus, the TSC goal is met by SFS circuits.

5.1.3 STRONGLY CODE-DISJOINT (SCD) CIRCUITS

Just as SFS circuits remain fault-secure in the presence of undetectable fault sequences, another class of circuits can be defined which maps non-codeword inputs to non-codeword outputs, even in the presence of undetectable fault sequences. These circuits are known as strongly code-disjoint (SCD) circuits [JhK90].

Definition 5.7:

A circuit is *strongly code-disjoint* with respect to a set of faults F if, before the occurrence of any fault, the circuit is code-disjoint, and for any fault $f \in F$, either

- a) the circuit is self-testing, or
- b) the circuit always maps non-codewords at its inputs to non-codewords at its outputs, and if another fault from F occurs in the circuit, then either a) or b) is true for the fault sequence [JhK90].

5.2 CODING THEORY FOR SELF-CHECKING CIRCUITS

Self-checking circuits are based on error-detecting codes. These codes employ information redundancy, which allows information to be recovered in the presence of errors. *Systematic codes* and *separable codes* are codes in which the k information bits of the original data are appended with r checkbits. The symbol r is called the *redundancy* of the code [JhK90]. The number of checkbits generally indicates the extent to which a codeword can be corrupted and still allow the original information to be recovered. Systematic codes use all 2^k possible combinations for the information bits, whereas separable codes use a proper subset of the available combinations. Systematic codes and separable codes are easily accommodated in circuit design, since the information bits and checkbits can be processed separately. *Non-separable codes* are a third general class of codes, in which the information bits cannot be separated from the check bits. These codes are generally less redundant than systematic codes and separable codes, but can result in more difficult circuit implementations, since circuits must process the codeword as a whole. This chapter will only consider systematic codes. A code is optimal with respect

to a given class of codes if it contains the fewest checkbits for a given number of information bits.

The encoding for the functional circuit may be based on any one of several error-detecting codes; *unordered codes*, including *two-rail codes* are commonly used for this purpose. Checker outputs are usually encoded in a two-rail code, whereas other circuits may employ a different unordered code.

A code is said to be *unordered* if no codeword contains 1s in all of the same bit positions as any other codeword has 1s. By counterexample, a code containing 1011 and 1010 as codewords would not be unordered, since 1011 contains 1s in all the same bit positions in which 1010 has 1s. Unordered codes are useful for detecting all *unidirectional errors*. Such errors are characterized by transitions from 0 to 1 only, or from 1 to 0 only. That is, all erroneous bits change in the same fashion, rather than some 0 bits changing to 1, and some 1 bits changing to 0. According to [PrS80], unidirectional errors are the most prevalent errors found in VLSI circuits. It is therefore desirable to detect these errors. If a code is unordered, no unidirectional errors can change a codeword into any other codeword, since the result is simply increasing or decreasing the number of 1s in the original codeword. Note that the vector containing all 0s and the vector containing all 1s are never codewords in an unordered code, since a unidirectional error could incorrectly produce either one of these vectors.

5.2.1 BERGER CODES

Berger codes are systematic, unordered codes, for which the checkbits can be obtained using one of two approaches. One approach derives the r checkbits from the binary representation for the number of 0s in the k information bits. The other approach derives the checkbits from the 1s complement of the binary representation for the number of 0s in the information bits. Both approaches produce the same checkbits for any given information vector if and only if $k = 2^r - 1$; such a code is known as a *maximal-length* Berger code. Designs for TSC checkers for Berger codes are presented in [AsR77]. One potential problem with using Berger codes in self-checking networks occurs when $k = 2^{r-1}$. In such cases, no known TSC checker network exists. However, there exists a class of codes, known as *modified Berger codes*, which are equivalent to these Berger codes, and for which TSC checkers can be designed [AsR77]. Berger codes are optimal systematic codes.

5.2.2 BOSE-LIN CODES

Bose-Lin codes belong to a class of codes known as *t-unidirectional error-detecting codes*, as they are capable of detecting up to t unidirectional errors. These codes may be appropriate for use in self-checking VLSI systems if an upper bound on the number of unidirectional errors is known to be t [JhK90]. For example, unidirectional errors may occur in bursts, where up to t bits may be unidirectionally changed. In such cases, Bose-Lin codes reduce the number of checkbits needed, and hence the amount of hardware required to support the encoding scheme. The number of checkbits for a Bose-Lin code increases with t , but is independent of the number of information bits. The parity

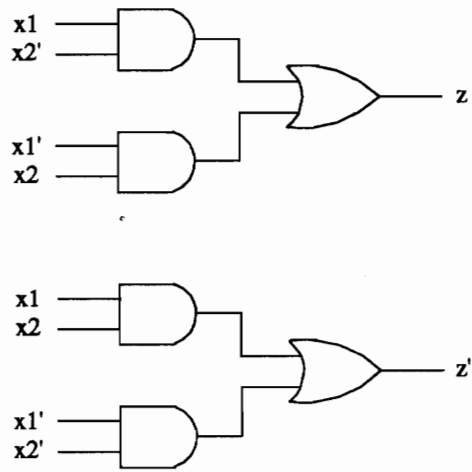
code is a special case of the Bose-Lin code, where $t = 1$. That is, any one-bit error is detectable. Determining the checkbits for a given information vector can be complex, as described in [BoL85]. Implementations for TSC checkers for Bose-Lin codes are given in [Jha91].

5.2.3 THE TWO-RAIL CODE

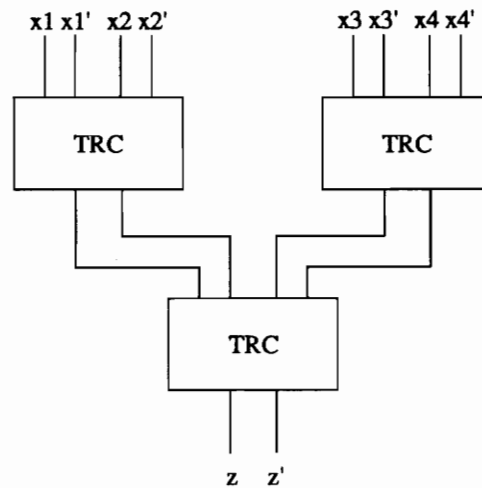
The output of the TSC checker of Figure 5.1 is often encoded using a *two-rail code*. That is, the error indicator consists of two lines which maintain complementary values, 01 or 10, during fault-free operation, and which produce the non-codeword outputs 00 or 11 when an error condition exists. Obviously, the two-rail code word is unordered, since neither codeword contains 1s in all the same bit positions as the other. If one of the outputs of a TSC checker contains a stuck-at fault, the self-testing property insures that the first erroneous output is a non-codeword. Figure 5.2 shows a 2-variable TSC checker for the two-rail code. Also shown in the figure is a 4-variable TSC checker for the two-rail code, which is a tree structure consisting of 2-variable checkers. In general, a 2^q -variable TSC checker for the two-rail code can be implemented by constructing a tree of 2-variable two-rail TSC checkers. Such designs are easily testable, as only 4 codewords are needed to check for all single and unidirectional faults in the tree [JhK90].

5.3 A SELF-CHECKING CONTROL CELL ARCHITECTURE

Using the fundamental concepts of self-checking circuits, a general plan for a self-checking control cell architecture can be presented. The functionality of the control cell is



a) A 2-variable TSC checker for the two-rail code.



b) A 4-variable TSC checker for the two-rail code composed of 2-variable TSC checkers.

Figure 5.2 - TSC checkers for the two-rail code.

largely governed by the Reconfiguration Finite State Machine (RFSM), first proposed by Kumar in [Kum84], and investigated further by Brighton in [Bri87]. With the advent of the local reconfiguration scheme proposed by White in [Whi91], it seems logical to integrate the current global and local reconfiguration algorithms, along with the distributed fault detection algorithm proposed in Chapter 4 of this thesis, so that the RFSM addresses all of these issues as well. Furthermore, a rollback methodology should be included to address transient faults, as these faults are much more prevalent in VLSI circuits than permanent faults [Jha93]. It should be noted that such a rollback mechanism is currently being investigated by Vinay Murthy, under the direction of Dr. F. Gail Gray.

It is not feasible at present for this researcher to integrate all of these schemes before presenting a self-checking cell architecture, as the local reconfiguration scheme is currently under revision, and the rollback mechanism is not fully developed. Therefore, the purpose of this section is to present a general methodology for creating a self-checking control cell architecture, so that future researchers may utilize this approach after the integrated RFSM is well-defined.

Figure 5.3 shows the interconnections between a pair of self-checking control cells. Although it is not shown in the figure, a White interconnection neighborhood is assumed. Neighbors communicate with each other via a pair of bidirectional communications links, and two pairs of unidirectional internal fault indicators. It is desired that neighbors exchange data in a serial fashion, but a single communications link is prone to undetectable faults. Thus, a two-rail encoding scheme is used to transmit the serial data from one cell to another. The fault indicator links are also encoded in a two-rail fashion, allowing cells to identify internal faults in neighboring cells. Cells which receive non-

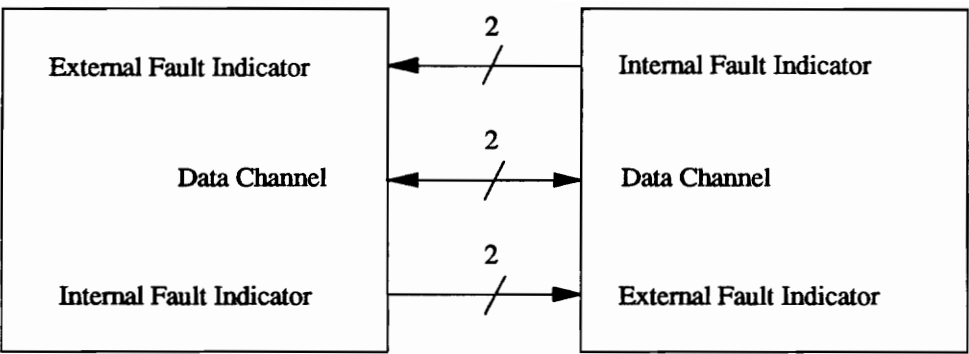


Figure 5.3 - Control cell interconnections for two-rail data transfers and error checking.

codewords from a neighbor on any links trap the corrupted data internally to prevent it from spreading to other parts of the system.

Figure 5.4 shows a block diagram of the self-checking control cell. The architecture reflects the anticipated needs of the hierarchical reconfiguration system, without any provisions for a rollback mechanism. The control cell contains the following components:

- 1) a two-rail encoded circuit for trapping erroneous incoming data,
- 2) bidirectional I/O circuitry to provide serial communications among neighboring cells,
- 3) a set of registers and buffers,
- 4) the RFSM,
- 5) a code conversion module between the RFSM and the register/buffer file,
- 6) a TSC checker for the outputs of the code conversion module, and
- 7) a set of 12 two-rail TSC checkers used as internal error indicators (not shown).

The last item is not included in the figure in order to reduce clutter. The two-rail TSC checkers monitor all of the two-rail encoded subsystem interfaces, including the two-rail control signals, with the exception of the data links to neighboring cells. The TSC outputs of the two-rail TSC code checker are also monitored, as well as the two-rail error indicator of the RFSM. The two-rail encoded outputs of the two-rail TSC checkers are provided as internal fault indicators to each of the neighboring cells.

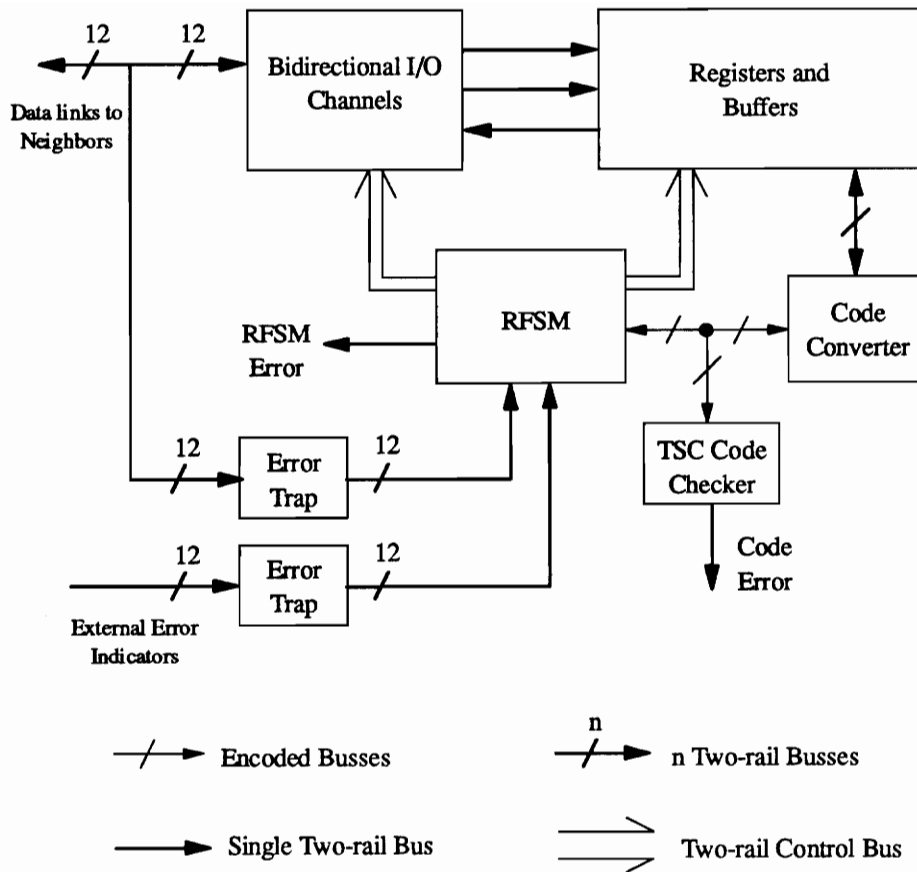
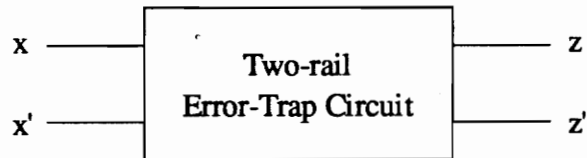


Figure 5.4 - Block diagram of the self-checking control cell architecture.

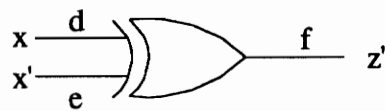
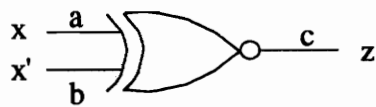
The registers and buffers are those needed to support the known global and local reconfiguration schemes. Despite the fact that the RFSM cannot currently be specified, other circuits contained in the self-checking cell architecture can be defined at the present time, and a general technique for a self-checking implementation of the RFSM also exists.

5.3.1 THE INCOMING DATA ERROR-TRAP CIRCUITRY

Figure 5.5 shows that the circuitry needed to trap incoming data errors is quite simple. Two-rail data in the form $\{x, x'\}$ enters the error-trap circuit, which produces the two-rail output $\{z, z'\} = \{0, 1\}$ for properly encoded two-rail input data, and $\{z, z'\} = \{1, 0\}$ when a non-codeword input is detected. Figure 5.4 shows that the internal fault indicators supplied by each neighboring cell are also monitored by an error-trap block, and the outputs of both circuits are used by the RFSM to set the appropriate bits in the fault register when erroneous data from a neighbor is detected. Thus, data from faulty neighboring cells can be ignored by the RFSM. Tables 5.1 and 5.2 show that the error-trap circuit of Figure 5.5 meets the TSC goal for single stuck-at faults, since any such fault produces either the correct codeword output or a non-codeword. In the tables, the symbol $a/0$, for example, indicates a stuck-at-0 fault on line a of Figure 5.5. Unidirectional faults are also covered by the error-trap circuit, since the two-rail code is unordered. The fault-free circuit is not code-disjoint, as non-codeword inputs are mapped to two-rail encoded outputs. Since the circuit is not intended as a self-checking checker, the lack of the code-disjoint property is not problematic in this case [Jha93]. However, it is desirable to detect internal faults in the error-trap circuit, so that neighboring cells can recognize the fault. Therefore, the outputs of the error-trap circuits are monitored by the 12 two-rail TSC checkers, each of which provides an internal fault indication to one of the



a) The two-rail error-trap circuit detects errors on incoming data lines.



b) An implementation of the two-rail error-trap circuit which meets the TSC goal.

Figure 5.5 - The two-rail error-trap circuit.

Table 5.1 - Results of single stuck-at faults in the XNOR gate for the error-trap circuit.

Fault	x	x'	z	z'	Result Code
a/0	0	0	1	0	C
	0	1	0	1	C
	1	0	1	1	N
	1	1	0	0	N
a/1	0	0	0	0	N
	0	1	1	1	N
	1	0	0	1	C
	1	1	1	0	C
b/0	0	0	1	0	C
	0	1	1	1	N
	1	0	0	1	C
	1	1	0	0	N
b/1	0	0	0	0	N
	0	1	0	1	C
	1	0	1	1	N
	1	1	1	0	C
c/0	0	0	0	0	N
	0	1	0	1	C
	1	0	0	1	C
	1	1	0	0	N
b/0	0	0	1	0	C
	0	1	1	1	N
	1	0	1	1	N
	1	1	1	0	C

Result Codes:

C - Correct Codeword Output

N - Non-Codeword Output

Table 5.2 - Results of single stuck-at faults in the XOR gate for the error-trap circuit.

Fault	x	x'	z	z'	Result Code
d/0	0	0	1	0	C
	0	1	0	1	C
	1	0	0	0	N
	1	1	1	1	N
d/1	0	0	1	1	N
	0	1	0	0	N
	1	0	0	1	C
	1	1	1	0	C
e/0	0	0	1	0	C
	0	1	0	0	N
	1	0	0	1	C
	1	1	1	1	N
e/1	0	0	1	1	N
	0	1	0	1	C
	1	0	0	0	N
	1	1	1	0	C
f/0	0	0	1	0	C
	0	1	0	0	N
	1	0	0	0	N
	1	1	1	0	C
f/1	0	0	1	1	N
	0	1	0	1	C
	1	0	0	1	C
	1	1	1	1	N

Result Codes:

C - Correct Codeword Output

N - Non-Codeword Output

neighboring cells. When the error-trap circuit produces a non-codeword output, the TSC checkers inform neighboring cells of the internal fault. The same technique can be used to directly monitor all of the two-rail variables in the subsystem interfaces, with the exception of the incoming data lines, which are indirectly monitored via the error-trap circuitry. The reason cells cannot share a bidirectional pair of internal fault indicator links is because a cell's own error-trap circuitry would then monitor the outputs of its TSC checkers, which in turn monitor the outputs of the error-trap circuitry, providing an undesirable feedback path.

5.3.2 THE BIDIRECTIONAL I/O CHANNELS

The I/O circuitry block shown in Figure 5.4 is essentially a set of four multiplexed bidirectional channels, each of which can select a data path to one of four physically neighboring cells. The diagonal neighbors (Northeast, Southeast, Southwest, and Northwest), are each connected to two channels, as shown in Figure 5.6. Incoming data from each of the twelve physical neighbors can be routed to the cell's internal buffers under the control of the RFSM. Outgoing data can be similarly routed from the cell's internal registers to the appropriate output path by the RFSM.

Note that the data channels can not only be used to select logical neighbors from the available physical neighbors, as suggested in [Whi91], but also to allow time-division-multiplexed serial communications with the far neighbors, as appropriate for the Lawson neighborhood suggested for use in the distributed fault detection scheme. As indicated in Figures 5.4 and 5.6, 3 two-rail unidirectional data paths provide full-duplex data transfers between the I/O block and the set of registers and buffers. One data path allows the

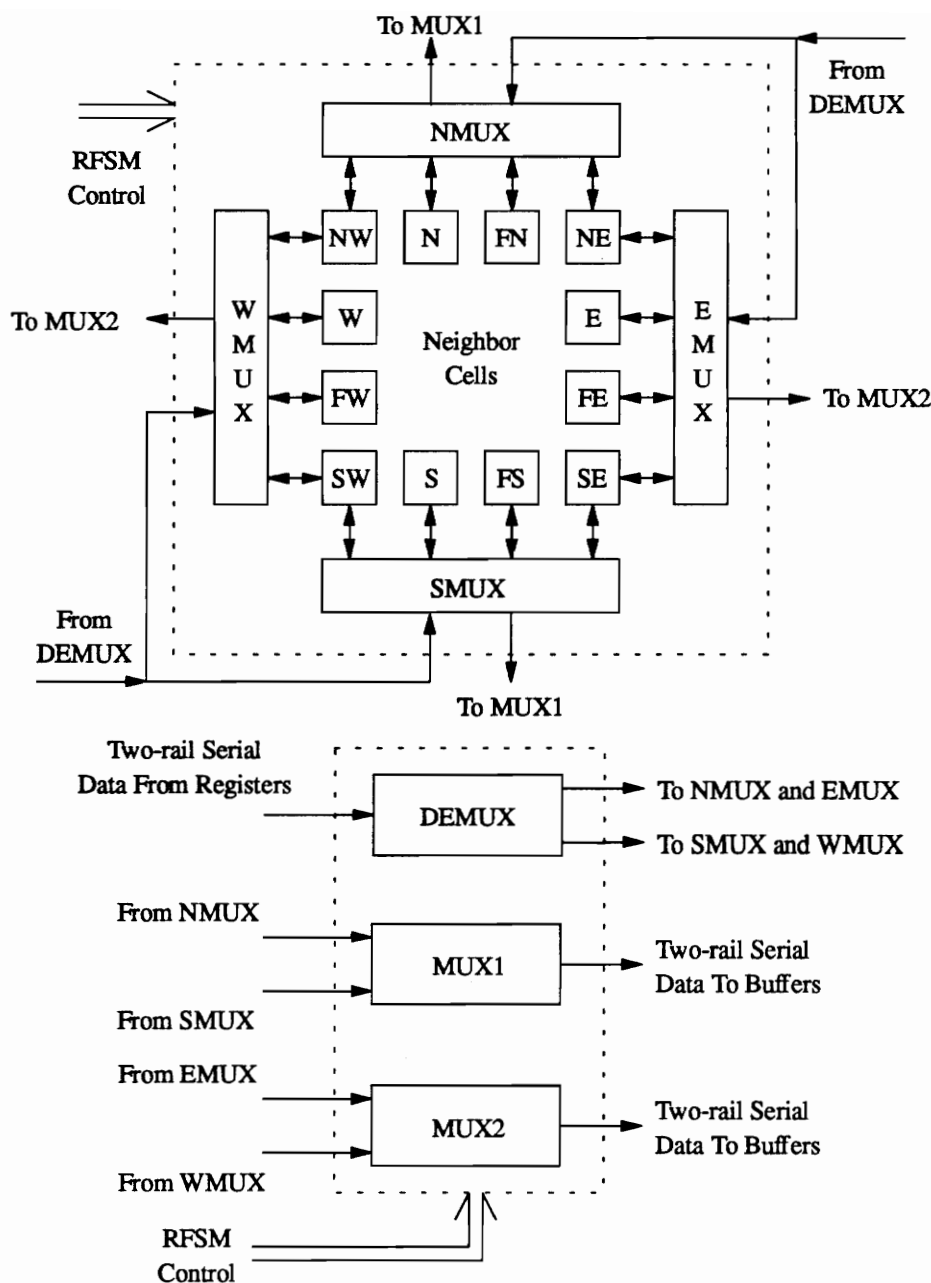


Figure 5.6 - Block diagram of the bidirectional I/O channels for the control cell.

contents of one of the cell's registers to be transmitted simultaneously to two neighbors. At the same time, the other two data paths allow data to be received from two neighbors for storage in the cell's incoming data buffers. Naturally, the two neighbors receiving data must be distinct from the two neighbors transmitting data. The TSC checker networks monitor these data paths and provide internal fault indicators to the neighboring cells.

Under the control of the RFSM, the communications channels can be switched to the correct neighbors at the appropriate time. Each data transmission cycle consists of the following steps:

- 1) Set the data channels for communications with the North, East, South, and West neighbors.
- 2) Transmit data to the North and East neighbors, and receive data from the South and West neighbors.
- 3) Transmit data to the South and West neighbors, and receive data from the North and East neighbors.
- 4) Set the data channels for communications with the Far North, Far East, Far South, and Far West neighbors.
- 5) Transmit data to the Far North and Far East neighbors, and receive data from the Far South and Far West neighbors.
- 6) Transmit data to the Far South and Far West neighbors, and receive data from the Far North and Far East neighbors.
- 7) Update the registers and buffers, including the fault register. Go to step 1.

Since each cycle of the RFSM includes an update of the fault register, each transmission cycle is equivalent to the time step described in Chapter 4. Cells

continuously participate in the transmission cycle. When a new fault is detected, a cell must wait t_{MAX} transmission cycles before attempting local reconfiguration, so that all tolerable faults may be properly detected.

Note that the bidirectional I/O channel circuitry is simply a combinational logic block whose inputs and outputs are both encoded using unordered codes. In [Mag73], it is shown that any such circuit has an inverter-free implementation. Furthermore, [SmM78] asserts that all inverter-free circuits are SFS. Therefore, SFS circuits can be implemented using inverter-free PLAs. Although design techniques for self-checking circuits using PLAs are discussed in [WaA79], the authors place restrictions on the number of product terms which can be simultaneously activated, and assume that the PLA produces all possible codeword outputs. A less restrictive approach for creating SFS circuits with PLAs is given in [MaA82]. The outputs of these PLAs can be monitored by the 12 two-rail TSC internal error indicators, providing the concurrent error detection needed.

5.3.3 THE REGISTERS AND BUFFERS

Figure 5.7 shows the set of registers and buffers from Figure 5.4. These components are needed to support the global reconfiguration scheme of [Bri87], as well as those anticipated for the improved local reconfiguration scheme based on [Whi91] which is currently under development. The distributed diagnosis algorithm of Chapter 4 contributes 4 additional fault data buffers for holding fault data from the far neighbors, as required by the Lawson neighborhood used to distribute the fault data. The buffers and registers are all two-rail encoded. All buffers can be serially loaded with incoming two-rail

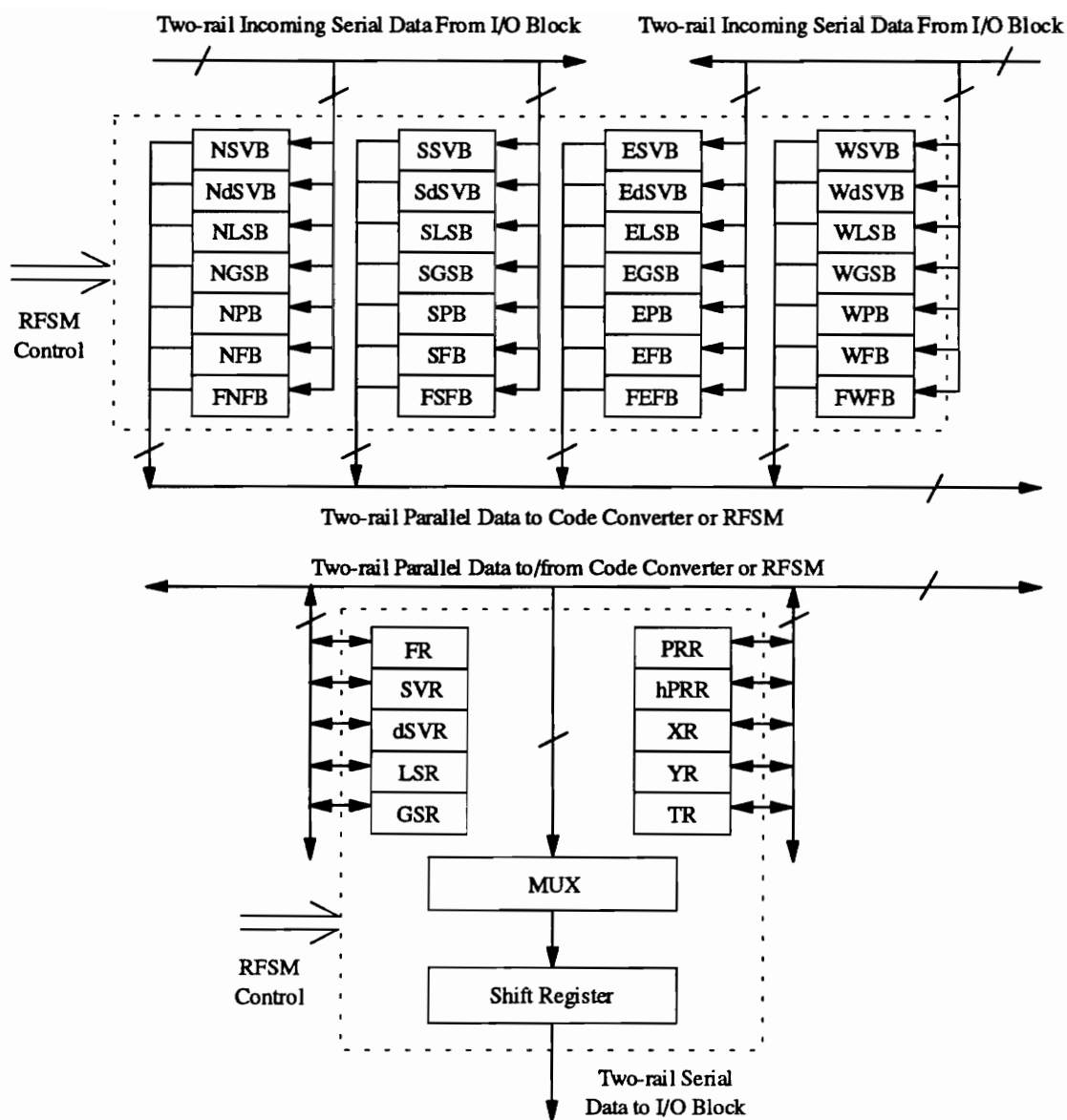


Figure 5.7 - The registers and buffers for the control cell.

data from the I/O block via a right-shift operation. A multiplexer under the control of the RFSM selects one of the registers as the current output, and the shift register serially transmits the data to the I/O block. The multiplexer can be implemented as an SFS PLA. The registers and buffers have a parallel load capability. The RFSM controls the load and shift operations of all of the buffers and registers, and is responsible for updating the contents of the registers. In turn, all of the registers and buffers provide two-rail parallel data as input to the code conversion module.

The registers and buffers needed for global reconfiguration are as follows:

NSVB/ESVB/SSVB/WSVB - These are the s-value buffers. They store s-value data from the North, East, South, and West neighbors, respectively, which indicates the amount of fault-free space surrounding each cell. See [Kum84] for an explanation of how the s-value is determined.

NdSVB/EdSVB/SdSVB/WdSVB - These are the diagonal s-value buffers. They store data from the North, East, South, and West neighbors, respectively, concerning the fault-free diagonal space surrounding each cell [Bri87].

NLSB/ELSB/SLSB/WLSB - These are the local state buffers. They store data from the North, East, South, and West neighbors, respectively, concerning the local states of these four cells, as described in Chapter 2 and [Bri87].

NGSB/EGSB/SGSB/WGSB - These are the global state buffers. They store data from the North, East, South, and West neighbors, respectively, concerning the global states of these four cells, as described in Chapter 2 and [Bri87].

NPB/EPB/SPB/WPB - These are the position buffers. They store data from the North, East, South, and West neighbors, respectively, concerning either the row or column locations of these four cells, as described in Chapter 2 and [Bri87]. Note that only one position buffer per neighbor is needed, since cells pass either the row or column location, but not both, to their neighbors.

SVR - The s-value register for this cell, which contains information concerning the amount of fault-free space surrounding this cell [Bri87].

dSVR - The diagonal s-value register for this cell, which contains information concerning the amount of diagonal fault-free space surrounding this cell [Bri87].

LSR - The local state register for this cell, as described in Chapter 2 and [Bri87].

GSR - The global state register for this cell, as described in Chapter 2 and [Bri87].

PRR - The priority register for this cell, as described in [Bri87]. The global reconfiguration mechanism uses it to determine which cell provides the new seed for control pattern relocation and regrowth.

hPRR - The high-priority register for this cell, as described in [Bri87]. It holds the highest priority value seen by the cell when a new seed must be created for control pattern relocation and regrowth.

XR/YR - The position registers for this cell, as described in Chapter 2 and [Bri87]. XR stores the row coordinate, and YR stores the column coordinate.

TR - The time register for this cell, as described in Chapter 2 and [Bri87]. It is used to mark the occurrence of *bloomtime*.

The registers and buffers anticipated for local reconfiguration and the distributed diagnosis algorithm of Chapter 4 are as follows:

FR - The fault register for this cell, as described in Chapter 4, [Whi91], and [Bri87].

NFB/EFB/SFB/WFB - These are the fault data buffers for the North, East, South, and West neighbors, respectively. They hold copies of the fault register data received from each of these neighbors.

FNFB/FEFB/FSFB/FWFB - These are the fault data buffers for the Far North, Far East, Far South, and Far West neighbors, respectively. They hold copies of the fault register data received from each of these neighbors.

Note that even though White did not discuss the NFB, EFB, SFB, and WFB fault data buffers in [Whi91], they are needed for that local reconfiguration scheme, so that the cell can store and use the fault data from its neighbors. Also, White suggests that one copy of the fault register be used for determining the new fault register contents, and another with identical contents be used for the transmission of fault data to neighboring cells. This assessment is derived from the software simulation of the system. Note that two memory locations are needed to store a register's current-state and its next-state in software. A hardware implementation has no such restriction, so only one fault register is needed.

As shown in Figures 5.4 and 5.6, two buffers can be simultaneously loaded by the incoming serial data from the I/O block, in accordance with the data transmission cycle discussed in Section 5.3.2. Since the RFSM controls the loading of data into the registers, there is no contention for the busses. The contents of a single register can also be sent to the I/O block, where the data can be transmitted to two neighbors simultaneously.

Figure 5.8 shows a single bit-slice for the registers and buffers, taken from [NaK85]. Since the registers and buffers use a two-rail code, two bit-slice registers are needed to store a single bit of two-rail data. Data on the D line is loaded by a pair of two-rail clock signals, labeled L and $/L$ in the diagram. When $Q = 0$, a positive pulse at L loads the data from the D input. When $Q = 1$, a negative pulse at $/L$ loads the data from the D input. This circuit is SFS for any unidirectional or stuck-at faults in the data input and load signals, as well as for unidirectional and stuck-at errors at the output [NaK85]. Consider the case when the signal at $/L$ is fixed and a positive pulse appears at L . This can occur if $/L$ experiences a stuck-at fault, or if an erroneous transient pulse appears at L . If $Q = 1$, then D cannot be loaded, and so the output appears as either the correct value, or as a unidirectional error from 0 to 1. If $Q = 0$, then L causes the D input to be loaded, which may result in $Q = 1$, preventing the bit-slice register from being loaded again. Similarly, if the signal at L remains fixed, and a negative pulse at $/L$ appears, a unidirectional error from 1 to 0 is detected. Note that in the shift mode, unidirectional faults are simply sent from one bit-slice to the next. Thus, the outputs are either the correct codeword, or contain a unidirectional fault. The parallel and serial load paths needed for the registers and buffers can be implemented as an SFS PLA, whose outputs can be monitored by the internal error indicators. If erroneous incoming data from the I/O block is the result of a faulty neighbor, the fault register allows the RFSM to ignore the corrupted data. The registers and buffers are assumed to be resettable under the control of a two-rail control signal from the RFSM. This can be accomplished by allowing the RFSM to write an appropriate two-rail codeword to all of the registers and buffers.

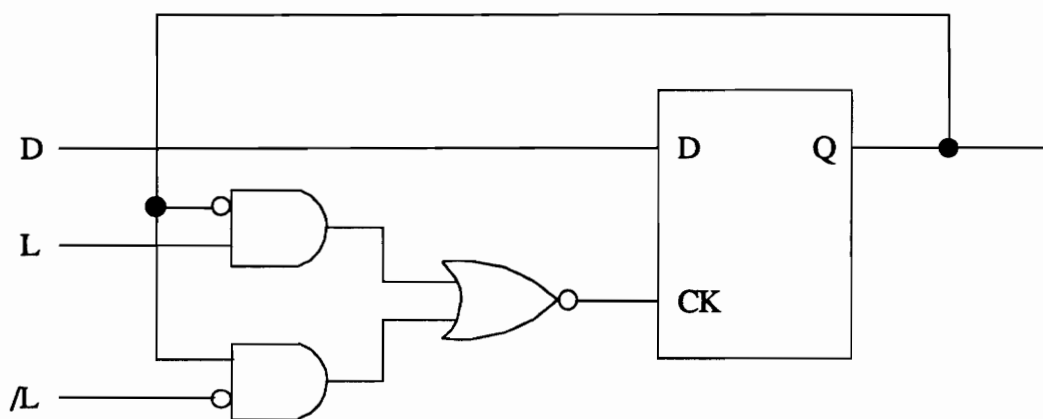


Figure 5.8 - The SFS register bit-slice is fault-secure for unidirectional and stuck-at faults.

5.3.4 THE RFSM

The Reconfiguration Finite State Machine of Figure 5.4 is assumed to have integrated all of the necessary functions for the control cell, including rollback, local, and global reconfiguration. It accepts as inputs the codewords held in the registers and buffers of the system after they have been translated by the code conversion module (see Section 5.3.5 below). The RFSM also generates output codewords which are converted and stored in the registers as needed, as well as the two-rail signals required to control the functions of the I/O circuitry and the registers and buffers.

Figure 5.9 shows a block diagram for a self-checking sequential system which meets the TSC goal [JaC88]. Figure 5.9 appears very similar to the self-checking combinational network shown in Figure 5.1. In the figure, a sequentially self-checking (SeSC) machine accepts sequences of inputs which conform to an *input language*, and it provides sequenced outputs belonging to an *output language*. The outputs of the sequential machine are monitored by a sequential checker circuit, called a *strongly language-disjoint* (SLD) checker. The SeSC and SLD properties are defined later in this section. Rather than performing a combinational checking function, the sequential checker accepts as an input language the output language symbols from the sequential machine. The checker produces an output language which indicates whether the sequence of input symbols supplied by the sequential machine belongs to the checker's input language set. Conveniently, the output language of the checker circuit can be defined as sequences of two-rail codewords if desired. Thus, the outputs of the checker circuit can be monitored by the TSC internal fault indicators.

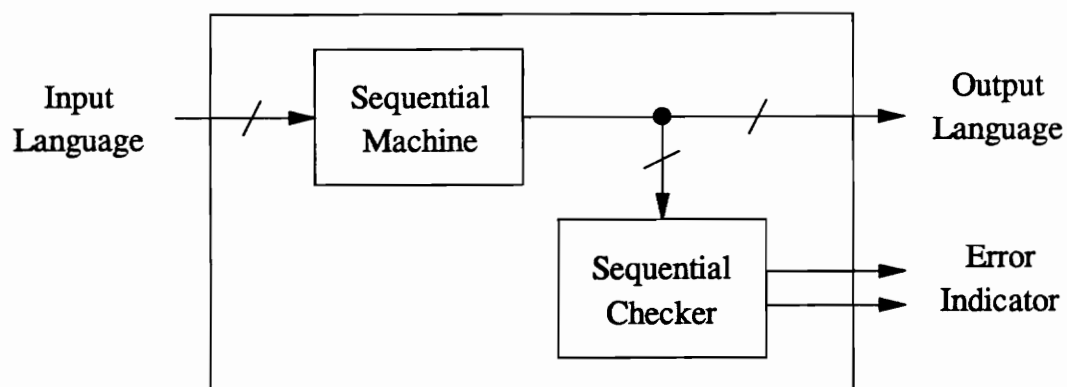


Figure 5.9 - Block diagram of a self-checking sequential system.

5.3.4.1 SEQUENTIALLY SELF-CHECKING (SeSC) CIRCUITS

Just as there are formal definitions for self-checking combinational circuits, so too are there formal definitions for the properties which characterize sequentially self-checking circuits. Formally, a sequential machine M can be defined in terms of the following parameters [ViD80]:

X - the set of valid input symbols

Q - the set of internal states

Z - the set of valid output symbols

δ - the next-state function, and

ω - the output function.

Thus, $M = (X, Q, Z, \delta, \omega)$. When the machine contains a fault f , the resulting faulty machine is given by $M^f = (X, Q^f, Z^f, \delta^f, \omega^f)$. For a given state $q \in Q$, and a given input sequence i , let the next state sequence be $\delta(i, q)$, and let the corresponding output sequence be $\bar{\omega}(i, q)$. If the fault f occurs in the state q of M , the machine immediately assumes the state $q^f \in Q^f$. Thus, $\delta^f(i, q) = \delta^f(i, q^f)$ is the next-state sequence, given the input sequence i , due to the fault f , and $\bar{\omega}^f(i, q) = \bar{\omega}^f(i, q^f)$ is the corresponding output sequence.

Let $i = i_1 \cdot i_2$, called the *concatenation* of i_1 and i_2 , be the input sequence obtained by following the input sequence i_1 by the input sequence i_2 . The sequences i_1 and i_2 are said to be a *suffix* and a *prefix* of i , respectively. Let $P(i)$ and $S(i)$ denote the sets of all prefixes and suffixes of the input sequence i , respectively.

Definition 5.8:

The *input language*, I_M , of a sequential machine, M , is the set of all input sequences i which may be applied from the initial state q_0 in normal operation [ViD80].

Definition 5.9:

The *output language*, O_M , of a sequential machine, M , is the set of all output sequences which may be obtained from the initial state q_0 in normal operation [ViD80]. That is, $O_M = \{\bar{\omega}(i, q_0) : i \in I_M\}$.

Thus, the input and output languages are determined by the expected behavior of the sequential machine, and may be obtained from a state diagram or flow table representation, for example.

Let the set of input sequences i_2 which may be applied to M when it is in state q during normal operation be given by $I_q = \{i_2 : i_1 \cdot i_2 \in I_M, \delta(i_1, q_0) = q\}$. That is, a valid input sequence $i_1 \in I_M$ is applied to M , which is in initial state q_0 . After the first sequence concludes, M is in state q , whereupon any valid sequence i_2 is applied. The input sequences of unbounded length can be defined as $I_q^\infty \subset I_q$.

Definition 5.10:

A circuit M is *sequentially self-testing* (SeST) for a fault f , a state q , and an input sequence $i_2 \in I_q$ iff : $\forall i_1$ such that $\delta(i_1, q_0) = q$ and such that $i_1 \cdot i_2 \in I_M$:

$$\bar{\omega}(i_1, q_0) \cdot \bar{\omega}(i_2, q) \notin O_M \text{ [ViD80].}$$

Definition 5.10 simply states that in response to a fault, a sequentially self-testing circuit can be driven by a particular valid input sequence which eventually produces an invalid output sequence. This is analogous to the self-testing property of combinational circuits as given in Definition 5.3. A definition analogous to the fault-secure property of combinational circuits given in Definition 5.2 also exists for sequential circuits. Let i_{2m} denote the shortest prefix of i_2 such that $\bar{\omega}(i_1, q_0) \cdot \bar{\omega}(i_{2m}, q) \notin O_M$.

Definition 5.11:

A circuit M is *sequentially fault-secure* (SeFS) for a fault f , a state q , and an input sequence $i_2 \in I_q$ iff :

$$\bar{\omega}^f(i_2, q_0) = \bar{\omega}(i_2, q), \quad \forall i_2 \in P(i_{2m}), \quad i_2 \neq i_{2m}, \quad \text{if } i_{2m} \text{ exists, } \forall i_2 \in P(i_2) \text{ otherwise}$$

[ViD80].

Definition 5.11 says that as long as the input sequence produces an output sequence which belongs to the output language set, the output sequence is correct. That is, no incorrect output sequences belonging to the output language are produced.

Definition 5.12:

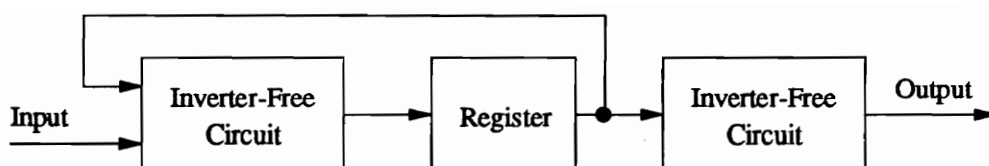
A circuit M is *sequentially self-checking* for a prescribed normal operation and for a fault set F iff, for any $f_1 \in F$, for any $q \in Q$, and for any $i_2 \in I_q^\infty$, either

- a) the circuit is both SeST and SeFS for (f_1, q, i_2) , or
- b) the circuit is only SeFS for (f_1, q, i_2) and, for any $f_2 \in F$, or any $i_4 \in S(i_2)$, either the property a), or the property b) is true for : $f_1 \cup f_2$ taking the

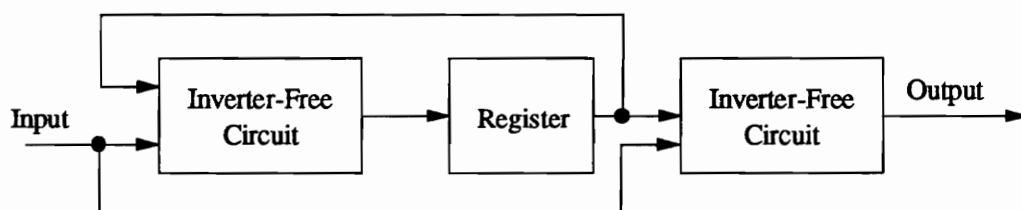
place of f_1 , $\delta^{f_1}(i_3, q)$ such that $i_2 = i_3 \cdot i_4$ taking the place of q , and i_4 taking the place of i_2 [ViD80].

This definition for SeSC circuits is recursive in nature, but is easily explained. If a) is true, then the input sequence i_2 detects the fault sequence by producing an output sequence whose first erroneous value results in an output sequence which does not belong to the output language. If b) is true, then as long as the input sequence does not detect the fault sequence, the machine continues to produce the correct output sequence until a detecting input sequence is applied. Thus, Definition 5.12 insures that a sequential system meets the TSC goal by producing an invalid output sequence as its first erroneous output. Note that if a) is true, the machine is self-testing under the fault sequence, which must be detected before another fault from F occurs.

The sequential machine in Figure 5.9 must be SeSC in order for the system to meet the TSC goal. In [ViD80], it is shown that if a sequential circuit is SFS for a fault set F , then the circuit is also SeSC for F . Figure 5.10 shows the block diagrams of two general sequential circuits proposed in [NaK85] which are SFS, and hence SeSC, for unidirectional faults. Both the Moore and Mealy sequential machine models are shown. Recall that in the Moore model, the output of the machine is a function of its current state only, whereas in the Mealy model, the current inputs also determine the output. The register block is composed of the bit-slice registers given in Figure 5.8, where an unordered code is used to encode the state values. The inverter-free circuits, which provide the next-state and output functions, accept unordered codewords as inputs and produce unordered codewords as outputs. These circuits can be implemented using SFS PLAs.



a) Moore model for the SFS sequential machine.



b) Mealy model for the SFS sequential machine.

Figure 5.10 - Sequentially self-checking machines made from SFS circuits.

5.3.4.2 STRONGLY LANGUAGE-DISJOINT (SLD) CHECKERS

The design proposed in [NaK85] does not include the sequential checker depicted in Figure 5.9, as required for a self-checking sequential system. As long as the machine reaches a valid state, the assumption is made in [NaK85] that the transition which produces the state is valid. Clearly, this is not an acceptable approach if invalid transitions can occur in response to a fault. In [JaC88], Jansch and Courtois define a strongly language-disjoint (SLD) checker which can be used as a checker for the system shown in Figure 5.9. More definitions are required before the SLD checker can be formally defined.

Definition 5.13:

A sequential circuit M is *language-disjoint* if $\forall i \in I_M, \omega(i, q_0) \in O_M$ and $\forall i \notin I_M, \omega(i, q_0) \notin O_M$ [JaC88].

The language-disjoint property for sequential circuits is analogous to the code-disjoint property for combinational circuits. It maps input sequences which are not contained in the input language to output sequences which are not contained in the output language. Recall that TSC checkers for combinational circuits require the code-disjoint property in addition to the self-testing and fault-secure properties. The following definition gives an analogous property for sequential machines.

Definition 5.14:

A sequential circuit M is a sequentially self-checking checker if it is both sequentially self-checking and language-disjoint [JaC88].

Note that the language-disjoint property does not impose restrictions on the ability of a faulty circuit to maintain that property. It is desirable for the checker circuit to maintain the language-disjoint property in the presence of faults. Hence, the need for the SLD checker. The definition for the SLD checker follows from the concept of the redundancy of a sequential circuit.

Definition 5.15:

A sequential checker C is *redundant* with respect to a fault f , and with respect to an input language I_C and with respect to an output language O_C if $\forall i \in I_C, \forall i_1 \in P(i), \omega_C(i_1, q_0) \cdot \omega'_C(i_2, q') \in O_C$, where $i = i_1 \cdot i_2$, and $q' = \delta'(i_1, q_0)$, and $\forall i \notin I_C, \forall i_1 \in P(i), \omega_C(i_1, q_0) \cdot \omega'_C(i_2, q') \notin O_C$ [JaC88].

Simply stated, Definition 5.15 says that in the presence of the fault f , the redundancy property of the checker insures that the language-disjoint property is maintained. This allows the faulty checker to produce invalid output sequences when the sequential circuit being checked produces invalid output sequences.

Definition 5.16:

A sequential checker is *strongly redundant* with respect to the fault sequence $\langle f_1, f_2, \dots, f_n \rangle$, and with respect to an input language I_C and with respect to an output language O_C if it is redundant with respect to the n fault subsequences $\langle f_1 \rangle, \langle f_1, f_2 \rangle, \langle f_1, f_2, f_3 \rangle, \dots, \langle f_1, f_2, \dots, f_n \rangle$ and with respect to the input language I_C , and with respect to the output language O_C [JaC88].

A strongly redundant sequential checker maintains the language-disjoint property, even in the presence of the sequence of faults. This is a desirable property, since undetected faults cannot prevent the checker from maintaining the language-disjoint property.

Definition 5.17:

Before the occurrence of any fault, the checker C is language disjoint. For a fault sequence $\langle f_1, f_2, \dots, f_n \rangle$, a state q' , and an input sequence $i_2 \in I_C$, let k be the smallest integer for which $\delta(i_1, q_o) = q'$ and such that

$$i_1 \cdot i_2 \in I_C : \omega_c(i_1, q_o) \cdot \omega_c^{(f_1, f_2, \dots, f_k)}(i_2, q') \in O_C.$$

If there is no such k , let $k = n + 1$. Then C is strongly language disjoint with respect to the fault sequence if

$$\forall i \notin I_C, \forall m \in \{1, 2, \dots, k-1\}, \omega_c(i_1, q_o) \cdot \omega_c^{(f_1, f_2, \dots, f_m)}(i_2, q') \notin O_C \text{ [JaC88]}.$$

The above definition states that if the shortest fault subsequence for which the checker maps valid input sequences to invalid output sequences is known, then the checker is SLD for the subsequence if, for any sub-subsequence of the subsequence, the checker maintains the language disjoint property.

Definition 5.18:

The checker C is strongly language disjoint with respect to the fault set F if C is SLD with respect to all fault sequences whose members belong to F [JaC88].

Thus, a checker which is SLD for a given fault set F maintains the language disjoint property for any sequence of faults in F . This is analogous to the strongly code-disjoint property for combinational circuits, as given in Definition 5.7.

Figure 5.11 gives the block diagram for an SLD checker proposed in [JaC85]. The external input sequence is supplied by the sequential machine under test, which is the functional part of the RFSM in this case. The code B is defined as the concatenation of the external input codewords and the state code, and is indicated by an ellipse associated with the SFS / SCD combinational logic block. A codeword of B is only defined if the external input codeword is valid under a given state codeword. If the SLD checker is currently in a valid state q , and the input vector x is a valid input codeword, then if a valid next-state mapping exists from q to q' under the input x , the concatenation of q and x is a valid codeword in B . In effect, x belongs to a valid input sequence when the checker is in state q . However, if there is no valid next-state mapping from q to any other state when x is the applied input vector, the concatenation of q and x is not a codeword in B , even if q and x are codewords in the next-state and input codes, respectively.

The SFS / SCD combinational logic block provides the next-state and output functions, and can be implemented with PLAs and some discrete gates, as given in [PaS91]. The output function of the block maps input codewords in B to output codewords in the code O_C . The next-state function of the block maps input codewords in B to output codewords in the next-state code C . The SFS property of the combinational block insures that for a codeword input, only correct codeword outputs are produced, even in the presence of a fault sequence. The SCD property of the combinational block

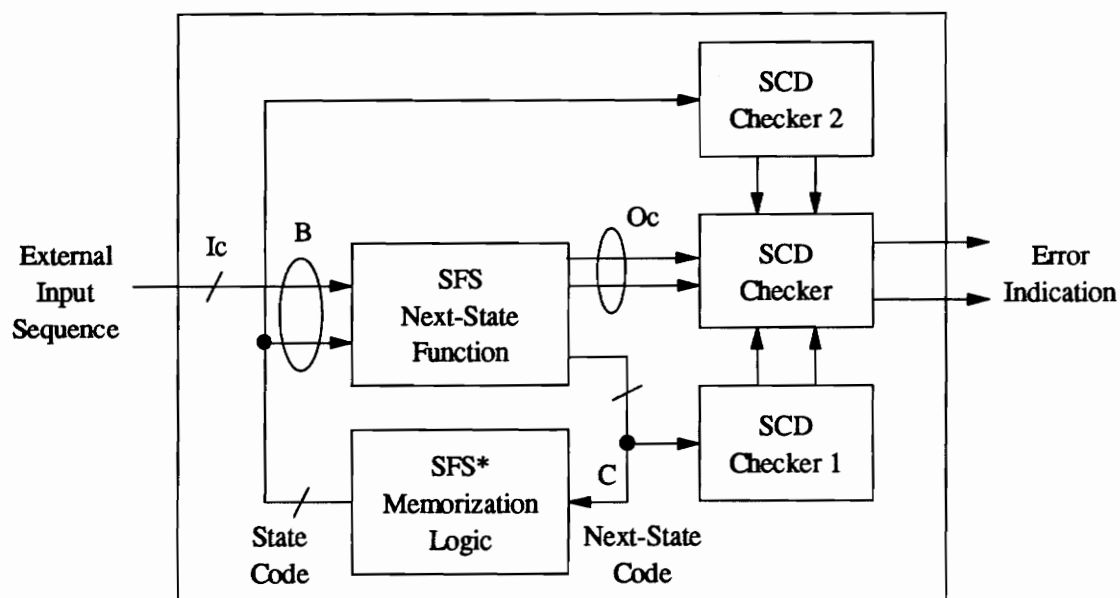


Figure 5.11 - Block diagram of a proposed SLD checker.

insures that the presence of a fault sequence, non-codeword inputs are mapped to non-codeword outputs.

The SFS* memorization logic described in [JaC85] and [JaC88] has special properties in addition to the SFS properties needed for combinational logic. In particular, it is assumed that the circuit outputs do not change until a clock signal is generated, and that when a value is loaded, it is either the correct codeword or a non-codeword, in accordance with the SFS property. A further assumption is that the circuit is SFS in the presence of faulty clock lines. If the stable output assumption is granted, then the SFS bit-slice register given in Figure 5. 8 is sufficient for this purpose.

There are three SCD checkers in Figure 5.11. SCD checker 1 monitors the code C, and SCD checker 2 monitors the code B. The third SCD checker accepts inputs from the other two SCD checkers, as well as from the output function of the SFS / SCD logic block. Its output is two-rail encoded, and is monitored by the TSC internal fault indicators. SCD functional circuits are useful in the construction of systems which meet the TSC goal, but SCD checkers may be inappropriate in such systems. In [Jha88] and [Jha90], it is pointed out that SCD checkers whose outputs are not directly observable, but are instead checked by other checkers, may not meet the TSC goal. This is due to the fact that the lack of the fault-secure property in SCD checkers may prevent the final checker from being adequately exercised to expose faults in the system. Such a case could arise in any of the SCD checkers in Figure 5.11, since their outputs are not directly observable; recall that the error signal is monitored by the TSC internal fault indicators. To guard against this problem, Jha introduces the strongly self-checking (SSC) property for checkers in [Jha88], which combines the SFS and SCD properties.

Definition 5.19:

A circuit is *strongly self-checking* with respect to a set of faults F if, before the occurrence of any fault, the circuit is code-disjoint, and for any fault $f \in F$, either

- a) the circuit is self-testing, or
- b) the circuit is fault-secure and always maps non-codewords at its inputs to non-codewords at its outputs, and if another fault from F occurs in the circuit, then either a) or b) is true for the fault sequence [JhK89].

SSC circuits can be used for both functional circuits and for checkers. In the case that an SSC checker can not be sufficiently exercised by the available codewords to make it self-testing, Jha proposes a weaker property than the SSC property, called the sufficiently strongly self-checking (SSSC) property.

Definition 5.20:

A circuit is *sufficiently strongly self-checking* with respect to a set of faults F if, before the occurrence of any fault, the circuit is code-disjoint, and for any fault $f \in F$, either

- a) the circuit is self-testing, or
- b) the circuit is fault-secure and always maps non-codewords at its inputs to non-codewords at its outputs [JhK89].

The SCD checker blocks of Figure 5.11 should be replaced by TSC or SSC checkers to insure that both the code-disjoint and fault-secure properties are maintained if the checkers are sufficiently exercised to expose faults. This arrangement is shown in Figure 5.12. If SSSC checkers are used instead, periodic off-line testing may be needed to

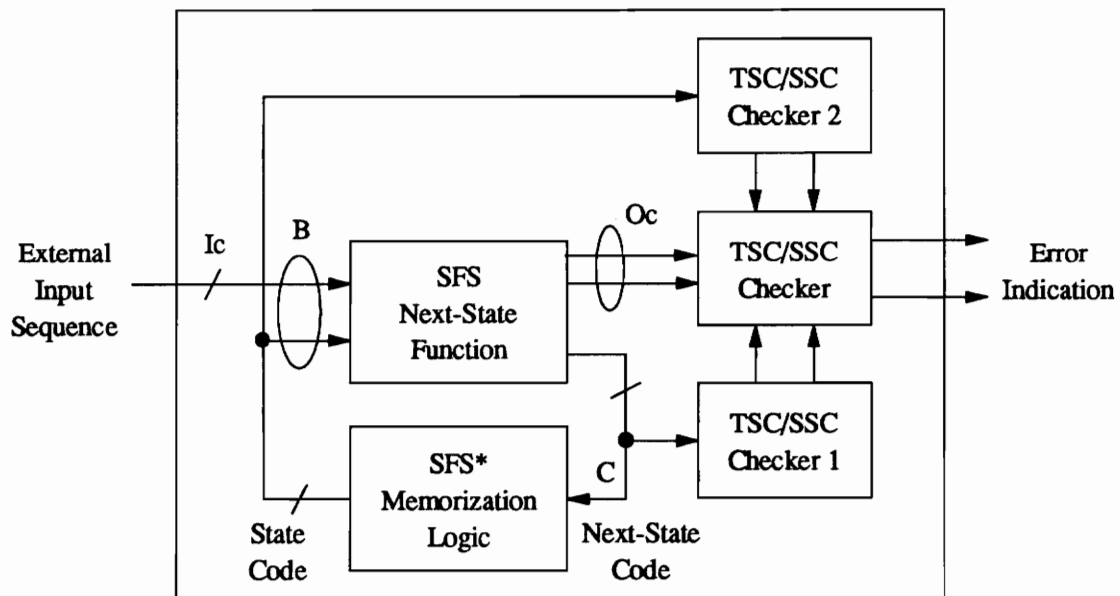


Figure 5.12 - Improved SLD checker design which eliminates SCD checkers.

fully exercise the checkers for faults [Jha90]. TSC and SSC checker implementations are given in [Jha88], and SSSC checker designs are given in [Jha89].

5.3.5 THE CODE CONVERSION MODULE

As shown in Figure 5.4, this functional block is located between the block of registers and buffers and the RFSM. The block converts data encoded in the two-rail code to a code with smaller codewords, and vice-versa. The lower cost code can be any unordered code for which a TSC checker can be implemented, such as a Berger code or a Bose-Lin code. The block is optional, but may reduce the hardware overhead of the cell, provided that the cost of implementing the code conversion block offsets the cost of the additional hardware which would otherwise be required by the RFSM to accommodate all of the two-rail data from the buffers and registers. The two-rail inputs and outputs of the code conversion module are monitored directly by the TSC internal error indicators. Those inputs and outputs encoded using the lower cost unordered code are monitored by a TSC checker for that code, which is in turn monitored by the internal error indicators. An SFS PLA can be used to implement the code conversion module.

5.3.6 THE TSC CHECKER FOR THE CODE CONVERSION MODULE

This TSC checker monitors the inputs and outputs of the code conversion module which are not encoded in a two-rail code, as described in Section 5.3.6. The monitored inputs and outputs are encoded using an unordered code with a smaller codeword size than the two-rail code. The checker provides a two-rail output which is monitored by the internal error indicators. Implementations of TSC checkers for Berger codes and modified

Berger codes are discussed in [Jha90] and [AsR77]. Implementations of TSC checkers for Bose-Lin and other unordered codes are given in [Jha91].

5.3.7 THE INTERNAL FAULT INDICATORS

The 12 internal fault indicators can be implemented as two-rail TSC checker trees, as explained in Section 5.2.2.3. Each of these two-rail TSC checkers sends its two-rail output pair to a single neighbor, as shown in Figure 5.3. Each neighbor feeds the received two-rail signal into the error trap circuit of Figure 5.5, whose output allows the neighbor to set its fault register accordingly. As noted in Section 5., neighboring cells cannot share these fault indicator lines as a bidirectional data path. This is due to the feedback loop which would be created between the error trap circuitry and the internal fault indicators.

5.4 RESULTS

The proposed self-checking control cell architecture combines many different techniques proposed for the design of systems which meet the TSC goal. The anticipated needs of the system are incorporated into the design, based on the information currently available concerning the global and local reconfiguration strategies. The proposed error-trap circuitry prevents corrupted incoming data from being used by a fault-free cell, and allows concurrent detection of faulty cells in the array by neighboring cells. The I/O circuitry provides bidirectional serial communications with neighboring cells, and is flexible enough to support the anticipated needs of the local reconfiguration algorithm, as well as the distributed diagnosis algorithm of Chapter 4. If implemented as a strongly fault-secure PLA, this block meets the TSC goal. The SFS two-rail registers and buffers

provide storage for local as well as incoming data. The sequentially self-checking functional part of the RFSM incorporates the SFS bit-slice register and SFS PLAs. The strongly language-disjoint checker, implied in the RFSM block of Figure 5.4, uses the SFS bit-slice register, self-checking checkers, and combinational logic which is both strongly code-disjoint and SFS to detect undesirable behavior in the RFSM; Appropriate implementations for all of these blocks exist, as discussed in this chapter. The code conversion module, which converts the data stored in the two-rail registers to a lower-cost code for use by the RFSM, and vice versa, is optional, and may be implemented using an SFS PLA. Depending on the chosen implementation of the control cell, the conversion module may substantially reduce the hardware overhead of the cell by decreasing the number of inputs needed by the RFSM. A TSC checker with a two-rail output is needed to monitor the data encoded in the lower-cost code, and so must be included as overhead. Finally, a set of 12 two-rail TSC checkers monitors all of the subsystem interfaces, so that neighboring cells are quickly informed of any faults in the system.

Estimating the cost of implementation for the proposed self-checking control cell architecture is extremely difficult at present. The rollback strategy, the distributed diagnosis strategy, and the local and global reconfiguration algorithms need to be integrated before implementation costs can be realistically evaluated. During the course of integrating these schemes, changes may be made to any or all of them which may cause the system specification to differ somewhat from the design proposed in this chapter. However, it is possible to make certain generalizations about the costs of the proposed architecture, which should hold for any implementation using the general scheme presented in this chapter.

The use of encoded inputs and outputs also adds to the cost of the circuit. In the case of the two-rail code, the registers, buffers, links, and busses must be fully duplicated. In addition, each of the bit-slice registers requires 3 gates and a D flip-flop. For an n -bit register without two-rail encoding, the cost is n flip-flops. The cost for the same register using a two-rail code and the SFS bit-slice design is $6n$ gates and $2n$ flip-flops. The use of a Lawson neighborhood to pass fault data adds 4 buffers to the design, as well as the additional combinational logic in the RFSM needed to set the fault register.

The cost of the error trap circuits is relatively small. There are two gates per block, two blocks per neighbor, and 12 neighbors per cell. Thus, the error trap circuitry requires 48 gates per cell. Note that in some implementations, the transistor count and general layout of the XOR and XNOR gates used in the error-trap circuits is exactly the same. The only difference is found in routing the various input signals to the appropriate points in the circuit.

The use of PLAs to implement the various SFS and SCD combinational logic blocks in the circuit guarantees that the TSC goal is met by these circuits. PLAs also aid in automating the design and layout processes, due to their regular structure. However, SFS and SCD circuits which do not require a PLA-based implementation may exist for some of these blocks, once the exact behavior is known. Such circuits should be considered wherever possible, since the PLAs may require substantially greater layout areas.

Two-rail TSC checker trees take the form of inverted full binary trees. For a 2^q -variable two-rail TSC checker tree, it can be shown from properties of binary trees that

the checker tree has q levels of cascaded 2-variable two-rail TSC checkers, and that $(2^q - 1)$ checkers are needed to construct the checker tree. Since there are 6 gates per 2-variable TSC checker, $6(2^q - 1)$ gates are needed. Faster, lower cost implementations for two-rail TSC checkers exist, but they do not enjoy the regularity of layout and easy testability of the TSC checker tree [Jha90].

The cost of the RFSM can be reduced if the SLD checker is eliminated from the system. One of the sequentially self-checking machines of Figure 5.10 can be used alone. However, the ability to detect valid next-state transitions is lost. The only checking that can take place is for a valid state, which is still an improvement over a system with no such capability.

The SFS implementation of the i8080 processor presented in [NaK85] makes extensive use of SFS PLAs, and uses the SeSC machine of Figure 5.10. It is similar to the proposed control cell architecture in many other respects, including the use of two-rail encoding for the busses and internal ALU registers, as well as the use of a code conversion module and TSC checkers. However, no SLD checker is used, and the outputs of the 4 TSC checkers in the system are directly observable by the operator. Assuming a microprogrammed architecture, the [NaK85] estimates that a 38% increase in the hardware is required for the system to be made SFS. This figure may increase somewhat for the self-checking control cell, since the SLD checker and additional checker modules are used. However, the similarity in the structure of the two systems suggests a reasonable similarity in the respective costs.

5.5 CONCLUSIONS

Several classes of self-checking combinational and sequential circuits have been examined in this chapter. If unidirectional faults are the assumed fault set, encoding the inputs and outputs of the self-checking circuits in an unordered code allows the system to achieve the TSC goal for unidirectional faults. That is, the first erroneous output produced by the system is a non-codeword. Some examples of unordered codes are the Berger codes, the Bose-Lin codes, and the two-rail code. TSC checkers for all of these codes are presented in [Jha90].

The proposed self-checking control cell architecture presented in this chapter meets the self-checking requirements of the distributed diagnosis algorithm of Chapter 4 for unidirectional faults. It utilizes many different approaches previously used in the design of self-checking systems. When this control cell architecture is combined with the Lawson neighborhood assumed for the distributed diagnosis algorithm, the system has a relatively low fault latency parameter. This allows for rapid detection and isolation of faulty cells in the array, whereupon the local reconfiguration strategy can be applied as needed.

The proposed error trap circuitry is completely new, as far as is known. It allows a non-codeword two-rail signal to be mapped to a two-rail codeword for use by other circuits in a self-checking system. TSC checkers are not applicable to this problem, as they map non-codeword inputs to non-codeword outputs. Thus, the output of a TSC checker would simply propagate the effect of the non-codeword to other parts of the system.

Note that the error-trap circuits monitor the bidirectional data link constantly, so that faults which appear at the links when the cell is transmitting data are detectable. Since the RFSM controls the transmission sequence, it can distinguish faults in the incoming data from faults in the outgoing data. A fault-free cell sharing the link with the faulty cell will detect the fault as an external error. However, an extra bit in the fault register could also allow the faulty cell to detect its own fault in this case. As long as the cell transmits valid codewords to at least one other neighbor on a different link, this other neighbor could be made aware of the link fault, which could conceivably reduce the fault latency time.

Certain assumptions are made in [JaC85] and [JaC88] concerning the presence of faults in the SeSC system of Figure 5.9. Specifically, between the occurrence of any two faults in the sequential machine, all codewords which can detect the current fault are assumed to be applied to the inputs of the machine, and no faults can be present in the SLD checker. That is, a valid input sequence is applied to the machine which causes the fault to be detected. Similarly, between the occurrence of any two faults in the SLD checker, all codewords in B which can detect the current fault are applied to the inputs of the checker, and the sequential machine must be fault-free. These assumptions are similar to the assumptions given in Section 5.1.1 for self-checking combinational circuits. If these assumptions are unacceptable, however, Jansch and Courtois define the very strongly language-disjoint (VSLD) property, which differs slightly from Definition 5.17. The VSLD checker requires that for a valid input sequence, the shortest fault sequence which causes the output sequence to be invalid does not cause the faulty circuit to lose the language-disjoint property. Under this definition, a fault in the sequential machine can be followed by a fault in the checker before the fault in the sequential machine is detected.

Similarly, a fault in the checker can be followed by a fault in the sequential machine before the fault in the checker is detected. Unfortunately, Jansch and Courtois do not offer any solutions for building a VSLD checker. If the self-checking system with an SLD checker cannot be sufficiently exercised to meet the assumptions made by Jansch and Courtois, the SSSC property discussed in [Jha89] may be more applicable. This question cannot be answered until the behavior of the control cell is fully specified.

6. CONCLUSIONS

This thesis has investigated many issues related to the global and local reconfiguration of fault-tolerant cellular processing arrays. The results presented in this work are all necessary for insuring a robust fault-tolerant system. These results provide part of the platform needed to integrate the available reconfiguration strategies into a unified system. However, many challenging problems remain to be investigated in future research efforts.

6.1 SUMMARY OF ACCOMPLISHMENTS

Chapter 2 demonstrated the need for a system which addresses reconfiguration for yield enhancement, run-time fault tolerance, and array polymorphism. Furthermore, such a system should not assume the existence of components which never fail, such as a global control mechanism or communications links. These hard core components limit the useful application of the system, since the failure of these components can result in the failure of the system. To address these issues, a hierarchical reconfiguration strategy was proposed which included the global reconfiguration algorithms developed in [Kum84] and [Bri87], as well as the local reconfiguration algorithms proposed in [Whi91]. No hard core components were assumed, and the previously developed distributed algorithms for reconfiguration were proposed for eliminating the need for a global control mechanism.

Chapter 3 presented an analysis of the relative complexities of the hardware needed to support the pattern growth algorithms of [Mar80], [Kum84], and [Bri87]. The latter algorithm was shown to provide an exponential savings in the size of the local

memory table needed to perform the next-state function over the other two approaches. The algorithm of [Bri87] also achieves a small savings in the amount of data which must be transferred between neighboring cells, as compared to [Kum84]. Even though [Bri87] requires that a greater amount of data be transmitted between neighboring cells than [Mar80], a substantial savings in the number of interconnection wires is realized by the former method over the latter. These factors indicate that the pattern growth algorithms presented in [Bri87] represent the best choice among the currently available options.

Chapter 4 supplanted the zero fault latency model assumed in all of the related research to date with a distributed diagnosis scheme for notifying cells of faults in the array. The fault register concept used in [Bri87] and [Whi91] was extended to support this algorithm. It was shown that the size and shape of the area represented by the flags in the fault register has a direct impact on the fault patterns which can be properly detected. It was also shown that the neighborhood used to communicate the locations of faulty cells in the array influences the fault patterns which can be properly detected, as well as the amount of time needed to inform cells of the fault. The neighborhood used to distribute the fault data also influences the amount of hardware needed to support the communications network and fault diagnosis algorithm. Simulation results for up to quadruple faults were given for the von Neumann, Moore, Lawson, and White neighborhoods. Some additional hardware in the form of buffers and combinational logic was needed to support the Lawson neighborhood in favor of the von Neumann neighborhood. However, it was shown that the Lawson neighborhood offered a very small fault latency period with respect to the four neighborhoods considered. Despite the fact that the use of the White neighborhood offered an incrementally lower fault latency period than the Lawson neighborhood, the substantial increase in the hardware overhead

cost over the Lawson neighborhood did not appear justifiable. Moreover, only the Lawson and White neighborhoods provided proper detection of all quadruple fault patterns. Thus, the Lawson neighborhood offered the best compromise between the opposing goals of zero fault latency and zero hardware overhead. When combined with the improved local reconfiguration algorithm under development by Joseph Wegner, the system tolerates all triple faults.

Chapter 5 proposed an architecture for a self-checking control cell. The control cell can meet the TSC goal for unidirectional faults by insuring that each of the subsystems meets the TSC goal. Data and control signals were encoded to enable the detection of unidirectional faults. Bidirectional data communications channels and a set of registers and buffers were presented which can be made to support the present global and local reconfiguration algorithms. Error trap circuitry was introduced for insuring that corrupted data from neighboring cells does not give a false error indication in a fault-free cell. A self-checking sequential machine was discussed which detects invalid next-state transitions in addition to reporting invalid current states and input codes. It was modeled after the Reconfiguration Finite State Machine proposed in [Kum84], and further investigated in [Bri87]. A code conversion module and its accompanying TSC checker were included as a possible means to reduce the hardware required by the circuit. A set of TSC checkers was included to monitor the subsystem interfaces and report internal faults to neighboring cells. The resulting system meets the self-checking assumptions made in Chapter 4, hence the distributed diagnosis algorithm can be used by the self-checking control cell architecture.

6.2 FUTURE RESEARCH OPPORTUNITIES

Upon completion, the improved local reconfiguration strategy should be integrated with the global reconfiguration strategy of [Bri87], so that the costs involved in implementing the self-checking control cell proposed in Chapter 5 can be evaluated. It is worthwhile to investigate techniques for implementing self-checking computational cells, as well as techniques for reconfiguring the system in the presence of either faulty control cells or faulty computational cells. Naturally, the functionality of the computational cell influences the methods used to meet the TSC goal. It is therefore desirable to investigate applications which can benefit from the proposed system, and to tailor the computational systems to those applications.

A rollback capability should also be included as a first line of defense against transient faults, since these faults occur more frequently than permanent or semi-permanent faults [Jha93]. If rollback and recomputation fails, a permanent or semi-permanent fault is present, and so the local reconfiguration algorithm should be invoked. If unsuccessful, the global reconfiguration strategy can be used to activate another area of the array before computation is resumed.

BIBLIOGRAPHY

- [AnM73] Douglas A. Anderson and Gernot Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers*, Vol. C-22, No. 3, March, 1973, pp. 263-269.
- [AsR77] Mohammad Ashjaee and Sudhakar M. Reddy, "On Totally Self-Checking Checkers for Separable Codes," *IEEE Transactions on Computers*, Vol. C-26, No. 8, August, 1977, pp. 737-744.
- [BeB92] K. P. Belkhale and P. Banerjee, "Reconfiguration strategies for VLSI processor arrays and trees using a modified Diogenes approach," *IEEE Transactions on Computers*, Vol. 41, No. 1, January 1992, pp. 83-96.
- [Bri87] B. A. Brighton, *Improved Pattern Growth and Reconfiguration Methods for a Fault Tolerant Cellular Architecture*, M. S. Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1987.
- [ChF89] M. Chean and J. A. B. Fortes, "FUSS: A Reconfiguration Scheme for Fault Tolerant Processor Arrays," *Abstracts from the International Workshop on Hardware Fault Tolerance in Multiprocessors*, June 19-20, 1989, pp. 30-32.
- [ChF90] Mengly Chean and Jose A. B. Fortes, "A taxonomy of reconfiguration techniques for fault-tolerant processor arrays," *Computer*, Vol. 23, No. 1, January 1990, pp. 55-69.

- [DiS89] F. Distanté, M. G. Sami, R. Stefanelli, "Reconfiguration Techniques in the Presence of Faulty Interconnections," *Proceedings of the International Conference on Wafer Scale Integration*, San Francisco, California, January 3-5, 1989, pp. 379-388.
- [GoG84] N. Gollakota and F. G. Gray, "Reconfigurable Cellular Architecture," *1984 International Conference on Parallel Processing*, August 21-24, 1984, Bellaire, Michigan, pp. 377-379.
- [GrW89] F. G. Gray and T. S. White, "Summary of a Distributed Control Algorithm for a Dynamically Reconfigurable Array Architecture," *Proceedings of the International Conference on Wafer Scale Integration*, San Francisco, California, January 3-5, 1989, pp. 131-140.
- [Hos89] S. H. Hosseini, "On Fault-Tolerant Structure, Distributed Fault-Diagnosis, Reconfiguration, and Recovery of the Array Processors," *IEEE Transactions on Computers*, Vol. 38, pp. 932-942, July 1989.
- [Jac85] Ingrid Jansch and Bernard Courtois, "Strongly Language Disjoint Checkers," *Fifteenth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers*, Ann Arbor, Michigan, June 19-21, 1985, pp. 390-395.
- [JaC88] Ingrid Jansch and Bernard Courtois, "Definition and Design of Strongly Language Disjoint Checkers," *IEEE Transactions on Computers*, Vol. 37, No. 6, June 1988, pp. 745-748.
- [Jha88] Niraj K. Jha, "SFS/SSC Domino-CMOS Implementations of TSC Circuits," *Proceedings of the Twenty-Sixth Annual Allerton Conference on Communications, Control, and Computing*, Monticello, Illinois, September 28-30, 1988, pp. 768-777.

- [Jha89] Niraj K. Jha, "Design of Sufficiently Strongly Self-Checking Embedded Checkers for Systematic and Separable Codes," *Proceedings of the 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, Massachusetts, October 2-4, 1989, pp. 120-123.
- [Jha90] Niraj K. Jha, "Strongly Fault-Secure and Strongly Self-Checking Domino-CMOS Implementations of Totally Self-Checking Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. 9, No. 3, March 1990, pp. 332-336.
- [Jha91] Niraj K. Jha, "Totally Self-Checking Checker Designs for Bose-Lin, Bose, and Blaum Codes," *IEEE Transactions on Computer-Aided Design*, Vol. 10, No. 1, January, 1991, pp. 136-143.
- [Jha93] Niraj K. Jha, Personal communication, January 3, 1993.
- [JhK90] Niraj K. Jha and Sandip Kundu, *Testing and Reliable Design of CMOS Circuits*, Kluwer Academic Publishers, 1990, pp. 177-221.
- [Joh89] Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, Reading, Massachusetts, 1982, pp. 49, 404-421.
- [KiR89] Jung Hwan Kim and Sudhakar M. Reddy, "On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement," *IEEE Transactions on Computers*, Vol. 38, No. 4, April 1989, pp. 515-525.
- [KuF87] S. Y. Kuo and W. K. Fuchs, "Efficient spare allocation for reconfigurable arrays," *IEEE Design and Test of Computers*, Vol. 4, No. 1, February 1987, pp. 24-31.

- [Kum84] R. Kumar, *A Fault-Tolerant Cellular Architecture*, Ph. D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1984.

- [Kun82] H. T. Kung, "Why systolic architectures?", *Computer*, Vol. 15, No. 1, January 1982, pp. 37-45.

- [LiE89] M. J. Little, R. D. Etchells, J. Grinberg, S. P. Laub, J. G. Nash, and M. W. Yung, "The 3-D Computer," *Proceedings of the International Conference on Wafer Scale Integration*, San Francisco, California, January 3-5, 1989, pp. 55-64.

- [MaA82] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *Twelfth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers*, Santa Monica, California, June 22-24, 1982, pp. 303-310.

- [Mag73] G. Mago, "Monotone Functions in Sequential Circuits," *IEEE Transactions on Computers*, Vol. C-22, No. 10, pp. 928-933.

- [MaG80] H. L. Martin and F. G. Gray, "A Two-Dimensional Self-Reconfigurable Tessellation Automaton," *Proceedings of Southeastcon '80*, Nashville, Tennessee, April 2-5, 1980, pp. 91-94.

- [Mar80] H. L. Martin, *A Self-Reconfigurable Cellular Structure*, Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1980.

- [NaK85] Takashi Nanya and Toshiaki Kawamura, "Error Secure/Propagating Concept and its Application to The Design of Strongly Fault Secure Processors," *Fifteenth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers*, Ann Arbor, Michigan, June 19-21, 1985, pp. 396-401.
- [PaS91] Sandeep Pagey, S. D. Sherlekar, G. Venkatesh, "A Methodology for the Design of SFS/SCD Circuits for a Class of Unordered Codes," *Journal of Electronic Testing: Theory and Applications*, Vol. 2, No. 3, August 1991, pp. 261-277.
- [Ros83] Arnold L. Rosenberg, "The Diogenes approach to testable fault-tolerant arrays of processors," *IEEE Transactions on Computers*, Vol. C-32, No. 10, October 1983, pp. 902-910.
- [SaS86] Mariagiovanni Sami and Renato Stefanelli, "Reconfigurable architectures for VLSI processing arrays," *Proceedings of the IEEE*, Vol. 74, No. 5, May 1986, pp. 712-722.
- [Sin88] Adit D. Singh, "Interstitial redundancy: an area efficient fault tolerance scheme for large area VLSI processor arrays," *IEEE Transactions on Computers*, Vol. 37, No. 11, November 1988, pp. 1398-1410.
- [SmL83] James E. Smith and Paklin Lam, "A Theory of Totally Self-Checking System Design," *IEEE Transactions on Computers*, Vol. C-32, No. 9, September, 1983, pp. 831-844.
- [SmM78] James E. Smith and Gernot Metze, "Strongly Fault Secure Logic Networks," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 491-499.

- [Sny82] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, Vol. 15, January 1982, pp. 47-55.
- [ThW77] R. A. Thompson, S. M. Walters, and F. G. Gray, "Stability in a Class of Tessellation Automata," *Proceedings of the Ninth Annual Southeastern Symposium on Systems Theory*, March 1977, pp. 404-414.
- [ViD80] Jacques Viaud and Rene' David, "Sequentially Self-Checking Circuits," *Tenth Annual International Symposium on Fault Tolerant Computing, Digest of Papers*, Kyoto, Japan, October 1-3, 1980, pp. 263-268.
- [WaA79] Shean Lin Wang and Algirdas Avizienis, "The Design of Totally Self-Checking Circuits Using Programmable Logic Arrays," *Ninth Annual International Symposium on Fault-Tolerant Computing*, Madison, Wisconsin, June 20-22, 1979, pp. 173-180.
- [Whi91] T. S. White, *Distributed Control Reconfiguration Algorithms for 2-dimensional Mesh Architectures*, Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1991.

APPENDIX A. SIMULATION SOURCE CODE

A.1 ARRAY.C

```
/*
  Filename:   ARRAY.C
  Programmer: Shannon Lawson
  Date:      October, 1992.

  ARRAY.C - This file uses the lookup table in FAULTMAP.C for setting the
  fault register flags in an array of control cells. See that file for
  a description of the lookup mechanism.

  The calling syntax is

      array [v|m|l|w] filespec

  where [v|m|l|w] specifies the neighborhood to use for passing fault data:
  von Neumann, Moore, Lawson, or White, respectively. The filespec parameter
  specifies the path of a text file containing the locations of faulty cells
  in the array. The upper left cell in the array has coordinates (0,0), and
  the lower right cell has coordinates (9,9). Each line in the text file
  simply contains the coordinates of a faulty cell as follows:

      row column

  The main program performs a simulation of the distributed diagnosis
  algorithm for up to quadruple fault patterns in a 10-by-10 array. All
  cells are initialized as fault-free, then the locations of the faulty
  cells are read from the user-specified fault data file. Each faulty
  cell is assumed to be detected by one neighboring fault free cell, of
  which up to 4 may exist. All possible detection combinations are
  simulated, and the simulator produces output reflecting the state of
  the array at each time step of the diagnosis. Each cell is represented
  in the output as follows:

      Xn - Faulty cell n
      Q  - Quarantine cell (adjacent to faulty cell)
      n  - Cell is aware of fault n

  5 fields are allowed per cell. Unused fields are filled with '-'
  characters. Thus, a cell marked '-----' is a fault free cell with no
  information pertaining to faults in the array. A cell marked 'Q1-3-' is
  a quarantine cell which is adjacent to either fault 1 or fault 3 (or both)
  and is aware of the faults in both cells. A cell marked '--23-' is aware
  of faults 2 and 3, but is not adjacent to either one.

  Each detection possibility constitutes a single simulation, during which
  fault data is distributed at each iteration. Thus, multiple simulations
  are run per invocation of the simulator. Each simulation terminates if
  either all faults are detected by all neighbors, or if the pattern of
  fault data in the array does not change from one iteration from the next.
  The former indicates a successful simulation, whereas the latter indicates
  failure.

  Each iteration produces a snapshot of the state of the fault data in
  the array. Each iteration is numbered in the output, and at the end of
  each simulation, the number of iterations is given. At the end of the
  run, the largest number of iterations for a single simulation is given.
*/

/* Standard system includes */
#include <stdio.h>
#include <stdlib.h>
```

```

/* Define boolean types */
#define FALSE 0
#define TRUE (!FALSE)
typedef unsigned char boolean;

#define MAX_FAULT 4          /* Simulate up to quadruple faults */
#define MAX_ROW 10          /* Number of rows in the array */
#define MAX_COL 10         /* Number of columns in the array */
#define MAX_PHYSICAL 12     /* Number of physical neighbor connections */

#define MAX_AWARE 24        /* Number of cells in area of awareness */
#define NUM_FAULT_BITS 24   /* Number of bits in the fault register */

#define NUM_OUT_BUFFS 2     /* Two iterations are printed per output */
#define NUM_OUT_ROWS MAX_ROW /* Number of rows in the output */
#define NUM_OUT_COLS 140   /* Number of columns in the output */

#define CELL_STRING_SIZE 7  /* Number of characters to output per cell */
#define FORM_FEED 0x0C     /* Define the form feed character */

/* Fault register associated definitions */
#define MAX_OFFSET 2        /* Maximum row/column distance to neighbor */

/*
   This value is not used in the simulation, but could be used to allow a cell
   to detect its own fault via information gained from a neighbor
*/
#define SELF -1            /* Cell detects its own fault via a neighbor */
#define NO_FAULT -1        /* Fault register data is of no concern */

/* Neighbor values */

/*
   Let the current cell be located at (i, j). Rows in the array are numbered
   from 0 to (MAX_ROW-1) from top to bottom. Similarly, columns in the array
   are numbered from 0 to (MAX_COL-1) from left to right.
*/
#define NORTH 0            /* Cell at (i-1, j ) */
#define SOUTH 1            /* Cell at (i+1, j ) */
#define EAST 2             /* Cell at (i , j+1) */
#define WEST 3            /* Cell at (i , j-1) */

#define N_EAST 4           /* Cell at (i-1, j+1) */
#define N_WEST 5           /* Cell at (i-1, j-1) */
#define S_EAST 6           /* Cell at (i+1, j+1) */
#define S_WEST 7           /* Cell at (i+1, j-1) */

#define F_NORTH 8          /* Cell at (i-2, j ) */
#define F_SOUTH 9          /* Cell at (i+2, j ) */
#define F_EAST 10          /* Cell at (i , j+2) */
#define F_WEST 11          /* Cell at (i , j-2) */

#define NN_EAST 12         /* Cell at (i-2, j+1) */
#define EN_EAST 13         /* Cell at (i-1, j+2) */
#define NN_WEST 14         /* Cell at (i-2, j-1) */
#define WN_WEST 15         /* Cell at (i-1, j-2) */

#define SS_EAST 16         /* Cell at (i+2, j+1) */
#define ES_EAST 17         /* Cell at (i+1, j+2) */
#define SS_WEST 18         /* Cell at (i+2, j-1) */
#define WS_WEST 19         /* Cell at (i+1, j-2) */

```



```

#define F_N_EAST      20      /* Cell at (i-2, j+2) */
#define F_N_WEST      21      /* Cell at (i-2, j-2) */
#define F_S_EAST      22      /* Cell at (i+2, j+2) */
#define F_S_WEST      23      /* Cell at (i+2, j-2) */

/*
   This macro maps a bit number into a bit mask by left-shifting 1.
   Equivalent to the following nested IF structure:

   if (n >= 0)          Valid bit positions are non-negative
       return(1 << n);  Left shift 1 by n bit positions

   else if (n == -1)    Use -1 argument to return a 0 mask
       return(0);

   else
       return(-1);      Other negative values return -1 mask (invalid)
*/

#define bit(n) ((n >= 0) ? (1L << (n)) : (n == -1) ? 0L : -1L)

/* Neighborhoods used */

/* Von Neumann neighborhood as North, South, East, and West neighbors */
#define VON_NEUMANN (bit(NORTH) | bit(SOUTH) | bit(EAST) | bit(WEST))

/*
   Moore neighborhood adds Northeast, Northwest, Southeast, and Southwest
   neighbors to Von Neumann neighborhood
*/
#define MOORE (VON_NEUMANN | bit(N_EAST) | bit(N_WEST) | \
bit(S_EAST) | bit(S_WEST))

/*
   Lawson neighborhood adds Far North, Far South, Far East and Far West
   neighbors to Von Neumann neighborhood
*/
#define LAWSON (VON_NEUMANN | bit(F_NORTH) | bit(F_SOUTH) | \
bit(F_EAST) | bit(F_WEST))

/*
   White neighborhood adds Far North, Far South, Far East, and Far West
   neighbors to Von Neumann neighborhood
*/
#define WHITE (MOORE | bit(F_NORTH) | bit(F_SOUTH) | bit(F_EAST) \
| bit(F_WEST))

/*
   The area of awareness of a cell currently includes all cells within a
   radial distance of two from that cell
*/
#define AWARE (WHITE | bit(NN_EAST) | bit(EN_EAST) | bit(NN_WEST) \
| bit(WN_WEST) | bit(SS_EAST) | bit(ES_EAST) | bit(SS_WEST) | bit(WS_WEST) \
| bit(F_N_EAST) | bit(F_N_WEST) | bit(F_S_EAST) | bit(F_S_WEST))

/* Flag for undetected faults */
#define UNDETECTED 0

/*
   Border and near-border cells are missing some neighbors, so we need to
   exclude the missing neighbors from the physical neighborhood
*/

```

```

/* Cells in the first row have no neighbors in the two rows above */
#define FIRST_ROW      (~(bit(NORTH) | bit(N_EAST) | bit(N_WEST) \
| bit(F_NORTH)))

/* Cells in the last row have no neighbors in the two rows below */
#define LAST_ROW       (~(bit(SOUTH) | bit(S_EAST) | bit(S_WEST) \
| bit(F_SOUTH)))

/* Cells in the second row have no Far North neighbors */
#define NEAR_FIRST_ROW (~(bit(F_NORTH)))

/* Cells in the next-to-last row have no Far South neighbors */
#define NEAR_LAST_ROW  (~(bit(F_SOUTH)))

/* Cells in the first column have no neighbors in two columns to the left */
#define FIRST_COL      (~(bit(WEST) | bit(N_WEST) | bit(S_WEST) \
| bit(F_WEST)))

/* Cells in the last column have no neighbors in two columns to the right */
#define LAST_COL       (~(bit(EAST) | bit(N_EAST) | bit(S_EAST) \
| bit(F_EAST)))

/* Cells in the second column have no Far West neighbors */
#define NEAR_FIRST_COL (~(bit(F_WEST)))

/* Cells in the next-to-last column have no Far East neighbors */
#define NEAR_LAST_COL  (~(bit(F_EAST)))

/* Types defined for this program */

typedef
    unsigned long direction_type;    /* North, south, east, etc. */

typedef long
    neighborhood_type;              /* Von Neumann, Moore, Lawson, White */

typedef struct
{
    int
        row,
        col;
} position_type;                    /* Relative row/column offsets */

/*
    Maps neighbor directions to row and column offsets from a given cell
    located at (i,j)
*/

position_type
offset[MAX_AWARE] =
{
    -1,  0, /* North = (i-1, j ) */
    1,  0, /* South = (i+1, j ) */
    0,  1, /* East   = (i , j+1) */
    0, -1, /* West   = (i , j-1) */

    -1,  1, /* NE     = (i-1, j+1) */
    -1, -1, /* NW     = (i-1, j-1) */
    1,  1, /* SE     = (i+1, j+1) */
    1, -1, /* SW     = (i+1, j-1) */

    -2,  0, /* Far N   = (i-2, j ) */
    2,  0, /* Far S   = (i+2, j ) */
    0,  2, /* Far E   = (i , j+2) */
    0, -2, /* Far W   = (i , j-2) */

```

```

-2, 1, /* NNE      (i-2, j+1) */
-1, 2, /* ENE      (i-1, j+2) */
-2, -1, /* NNW     (i-2, j-1) */
-1, -2, /* WNW     (i-1, j-2) */

2, 1, /* SSE      (i+2, j+1) */
1, 2, /* ESE      (i+1, j+2) */
2, -1, /* SSW     (i+2, j-1) */
1, -2, /* WSW     (i+1, j-2) */

-2, 2, /* Far NE   (i-2, j+2) */
-2, -2, /* Far NW  (i-2, j-2) */
2, 2, /* Far SE   (i+2, j+2) */
2, -2 /* Far SW   (i+2, j-2) */
);

/*
We will need to map neighbor row/column offsets to directions. To do so,
a 2D array that spans the maximum distance in the 4 cardinal directions
is used. Thus, we need (two times the maximum offset distance plus one)
rows and columns.
*/
#define DIR_MAP_SIZE (2*MAX_OFFSET + 1)

direction_type
direction_map[DIR_MAP_SIZE][DIR_MAP_SIZE] =
{
    F_N_WEST,      NN_WEST,      F_NORTH,      NN_EAST,      F_N_EAST,
    WN_WEST,      N_WEST,      NORTH,      N_EAST,      EN_EAST,
    F_WEST,      WEST,      SELF,      EAST,      F_EAST,
    WS_WEST,      S_WEST,      SOUTH,      S_EAST,      ES_EAST,
    F_S_WEST,      SS_WEST,      F_SOUTH,      SS_EAST,      F_S_EAST
};

/*
This maps directions to their opposites. That is, the opposite of North
is South, the opposite of Far East is Far West, etc.
*/
direction_type
opposite_direction[MAX_PHYSICAL] =
{
    SOUTH, NORTH, WEST, EAST,
    S_WEST, S_EAST, N_WEST, N_EAST,
    F_SOUTH, F_NORTH, F_WEST, F_EAST
};

/* FAULTMAP.C contains the lookup table for setting the fault register bits */
#include "faultmap.c"

/* The fault register */
typedef unsigned long fault_reg;

/* Initialize fault registers to 0 to simulate a fault-free startup */
fault_reg
init_fault_reg = 0L;

```

```

/* The contents of a cell */
typedef struct
{
    boolean
        is_faulty,                /* Is the cell faulty? */
        is_quarantine,            /* Is the cell a quarantine cell? */
        fault_detected[MAX_FAULT]; /* Does the cell detect a given fault? */

    unsigned char
        fault_num;                /* Fault number assigned to faulty cell */

    fault_reg
        fault_buffer[MAX_PHYSICAL], /* Buffer for neighbor fault data */
        old_fault,                 /* Current fault register image */
        new_fault;                 /* New fault register image */

    neighborhood_type
        log_neighbor,             /* Cells in the logical neighborhood */
        phys_neighbor,            /* Cells in the physical neighborhood */
        detected_by;              /* Neighbors detecting the cell's fault */
} cell_type;

/* The array of control cells is a global variable */
cell_type
    array[MAX_ROW][MAX_COL];

/* ANSI Function prototypes */
void show_usage(char *prog_name);
void init_array(neighborhood_type neighborhood);
int read_faults(char *fault_name, position_type *fault_pos);
void set_faults(position_type *fault_pos, direction_type *fault_dir_list,
    int num_faults);
boolean faults_isolated(boolean *isolated, position_type *fault_pos,
    int num_faults);
void update_faults(int num_faults);
boolean update_array(int num_faults);
void output_array(int num_faults, boolean changed, int *iteration);
void set_neighborhood(char *neighborhood_string,
    neighborhood_type *neighborhood, char *prog_name);
boolean next_fault_dir(direction_type *fault_dir_list, int num_faults,
    position_type *fault_pos);
int main(int argc, char *argv[]);

```

```

/*
  show_usage() - This function displays the correct calling syntax for the
  program and exits.

  Input Parameters:

      prog_name - the name of the executable

  Called By:  main(), set_neighborhood()
*/

void show_usage(char *prog_name)
{
    /* Show calling convention and exit */

    printf("\nUsage:  %s [V|M|W|L] fault_file\n", prog_name);
    printf("\nWhere [V|M|W|L] specifies one of the following neighborhoods:\n");
    printf("\n\t(V)on Neumann = N S E W");
    printf("\n\t(M)oore = Von Neumann + NE SE NW SW");
    printf("\n\t(W)hite = Moore + FN FS FE FW");
    printf("\n\t(L)awson = Von Neumann + FN FS FE FW\n\n");
    exit(1);
}

/*
  init_array() - initializes the array of cells so that no faults are
  present, and each cell is informed as to the cells in its neighborhood

  Input Parameters:

      neighborhood - VON_NEUMANN, MOORE, LAWSON, or WHITE

  Globals Modified:

      array - the array of control cells

  Called By:  main()

*/

void init_array(neighborhood_type neighborhood)
{
    int
        row,           /* The row of the current cell      */
        col,           /* The column of the current cell   */
        fault_num;     /* The current fault number         */

    direction_type
        direction;     /* The current relative direction   */

    cell_type
        *cell;         /* The current cell                 */

    /* Initialize all cells in the array */
    for (row = 0; row < MAX_ROW; row++)
        for (col = 0; col < MAX_COL; col++)
        {
            cell = &array[row][col];

```

```

/* Reset all variables and flags relating to faults */
cell->is_faulty = FALSE;

cell->is_quarantine = FALSE;

cell->fault_num = 0;

cell->detected_by = UNDETECTED;

cell->old_fault = cell->new_fault = init_fault_reg;

for (fault_num = 0; fault_num < MAX_FAULT; fault_num++)
    cell->fault_detected[fault_num] = FALSE;

/* Set up the appropriate neighborhood */
cell->log_neighbor = neighborhood;
cell->phys_neighbor = WHITE;

/* Mask out nonexistent border cell neighbors */
switch (row)
{
    /* Cell is in first row. Mask out all northern neighbors */
    case 0:
        cell->log_neighbor &= FIRST_ROW;
        cell->phys_neighbor &= FIRST_ROW;
        break;

    /* Cell is in second row. Mask out far north neighbor */
    case 1:
        cell->log_neighbor &= NEAR_FIRST_ROW;
        cell->phys_neighbor &= NEAR_FIRST_ROW;
        break;

    /* Cell is in next-to-last row. Mask out far south neighbor */
    case (MAX_ROW - 2):
        cell->log_neighbor &= NEAR_LAST_ROW;
        cell->phys_neighbor &= NEAR_LAST_ROW;
        break;

    /* Cell is in last row. Mask out all southern neighbors */
    case (MAX_ROW - 1):
        cell->log_neighbor &= LAST_ROW;
        cell->phys_neighbor &= LAST_ROW;
        break;

    default:
        break;
}

switch (col)
{
    /* Cell is in first col. Mask out all eastern neighbors */
    case 0:
        cell->log_neighbor &= FIRST_COL;
        cell->phys_neighbor &= FIRST_COL;
        break;

    /* Cell is in second col. Mask out far east neighbor */
    case 1:
        cell->log_neighbor &= NEAR_FIRST_COL;
        cell->phys_neighbor &= NEAR_FIRST_COL;
        break;

    /* Cell is in next-to-last col. Mask out far west neighbor */
    case (MAX_COL - 2):
        cell->log_neighbor &= NEAR_LAST_COL;
        cell->phys_neighbor &= NEAR_LAST_COL;
        break;
}

```

```

        /* Cell is in last col. Mask out all western neighbors */
        case (MAX_COL - 1):
            cell->log_neighbor &= LAST_COL;
            cell->phys_neighbor &= LAST_COL;
            break;

        default:
            break;
    }
}

/*
read_faults() - reads in the locations of the faulty cells from the
specified file.

Input Parameters:

    fault_name - the name of the file containing the fault locations

Output Parameters:

    fault_pos - an array containing the fault locations

Globals Modified:

    array - the array of control cells

Returned Value:

    num_faults - the number of faults read into the array

Called By:  main()

*/
int read_faults(char *fault_name,
                position_type *fault_pos)
{
    FILE
        *fault_file;          /* The file containing the fault locations */

    boolean
        done;                 /* Have all faults been read? */

    int
        num_faults = 0,       /* The number of faults read from the file */
        fault_num,            /* The number assigned to the current fault */
        fault_row,            /* The row containing the current fault */
        fault_col,            /* The column containing the current fault */
        neighbor_row,         /* The row of a neighbor of the fault */
        neighbor_col;         /* The column of a neighbor of the fault */

    cell_type
        *faulty,              /* The current faulty cell */
        *neighbor;            /* A neighbor of the faulty cell */

    direction_type
        fault_dir,            /* The relative direction of the faulty cell */
        neighbor_dir;         /* The relative direction of the neighbor */

```

```

/* Try to open the file with the fault data in it */
if (fault_file = fopen(fault_name, "r"))
{
    /* Read all the faults in */
    for (num_faults = 0, done = FALSE; !done; )
    {
        if (fscanf(fault_file, "%d %d", &fault_row, &fault_col) == 2)
        {
            fault_pos[num_faults].row = fault_row;
            fault_pos[num_faults].col = fault_col;

            faulty = &array[fault_row][fault_col];

            /* Flag faulty cell */
            faulty->is_faulty = TRUE;
            faulty->fault_num = num_faults;

            /* Track number of faults so far */
            num_faults++;
        }
        else
            done = TRUE;
    }

    fclose(fault_file);
}

/* Force flags so we don't check for detection by faulty neighbors */
for (fault_num = 0; fault_num < num_faults; fault_num++)
{
    fault_row = fault_pos[fault_num].row;
    fault_col = fault_pos[fault_num].col;

    faulty = &array[fault_row][fault_col];

    /* Just pretend a cell is already detected by a faulty neighbor */
    for (neighbor_dir = NORTH; neighbor_dir <= S_WEST; neighbor_dir++)
    {
        neighbor_row = fault_row + offset[neighbor_dir].row;
        neighbor_col = fault_col + offset[neighbor_dir].col;

        neighbor = &array[neighbor_row][neighbor_col];

        if (neighbor->is_faulty)
            faulty->detected_by |= bit(neighbor_dir);
    }
}

/* Need to know how many faults were read in */
return(num_faults);
}

```



```

/*
    set_faults() - sets up a detecting cell for each fault in the array.

Input Parameters:

    fault_pos - an array containing the fault locations
    fault_dir_list - an array of relative locations of detecting neighbors
    num_faults - the number of faults in the array

Globals Modified:

    array - the array of control cells

Called By:  main()

*/

void set_faults(position_type *fault_pos,
                direction_type *fault_dir_list,
                int num_faults)
{
    boolean
        done;                /* Are we done setting up faults? */

    int
        fault_num,           /* The number of the current faulty cell */
        fault_row,           /* The row containing the current fault */
        fault_col,           /* The column containing the current fault */
        fault2_num,          /* The number of another fault */
        fault2_row,          /* The row containing another fault */
        fault2_col,          /* The column containing another fault */
        neighbor_row,        /* The row containing the detecting neighbor */
        neighbor_col,        /* The column containing the detecting neighbor */
        fault_row_index,     /* Offset from cell row to neighbor row */
        fault_col_index,     /* Offset from cell column to neighbor column */
        row_diff,            /* Row difference between two faults */
        col_diff;            /* Column difference between two faults */

    cell_type
        *faulty,             /* The current faulty cell */
        *faulty2,            /* Another faulty cell */
        *neighbor;           /* A neighbor of the current faulty cell */

    direction_type
        fault_dir,           /* Relative direction to the current faulty cell */
        fault2_dir,          /* Relative direction to another faulty cell */
        neighbor_dir;        /* Relative direction to the detecting neighbor */

    /* Process all faults in the array */
    for (fault_num = 0; fault_num < num_faults; fault_num++)
    {
        fault_row = fault_pos[fault_num].row;
        fault_col = fault_pos[fault_num].col;

        faulty = &array[fault_row][fault_col];

        /* Find the relative direction of the detecting neighbor */
        neighbor_dir = fault_dir_list[fault_num];

        neighbor_row = fault_row + offset[neighbor_dir].row;
        neighbor_col = fault_col + offset[neighbor_dir].col;
    }
}

```

```

neighbor = &array[neighbor_row][neighbor_col];

neighbor->fault_detected[fault_num] = TRUE;

/* Calculate offset from faulty cell to detecting neighbor */
fault_row_index = fault_row - neighbor_row + MAX_OFFSET;
fault_col_index = fault_col - neighbor_col + MAX_OFFSET;

/* Use direction information to set neighbor fault register */
fault_dir =
    direction_map[fault_row_index][fault_col_index];

neighbor->new_fault =
    (neighbor->old_fault != bit(fault_dir));

/* Exclude faulty cell from the neighborhood */
neighbor->log_neighbor &= ~bit(fault_dir);
neighbor->phys_neighbor &= ~bit(fault_dir);

/* Quarantine faulty cells in the Von Neumann neighborhood */
neighbor->is_quarantine = ((bit(fault_dir) & VON_NEUMANN) != 0);

/* Let faulty cell know who has detected it */
neighbor_dir = opposite_direction[fault_dir];

faulty->detected_by != bit(neighbor_dir);

/*
    Faulty cells are assumed not to detect other faulty cells,
    so the simulation will not check whether a faulty cell is
    detected by other faulty cells in its physical neighborhood.
    Therefore, the simulation just forces these cells to "think"
    they are detected by their faulty neighbors so it doesn't have
    to worry about it later.
*/

/* Check all faults */
for (fault2_num = 0; fault2_num < num_faults; fault2_num++)
{
    fault2_row = fault_pos[fault2_num].row;
    fault2_col = fault_pos[fault2_num].col;

    faulty2 = &array[fault2_row][fault2_col];

    /* Is any faulty cell physically connected to the current fault? */
    if ( ((row_diff = fault2_row - fault_row) <= MAX_OFFSET) &&
        ((col_diff = fault2_col - fault_col) <= MAX_OFFSET) )
    {
        /* Force faulty cells to think they "detect" each other */
        fault2_dir =
            direction_map[row_diff+MAX_OFFSET][col_diff+MAX_OFFSET];

        fault_dir =
            direction_map[MAX_OFFSET-row_diff][MAX_OFFSET-col_diff];

        if (fault2_dir && WHITE)
        {
            faulty->detected_by != bit(fault2_dir);

            faulty2->detected_by != bit(fault_dir);
        }
    }
}
}
}

```

```

/*
faults_isolated() - determines whether faults are detected by all neighbors
in the current neighborhood.

Input Parameters:

    isolated - an array which tracks whether faults are isolated yet
    fault_pos - an array of the locations of faulty cells
    num_faults - the number of faults in the array

Output Parameters:

    isolated - updated to reflect newly isolated faults

Returned Value:

    done - indicates when all faults are detected

Called By:  main()

*/

boolean faults_isolated(boolean *isolated,
                        position_type fault_pos[],
                        int num_faults)
{
    int
        fault,                /* The number of the current fault */
        fault_col,           /* Column containing the faulty cell */
        fault_row,           /* Row containing the faulty cell */
        neighbor_col,        /* Column containing a neighbor */
        neighbor_row;        /* Row containing a neighbor */

    boolean
        done;                /* All faults isolated? */

    cell_type
        *faulty;             /* The faulty cell */

    /* Check all faults for isolation */
    for (fault = 0; fault < num_faults; fault++)
    {
        /* If a fault is not isolated, check for detection by all neighbors */
        if (!(isolated[fault]))
        {
            fault_row = fault_pos[fault].row;
            fault_col = fault_pos[fault].col;

            faulty = &array[fault_row][fault_col];

            isolated[fault] =
                (faulty->detected_by & WHITE) == faulty->phys_neighbor;
        }
    }

    /*
    for (fault = 0, done = TRUE; (fault < num_faults) && done; fault++)
        done = isolated[fault];

    return(done);
    */
}

```

```

/*
update_faults() - the heart of the simulator. Uses the lookup table
contained in FAULTMAP.C to determine how bits in the fault register
should be set, based on fault data received from neighboring cells.

Input Parameters:

    num_faults - the number of faults in the array

Globals Modified:

    array - the array of control cells

Called By:  main()

*/

void update_faults(int num_faults)
{

    direction_type
        direction,          /* Direction to current neighbor      */
        fault_dir,          /* Direction to current fault         */
        buff_bit,           /* Current fault data buffer bit      */
        detect_dir;         /* Direction of detecting cell        */

    int
        row,                /* Current row of the array           */
        col,                /* Current column of the array        */
        neighbor_row,        /* Row containing the neighbor        */
        neighbor_col,        /* Column containing the neighbor     */
        fault,              /* Number of the current fault       */
        faulty_row,         /* Row containing the fault           */
        faulty_col;         /* Column containing the fault        */

    cell_type
        *cell,              /* The current cell in the array      */
        *neighbor,          /* The current neighbor               */
        *faulty;            /* The current faulty cell            */

    unsigned long
        cell_fault;         /* The bit to set in the fault register */

    /* Update fault data in row major order */
    for (row = 0; row < MAX_ROW; row++)
        for (col = 0; col < MAX_COL; col++)
        {
            /* Do not worry about updating the fault registers of faulty cells */
            if (!(cell = &array[row][col])->is_faulty)
            {
                /* Check all neighbors of the current cell */
                for (direction = NORTH; direction <= F_WEST; direction++)
                {
                    /* Cells in the neighborhood can give us valid fault data */
                    if (bit(direction) & cell->log_neighbor)
                    {
                        /* Find neighbor in current direction */
                        neighbor_row = row + offset[direction].row;
                        neighbor_col = col + offset[direction].col;

                        neighbor = &array[neighbor_row][neighbor_col];
                    }
                }
            }
        }
}

```

```

/* Copy neighbor fault register */
cell->fault_buffer[direction] = neighbor->old_fault;

/*
Check all bits in neighbor's fault register. If we can
map a fault in the neighbor's register to ours, we need
to do that. If the fault corresponds to a physical
neighbor, we'll flag it as detected
*/
for (buff_bit = NORTH; buff_bit <= F_S_WEST; buff_bit++)
if (cell->fault_buffer[direction] & bit(buff_bit))
{
/* Where's the fault in relation to us? */
fault_dir = fault_map1[direction][buff_bit];

/* Is this a valid fault? */
if (cell_fault = bit(fault_dir))
{
/* Flag the fault */
cell->new_fault |= cell_fault;

/* Is the fault is in our region of awareness? */
if (cell_fault & AWARE)
{
/* Quarantine faulty cells */
cell->is_quarantine |=
((cell_fault & VON_NEUMANN) != 0);

/* Exclude faulty cell from the neighborhood */
cell->log_neighbor &= ~cell_fault;

/* Find faulty cell and set detection flags */
faulty_row = row + offset[fault_dir].row;
faulty_col = col + offset[fault_dir].col;

faulty = &array[faulty_row][faulty_col];

cell->fault_detected[faulty->fault_num] = TRUE;

if (cell_fault & WHITE)
{
detect_dir = opposite_direction[fault_dir];
faulty->detected_by |= bit(detect_dir);
}
}
}
}

/* Otherwise, clear fault buffer */
else
cell->fault_buffer[direction] = init_fault_reg;
}
}
}
}
}

```

```

/*
update_array() - simply copies the new fault data over the previous data
so that another iteration of the simulation can be performed

Input Parameters:

    num_faults - the number of faults in the array

Globals Modified:

    array - the array of control cells

Returned Value:

    changed - indicates whether any changes occurred this iteration. If no
              changes occur from one iteration to the next, the simulation
              aborts, since nothing will ever change from that point on.

Called By:  main()

*/
boolean update_array(int num_faults)
{
    int
        fault,          /* The current fault          */
        row,            /* The row containing the current fault */
        col;            /* The column containing the current fault */

    boolean
        changed;        /* Did anything change this iteration? */

    fault_reg
        old_fault,      /* Old fault data          */
        new_fault;      /* New fault data          */

    /* Check all cells in the array in row major order */
    for (changed = FALSE, row = 0; row < MAX_ROW; row++)
        for (col = 0; col < MAX_COL; col++)
        {
            old_fault = array[row][col].old_fault;
            new_fault = array[row][col].new_fault;

            /* Need to save new fault data if it differs from old data */
            if (old_fault != new_fault)
            {
                array[row][col].old_fault = new_fault;
                changed = TRUE;
            }
        }

    /* If the array has not changed, the simulation will abort */
    return(changed);
}

```

```

/*
output_array() - produces a visual representation of the state of the array
after every 2 iterations, or at the end of simulation.

Input Parameters:

    num_faults - the number of faults in the array
    changed - indicates whether the array changed since the last iteration

Output Parameters:

    iteration - counts simulation iterations

Called By:  main()

*/
void output_array(int num_faults,
                  boolean changed,
                  int *iteration)
{
    int
        row,                /* The current row */
        col,                /* The current column */
        fault_num,          /* The Number of the current fault */
        buff_num = (*iteration) & 1, /* The current print buffer */
        num_print_buffs;     /* The number of print buffers */

    cell_type
        *cell;              /* The current cell */

    /* Output string buffers for cells and rows of the array */
    char
        row_string[NUM_OUT_BUFFS][MAX_ROW][NUM_OUT_COLS+1],
        cell_string[CELL_STRING_SIZE+1];

    boolean
        odd = (buff_num == 1); /* Print buffers alternate */

    /* Copy array info into appropriate buffer if it changed */
    if (changed)
    {
        /* Process array in row major order */
        for (row = 0; row < MAX_ROW; row++)
        {
            /* Initialize row output string */
            row_string[buff_num][row][0] = '\0';

            /* Concatenate cell strings to get the row string */
            for (col = 0; col < MAX_COL; col++)
            {
                cell = &(array[row][col]);

                /* Determine display characteristics of the current cell */
                if (cell->is_faulty)
                    sprintf(cell_string, "-X%d-- ", cell->fault_num + 1);
                else if (cell->is_quarantine)
                    sprintf(cell_string, "Q---- ");
                else
                    sprintf(cell_string, "----- ");
            }
        }
    }
}

```

```

        /* Indicate the faults detected by the current cell, if any */
        if (!(cell->is_faulty))
        {
            for (fault_num = 0; fault_num < num_faults; fault_num++)
                if (cell->fault_detected[fault_num])
                    cell_string[fault_num+1] = fault_num + '1';
        }

        strcat(row_string[buff_num][row], cell_string);
    }
}

/*
We are trying to output two iterations at a time to conserve paper.
Therefore, each output cycle may consist of an even iteration and the
following odd iteration.

Exceptions occur when the array has not changed since the last
iteration. This indicates failure on the part of the data passing
mechanism, due to an intolerable fault pattern.

If an even iteration matches the previous odd iteration, we need not
output any data. If an odd iteration matches the previous even
iteration, we need to only output the even iteration.
*/

/* Determine the number of iterations to output */
if ((num_print_buffs = odd ? (changed ? 2 : 1) : 0) > 0)
{
    printf("\n\n");

    /* Output iteration counters */
    for (buff_num = 0; buff_num < num_print_buffs; buff_num++)
        printf("%-6ld", (*iteration) - 1 + buff_num);

    printf("\n\n");

    /* Output all rows */
    for (row = 0; row < MAX_ROW; row++)
    {
        for (buff_num = 0; buff_num < num_print_buffs; buff_num++)
            printf("%-6ls", row_string[buff_num][row]);

        printf("\n");
    }
}

/* Keep going if the array is still changing with each iteration */
if (changed)
    (*iteration)++;
}

```



```

/*
set_neighborhood() - determines the neighborhood to use in passing the
fault data, based on the user-specified command line parameter

Input Parameters:

    neighborhood_string - character indicating the neighborhood to use
    prog_name - the name of the simulator program

Output Parameters:

    neighborhood - the type of neighborhood to be used in the simulation

Called By:  main()

*/
void set_neighborhood(char *neighborhood_string,
                     neighborhood_type *neighborhood,
                     char *prog_name)
{
    switch (toupper(neighborhood_string[0]))
    {
        case 'V':
            *neighborhood = VON_NEUMANN;
            printf("\nVon Neumann\n");
            break;

        case 'M':
            *neighborhood = MOORE;
            printf("\nMoore\n");
            break;

        case 'W':
            *neighborhood = WHITE;
            printf("\nWhite\n");
            break;

        case 'L':
            *neighborhood = LAWSON;
            printf("\nLawson\n");
            break;

        default:
            show_usage(prog_name);
            break;
    }
}

```

```

/*
next_fault_dir() - changes the relative locations of the detecting
neighbors from one simulation run to the next. This way, all possible
worst-case detection patterns are simulated. It is best to think of this
as a 4-digit (maximum), modulo-4 counter. Each "digit" represents a
detecting neighbor, and the value of the digit indicates the relative
location from a faulty cell to the detecting neighbor. The counter
is initially set such that all faults are detected by North neighbors,
except one, which is initially detected by its East neighbor. The counter
is incremented after each simulation run, corresponding to rotating the
direction of detection clockwise. When a digit returns to the North
neighbor, the next most significant digit is incremented, and the cycle
repeats. When all digits return to the North neighbor, the simulation
is completed. If two faulty cells are adjacent, such that one is selected
as the detecting neighbor of the other, the counter is incremented as
needed until all faulty cells are only detected by non-faulty cells.

Input Parameters:

    fault_dir_list - the relative locations of the detecting neighbors
    num_faults     - the number of faults in the array
    fault_pos      - the locations of faults in the array

Output Parameters:

    fault_dir_list - updated to indicate new detection pattern

Returned Value:

    full_circle - indicates when the simulation is completed

Called By:  main()

*/

boolean next_fault_dir(direction_type *fault_dir_list,
                      int num_faults,
                      position_type *fault_pos)
{
    int
        fault_num = 0,          /* The number of the current fault          */
        bump_count,            /* Counts the number of detection changes    */
        neighbor_row,          /* Row containing a detecting neighbor        */
        neighbor_col;          /* Column containing a detecting neighbor     */

    static boolean
        first_time = TRUE;     /* Is this the first time through?          */

    boolean
        done,                  /* Finished finding new detecting neighbors? */
        bump_done,             /* Finished with the current fault?          */
        no_faulty,             /* No faulty detecting neighbors?            */
        full_circle;           /* All detection patterns attempted?         */

    cell_type
        *neighbor;             /* The detecting neighbor                    */

    direction_type
        detect_dir;            /* Direction to detecting neighbor          */

```

```

/* Repeat until finished updating the current detection pattern      */
for (done = FALSE; !done; )
{
    for (bump_count = 0, bump_done = FALSE; !(bump_done || first_time); )
    {
        /*
         * Bump the current direction one step. If the resulting direction
         * is NORTH, we have cycled this direction counter all the way around.
         * If there is a next most significant digit, we need to bump it also.
         */

        switch(fault_dir_list[fault_num])
        {
            case NORTH:
                fault_dir_list[fault_num] = EAST;
                bump_done = TRUE;
                break;

            case SOUTH:
                fault_dir_list[fault_num] = WEST;
                bump_done = TRUE;
                break;

            case EAST:
                fault_dir_list[fault_num] = SOUTH;
                bump_done = TRUE;
                break;

            case WEST:
                fault_dir_list[fault_num] = NORTH;

                if (fault_num < (num_faults - 1))
                    fault_num++;
                bump_count++;
                break;
        }
    }

    first_time = FALSE;

    /*
     * If we've incremented all of the direction counters, we've come full
     * circle and need to stop now. Otherwise, we need to do some more
     * checking.
     */
    if (!(full_circle = (bump_count == num_faults)))
    {
        /*
         * Now we need to determine whether any of the detecting cells is a
         * faulty cell. If not, we're done. Otherwise, we need to generate
         * another set of detecting cells by repeating the bump loop above.
         */

        for (fault_num = 0, no_faulty = TRUE;
             (fault_num < num_faults) && no_faulty; fault_num++)
        {
            detect_dir = fault_dir_list[fault_num];

            /* Find neighbor in current direction */
            neighbor_row = fault_pos[fault_num].row + offset[detect_dir].row;
            neighbor_col = fault_pos[fault_num].col + offset[detect_dir].col;

            neighbor = &array[neighbor_row][neighbor_col];

            /* Make sure the detecting neighbor isn't faulty! */
            no_faulty = !(neighbor->is_faulty);
        }
    }
}

```

```

        /*
        If none of the detecting cells are faulty, we're done. Otherwise,
        we need to bump the direction counters again!
        */
        if (no_faulty)
            done = TRUE;
        else
        {
            done = FALSE;
            fault_num = 0;
        }
    }
    else
        done = TRUE;
}

return(full_circle);
}

/*
main() - the main simulation loop. Command line parameters are checked,
and the neighborhood is set up accordingly. Detections are initially
set up to be the North neighbor of each faulty cell. The fault-free
status of the array is initialized,

Input Parameters:

    argv[1] - the neighborhood to use: [v|m|l|w]
    argv[2] - the path for the fault location file
*/

int main(int argc, char *argv[])
{
    neighborhood_type
        neighborhood;                /* The neighborhood to use          */

    boolean
        first_time = TRUE,           /* First time through?              */
        changed,                      /* Has the array changed?           */
        group_done,                   /* Is this run completed?           */
        sim_done,                     /* Is the simulation completed?     */
        isolated[MAX_FAULT];         /* Is this fault isolated?          */

    position_type
        fault_pos[MAX_FAULT];        /* Holds fault locations            */

    direction_type
        direction,                    /* Indicates detection direction    */
        fault_dir_list[MAX_FAULT];    /* Indicates detection direction    */

    int
        num_faults,                  /* How many faults?                 */
        fault_num,                   /* The number of the current fault  */
        sim_count,                   /* The number of simulations        */
        iteration,                   /* The current iteration            */
        max_iteration = 0;           /* Maximum iterations this run      */

    /* Check calling parameters and set up neighborhood */
    if (argc != 3)
        show_usage(argv[0]);

    set_neighborhood(argv[1], &neighborhood, argv[0]);

```

```

/* All faults initially detected by their North neighbors */
for (fault_num = 0; fault_num < MAX_FAULT; fault_num++)
    fault_dir_list[fault_num] = NORTH;

/* One loop per simulation */
for (sim_count = 0, group_done = FALSE; !group_done; sim_count++)
{
    /* New simulation. Reset the array */
    changed = TRUE;

    iteration = 0;

    init_array(neighborhood);

    for (fault_num = 0; fault_num < MAX_FAULT; fault_num++)
        isolated[fault_num] = FALSE;

    /* Read in the fault locations */
    if ((num_faults = read_faults(argv[2], fault_pos)) == 0)
    {
        printf("\nUnable to open fault file: %s\n", argv[2]);
        exit(1);
    }

    /* Set up initial detection pattern */
    if (first_time)
    {
        first_time = FALSE;
        group_done = next_fault_dir(fault_dir_list, num_faults, fault_pos);
    }

    /* Set up initial fault data */
    set_faults(fault_pos, fault_dir_list, num_faults);

    /* Show initial array configuration */
    output_array(num_faults, changed, &iteration);

    /* Iteration loop */
    for (sim_done = FALSE; !sim_done; )
    {
        /* Update fault register information */
        update_faults(num_faults);

        /* Copy new fault register data over old */
        changed = update_array(num_faults);

        /* Show the new configuration */
        output_array(num_faults, changed, &iteration);

        /* Check for end of simulation and output iterations as needed */
        if (changed)
        {
            if(sim_done =
                faults_isolated(isolated, fault_pos, num_faults))
            {
                changed = FALSE;
                output_array(num_faults, changed, &iteration);
                printf("\n\nAll faults isolated!\n\n");
            }
        }
    }
}

```

```

        else
        {
            printf("\nFAILURE! No changes since last iteration!\n");
            sim_done = TRUE;
        }
    }

    /* Track maximum number of iterations this simulation run */
    if ((iteration - 1) > max_iteration)
        max_iteration = iteration - 1;

    /* Set up new fault detection pattern */
    group_done = next_fault_dir(fault_dir_list, num_faults, fault_pos);
}

printf("\n\nSimulations: %d\n", sim_count);
printf("Max Iterations: %d\n\n", max_iteration);
return(1);
}

```

A.2 FAULTMAP.C

/*

Filename: FAULTMAP.C
Programmer: Shannon Lawson
Date: October, 1992.
Used by: ARRAY.C

FAULTMAP.C - This file contains the lookup table for setting the fault register. It's very simple, actually. A cell gets fault data via one of several neighborhoods - Von Neumann, Moore, Lawson, or White. Entries in the table are first grouped according to which neighbor is providing the data, and then by the position of bits in the fault register. Thus, if a cell is receiving data from its North neighbor, it would look in the first set of entries. If the cell needs to examine the West fault bit of its North neighbor, then it looks at the third entry in the first set of entries. The value stored in each entry tells the cell which, if any, of its own neighbors is faulty, based on the data received from a given neighbor.

EXAMPLE - A cell finds that the fault register data received from its North neighbor has a bit set corresponding to the North neighbor's West neighbor. Looking in the first set of entries, which corresponds to the North neighbor, we see that the fourth entry in that set corresponds to the North neighbor's West neighbor, and that the value stored in the entry indicates that the original cell has a faulty Northwest neighbor:

N-X	* - the original cell
/	N - **s North neighbor
*	X - N's West neighbor (**s Northwest neighbor)

Although not used, a value of SELF indicates that the original cell is faulty. This could conceivably happen if one neighbor diagnoses the original cell as faulty, and another neighbor learns of the fault before the cell itself does.

A value of NO_FAULT indicates that the indicated bit position in the neighbor's fault register has no significance to the given cell. For example, if a cell's Far North neighbor has a faulty Northeast neighbor, the faulty cell is too far away from the given cell to consider under the current algorithm.

The table can be easily modified to accommodate more bits and/or more neighbors with minimal changes to the table-driven code in ARRAY.C

*/

```

direction_type
fault_map1[MAX_PHYSICAL][NUM_FAULT_BITS] =
{
    /* North neighbor fault info */

    /* Fault */    /* Relation of fault to North neighbor */    */
    F_NORTH,      /* North */    */
    SELF,         /* South */    */
    N_EAST,       /* East */    */
    N_WEST,       /* West */    */

    NN_EAST,      /* Northeast */    */
    NN_WEST,      /* Northwest */    */
    EAST,        /* Southeast */    */
    WEST,        /* Southwest */    */

    NO_FAULT,     /* Far North */    */
    SOUTH,       /* Far South */    */
    EN_EAST,     /* Far East */    */
    WN_WEST,     /* Far West */    */

    NO_FAULT,     /* North of Northeast */    */
    F_N_EAST,    /* East of Northeast */    */
    NO_FAULT,    /* North of Northwest */    */
    F_N_WEST,    /* West of Northwest */    */

    S_EAST,      /* South of Southeast */    */
    F_EAST,      /* East of Southeast */    */
    S_WEST,      /* South of Southwest */    */
    F_WEST,      /* West of Southwest */    */

    NO_FAULT,    /* Far Northeast */    */
    NO_FAULT,    /* Far Northwest */    */
    ES_EAST,     /* Far Southeast */    */
    WS_WEST,     /* Far Southwest */    */

    /* South neighbor fault info */

    /* Fault */    /* Relation of fault to South neighbor */    */
    SELF,        /* North */    */
    F_SOUTH,     /* South */    */
    S_EAST,      /* East */    */
    S_WEST,      /* West */    */

    EAST,        /* Northeast */    */
    WEST,        /* Northwest */    */
    SS_EAST,     /* Southeast */    */
    SS_WEST,     /* Southwest */    */

    NORTH,       /* Far North */    */
    NO_FAULT,    /* Far South */    */
    ES_EAST,     /* Far East */    */
    WS_WEST,     /* Far West */    */

    N_EAST,      /* North of Northeast */    */
    F_EAST,      /* East of Northeast */    */
    N_WEST,      /* North of Northwest */    */
    F_WEST,      /* West of Northwest */    */

    NO_FAULT,    /* South of Southeast */    */
    F_S_EAST,    /* East of Southeast */    */
    NO_FAULT,    /* South of Southwest */    */
    F_S_WEST,    /* West of Southwest */    */

```



```

EN_EAST,      /* Far Northeast      */
WN_WEST,      /* Far Northwest      */
NO_FAULT,     /* Far Southeast      */
NO_FAULT,     /* Far Southwest      */

```

```
/* East neighbor fault info */
```

```

/* Fault */      /* Relation of fault to East neighbor */

N_EAST,          /* North              */
S_EAST,          /* South              */
F_EAST,          /* East               */
SELF,           /* West              */

EN_EAST,         /* Northeast          */
NORTH,          /* Northwest          */
ES_EAST,         /* Southeast          */
SOUTH,          /* Southwest          */

NN_EAST,         /* Far North          */
SS_EAST,         /* Far South          */
NO_FAULT,        /* Far East           */
WEST,           /* Far West           */

F_N_EAST,        /* North of Northeast */
NO_FAULT,        /* East of Northeast  */
F_NORTH,         /* North of Northwest */
N_WEST,          /* West of Northwest  */

F_S_EAST,        /* South of Southeast */
NO_FAULT,        /* East of Southeast  */
F_SOUTH,         /* South of Southwest */
S_WEST,          /* West of Southwest  */

NO_FAULT,        /* Far Northeast      */
NN_WEST,         /* Far Northwest      */
NO_FAULT,        /* Far Southeast      */
SS_WEST,         /* Far Southwest      */

```

```
/* West neighbor fault info */
```

```

/* Fault */      /* Relation of fault to West neighbor */

N_WEST,          /* North              */
S_WEST,          /* South              */
SELF,           /* East               */
F_WEST,          /* West              */

NORTH,          /* Northeast          */
WN_WEST,         /* Northwest          */
SOUTH,          /* Southeast          */
WS_WEST,         /* Southwest          */

NN_WEST,         /* Far North          */
SS_WEST,         /* Far South          */
EAST,           /* Far East           */
NO_FAULT,        /* Far West           */

F_NORTH,         /* North of Northeast */
N_EAST,          /* East of Northeast  */
F_N_WEST,        /* North of Northwest */
NO_FAULT,        /* West of Northwest  */

```

```

F_SOUTH,      /* South of Southeast      */
S_EAST,       /* East of Southeast       */
F_S_WEST,     /* South of Southwest      */
NO_FAULT,     /* West of Southwest       */

NN_EAST,      /* Far Northeast           */
NO_FAULT,     /* Far Northwest           */
SS_EAST,      /* Far Southeast           */
NO_FAULT,     /* Far Southwest           */

```

```

/* Northeast neighbor fault info */

```

```

/* Fault */      /* Relation of fault to Northeast neighbor */

NN_EAST,        /* North                    */
EAST,           /* South                    */
EN_EAST,        /* East                     */
NORTH,          /* West                     */

F_N_EAST,       /* Northeast                */
F_NORTH,        /* Northwest                */
F_EAST,         /* Southeast                */
SELF,           /* Southwest                */

NO_FAULT,       /* Far North                */
S_EAST,         /* Far South                */
NO_FAULT,       /* Far East                 */
N_WEST,         /* Far West                 */

NO_FAULT,       /* North of Northeast       */
NO_FAULT,       /* East of Northeast        */
NO_FAULT,       /* North of Northwest       */
NN_WEST,        /* West of Northwest        */

ES_EAST,        /* South of Southeast       */
NO_FAULT,       /* East of Southeast        */
SOUTH,          /* South of Southwest       */
WEST,           /* West of Southwest        */

NO_FAULT,       /* Far Northeast            */
NO_FAULT,       /* Far Northwest            */
NO_FAULT,       /* Far Southeast            */
S_WEST,         /* Far Southwest            */

```

```

/* Northwest neighbor fault info */

```

```

/* Fault */      /* Relation of fault to Northwest neighbor */

NN_WEST,        /* North                    */
WEST,           /* South                    */
NORTH,          /* East                     */
WN_WEST,        /* West                     */

F_NORTH,        /* Northeast                */
F_N_WEST,       /* Northwest                */
SELF,           /* Southeast                */
F_WEST,         /* Southwest                */

NO_FAULT,       /* Far North                */
S_WEST,         /* Far South                */
N_EAST,         /* Far East                 */
NO_FAULT,       /* Far West                 */

```

```

NO_FAULT,      /* North of Northeast      */
NN_EAST,       /* East of Northeast       */
NO_FAULT,      /* North of Northwest     */
NO_FAULT,      /* West of Northwest      */

SOUTH,         /* South of Southeast     */
EAST,          /* East of Southeast       */
WS_WEST,       /* South of Southwest     */
NO_FAULT,      /* West of Southwest      */

NO_FAULT,      /* Far Northeast          */
NO_FAULT,      /* Far Northwest          */
S_EAST,        /* Far Southeast          */
NO_FAULT,      /* Far Southwest          */

```

```
/* Southeast neighbor fault info */
```

```

/* Fault */      /* Relation of fault to Southeast neighbor */

EAST,            /* North                  */
SS_EAST,         /* South                  */
ES_EAST,         /* East                   */
SOUTH,          /* West                   */

F_EAST,          /* Northeast              */
SELF,            /* Northwest              */
F_S_EAST,        /* Southeast              */
F_SOUTH,         /* Southwest              */

N_EAST,          /* Far North              */
NO_FAULT,        /* Far South              */
NO_FAULT,        /* Far East               */
S_WEST,          /* Far West               */

EN_EAST,         /* North of Northeast     */
NO_FAULT,        /* East of Northeast     */
NORTH,          /* North of Northwest     */
WEST,           /* West of Northwest     */

NO_FAULT,        /* South of Southeast     */
NO_FAULT,        /* East of Southeast     */
NO_FAULT,        /* South of Southwest     */
SS_WEST,         /* West of Southwest     */

NO_FAULT,        /* Far Northeast          */
N_WEST,          /* Far Northwest          */
NO_FAULT,        /* Far Southeast          */
NO_FAULT,        /* Far Southwest          */

```

```
/* Southwest neighbor fault info */
```

```

/* Fault */      /* Relation of fault to Southwest neighbor */

WEST,           /* North                  */
SS_WEST,        /* South                  */
SOUTH,          /* East                   */
WS_WEST,        /* West                   */

SELF,           /* Northeast              */
F_WEST,         /* Northwest              */
F_SOUTH,        /* Southeast              */
F_S_WEST,       /* Southwest              */

```

```

N.WEST,      /* Far North      */
NO_FAULT,    /* Far South     */
S.EAST,      /* Far East      */
NO_FAULT,    /* Far West      */

NORTH,       /* North of Northeast */
EAST,        /* East of Northeast  */
WN.WEST,     /* North of Northwest */
NO_FAULT,    /* West of Northwest  */

NO_FAULT,    /* South of Southeast */
SS.EAST,     /* East of Southeast  */
NO_FAULT,    /* South of Southwest  */
NO_FAULT,    /* West of Southwest   */

N.EAST,      /* Far Northeast     */
NO_FAULT,    /* Far Northwest     */
NO_FAULT,    /* Far Southeast     */
NO_FAULT,    /* Far Southwest     */

```

/* Far North neighbor fault info */

```

/* Fault */      /* Relation of fault to Far North neighbor */

NO_FAULT,        /* North      */
NORTH,           /* South     */
NN.EAST,         /* East      */
NN.WEST,         /* West      */

NO_FAULT,        /* Northeast */
NO_FAULT,        /* Northwest */
N.EAST,          /* Southeast */
N.WEST,          /* Southwest */

NO_FAULT,        /* Far North */
SELF,            /* Far South */
F_N.EAST,        /* Far East  */
F_N.WEST,        /* Far West  */

NO_FAULT,        /* North of Northeast */
NO_FAULT,        /* East of Northeast  */
NO_FAULT,        /* North of Northwest */
NO_FAULT,        /* West of Northwest  */

EAST,            /* South of Southeast */
EN.EAST,         /* East of Southeast  */
WEST,           /* South of Southwest  */
WN.WEST,        /* West of Southwest   */

NO_FAULT,        /* Far Northeast */
NO_FAULT,        /* Far Northwest */
F_EAST,          /* Far Southeast */
F.WEST,          /* Far Southwest */

```

/* Far South neighbor fault info */

```

/* Fault */      /* Relation of fault to Far South neighbor */

SOUTH,          /* North      */
NO_FAULT,       /* South     */
SS.EAST,        /* East      */
SS.WEST,        /* West      */

```

```

S_EAST,      /* Northeast      */
S_WEST,      /* Northwest      */
NO_FAULT,    /* Southeast      */
NO_FAULT,    /* Southwest      */

SELF,        /* Far North      */
NO_FAULT,    /* Far South      */
F_S_EAST,    /* Far East       */
F_S_WEST,    /* Far West       */

EAST,        /* North of Northeast */
ES_EAST,     /* East of Northeast */
WEST,        /* North of Northwest */
WS_WEST,     /* West of Northwest */

NO_FAULT,    /* South of Southeast */
NO_FAULT,    /* East of Southeast  */
NO_FAULT,    /* South of Southwest  */
NO_FAULT,    /* West of Southwest  */

F_EAST,      /* Far Northeast    */
F_WEST,      /* Far Northwest    */
NO_FAULT,    /* Far Southeast    */
NO_FAULT,    /* Far Southwest    */

/* Far East neighbor fault info */

/* Fault */ /* Relation of fault to Far East neighbor */

EN_EAST,     /* North           */
ES_EAST,     /* South           */
NO_FAULT,    /* East            */
EAST,        /* West            */

NO_FAULT,    /* Northeast       */
N_EAST,      /* Northwest       */
NO_FAULT,    /* Southeast       */
S_EAST,      /* Southwest       */

F_N_EAST,    /* Far North       */
F_S_EAST,    /* Far South       */
NO_FAULT,    /* Far East        */
SELF,        /* Far West        */

NO_FAULT,    /* North of Northeast */
NO_FAULT,    /* East of Northeast */
NN_EAST,     /* North of Northwest */
NORTH,       /* West of Northwest */

NO_FAULT,    /* South of Southeast */
NO_FAULT,    /* East of Southeast  */
SS_EAST,     /* South of Southwest  */
SOUTH,       /* West of Southwest  */

NO_FAULT,    /* Far Northeast    */
F_NORTH,     /* Far Northwest    */
NO_FAULT,    /* Far Southeast    */
F_SOUTH,     /* Far Southwest    */

```

```

/* Far West neighbor fault info */

/* Fault */      /* Relation of fault to Far West neighbor */

WN_WEST,         /* North */
WS_WEST,         /* South */
WEST,           /* East */
NO_FAULT,        /* West */

N_WEST,          /* Northeast */
NO_FAULT,        /* Northwest */
S_WEST,          /* Southeast */
NO_FAULT,        /* Southwest */

F_N_WEST,        /* Far North */
F_S_WEST,        /* Far South */
SELF,           /* Far East */
NO_FAULT,        /* Far West */

NN_WEST,         /* North of Northeast */
NORTH,          /* East of Northeast */
NO_FAULT,        /* North of Northwest */
NO_FAULT,        /* West of Northwest */

SS_WEST,         /* South of Southeast */
SOUTH,          /* East of Southeast */
NO_FAULT,        /* South of Southwest */
NO_FAULT,        /* West of Southwest */

F_NORTH,         /* Far Northeast */
NO_FAULT,        /* Far Northwest */
F_SOUTH,        /* Far Southeast */
NO_FAULT,        /* Far Southwest */

};

```

VITA

Shannon Edward Lawson was born on October 3, 1963 in Newport News, Virginia. He was awarded the degree of Bachelor of Science in Computer Engineering in May 1991 from Virginia Polytechnic Institute and State University, and entered the graduate program in the university's Bradley Department of Electrical Engineering in August 1991 to pursue a Master of Science degree. His current interests include fault-tolerant computing, VLSI circuit design and testing, and microprocessor systems design. He is a member of the IEEE, and the IEEE Computer Society. Following the completion of his Master's degree, he will be accepting a position as a Systems Design Engineer with the Motorola Semiconductor Products Sector in Austin, Texas.