DISTRIBUTED CONTROL RECONFIGURATION ALGORITHMS FOR 2-DIMENSIONAL MESH ARCHITECTURES

by

Tennis S. White

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

APPROVED:

F. G. Gray

Dr. F. G. Gray, Chairman

Dr. E. A. Brown

Dr. J. C. Mckeeman

Dr. M. Nadler

Dr. J. G. Tront

June, 1991

Blacksburg, Virginia

Abstract

Reconfiguration algorithms for 2-dimensional-mesh architectures are presented based upon different interconnect schemes and various numbers of spare cells, arranged in spare rows and/or spare columns. Advantages of these algorithms over existing algorithms is discussed.

ACKNOWLEDGMENTS

The author wishes to thank the chairman of his committee, Dr. F. G. Gray, for his inspiration, guidance and assistance over the past three years, and the other members of the committee, Dr. J. G. Tront, Dr. J. C. Mckeeman, Dr. E. A. Brown and Dr. M. Nadler for their advice and assistance on this dissertation.

Thanks also goes to Research Triangle Institute, Research Triangle Park, N. C. for their financial support. Their assistance over the last five years is greatly appreciated.

Special thanks go to my wife, Carol, and to my daughters Stephanie and Stacy. Without the love, understanding and support of these very important people, none of this would have been possible.

Table of Contents

List of Illustrations

List of Tables

1.0 Introduction

This chapter is broken into four logical sections:

1. A discussion of Parallel processing, including systolic arrays, and the problems encountered with it.
2. An introduction to the cellular architecture, and its problems.
3. A discussion of existing reconfiguration algorithms for mesh arrays.
4. Proposed Research.

1.1 Parallel Processing

There exist today several classes of computation problems that require the use of parallel processors. These include, but are not limited to, weather forecasting, simulation, pattern and speech recognition, computer aided design and artificial intelligence [Ston87]. The algorithms for the solution of these problems are subdivided in a manner that allows the use of multiple processors doing concurrent processing to achieve solutions in much less time than that required by a single serial processor.

However, unless a computing facility has a problem or problems that take full advantage of a parallel processing system, much of the cost of the system will be wasted because of lack of use. This may make the use of a parallel processing system financially unreasonable. One of the most promising methods of parallel processing that is technologically feasible with the advent of VLSI/WSI is the systolic array.

1.1.1 Systolic Arrays

Systolic arrays [Kung82] are special purpose parallel processors that are well suited to implementation in VLSI/WSI technology. They are designed using only a few simple processing elements connected with a very simple and regular communications network. These characteristics make them ideally suited for WSI fabrication. The normal "step and repeat" process [Mead80] used in wafer fabrication, which normally produces many copies of individual circuits on a wafer, can now be used to produce a large array of identical circuits. If the communications network is regular, the cells can be connected internally on the wafer, without having to make connections through external circuit boards.

Data for systolic processors is input to cells, used for one clock cycle, and passed on to the next processor in the chain. Since each piece of data is used many times [Li89], and only one memory reference is required in the main computer, memory bandwidth problems are reduced relative to implementations using serial processors. Problems which are computationally intensive require the use of systolic architectures [DeGr87] to allow the main processor freedom to accomplish other tasks.

Performance improvements achieved in systolic arrays are a result of algorithm specific design, and implementation in VLSI which leads to shorter propagation delays [Kore86] and lower power requirements because conversion of signals to board-level logic and back to chip levels is not required.

A direct result of the algorithm specific design is the use of many different interconnection networks, including rings, trees, meshes and numerous other networks [Feng81].

1.1.1.1 Problems

However, some of the same attributes which make systolic arrays so attractive also create problems. The use of only a few processing element types in the array reduces the cost, but also reduces flexibility because the array is now limited to specific applications. However, manufacturing cost of the array is increased because of a lack of flexibility [Hwan85]. The array is algorithm specific, allowing only one program to be executed. If several problems must be solved, several arrays must be used. The cost will rise significantly with each additional array. Structures with fixed interconnection schemes have a limited set of computational algorithms they can support [Yala85]. The major cost of a systolic array is the array design [Fous87], and being able to use the array for only one, or even a few applications contributes greatly to its cost, because of low fabrication volume. Although the cost of a systolic array is far less than that of a multiprocessing system using mainframe computers, it is still significant.

1.1.1.2 Solutions

To be financially and technologically feasible, systolic arrays must have programmable processing elements and configurable communications networks to achieve more flexibility [Baer84]. These attributes are present in the "CHiP" computer [Snyd82]. This architecture employs an array of polymorphic processors, embedded in a communication network composed of a lattice of programmable switches and connecting links.

The state of each switch in the communication network is controlled by connections to a central computer. If the central computer fails, the system fails. Such a single point failure reduces the

system reliability. To eliminate this undesired attribute, control of the array must be distributed. The system must be configured without the aid of a central computer.

A very good design for the switches in the lattice is provided in [Patr89]. A switch node is composed of six CMOS transmission gates [West85] controlled by two flip-flops. However, they too recommend control of the switches from an external source.

1.2 Cellular Architecture

Figure 1 on page 5 shows a cellular architecture[Kuma84a] [Kuma84b] [Goll84] [Mart80] which achieves distributed control by augmenting the processor-switch lattice with a control hyperplane. The control hyperplane is composed of simple processing elements connected logically as a two-dimensional mesh.

The cells of the control plane must determine the size of the surrounding fault-free area and their state. The state information is then passed to the computational cells of the execution plane, and to the processor switch lattice, which determines the operation to be performed by each computational cell, and the state of each switch.

The size of the fault-free area is defined as the s-value and is determined by taking the s-value of the four neighbors of a cell and adding one to the smallest value. This produces an s-value pattern which determines the size of a diamond shaped state pattern that can be centered on a cell [Kuma84b]. This algorithm was later enhanced independently by Brighton [Brig87] and White [Whit88] to add the possibility of calculating an s-value which would determine the size of a square state pattern which could be supported by each cell. The difference between the two algorithms is

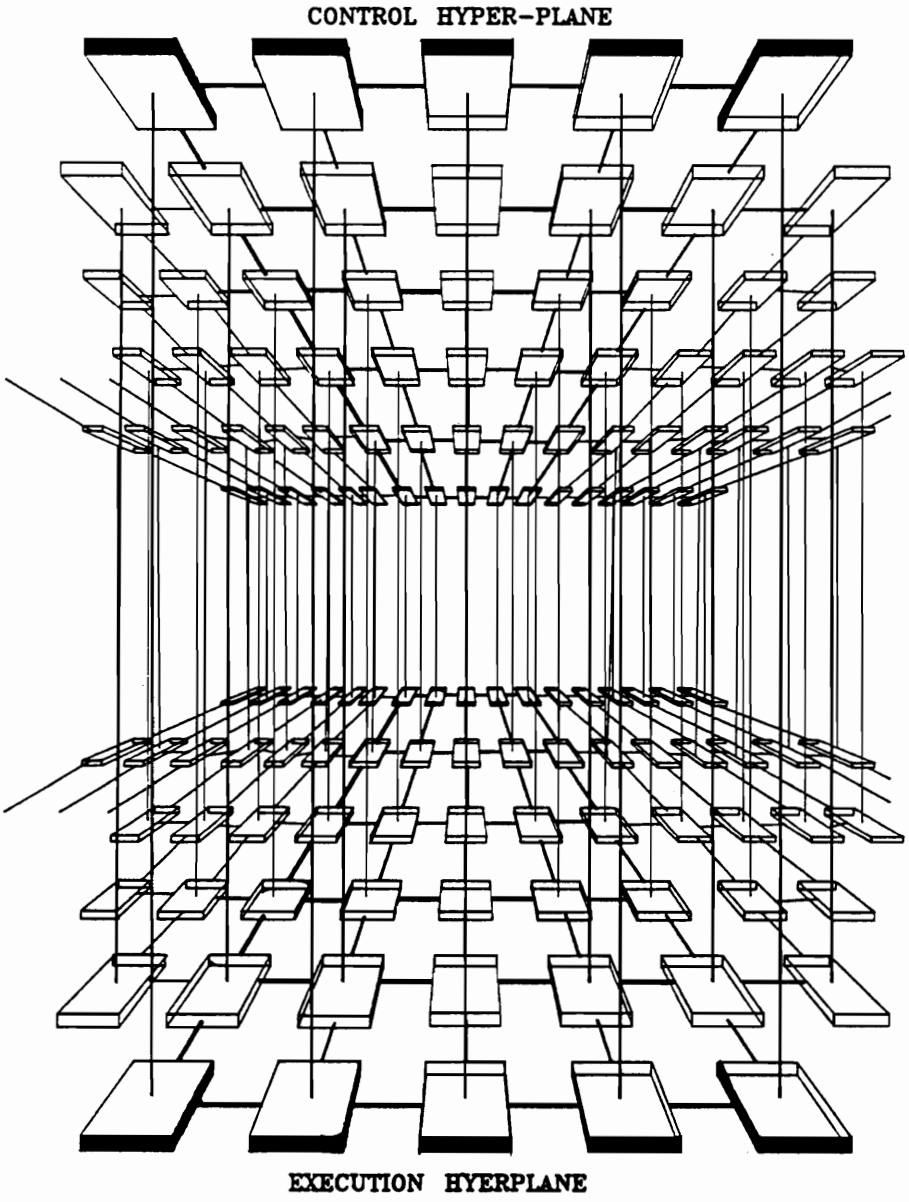


Figure 1. Cellular Architecture: Switch Lattice Execution Plane with Control hyperplane.

the addition of an extra communications cycle in the Brighton algorithm, while the White algorithm communicates the enhancement data with the normal size information.

Placing a seed in any cell of the control plane initiates configuration. The seed contains the global state pattern information for the array, as well as the size of the s-value required to grow the pattern. Each unique seed will grow a unique state pattern. Once a cell in the control hyperplane contains a seed, it examines its s-value to determine if a fault-free area of sufficient size is available. If not, the seed is passed to one of the neighboring cells. Once the seed enters a cell which is surrounded by a fault-free area large enough to grow the pattern, the pattern growth algorithm is entered. Each cell in the array knows its state and the current states of its four neighbors. The next state can be held in ROM memory and merely read out using the combination of the current states as the address. This growth process is a continuous operation, executed each major cycle, even when the cells are in a quiescent state. The array will eventually reach a static global state in which the next state of each cell is identical to its current state [Kumar84b].

This algorithm was later improved by Brighton [Brig87]. In each algorithm, communications is assumed to be simplex, meaning only one cell in a pair may use the link at any given time. The communications pattern, which Kumar had controlled by use of a cell priority register, with cells communicating only with cells of different priority, was changed so that all cells communicated north and west on one cycle, and south and east on the next cycle. This created the possibility of bypassing faulty cells and communicating with a cell which would have had the same priority. Changes were also made in the algorithm of assigning cells the quiescent state. Under the algorithms in both Kumar's Dissertation and Brighton's Thesis, the Q state was assigned to cells having faulty neighbors. Active cells in the state pattern could communicate with other active cells, or cells in the Q state. This was later shown to be unnecessary by White by using switches on communications links to isolate faulty cells rather than using fault-free cells in the Q state. Kumar also assumed a control cell for each switch, an assumption not made by White. A control cell for each switch consumes large amounts of silicon but simplifies the switch control because state pattern growth within the switch lattice completes the entire logical array in one step, assigning states to the

processing elements and configuring the communications network at the same time. However, because the state pattern growth process adds only one or two cells to a row or column each clock cycle, the additional control cells increase the pattern growth time. The inclusion of the control cells for the switches in the same plane as the control cells for the computational units also makes reconfiguration difficult because computational cell controllers would not communicate with each other, and would not know the states of computational neighbors.

When the control hyperplane has reached a stable state configuration, the array is ready to form the required communications network. The output network for each cell is contained within the state information determined in the above step. A cell in the computational plane uses its state to determine not only its computational function, but also the logical location of the cells with which it must communicate.

K. Connell developed an algorithm for forming the connections to the outside world [Conn85]. In it, she assumed each switch and processing element in the computational plane was under the direct control of a unique cell in the control plane. As part of the pattern growth algorithm, cells on the outside perimeter of the active array are designated as input/output cells. The direction priority of the desired path is built into the logic, and the path is formed one cell at a time by designating the input/output cell as the path pointer. Based upon the direction priority, the pointer sends a message to its neighbors requesting their availability to become part of a path. The cell that responds affirmatively that has the highest priority is given the pointer, and the process is repeated until one of the pointer's neighbors is an unassigned port cell. The path is completed by connecting to the port cell. To prevent dead-ends, the algorithm is provided with a backtrack process should no cell respond to the pointers request.

An algorithm for forming the communications paths within the active array was developed by Zaidi [Zaid87]. The assumptions made in this algorithm included processing element control of the switches. The control plane consisted only of cells that controlled the processing elements. The information passed to the processing element as part of the state data included the location of the

processing element, as well as the relative coordinates of the cell to which a processing element must transmit data. The algorithm is executed much the same as the input/output path algorithm, except the processing element has complete control. It determines the priority directions based on its own address and that of the receiving cell. A priority path is chosen, and a connect request sent to the first switch in the path. If the switch responds correctly, the proper connection through the switch is completed, and the process repeated until the destination processor was connected. This algorithm also has backtrack capabilities in case a faulty switch was reached.

1.2.1 Unsolved Problems in the Cellular Architecture

The above array structure has several problems which make it operate in a less-than-optimal fashion. The array is unable to operate if a faulty cell is located within the boundaries of the active area of the array. It must clear the entire state pattern and create a cell with the seed, and regrow the state pattern in another location of the array. This is a relatively lengthy process, since the pattern clearing process must assume a worst case clearing operation in which the clearing function must cover the entire array, not just the active region, to insure all active cells have been cleared. The array must then enter the seed migration algorithm again, locate a sufficiently large area, regrow the state pattern and reestablish the communications network. This is a very inefficient use of the capabilities of the array.

The computational plane has the ability to reconfigure around faulty cells since it is a processor-switch lattice with a reconfigurable communications network. However, the control plane has no such capabilities, creating the necessity of the relocation process mentioned above. To make the cellular architecture more feasible, reconfiguration techniques which make more efficient use of spare cells must be incorporated in the control plane. Reconfiguration should be available in both the horizontal and vertical planes. Since the capacitance of the interconnection links, and therefore

the propagation delay, grows with the square of the length, the reconfigurable array must be constructed using minimum length wires [Leig85].

1.3 Two-Dimensional Mesh Reconfiguration Algorithms

Reconfiguration algorithms can be placed in two general categories, hardware reconfiguration and time sharing [Abra87] or degraded operations mode. In hardware reconfiguration processes, some method is used to logically replace the faulty cell or cells, thus maintaining the integrity of the array. This can be done using local replacement of the faulty cell, where the operations of the faulty cell are performed by a spare cell, or by having a neighboring cell in the active array assume the state of the faulty cell and shifting its own operations to another cell in the same row or column. This shifting continues until a spare cell is included in the active array. Hardware reconfiguration maintains the same performance level achieved by the fault-free array, although this may be less than the same array without reconfiguration capabilities since the clock must be slowed to allow for longer communication times.

Degraded operations, or algorithm reconfiguration, is accomplished by relocating the tasks, which were assigned to the faulty cell, to fault-free cells. These tasks are sometimes executed entirely by one neighboring cell or shared among the four neighbors, and in some cases shared equally among all the remaining fault-free cells in the array. Since at least one processing element must perform more than its original load, the clock must be slowed or extra clock cycles added to enable the longer tasks.

1.3.1 Algorithm Reconfiguration

The Successive-Row-Elimination (SRE) and Alternate-Row-Column-Elimination algorithms (ARCE) [Fort84] [Fort85] eliminate the entire row or column in which a fault occurs. The work performed by the cells in the eliminated row or column is assumed by adjacent rows or columns. Reassigning the tasks of a complete row or column causes no more system degradation than would be created by reassigning the task of a single cell, assuming the workload of the array is evenly distributed. The operations of the array must be slowed by a full clock cycle to allow cells to perform the additional work, regardless of the number of cells involved, and the elimination of an entire row or column simplifies the reconfiguration algorithm. There must also be a modification to the hardware. Cells with additional loads must have the capability of communications with the neighbors of the row or column containing the faulty cell.

In the algorithm presented by Hosseini [Hoss89], modifications are made to the hardware and the algorithm. Processing elements are grouped in clusters, all clusters having the same size. The individual elements of a given cluster are connected vertically to the corresponding elements of the rows above and below. In addition, communications channels are added which allow any cell in a cluster to communicate with the northern and southern neighbors of any other cell in the cluster. Horizontal connections are formed by connecting cell i in an (n + 1)-cell cluster to cell n-i of the following cluster. These redundant links allow the hardware to reconfigure around either faulty cells or faulty communications links. The reconfiguration process is controlled by the cells within the cluster, with knowledge of the location of faulty links or cells within their cluster. Once the reconfiguration has been performed by the hardware, the operations of the faulty cells and those that are unreachable because of faulty links are distributed among the remaining fault-free cells within the cluster.

This algorithm not only has the problem of running in degraded mode, but also has a problem with the length of the communications links, especially if the size of the clusters is large, since the first

cell in each cluster has a link that has a length that is equivalent to the width of 2n + 1 cells. In a cluster containing 3 cells, cell 1 of each cluster is connected to cell 3 of the following cluster. In addition, the loss of all cells within a cluster constitutes a catastrophic failure since all communications between neighboring clusters is lost. This indicates the use of large clusters to achieve fault-tolerance, which increases the communications link length, which causes the array to operate more slowly.

Two approaches to algorithm reconfiguration are presented in [Uyar88]. The first, called Uniform Data Distribution (UDD) distributes the data which would have been processed by a faulty processor evenly among the remaining fault-free processors. The Reduced Data Column (RDC) algorithm places the tasks of a faulty processor in its nearest neighbors. It may be distributed to one of its neighbors, or all of them.

The Time-Redundancy algorithm [Negr85a] [Negr85b] [Negr86] involves both hardware modifications and degraded operations. Each cell is connected to its eight nearest neighbors. When a fault occurs, the input data for that cell is routed to the cell directly above it. Registers are provided to store the data, and an extra clock cycle is introduced to allow the one cell to execute its computation on both sets of data. The output data is also stored in registers, and at the conclusion of the last processing cycle, the data is transferred to the receiving cell, which may be in the row of the faulty cell, or in the same row, depending on the status of the eastern neighbor of the original faulty cell. This algorithm has no way to handle faults in the top row of the array, and contiguous faults in any column cannot be tolerated.

In general, the design of a systolic array to achieve maximum speed of computation precludes the use of algorithm reconfiguration as a first option. In applications where maximum speed must be maintained, degraded operations is unacceptable. However, in applications where catastrophic failure cannot be tolerated, algorithm reconfiguration might be built into a structure and used when hardware reconfiguration algorithms can no longer achieve a full-sized processing array [Fort85].

1.3.2 Hardware Reconfiguration Algorithms

Hardware reconfiguration algorithms can be placed in three general categories, those which embed the two-dimensional mesh array within a processor switch lattice, those which reconfigure by using large multiplexers or switch networks at the input or output of a cell, and those which incorporate direct cell-to-cell links. The algorithms which embed the mesh within the lattice usually use an external controller to determine the switch settings while those using multiplexers depend upon each processor to be self-testing and to generate an error-free signal on the condition of the cell. This signal, or a combination of signals from cells in the same row or column, and possibly all cells in the array, is used to determine the control signals of the multiplexers, and therefore accomplish the reconfiguration.

1.3.3 Embedding Algorithms

The Full-Use-of-Suitable-Spares (FUSS) algorithm [Chea90] achieves near one-hundred percent use of spare cells, even when the faulty cells are clustered within a few rows, by providing the controller with a register that contains the cumulative number of available spare cells for each row. The register entry for row i contains the total number of spare cells available from row 1 thru row i. The controller scans down the register until it encounters a row where the cumulative number of spares is insufficient to handle the number of faulty cells in those rows, giving an index with a negative value. If the index value for row i is -k, then the operations of k faulty cells is shifted to a supporting cell, that is a cell which is in the neighborhood of the faulty cell.

The operations of these cells is shifted downwards to row[i + 1] whenever possible, and the value of the index is incremented to zero for row i (iff all k faulty cells were shiftable to row[i + 1]). The controller continues to scan to the last row of the array, and then scans upward until a positive value is found in the index register. When this occurs in row[i], a faulty cell from row[i + 1] is shifted

upwards to row[i], and the index for row[i] decremented. When the top of the spares vector is reached, all values should be greater than or equal to zero, meaning sufficient spares are available to support reconfiguration. Horizontal shifting of cells is now done to replace any logical cell not available because another logical cell was shifted into its physical location. The reconfiguration is now complete.

Although this algorithm allows for near maximum use of spare cells, the necessary communication link length is very large. To support C spare columns in the original array, each cell must have 5(C+1) horizontal neighbors to the east and 4(2C + 1) -1 neighbors to the south. This creates long paths for communications and slows the operations of the array. Also, to implement distributed control of an N by M array, a delay on N + M clock periods would be necessary before beginning reconfiguration, since the fault status must propagate from one corner of the array to the diagonal corner. The algorithm also assumes all cells are identical and perform the same operation. Since any cell assuming the duties of a faulty cell must know the state of that cell, this algorithm is not acceptable for the control of the processor-switch lattice because reconfiguration to an active neighbor is not guaranteed. A modification to the algorithm which allowed reconfiguration of a faulty cell into one of its four active neighbors, together with fault-tolerant global communications of fault occurrences might make this algorithm better suited to reconfigure the control plane.

The algorithm in [Dist89] not only provides spare cells but also spare communications channels in the processor-switch array. In this way it is able to bypass not only faulty cells, but also faulty communications links. However, the algorithm controlling the reconfiguration process is very complex, and would be very difficult to implement in a distributed control environment.

The basic algorithm in [Kung89] uses a spare column of processing elements on the eastern edge of the active array. When a fault occurs, the operations of all cells to the east of the faulty one are shifted one cell to the east, with the eastern neighbor taking on the tasks of the faulty cell. Communication is rerouted also by changing the state of the switches between the row with the faulty cell and the rows to the north and to the south. It is assumed that even though the cell itself is

faulty, data to be communicated along the row of the faulty cell can be communicated through the faulty cell, eliminating the need for row reconfiguration paths. Although this is similar to other such algorithms, the authors do investigate the possibility of adding spare rows and columns on either side of the array, and the possibility of placing an extra spare row and column in the center of the array, allowing both row and column reconfiguration, and surviving a minimum of three faulty cells per row or column, as well as speeding the reconfiguration process. The above capabilities are achieved with only a single reconfiguration channel for the horizontal data. Additional capabilities could be added by increasing the number of switches and busses on the vertical paths.

The algorithm has the almost universal problems of assuming fault-free communications links, and the ability of a cell to properly declare itself faulty. This assumption also makes it undesirable to use the algorithm of Roychowdhury [Royc90].

A distributed control algorithm similar to the above was presented in [Kung86] [Kung87]. The active array is augmented with a spare column on the east and a spare row on the southern border. Each cell in the array has a state register containing two bits. A state of '00' indicates no faulty cells in the vicinity, state '01' indicates the vertical reconfiguration has taken place to the east of the cell, state '10' is reserved for all cells to the north and east of a fault, and state '11' is used to indicate that a cell is part of a reconfiguration path.

When a fault occurs, reconfiguration may take place in any direction if the state of the faulty cell is '00'. If the state is '01', reconfiguration must be done vertically, '10' requires horizontal reconfiguration, and a faulty cell with a state of '11' indicates a catastrophic failure. This causes no problems if early faults occur near a spare column or row and are reconfigured to the nearest possible spare, since the number of fault-free cells in the reconfiguration path will be small. However, should the early faults occur in the north-east corner of the array, a later faulty cell in the reconfiguration path is a catastrophic failure.

Fault detection for the array is recommended to be processing element duplication, with a comparison network. When a fault is detected, a global interrupt causes all cells to enter the reconfiguration mode. When the reconfiguration is complete the input data for the faulty cell is transferred to its replacement and the computation repeated. Although not stated, all input data must be halted during the reconfiguration cycle. No mention is made about the outputs of the fault-free cells when the fault occurred, so presumably they are discarded and all output data from the previous cycle retransmitted and the entire computation cycle repeated since a large number of the cells will now receive input data from different predecessors.

This algorithm was also developed assuming switches and communications links are fabricated fault-free, and the communication section of the processing element is always fault-free.

An algorithm is presented by Lombardi et al in [Lomb87]. The architecture is modified to provide multiple vertical and horizontal switch busses for reconfiguration. The algorithm is very complex and requires a powerful control processor for the external control, but probability of survival for a 400 cell square mesh with a single row and column of spares is claimed to be near one hundred percent for as many as 30 faulty cells.

An algorithm presented by Moore and Mahat in [Moor85] uses a modified processor switch lattice in which each horizontal path that is the input to a cell is provided with a switching mechanism to allow the input to be passed on to the eastern neighbor should the cell become faulty. The columns are then deformed one cell to the east by switching the outputs to horizontal paths instead of vertical paths in the lattice. The number of spare columns that can be incorporated using this scheme depends upon the number of horizontal busses available between each pair of rows. The algorithm only allows column deformation to the east, which means if three cells are faulty, three spare columns are required and links become long.

Another reconfiguration algorithm that assumes the capability of a faulty cell to function as a connecting link is given in [Popl88]. The main function of the algorithm is not total survivability

when spare cells exist, but maintaining short communications paths. The algorithm allows both row and column reconfiguration and is externally controlled. The algorithm will choose either vertical or horizontal reconfiguration as long as there are no other faults in the region. However, as in the case of [Kung87], once a fault has occurred, reconfiguration around succeeding faults which require reconfiguration that would intersect an existing reconfiguration path are catastrophic. To prevent this, the algorithm includes a provision to backtrack one step when an intersection occurs. Should this happen, the algorithm attempts to reconfigure the previous fault in another plane.

Although no implementation is presented, several embedded algorithms are given in [Sami86b], and a comparison is given of different techniques of implementing the row and/or column reconfigurations. It is shown that the best possibilities of survivability can be achieved if reconfiguration is allowed not only in vertical or horizontal directions, but also on a diagonal. Each array is provided with a spare row and a spare column of identical cells. In their "best" algorithm, each row is scanned left to right until a faulty or stolen cell (i,j) appears. If the cell directly beneath it (i + 1,j) is neither faulty nor stolen, (i,j) is logically relocated to it. However, if (i + 1,j) is unavailable, relocation is attempted to (i + 1,j + 1). If this is unsuccessful, the attempt is made to transfer (i,j) to (i,j + 1). The scan is completed one row at a time, from the top of the array. The survivability of this algorithm is claimed to be near one-hundred percent for fewer than 30 faults in a 20 by 20 array with 41 spare cells. Unfortunately, each cell must be provided with a complement of fourteen northern neighbors and 11 western neighbors to accomplish this, and reconfiguration to a faulty cell's active neighbors is not guaranteed.

A paper by Wang et al. [Wang89] discusses several existing algorithms, and presents one of the most interesting concepts, that of two-level redundancy. The array is partitioned into blocks. Each block has its own row or column reconfiguration algorithm which it implements as long as feasible. Then when a catastrophic failure occurs within a block, the same reconfiguration process used within the blocks is used at the block level to reconfigure blocks and substitute a spare block for the faulty one. However, this algorithm induces extremely long communications paths. The clock must be

set to allow a signal to traverse the width or length of the faulty block, where other algorithms need only bypass faulty cells.

1.3.4 Multiplexer or Switch Controlled Algorithms

The diogenes algorithm [Rose83] [Chun83] constructs arrays as linear arrays bordered by a number of communications busses which may or may not be connected to a cell. The number of busses required depends on the size of the two-dimensional array required. Fault-free cells are connected to the busses as required to form the desired architecture. Since the array is formed one row at a time, the connections between a cell in logical row i and its neighbor in row[i + 1] is at least equal to the width of one row, because the array is fabricated as a linear array. The reconfiguration algorithm requires an external controller. This coupled with the long communications delays and the assumption that busses can be fabricated to be fault-free make this algorithm unsuited for our purposes. A clustered fault, in which relatively complete areas of a wafer are faulty, is one of the greatest reasons for yield problems [Jerv88]. Using the buss communications technique, it is very likely that a buss will eventually enter a fault area, limiting the maximum size of an array.

Two algorithms are presented in [Cimi87] that were devised to enhance fabrication yields. They assert the worst case reconfiguration is created by clusters of faulty cells, and recommend an interconnection scheme in which each cell has blocks of eastern and southern neighbors sufficiently large to bypass the largest expected cluster of faults. These reconfigured arrays, although given a high probability of survival, must run slowly to accommodate the long communications paths.

M. Sami and R. Stefanelli present algorithms in [Sami83] that assume the self-testing capability of each cell. The error signal produced is used to control the data link multiplexers. The amount of reconfigurability that is available is a function of the amount of support logic included in the multiplexer control circuitry. When a cell declares itself to be faulty, the multiplexer controlling its

input from the west is disabled, and the input to that multiplexer is routed to the input of the eastern cell. At the same time, the input from the north is routed one cell to the east. The output of the eastern cells are routed one cell to the west. Reconfiguration of one or more rows or columns is possible, as well as a combination of each.

Algorithms are presented in [Sami86b], [Negr85a] and [Negr86] which tolerate multiple faults in a row and a column. The lowest faulty cell in each column is classified as a vertical fault, and the remaining faults in that column are horizontal faults. The vertical fault is bypassed by reconfiguring to the next lower row. Horizontal faults are bypassed by reconfiguring their columns. Reconfiguration is possible as long as no row has multiple horizontal faults. This algorithm was implemented using one spare row and one spare column.

The second algorithm is the "fault-stealing" algorithm. If two faulty cells occur in one row, one of them is configured up or down one row. If the adjacent row contains no faulty cells of its own, the reconfiguration process is complete. If it does contain a faulty cell, then that fault is bypassed by transferring it either up or down to another row, but not back into the row from which the original double faults occurred. This reconfiguration algorithm will fail if three rows with double faults exist in rows separated only by other rows which contain single faults. The reconfiguration algorithms in this paper assume that faults that occur in a processing element do not affect the communications network of that cell, and data can be passed unaltered through a faulty cell. Algorithms similar to those of Sami and Stefanelli above are also discussed in [Negr85a] and [Doni84].

Problems with these algorithms include assumptions that faulty cells occur only when the array is in a stable state, and only one at a time. Since the error signals are used in combinations with all other similar signals on the array, they must be transmitted on busses which run the length and width of the array. Long busses have an increased probability of crossing an area of clustered faults, which may terminate the signal. Should this occur, the array is in a catastrophic state, since one of the main assumptions in this algorithm is the capability of producing fault-free communications links.

Algorithms similar to those above are also presented in [Gent84]. The multiplexers are replaced with banks of switches, with each cell having as possible neighbors 6 cells from the west and 8 cells from the north. In this case, the switches are controlled by the error signals, with a given input to a cell possibly traversing several switches. Because of the number of switches that might be used in a path, this implementation must take into account large delays.

The multiplexed algorithm presented by Kim and Reddy [Kim89] modifies the structure of the cell by adding switches, multiplexers on both the input and output of the cell, holding registers for vertical input data and a bypass register for the horizontal data lines. It is assumed that if a cell becomes faulty, the added logic will still be functional.

When a cell becomes faulty, data that would have been input on the horizontal data line is routed to the bypass register. Vertical input data for the faulty cell as well as the cells to the right of the fault are rerouted one cell to the east through a switching mechanism. Because the horizontal data is delayed one clock period by being placed in the holding register, a holding register is introduced in the vertical input lines of the cells to the east of the faulty one. This is necessary since all horizontal data will arrive one clock period later than it would have in a fault free state. In the faulty cell, the output multiplexer is set to pass the output of the bypass register instead of the computational output, and in the cells to the east of the faulty cell, the input multiplexer is controlled to accept the output of the vertical holding register instead of the vertical buss. Processing elements in the row to the south of the faulty cell have their switches set to enable the vertical buss to be rerouted back to its original columns. Although some of the output data will be available one clock period later than usual, the input data flow is maintained at the original rate. This algorithm tolerates only one faulty cell per row, but could be enhanced by additional holding registers for vertical data and additional switches and busses for rerouting vertical data.

1.3.5 Direct Link Algorithms

An algorithm presented in [Evan86] has an interconnection network in which each processing element is directly connected to its northwest, west and southwest neighbors as well as it northeast, east and southeast neighbors. The configuration process is begun by having each cell transmit a "request" signal to each of its western neighbors. Each cell that receives a request signal will respond to one of the three possible requests from eastern neighbors based on a set priority. Each cell that receives a response signal then enables the communications link to one of its three western neighbors based upon another set of priorities.

To prevent dead-end rows, in which the communications links in a row of the array are enabled up to a point where they can no longer be extended, each cell of the array must know the locations of all faulty cells in the array. Given this information, each cell is able to respond only to those requests which will not lead to a dead-end.

Problems with this algorithm include the length of time reconfiguration could take. When a cell accepts a response, it drops its other request signals. When these signals are dropped, the priority may change in one of its western neighbors. For example, if two cells have responded to the same eastern neighbor, only one of them can be accepted. The one with the lowest priority will lose its request from its eastern neighbor, so its priority will change and it will have to send out another response signal. The ripple effect of these priority changes in arrays with faulty cells may require large numbers of request/response cycles before a stable state is achieved. Vertical communications is accomplished via a long buss.

A data-skewing algorithm is presented in [Li89]. In data-skewing algorithms, the reconfiguration is accomplished by either changing the schedule of the input data or introducing delays in the data stream in the array, or by both. Data-skewing is necessary because faulty cells are used to transmit data. It is assumed that even though the cell itself is faulty, its communications channels are

fault-free. A faulty cell latches both its horizontal and vertical input data and transmits them to the next cell on the next clock pulse. Each cell is provided with dual communications links in both the vertical and horizontal planes.

In some cases, when the data arrives at a fault-free cell, it is operated upon and then transmitted to two new cells. In other cases, even though the data is in a fault-free cell, it will be passed along to another cell. The actual location of the necessary computations and the amount of skewing necessary is determined by the number and location of the faulty cells. Because of the necessity for skewing, it may not be possible to incorporate all fault-free cells in the active array. This algorithm also makes the assumption that communications sections of the cell are always fault free.

The interstitial redundancy algorithm [Sing88] uses spare processing elements instead of spare rows or columns. The spares are distributed throughout the array, with each spare able to replace only a fixed subset of the primary cells. The spare cells are connected to each of the communications links of their primary cells, enabling them to replace any of them with very small increases in communication link length.

In the basic configuration, the array is partitioned into groups of four processing elements in two adjoining rows and two adjoining columns. The spare is placed equidistant from each of the four cells. The spare can replace any one of these cells should they become faulty, and each of the four primary cells can only be replaced by this one spare. Should any two of the group become faulty, the array fails. Enhancements can be made to the spares complement, increasing the percentage of spares up to two-hundred percent or more. The algorithm for these enhanced versions however becomes more complicated when deciding which spare replaces which primary cell. These algorithms provide very good probability of survivability, but at the expense of using a large amount of silicon area for spare cells.

The Yanney and Hayes algorithm [Yann86] accomplishes reconfiguration by assigning processing elements to be reconfiguration supervisors. A supervisor is responsible for examining the local state

pattern of its neighbors, and when a pattern other than the valid pattern is found, determining the best possible method of replacing the faulty cell. The supervisor first looks for a spare cell which can be directly connected to the same neighbors as the faulty cell. If it finds one, that spare is assigned the state of the faulty processor. If no spare is found, the supervisor itself assumes the state of the faulty processor. This then presents another supervisor with an invalid state pattern and the reconfiguration process continues until all supervisors see only valid state patterns. No processing element can be its supervisor's supervisor, and no two cells may be the supervisor on one cell.

This algorithm also provides good survivability, but has the serious problem of being able to handle only one faulty cell at a time. The existence of two faulty cells could possibly allow two supervisors to assign different states to a single spare cell, causing a catastrophic failure.

A reconfiguration algorithm has been developed [Whit88] [Whit89] specifically to add reconfiguration to the control plane. Using a modified interconnection scheme and information about the relative location of faulty cells in a given row, and the two neighboring rows, a cell is able to determine an interconnection pattern that allows the entire array to tolerate one faulty cell in each row of the active pattern. Each cell is connected to not only its four nearest neighbors, north, south, east and west, but also to the cell to the east of the north, east and south neighbors and to the west of the north, west and south neighbors.

When a faulty cell is detected by its four fault-free neighbors, the states of all cells to the east of it are shifted one cell to the east, and the communications channels adjusted accordingly to maintain the column and row state pattern integrity. The faulty cell is bypassed by enabling a connection directly between its eastern and western neighbors. This algorithm is able to withstand concurrent fault occurrences, as long as no more than one fault occurs in any row.

Once this limit is exceeded however, the array must enter the pattern growth algorithms to relocate the state pattern in an area of the array large enough to support the growth process.

1.4 *Proposed Research*

The primary aims of any reconfiguration must be:

1. Distributed control of the reconfiguration process in which a processing element has control over its local communications network and its state function.

2. The functions of a faulty cell must be assumed by one of its four active fault-free neighbors to preserve the state configuration of the array without additional communications.

3. Locality of Communications. Communications with neighboring cells must be achieved using minimum length links. A link of no more than two cell widths is desirable, but may not achieve the desired survivability.

4. Survivability must be high, approaching one-hundred percent as long as fault-free spare cells exist.

5. The algorithm must have the ability to handle concurrent fault occurrences.

The aim of this research will be to derive a class of reconfiguration algorithms for two dimensional mesh arrays, building on the existing algorithm in [Whit88]. They must maintain short communications paths while making maximum use of the available spare cells. Reconfiguration control must be distributed, to eliminate the hard-core failure problem, with each cell using only information communicated by its local neighbor reconfiguration processes. When reconfiguration fails, the array should configure itself to make available the largest possible number of complete rows and columns to allow growth of maximum state patterns. This will add significant local reconfiguration to the existing architecture, and decrease the occurrences of the lengthy global reconfiguration process.

A second aim of the research will be to develop means to allow configuration of the communications network using state and fault information available within each control cell, placing the control of the switches in the control section of the cellular array. Two methods have been proposed for control of the switches. The first is direct control via the control plane cells, in which no distinction is made between a switch and a processing element. The second is path-growth, with the switches being controlled by the processing elements. Two more options will be investigated. The first is placing the switches in the vicinity of a computational cell under the control of the control cell which governs the computational cell. The second is the generation of a secondary control plane that controls only the switches, allowing the existing control plane to communicate with the computational plane and the switch control plane. Reconfiguration of the computational control plane would then be possible under local control, while reconfiguration of the communications network would be handled separately.

2.0 Algorithm Overview

This chapter defines constraints which are placed upon the algorithms discussed in the following chapters, and gives a general discussion of the fault register, which will contain data which controls the reconfiguration process.

These reconfiguration algorithms are developed for two-dimensional mesh connected array architectures under the following constraints:

1. First, control of the reconfiguration process must be distributed. Each cell must determine its active communication links using only information contained in its own memory. This reduces the possibility of a single-point failure which can be catastrophic to the operations of the array. If control is not distributed, and an external processor is employed to control the state of each cell and the configuration of the communications network, the failure of this processor means the array can no longer be used.

2. The algorithms were developed for architectures in which all cells may be physically identical, but the operation of each cell is determined by its state. A cell determines its current state by examining the states of its four logical neighbors. Each cell transmits its state to its four neighbors once each major clock cycle. Since the state of any cell is known only by its four

neighbors, the operations of a cell which becomes faulty must be assumed by one of its fault-free neighbors.

3. No global communications is allowed. Implementation of global communications within an array necessitates the use of relatively long conductors. As the length of the conductors increases, the probability of a fault occurrence in the conductor increases.

4. A cell is either fault-free or faulty. Assumptions of fault-free communications resources within otherwise faulty cells are not allowed.

5. Minimum distance connections are to be used to insure maximum speed of the reconfiguration process.

6. The array must be in a steady-state condition when the fault occurs. A cell which becomes faulty while changing states will require the implementation of the global reconfiguration algorithm because none of its neighbors will become aware of its new state until the completion of the next major clock cycle, when a fault-free cell would have transmitted its new state information to its neighbors. Without knowledge of this new state, no possibility exists for a neighboring cell to assume the state of the faulty cell.

7. A suitable testing algorithm must be developed to allow each cell to determine the exact status of cells with which it has the capability to communicate. A definite problem exists if a faulty cell does not enable the communications links to cells which are trying to test it. Since the reconfiguration algorithms discussed here change configuration as soon as a fault condition is detected, cells which are in the process of changing their communications network will not have a complete neighborhood at all times. For this reason, an incomplete connection from a fault-free cell to another of unknown condition cannot be used as an indication that a cell is faulty.

8. Each cell has a "state" which is determined by an algorithm[Kumar84b] which uses a cells current state, and the current state of its four logical neighbors to determine the next state of the cell. Each cell has a register which contains its state and the state of its four logical neighbors which can be used during reconfiguration to allow a cell to take on the previous state of a neighbor.

2.1 Communications

Communication between cells is half-duplex, meaning cells cannot receive data from and transmit data to the same cell in any clock period. This scheme increases the amount of time required to complete the necessary communications between a pair of cells, but has the advantage of using much less cell area for communications and decreasing the possibility of communications failure. For this reason, five minor clock cycles are required to fully transmit all the necessary information for reconfiguration. These cycles are:

1. Transmit data to the northern and western neighbors.
   Receive data from the southern and eastern neighbors.

2. Transmit data to the southern and eastern neighbors.
   Receive data from the northern and western neighbors.

3. Compile received data and execute the reconfiguration algorithm.

If the algorithms are used with the cellular architecture described by Kumar[Kuma84b], the data required for reconfiguration can be transmitted along with s-value or state data. It should be transmitted in such a manner that the number of bits transmitted during all transmission cycles is the same, to make communications overhead a minimum length of time.

2.2 *The Fault Register*

Reconfiguration is accomplished by decoding the contents of a "fault" register located within the logic of each cell in the control plane. The decoded results are used to enable or disable gates which connect a cell to a communications path to one of its neighbors. Each cell controls its own input and output paths. Therefore, two cells must enable communications links to complete the path. This prevents the output of a faulty cell from affecting the input of a fault-free cell because the fault-free cell will disconnect the link at its end.

The fault register is actually two registers, one used for transmitting data to neighboring cells and the second used for decoding to determine the configuration of the communication network. At the beginning of each major clock cycle, the contents of register 1 is copied to fault-register 2. As new data is received during the communications cycles it is used to set the contents of fault-register 1. During minor clock cycle 3, the contents of fault-register 2 is decoded and the results used to set the state of the communications.

3.0 Algorithm 2-10

Algorithms discussed will be named algorithm x-y, where x is the number of columns or rows of spare cells, and y is the number of cells to which a given cell is physically connected.

Throughout this text, reference will be made to a cells physical location within the array by referring to its co-ordinates. Cell[i,j] will indicate the cell in the i'th row and the j'th column, with i and j being a positive number. To eliminate confusion, compass directions will have three meanings. Capitalized initials (N, NE, etc.) will indicate bits in the fault register. Lower case names (north, north-east etc.) will be used to reference the physical neighbors of a cell. For example, with cell[i,j] as a reference, if cell[i-1,j + 1] is faulty, this is the north-east neighbor of cell[i,j], but cell[i,j] will set the NE bit in its fault register, which it would set also if cell[i-1,j + n] | n > = 0 were faulty. Uppercase names, ie. NORTH will be used to refer to a logical neighbor. Only NORTH, SOUTH, EAST and WEST are used.

The first algorithm to be discussed uses a communications network which allows a cell to select four of its ten physical neighbors as its logical neighbors. A cell must choose a NORTH, SOUTH, EAST and WEST neighbor. Figure 2 on page 67 shows this neighborhood configuration.

The cells which may serve as neighbors of cell[i,j] are:

	Designator	location	Can Serve as Logical Neighbor
•	north-east	Cell[i-1,j + 1]	NORTH
•	north	Cell[i-1,j]	NORTH
•	north-west	Cell[i-1,j-1]	NORTH
•	far-east	Cell[i,j + 2]	EAST
•	east	Cell[i,j + 1]	EAST
•	far-west	Cell[i,j-2]	WEST
•	west	Cell[i,j-1]	WEST
•	south-east	Cell[i + 1,j + 1]	SOUTH
•	south	Cell[i + 1,j]	SOUTH
•	south-west	Cell[i + 1,j-1]	SOUTH

Figure 3 on page 68 shows the spares complement for this array, which consists of two columns of cells, on either end of the array. For reference, this algorithm will be called the 2-10 algorithm.

3.1 The Fault Register

The four communications channels which are enabled are determined by decoding 16 bits of a 24 bit fault register. The fault register is composed of two separate 24 bit registers, one used for communications to neighboring cells, and the other used for the reconfiguration process. The registers contain the relative locations of faulty cells within a five row area bounded by row[i-2] to the north and row[i + 2] to the south. The eight bits which contain fault information about row[i + 2] and row[i-2] are not used in decoding, but are used to determine the condition of some of the other bits. Table 1 on page 65 shows the contents of the fault register and their definitions. Figure 4 on page 69 shows cell[i,j] and its surrounding cells, indicating which bit of the fault register would be set in cell[i,j] if the neighboring cell were faulty.

In addition to the contents of the two fault registers, each cell keeps the status of each of its ten neighbors. These ten bits are transmitted to each of the neighbors, but are not decoded. These bits are used to set fault register bits in other cells. An example of this is the one bit in each of the algorithms which refers to cell[i-2,j] being faulty. If cell[i-1,j] is faulty, it may not be possible for cell[i,j] to learn directly if cell[i-2,j] is faulty. This information is normally transmitted by cell[i-1,j]. In this case, cell[i,j] will learn of a faulty cell[i-2,j] by decoding either a northeast-neighbor-is-faulty bit from cell[i-1,j-1] or a northwest-neighbor-is-faulty bit from cell[i-1,j + 1].

3.1.1 Setting the Fault-Register

This is a general discussion of how fault register bits are set. A more complete definition of the logic involved can be found in the text below and in the simulation code found in the Appendix under the file name faultrg_x.c, where _x is the number of the algorithm by position in this text.

Bit 0, indicating a fault in cell[i-1,j] can be set by direct testing. It may also be set by receiving a bit(8) from its northern neighbor if it is connected to cell[i-1,j + 1], or from a similar input of bit(6) from cell[i-1,j-1]. Bit(1), NW, is set by direct testing if the northern neighbor is cell[i-1,j-1], or by receiving a N or NW fault indicator from cell[i,j-1], the western neighbor. Bit(2), NE, is set by testing cell[i-1,j + 1] or receiving a N or NE from cell[i,j + 1], the eastern neighbor. Bits 3, 4 and 5, indicating S, SW or SE faults exist in row[i + 1] are set in the same way as the first three bits.

Bits 6 and 8, E and W, indicating whether the eastern and western neighbors, cell[i,j + 1] and cell[i,j-1], are faulty, are set by direct testing. Bit 7, FE, indicates that a faulty cell exists in row[i] which is greater than one cell distance from cell[i,j]. This bit may be set by directly testing cell[i,j + 1] or by receiving either an E or FE from an eastern neighbor. Bit 9, indicating a faulty cell to the far-west of cell[i,j] is set in a similar fashion with data received from western neighbors.

Bit 10 (NN) indicates a fault in cell[i-2,j]. It is normally set by receiving a bit 0 from cell[i-1,j], but may also be set by receiving one of the auxiliary bits from either cell[i-1,j-1] or cell[i-1,j + 1], as described in Chapter 2. The bit which indicates a NNW fault, row[i-2] and west of column[j], can be set with input from any northern neighbor except cell[i-1,j + 1]. If the input is from cell[i-1,j], then bit 11 must be received before cell[i,j] can set bit 11. If cell[i-1,j-1] is the northern neighbor, then bit 11 is set if cell[i-1,j-1] has either a NN or a NNW fault. Bit 11 may also be set if the input from a western neighbor contains bit 11. Bit 12, NNE, is set using input from the north, north-east or east neighbors. Bit 13 indicates more than one fault exists in row[i-2]. This bit is labeled MFNN, or multiple-fault-north-north. This bit is necessary for cells which are either east or west of both faults in row[i-2], and is set initially by a cell in row[i] which has both a NN and either a NNE or NNW bit set. It then is propagated to the east and west in row[i]. Bits 14, 15, 16 and 17 hold the same information about row[i + 2] as do bits 10, 11, 12 and 13 about row[i-2], and these bits are set similarly.

Bit 18 is used to identify situations in which two faults exist in row[i]. This bit is set by four conditions, and then propagated to the east and west. The four conditions are:

- E and FE set.
- W and (E or FE) is set.
- E and (W or FW).
- W and FW set.

Bit 19, MFN, is used to indicate two faults in row[i-1]. Two possibilities exist for setting this bit, (N) and either a (NE) or (NW) bit must be set. Bit 20 is set to indicate two faults in row[i + 1] and is set in a similar manner. Bit 20 is labeled MFS. Both bits are then propagated to the east and west from the generating cell.

Three bits are needed to identify instances in which configuration must be reversed for cases in which a single fault in row[i] is accompanied by multiple faults in either row[i-1] or row[i + 1]. These bits are labeled FCR, FCRN and FCRS, bits 21, 22 and 23. These bits are set if the single

fault lies west of a multiple fault in an adjacent row. The conditions required to set bit 21, FCR, Fault-in-Critical-Region, are:

- MFN and not NW and W
- MFS and not SW and W

Bit 22, FCRN, Fault-in-Critical-Region-North, is set for

1. MF and (NN or NNW) and not (W or FW)
2. MFNN and not NNW and (N or NW)

This would indicate a fault existed in row[i-1] which was west of a double fault in row[i-2]. Bit 23, FCRS, is set with conditions which are the mirror image of the conditions used to set bit 21. The last three bits will be set in unison in three separate rows surrounding the row in which the critical fault occurred.

The conditions listed above are only the major conditions used to set fault register bits. Other means exist for setting bits should the normal methods not exist because of communications network mismatches. As stated earlier, this is only a general discussion. A more complete set of conditions used in setting bits in the fault register is given in Appendix A, in file name Faultrg1.c.

3.2 The Algorithm

This algorithm requires only one piece of information to determine the east or west connection enabled by a cell. If the western neighbor is faulty, as indicated by a bit(8) set in the second fault register, a connection to the far-west neighbor, cell[i,j-2], is enabled. If the eastern neighbor is faulty, resulting in bit(6) being set, the connection to cell[i,j + 2] is enabled. The configuration occurs at all times unless both neighboring cells are faulty. This induces a fatal condition and the global reconfiguration algorithm must be invoked to clear the state pattern and regrow the pattern elsewhere in the array. A formal definition of the algorithm is given below using the contents of

the fault register for determining the connections. The statement of the algorithm is in the form of switch/case statements, with the switch being the conditions:

- multiple-fault (MF)
- multiple-fault-north (MFN)
- multiple-fault-south (MFS)
- critical-region-fault (FCR)
- critical-region-north-fault (FCRN)
- critical-region-south-fault (FCRS)

The connections to the north are not a function of the fault conditions to the south, nor is the connection to the south based on faults which occur in rows to the north except in special cases. In these cases, the fault conditions to the south or north are used to determine if a critical area fault exists, and the FCR bits are used instead of the actual fault conditions.

Algorithm 2-10

```
{ /* begin algorithm 2-10 */

if(W)  connect far-west
else   connect west

if(E)  connect far-east
else   connect east

/* begin north link code */
switch( MF MFN FCR FCRN)
 {

case (null){
1.  if((not(W or FW) and  not(N or NW)) or ((W or FW) and  NW) )
                                          connect north;
2.  else  if(((N or NW) and  not(W or FW)))
                                          connect north-east;
3.  else  if (((W or FW) and  not(NW)))
                                          connect north-west;
                                          break; }

case (FCR){
4.  if(((W or FW) and  not(N or NW)) )
                                          connect north;
5.  else  if(((W or FW) and  (N or NW)) or (not(W or FW)))
                                          connect north-east
```

```
6.  else connect north-west /* not possible in this configuration */
                                                    break; }

    case (FCRN)   {
7.  if (((NW) and  not(W or FW)))
                                                    connect north;
8.  else  if ((not(NW)) or ((NW) and  (W or FW)))
                                                    connect north-west;
9.  else  connect north-east;/* not possible in this configuration */
                                                    break; }

    case (FCR AND FCRN){
10. if((not(W or FW) and  not(N or NW)) or ((W or FW) and  (NW)))
                                                    connect north;
11. else  if((not(W or FW) and (N or NW)))
                                                    connect north-east;
12. else  if (((W or FW) and  not(NW)))
                                                    connect north-west;
                                                    break; }

    case (MF)   {
13. if(((W or FW) and  (E or FE) and          not(N or NW))
                          or (NW and  not(E or FE)))
                                                    connect north;
14. else if((not(W or FW)) or ((W or FW) and  (E or FE) and  (N or NW)))
                                                    connect north-east;
15. else if((not(E or FE) and  not(NW)))
                                                    connect north-west;
                                                    break; }

    case (MF AND FCRN){
16. if((NE) or ((W or FW) and  (E or FE)))
                                                    connect north;
17. else if((not(W or FW) and  (N or NW)))
                                                    connect north-east;
18. else if((not(E or FE)))
                                                    connect north-west;
                                                    break; }

    case (MFN)   {
19. if((not(W or FW) and  (NW and NE)
          or ((W or FW)  and  (not(N or NE))))
                                                    connect north;
20. else if((not(W or FW or NE)))
                                                    connect north-east;
```

```
21. else if((not(NW)) or ((W or FW) and  (N or NE) and (NW)))
                                                connect north-west;
                                                break; }

    case (MFN AND FCR){

22. if((not(W or FW)) or ((NW and NE))
                                                connect north;

23. else if((not(NE)))
                                                connect north-east;

24. else if(((W or FW) and not(NW)))
                                                connect north-west;
                                                break; }

    case (MF AND MFN)
    {

25. if((not(W or FW) and  not(N or NW))
          or ((W or FW) and (E or FE) and (NW and NE))
          or (not(E or FE)                      and  not(N or NE)))
                                                connect north;

26. else if((not(W or FW) and  (N or NW))
        or ((E or FE) and  (W or FW) and  not(NE)))
                                                connect north-east;

27. else if(((W or FW) and  not(NW)) or (not(E or FE) and  (N or NE)))
                                                connect north-west;
                                                break; }
     } /* end switch north link code */

    /*  begin south link code  */
    switch(MF MFS FCR FCS)
    { /* south link switch code */

    case (null):
      {

28. if((not(W or FW) and  not(S or SW)) or ((W or FW) and  (SW)))
                                                connect south;

29. else if(((W or FW) and  not(SW)))
                                                connect south-west;

30. else if((not(W or FW) and  (S or SW)))
                                                connect south-east;
                                                break; }

    case (FCR)
      {

31. if(((W or FW)  and  (not(S or SW))))
                                                connect south;

32. else if(((W or FW)  and  ((S or SW))) or (not(W or FW)))
                                                connect south-east;
```

```
33. else connect south-west; /* not possible this configuration */
                                                    break; }

    case (FCRS)
      {
34. if((not(W or FW) and  SW))
                                                    connect south;
35. else if((not(SW)) or ((SW)  and  (W or FW)))
                                                    connect south-west;
36. else   connect south-east;/* not possible this configuration */
                                                    break; }

    case (FCR AND FCRS)
      {
37. if((not(W or FW)) or ((W or FW) and  SW))
                                                    connect south;
38. else if(((W or FW) and  not(SW)))
                                                    connect south-west;
39. else if((not(W or FW) and (S or SW)))
                                                    connect south-east;
                                                    break; }

    case (MF)
      {
40. if(((W or FW) and  (E or FE) and  not(S or SW)))
                                                    connect south;
41. else if((not(E or FE or SW)))
                                                    connect south-west;
42. else if((not(W or FW)) or ((W or FW) and  (E or FE) and (S or SW)))
                                                    connect south-east;
                                                    break; }

    case (MF AND FCRS)
      {
43. if((SE) or ((W or FW) and  (E or FE)))
                                                    connect south;
44. else if((not(E or FE)))
                                                    connect south-west;
45. else if(((S or SW) and  not(W or FW)))
                                                    connect south-east;
                                                    break; }

    case (MFS)
     {
46. if(((SW and SE) and not(W or FW)))
                                                    connect south;
```

```
47. else if((not(SW)) or ((SE or S) and  (SW) and  (W or FW)))
                                                connect south-west;

48. else if(not(W or FW or SE))
                                                connect south-east;
                                                break; }

    case (MFS AND FCR)
      {
49. if((SW and SE) or (not(W or FW)))
                                                connect south;

50. else if(((W or FW) and  not(SW)))
                                                connect south-west;

51. else if(not(SE))
                                                connect south-east;
                                                break; }

    case ( MF AND MFS)
      {
52. if(((W or FW) and  (E or FE) and  (SW and SE))
          or (not(E or FE) and  not(SE or S))
          or (not(W or FW) and  not(S or SW)))
                                                connect south;

53. else if(((W or FW) and  not(SW))
          or (not(E or FE) and  (S or SE)))
                                                connect south-west;

54. else if(((W or FW)and  (E or FE) and  not(SE))
          or (not(E or FE) and  (S or SE))
          or (not(W or FW) and (S or SW)))
                                                connect south-east;
                                                break; }
    } /* end south link switch code */

    } /* end reconfigure */
```

Reconfiguration around single fault occurrences is accomplished by forcing the cells to the east of the fault to take on the function of its western neighbor. Reconfiguration around occurrences of two faults within one row is accomplished by configuring east of the eastern fault, and west of the western fault. Because of the limited inter-connections, with no possibility of accomplishing row-deformation, this architecture is not capable of tolerating a fault condition in which two adjacent cells in one row are faulty. Execution of the next-state algorithm is suspended during the reconfiguration process.

In the following discussion, reference to algorithm configurations is given as A-B, where A is the northern algorithm configuration number and B is the southern configuration number, referring to the reference numbers in the algorithm definition. For example, configuration 1-28 indicates a cell with NORTH configuration 1 and SOUTH configuration 28 referring to the formal definition of the algorithm given above.

3.3 Fault Coverage

3.3.1 Single Faults.

Theorem 1.1: An array equipped with hardware to use algorithm 2-10 will tolerate all single-fault occurrences.

Proof: Suppose cell[i,j] is discovered to be faulty at time t = 0. Because this is the first fault to occur, the condition is detected by cell[i-1,j], cell[i + 1,j], cell[i,j-1] and cell[i,j + 1]. Figure 5 on page 70 shows this configuration, with the state of each cell shown above each cell, and the non-zero fault register contents shown below each cell.

3.3.1.1 Cycle Two

During the next minor clock cycle 1, this data is transmitted to the northern and western neighbors of each cell. In minor cycle 2 it is transmitted to the southern and eastern neighbors. In cycle 3, the current data contained by fault register 1 is copied into fault register 2, and the new input data received from neighboring cells in minor cycle 1 and 2 is then used to determine the new contents of fault register 1. Cell[i,j-1] will be in configuration 1-28, north and south, while cell[i,j + 1] will

be in configuration 3-29, north-west and southwest. These cells will disconnect from the faulty cell and connect to each other as E/W neighbors. Cell[i-1,j] has configuration 1-30, and changes its SOUTH link to the south-east. To complete communications network reconfiguration, cell[i + 1,j] is in configuration 2-28, and changes its NORTH link to north-east.

Figure 6 on page 71 shows the configuration of the array at the end of cycle 2. The contents of the fault register 1 of each cell is shown under the cell boundary of each cell. The partial links leaving cells [i-1,j + 1] and [i + 1,j + 1] indicate that they have no neighbors on these links. Cells [i-2,j] and [i + 2,j] are not involved in the reconfiguration process, although their fault register contents are not null. These two cells are in configuration 1-28, north and south, because single faults in row[i] do not affect the configuration of cells in row[i-2] or [i + 2]. Configuration for t = 1 is completed when cell[i,j + 1] shifts the contents of its state register into the register which holds the state of its EAST neighbor, and state of cell[i,j], which it has in a holding register, into its own state register.

3.3.1.2 Third Cycle

In the third cycle, all cells which contain information about the fault transfer it to their neighbors during the communications cycles. Figure 7 on page 72 shows the contents of the fault registers of these cells at the end of the third cycle. Cells [i-1,j + 2], [i,j + 3] and [i + 1,j + 2] receive data from their western neighbors. Cell[i-1,j + 1] will transfer its fault register contents to the reconfiguration register, and will be in algorithm configuration 1-30, fault to the SW, and alter its southern link to connect to south-east. Cell[i,j + 2] will be in configuration 3-29 after its register transfer and will connect to the north-west and the south-west, cells [i-1,j + 1] and [i + 1,j + 1]. Cell[i + 1,j + 1] will be in configuration number 2-28, fault to the NW, and alter its northern link to north-east, cell[i,j + 2]. Cell[i,j + 2] will then repeat the state reconfiguration steps executed by cell[i,j + 1] in the second cycle and will take on the state previously held by cell[i,j + 1].

3.3.1.3 Fourth Cycle

In cycle 4, information is again transferred to the east and west to cells [i-1,j + 4], [i,j + 5] and [i + 1,j + 4]. The cells from which these three cells received their fault data, [i-1,j + 3], [i,j + 4] and [i + 1,j + 3] will then configure exactly as their western neighbors did in cycle 3, with cell[i,j + 4] taking on the state previously held by cell[i,j + 3]. Figure 8 on page 73 shows the array configuration at the end of the fourth cycle.

3.3.1.4 Final configuration

Figure 9 on page 74 shows the configuration of the array at the conclusion of the reconfiguration process. Each cell in the five rows now knows of the faulty cell[i,j]. However, only those cells in the two adjoining rows, [i-1] and [i-2], and row[i] and in column[j] or east in those three rows have an active part in the reconfiguration process. All cells to the west of column[j] in row[i-1], cell[i-1,k] | k < j, will be in algorithm configuration 1-28, with a fault to the SE. Cell[i,k] |k < j-1 will be in configuration 1-28 with a fault to the FE and cell[i + 1,k] | k < j will be in configuration 1-28, with a fault to the NE. These cells will maintain their normal north/south communications links. Cell[i-1,k] | k > = j will be in configuration 1-30, fault to the S or SW. Cell[i,k] | k > j has configuration 3-29, and cell[i + 1,k] | k > = j has configuration 2-28.

We have shown that an array which uses algorithm 2-10 for reconfiguration can tolerate all single faults.

■.

To simplify further discussion, references to the fault register will mean the reconfiguration register.

3.3.2 Double Faults

Occurrences of two faults can exist as:

1. Two faulty cells in the same row but not adjacent cell

2. Single faults, or a 1-1 configuration in which the faults are separated by two or more rows

3. Single faults in a 1-1 configuration in adjacent rows

4. Single faults in a 1-1 configuration separated by one row

5. Two faults in the same row and in adjacent cell

Theorem 1-2: An array equipped with hardware to support algorithm 2-10 will tolerate all two-fault occurrences except those of case 2-5, in which adjacent cells are faulty.

Proof: Faults in adjacent cells cannot be handled because communications links to bypass two faulty cells are not available. All others are handled as listed below.

3.3.2.1 Case 2-1, Two Faults in One Row.

For the purpose of discussion we will first assume the faults occur simultaneously in cell[i,j] and cell[i,k] where $k > j + 1$. Figure 10 on page 75 shows the configuration of the array at the end of the cycle in which the two faults occurred. The four neighbors of cell[i,j], cell[i,j-1], cell[i,j + 1], cell[i-1,j] and cell[i + 1,j] and the neighbors of cell[i,k] behave as if only their neighbor was faulty, and the reconfiguration process begins as two separate instances of a single fault as described previously. Figure 11 on page 76 shows the results of the first reconfiguration step and Figure 12 on

page 77 shows the results of the second step. The initial reconfiguration steps are executed as if each fault were a single fault, as described in the previous section.

At some point in time, t = n, cell[i,j + n] will shift data into its reconfiguration register indicating the existence of both faulty cells. Its northern and southern neighbors will have fault register contents that are still null, or contain no information about the western fault. Cell[i,j + n] will be in algorithm configuration 13-40, having faults to its FE and FW. It will maintain its normal north/south connections. Because cell[i,j + n] has received information about both faults, it will make no state change. Figure 13 on page 78 shows the configuration of the array at the end of this reconfiguration step.

At time t = n + 1, cell[i-1,j + n] and [i + 1,j + n] will gate information about both faults into their reconfiguration register. Cell[i-1,j + n] will be in configuration 1-46, SE and SW, and will maintain its southern link. Cell[i + 1,j + n] will be in configuration 19-28 with faults to its NE and NW and will also maintain its north/south links. In the same clock cycle, cell[i,j + n-1] will gain knowledge of the fault to the east, and will be in configuration 13-40. The configuration of this time cycle is completed when cell[i,n-1] shifts the contents of its western state register to its own state register, reverting to its state before the faults occurred. Figure 14 on page 79 shows the configuration of the array at the end of cycle t = n + 1.

Figure 15 on page 80 shows the configuration of the array at the end of t = n + 2. Cell[i,n-2] has gained knowledge of the fault in cell[i,k] and enters configuration 13-40. At the same time it takes on the state of its western neighbor, which was the state it held before reconfiguration began.

Configuration to the west of cell[i,j + n] proceeds, with cells returning to north/south links as they gain knowledge of the eastern fault until time t = m, when cells [i,j-1], [i-1,j] and [i + 1,j] obtain data about the fault in cell[i,k]. At the end of this configuration cycle, cell[i-1,j] will be in configuration 1-47, cell[i + 1,j] will be in configuration 21-28 while cell[i,j-1] will enter configuration 14-42, having faults E and FE. Cell[i-1,j] will connect north and south-west, cell[i,j-1] will connect north-east and

south-east, cell[i + 1,j] will connect north-west and south. These three cells are now part of logical column[j]. Cell[i,j-1] will complete the operations of this cycle by adopting the state of its western neighbor, cell[i,j-2]. Figure 16 on page 81 shows the configuration of the array at the end of time t = m.

Figure 17 on page 82 shows the final configuration of the array after two faults occur in one row. Cells to the east of cell[i,j] have configured as if cell[i,j] were not faulty, and those west of row[j] have configured to the west, with the states of all cells west of column[j] shifted one cell to the west. Cell[i,m] | m < j is in configuration number 14-42 with the MF bit set, but no faults to the W or FW. Cell[i-1,m] | m < = j is in configuration pattern 1-46, with the MFS bit set and no fault to the SW, and sets its communications links to the north and south-west. The contents of the fault register of cell[i + 1,m] | m < = j places it in configuration 21-28, because the MFN bit is set and there are no faults to the NW. Cell[i,m] | j < m < k has fault pattern 13-40. Cell[i,m] | m > k knows only of faults to its far-west and is in configuration pattern 14-42. Cell[i-1,m] | j < m < k has configuration 1-46, while cell[i-1,m] | m > = k has configuration 1-48. Cell[i + 1,m] | j < m < k is in algorithm pattern 19-28 and cell[i + 1,m] | m > = k has configuration 20-28. The complete list of state transitions for row[i] is shown below. (F) indicates a faulty cell, with state "S" being a spare cell.

time	states of row[i]											
t = 0-1	S	A	B	C	D	E	G	H	I	J	K	S
t = 0	S	A	B	f	D	E	G	H	I	f	K	S
t = 1	S	A	B	f	C	E	G	H	I	f	J	S
t = 2	S	A	B	f	C	D	G	H	I	f	J	K
t = n	S	A	B	f	C	D	G	H	I	f	J	K
t = n + 1	S	A	B	f	C	E	G	H	I	f	J	K
t = n + 2	S	A	B	f	D	E	G	H	I	f	J	K
t = m	S	A	C	f	D	E	G	H	I	f	J	K
t = m + 1	S	B	C	f	D	E	G	H	I	f	J	K
t = fnl	A	B	C	f	D	E	G	H	I	f	J	K

Now we assume the two faults were sequential, with the fault in cell[i,j] occurring earlier, and configuration around that fault already completed. Figure 18 on page 83 shows the configuration process that has completed at time t = 1. Cells to the east of the new fault in cell[i,k] will not be affected and will retain the same configurations they had in the description above where the faults

occurred simultaneously. Cell[i,k-1] learns of the second faulty and changes its communications links to the north and south. Because cell[i-1,k-1] and cell[i + 1,k-1] were neighbors of the faulty cell[i,k], they have also learned of the fault, and configure to connect to cell[i,k-1]. Figure 19 on page 84 shows the results of the second reconfiguration step. The knowledge of the second fault has propagated one column to the west, so cell[i-1,k-2], cell[i,k-2] and cell[i + 1,k-2] now take on the same configurations as their neighbors in column[k-1]. This process proceeds until cell[i-1,j], cell[i,j-1] and cell[i + 1,j] learn of the fault in cell[i,k]. This is identical to time t = m in the previous description.

If cell[i,k] is the first faulty cell, Figure 20 on page 85 shows the reconfiguration process will begin at t = m of the description of the simultaneous occurrence. All discussions will now assume simultaneous fault occurrence.

3.3.2.2 Case 2-2, Two faults, only one per row.

Figure 21 on page 86 shows the configuration of the array after configuration around two faults which are separated by two rows. These faults are treated as two distinct instances of single faults which was discussed in a previous section.

3.3.2.3 Case 2-3 Single Faults in adjacent rows.

There are two possibilities for single faults in adjacent rows at cell[i,j] and cell[i + 1,k]. In case 2-3a, j > k and in case 2-3b, j < = k. Figure 22 on page 87 shows the first possibility. The rows involved in reconfiguration around this pattern are [i-1], [i], [i + 1] and [i + 2]. Rows [i-1] and [i + 2] are identical to row[i-1] and row[i + 1] in the single fault discussion. In row[i], cell[i,m] | m < k is placed in configuration 1-28 and maintain their north/south connections. Cell[i,q] | k < = q < j

enters configuration number 1-30, with faults to the far-east and south or south-west. For cell[i,q] | q > j, the configuration pattern is 3-28, west and south-west.

In row[i + 1], cell[i + 1,q] | q < k-1 maintains its north/south links in configuration number 1-28, while cell[i + 1,q] | k < q < = j is in configuration pattern 3-39 and connects to the north-west and the south-west. The final configuration used in this case is 1-29, with faults to the north-west and the far-west, adopted by cell[i + 1,q] | q > j. These cells connect to the north and south-west.

Figure 23 on page 88 shows case 2-3b, j < k. Cell[i,q] | j < q < + k connects in configuration pattern 3-29 and cell[i,q] | q > k in pattern 3-28. Cell[i + 1,q] | j < = q < k has configurations 2-28 and cell[i + 1,q] | q > k has pattern 1-28. All other cells are identical to case 2-3a. The contents of the fault-registers are shown under each cell in Figure 23 on page 88.

3.3.2.4 Case 2-4, Single faults separated by one fault-free row.

Two possibilities exist for case 2-4, the fault in the northern row is east of the southern fault, and the fault in the northern row is west of the southern fault. Figure 24 on page 89 shows the first possibility. In this instance, faulty cell[i,j] is east of faulty cell[i + 2,k]. Cells in row[i-1] and row[i] are not affected by the fault in row[i + 2]. These cells have the same configurations as row[i] and row[i-1] in case 1-1. Row[i + 3] cells and those in row[i + 2] are not affected by the fault in cell[i,j].

Cell[i + 1,q] | q < k is in configuration 1-28, cell[i + 1,q] | k < = q < j in configuration 1-30 and cell[i + 1,q] | q > j is in configuration 2-30. There are no cell state changes in row[i + 1], and those in row[i] and row[i + 2] are described previously in the single fault proof.

Figure 25 on page 90 shows the second configuration in case 2-4. Again only row[i + 1] is unique to this case, and no state changes are required. Faults are assumed to have occurred in cell[i,j] and [i + 2,k]. Cell[i + 1,q] | q < j will be in configuration 1-28 and configure north and south.

Cell[i + 1,q] | j < = q < k has configuration number 2-28 and connects to the north-east and the south, while cell[i + 1,q] | q > = k assumes configuration number 2-30 and connects to the north-east and the south-east. As before, no state changes are required in row[i + 1]. Case 2-4a and 2-4b are mirror images of each other.

We have shown that an array equipped to use algorithm 2-10 will tolerate all two fault occurrences except those that occur in adjacent cells.

■

3.3.3 Three Faults

Three faults which can be tolerated can occur as either:

1. 1-1-1

2. 1-2, with a single fault in one row and double faults in another row.

Theorem 1.3: A cellular array equipped to use algorithm 2-10 will tolerate all cases of three faults except those cases in which two are adjacent in the same row or three faults are in the same row.

Proof: If the faults occur as a 1-1-1, case 3-1, and they are separated by one or more rows, or if two of the faults occur in adjacent rows, but are separated from the third by one or more rows, reconfiguration will be handled exactly as in single and double cases already described. The only possibility not described so far is if the faults occur in rows [i], [i + 1] and [i + 2]. Figure 26 on page 91 shows an example of this configuration. Cells in row[i-1] will configure as they did in case 1-1, with their configuration depending upon whether they are west of faulty cell[i,j] or not. Cells in row[i + 3] will also configure the same as row[i + 3] in case 1-1. The configuration for cells in row[i]

and row[i + 2] will be the same as those of row[i] and row[i + 1] in case 2-3a and 2-3b in which the two faults occurred in adjacent rows. Only row[i + 1] will be unique to this configuration.

3.3.3.1 Case 3-1

Six possible configurations exist for three faults in adjacent rows. If the faulty cells are cell[i,j], cell[i + 1,k] and cell[i + 2,ℓ], then the possibilities, using cell[i + 1,k] as a reference are:

a. $j <= k <= \ell$
b. $j < \ell < k$
c. $\ell <= j < k$
d. $\ell >= j > k$
e. $j > \ell > k$
f. $j > k > \ell$

Figure 26 on page 91 shows case 3-1.a. Cell[i + 1,m] | m < j is in configuration number 1-28, and remains connected to the north and south. If $j <= m < k$ the cells have configuration number 2-28, with faults N or NW and not SW. These cells connect north-east and south. Cell[i + 1,m] | $k < m <= \ell$ will connect to the north and south-west because it is in configuration 1-29 while cell[i + 1,m] | | $m > \ell$ has a configuration of 1-28 and connects to the north and south. Cells in row[i-1] are affected only by the fault in cell[i,j] and configure as they did in case 1-1, the single fault. Cell[i,m] is not affected by the fault in cell[i + 2,ℓ] and configures the same as the same cell in case 2-3, as does cell[i + 2,m] because they are not affected by the fault in cell[i,j]. Cell[i + 3,m] is affected only by the fault in cell[i + 2,l] and configures identically to cells in the same position relative to a single fault in case 1-1. State changes for the three rows containing faulty cells is identical to those of case 1-1.

Figure 27 on page 92 shows case 3-1.b, in which cell[i,j] is west of cell[i + 2,ℓ] which is west of cell[i + 1,k]. As in the previous case, cells in row[i-1], row[i], row[i + 2] and row[i + 3] configure as case 1-1 single faults, case 2-3b, case 2-3a and case 1-1 respectively. Only row[i + 1] is unique to

this fault pattern. Cell[i + 1,m] | m < j is in pattern 1-28, cell[i + 1,m] | j < = m < ℓ has pattern 2-28 and cell[i + 1,m] | ℓ < = m < k is in configuration number 2-30. Figure 28 on page 93 shows an array in fault condition 3-1.c, in which cell[i + 2,ℓ] is west of cell[i,j] which is west of cell[i + 1,k]. In this configuration, cell[i + 1,m] | ℓ < = m < j connects to the north and south-east, being in configuration 1-30. Cell[i + 1,m] | j < m < k is in algorithm configuration pattern number 2-30. Again, cells in row[i + 1] which are east of all three faults by more than one cell are in pattern 1-28. Figure 29 on page 94 shows case 3-1.d. In this configuration, cell[i + 1,m] | m < k is in configuration 1-28, cell[i + 1,m] | k < m < = j fits algorithm pattern number 3-29 and cell[i + 1,m] | j < m < = ℓ configures in pattern number 1-29. Cell[i + 1,m] | m > ℓ is east of all three faults and fits algorithm configuration 1-28.

Figure 30 on page 95 shows one possibility for case 3-1.e, in which the faults in rows [i] and [i + 2] are east of cell[i + 1,k]. With this fault pattern, all cells west of column[k], cell[i + 1,m] | m < k are in algorithm configuration number 1-28. Cell[i + 1,m] | k < m < = ℓ has configuration pattern 3-29 and cell[i + 1,m] | ℓ < m < = j fits algorithm configuration number 3-28 and connects north-west and south. Cell[i + 1,m] | m > j fits configuration pattern 1-28 and connects north and south.

Figure 31 on page 96 shows a fault pattern in which the fault in row[i] is east of the fault in row[i + 2], j > k > ℓ. This constitutes case 3-1.f. In this configuration, cell[i + 1,m] | m < ℓ fits pattern 1-28, cell[i + 1,m] | m > j fits 1-28, and cell[i + 1,m] | ℓ > = m < k has algorithm configuration number 1-30. Cell[i + 1,m] | k < m < = j fits algorithm pattern number 3-28.

3.3.3.2 Case 3-2, Two faults in one row.

In case 3-2, two faults occur in one row, while the other fault occurs as a single fault in another row. If the rows are separated by two or more rows, the faults are independent and reconfiguration in the single fault neighborhood proceeds as if that were the only fault. The configuration around the two faults proceeds as described in the two fault case 2-1. Case 3-2.a exists if the single fault is in

either row[i] or row[i + 4] and the double fault is in row[i + 2]. These instances are the mirror image of each other, so only one will be discussed, the single fault in row[i]. Case 3-2.b has the single fault in row[i] or row[i + 2] and double faults in row[i + 1]. Again only the first case will be discussed.

Figure 32 on page 97 shows one instance of case 3-2.a. For this case, the configuration patterns of rows [i-1], and [i] are the same as those of a single fault in row[i] and no faults in row[i + 2]. Those in rows[i + 2] and [i + 3] do not depend upon the fault in row[i] and are the same as if only the double fault in row[i + 2] existed. Again, only the configuration of row[i + 1] is unique.

Two configurations may exist for case 3-2.a. Case 3-2.1a will consist of a single fault in cell[i,j] and double faults in cell[i + 2,k] and cell[i + 2,ℓ], with $j < k < \ell$. Figure 32 on page 97 shows that cell[i + 1,m] | m < k connects to the north and south-west, being in algorithm configuration number 1-47 because the MFS bit is set in row[i + 1]. Cell[i + 1,m] | $j <= m <= k$ is in configuration number 2-47 and connects north-east and south-west. Cell[i + 1,m] | $k < m < \ell$ has faults to its NW, SW and SE, so it is in configuration 2-46. Cell[i + 1,m] | $\ell < m$ has configuration number 2-48 because it has no faults to its SE, so it connects to the north-east and south-east.

Other examples of case 3-2.a may have the single fault in any location in row[i] or row[i + 4]. In all cases, the northern connection of row[i + 1] (the southern connection in row[i + 3]) depends only upon whether the cell is east or west of cell[i,j](cell[i + 4.j]). The southern connection in row[i + 1] (northern connection for row[i + 3]) cells remains identical to those defined in this case.

3.3.3.3 Case 3-2.b

Case 3-2.b occurs if the single fault is in the row adjacent to the double faults. Figure 33 on page 98 shows an example of this pattern. The faults exist in cell[i,j], cell[i + 1,k] and cell[i + 1,ℓ] with $\ell > k + 1$. As in the cases above, this configuration would also exist if the single fault were in cell[i + 2,j], but this case will not be discussed because of symmetry.

In example 1 of case 3-2b, the single fault lies west of the double fault, ie. $j <= k$. In this configuration, bits in the fault register are set called FCR, FCRN and FCRS, fault-in-critical-region. When this occurs with a single fault in a row, the configuration around this fault must be reversed and moved to the west. Cell[i-1,m] | $m <= j$ has the FCRS bit set and is in configuration number 1-35 and connects north and south-west. Cell[i-1,m] | $m > j$ has configuration 1-34 and connects north and south. Cell[i,m] | $m < j$ has the FCR and MFS bits set placing it in configuration number 5-49. Cell[i,m] | $j < m <= k$ still has the FCR and MFS bits set, but has its fault to the west, placing it in configuration number 4-50. Cell[i,m] | $k < m < \ell$ has configuration 5-49 and cell[i,m] | $\ell <= m$ has configuration 5-51.

Cells in row[i + 1] have the FCRN and the MF bits set. Cell[i + 1,m] | $m < j$ has configuration 16-42. Cell[i + 1,m] | $j <= m < k$ configures in pattern number 17-42. Cell[i + 1,m] | $k < m < \ell$ has configuration number 16-40, while cell[i + 1,m] | $m > \ell$ has algorithm configuration number 18-41. The cells in row[i + 2] configure identically to the double-fault case 2-1.

Figure 34 on page 99 shows case 3-2.b, which exists if the single fault is in cell[i,j] with $j > k$. In this example, no critical-region faults exist. Cells to the west of cell[i,j] in any of the rows involved are handled as the same cells would be in case 2-1. The single fault to the east or north-east have no affect on cells in rows [i] and [i + 1]. Those cells east of column[j-1] are not affected by the western most of the double faults, cell[i + 1,k]. Reconfiguration to the east of this column is handled the same as in case 2-3, single faults in adjacent rows.

Cells in row[i-1] have the configuration they had in case 1-1, with only a single fault. Cells in row[i + 2] have the configuration they would have had if the single fault did not exist, case 2-1. Cell[i,m] | $m < k$ has configuration 1-47, cell[i,m] | $k < m < j$ has configuration 1-46, and cell[i,m] | $j <= m <= \ell$ configures in pattern 3-47. Cells to the east of all faults, cell[i,m] | $m > \ell$ have fault neighborhoods which place them in configuration number 3-46.

Cell[i + 1,m] | m < k configures in patterns 14 and 42, cell[i + 1,m] | k < m < j uses pattern 13-40 and cell[i + 1,m] | j < = m < ℓ connects in pattern 14-40. The remaining cells, cell[i + 1,m] | m > ℓ connect in pattern 14-41.

This ends proof that an array equipped to use algorithm 2-10 will reconfigure all triple faults except those with adjacent faults in one row, or three faults in one row.

■

3.3.4 Four Faults

Theorem 1-4: An array equipped to implement algorithm 2-10 will tolerate all occurrences of four faults except:

- Occurrences in which any two of the four faults are in adjacent cells in the same row.
- Occurrences in which three faults occur in one row.
- Occurrences in which two faults occur in each of adjacent rows, and there is no overlap in the fault patterns. This occurs when both faults in one row are to the east of both faults in the second row.
- Occurrences in which all four faults are in one row.

Proof: More than two faults in any one row cannot be tolerated because only two spare cells exist in each row. The pattern in which two double-fault-per-row conditions exist in adjacent rows cannot be tolerated if there is no overlap in the fault patterns. This can be shown by assuming the western fault of one of the patterns exists in cell[i,j]. If the columns are numbered beginning at zero, then there are "j" cells to the west of cell[i][j]. All of these must be used in any successful recon-

figuration process. Moreover, cell[i-1,j] must form a north-south link with cell[i,j-1]. This means that there are still j-1 cells which must be connected into the pattern. However, if two of the cells to the west of cell[i-1,j] are faulty, there are only j-2 available fault-free cells in row[i-1], meaning only j-1 columns can be configured to the west of cell[i,j].

Fault patterns which can be tolerated occur in:

1. 1-1-1-1

2. 1-1-2

3. 2-2

Case 4-1, 1-1-1-1 is identical to case 3-1, in which three single faults occurred. If the faults occur in non-consecutive rows, the analysis and configuration patterns are identical to case 3-1. If they are in adjacent rows, row[i], row[+ 1], row[i + 2] and row[i + 3], then the analysis is identical to case 3-1 in which three adjacent rows had faults. In case 4-1, rows [i + 1] and [i + 2] will have the same configurations as row[i + 1] in case 3-1, while row[i + 3] is identical in configuration to row[i + 2] in case 3-1. Row[i] is identical in both cases.

Case 4-2 has single faults in two rows and a double fault in another row. If the rows are all separated by two fault-free rows, the reconfiguration process is handled as described in the single fault case and the double fault case 2-1. If one of the single faults is in the adjacent to the double fault, and the other is at least two rows away from either of the two rows, the double and adjacent single fault is configured as in case 3-2 with a single and double fault and the other single fault is treated as a single fault. The only configurations for case 4-2 which cannot be reduced to previously discussed cases are those in which single faults are adjacent to the double fault, that is faults in row[i], row[i + 1] and row[i + 2]. The double fault may be in any of these three rows, and the singles in the other two rows. Case 4-2a is a 1-1-2 pattern, with single faults at cell[i,j] and cell[i + 1,k], with the double fault at cell[i + 2,ℓ] and cell[i + 2,m]. Figure 35 on page 100 shows one instance of this case,

in which both single faults are west of the double fault. In this example, $j < k < \ell$, forces a critical fault designation for cell[i + 1,k] because it is west of the double fault. A new case is introduced, that of the FCR created when cell[i,j] became faulty. Because this cell is west of a critical fault, and it is a single fault, it must also be declared a critical fault because configuration around it must be to the west. The FCR bit in this case is generated in row[i] because FCRS is set, and a faulty cell exists to the east of cell[i,j-1], and no faulty cell exists to the south or south-west in row[i + 1]. This generates FCRS bits in row[i-1] and FCRN in row[i + 1]. Row[i] now has FCR and FCRS bits set while row[i + 1] has FCR and FCRN bits set. These two bits create a new fault configuration patterns.

Cells in row[i-1] have the same configuration as row[i-1] in case 3-2a. Cell[i,n] | $n < j$ is in configuration 5-37 and connects north and south. Cell[i,n] | $j < n <= k$ is in configuration 4-38, and cell[i,n] | $n > k$ has configuration pattern 4-37.

Cell[i + 1,n] | $n < j$ configures in pattern 10-49. Cell[i + 1,n] | $j <= n <= k$ has a fault neighborhood which places it in configuration 11-49 and cell[i + 1,n] | $k < n <= \ell$ configures in pattern 10-50. Cell[i + 1,n] | $\ell < n < m$ configures as a 10-49 and cell[i + 1,n] | $n >= m$ configures as 10-51. Row[i + 2] and row[i + 3] are identical to row[i + 1] and row[i + 2] of case 3-2.b. Row[i-1] is also identical to row[i-1] in case 3-2.2.

Figure 36 on page 101 shows an example of a configuration in which $j > k$. In this case cell[i,n] | $n <= k$ configures in pattern 1-35 while cell[i,n] | $k < n < j$ configures as a 1-34 and cell[i,n] | $j < n$ has a 3-35 pattern. Cell[i + 1,n] | $n < k$ configures in pattern 5-49 and cell[i + 1,n] | $k < n <= \ell$ satisfies a 4-50 pattern. Cell[i + 1,n] | $\ell < n < j$ is placed in a 4-49 pattern and cell[i + 1,n] | $j <= n < m$ satisfies the requirements for pattern 5-49. Those cells east of column[m], cell[i + 1,n] | $n > m$ configure in a 5-51 pattern. Cells in row[i + 2] and row[i + 3] have identical configurations to the same rows in case 3-2.b.

Figure 37 on page 102 shows an occurrence of case 4-3.1, a 2-2 configuration in which the faults are separated by one row. If the faulty rows are separated by more than one row, they will configure as separate occurrences of two faults already described in case 2-1. If the faulty cells are cell[i-1,j] and cell[i-1,k] | k > = j + 1) and cell[i + 1,ℓ] and cell[i + 1,m] | m > = 1 + 1, rows [i-2], [i-1], [i + 1] and [i + 2] configure as in case 2-1. Only row[i] is unique to this configuration. Cell[i,n] | n < j fits into configuration pattern 21-47, with MFN and MFS set, and no faults to the NW or SW. Cell(i,n) | j < n < = ℓ has configuration 19-47 and connects north and south-west. If cell[i + 1,ℓ] had been west of cell[i-1,j] the connections would have been the mirror image, being made to the north-west and south instead. Cell[i,n] | ℓ < n < k configures in pattern 19-46 and cell[i,n] | k < = n < m has configuration 20-46. Cells to the east of column[m], cell[i,n] | n > m has pattern 20-48.

Figure 38 on page 103 shows case 4-3.2, in which the double faults are in adjacent rows. The faults occur in cell[i,j], cell[i,k], cell[i + 1,ℓ] and cell[i + 1,m]. Cells in row[i-1] and row[i + 2] configure identically to row[i-1] and row[i + 1] in case 2-1. Only row[i] and row[i + 1] are unique to this case.

Cell[i,n] | n < ℓ has a configuration of MF and MFS and no faulty cell to the west or south-west, number 14-52. Cell[i,n] | ℓ < = n < j has pattern 14-54 and cell[i,n] | j < n < m configures as a 13-52. Cell[i,n] | m < = n < k holds a 13-54 pattern and cell[i,n] | n > k is in configuration 15-52. Cell[i + 1,n] | n < ℓ has configuration number 25-42. Cell[i + 1,n] | ℓ < n < = j configures in a 27-40 pattern and cell[i + 1,n] | j < n < m has a 25-40 pattern. Cell[i + 1,m] | m < n < = k is placed in a 26-41 configuration, with cells to the east of column[k], cell[i + 1,n] | n > k has a fault pattern which dictates a 25-41 configuration. All occurrences of case 4-3.2 can be configured in this manner, as long as there is overlap between the double fault patterns.

It has been shown that an array equipped to implement algorithm 2-10 will tolerate many 4-fault occurrences.

■

3.3.5 Five faults

Theorem 1.5: An array equipped to use algorithm 2-10 can tolerate all instances of five faults which can be reduced to instances of four faults with a single fault in another row.

Proof: Instances of five faults can occur as:

1. 1-1-1-1-1

2. 1-1-1-2

3. 1-2-2

Only case 5-3, with the faults in adjacent rows has a configuration not discussed previously.

Assume the faults occur in cell[i,j], cell[i + 1,k], cell[i + 1,ℓ], cell[i + 2,m] and cell[i + 2,n]. Cells in row[i-1], and row[i] configure as did the same cells in case 3-2. Cells in row[i + 2] and row[i + 3] configure exactly as those in case 4-3.2. Only those in row[i + 1] are unique, having a single fault in the northern row and a double fault in the row to the south. Figure 39 on page 104 shows an instance of case 5-3 in which the single fault is west of the double fault in row[i + 1]. Cells in row[i-1] and row[i] have the same fault patterns as the same rows in case 3-2. Row[i + 1] has the same northern configuration as case 3-2, and the southern configuration of case 4-3.2. Row[i + 2] and row[i + 3] have the same fault configurations as case 4-3.2, row[i + 1] and row[i + 2].

We have proven that an array equipped to use algorithm 2-10 will tolerate many occurrences of five faults.

■

3.3.6 Six faults

Theorem 1-6: An array equipped with hardware to use algorithm 2-10 can tolerate all occurrences of six faults which do not have adjacent cells faulty, or three or more faults in one row, or double faults without overlap.

Proof: Six faults can occur as:

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

2. 1-1-1-1-2

3. 1-1-2-2

4. 2-2-2.

In the first three cases, they can be reduced to previously discussed cases. Case 6-1, 1-1-1-1-1-1 configures exactly as case 3-1, with all the interior rows of the distribution behaving as the interior row of the 1-1-1 configuration. Case 6-2, with the single occurrence of a multiple fault configures as did case 4-2, noting the possibility of carrying the critical region faults to the 4 rows with single faults. Likewise, case 6-3, with 1-1-2-2 distribution configures the same as the 1-2-2 case of five faults, with the addition of the single fault in an adjacent row.

Only case 6-4 with a 2-2-2 distribution which occurs in adjacent rows is new. Figure 40 on page 105 shows this case. The faults occur in row[i-1], row[i] and row[i + 1]. Only those in row[i] are unique, the others having been discussed in four and five fault cases. Again, only row[i + 1] has a fault pattern configuration which is new. The cells in row[i + 1] have the same northern configuration as row[i + 1] in case 4-3, and the southern configuration as row[i] in the same fault case.

It has been shown that an array with logic to support algorithm 2-10 will tolerate many six-fault occurrences.

■

3.3.7 Conditions with More Than Six Faults.

Fault conditions with more than six faults in the array can be tolerated only if they exist as one of the conditions previously discussed. For example, seven faults can be tolerated in a 2-2-2-1 configuration, iff the 2-2 combinations can be tolerated, because all 2-1 configurations can be tolerated. Eight faults can be tolerated in a 2-2-2-2 pattern if each of the 2-2 combinations can be tolerated. In the ten by eight active array, the maximum number of faults which can be successfully reconfigured is 20 (Figure 41 on page 106).

3.3.8 Clustered Faults

Fault clusters which are too large to be tolerated by the reconfiguration algorithm can be handled in the following manner. Whenever a cell finds itself without a neighbor on any of its boundaries, for example both its western neighbors are faulty, the fault-free cell ceases to attempt communications with cells in that direction and assumes an s-value[Kumar84b] of 1. This will affect the s-values of surrounding cells which are still active neighbors. This is the same mechanism used to force the cells on the array boundary to assume an s-value of 1.

Figure 42 on page 107 shows one possible method of configuring around the cluster. In this example, cells to the east of the cluster configure as if a single fault existed in the cluster. A pattern grown in this area has the disadvantage that one fault in the row with the cluster is assumed, and only one more can be tolerated. Figure 43 on page 108 shows a second possibility, in which the

cells which surround the cluster would clear their fault register bits which denote faulty cells in the cluster. This configuration has the advantage of still being able to tolerate two faults in the row east or west of the cluster.

3.4 Coverage Analysis

Fault analysis is given here for one through eight faults, for an array with ten rows and eight cells in each row. This combined with the two spare columns composes a ten by ten array.

There are

$$\binom{100}{1} = 100$$

choices of the position of a single fault. The reconfiguration algorithm will tolerate all 100 of these.

Double faults can occur in one of two formats, either single faults occurring in separate rows, or two faults that occur in the same row. The total number of double faults tolerated is the sum of those tolerated in each case. For the single faults, case 2-2 through 2-4, there are

$$\binom{10}{2} \times 10^2 = 4500$$

choices, representing the number of choices of two rows out of ten times the number of choices for the location of a single faulty cell within a row of ten cells. This is squared because there are two rows containing a faulty cell. To this is added

$$36 \times 10 = 360$$

choices for the two fault per row possibility. This represents the 36 possible two-fault-per-row configurations that the algorithm will successfully reconfigure for case 2-1. It will not tolerate the 9 possibilities in which the faults are adjacent. The total number of double faults tolerated is 4860. Of the possible

$$\binom{100}{2} = 4950$$

fault patterns, the algorithm will survive a total of 98% of them.

There are

$$\binom{100}{3} = 161{,}700$$

possible occurrences of three faults in the 100 cell array. These can be broken into the 1-1-1, single fault in three rows category, and the 1-2 category. The tolerance of the algorithm to three faults is the sum of the tolerances of the two categories. For the 1-1-1 category, all of which can be tolerated by the algorithm, there are

$$\binom{10}{3} \times 10^3 = 120{,}000$$

possibilities. The 1-2 tolerance is expressed by the equation

$$\binom{10}{2} \times 10 \times 36 = 16{,}200.$$

The total tolerance for three faults is

$$120{,}000 + 16{,}200 = 136{,}200$$

which gives a tolerance percentage of 84%.

Four faults can be arranged as 1-1-1-1, 1-1-2 or 2-2 occurrences. The 2-2 arrangement must be further broken down to those in which the double faults do not occur in adjacent rows, and those in which they do. This must be done because all 36 of the fault combinations cannot tolerate occurrences of double faults in an adjacent row if there is no overlap in the fault patterns. The number of 1-1-1-1 occurrences is

$$\binom{10}{4} \times 10^4 = 2{,}100{,}000.$$

The 1-1-2 pattern can occur in

$$\binom{10}{3} \times 10^2 \times 36 = 432{,}000$$

ways. The 2-2 patterns can occur either in patterns in which the rows containing the faults are adjacent, or the rows are not adjacent. This yields

$$\left[\binom{10}{2} - 9\right] \times 36^2 = 46{,}656$$

combinations for the non-adjacent configuration and

$$9 \times 1065 = 9585$$

for the adjacent occurrences. The first part of the equation represents 45 ways to choose 2 rows from the 10 available, minus the nine choices which would yield adjacent rows, times the 36 valid choices for placing two faulty cells in one row. Since there are two rows, the 36 is squared. The second part of the equation represents the number of ways to place the two faults in adjacent rows. There are 9 combinations of 2 adjacent rows. The 1065 represents the total combinations of 2-2 in adjacent rows, which is slightly less than the 1296 combinations which can be tolerated if the faults are not in adjacent rows. This difference is created because adjacent double fault conditions must have an overlap condition to be tolerated. Assuming the faults exist in cell[i,j], cell[i,k], cell[i + 1,ℓ] and cell[i + 1,m], either $\ell < j < m$ or $\ell < k < m$. The number of these conditions can be found with the summation

$$\sum_{j=0}^{j=7}\sum_{k=j+2}^{k=9}\sum_{\ell=0}^{\ell=k-1}\sum_{m=9}^{m=9} bove(m=\max(j+1,l+2))1 = 1065.$$

This is a row by row count of the number of ways in which two faults can exist in each of two adjacent rows in the ten by ten array. The total tolerance for 4 faults in the algorithm is 2,100,000 + 432,000 + 46,656 + 9585 = 2,588,241. This gives a tolerance figure for the 3,921,580 possibilities of 66%.

Table 2 on page 66 provides a tabulation of the analysis results. Because of the slight difference between the adjacent and non-adjacent occurrences of double faults, a value of 30 was used in the analysis to give a conservative estimate of tolerances to five or more faults.

The algorithm will tolerate as many as 20 faults in a 10 row array, however there are 3.54×10^{18} possibilities of 20 faults, with the algorithm surviving only 1.6×10^{13} of them. Figure 41 on page 106 shows one possibility for 20 faults.

3.4.1 Survivability Analysis

The MTBF for the 10 by 8 cell array in which each cell has a probability of survival of 0.9999 is

$\int_0^\infty e^{-80 \times 0.0001t} = 125$. The probability of surviving as a function of time for the array without spares is given as $R = e^{-80 \times 0.0001t}$. For the 10 by 10 array with hardware added to implement algorithm 2-10, which is assumed to be small compared to the original size of each cell and adds no significant amount to the failure rate, the probability of survivability is given as

$$R = e^{-100 \times 0.0001t}$$

$$+ 100e^{-99 \times 0.0001t}(1 - e^{-0.0001t})$$

$$+4860e^{-98 \times 0.0001t}(1 - e^{-0.0001t})^2$$

$$+1.36 \times 10^5 e^{-97 \times 0.0001t}(1 - e^{-0.0001t})^3$$

$$+2.58 \times 10^6 e^{-96 \times 0.0001t}(1 - e^{-0.0001t})^4$$

$$+3.73 \times 10^7 e^{-95 \times 0.0001t}(1 - e^{-0.0001t})^5$$

$$+3.77 \times 10^8 e^{-94 \times 0.0001t}(1 - e^{-0.0001t})^6$$

$$+1.56 \times 10^9 e^{-93 \times 0.0001t}(1 - e^{-0.0001t})^7$$

$$+1.18 \times 10^{10} e^{-92 \times 0.0001t}(1 - e^{-0.0001t})^8$$

Table 3 on page 66 shows the results of the computations of the probability of surviving as a function of time for both the 80 cell array without spares and the 100 cell array with spares. These data show that the array without spares has a 36% probability of surviving until its MTBF value of 125. When the spares and the logic to implement algorithm 2-10 are added, the probability of surviving at this same time is approximately 97%. Figure 44 on page 109 shows a plot of these values from t = 0 until t = 250.

3.5 *Cost of Implementation*

To implement algorithm 2-10, an array must add to its communications load the transmission of 31 bits to allow transmission of the bits of the fault register to neighboring cells.

An estimate of the additional gates needed to design the logic for the fault register and the decoder is based upon the code used in the simulation files for each function. Two registers with 24 bits each are needed with four additional latches. The number of gates required for the decoder is estimated to be 102 'and' gates, 146 'or' gates and 66 inverters. This estimate is based upon the code used in the simulator.

3.6 Time for Reconfiguration

The time required to complete reconfiguration using algorithm 2-10 is a function only of the number of cells in each row in the array. In an m by n array, a maximum of n reconfiguration cycles are required after detection of the last fault are required to complete reconfiguration.

Table 1. Fault Register Contents

BIT NO.	DESCRIPTION	PHYSICAL LOCATION OF FAULTY CELL
0	N	[i-1,j]
1	NW	[i-1,k] | 0 < k < j
2	NE	[i-1,k] | k > j
3	S	[i + 1,j]
4	SW	[i + 1,k] | 0 < k < j
5	SE	[i + 1,k] | k > j
6	E	[i,j + 1]
7	FE	[i,k] | k > j
8	W	[i,j-1]
9	FW	[i,k] | 0 < k < j-1
10	NN	[i-2,j]
11	NNW	[i-2,k] | 0 < k < j
12	NNE	[i-2,k] | k > j
13	MFNNE	Two Faults in Row [i-2],
14	SS	[i + 2,j]
15	SSW	[i + 2,k] | 0 < k < j
16	SSE	[i + 2,k] | k > j
17	MFSS	Two Faults in Row [i + 2],
18	MF	Two Faults in Row [i]
19	MFN	Two Faults in Row [i-1]
20	MFS	Two Faults in Row [i + 1]
21	FCR	Fault in Critical Region, That is a Fault Row [i], with MFN and no faults west of this fault in row[i-1], or MFS set and no faults west of this fault in row[i + 1]
22	FCRN	FCR set in row[i-1]
23	FCRS	FCR set in row[i + 1]

Table 2. Coverage Analysis Results

Number of Faults	Possibilities	Covered	Percentage
1	100	100	100
2	4950	4860	98.1
3	161,700	136,200	84.2
4	3.92×10^{6}	2.59×10^{6}	66
5	7.53×10^{7}	3.73×10^{7}	49.5
6	1.19×10^{9}	3.77×10^{8}	31.6
7	1.60×10^{10}	1.56×10^{9}	9.8
8	1.86×10^{11}	1.18×10^{10}	6.3
20	5.35×10^{20}	1.60×10^{13}	0

Table 3. Survivability Probabilities

t	W/O Spares	W/Spares	t	W/O Spares	W/Spares
0	1	1	10	0.923116	0.999894
20	0.852144	0.999517	30	0.786628	0.998793
40	0.726149	0.997661	50	0.67032	0.99607
60	0.618783	0.993983	70	0.571209	0.991371
80	0.527292	0.988214	90	0.486752	0.9845
100	0.449329	0.98022	110	0.414783	0.975375
120	0.382893	0.969966	125	0.367879	0.967053
130	0.353455	0.964002	140	0.32628	0.957492
150	0.301194	0.95045	160	0.278037	0.942889
170	0.256661	0.934827	180	0.236928	0.926281
190	0.218712	0.917272	200	0.201897	0.907818
210	0.186374	0.897941	220	0.172045	0.887661
230	0.158817	0.877001	240	0.146607	0.865981
250	0.135335	0.854624			

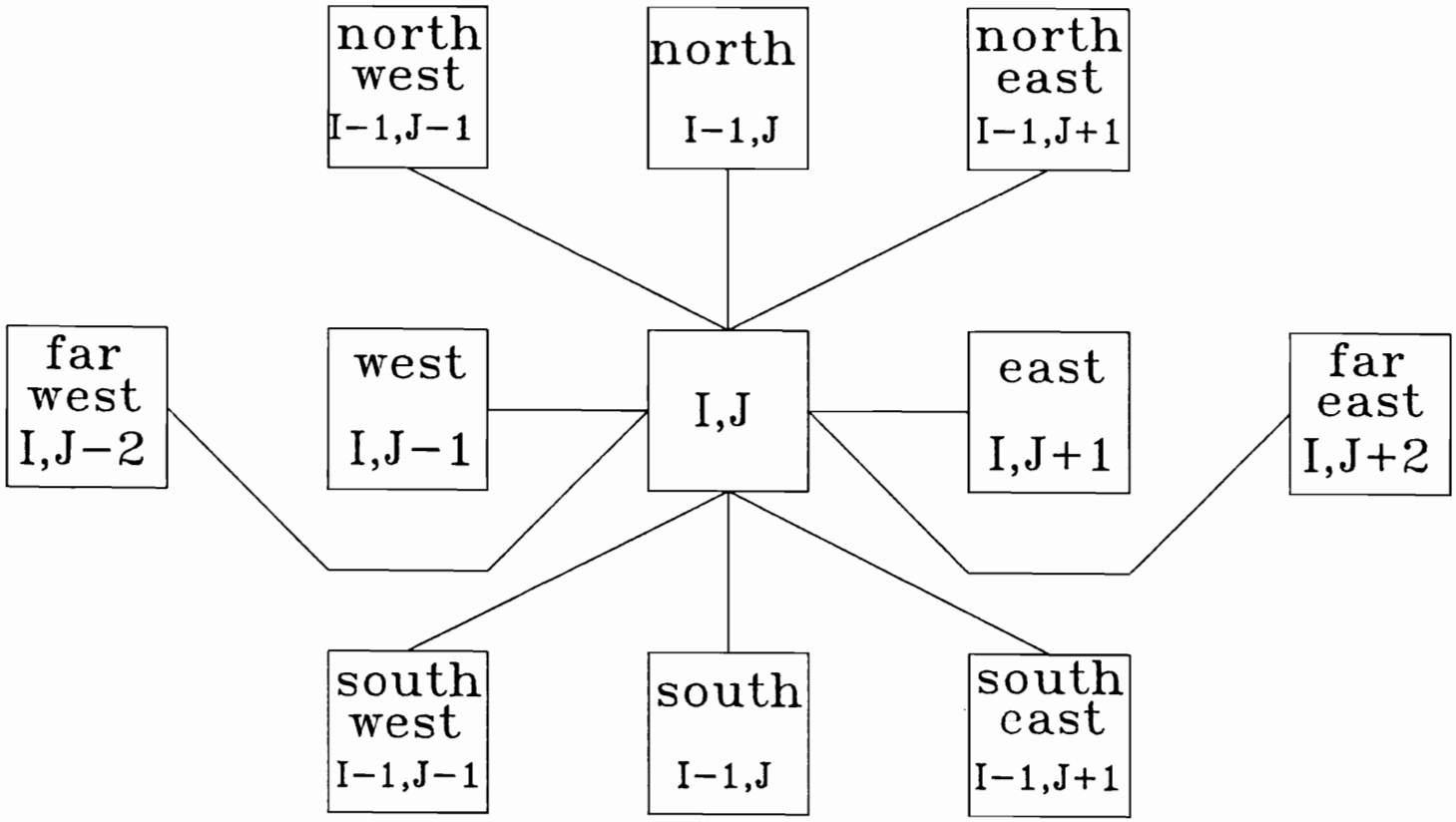


Figure 2. Ten Neighbor Interconnection: cell[i,j] and its ten possible neighbors.

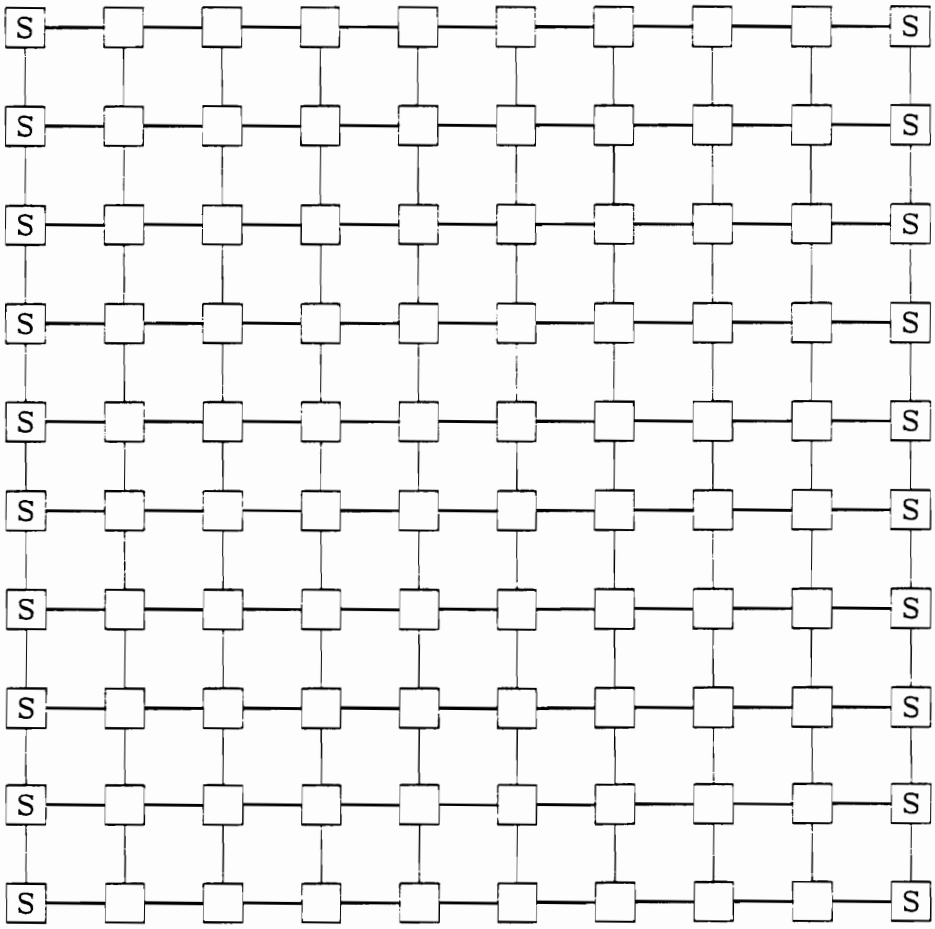


Figure 3. Ten By Ten Array: Ten by Eight Active Array with Two Spare Columns

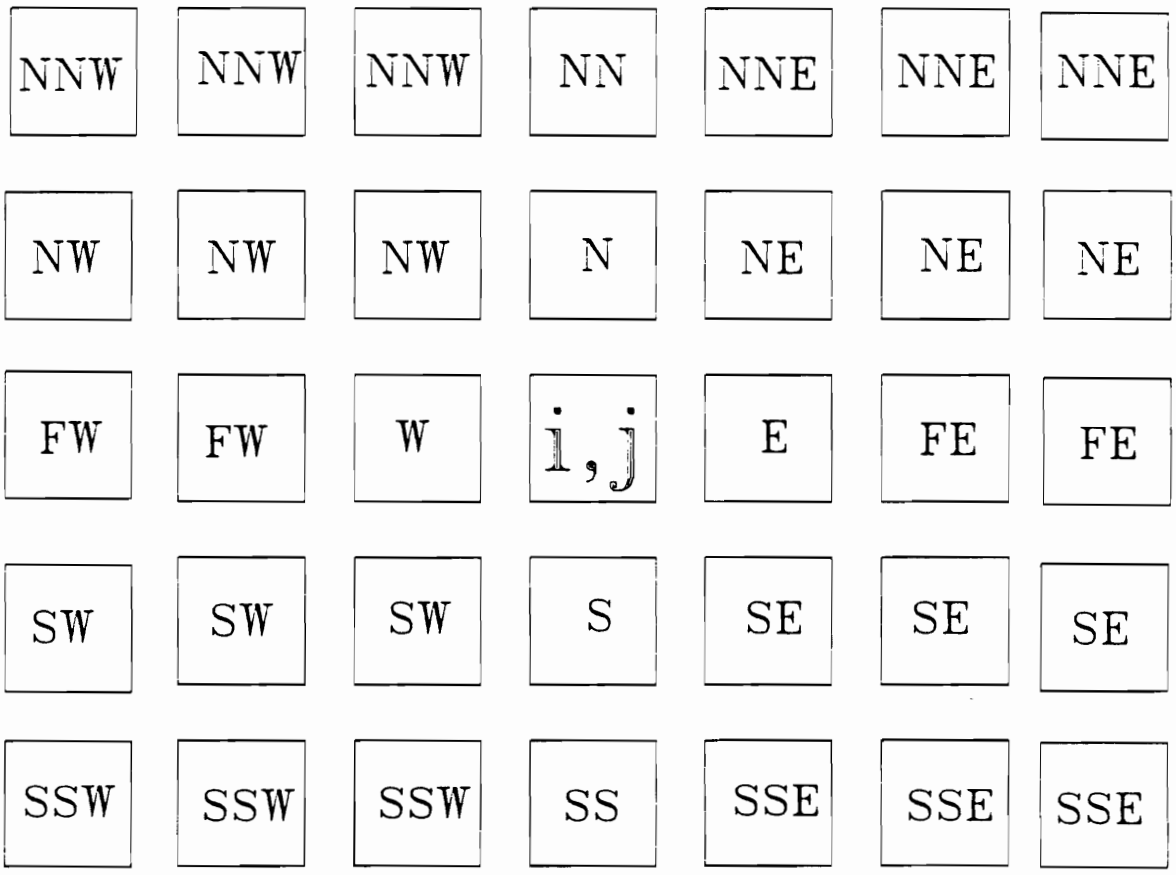


Figure 4. Cell[i,j] and Surrounding Cells: With each cell containing the bit set in cell[i,j] if the cell is faulty.

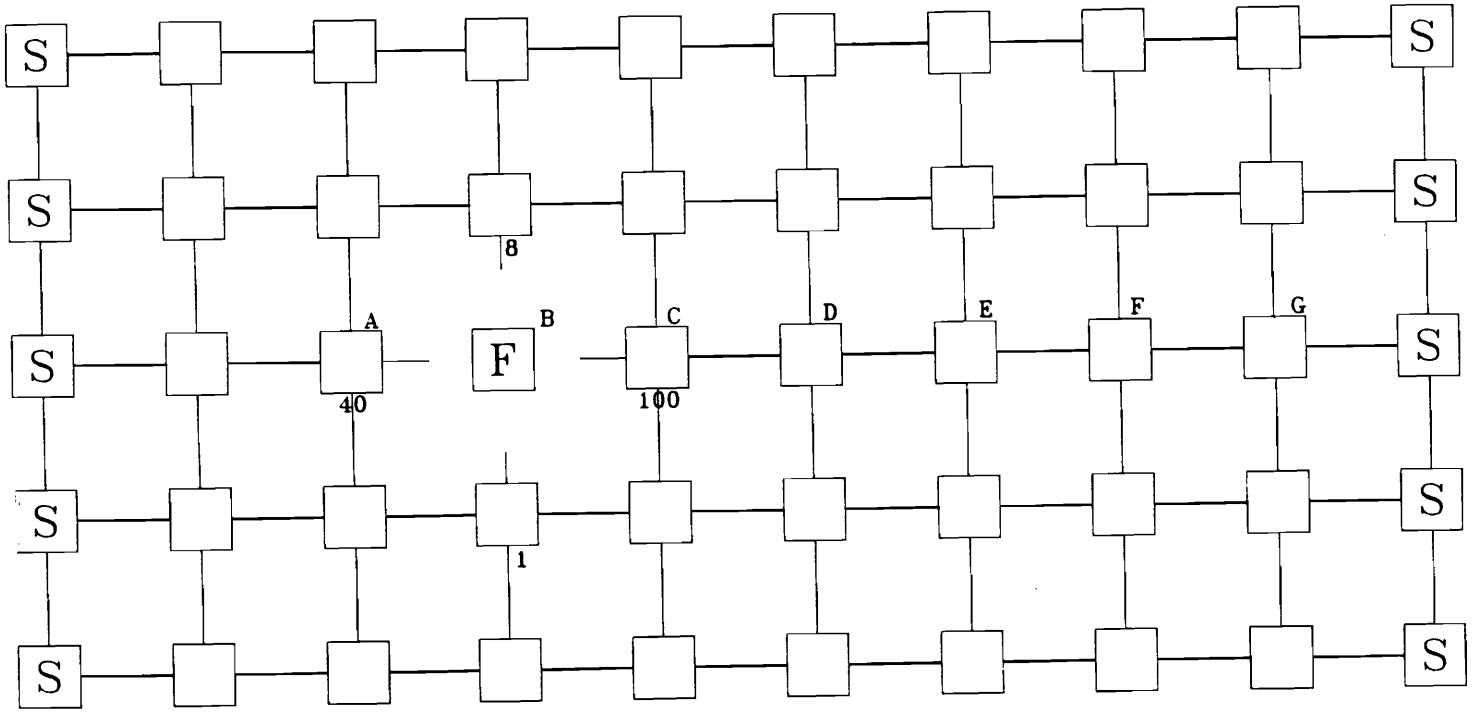


Figure 5. Single Fault Condition Case 1-1: Array showing conditions at the end of t = 0, fault detection time.

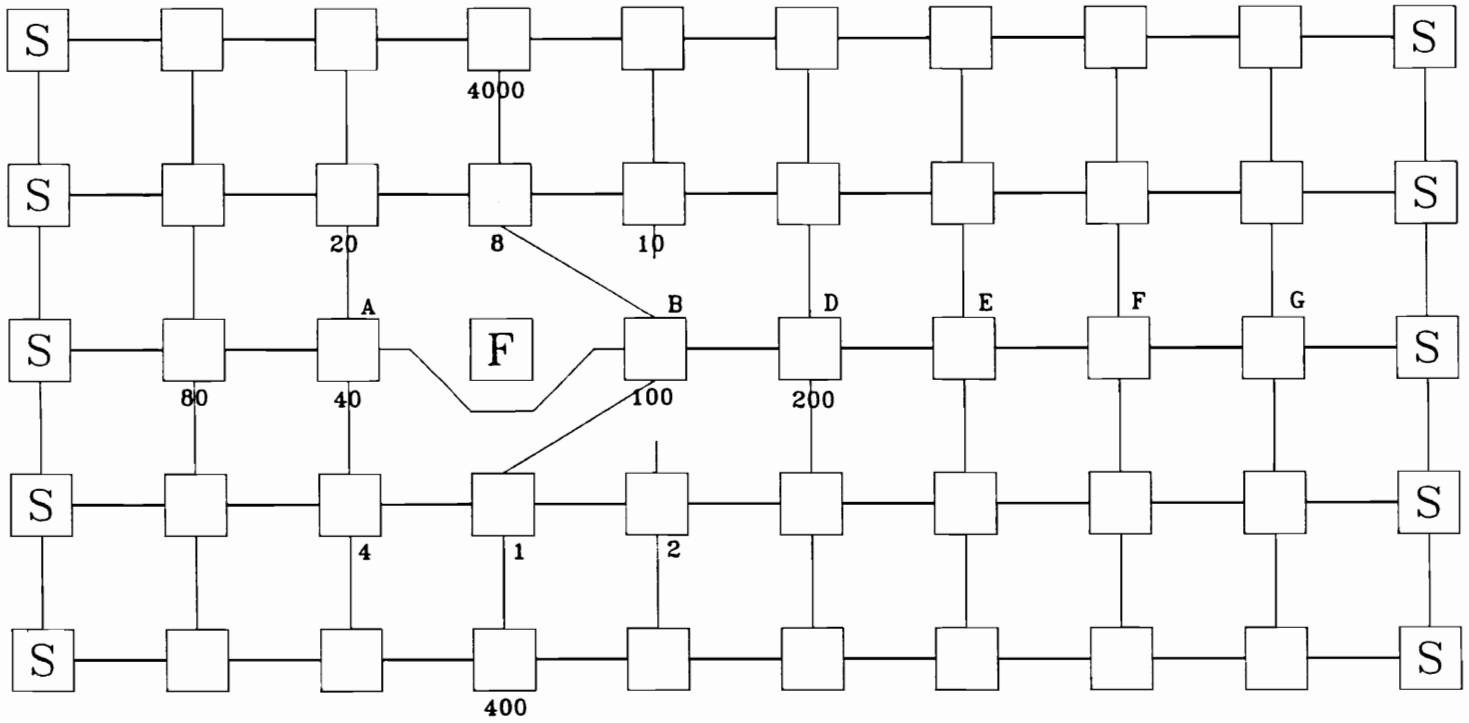


Figure 6. Single Fault Condition Case 1-1: Array showing configurations at t = 1.

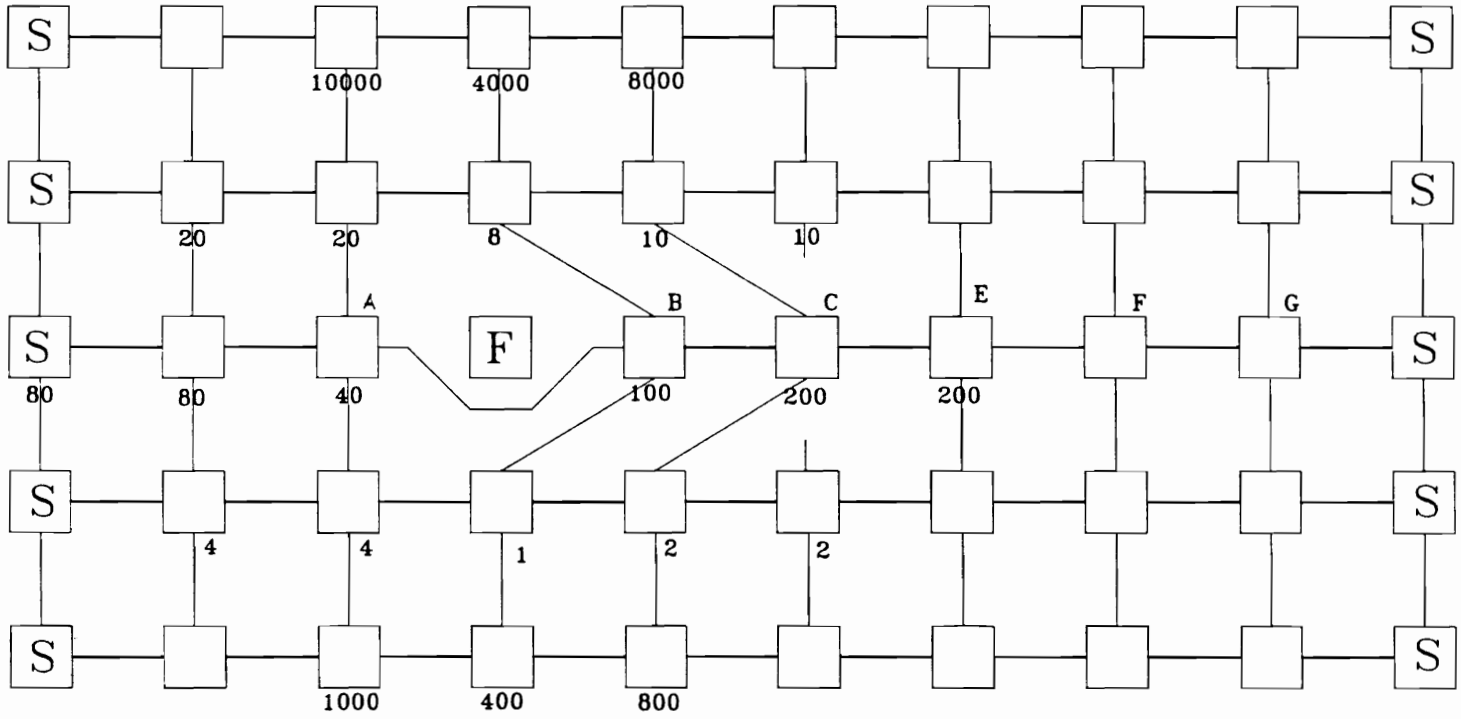


Figure 7. Single Fault Condition Case 1-1: Array showing configurations at t = 2.

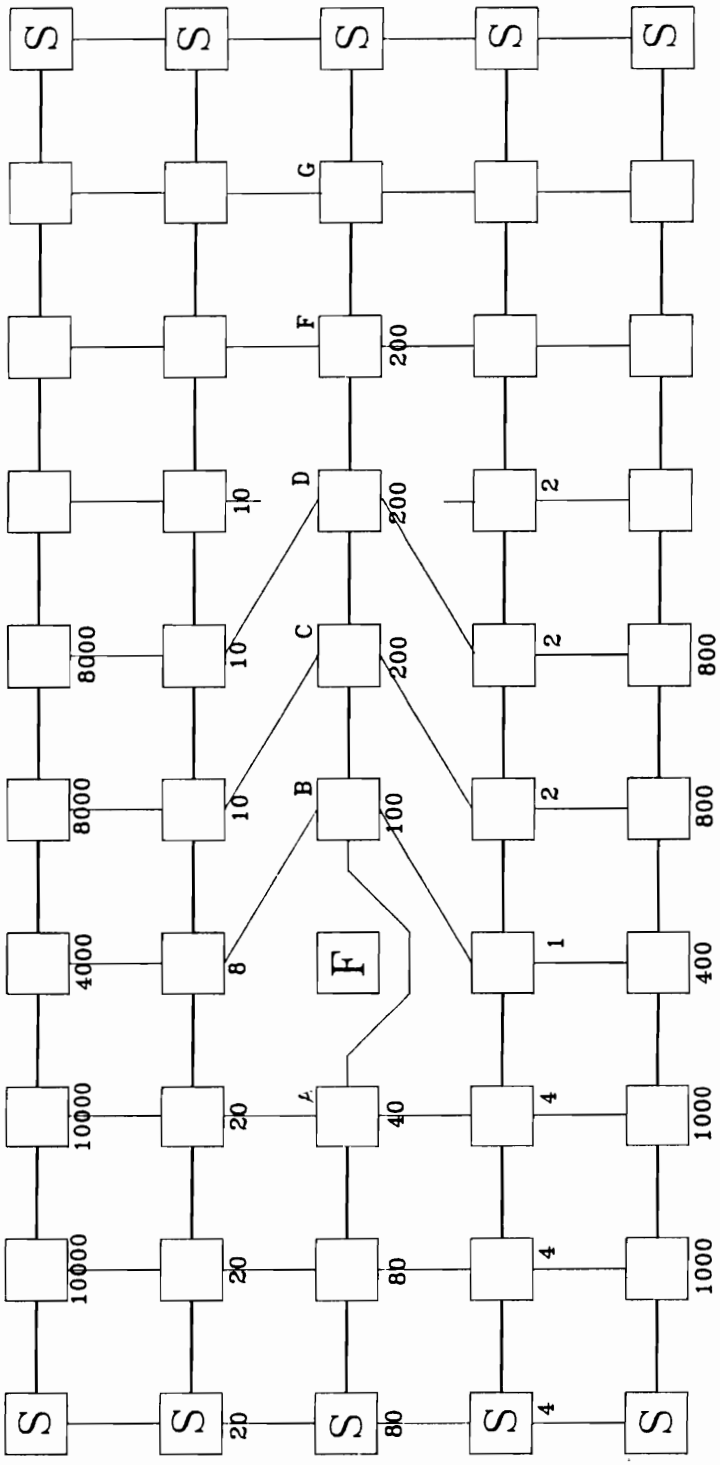


Figure 8. Single Fault Condition Case 1-1: Array showing configuration at t = 3.

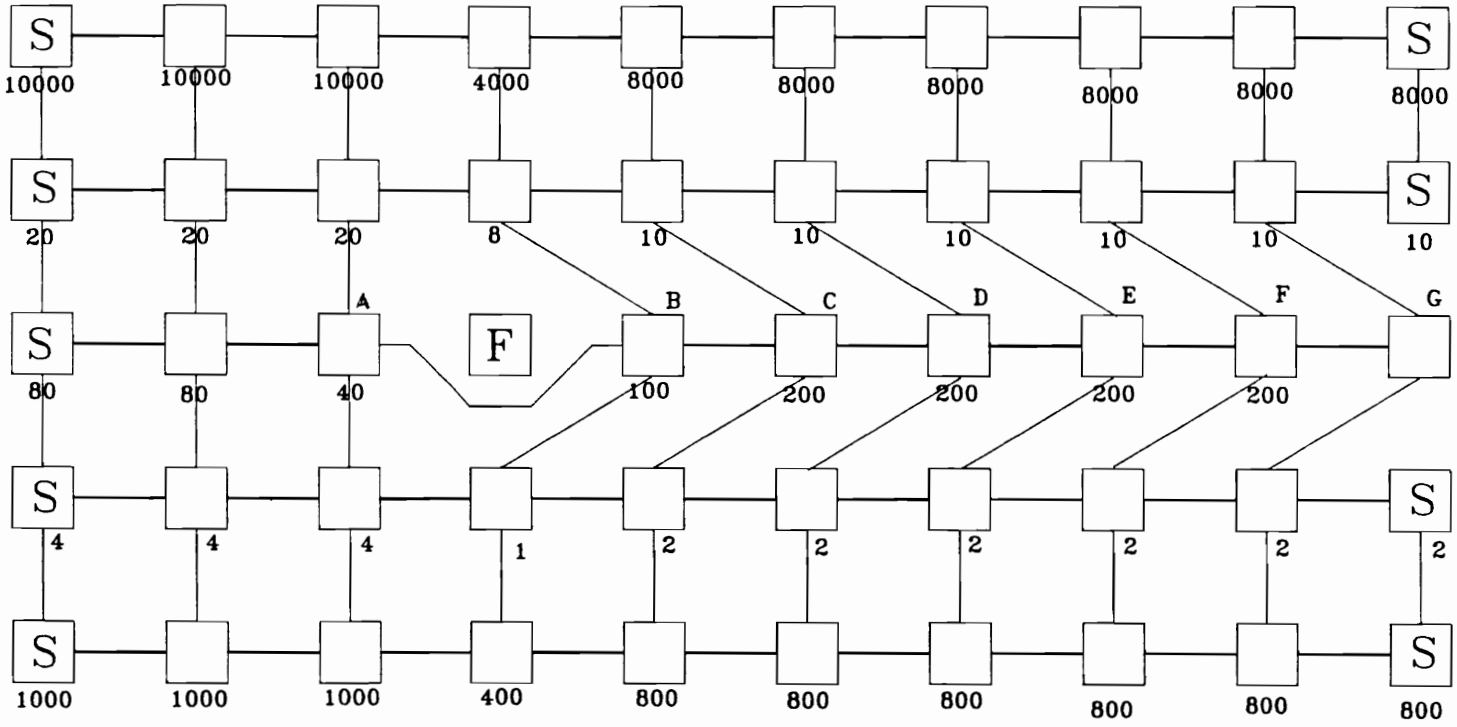


Figure 9. Single Fault Condition Case 1-1: Array showing final configuration.

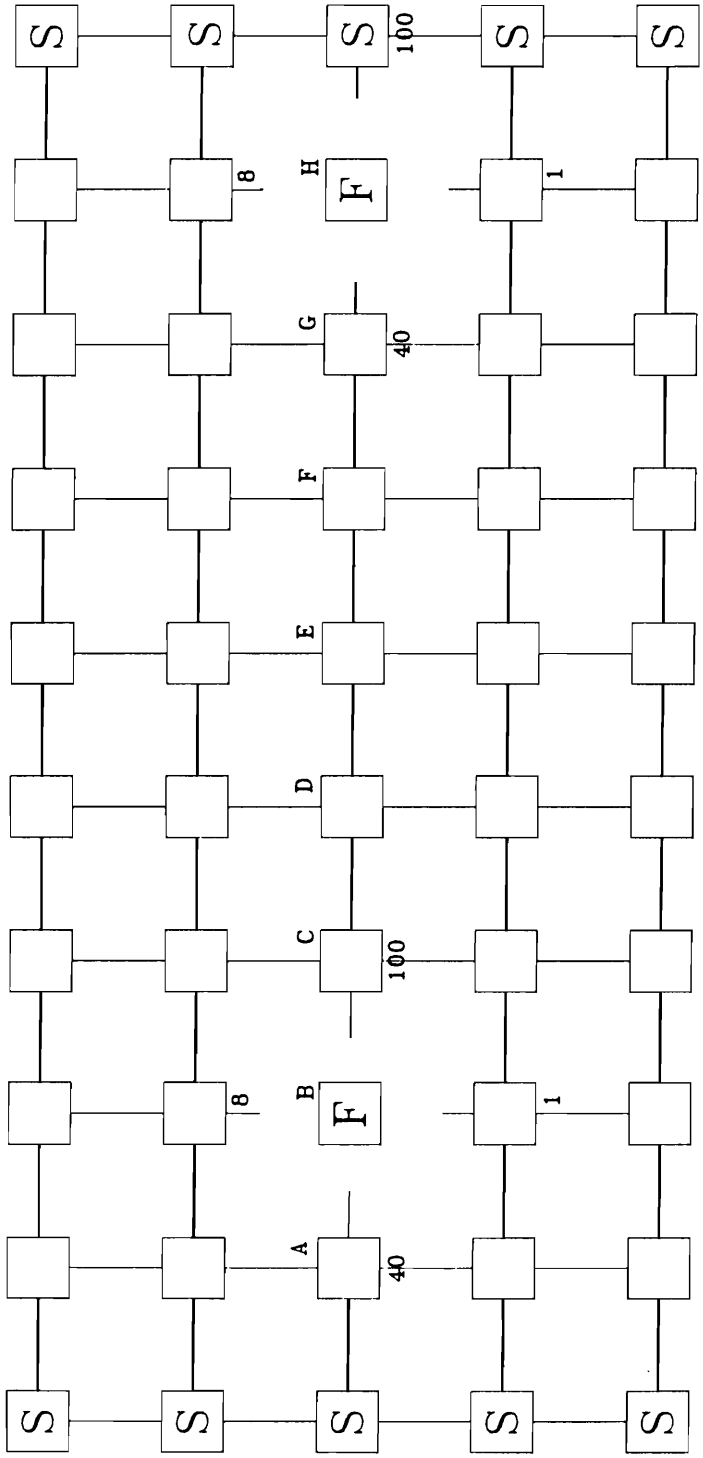


Figure 10. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time $t = 0$.

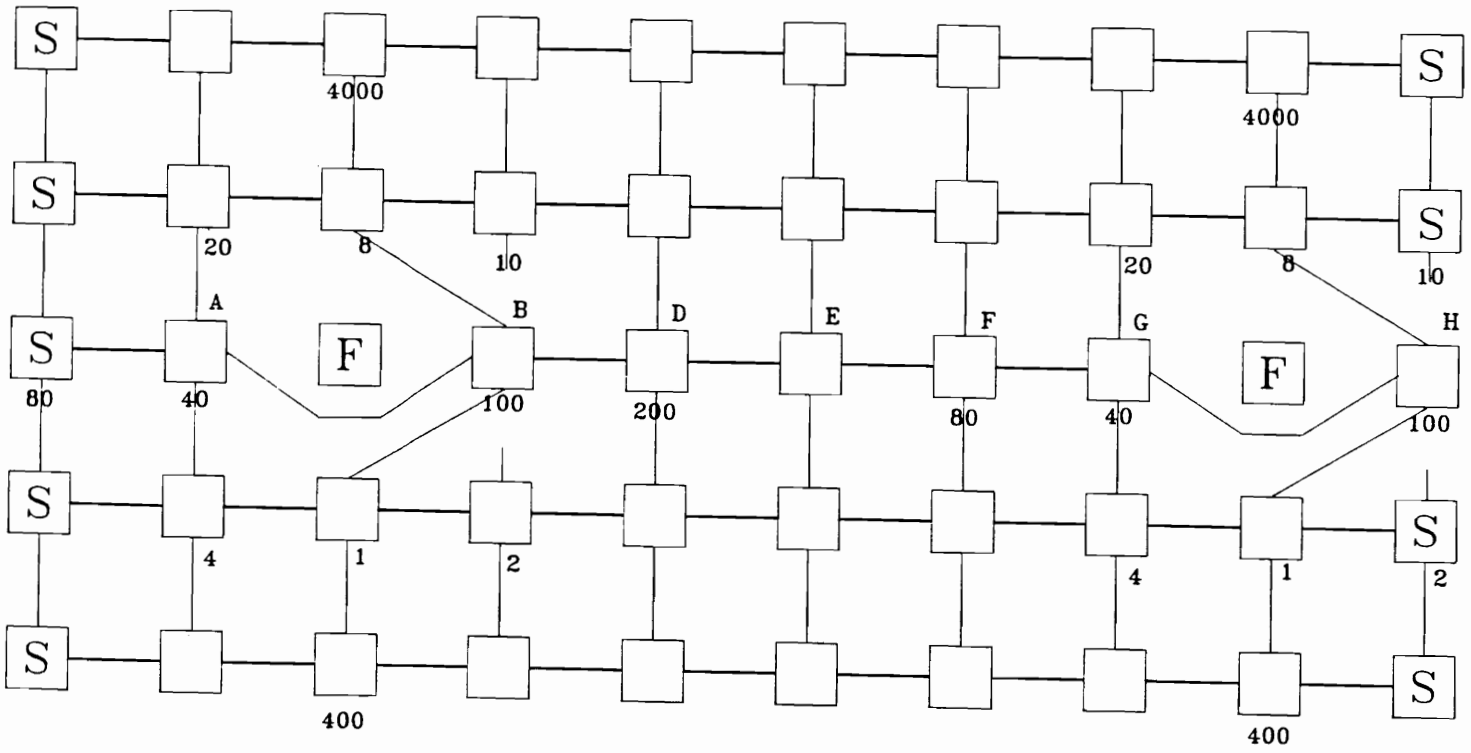


Figure 11. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = 1.

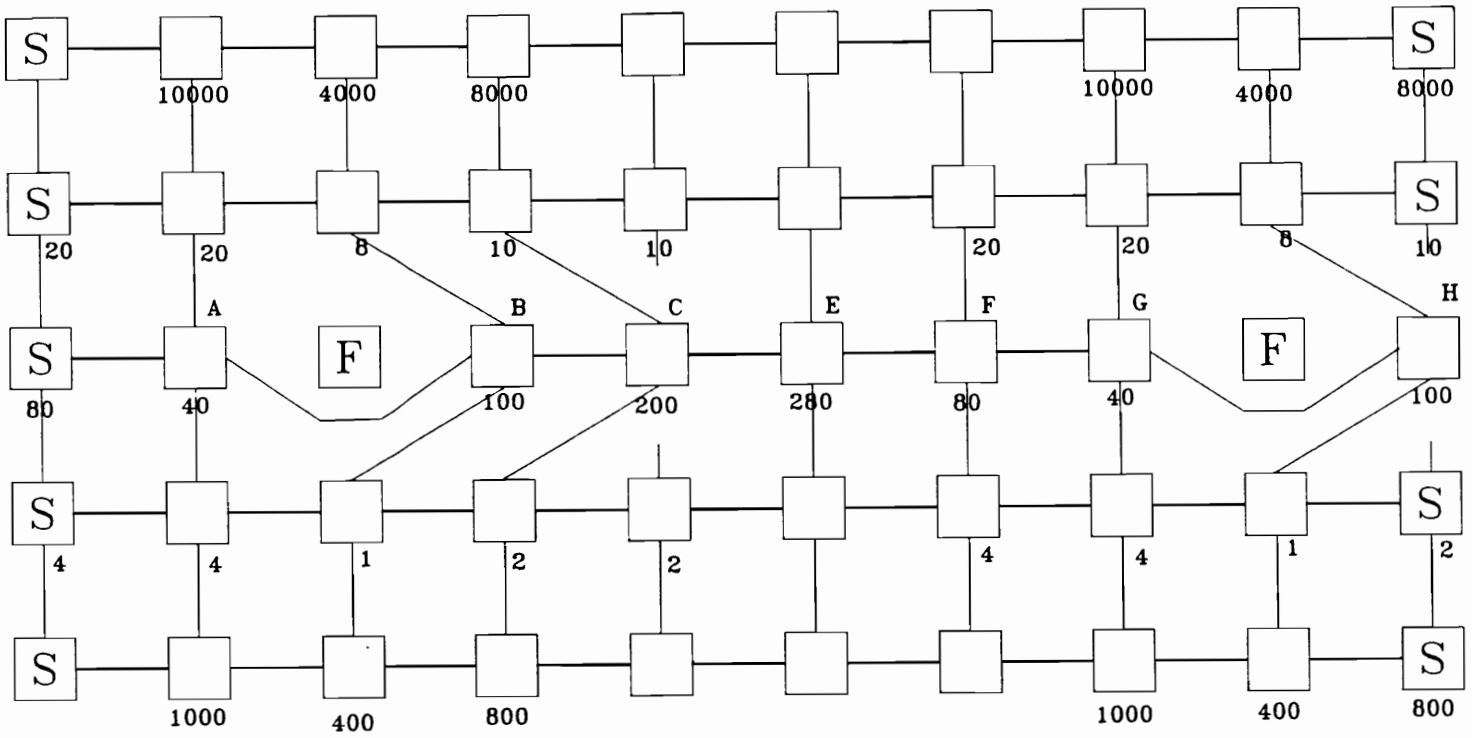


Figure 12. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = 2.

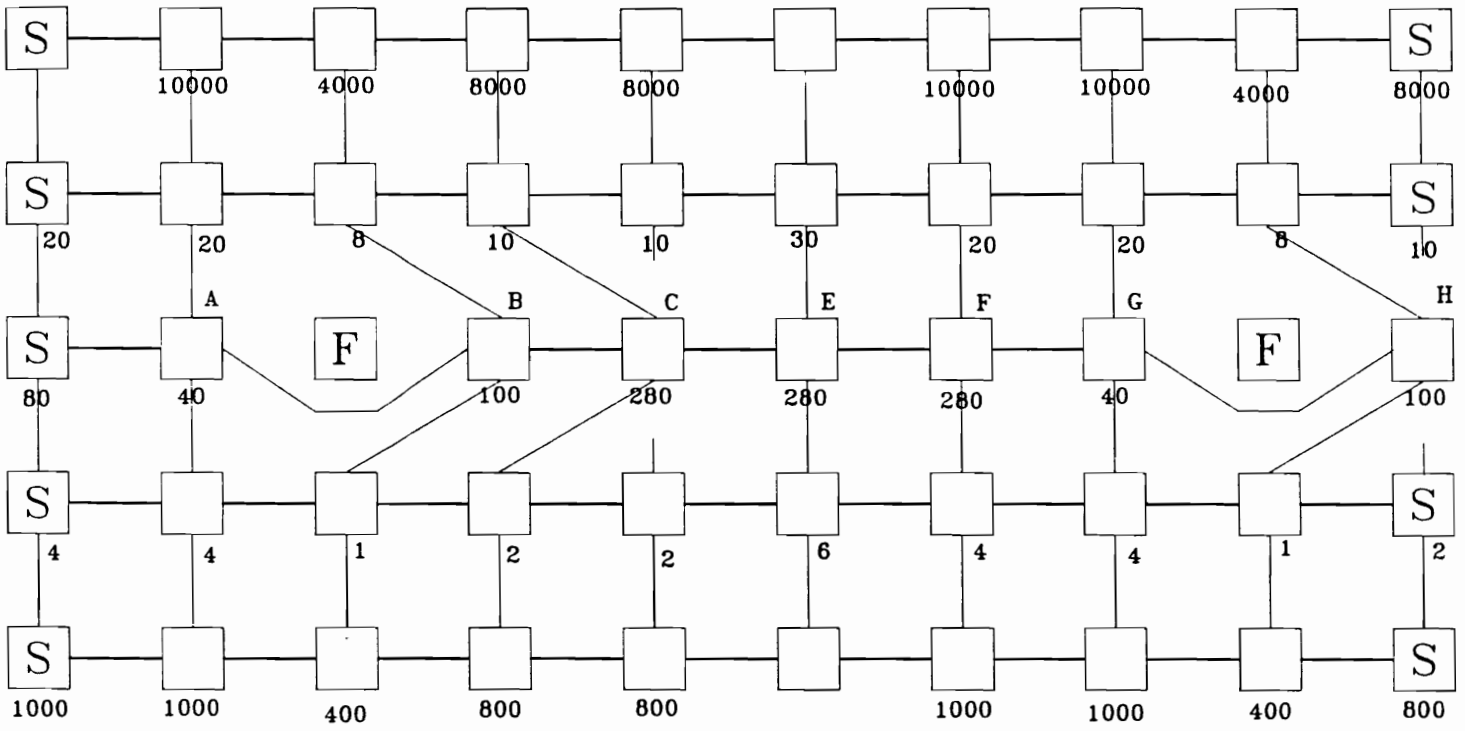


Figure 13. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = n.

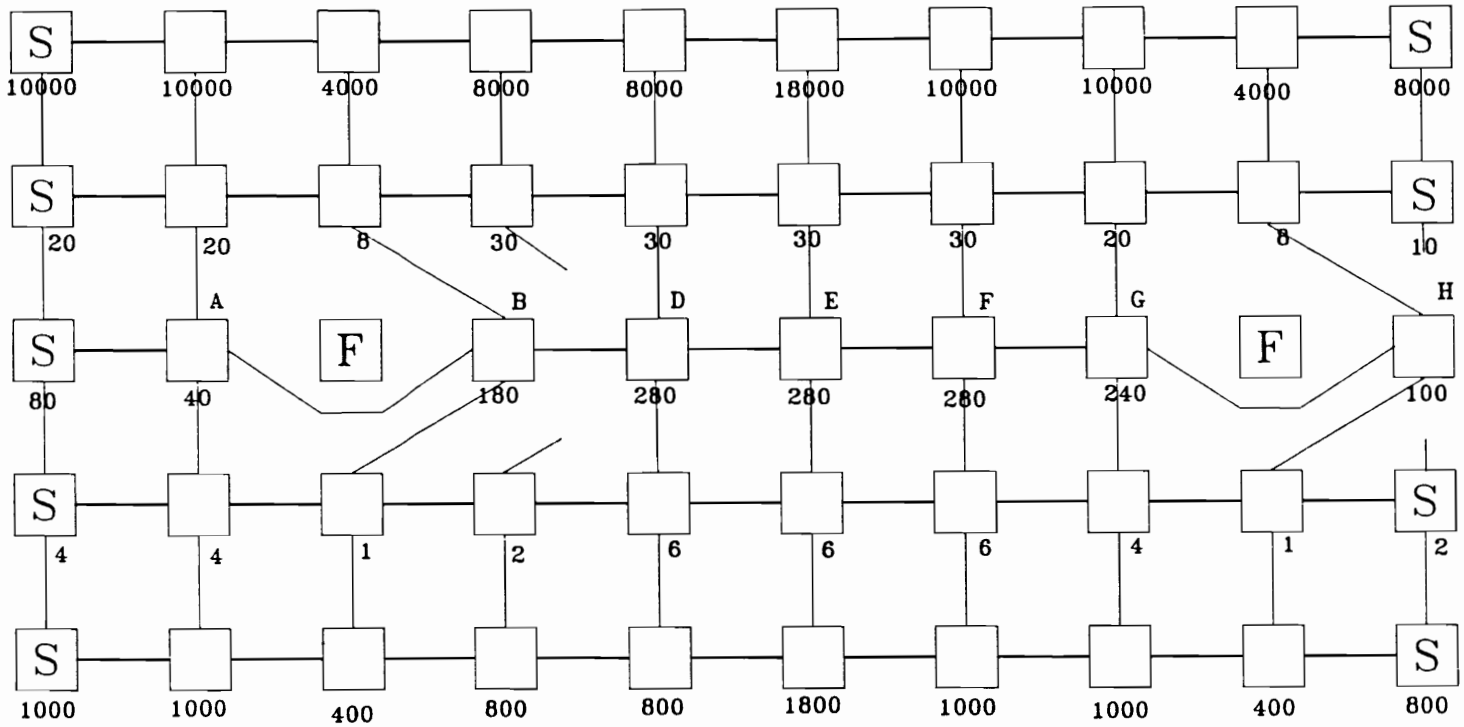


Figure 14. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t=n+1.

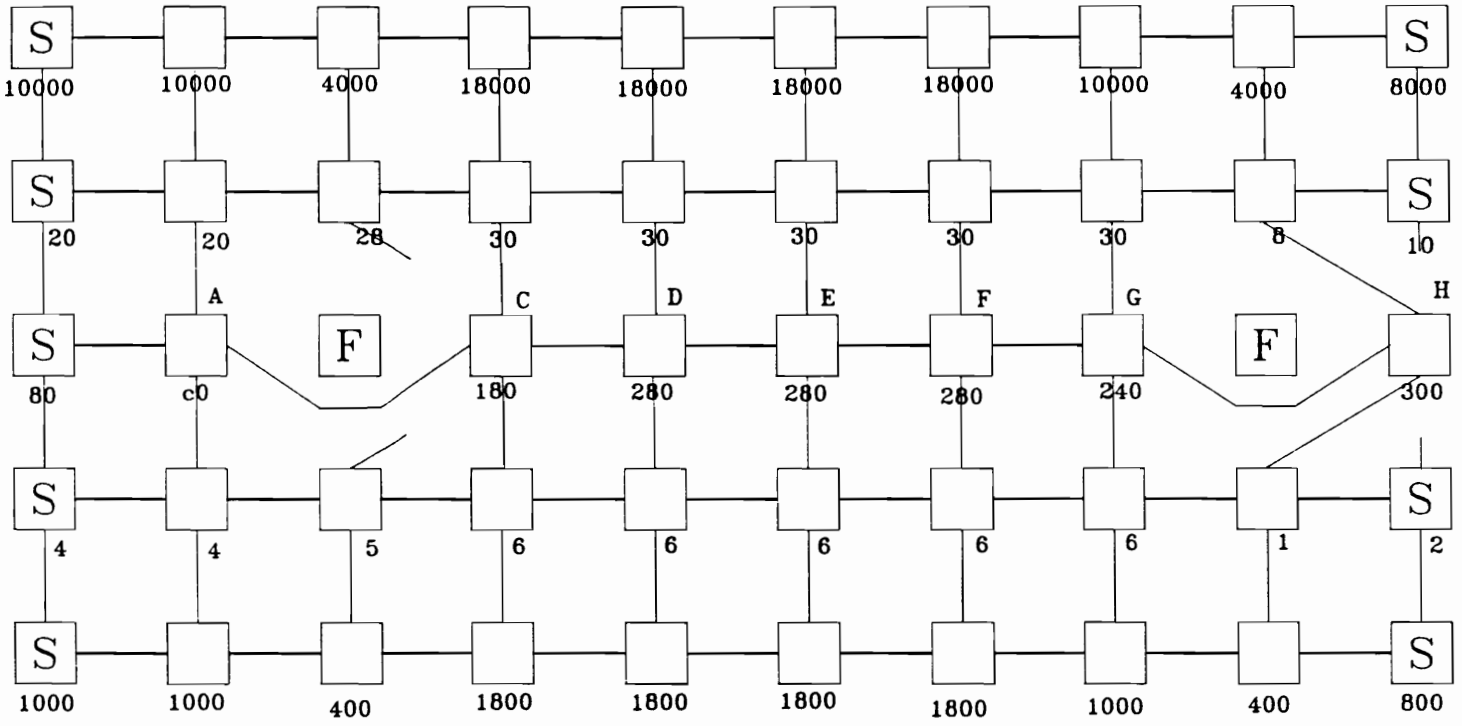


Figure 15. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time $t = n + 2$.

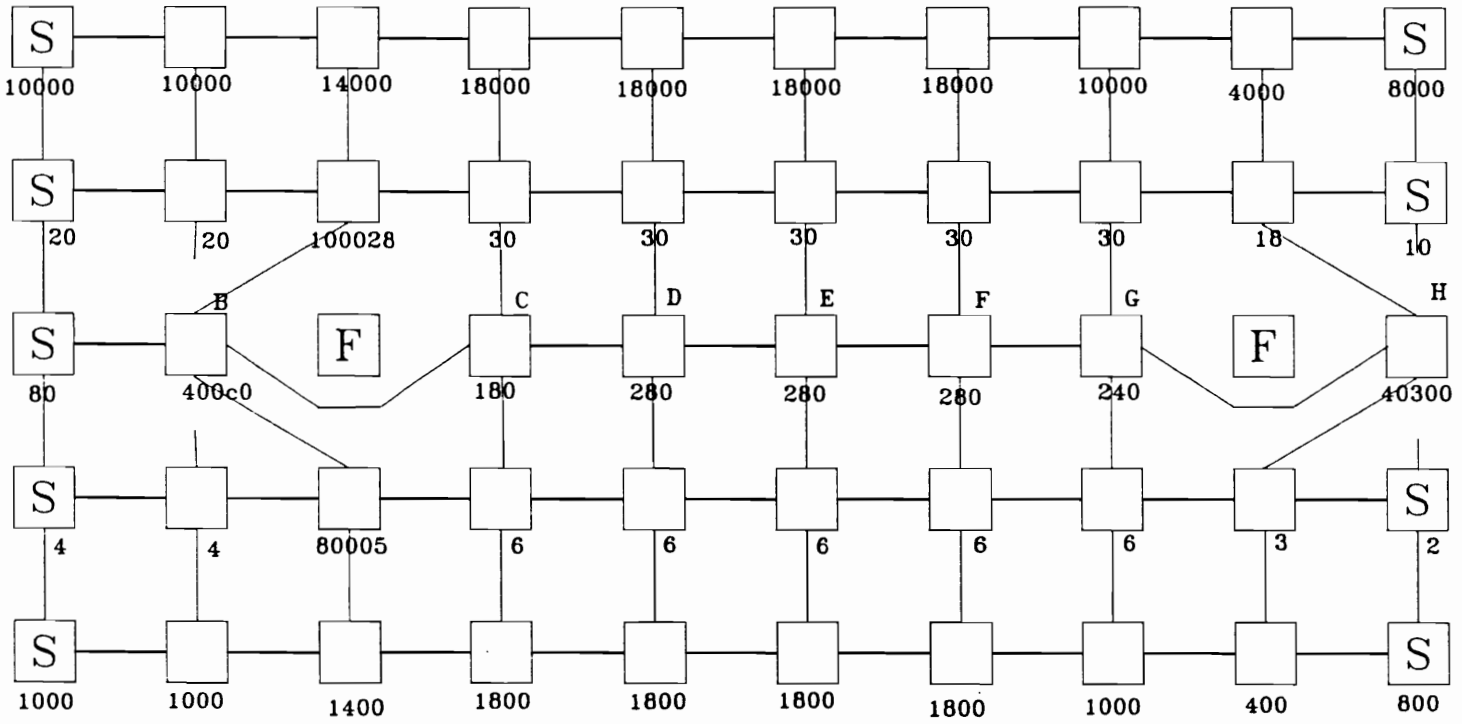


Figure 16. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = m.

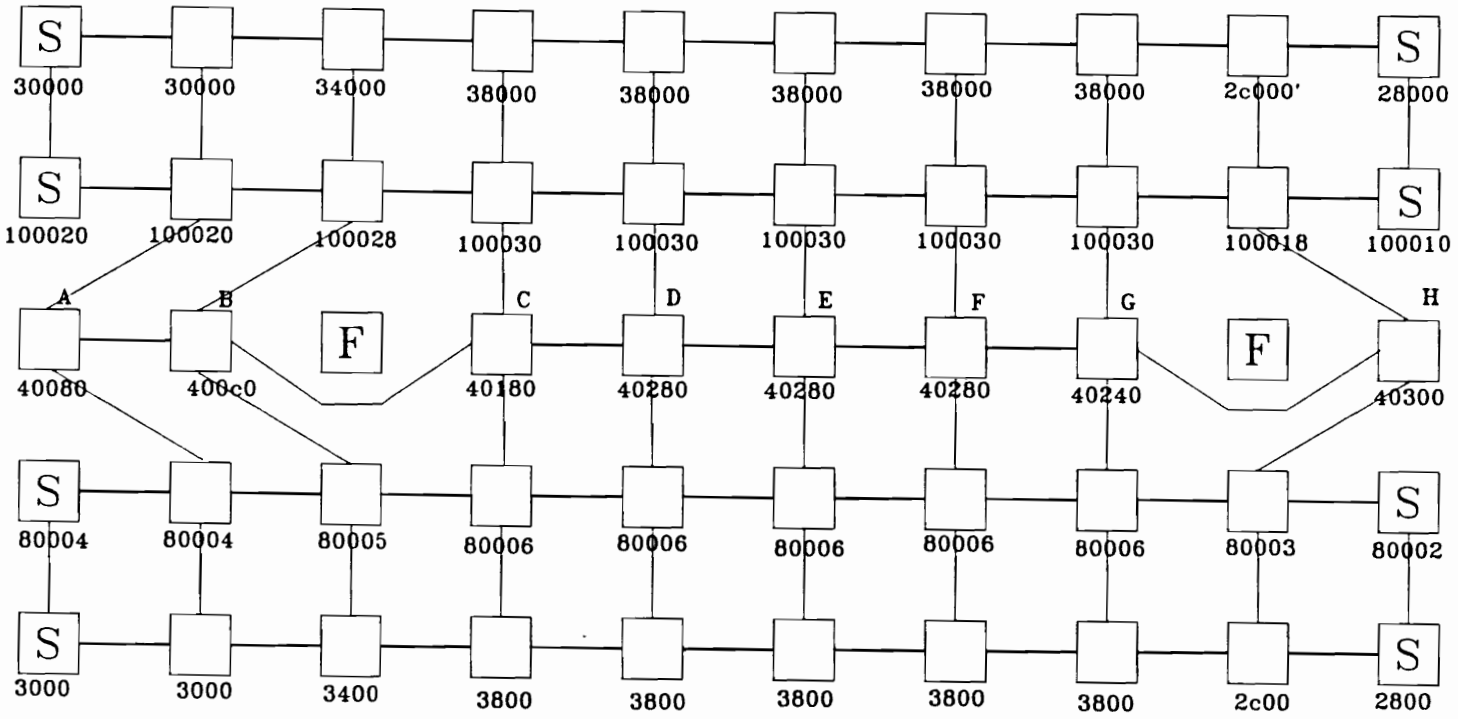


Figure 17. Double Fault Occurrence Case 2-1: Array showing the final configuration of the array

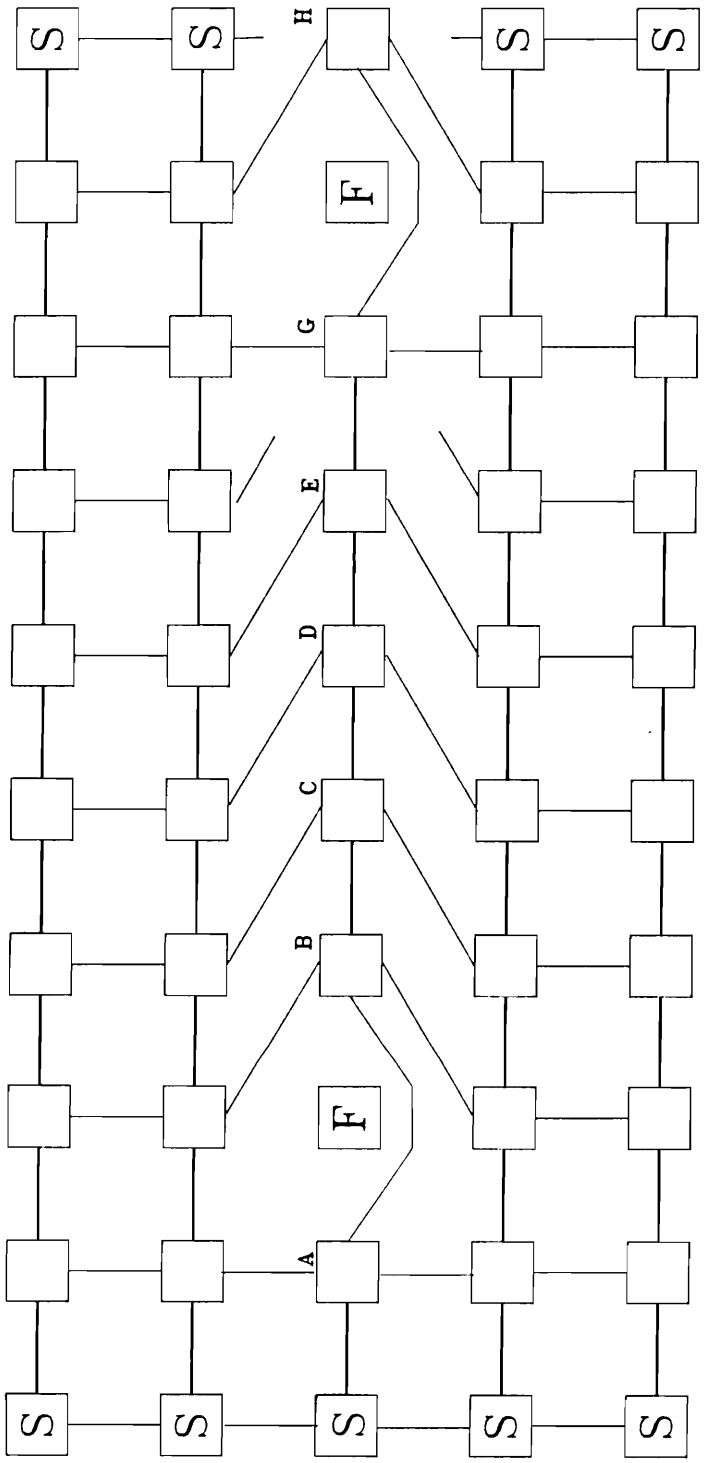


Figure 18. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = 1, with cell[i,j] previously faulty.

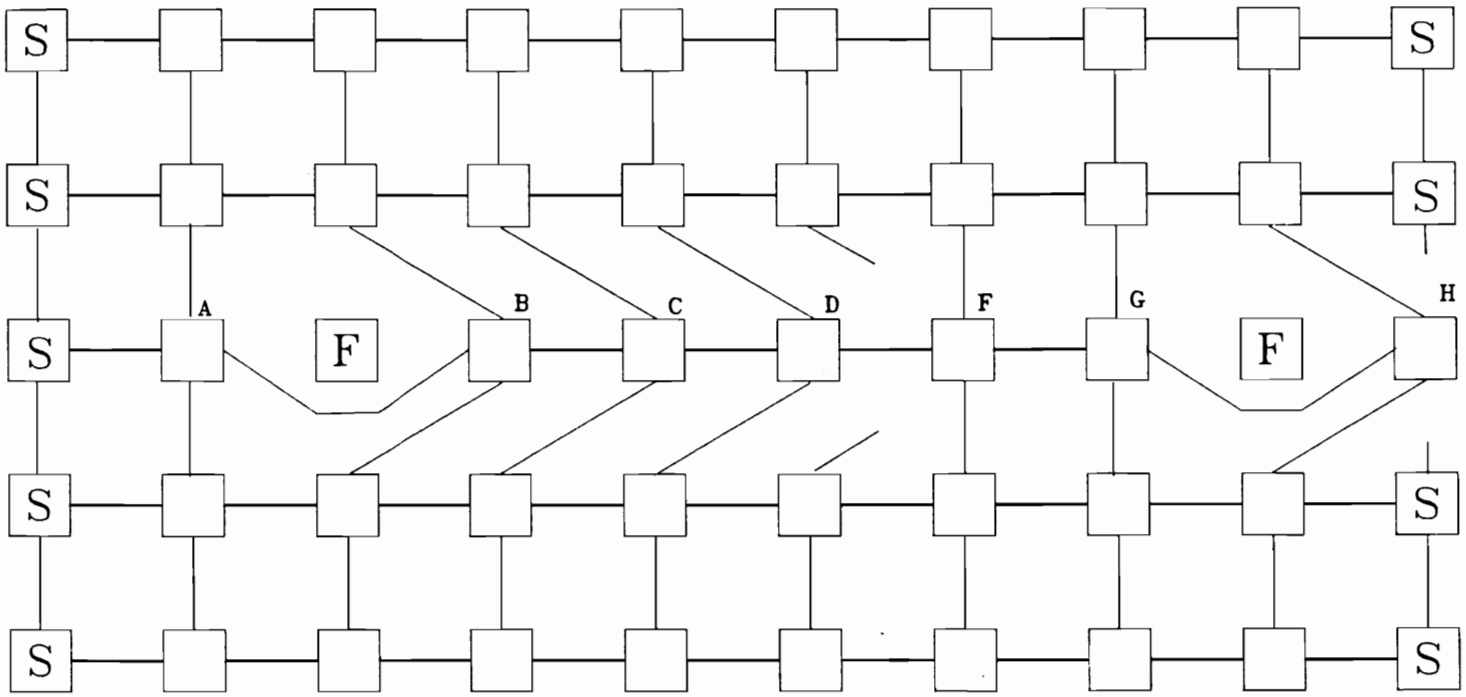


Figure 19. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = 2, with cell[i,j] previously faulty.

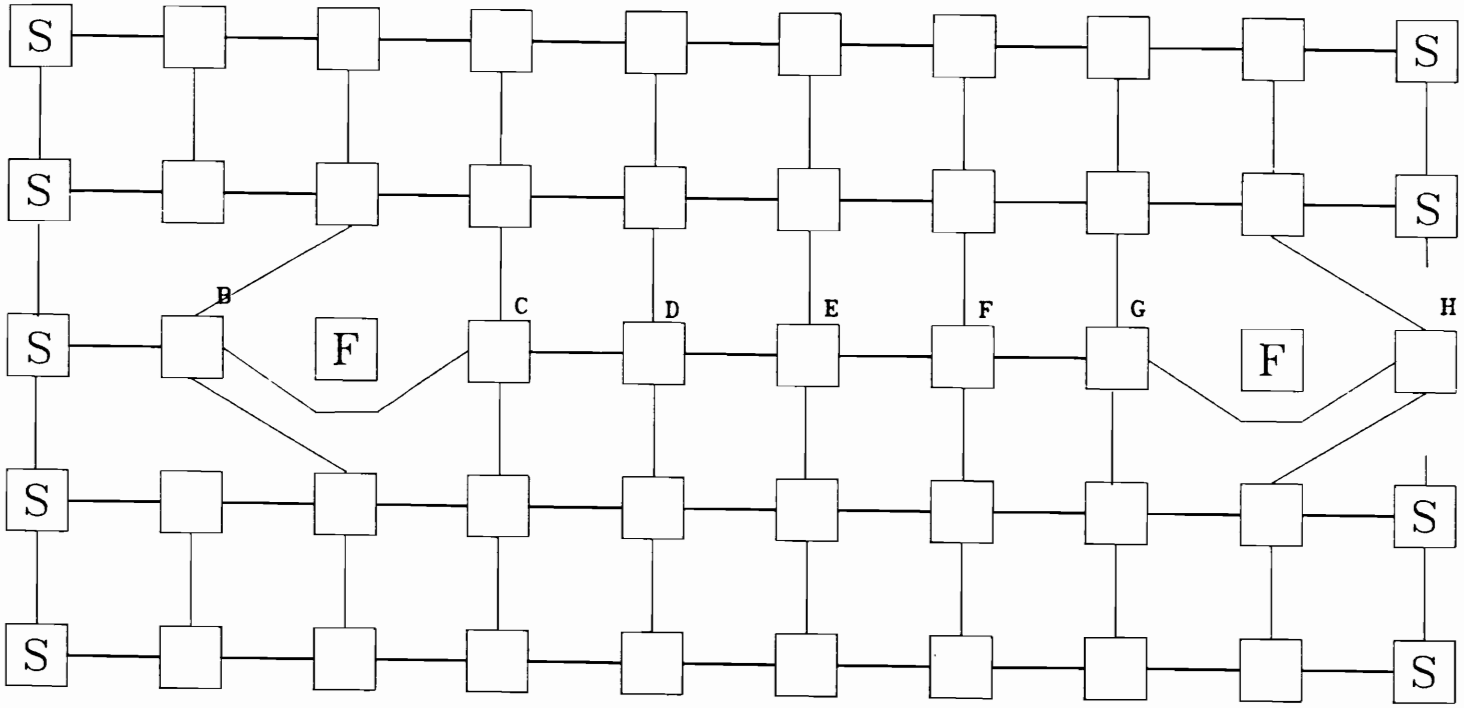


Figure 20. Double Fault Occurrence Case 2-1: Array showing configuration of the array at time t = 1, with cell[i,k] previously faulty.

Figure 21. Double Fault Occurrence Case 2-2: Array showing configuration around faults which are independent.

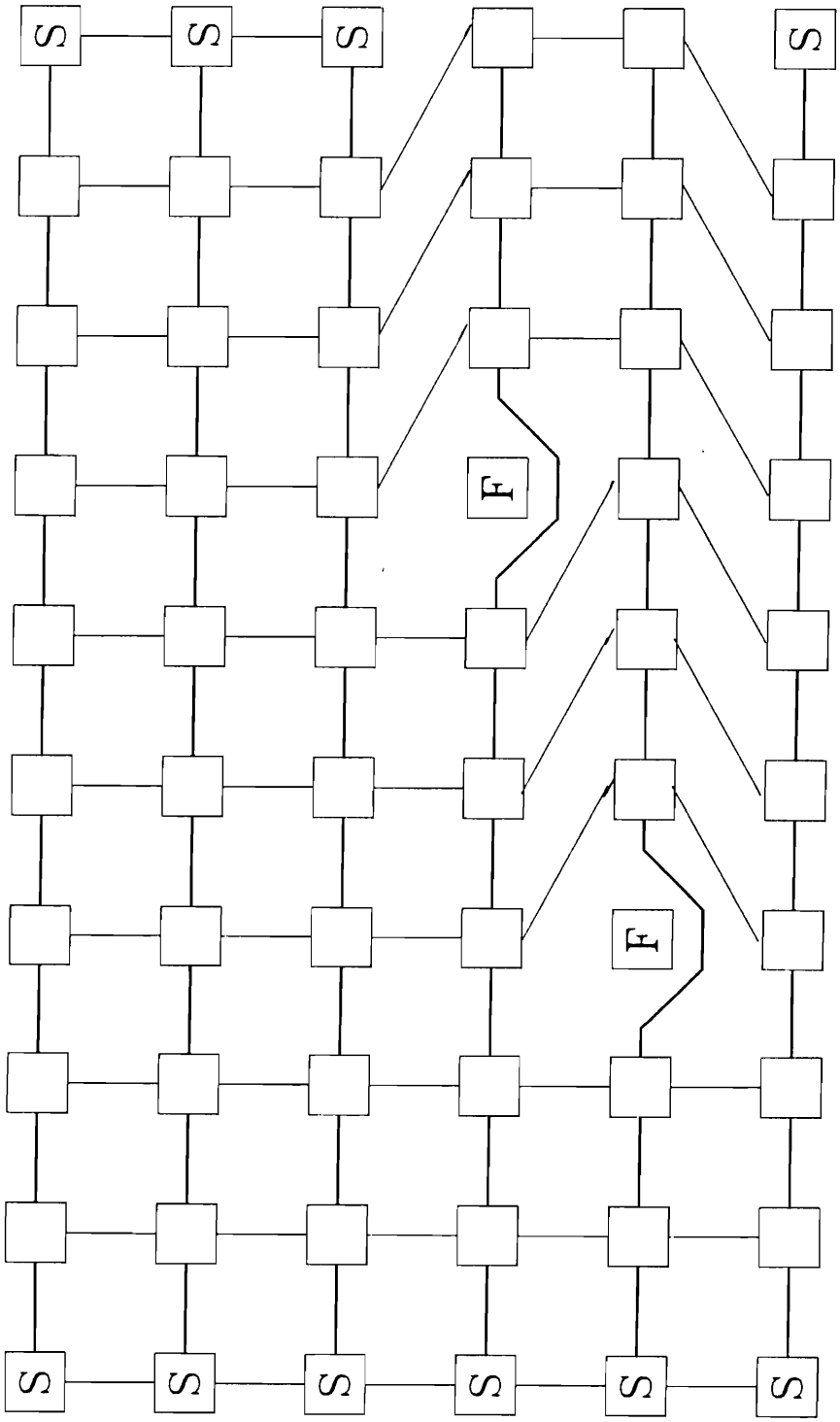


Figure 22. Double Fault Occurrence Case 2-3a: Array showing configuration around two faults in adjacent rows.

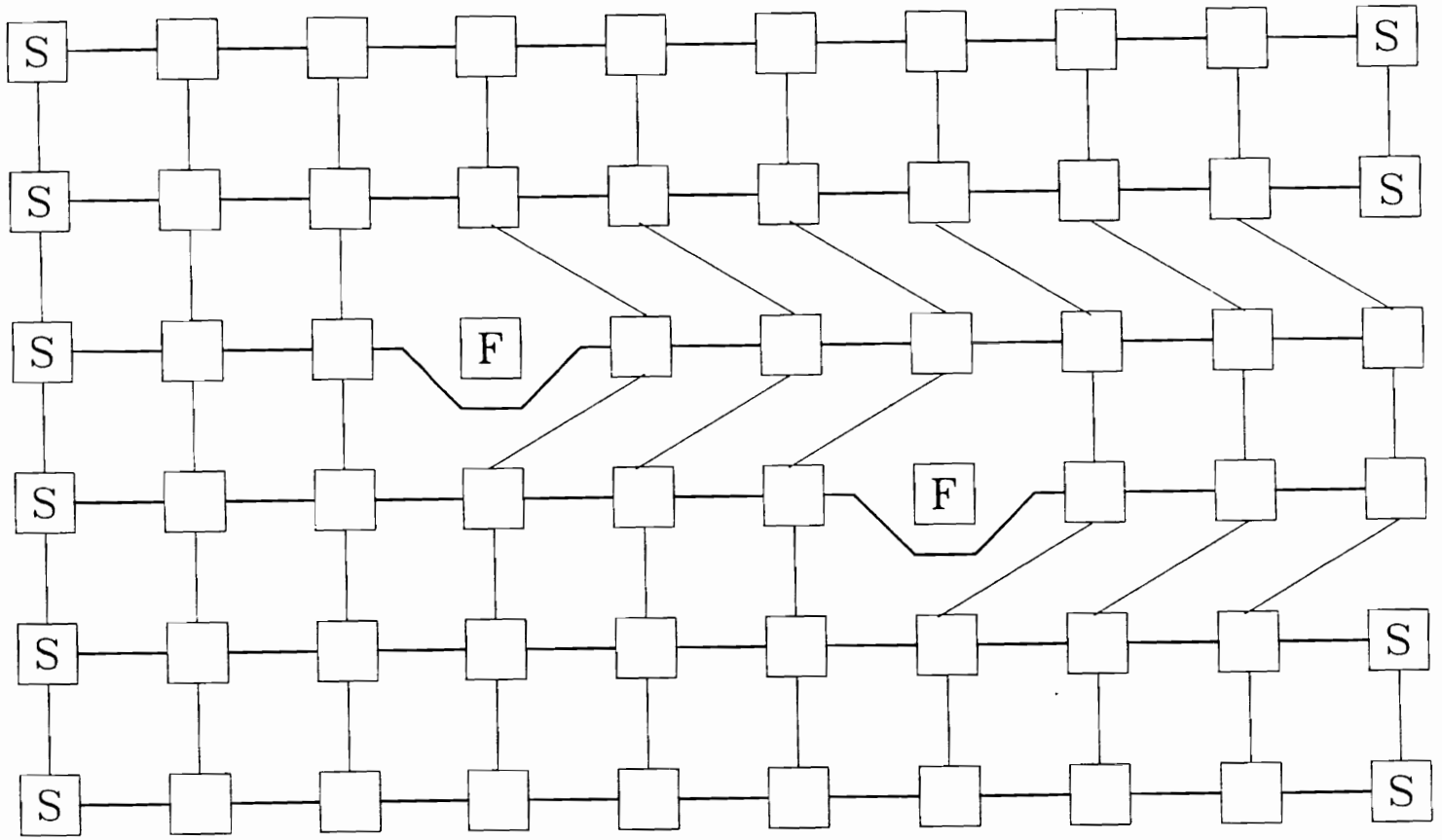


Figure 23. Double Fault Occurrence Case 2-3b: Array showing configuration around two faults in adjacent rows.

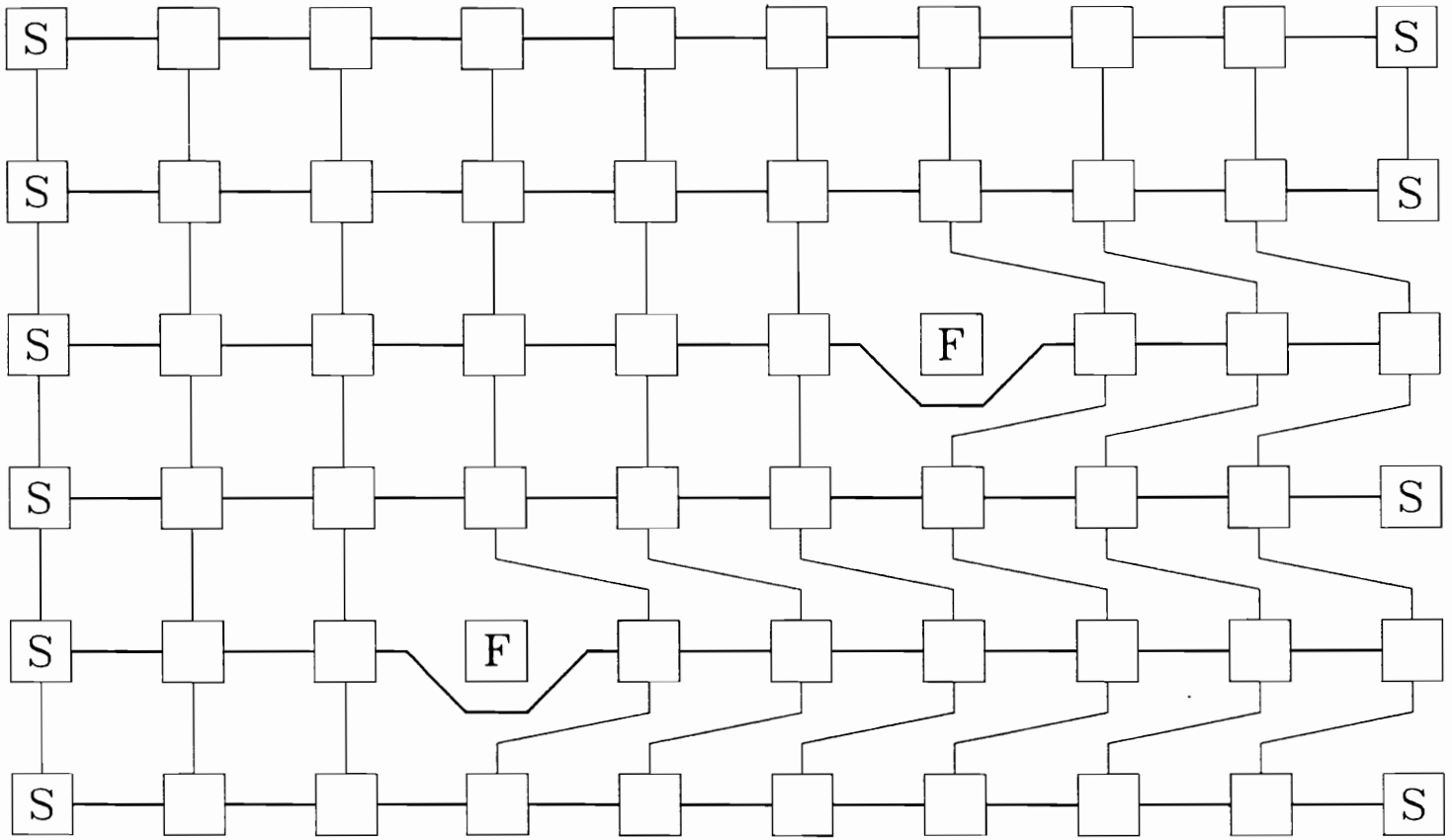


Figure 24. Double Fault Occurrence Case 2-4: Array showing configuration around two faults separated by one row.

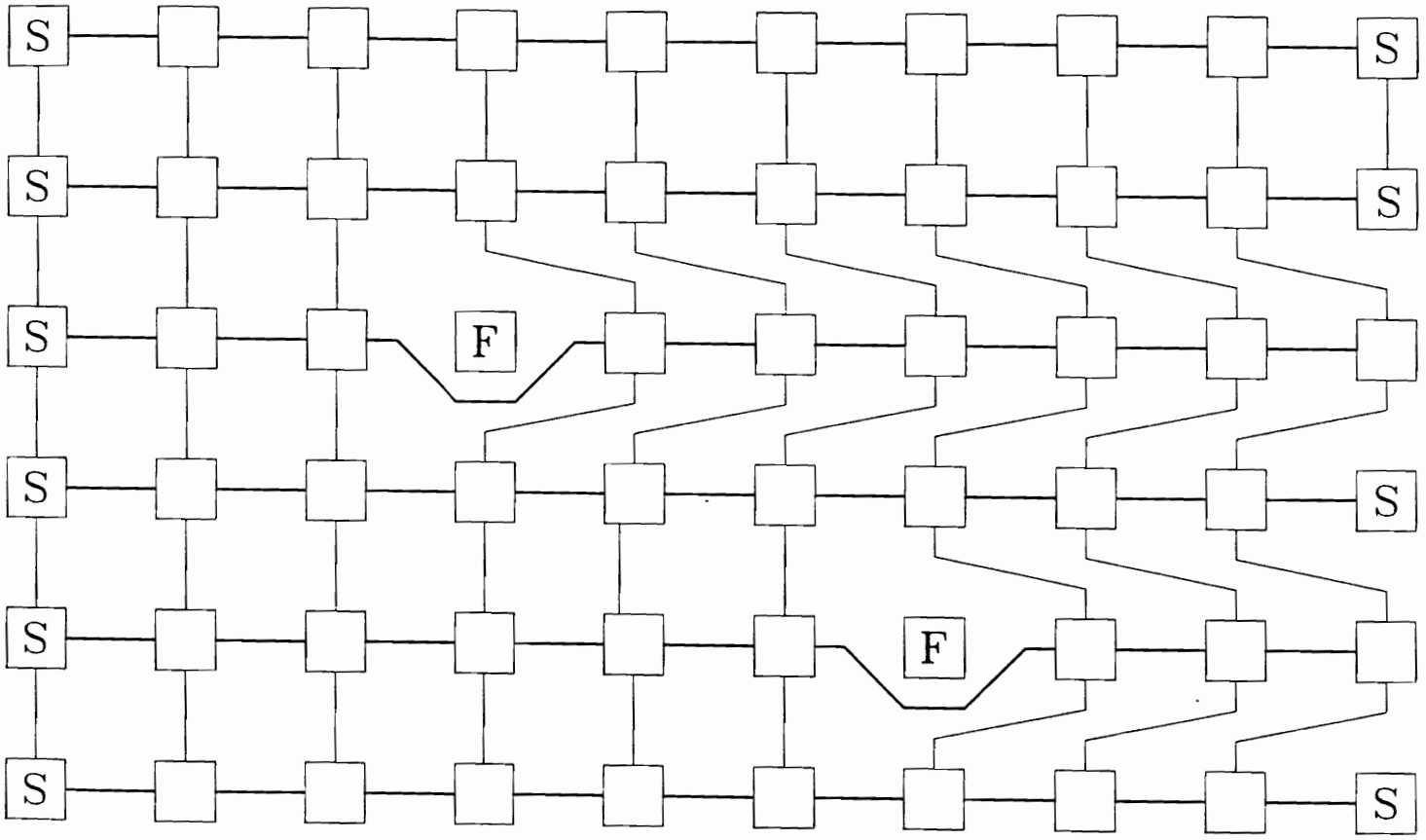


Figure 25. Double Fault Occurrence Case 2-4: Array showing configuration around two faults separated by one row.

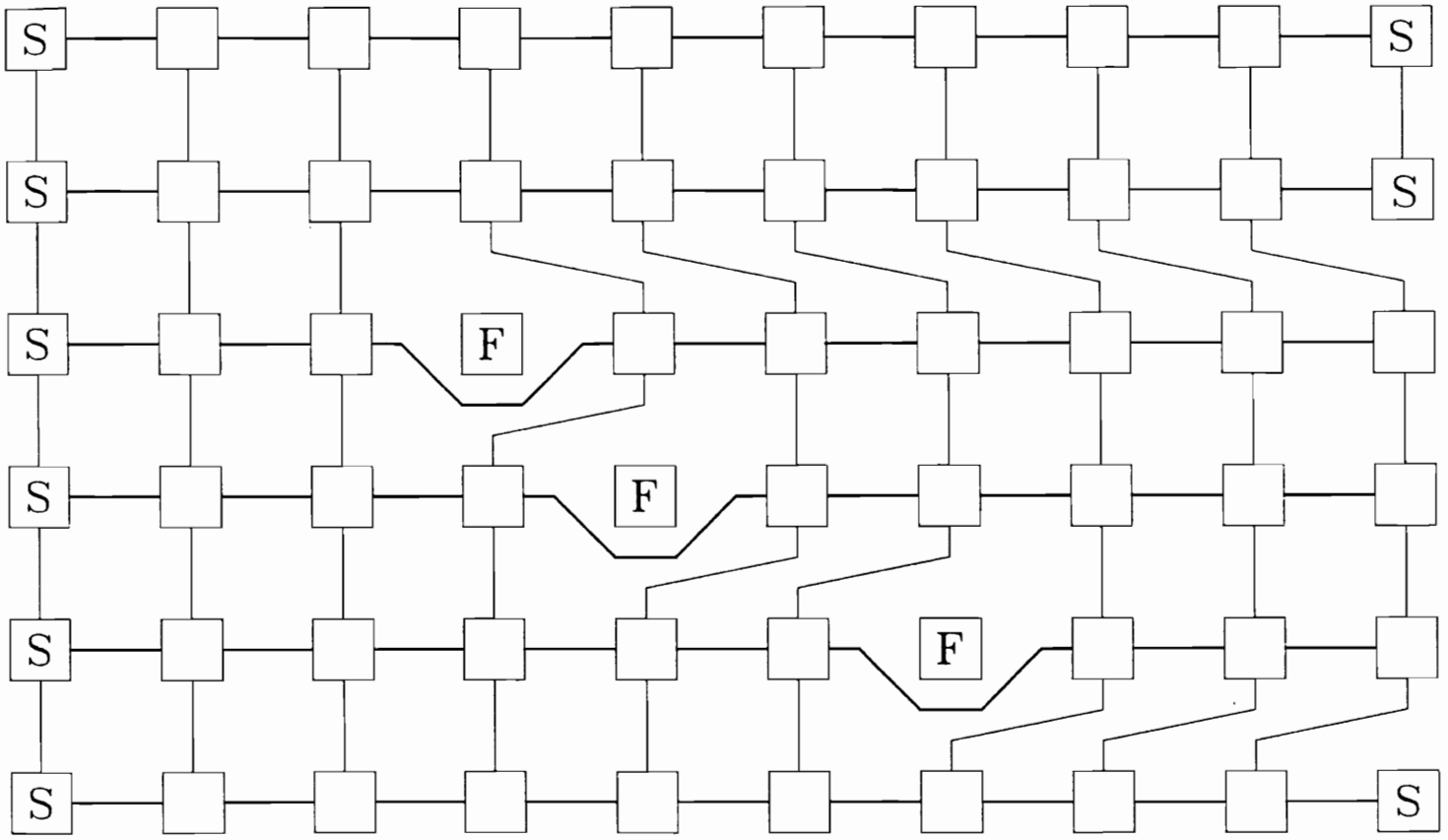


Figure 26. Three Fault Occurrence Case 3-1.a: Array showing configuration three faults in adjacent rows.

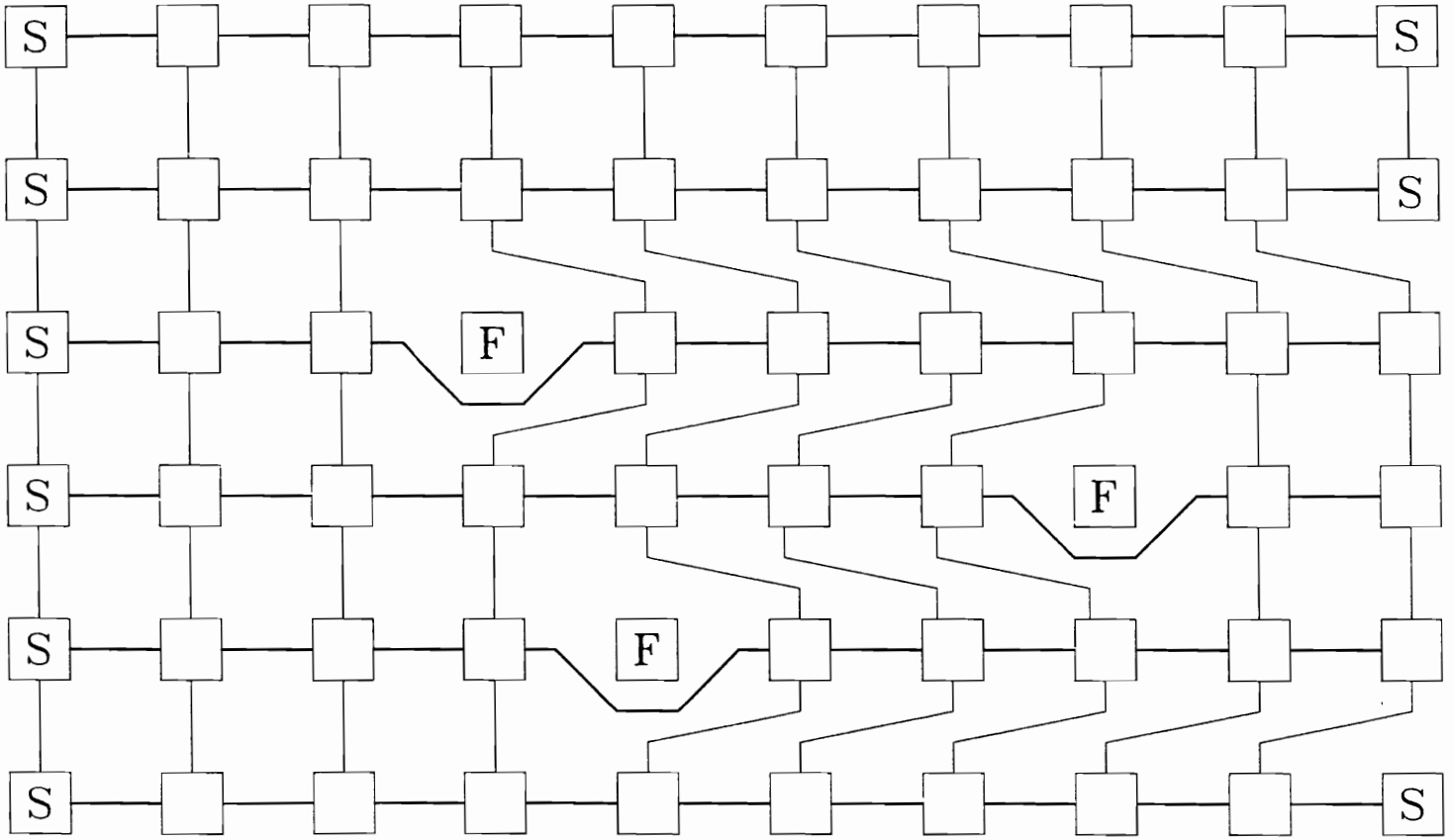


Figure 27. Three Faults Case 3-1.b: Array showing three faults in adjacent rows.

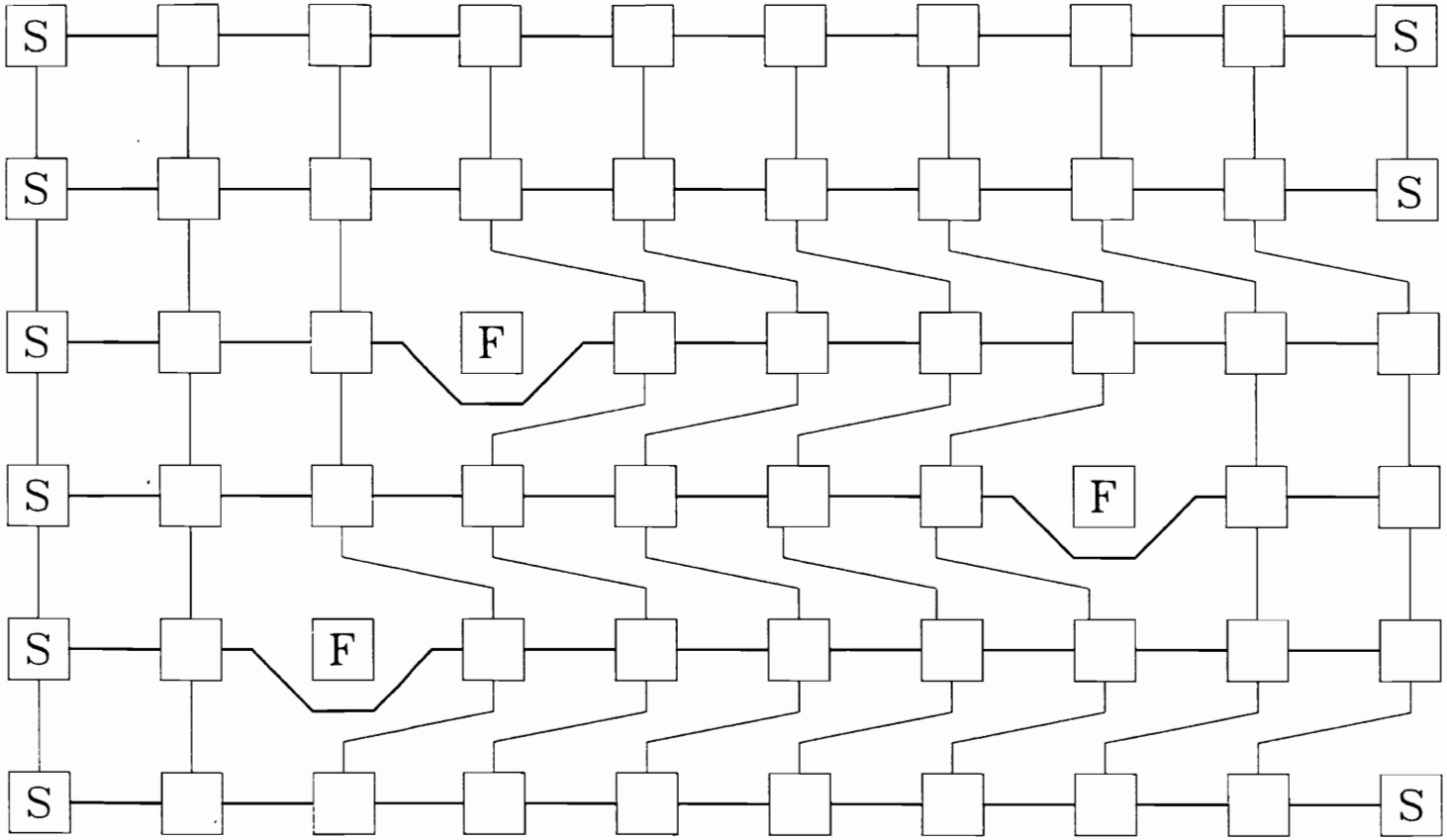


Figure 28. Three Faults Case 3-1.c: Array showing three faults in adjacent rows.

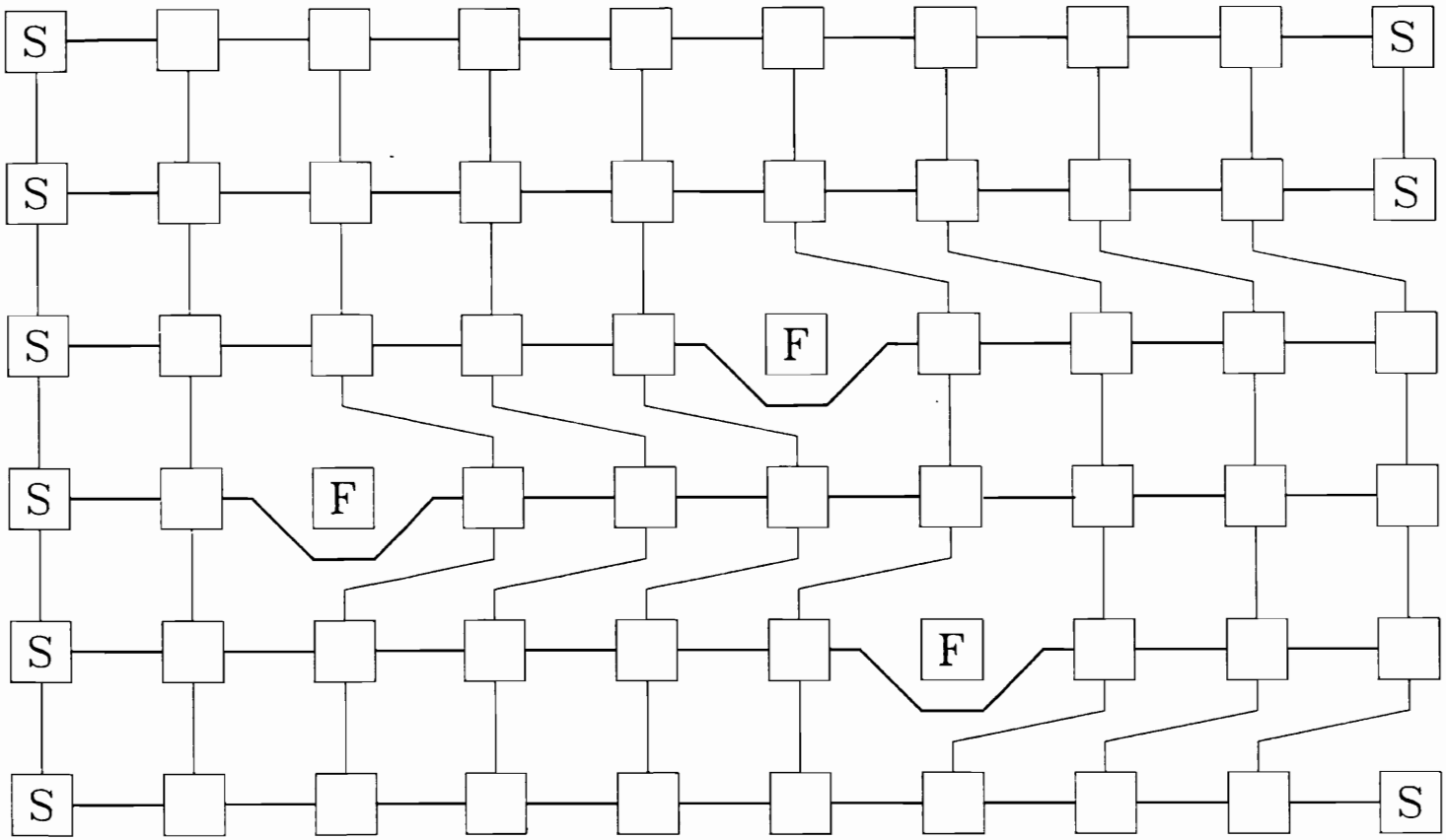


Figure 29. Three Faults Case 3-1.d: Array showing three faults in adjacent rows.

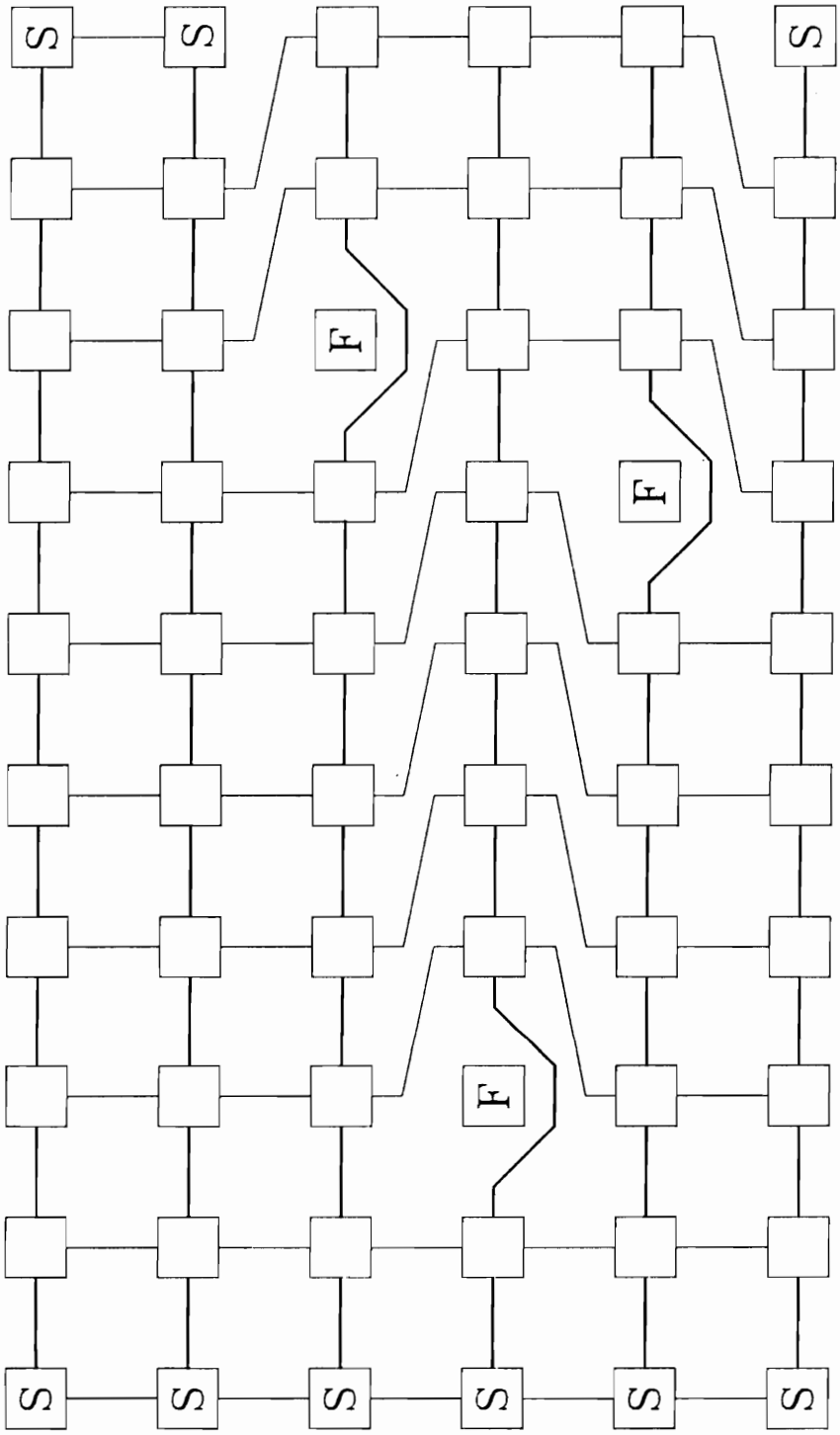


Figure 30. Three Faults Case 3-1.c: Array showing three faults in adjacent rows.

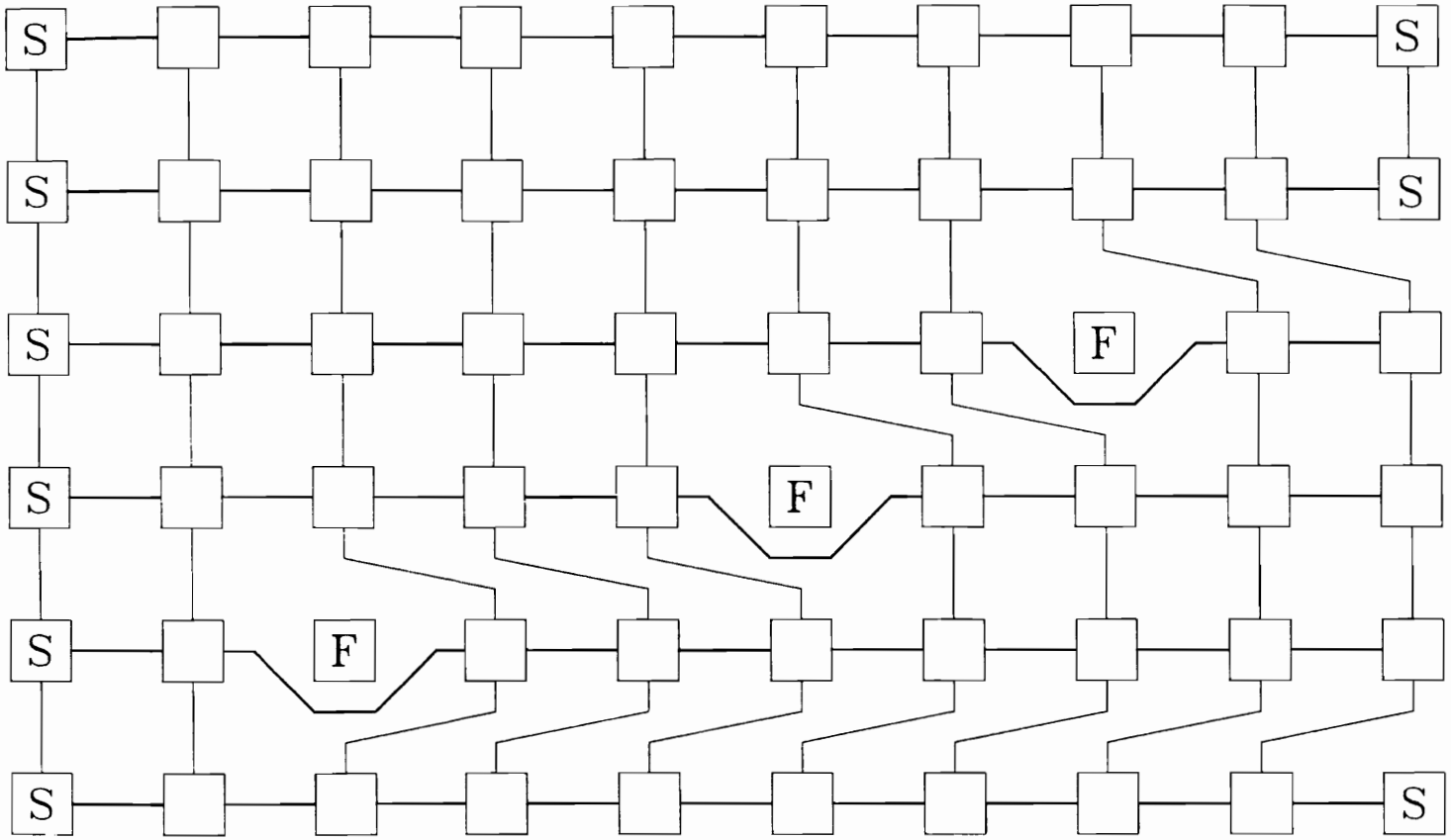


Figure 31. Three Faults Case 3-1.f: Array showing three faults in adjacent rows.

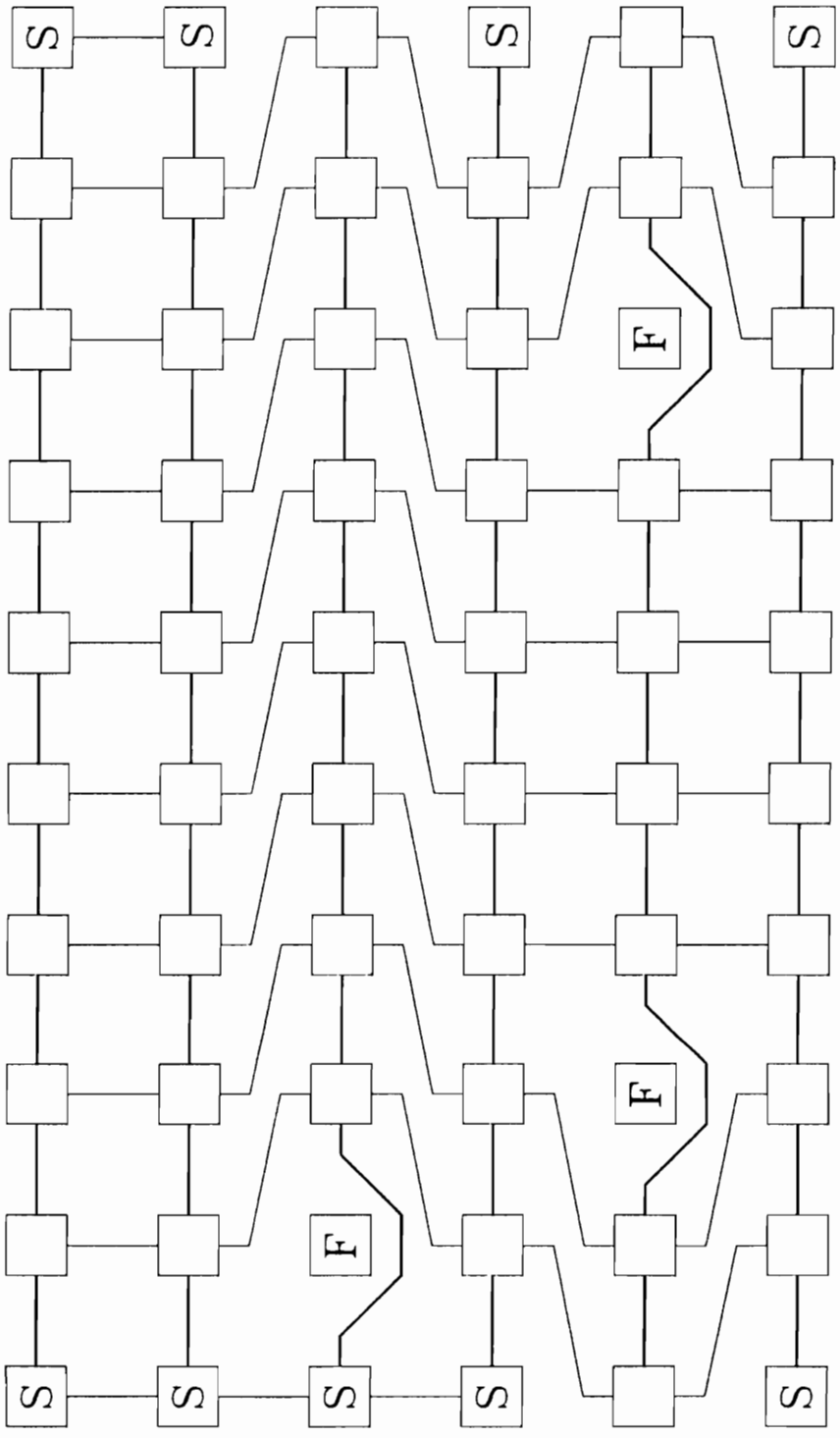


Figure 32. Three Faults Case 3-2.a: Array showing 2-1 pattern of faults in non-adjacent rows.

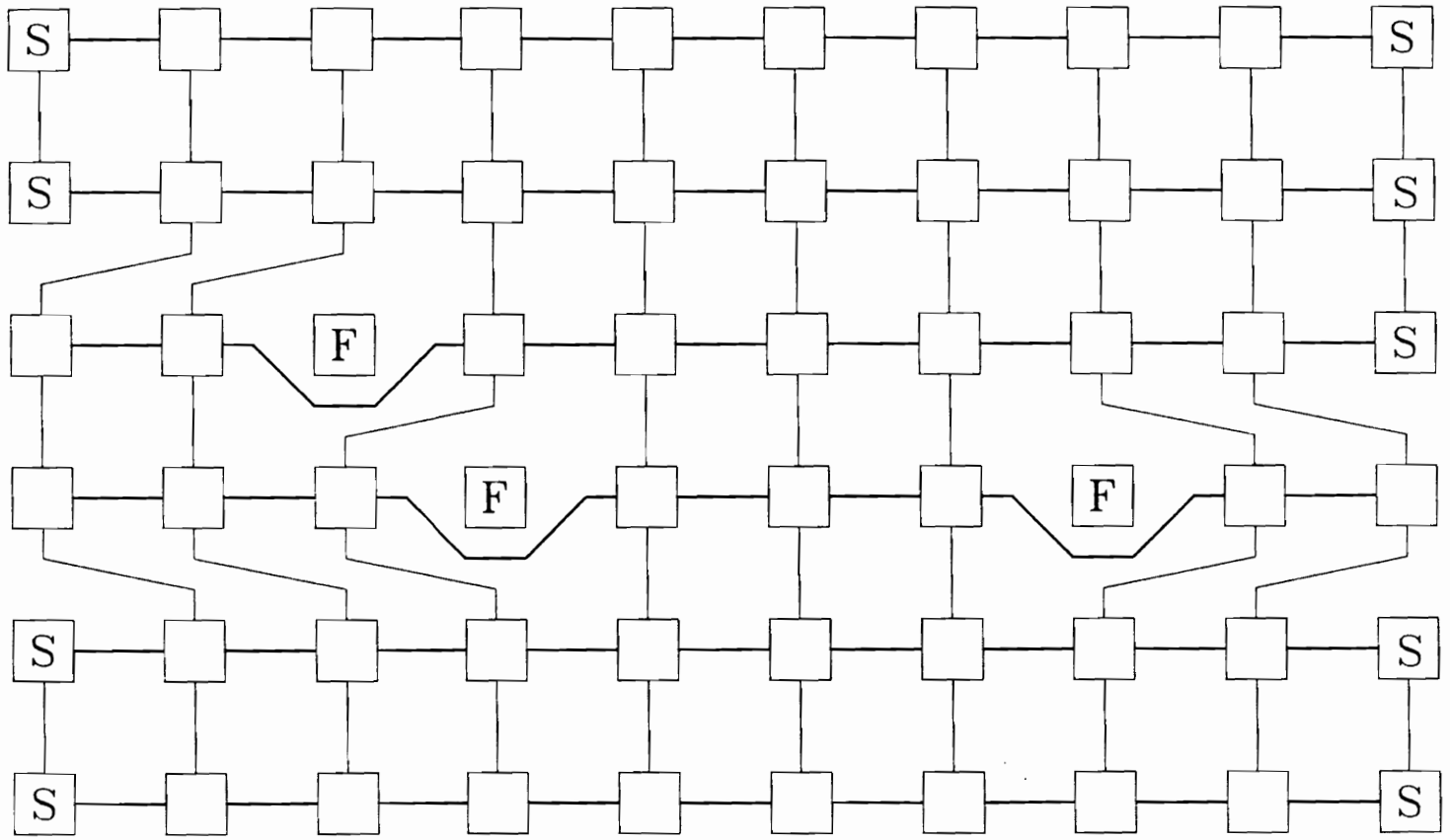


Figure 33. Three Faults Case 3-2.b: Array showing 2-1 pattern of faults in adjacent rows.

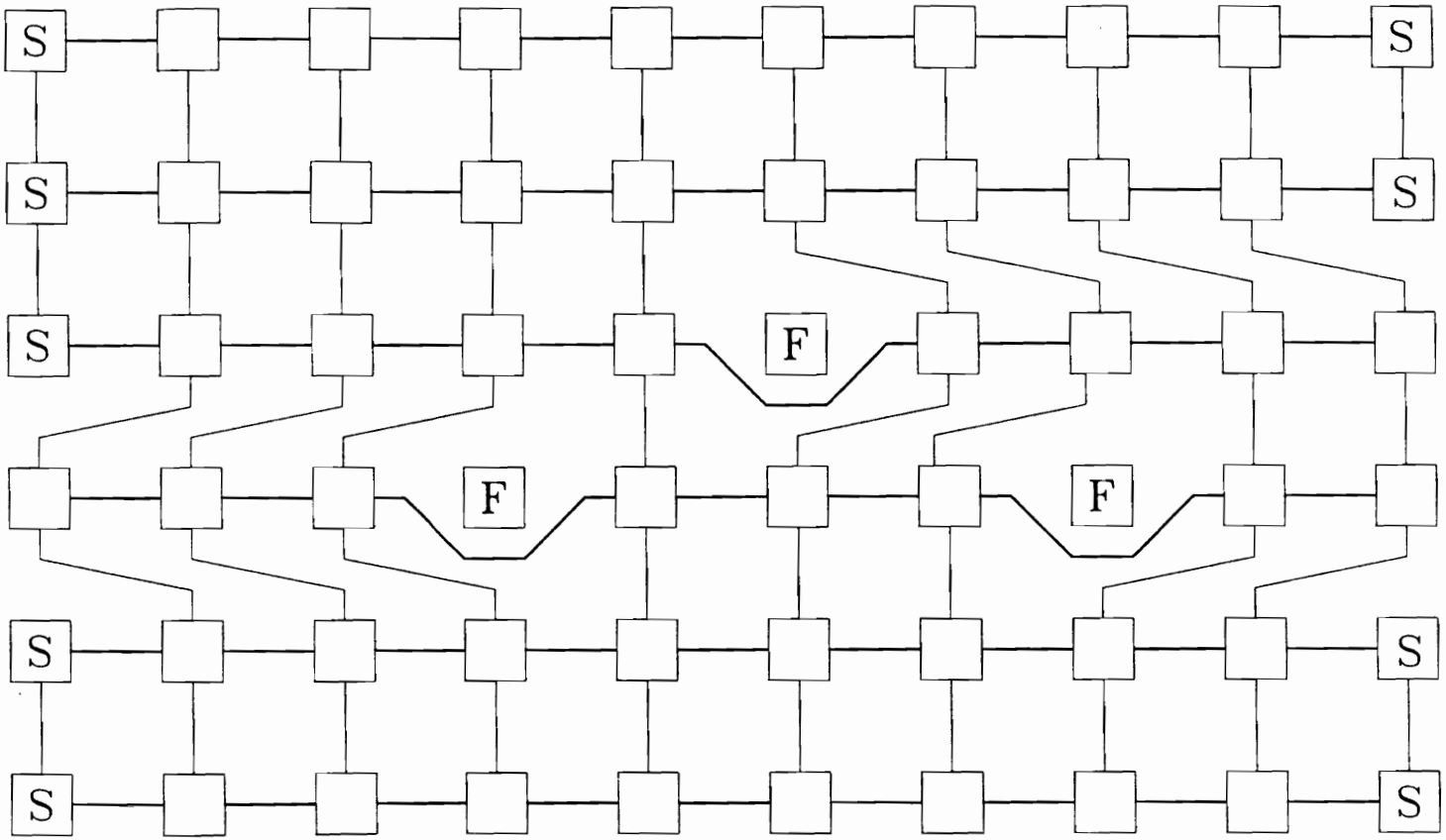


Figure 34. Three Faults Case 3-2.b: Array showing 2-1 pattern of faults in adjacent rows.

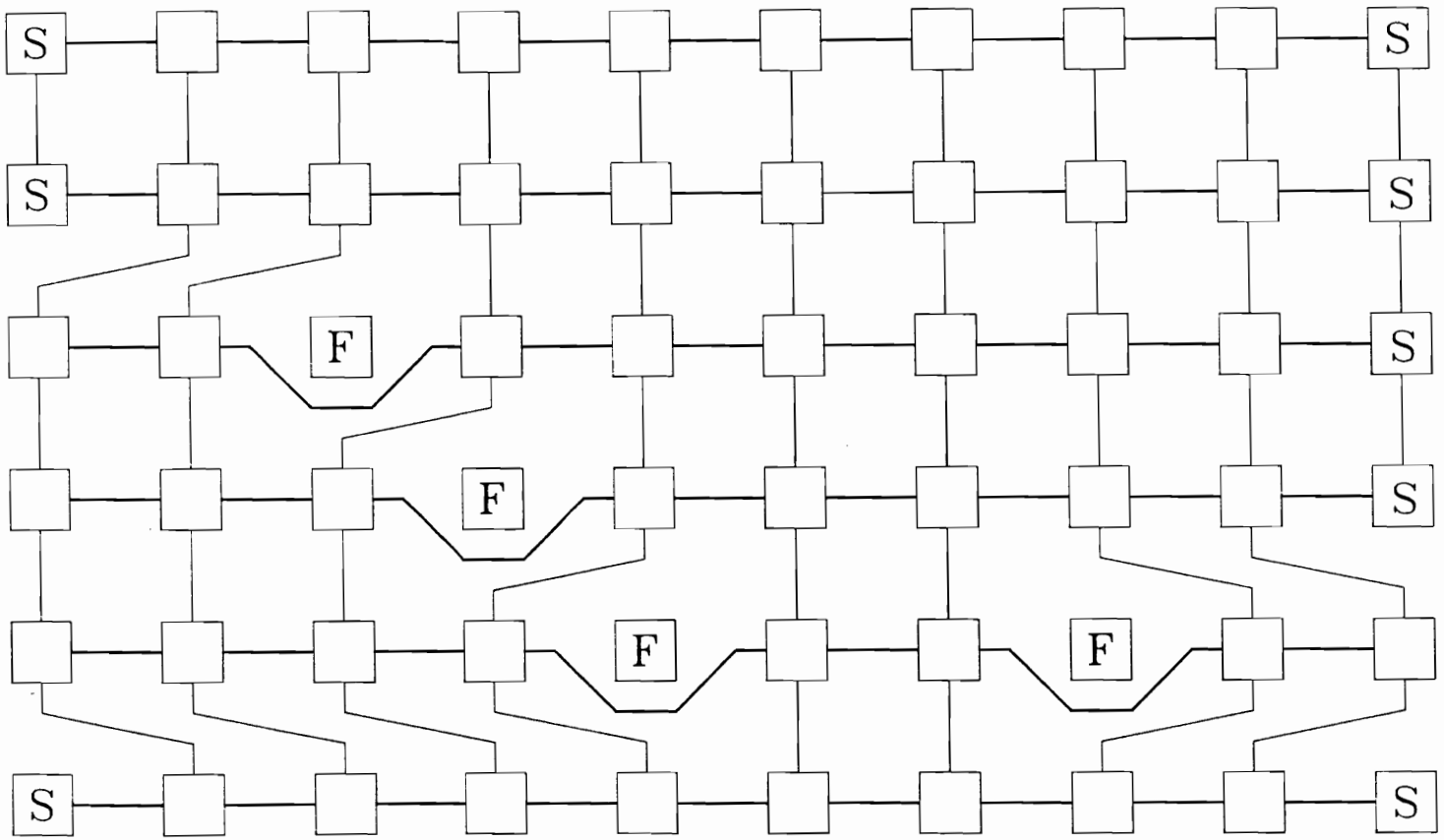


Figure 35. Four Faults Case 4-2a: Array with four faults in a 1-1-2 pattern.

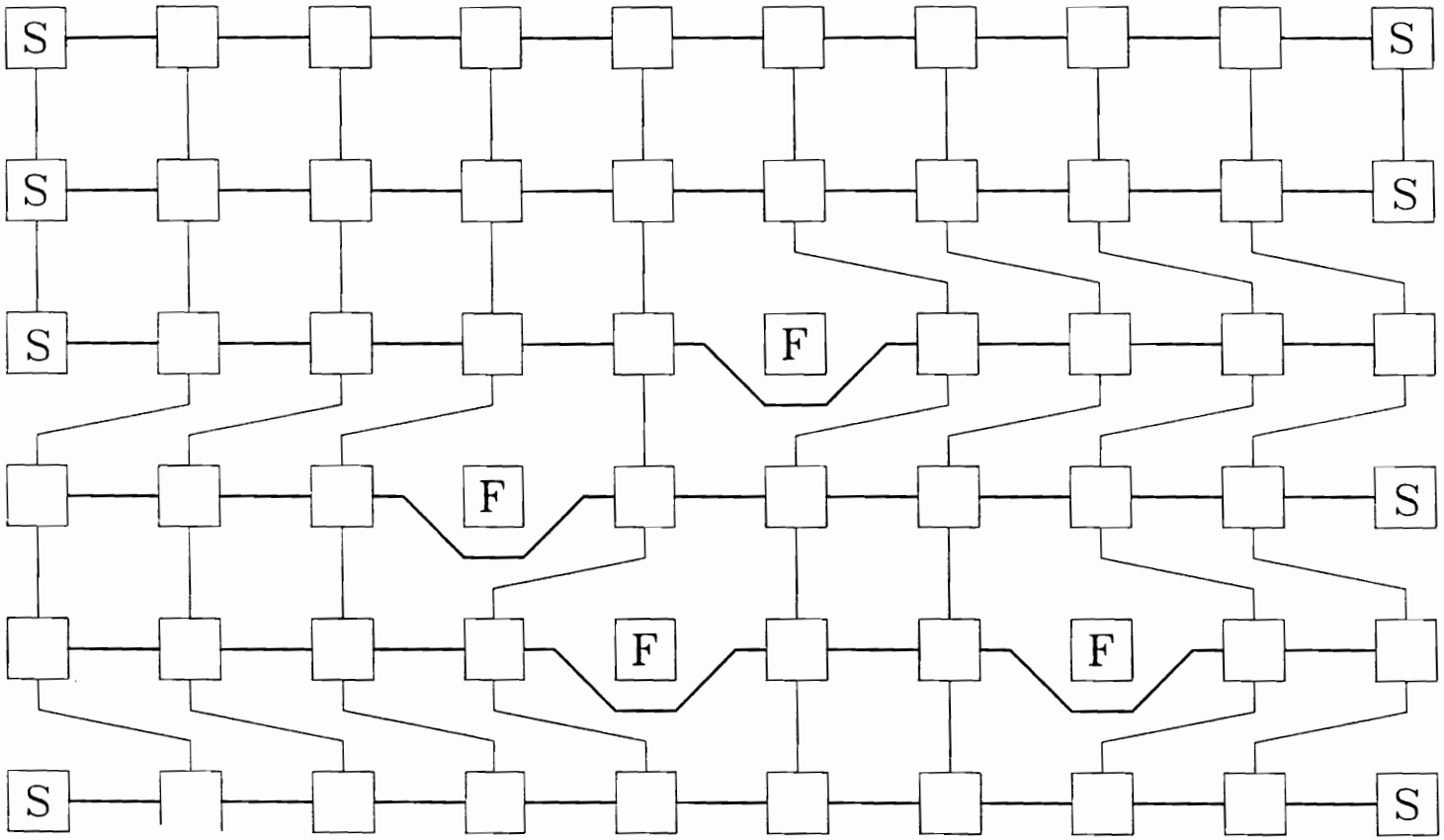


Figure 36. Four Faults: Array with four faults in a 1-1-2 pattern.

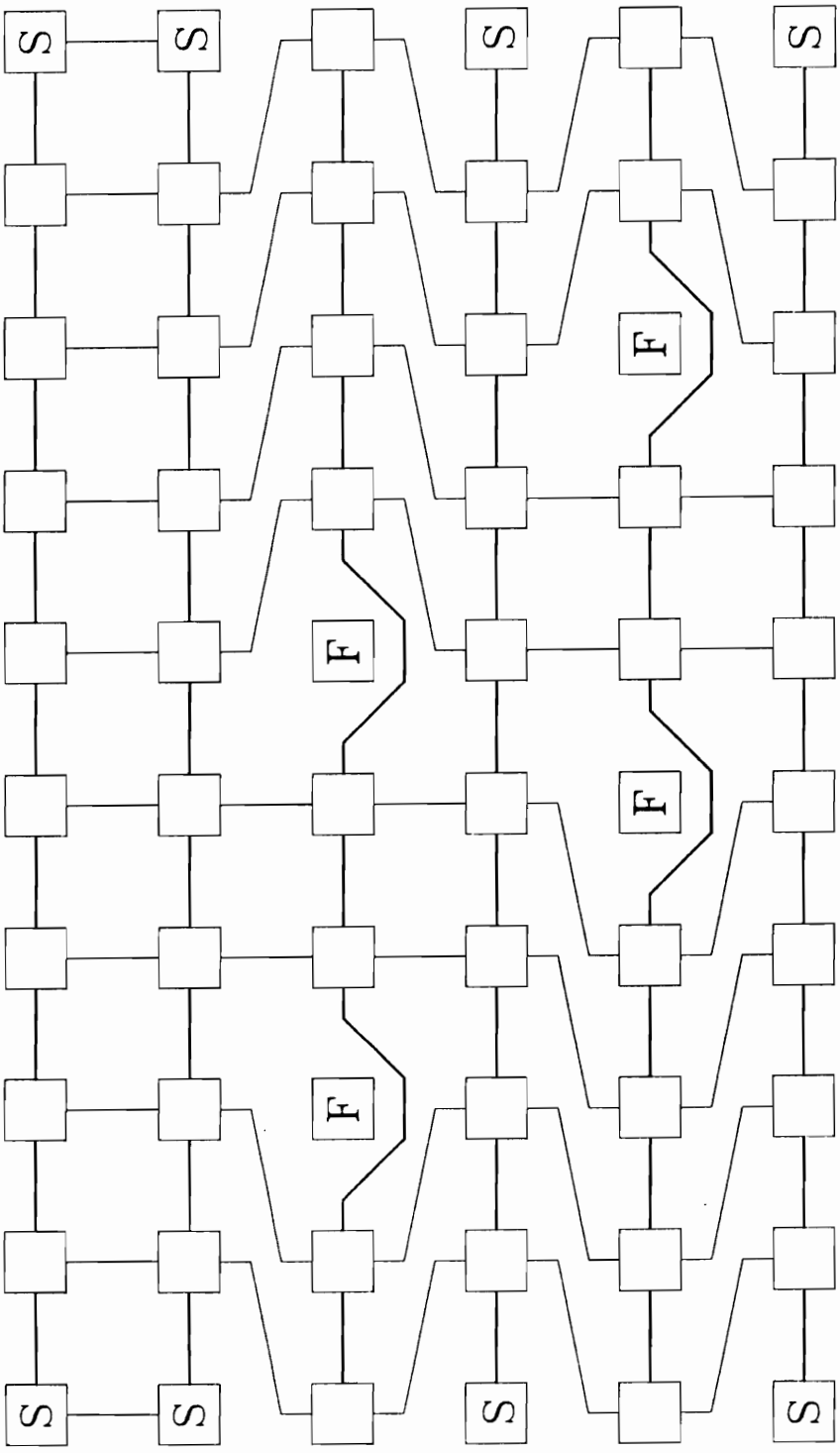


Figure 37. Four Faults: Array with four faults in a 2-2 pattern.

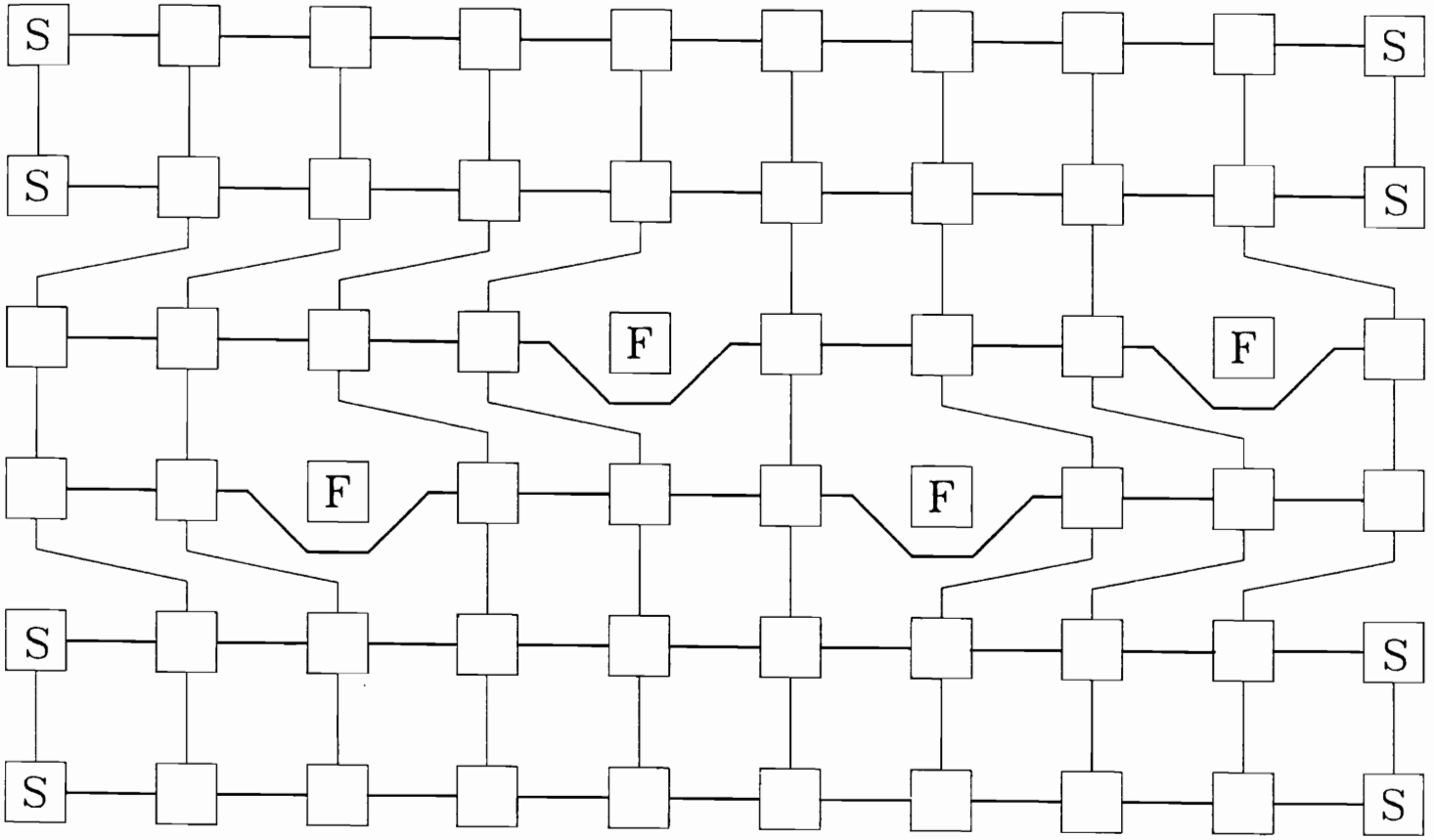


Figure 38. Four Faults Case 4-3.2: Array with four faults in a 2-2 pattern.

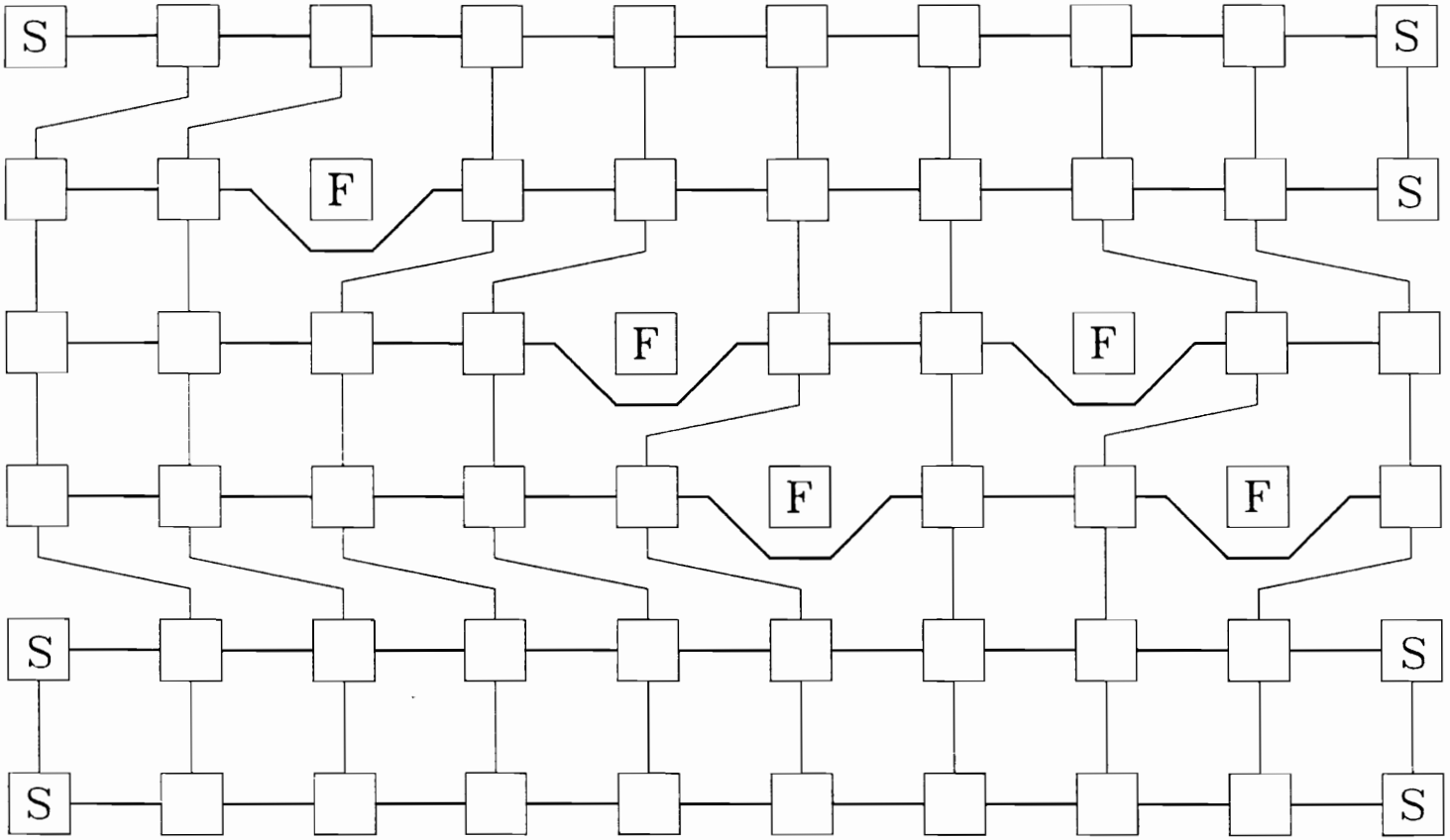


Figure 39. Five Faults Case 5-3: Array with five faults in a 1-2-2 pattern.

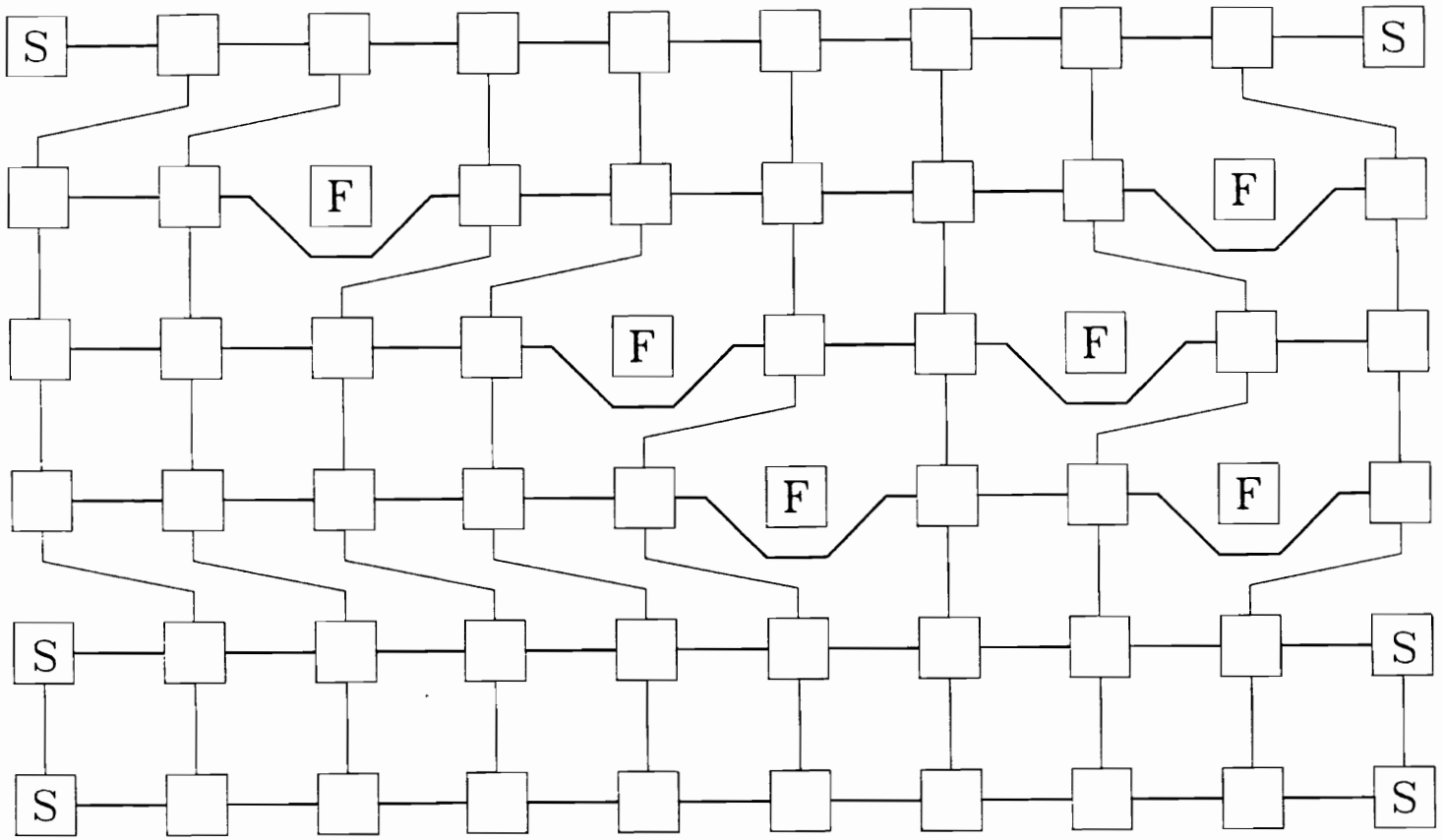


Figure 40. Six Faults: Array with six faults in a 2-2-2 pattern.

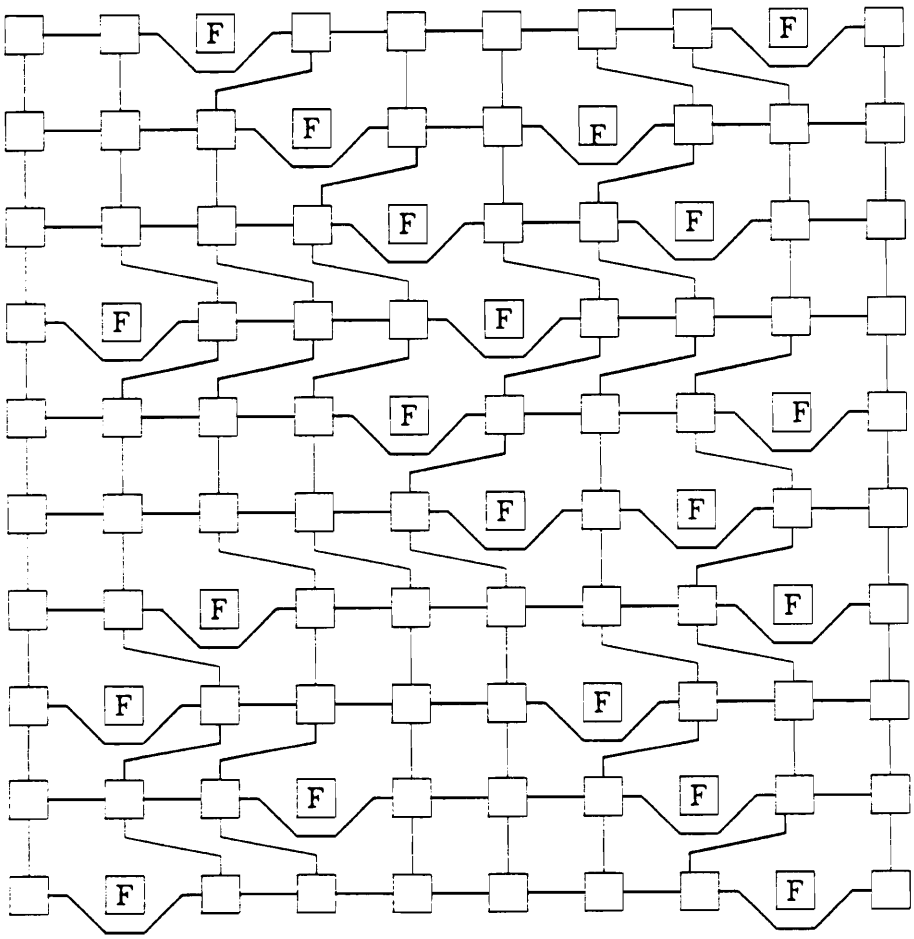


Figure 41. Maximum Tolerance: Array showing two faults in each row.

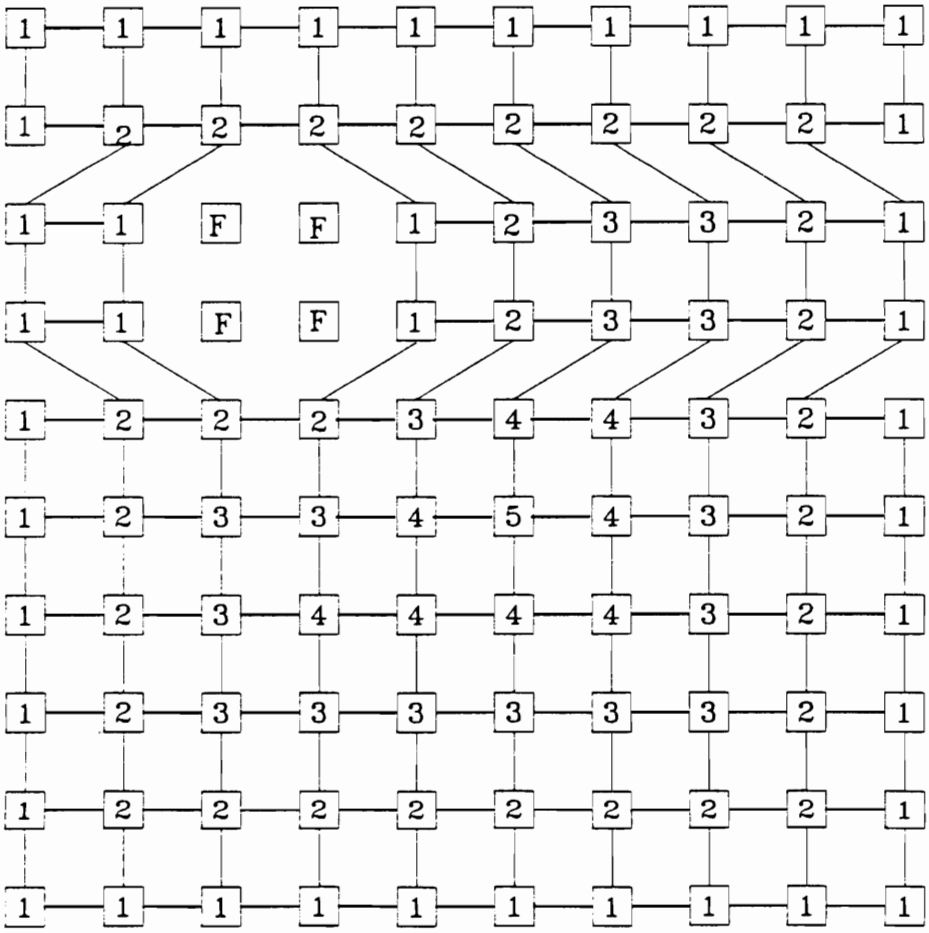


Figure 42. Clustered Faults: Array showing the effect of clustered faults on the s-value.

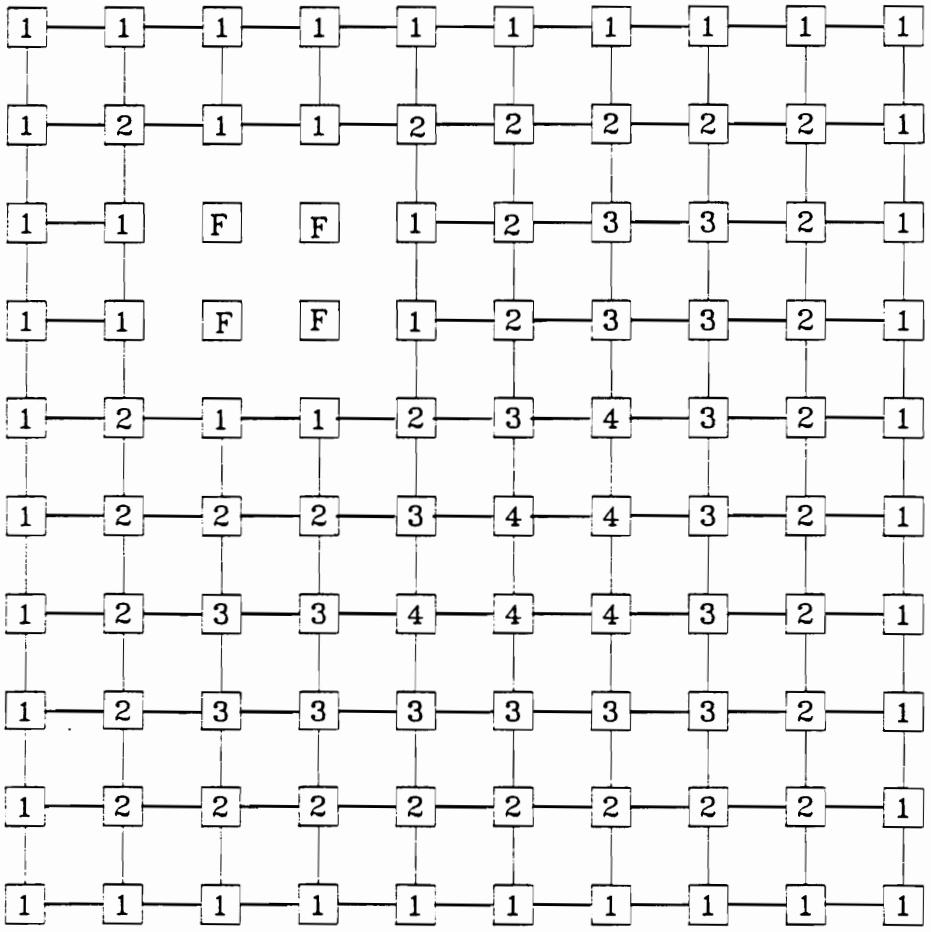


Figure 43. Clustered Faults: Array showing the effect of clustered faults on the s-value.

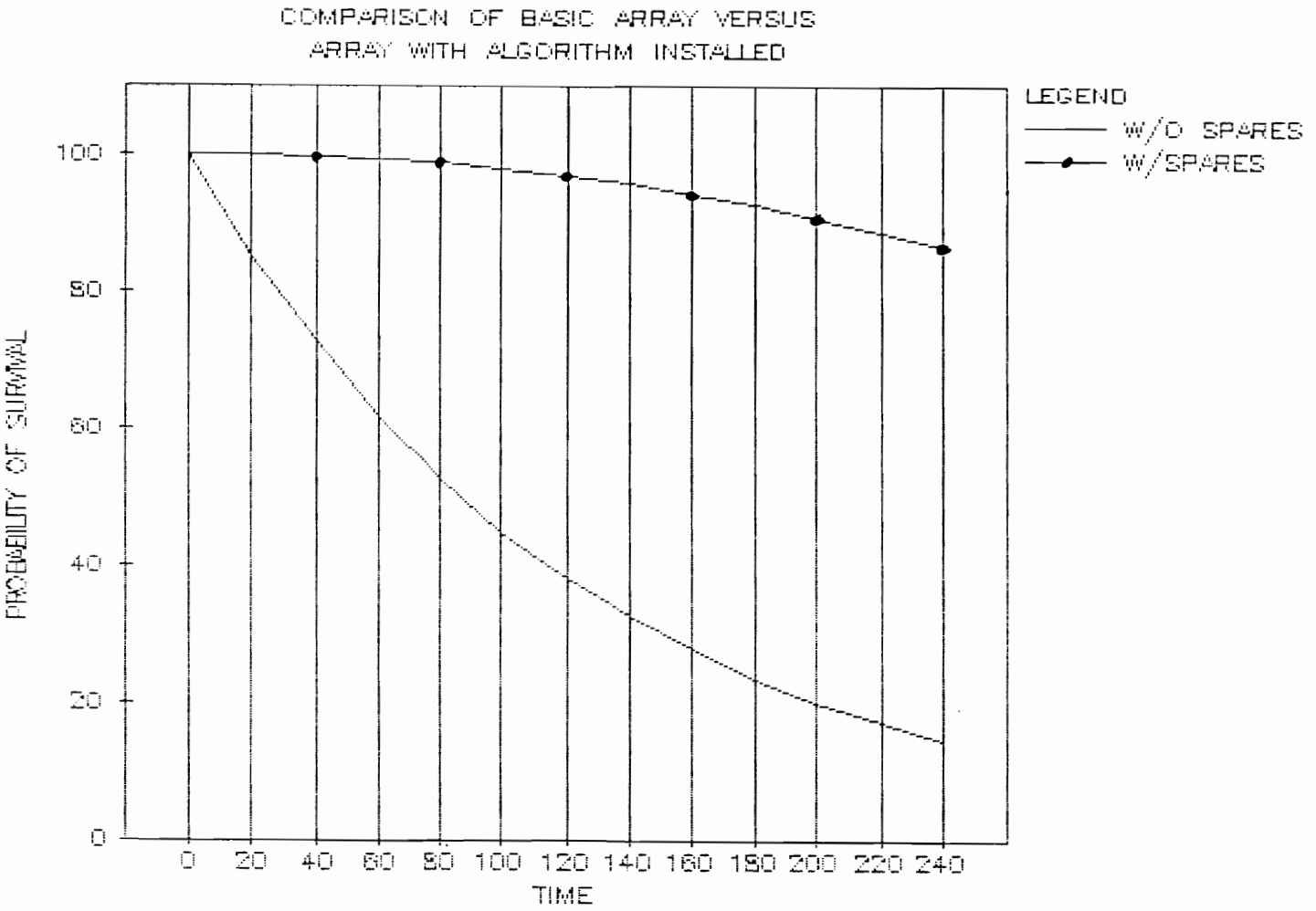


Figure 44. Probability of Surviving: Plot of survivability probabilities of the array with and without spares.

4.0 Algorithm 2-12

The second algorithm has a communications network which allows it to communicate with its twelve nearest neighbors. This includes all cells which are cell distance 1 or 2. Figure 45 on page 128 shows this neighborhood. In addition to the logic needed to incorporate the additional communications links, logic is also added to control the internal destination of several of the links. The cells in this algorithm which can be designated as the NORTH neighbor of cell[i,j] are:

- Cell[i-1,j]
- Cell[i-1,j-1]
- Cell[i-1,j + 1]
- Cell[i-2,j]
- Cell[i,j-1]
- Cell[i,j + 1]

The possible SOUTH neighbors of cell[i,j] are:

- Cell[i + 1,j]
- Cell[i + 1,j-1]
- Cell[i + 1,j + 1]
- Cell[i + 2,j]
- Cell[i,j-1]
- Cell[i,j + 1]

Control logic must be added to designate two cells, [i,j-1] and [i,j + 1] as either the NORTH, SOUTH or WEST neighbor in the case of the former, and either a NORTH, SOUTH or EAST neighbor for the latter. This additional logic is shown in Figure 46 on page 129

Figure 3 on page 68 shows the ten by eight logical array with two columns of cells to be used as spares.

4.1 Fault Register

Table 4 on page 127 shows the fault register used for this algorithm. Seven bits have been added to control the reconfiguration in cases of non-overlapping double faults and three and four faults in a row. Only 24 of the 32 bits are decoded. The remaining 8 bits are used to determine the values of some of the 24 decoded bits.

4.1.1 Setting the Fault-Register

Bits 0 through 20 and bits 24 through 26 are set exactly as bits 0 through 20 and bits 21 through 23 were set in algorithm 2-10 in the previous chapter. These bits perform the same function in both algorithms. Bit 21, indicating the necessity to generate a pseudo fault is set in cell[i,j] whenever a cell detects its south neighbor is faulty, and there are no faults to the W, and there are faults to its SW and SE. It is also set in the three cells which are nearest neighbors of cell[i,j]. In cell[i,j] it is used when combined with the MFS bit to create the pseudo fault in cell[i,j]. In cell[i-1,j], it is used to force the configuration of columns to the west. In row[i-1], it is propagated to the west to create the south-west configuration of the entire row.

If the input from the east indicates the EAST neighbor is the pseudo fault cell, with an input of PF, S, SE and SW, then both bits 21 and 22 are set. This happens only for cell[i,j-1], and this combination is used to force this cell to denote cell[i,j] as its SOUTH neighbor, and cell[i,j + 1] as its EAST neighbor. A similar occurrence happens in cell[i,j + 1] in which bits 21 and 27 are set. This combination forces cell[i,j-1] to be denoted as the WEST neighbor. Bit 27 is propagated to the east in row[i], and bit 22 is propagated to the west. If a cell has both bit 27 and bit 22 set, indicating pseudo faults to the east and the west, they essentially cancel each other.

Bit 29 is set if a cell has the conditions necessary to generate bit 21 above, and the input from the WEST cell indicates a pseudo fault to the west by having bit 27 set. If this occurs, cell[i,j] will have bits 29, 27 and 21 set, indicating it is the eastern fault of two pseudo faults. Bit 29 is propagated to the east in row[i]. Bit 28 is set if the 21-27-29 combination is received from the SOUTH neighbor. This bit indicates two pseudo faults to the south-west, and it is propagated to the east in row[i-1].

Bit 30, PFNW, is set if the input from the NORTH neighbor contains bit 29. This bit is propagated to the east in row[i + 1].

4.2 The Algorithm

Unlike the previous algorithm 2-10, the east/west links are not determined solely by the condition of the east or west neighbors. Because of the introduction of the pseudo fault condition, the EAST or WEST neighbor of a cell may be in one of the adjoining rows. For this reason, the definition of the algorithm is changed to include the EAST and WEST neighbors, as well as the NORTH and SOUTH neighbors. The following definition contains the complete list of a cell's neighbors in the order NORTH neighbor, SOUTH neighbor, EAST neighbor and WEST neighbor. Note that in

the switch and case statements, the pseudo-fault term is used to reference 8 bits, with the case statements listing the values which are set.

Note that coverage analysis has determined that algorithm 2-12 offers only marginal benefits over algorithm 2-10. Therefore, the following algorithm definition is not intended to be complete, but defines only the major points of the new sections of the algorithm.

Algorithm 2-12

```
{ /* begin algorithm 2-12 */

                                        East

    switch(E W FE FW PF)
1.  case(null)              connect east
2.  Case(E and not((FE and (W or FW)) or east = PF))
                            connect far-east
3.  Case(E and FE and (W or FW) and north=east = PF )
                            connect north-east
4.  Case((E and FE and (W or FW) and south-east = PF )
        or(PF))             connect south-east

                                        West

    switch(E W FW FE FW PF)

5.  case(null)              connect west
6.  Case(W and not((FW and(E or FE) or west = PF))
      connect far-west
7.  Case((W and FW and (E or FE)) or north-west = PF)
                            connect north-west
8.  Case((W and FW and (E or FE)) or south-west = PF)
        or PF)              connect south-west
    /* begin north link code */
    switch( MF MFN FCR FCRN PF)
     {

    case (null){
9.  if((not(W or FW) and  not(N or NW)) or ((W or FW) and  NW) )
                                                    connect north;
```

```
10. else  if(((N or NW) and  not(W or FW)))
                                                    connect north-east;
11. else  if (((W or FW) and  not(NW)))
                                                    connect north-west;
                                                    break; }

    case (FCR){
12. if(((W or FW) and  not(N or NW)) )
                                                    connect north;
13. else  if(((W or FW) and  (N or NW)) or (not(W or FW)))
                                                    connect north-east
14. else connect north-west /* not possible in this configuration */
                                                    break; }

    case (FCRN)    {
15. if (((NW) and  not(W or FW)))
                                                    connect north;
16. else  if ((not(NW)) or ((NW) and  (W or FW)))
                                                    connect north-west;
17. else  connect north-east;/* not possible in this configuration */
                                                    break; }

    case (FCR AND FCRN){
18. if((not(W or FW) and  not(N or NW)) or ((W or FW) and  (NW)))
                                                    connect north;
19. else  if((not(W or FW) and (N or NW)))
                                                    connect north-east;
20. else  if (((W or FW) and  not(NW)))
                                                    connect north-west;
                                                    break; }

    case (MF)    {
21. if(((W or FW) and  (E or FE) and not(N or NW))
                        or (NW and  not(E or FE)))
                                                    connect north;
22. else if((not(W or FW)) or ((W or FW) and  (E or FE) and  (N or NW)))
                                                    connect north-east;
23. else if((not(E or FE) and  not(NW)))
                                                    connect north-west;
                                                    break; }

    case (MF AND FCRN){
24. if((NE) or ((W or FW) and  (E or FE)))
                                                    connect north;
```

```
25. else if((not(W or FW) and  (N or NW)))
                                                    connect north-east;
26. else if((not(E or FE)))
                                                    connect north-west;
                                                    break; }

    case (MFN)  {
27. if((not(W or FW) and  (NW and NE)
        or ((W or FW)  and  (not(N or NE))))
                                                    connect north;
28. else if((not(W or FW or NE)))
                                                    connect north-east;
29. else if((not(NW)) or ((W or FW) and  (N or NE) and (NW)))
                                                    connect north-west;
                                                    break; }

    case (MFN AND FCR){
30. if((not(W or FW)) or ((NW and NE))
                                                    connect north;
31. else if((not(NE)))
                                                    connect north-east;
32. else if(((W or FW) and not(NW)))
                                                    connect north-west;
                                                    break; }

    case (MF AND MFN)
    {
33. if((not(W or FW) and  not(N or NW))
       or ((W or FW) and (E or FE) and (NW and NE))
       or (not(E or FE)                             and  not(N or NE)))
                                                    connect north;
34. else if((not(W or FW) and  (N or NW))
      or ((E or FE) and  (W or FW) and  not(NE)))
                                                    connect north-east;
35. else if(((W or FW) and  not(NW)) or (not(E or FE) and  (N or NE)))
                                                    connect north-west;
                                                    break; }
    }

     case(PF1E )    /* single pseudo fault to the east */
     {
36.  if(not FCRN or (FCRN and( N or NW)))
       connect north-east(NE)
37. if(FCRN and not(N or NW))
       connect north(N)
38. if (PF )  /* pseudo fault injected this cell)
```

```
      connect west(W)
    }
  case(PF1NE)
    {
39. if(not(W or FW))
      connect north
40. if(W or FW)
      connect north-west(NW)
    }
  case(PF2W)   /* this is set only if two pseudo faults in a row */
    {
41. if(not NW and not(PF))
    connect north-west
42. if(NW  and not(PF ))
    connect north
43. if(PF )
    connect east
    }
  case(PF2NW) /* set only if two pseudo faults in a row */
    {
44. if(E or FE)
    connect north-east
45. if(not(E or FE))
    connect north
    }
46. case(N and NW and NE)    connect far-north
                          /*  end north configuration algorithm   */
  /*  begin south link code  */
  switch(MF MFS FCR FCS PF)
  { /* south link switch code */

  case (null):
    {
47. if((not(W or FW) and  not(S or SW)) or ((W or FW) and  (SW)))
                                                    connect south;
48. else if(((W or FW) and  not(SW)))
                                                    connect south-west;
49. else if((not(W or FW) and  (S or SW)))
                                                    connect south-east;
                                                    break; }

  case (FCR)
   {
50. if(((W or FW)  and  (not(S or SW))))
                                                    connect south;
51. else if(((W or FW)  and  ((S or SW))) or (not(W or FW)))
```

```
                                                    connect south-east;
52. else connect south-west; /* not possible this configuration */
                                                    break; }

    case (FCRS)
      {
53. if((not(W or FW) and  SW))
                                                    connect south;
54. else if((not(SW)) or ((SW)  and  (W or FW)))
                                                    connect south-west;
55. else   connect south-east;/* not possible this configuration */
                                                    break; }

    case (FCR AND FCRS)
      {
56. if((not(W or FW)) or ((W or FW) and  SW))
                                                    connect south;
57. else if(((W or FW) and  not(SW)))
                                                    connect south-west;
58. else if((not(W or FW) and (S or SW)))
                                                    connect south-east;
                                                    break; }

    case (MF)
      {
59. if(((W or FW) and  (E or FE) and  not(S or SW)))
                                                    connect south;
60. else if((not(E or FE or SW)))
                                                    connect south-west;
61. else if((not(W or FW)) or ((W or FW) and  (E or FE) and (S or SW)))
                                                    connect south-east;
                                                    break; }

    case (MF AND FCRS)
      {
62. if((SE) or ((W or FW) and  (E or FE)))
                                                    connect south;
63. else if((not(E or FE)))
                                                    connect south-west;
64. else if(((S or SW) and  not(W or FW)))
                                                    connect south-east;
                                                    break; }

    case (MFS)
     {
```

```
65. if(((SW and SE) and not(W or FW)))
                                                    connect south;
66. else if((not(SW)) or ((SE or S) and  (SW) and  (W or FW)))
                                                    connect south-west;
67. else if(not(W or FW or SE))
                                                    connect south-east;
                                                    break; }

    case (MFS AND FCR)
      {
68. if((SW and SE) or (not(W or FW)))
                                                    connect south;
69. else if(((W or FW) and  not(SW)))
                                                    connect south-west;
70. else if(not(SE))
                                                    connect south-east;
                                                    break; }

    case ( MF AND MFS)
      {
71. if(((W or FW) and  (E or FE) and  (SW and SE))
          or (not(E or FE) and  not(SE or S))
          or (not(W or FW) and  not(S or SW)))
                                                    connect south;
72. else if(((W or FW) and  not(SW))
          or (not(E or FE) and  (S or SE)))
                                                    connect south-west;
73. else if(((W or FW)and  (E or FE) and  not(SE))
          or (not(E or FE) and  (S or SE))
          or (not(W or FW) and (S or SW)))
                                                    connect south-east;
                                                    break; }

    case(PF1SE)
      {
74. if(not FCR or(FCR and (W or FW))
        connect south-west
75. if(FCR and not(W or FW))
        connect south
      }
    case(PF1E)
      {
76. if(S or SW)
        connect south
77. if(S or SW and not(east = PF)/* pseudo fault not in cell j + 1 */     [i,j + 1] */
        connect south-east
78. if(east = PF)
```

```
        connect east
79.  if(PF)  /* pseudo fault induced this cell */
        connect farsouth
       }
     case(PF2W)
       {
80.  if(S or SE)
       connect south-west
81.  if(not(S or SE))
       connect south
82.  if(west = PF) /* second pseudo fault injected cell[i,j-1] */
       connect west
83.  if(PF)
       connect farsouth
      }
                                              case(PF2SW)
     {
84.  if((W or FW) and not FCR)
       connect south
85.  if(not(W or FW))
       connect south-east
      }}
    } /* end south link switch code */

    } /* end reconfigure */
```

4.3 Fault Coverage

4.3.1 Single Faults

Theorem 2.1: An array equipped to use algorithm 2-12 will tolerate all single fault occurrences.

Proof: Algorithm 2-12 is identical to algorithm 2-10 for single faults. See the previous chapter for single fault coverage.

4.3.2 Double Faults

Theorem 2-2: An array equipped with hardware to implement this algorithm will successfully re-configure around all double faults except those in which the faults exist in adjacent cells.

Proof: Algorithm 2-12 handles the same double fault configurations tolerated by algorithm 2-10. See the previous chapter for this proof.

4.3.3 Three faults.

Theorem 2-3: An array implementing this algorithm will tolerate all triple faults except those in which two or three faults occur in adjacent cells in the same row.

Proof: Triple faults can exist in the following cases:

1. 1-1-1
2. 1-2
3. 3

4.3.3.1 Case 3-3 Three Faults in one Row.

Cases 1 and 2 are identical to those in algorithm 2-10 and described in the previous chapter. Case 3-3 is the only triple fault instance which is new to this algorithm. This case is tolerated if no two of the three faults is in adjacent cells. Figure 47 on page 130 shows one possibility for a three fault occurrence in one row. For discussion, assume the faults occur in cell[i,j], cell[i,k] and cell[i,ℓ] | $j+1<k<$ &l.-1. Cells in row[i-2] which are north or west of the center fault, cell[i-2,m] | $m<=k$, are in algorithm configuration 9-74-1-5, and configure to the north, south-west, east and west. All other cells in row[i-2] are in configuration 9-47-1-5. This configuration change in row[i-2] is created

because the pseudo-fault-south-east(PFSE) bit is only set in cells north and west of the center fault. Cell[i-1,m] | m < j is in configuration number 38-76-1-5 and connects to the north-east, south, east and west. Cell[i-1,m] | j < = m < k-1 has configuration 36-77-1-5 and connects to the north-east and south-east. Cell[i-1,k-1] takes on the roll of cell[i-1,k] which has generated the pseudo fault, and is in configuration number 38-79 -2-5. It connects to cell[i-1,k] as its southern neighbor, and to cell[i-1,k + 1] as its eastern neighbor. Cell[i-1,k] has configuration number 43-79-4-8. This cell logically switches the input from its western neighbor to its northern neighbor circuits, making cell[i-1,k-1] its northern neighbor, and the cell[i + 1,j] as its southern neighbor. It also connects eastern and western input circuits to accept input from its south-east and its south-west neighbors respectively. Cell[i-1,k + 1], which is placed in configuration number 9-65-1-6 completes the configuration of row[i-1] by connecting to its far-west neighbor, cell[i-1,k-1]. Cells to the east of the center fault, cell[i-1,n] |n > k + 1 have the same configuration as the same cells in case 2-1, the double fault.

Cell[i,m] | m < j-1 has a pseudo fault to its north-east and no faults to its west, placing it in configuration number 39-61-1-5 with connections to the north and south-east, while cell[i,j-1] has configuration 39-61-2-5 and connects to the north, south-east, far-east and west. Cell[i,j + 1] has the pseudo fault to its north-east and faults to its west, placing it in configuration 40-59-1-6. This cell connects north-west, south, east and far-west. Cell[i,m] | j + 1 < m < k-1 connects to the north-east, south, east and west because it has a fault configuration of 40-49-1-5. The remaining two cells involved in configuring around the pseudo fault are cell[i,k-1] and cell[i,k + 1]. Cell[i,k-1] will be in either configuration number 40-59-2-5 or 40-59-2-6. Figure 47 shows configuration 40-59-2-5. This cell connects to the north-west, south, north-east and west. If j = k-2, meaning there is only one fault-free cell between the faults, cell[i,k-1] would configure to the far-west instead of the west. Cell[i,k + 1] has faults west, far-west and far-east, placing it in configuration 21-59-1-7, with connections to the north, south, east and north-west. If 1 = k + 2, this cell would be in configuration 21-59-2-7, and connect to its far-east instead of its east. All other cells in row[i] configure identically to those in case 2-1.

Cells in row[i + 1] configure identically to algorithm 2-10, case 2-1, with two faults in a single row, except for cell[i + 1,k]. Cell[i + 1,m] | m < k is affected only by the faults at cell[i,j] and cell[i,k], while cell[i + 1,m] | m > k sees only faults at cell[i,k] and cell[i,ℓ]. Cell[i + 1,k] has faults north, north-west and north-east, with no fault to its north-north-west, meaning no fault in cell[i-1,m] | m < k. This places this cell into configuration number 46-48-1-5 and it connects to cell[i-1,k] as its northern neighbor. All other cells in the four rows involved in this fault pattern see only a double fault, and configure identically to algorithm 2-10, case 2-1.

It has been shown that an array equipped to support reconfiguration algorithm 2-12 will tolerate three faults.

■

4.3.4 Four Faults

Theorem 2-4: An array equipped with hardware to invoke this algorithm will tolerate all instances of four faults except those in which two faults lie in adjacent cells in the same row or two faults are adjacent in the same column, if that column contains the center fault of three in one row.

Proof: Four faults can exist in the following cases:

1. 1-1-1-1
2. 1-1-2
3. 1-3
    a. Single and triple faults separated two or more rows.
    b. Single and triple faults separated by one fault-free row
    c. Single in adjacent row, north-west of center fault
    d. Single in adjacent row, north-east of center fault
    e. Single in adjacent row, north or south of center fault

f. Single in adjacent south-west of center fault

g. Single in adjacent south-east of center fault.

4. 4

4.3.4.1 Case 4-3

Cases 4-1 and 4-2 are identical to cases in algorithm 2-10 and proof of their configuration is given in the previous chapter. Case 4-3, with a 1-3 fault configuration has seven variations which affect the reconfiguration process.

Figure 48 on page 131 shows case 4-3a in which single and triple faults are separated by two rows. They reconfigure as separate fault occurrences, a single fault and a triple fault. Reconfiguration of one does not affect the other. These processes were described in the previous chapter case 1, and in this chapter in case 3-3.

Figure 49 on page 132 shows one possibility for case 4-3b. In this case the faulty cells are cell[i-2,j], cell[i,k], cell[i,ℓ] and cell[i,m] with $j < \ell$, placing the single fault to the north-west of the pseudo fault. If the single fault had been east of the pseudo fault, or in row[i + 2], the reconfiguration process around the single fault and the triple fault would not affect each other. Cells in row[i] and row[i + 1] have identical configurations to the same cells in case 3-3, with three faults in a single row. Cells in row[i-3] are identical to those in a single fault occurrence in which the FCRS bit is set as described in case 3-2.2a. This bit was set because the single fault occurred in an area in which the PFSE bit was set. Cell[i-2,n] | $n < j$ has the PFSE and FCR bits set with no fault to the west, so they connect to the north-east and south. Cell[i-1,ℓ] | $\ell < j$ has PFE and FCRN and no fault to the N or NW, so it connects to the north, with the southern link identical to case 3-3, three faults in one row. If the single fault occurred either in row[i + 2] or in row[i-2] east of the center fault, that is cell[i,j] | $j > \ell$, the reconfiguration process would treat them as independent faults. The single fault

would have no affect on the triple fault configuration. All other cells involved in the reconfiguration process in row[i-1] through row[i + 1] configure identically to case 3-3, three faults in one row.

4.3.4.2 Case 4-3c

Figure 50 on page 133 shows the first occurrence of case 4-3c, a 1-3 configuration in which the single fault occurs in a row which is adjacent to the row with the triple fault. In this example, the faults are in cell[i,j], cell[i + 1,k], cell[i + 1,ℓ] and cell[i + 1,m], with j = k. Reconfiguration for row[i-1] and row[i] are identical to case 3-2 in algorithm 2-10, with the exception of cell[i,ℓ], which is directly north of the center fault of the triple. This cell is in configuration pattern number 9-67-1-5 and selects cell[i + 2,ℓ] as its southern neighbor. Configuration around the pseudo fault, which is now placed in cell[i + 2,ℓ] is the mirror image of case 3-2 of this algorithm.

Figure 51 on page 134 shows an occurrence of the 1-3 pattern in which the single cell was east of the center fault. This is case 4-3d. Cells to the west of the single fault configure as if the single fault did not exist, case 3-3, and cells to the east configure as if the fault pattern were a 1-2, case 3-2 in the previous chapter. If the single fault occurs in row[i + 2], the location of the pseudo fault is left in row[i-1]. Figure 52 on page 135 shows this occurrence, with the single fault west of the center fault in row[i + 1] and the triple fault in row[i]. If the center fault of the triple occurs in cell[i,j], the single fault may occur in either row[i-1] or row[i + 1] except cell[i-1,j] and cell[i + 1,j]. These two conditions constitute a fatal error, case 4-3e.

4.3.4.3 Case 4-4.

Figure 53 on page 136 shows one possibility for case 4-4, with four faults in one row. In this case, two pseudo faults are generated in row[i-1]. We assign faulty cells the designators cell[i,j], cell[i,k], cell[i,ℓ] and cell[i,m]. To simplify the discussion, we also assume cell[i,j] and cell[i,m] were previ-

ously faulty, and the reconfiguration process around these faults is complete when the interior faults occur. When the interior fault occurs, two groups of 4 cells know a pseudo fault is being injected. They are cell[i-1,k], cell[i + 1,k], cell[i,k-1] and cell[i,k + 1] for the fault in cell[i,k] and cell[i-1,ℓ], cell[i + 1,&l.], cell[i,l-1] and cell[i,l + 1] for cell[i,ℓ]. At this time, information about both pseudo faults is being transmitted westward in row[i-1]. A single bit in the fault register is also propagated to the east in the same row. At this point in time, reconfiguration is proceeding as if only one pseudo fault exists.

At some point in time, cell[i-1,n] learns of both pseudo faults. The existence of the PFE and PFW bits in its fault register triggers an operation which clears the PFE bit in row[i] as well as the bits in the cells in rows[i-1] and row[i + 1] with which cell[i-1,n] is communicating. This process continues until cell[i-1,ℓ] learns of the pseudo fault at cell[i-1,k]. At this time, cell[i-1,ℓ] begins propagating information to the east which will force this pseudo fault to be configured to the east instead of the west. Figure 53 on page 136 shows the final configuration around two pseudo faults. Cells in columns west of column[l-1] configure exactly as they did in case 3-3 with only three faults in the row. Cells to the east of the eastern pseudo fault are the mirror image of those west of the western pseudo fault.

■

4.3.5 Coverage Analysis

Coverage analysis of a ten by ten array equipped with this algorithm has been analyzed. It has been determined that although this algorithm provides some advantages over algorithm 2-10, the advantages are only marginal. Tolerance to single faults is 100 percent, as was that of algorithm 2-10, and algorithm 2-12 will tolerate the same number of occurrences of two faults.

With three faults, the only advantage of an array equipped with hardware to support algorithm 2-12 over one equipped to support algorithm 2-10 is the tolerance of all three faults in one row. There are 56 possible occurrences of three faults in a single ten-cell row which can be tolerated by this algorithm. With an array with ten rows, this adds only 560 possible conditions which can be tolerated. When compared to the 161,700 possible three-fault combinations, and added to the 136,200 already tolerated by the previous algorithm, the increase is less than one-half of one percent.

Algorithm 2-10 equipped arrays would tolerate 66 percent of the possible occurrences of four faults. With the additional hardware necessary to implement algorithm 2-12, the number of four-fault occurrences which could be tolerated would be increased by only 55,182. When compared to the faults already tolerated by algorithm 2-10, the increase is less than 1.5 percent.

The marginal increase in survivability combined with the extra logic necessary to implement this algorithm to allow switching the inputs to various input/output registers makes this particular algorithm of only marginal interest.

Table 4. Fault Register Contents

BIT NO.	DESCRIPTION	PHYSICAL LOCATION OF FAULTY CELL
0	N	[i-1,j]
1	NW	[i-1,k] | 0 < k < j
2	NE	[i-1,k] | k > j
3	S	[i + 1,j]
4	SW	[i + 1,k] | 0 < k < j
5	SE	[i + 1,k] | k > j
6	E	[i,j + 1]
7	FE	[i,k] | k > j
8	W	[i,j-1]
9	FW	[i,k] | 0 < k < j-1
10	NN	[i-2,j]
11	NNW	[i-2,k] | 0 < k < j
12	NNE	[i-2,k] | k > j
13	MFNN	Two Faults in Row [i-2],
14	SS	[i + 2,j]
15	SSW	[i + 2,k] | 0 < k < j
16	SSE	[i + 2,k] | k > j
17	MFSS	Two Faults in Row [i + 2],
18	MF	Two Faults in Row [i]
19	MFN	Two Faults in Row [i-1]
20	MFS	Two Faults in Row [i + 1]
21	PF	Psuedo fault
22	PF1SE	Pseudo fault south and east
23	PF1E	Pseduo fault east
24	FCR	Fault in Critical Region, That is a Fault Row [i], with MFN and no faults west of this fault in row [i-1], or MFS set and no faults west of this fault in row [i + 1]
25	FCRN	FCR set in row [i-1]
26	FCRS	FCR set in row [i + 1]
27	PF1NE	Pseudo fault west
28	PF2SW	Two Pseudo faults south-west
29	PF2W	Two Pseudo faults west
30	PF2NW	Two Pseudo faults north-west

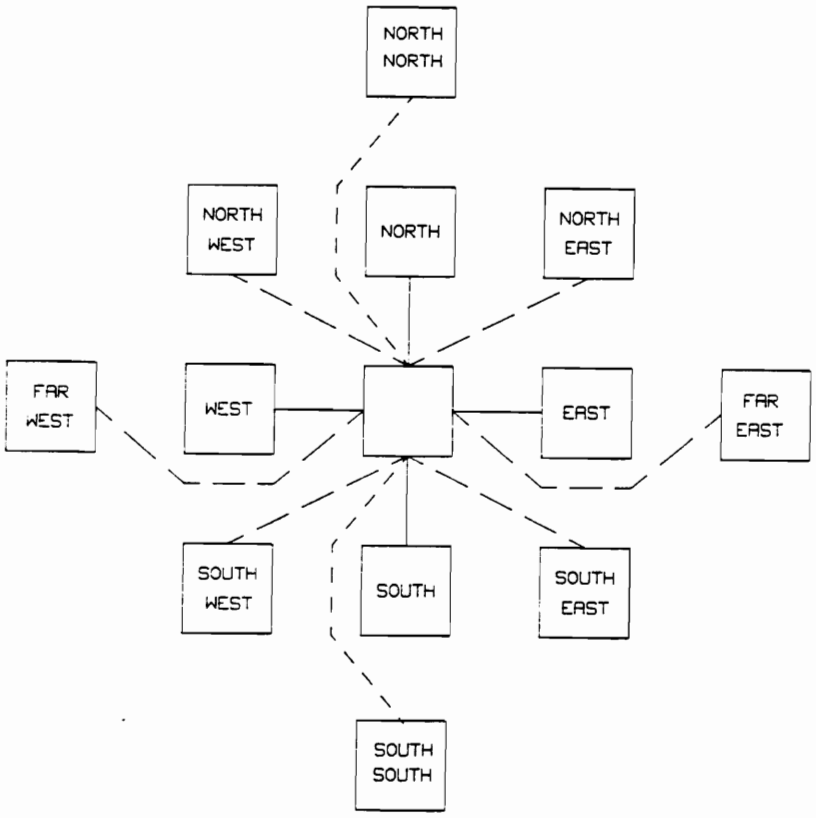


Figure 45. Twelve Neighbor Interconnection: Cell[i,j] and its twelve possible neighbors.

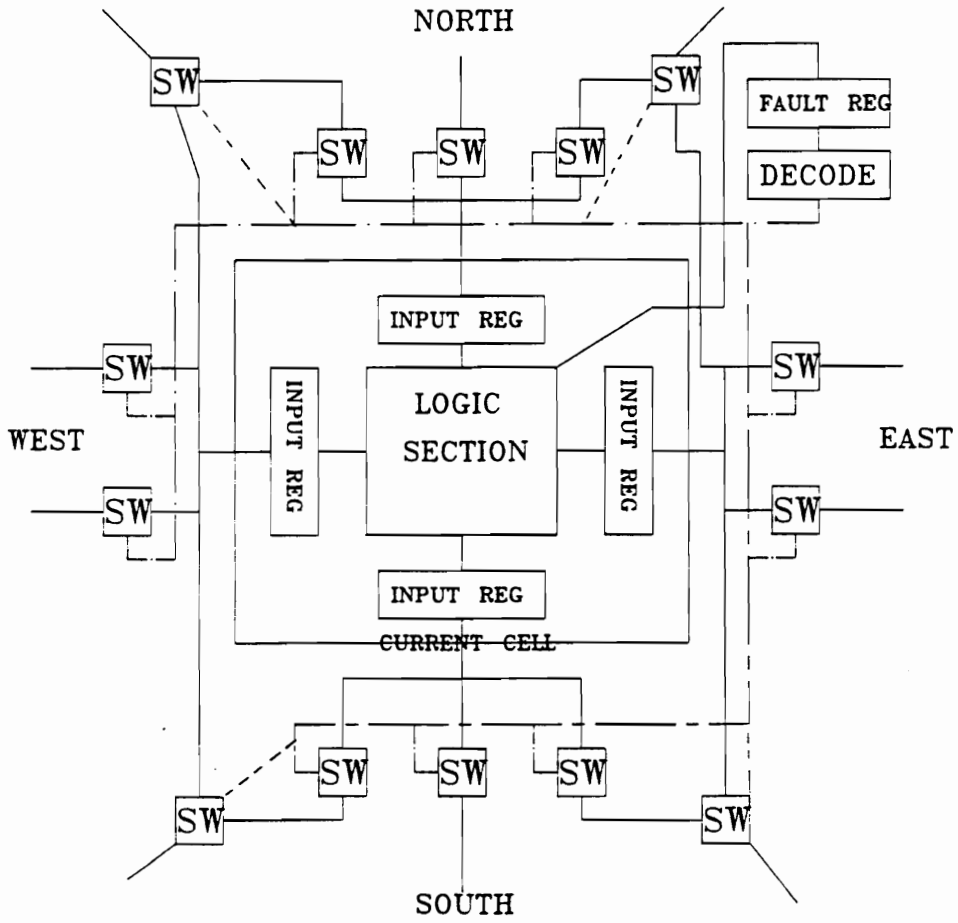


Figure 46. Typical Cell: Showing new logic needed to gate eastern and western cells to north or south link.

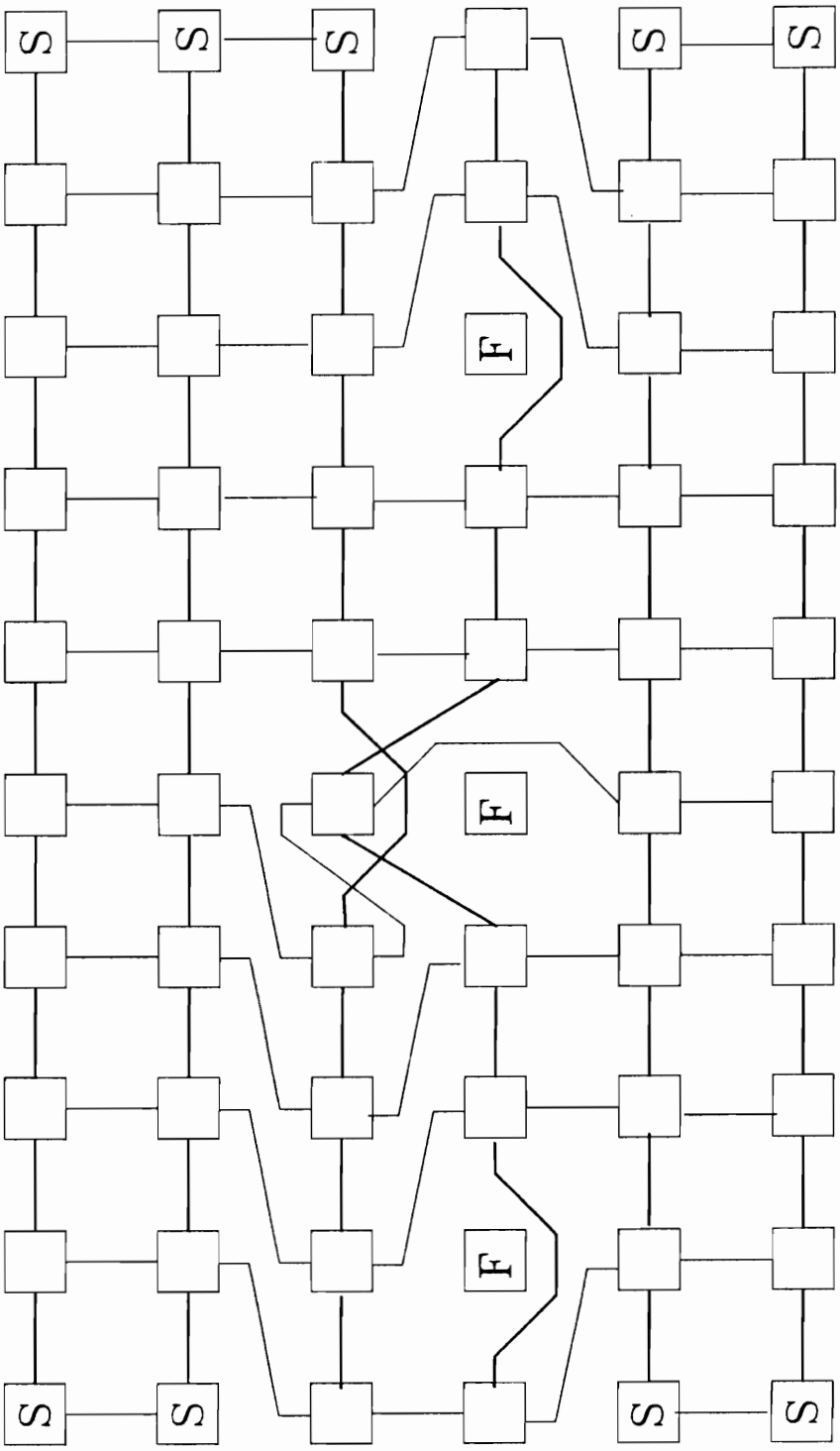


Figure 47. Triple Faults Case 3-1: Array with three faults in one row.

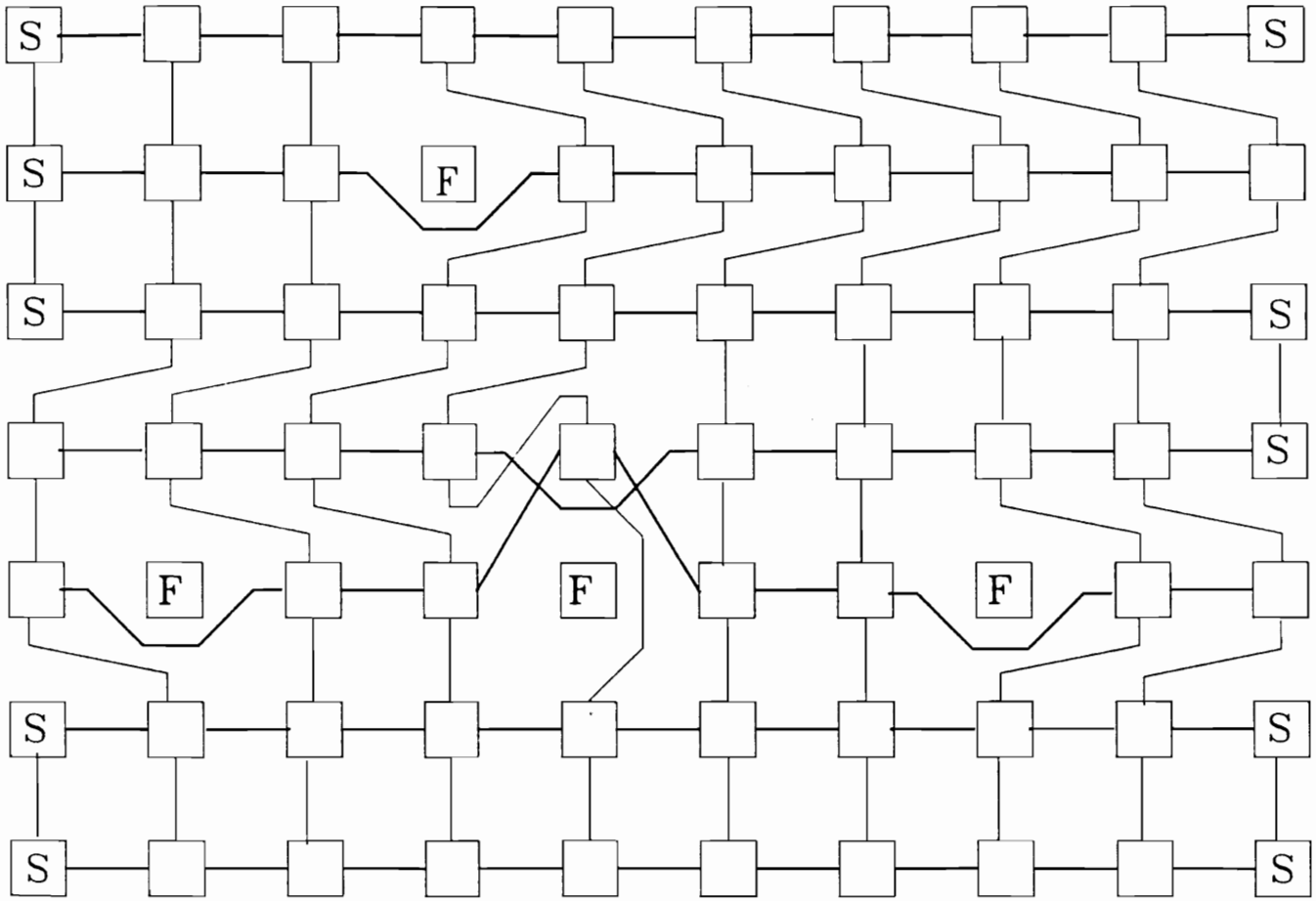


Figure 48. Four Faults Case 4-3a: Array with faults in a 1-3 pattern separated by two rows.

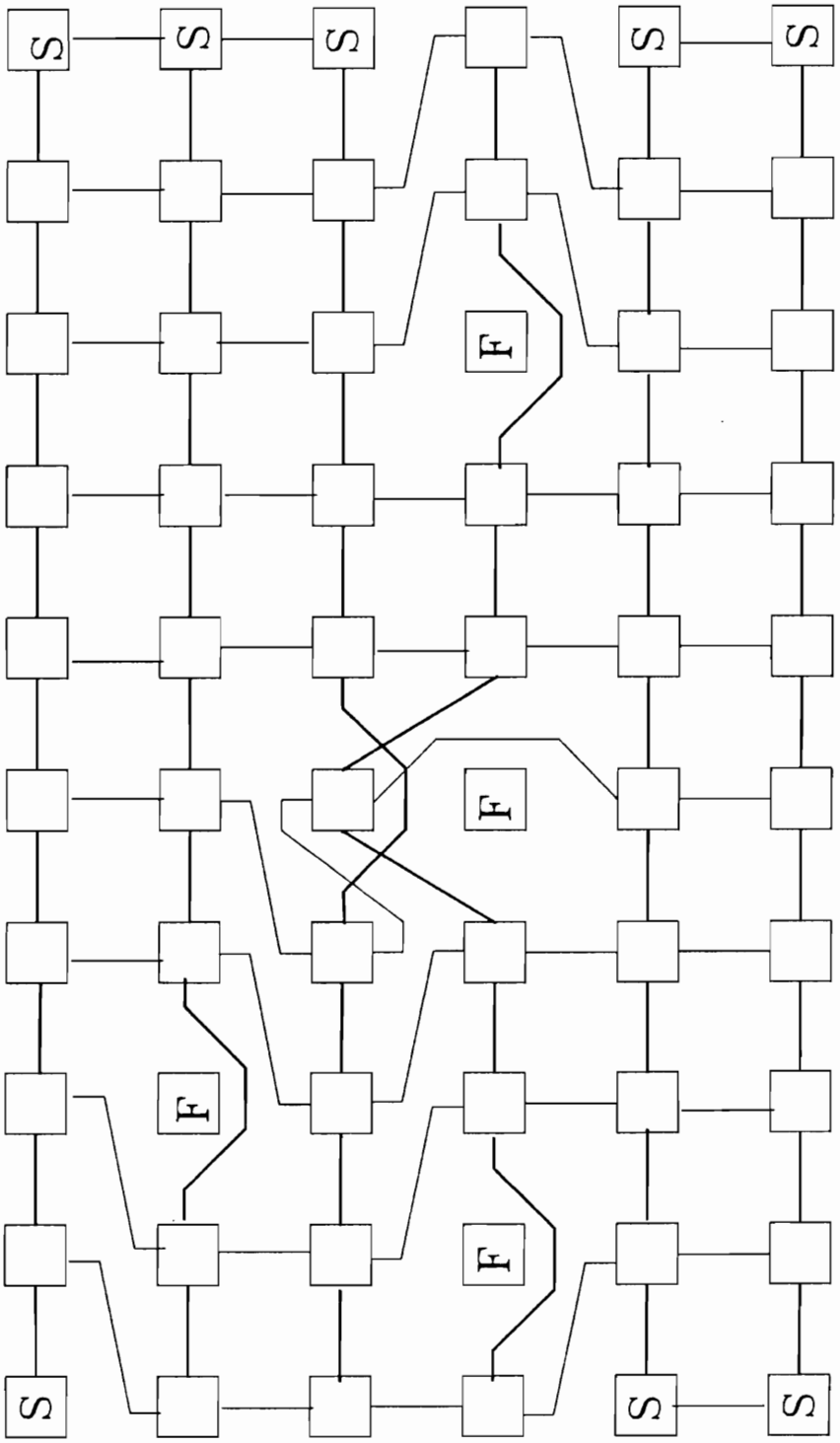


Figure 49. Four Faults Case 4-3b: Array with faults in a 1-3 pattern separated by one row.

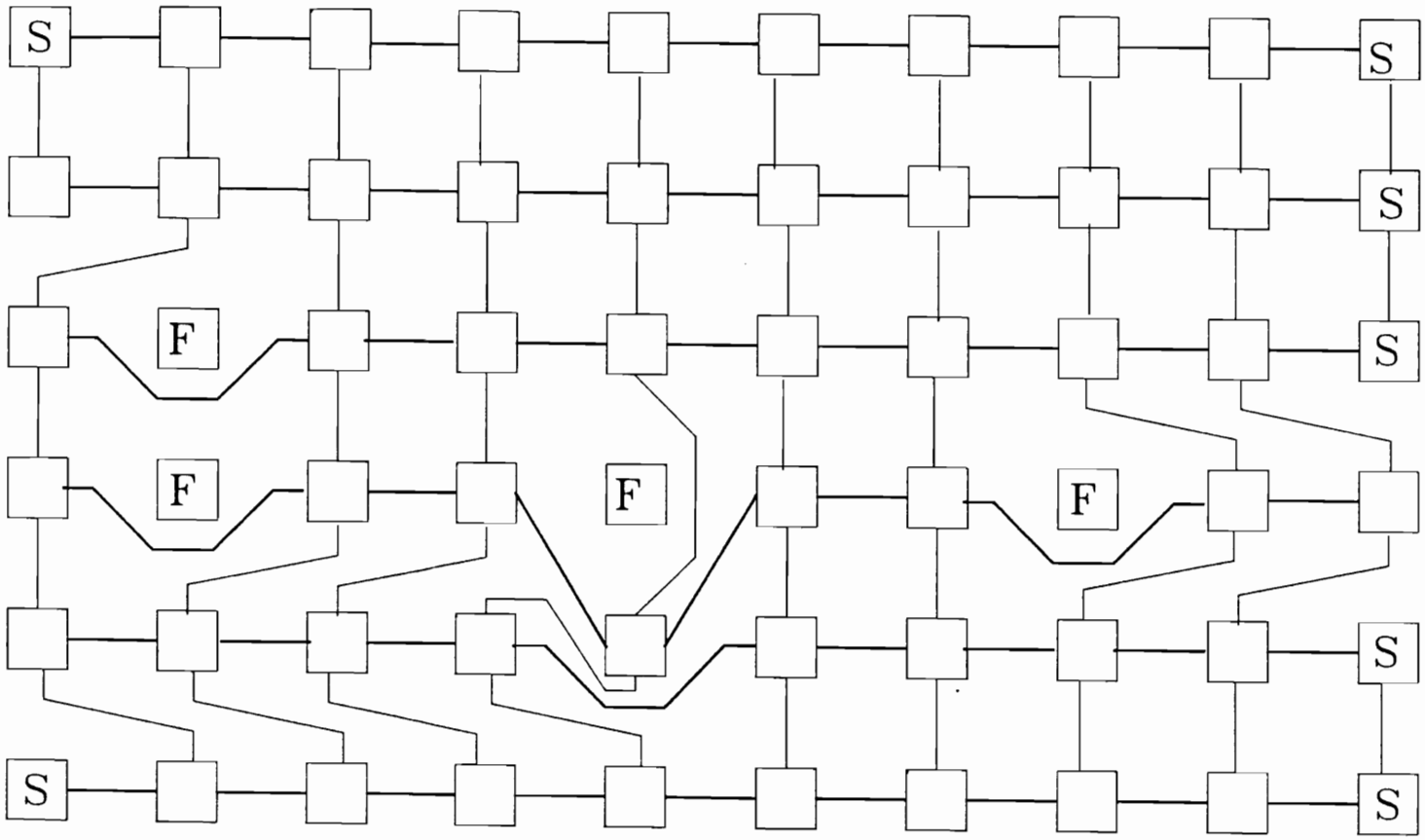


Figure 50. Four Faults Case 4-3c: Array with four faults in a 1-3 pattern.

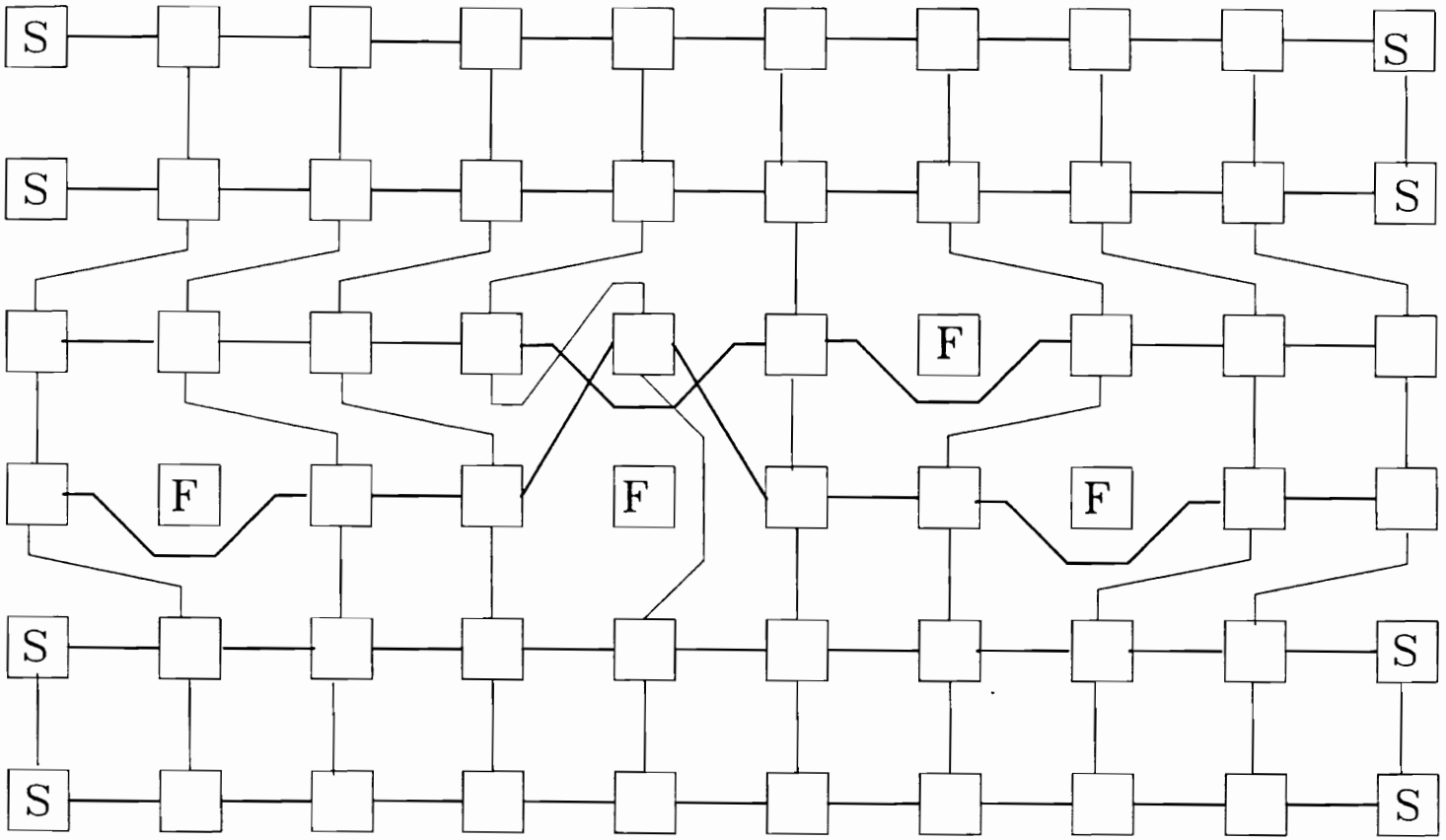


Figure 51. Four Faults Case 4-3d: Array with four faults in a 1-3 pattern.

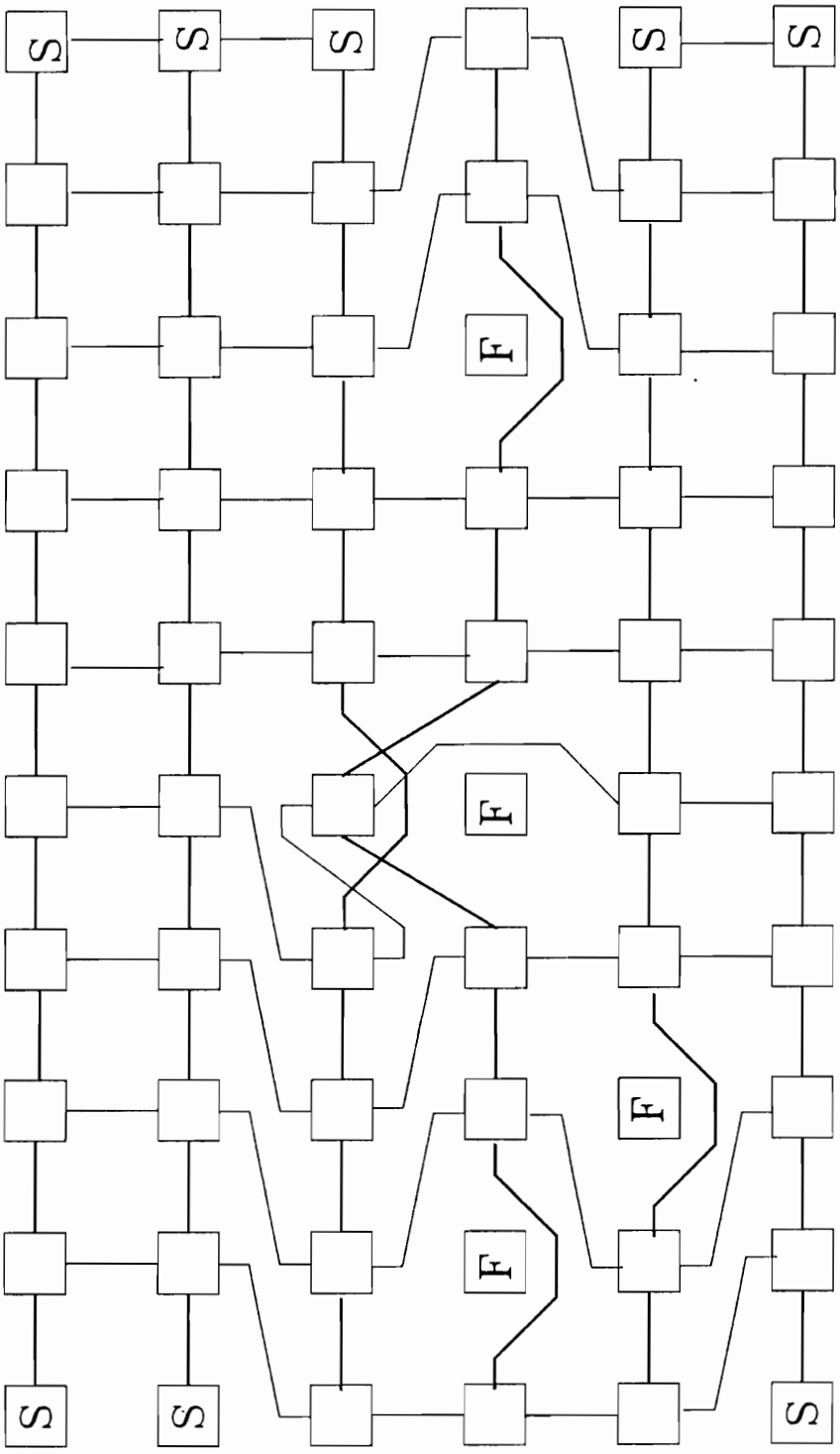


Figure 52. Four Faults: *Array with four faults in a 1-3 pattern.*

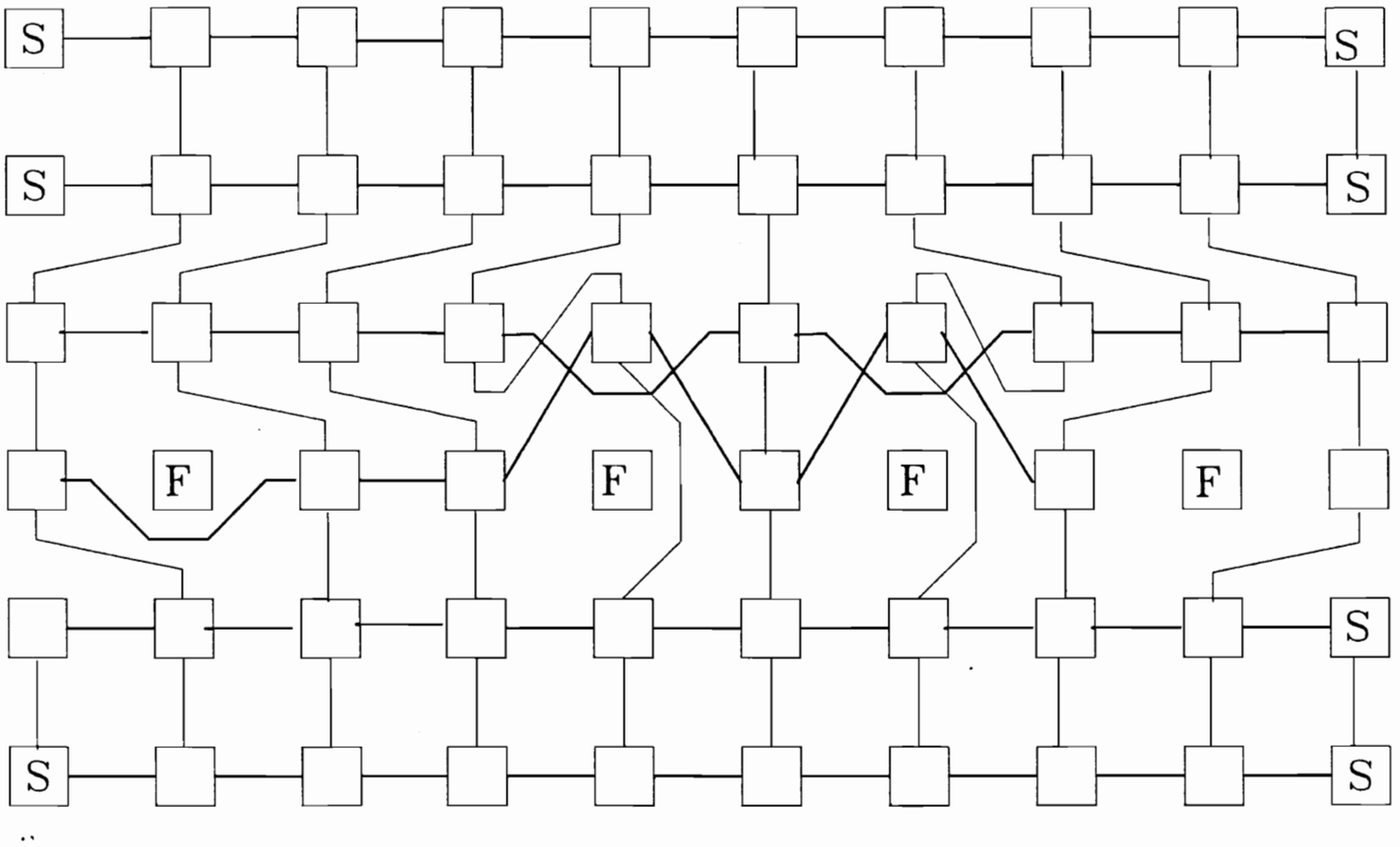


Figure 53. Four Faults Case 4-4: Array with four faults in one row.

5.0 Algorithm 4-12

Algorithm 4-12 is implemented using the twelve-neighbor interconnect scheme described in the previous algorithm. Figure 45 on page 128 shows the implementation of this neighborhood.

The cells which can be configured as cell[i,j]'s NORTH neighbor are:

- Cell[i-1,j]
- Cell[i-1,j-1]
- Cell[i-1,j + 1]
- Cell[i-2,j]

The possible SOUTH neighbors of cell[i,j] are:

- Cell[i + 1,j]
- Cell[i + 1,j-1]
- Cell[i + 1,j + 1]
- Cell[i + 2,j]

Cells which can be logically WEST neighbors are:

- Cell[i,j-1]
- Cell[i,j-2]
- Cell[i-1,j-1]
- Cell[i + 1,j-1]

Those that are designated as possible EAST neighbors are:

- Cell[i,j + 1]
- Cell[i,j + 2]
- Cell[i-1,j + 1]
- Cell[i + 1,j + 1]

As was the case in algorithm 2-12, logic must be added to the basic control circuitry to allow gating input/output data from the proper registers to the correct communications link.

5.1 Spares Complement

Two spare rows of cells, one on the northern boundary and one on the southern boundary, are included, in addition to the two spare columns of spare cells used in algorithm 2-10 and 2-12. Figure 55 on page 168 shows an 8 by 8 active array with two rows and two columns of spare cells.

5.2 Fault Register

There are 33 bits in the fault register in this algorithm. The twenty-four bits in algorithm 2-10 are used, plus nine additional bits to force row deformation, reverse the effects of northern row deformation and force southern deformation, and to negate the normal effects of multiple fault conditions when the faults are in adjacent cells. Only 25 of the bits are decoded.

ALGORITHM 4-12 FAULT REGISTER

Bit Position	Definition	Fault Condition
0	N	Cell[i-1,j]
1	NW	Cell[i-1,k]|k < j
2	NE	Cell[i-1,k]|k > j
3	S	Cell[i + 1,j]
4	SW	Cell[i + 1,k]|k < j
5	SE	Cell[i + 1,k]|k > j
6	E	Cell[i,j + 1]
7	FE	Cell[i,k]|k > j + 1
8	W	Cell[i,j-1]
9	FW	Cell[i,k]|k < j-1
10	NN	Cell[i-2,j]
11	NNW	Cell[i-2,k]|k < j
12	NNE	Cell[i-2,k]|k > j
13	MFNN	Two or more faults, row [i-2]
14	SS	Cell[i + 2,j]
15	SSW	Cell[i + 2,k]|k < j
16	SSE	Cell[i + 2,k]|k > j
17	MFSS	Two or more faults, row [i + 2]
18	MF	Two or more faults, row [i]
19	MFN	Two or more faults, row [i-1]
20	MFS	Two or more faults, row [i + 1]
21	RDN	Row deformation north
22	RDNE	Row deformation northeast
23	RDNW	Row deformation northwest
24	FCR	Fault in row [i] and west of a multiple fault in row [i + 1] or [i-1]
25	FCRN	Fault in row [i-1] which is west of a multiple fault in row [i-2] or [i]
26	FCRS	Fault in row[i + 1] which is west of a multiple fault in row [i] or [i + 2]
27	RDFS	Row deformation begins in this row A condition exists in row[i + 1] which requires row deformation
28	RDFT	Condition exists in this row which requires row deformation
29	RDFN	Condition exists in row[i-1] which requires row deformation
30	RDS	Force row deformation to the south instead of the north. This bit is set when a fault occurs north of a deformed row

31	RDE	Row deformation to the east. This bit is set whenever a fault exists in a row with a deformation to its east
32	RDW	Row deformation to the west.

5.3 Setting the Fault Register

With the exception of the following bits which were not used in algorithm 2-10, all bits are set exactly as the same bit in algorithm 2-10. Description of the logic controlling the common bits is found in the chapter dealing with that algorithm.

Bit 21 is used to indicate the presence of a row deformation in this column. A special set of bits called row reformation blockers is used to determine the set of cells which will set the deformation bits. If the blocker bits are zero, indicating row deformation to the north may take place, bit 21 is set by:

- [i + 1,j] and ([i + 1,j + 1] or [i + 1,j-1]) and not blocker
- S and SE and SW and not blocker
- N and NE and NW and blocker
- [i-1,j] and ([i-1,j + 1] or [i-1,j-1]) and blocker
- S and SW and RDE and not blocker
- N and NW and RDE and blocker
- S and SE and RDW and not blocker
- N and NE and RDW and blocker
- S and RDE and RDW and not blocker
- N and RDE and RDW and blocker

This bit is then propagated to the north or south in the same column. If adjacent cells are faulty bit 21 is set in both columns.

Bit 22 is used to indicate the presence of a row deformation beginning in column[j + 1]. This bit is used to trigger the usage of cell[i-1,j + 1] or cell[i + 1,j + 1] as the eastern neighbor instead of cell[i,j + 1]. It is set by:

- cell[i,j + 1] and cell[i,j + 2] are faulty
- E and FE and (W or FW)
- E and FE and RDW
- E and (W or FW) and RDE
- E and RDE and RDW
- input of bit 21 from the EAST neighbor

The last term in the above list is used to set bit 22 and bit 21 to disable the southeast link and enable the eastern link when both bits are set. Bit 22 is used in combination with bit 30 to force the deformation to the south instead of the north.

Bit 23 is used to indicate the presence of a row-deformation beginning in column[j-1]. It enables the usage of either cell[i-1,j-1] or cell[i + 1,j-1] as the western neighbor. It is set by:

- cell[i,j-1] and cell[i,j-2] are faulty
- W and FW and (E or FE)
- W and (E or FE) and RDW
- W and FW and RDE
- W and RDE and RDW
- input from the WEST neighbor contains bit 23 set

As was the case with bit 22, the combination of bit 21 and bit 23 set in a cell disables the southwest link and enables the western link. Again bit 30 is used to force the deformation to the south. The same propagation rule applies to bits 22 and 23 that was used on bit 21.

Bits 27, 28 and 29 are used to counteract the action of the multiple fault bits in certain instances. When more than one fault exists in a row, a bit is set indicating the existence of the multiple faults in row[i], row[i + 1] and row[i-1]. If this multiple fault consists of two or more adjacent faulty cells only, or if the faults at the eastern or western fault boundary consists of adjacent faults, the recon-

figuration process must use row-deformation. The existence of the multiple fault bits forces deformation of columns on each end of the fault pattern. Bits 27, 28 and 29 are set in the three rows which are affected by the row-deformation, row[i], and row[i-1] and row[i + 1]. The same conditions used to set bit 21 is used to set bit 27. This bit is propagated in each direction in row[i-1], assuming the multiple fault occurred in row[i]. Bit 28 uses the same conditions as bit 22 to set it. It is then propagated along row[i]. Bit 29 is set by north and northwest and northeast or by cell[i-1,j] and either cell[i-1,j-1] or cell[i-1,j + 1] being faulty. It is propagated along row[i + 1]. Bits 27, 28 and 29 are propagated in their respective rows only until they encounter another fault in row[i]. They are not propagated past a faulty cell.

Bit 30 is the bit used to force the row deformations to the south. A set of two bits called deformation blockers is generated whenever a fault occurs. These two bits are generated in the cell south of the fault and propagated to the south in the same column. When a condition exists requiring row-deformation, if the blocker bits are set in the cell or cells north of the fault requiring deformation, generation of the deformation bits is blocked to the north, and enabled to the south. With the blocker bits set, bit 30 is generated along with bits 21, 22 and 23. If a cell[m,n] has an input from its northern neighbor, cell[m-1,n], which includes a combination of bits 21, 22 and 23 and also includes bit 30, the same bits are set in the fault register of cell[m,n].

Bits 31 and 32 are set to indicate the presence of a row deformation to the east or the west of a particular cell. Bit 31 is set if the input from the western neighbor includes bit 23, row-deformation-north-west. This bit is propagated to the east until another row deformation is encountered or the boundary of the array is reached. Bit 32 is used to indicate a deformation to the east if the input from the eastern neighbor contains bit 22. This bit is propagated to the west until the array boundary or another deformation is reached.

5.4 *The Algorithm*

Algorithm 4-12 tolerates faults which were tolerated by algorithm 2-10 in exactly the same way as that algorithm. The main advantage of this algorithm is its ability to deform a row to use a spare cell in one of the spare rows whenever both spare cells in the row are already used. This is done by shifting the row up one cell at the point of the extra fault or faults. If a fault already exists to the north and in the column at which the deformation must take place, deformation is forced to the south. If a fault exists in the column both north and south of the deformation, a fatal condition exists.

If two or more faults lie to the east of a row deformation, in cell[i,j] and cell[i,j + n] | n > 0, the fault which lies nearest to the spare column of cells, cell[i,j + n], uses the column deformation reconfiguration process to include a spare cell in the same row into the active array. All other faults east of the row deformation must generate row deformations to use cells from the spare rows. A similar process is required for faults which lie to the west of a deformation. If a faulty cell lies between deformations, deformation of the row must be used.

In the case where only column deformations are required, the EAST/WEST neighbors are always east and west respectively, unless those cells are faulty. In a similar manner, in row deformations, the NORTH/SOUTH links are always set to north or south respectively unless that neighbor is faulty.

The following is a formal definition of the algorithm. Fault configurations are expressed in terms of the contents of the fault register, with the term "RD" being used in the switch statements to denote any of the row deformation bits. For example, "RDNE" meaning the row-deformation_northeast bit is set in the fault register may be used in the case statements, but in the switch statement, only "RD" appears.

```
                                        { /* begin algorithm 4-12 */
                                          /* begin WEST code  */
Switch(RD) /* any row deformation bit */
   {
Case (null) {
1.  if(W)                                                    connect far-west
2.  else                                                     connect west  }
    Case(RD){
3.  if((RDS and RDN and not RDNW)
                or(RDNW and not(RDS or RDN))                 connect north_west
4.  else if(not RDS and RDN and not RDNW)                    connect south-west
5.  else if((RDE and not RDW and W and not FW)
          or(RDW and not RDE and W and not(E or FE))         connect far-west
6.  else                                                     connect west}
     } /* end WEST code   */

                                        /* begin EAST code */
Switch(RD)  { /* and row-deformation bit */
 Case(null){
7.  if(E)                                                    connect far-east
8.  else                                                     connect east
     }
     case(RD){
9.  if((RDS and RDN and NOT RDNE)
                or (RDNE and NOT(RDN or RDS))                connect north-east
10. else if((RDS and RDNE and NOT RDN)
                or (RDN and NOT(RDNE or RDS))                connect south-east
11. else if((RDW and not RDE and E and not FE)
      or (RDE and not RDW and E and not(W or FW)))           connect far-east
12. else                                                     connect east
    }
    } /* end EAST code   */

    /* begin NORTH link code */

    switch(MF and MFN and FCR and FCRN and RD)
     {
    case (null){
13. if(
      (not(W or FW) and not(N or NW))
        or((W or FW) and  NW))                               connect north
14. else if(
      ((N or NW) and not(W or FW)))                          connect north-east
15. else if(
```

```
      ((W or FW) and not( NW)))                          connect north-west
    } /* end case  */
  case (FCR){
16. if(
      ((W or FW) and not(N or NW))
       )
                                                         connect north
17. else if(
      ((W or FW) and (N or NW))
      or(not(W or FW))
            )
                                                         connect north-east
18. else  /* not possible in this configuration */
                                                         connect north-west

    }
  case (FCRN){
19. if(
      (( NW) and not(W or FW))
            )
                                                         connect north

20. else if(
      (not( NW)) or(( NW) and (W or FW))
            )
                                                         connect north-west
21. else                                                 connect north-east
      }
  case (FCR and FCRN){
22. if(
      (not(W or FW) and not(N or NW))
      or((W or FW) and ( NW))
           )
                                                         connect north
23. else if(
       (not(W or FW) and (N or NW))
            )
                                                         connect north-east
24. else if(
      ((W or FW) and not( NW))
            )
                                                         connect north-west

    }
  case (MF){
25. if(
      ((W or FW) and ( E or FE) and not(N or NW))
      or(NW and not( E or FE))
           )
                                                         connect north
```

```
26. else if(
      (not(W or FW))
      or((W or FW) and ( E or FE) and (N or NW))
          )                                              connect north-east
27. else if(
      (not( E or FE) and not( NW))
          )                                              connect north-west
     }

    case (MF and FCRN){
28. if(
      ( NE)  or((W or FW) and ( E or FE))
          )                                              connect north
29. else if(
      (not(W or FW) and (N or NW))
          )                                              connect north-east
30. else if(
      (not( E or FE))
          )                                              connect north-west
     }

    case (MFN){
31. if(N and NW and NE)
                                                         connect far-north
32. else
33. if(
      (not(W or FW) and (NE and NW)
       or((W or FW) and (not( N or NE)))
        )                                                connect north
34. else if(
      (not( W or FW or NE))
      )                                                  connect north-east
35. else if(
      (not( NW))  or((W or FW) and ( N or NE) and ( NW))
          )                                              connect north-west
     }

    case (MFN and FCR){
36. if(N and NE and NW)
                                                         connect far-north
37. else if(
      (not(W or FW))   or(NE and NW)
      )                                                  connect north
38. else if(not( NE))
                                                         connect north-east
39. else if(
```

```
    ((W or FW) and not( NW))
        )                                                    connect north-west
     }

   case (MF and MFN){

40. if( (not(W or FW) and not(N or NW))
      or((W or FW) and ( E or FE) and (NE and NW)
      or(not( E or FE)   and not( N or NE))
          )                                                  connect north

41. else if(
      (not(W or FW) and (N or NW))
      or(( E or FE) and (W or FW) and not( NE))
          )                                                  connect north-east

42. else if(
      ((W or FW) and not( NW))
      or(not( E or FE) and ( N or NE))
          )                                                  connect north-west
     }
   case(RD){

43. if(RDS and N and not(RDE or RDW))
                                                             connect far-north

44. else if(RDE and not RDW and( not(E or NE)  or (E and NE)))
                                                             connect north

45. else if(RDE and not RDW and E and not NE)
                                                             connect north-east

46. else if(RDE and not RDW and NE and not E)
                                                             connect north-west

47. else if(RDW and not RDE and( not(W or NW)  or (W and NW)))
                                                             connect north

48. else if(RDW and not RDE and W and not NW)
                                                             connect north-west

49. else if(RDW and not RDE and NW and not W)
                                                             connect north-east

50. else if(RDW and RDE and not N)
                                                             connect north

51. else if(RDW and RDE and N)
                                                             connect far-north

52. else                                                     connect north
        }
   } /* end switch NORTH link code */

   switch(MF and MFS and FCR and FCRS and RD)
   { /* SOUTH link switch code */
   case (null)
     {
```

```
53. if(
      (not(W or FW) and not( S or SW))
      or((W or FW) and ( SW))
          )                                          connect south

54. else if(
      ((W or FW) and not( SW))
          )                                          connect south-west

55. else if(
      (not(W or FW) and ( S or SW))
          )                                          connect south-east
     }

    case (FCR)   {

56. if(
      ((W or FW)
         and (not( S or SW)))
      )                                              connect south


57. else if(
      ((W or FW) and (( S or SW)))
      or(not(W or FW))
      )                                              connect south-east

58. else                                             connect south-west
     }

    case (FCRS)  {

59. if(
      (not(W or FW) and  SW)
      )                                              connect south

60. else if(
      (not( SW))
      or(( SW) and (W or FW))
          )                                          connect south-west

61. else connect south-east  /* not possible this configuration */
     }

    case (FCR and FCRS)  {

62. if(
      (not(W or FW))
      or((W or FW) and  SW)
          )                                          connect south

63. else if(
      ((W or FW) and not( SW))
          )                                          connect south-west

64. else if(
      (not(W or FW) and( S or SW))
          )                                          connect south-east
```

```
        }
    case (MF)           {
65. if(
      ((W or FW) and ( E or FE) and not( S or SW))
            )                                          connect south
66. else if(
      (not( E or FE or SW))
          )                                            connect south-west
67. else if(
      (not(W or FW))
      or((W or FW) and ( E or FE) and( S or SW))
            )                                          connect south-east
      }
    case (MF and FCRS)     {
68. if(
      ( SE )
      or((W or FW) and ( E or FE))
          )                                            connect south
69. else if(
      (not( E or FE))
          )                                            connect south-west
70. else if(
      (( S or SW) and not(W or FW))
          )                                            connect south-east
       }
    case (MFS)          {
71. if(
       (S and SE and SW) or(RDFS and S)
        )
        south_link[cell_number] = 18;
72. else
73. if(
      ((SE and SW) and  not(W or FW))
          )                                            connect south
74. else if(
      (not( SW))  or(( S or SE ) and ( SW) and (W or FW))
          )                                            connect south-west
75. else if(
      (not( W or FW or SE ))
      )                                                connect south-east
       }
    case (MFS and FCR)    {
76. if(
```

```
      (SE and SW) or(not(W or FW))
          )                                              connect south

77.  else if(
      ((W or FW) and not( SW))
          )                                              connect south-west

78.  else if(
      (not( SE ))
          )                                              connect south-east
       }

     case (MF and MFS)
       {

79.  if(
       ((W or FW)
       and ( E or FE) and (SE and SW)
       or(not( E or FE) and not( S or SE ))
       or(not(W or FW) and not( S or SW))
       )                                                 connect south

80.  else if(
       ((W or FW)  and not( SW))
       or(not( E or FE) and ( S or SE ))
       )                                                 connect south-west

81.  else if(
       ((W or FW)and (E or FE) and not( SE ))
        or(not( E or FE) and ( S or SE ))
        or(not(W or FW) and( S or SW))
        )                                                connect south-east
        }
     case(RD) {

82.  if((RDN and S and not(RDE or RDW) or (RDFS and S))
                                                         connect far-south

83.  else if(RDE and not RDW and( not(E or SE)  or (E and SE)))
                                                         connect south

84.  else if(RDE and not RDW and E and not SE)
                                                         connect south-east

85.  else if(RDE and not RDW and SE and not E)
                                                         connect south-west

86.  else if(RDW and not RDE and( not(W or SW)  or (W and SW)))
                                                         connect south

87.  else if(RDW and not RDE and W and not SW)
                                                         connect south-west

88.  else if(RDW and not RDE and SW and not W)
                                                         connect south-east

89.  else if(RDW and RDE and not S)
                                                         connect south
```

```
90.  else if(RDW and RDE and N)                    connect far-south
91.  else                                          connect south
        }
     } /* end SOUTH link switch code */
     } /* end reconfigure */
```

5.5 Fault Coverage

5.5.1 Single Faults

Theorem 3-1: An array equipped with logic to use algorithm 4-12 tolerates all single faults.

Proof: This algorithm provides coverage for single faults in exactly the same way as algorithm 2-10. The proof can be found in the chapter describing that algorithm.

5.5.2 Double Faults

Theorem 3-2: An array which can use algorithm 4-12 can tolerate all occurrences of double faults.

Proof: Double faults can exist as

1. single faults in a 1-1 occurrence

2. two faults in one row, non-adjacent cells

3. two faults in adjacent cells

Case 2-1 and 2-2 are configured exactly as the same cases in algorithm 2-10. Case 2-3 is an occurrence of two faults which was not tolerated by algorithm 2-10. Figure 56 on page 169 shows the array with two faults in adjacent cells at time t = 1, after the first reconfiguration cycle. At this time, assuming the faults exist in cell[i,j] and cell[i,j + 1], the neighbors of these two cells detect the faults and begin the reconfiguration process. Initially, reconfiguration begins with each cell detecting only a single fault, as in case 1.

At time t = 2, the six neighbors of the two cells will all detect the existence of the adjacent faults and begin the row deformation process. Figure 57 on page 170 shows that cell[i,j-1] detects both its eastern neighbors are faulty and configures north, south, north-east and west. This cell has algorithm configuration numbers 9-80-12-6. Cell[i,j + 2], the cell which is immediately east of the double fault configures to the north, south,east and northwest because it has a fault neighborhood which places it in configuration number 38-78-12-3 Cells to the east of the double fault configure to the east and those to the west of the double fault receive information about the multiple faults at the same time they set their RDF bits and continue to connect north and south. Cell[i + 1,j] and cell[i + 1,j + 1] have the same configuration with the RDFN bit set and a fault to the north. These cells connect to the farnorth, south-east and west, with algorithm configuration number 36-47-12-6. Cell[i-1,j] which is directly north of the western fault, sets its RDN bit and the RDNW bit. This combination is used to enable links to the southwest and east, rather than the southwest and southeast. This cell is in configuration 52-82-12-4. Cell[i-1,j + 1] sets its RDN and RDNW bit, which sets its communications links to the west and southeast. It has configurations 52-82-10-6. The cycle is completed when cell[i-1,j] and cell[i-1,j + 1] assume the states previously held by their southern neighbors, the faulty cells. These states still are available in the fault-free cells state registers because the reconfiguration process halted the next-state algorithm execution.

Figure 57 shows that after cycle 2, knowledge of the adjacent fault condition has been transmitted to the north to cell[i-2,j], cell[i-2,j + 1], cell[i-1,j-1] and cell[i-1,j + 2]. In addition the appropriate "RDF" bits have been generated in the cell[i-1,j + 2], cell[i-1,j-1], cell[i + 1,j-1] and cell[i + 1,j + 2]. These bits now begin to reverse the configuration process to the right. Cell[i + 1,j + 2] enters con-

figuration number 52-91-12-6 and connects north-south-east and west. Cell[i + 1,j + 2] sets the RDFN bit in this cycle and connects to the north,south,east and west. Cells[i-1,j-1] and [i + 1,j-1] follow a similar process and return to their normal north/south connections. At this time, cell[i-2,j] and cell[i-2,j + 1] take on the states previous ly held by cell[i-1,j] and cell[i-1,j + 1] respectively.

Reconfiguration continues in this manner until the final configuration is reached. Figure 58 on page 171 shows the final configuration. Cells [1,j] and [1,j + 1] were spare cells which have been included in the active array. The RDF-x bits, which are propagated east and west from the fault area until another fault is encountered in the same row, have returned the configuration of the cells in rows[i-1], [i] and [i + 1] not involved in the row deformation to normal north-south links. All cells with an RDF_x bit set have north-south configurations. If another fault occurs in a row with adjacent faults, a RDE or RDW bit will be set in the row, overriding the RDF_x bits.

We have shown that an array equipped with hardware to support algorithm 4-12 will tolerate all two-fault conditions.

■

5.5.3 Three Faults.

Theorem 3-2: An array with hardware to implement algorithm 4-12 can tolerate all three fault occurrences except those which occur in a cluster of the type cell[i,j]-cell[i,j + 1]-cell[i + 1,j].

Proof: Three faults may occur in the following configurations:

1. 1-1-1
2. 1-2
    a. 1-2, two not adjacent
    b. 1-2, two in adjacent cells
3. 3

a. all three adjacent

b. two adjacent, one non-adjacent

c. none in adjacent cells.

Case 3-1 is identical to case 3-1 in algorithm 2-10. Case 3-2a is identical to case 3-2a and 3-2b in algorithm 2-10. Case 3-2b, with two of the faults in adjacent cells, can be tolerated if the adjacent cells are cell[i,j] and cell[i,j + 1] and the single fault is not in either cell[i-1,j] or cell[i-1,j + 1] or cell[i + 1,j] or cell[i + 1,j + 1]. Reconfiguration will be successfull if the single fault is in any other cell in the array. Figure 59 on page 172 shows one of the configurations which cannot be tolerated. Figure 60 on page 173 shows one configuration for case 3-2b which can be tolerated. In this fault configuration, the single fault is north of the double fault located at least two rows north and in one of the columns occupied by the double fault. Reconfiguration begins around each fault identically to case 1 of algorithm 2-10 for the single fault and case 2-3 described earlier in this algorithm for the double fault. Assuming the faults occur in cell[i,j] and cell[i + 4,j] and cell[i + 4,j + 1], the first step in the configuration process will be column deformation around the single fault at t = 1, followed by row deformation to row[i + 3] north of the double fault and column deformation to cell[i,j + 2] at time t = 2. At this time, cell[i + 1,j] which is directly south of the single fault, generates one of the auxiliary bits called row-deformation-blocker. This bit will be propagated to the south in the same column until it encounters another faulty cell.

At some point in time, the deformation blocker bit will reach cell[i + 3,j] north of the double faults. This bit is then propagated along row[i + 3] in both directions until it encounters a cell in row[i + 4] which is not faulty. The effect of this bit is to clear the row-deformation bits 21-23 in row[i + 3] north of the double fault. When the bit is propagated to cell[i + 4,j-1] and cell[i + 4,j + 2], they cause the generation of bit[30], RDS. This bit is also generated in cells [i + 5,j] and [i + 5,j + 1] when the blocker bit reaches them, as well as the same row-deformation bits which did exist in cells[i + 3,j] and [i + 3,j + 1].

When the deformation bits were cleared from the cells north of the double fault, the source for this data was removed from the input to cells in row[i + 2]. The deformation bits are then cleared in this row. The deformation bits are then cleared from all cells north of the double fault in the same manner in which they were originally generated. The elimination of these bits propagates to the north one row each clock cycle. At the same time, the combination of deformation bits combined with the RDFS bit is propagated to the south, causing the row deformation to proceed to the south until spares are encountered.

The single fault is reconfigured identically to the process described in case 1 of algorithm 2-10. Cell[i + 3,j], with RDFS set, a fault to the south and no deformation bits set, is in configuration 52-82-12-6 and connects north, farsouth, east and west. Cell[i + 3,j + 1] has an identical configuration in this case. Cell[i + n,j-1] | n > = 4 forms the western edge of the row deformation. This cell is in configuration 52-91-10-6 Cell[i + 5,j] connects farnorth, south, east and northwest because its fault pattern places it in configuration 43-91-12-3. Cell[i + 5,j + 1] connects to the farnorth, south, northeast and west. It has configuration 43-91-8-6. Cell[i + n,j] |n > 5 is the list of cells south of the fault cell in column j. These cells connect north-south-east and northwest because the contents of their fault registers place them in configuration 52-91-12-3. Cell[i + n,j + 1] |n > 5 has configuration 52-91-9-6 and connects to the north, south, west and northeast. Cell[i + n,j + 2] |n > = 4 forms the eastern edge of the deformation to the south. These cells connect to the north, south, east and southwest because their configurations are 52-91-12-4.

Figure 61 on page 174 shows another tolerated configuration for case 3-2. In this case, the single fault in cell[i,j] is west of the double fault in cell[i + n,j + m] and cell[i + n,j + m + 1] where m and n are positive integers. The configuration around these faults, assuming they occur simultaneously, proceeds initially as if both fault patterns were independent. The process around the single fault proceeds as described in case 1, and around the double fault as described earlier in this chapter. When the row deformation reaches row[i + 1], assuming the column deformation around the single fault has already passed this point, the setting of the row deformation bits in cell[i + 1,j + m-1], which is the western boundary of the row deformation, generates the row-deformation-east bit be-

cause the northwest bit is also set in this cell. This bit will then be propagated to the west along row[i + 1]. Propagation of the northwest bit to the east in row[i + 1] is stopped by the row deformation bits. Information about faults on either side of a row deformation is not propagated across the row deformation.

In the next communications cycle, cell[i,j + m] and cell[i,j + m + 1] will set bit 21 because they receive bit 22 from a southern cell on the southwest link. This bit then generates the RDE bit in this row and blocks propagation of the farwest bit to the east in row[i]. Bit 22 is generated in cell[i-1,j + m-1] in the next cycle because it receives a bit 21 from its southeast neighbor. This blocks propagation of the southwest fault bit in row[i-1] and allows all cells to the east of the row deformation to return to null fault register contents. Those columns involved in the row deformation can then continue with the deformation process. The deformation will continue until the spare cells in row 1 are incorporated in the pattern.

RDE bits have now been generated in row[i-1], row[i] and row[i + 1]. As these bits are propagated to the west, cells to the east of the single fault will return to normal north-south-east-west communications patterns. Cell[i-1,ℓ] | j < ℓ < j + m-1 has algorithm configurations 52-91-12-6 Cell[i,ℓ] | j + 1 < ℓ < j + m-1 connects to the north-south-east and west because it has the RDE bit set and no faults to the east or southeast or north or northeast. Likewise, cell[i + 1,ℓ] | j < ℓ < j + m-1 has a similar configuration.

Cells west of the single fault will change communications links to configure around the fault to the west. Cell[i-1,ℓ] | ℓ < = j connects N-SW-E-W because it falls into configuration 52-85-12-6. Cell[i + 1,ℓ] | ℓ < = j has a configuration of 46-91-12-6. Cell[i,j-1] connects NE-SE-FE-W, with a configuration of 45-84-11-6. Cell[i,ℓ] |ℓ < j-1 has the same configuration except for the eastern configuration which is 12. It connects NE-SE-E-W

Figure 62 on page 175 shows another configuration of case 3-2 in which the single fault is in row[i + 1] and the double fault in row[i]. If the single fault had occurred south of row[i + 1] the two

fault patterns would configure independently. For discussion, assume the faults occur in [i,j], [i,j + 1] and cell[i + 1,ℓ]. With the FCRN bit in row[i + 1] and no faults to the northwest, the critical fault bits are set in row[i], row[i + 1] and row[i + 2]. Cell[i,m] | m < = ℓ now changes only its southern link, connecting to the southeast with configuration number 82. Cell[i + 1,m] | m < ℓ connects NE and SE with configurations 46-83. Cell[i + 2,n] | n < = ℓ connects to the northwest, with northern configuration of 17 with the FCRN bit set and no fault to the northwest.

If the single fault occurs two or more rows to the south of the double fault, they will be treated as independents. If it occurs to the northwest of the double fault, the configuration will be opposite to that in the example with the fault to the west of the deformation. The single will be bypassed by configuring to the east as if the double deformation did not exist.

Figure 63 on page 176 shows a configuration around three faults in case 3-2 in which the single fault lies in the column used as one of the boundary cells in the row deformation. In this case, with faults in cell[i,j], cell[i + 2,j-2] and cell[i + 2,j-1], cell[i-1,j] generates the RDN bit because it has RDNW set as the only deformation bit and a fault to the south. RDNW will continue to be set because it is received from cell[i + 1,j]. Cell[i-1,j-1], which had RDN and RDNW set, will set RDNE. With RDN, RDNW and RDNE, cell[i-n,j-1] | n < = i-1 will configure N-S-E-W instead of N-SW-E-W which it had before cell[i,j] became faulty. When cell[i,j + 1] connects to cell[i-1,j] because its fault register decoding logic declares that cell to be its northern neighbor, it will receive a bit 23, RDNW. Because it has a faulty cell to its west, it will set the RDNW bit in its fault register and will not transmit the fault information about cell[i,j] to its neighbors. This will clear the information from cells to the east and they will return to null fault registers and connect N-S-E-W. RDNW will then be propagated northward in column[j + 1] and complete the configuration shown.

Figure 64 on page 177 and Figure 65 on page 178 show occurrences of case 3-3, in which all three faults are in one row. In these configurations the adjacent faults will configure exactly as in case 2-3 with row deformation. Because the RDFS, RDFN and RDFT bits are only propagated until another fault is reached in the row with the adjacent faults, the single faults on either end will have

the same configuration as in case 2-2 of algorithm 2-10, with two faults in one row. In Figure 64 on page 177, the cells to the west of the single fault know only of a multiple fault and no faults to the west. In Figure 65 on page 178 the same is true of cells to the east of the single fault. If the faults occur in cell[i,j], cell[i,j + 1] and celli,k], the RDF_x bits are only propagated to column[k-1].

Figure 66 on page 179 shows the configuration around three faults in one row with no faults in adjacent cells. In this configuration, with the faults in cell[i,j], cell[i,k] and cell[i,ℓ], cell[i-1,k] has faults to the south, southeast and southwest, causing it to set bit 21, RDN, of the fault register. Cell[i,k-1] sets the RDNE bit and cell[i,k + 1] sets the RDNW bit. Since the RDS bit is not set, these bits are propagated to the north.

Cell[i-1,k-1] $|\ell <= i$ has an eastern configuration of 9 because it has "RDNE" set with no other deformation bits set. These cells connect to the northeast. Cell[i-1,k + 1] $|\ell <= i$ has the "RDNW" bit set and connects to the northwest, with western configuration number 3. Cell[i-1,k] has configuration 52-82-10-4. This cell connects to N,SS,SE,SW. Cell[i-1,k] $|\ell < i-1$ connects to N,S,SE and SW because its southern algorithm reference number is 91.

Figure 67 on page 180 shows case 3-3a in which all three faults are adjacent. The faults occur in cell[i,j], cell[i,j + 1] and cell[i,j + 2]. This configuration is identical to case 2-3 with two faults in adjacent cells. Cells in column[j-1] have the same configuration as cells in column[j-1] in case 2-3. Cells in column[j + 3] are identical to cells in column[j + 2] in case 2-3. Only cells in column[j + 1] are unique to this configuration. These cells have the RDN, RDNW and RDNE bits set in their fault registers, which changes their algorithm configuration numbers for the east and west links. These cells connect to the east and west with algorithm configurations of 12-6.

It has been shown that an array equipped to implement algorithm 4-12 will tolerate a large percentage of three faults.

■

5.6 *Four Faults.*

Theorem 3-4: An array equipped with hardware to implement algorithm 4-12 will tolerate all occurrences of four faults except:

- Fault patterns which contain the three fault pattern which could not be tolerated.
- Patterns in which two adjacent cells in one row are faulty with a single fault to the north in either of the columns in which the double fault occurs and a single fault to the south in either of the columns.

Proof: Four faults can occur in the following configurations.

1. 1-1-1-1

2. 1-1-2

3. 2-2

4. 1-3

5. 4

Case 4-1 is identical to case 4-1 in algorithm 2-10. All occurrences can be tolerated.

Case 4-2, with two single faults and one double fault, which do not involve adjacent faults is identical to case 4-2 in algorithm 2-10. Figure 68 on page 181 shows one instance of this case which cannot be reconfigured. In this example, row deformation is blocked both to the north and the south.

Analysis of this fault pattern is made easier if the array is divided into three sections, northeast of the double fault, northwest of the double fault and south of the double fault. If the single faults fall in two of these three sections, the faults are independent and will configure identically to case 3-2 described earlier.

Figure 69 on page 182 shows a configuration of case 4-2 which is tolerated. The faults occur in cell[i,j], cell[i + n,k], cell[i + n,k + 1] and cell[i + m,ℓ]. The double fault in row[i + n] and the single fault in cell[i,j] have configurations identical to case 3-2b, Figure 61 on page 174. The fault in cell[i + m,ℓ], with m > i + n + 2, has a configuration which is independent of any of the other faults, and configures as a single fault occurrence, case 4-1.

Figure 70 on page 183 shows four faults in case 4-2 in which the double fault has a single fault to its north. This configuration is identical to case 3-2b in Figure 60 on page 173. The double fault will deform the rows to the south. When the row deformation bits are removed from the cells north of the double fault, the single faults which are north of the double fault will configure identically to two single faults in case 2-1.

Figure 71 on page 184 shows a configuration of case 4-2 in which the row deformation to the north is blocked. The double faults occur in cell[i,j] and cell[i,j + 1]. Cell[1,j-1] through Cell[1,j + 2] | 1 > = i configure identically to case 3-2c. The single fault which is west of the row deformation in cell[i + 2,j-2] would configure identically to the center fault in case 3-3c.

5.6.1.1 Case 4-3

Figure 72 on page 185 shows a fault pattern which is case 4-3, or two faults in each of two rows. If none of the faults are in adjacent cells, the reconfiguration is identical to algorithm 2-10, case 4-3. In this example, cell[i,j] and cell[i,j + 1] are faulty, forcing a row deformation to the north. Cell[i-m,k] and cell[i-m,ℓ] are faulty to the northeast. Cell[i-m,ℓ] has no faults to its east, so it will con-

figure as a normal two fault configuration, changing communications links to the east. Cell[i-m,k] has a fault to its east, so it must reconfigure by deforming the row to the north.

Figure 73 on page 186 shows an instance in which the double fault in row[i] has one faulty cell on either side of the row deformation. In this case, cell[i,j] to the west of the deformation has the same configuration as the single cell in case 3-2b of this algorithm and configures to the west. The remaining fault in row[i] is configured to the east, a mirror image of cell[i,j]. This is identical to case 3-2c of this algorithm.

Figure 74 on page 187 shows an example of case 4-3 in which both occurrences of two faults are adjacent cell faults. Reconfiguration around each of the pairs of faults is identical, with row deformation required by each. This is two instances of case 2-3.

5.6.1.2 Case 4-4 Three faults in one row.

Figure 75 on page 188 shows an instance of case 4-4 in which the single fault is east of the center fault. In this case, the three faults in one row configure identically to case 3-3c, in Figure 66 on page 179. The single fault is then configured as a normal single fault occurrence. Figure 76 on page 189 shows case 4-4 with the single fault in cell[i-n,j] north of the center fault cell[i,j]. This configuration proceeds as a row deformation until the deformation reaches cell[i-n + 1,j]. At this time, the blocker bits are generated, and are propagated to the south until they reach cell[i-1,j]. The northeast blocker bit is then generated in cell[i,j-1] and the northwest blocker generated in cell[i,j + 1]. Both bits are set in cell[i + 1,j]. This combination causes the RDN, RDNE and RDNW bits to be cleared north of row[i] by clearing the RDN bit in cell[i-1,j], and setting the RDS bit in cell[i,j-1], cell[i,j + 1] and cell[i + 1,j]. These RDN_x bits are propagated to the north only if the RDS bit is not set. With no input from the south to continue transmitting the bits, they will be cleared to the north one logical row per clock cycle. The combination of these four bits is propagated to the south from row[i]. Cell[i + 1,j-1] | 1> = 0 then has a eastern configuration of 10 and connects to the southeast.

Cell[i+1,j+1] | 1> = 0 has a western configuration pattern of 4 and connects to the southwest. Cell[i+1,j] | 1> 0 has a western configuration of 3 and connects to the northwest, and an eastern configuration of 9 and connects to the northeast. Cell[i+1,j] has a northern configuration of 31 and connects to cell[i-1,j]. Cell[i-1,j] has a southern configuration of 82.

Figure 77 on page 190 shows the last example of three faults in one row to be discussed. All three faults are in adjacent cells in columns [i], [i+1] and [i+2]. If the single fault is in a more than one row north of the triple fault, and in any column other than these three columns, they will be treated as described in previous examples, with the triple fault configured identically to case 3-1 and the single faults configuration process dependent upon its location relative to the row deformation. If the single fault occurs north of the triple and in one of the same columns, the blocker bits will be generated in the cell to the south of the single fault and propagated to the south, forcing the five columns involved in the row deformation to deform to the south as described in the previous case.

5.6.1.3 Case 4-5

Figure 78 on page 191 shows only one configuration for four faults in one row. If two of the faults are adjacent, they will deform to the north as in case 2-3. If three are adjacent, they will deform as in case 3-3a. If none of the faults are adjacent, the two interior faults will generate row deformations identical to case 3-3c..

It has been shown that an array equipped to use algorithm 4-12 will tolerate a large percentage of four-fault occurrences.

■

5.6.2 Five Faults.

Theorem 3-5: A cellular array which has hardware to implement algorithm 4-12 will tolerate all occurrences of 5 faults except those containing fault patterns listed in the 3 and 4 fault theorems and an occurrence of three faults in one row with none of the faults in adjacent cells and faults to the north and south in the column containing the center fault of the three faults.

Proof: Five faults can occur as:

1. 1-1-1-1-1
2. 1-1-1-2
3. 1-2-2
4. 1-1-3
5. 2-3
6. 1-4
7. 5

Case 5-1 and case 5-2 are identical to previous cases in three and four fault descriptions. Case 5-3 contains one pattern which has not been described before. Figure 79 on page 192 shows a fault pattern not previously discussed, that of reconfiguration around a fault or around faults which lie between two row deformations. Both instances of double faults configure identically to case 2-3 with row deformation. The single fault in cell[i,j] has row deformations to its east and west. This combination sets the RDN bit in cell[i-1,j], the RDNE bit in cell[i,j-1] and the RDNW bit in cell[i,j + 1]. Since bit 30, RDS is not set, this row will be deformed to the north.

Many other configurations of five or more faults can be tolerated. However, all algorithm cases have been described, and analysis of previous algorithms has shown six or more faults have insignificant impact on the coverage data.

5.7 *Coverage Analysis*

This section gives coverage analysis for an 8 by 8 active array, with two spare rows and two spare columns, or a ten by ten array. This analysis is intended to be conservative. When counting configurations which cannot be tolerated, very little effort was made to eliminate double counting. If there was a question about whether or not a configuration had been counted before, it was counted again. It is assumed that the ten by ten array is only a part of a larger array, and that other cells will be available to assume the state functions of the spare cells should they become faulty. The target array for this architecture has unbounded size, so it was assumed that the spare cells are needed for the state pattern algorithm.

An array equipped with hardware to support algorithm 4-12 will tolerate all single and double faults. All three faults will be successfully reconfigured except those which contain a pattern of two faults adjacent in one row with one adjacent neighbor in one of the columns also being faulty. There are nine choices for getting two faults in adjacent cells in one row. In all rows there are four choices for the third fault. This gives

$$10 \times 9 \times 4 = 360$$

ways in which three faults can be catastrophic ignoring boundary conditions. Four faults which cannot be tolerated can occur as a three fault pattern which cannot be tolerated with the fourth fault in any other cell in the array, or as an adjacent-cell two-fault pattern with a faulty cell in either of the same columns to the north and to the south. Configurations in which the adjacent cell in the column is faulty was counted in three faults. If the adjacent faults are in row 0 or row 9, there are

$$2 \times 9 \times 16 \times 95$$

patterns which cannot be tolerated. This represents 2 choices for the row, with 9 choices for the adjacent faults, 16 possibilities for the third fault, which is in one of the columns containing the

adjacent faults, but not in the adjacent row, and 95 choices for the fourth fault, which can be in any cell except those which are immediately north or south of the adjacent faults. Faults in these boundary rows are special cases because only one other fault in the columns is catastrophic. If the adjacent faults are in row 1 or row 8, there are 2×9×2×14 possibilities, 2 rows, 9 choices for adjacent faults, 2 choices for the third fault and 14 choices for the fourth fault. In the following numbers for rows 2/7, 3/6 and 4/5, the numbers represent 2 choices for the row, 9 choices for the adjacent faults, and the choices for the third and fourth fault respectively. These figures are:

$$2\times9\times4\times12 + 2\times9\times6\times10 + 2\times9\times8\times8.$$

This gives a total of 30396 adjacent-fault patterns of four faults which cannot be tolerated. The number of four fault occurrences which cannot be tolerated is

$$360\times97 + 30396 = 65026.$$

Five faults can be catastrophic if they include the fatal patterns with four faults, plus a triple fault which is non-adjacent with faults to the north and south in the same column as the center fault. This blocks row deformation to the north and the south. An estimate of these configurations gives 3.10×10^6 five fault configurations which cannot be tolerated.

Because of the difficulty in being certain that all fatal configurations have been counted in the larger fault configurations, and our desire to be conservative in the reliability figures, estimates of greater fault tolerances will not be made. This will cause the survivability figures given later to be low, but accurate survivability calculations are only possible if all 40 patterns are counted. The figures given for survivability will be somewhat lower than the actual figures.

Table 5 on page 167 shows the results of coverage analysis for an array using this reconfiguration algorithm. Figure 54 on page 167 shows the calculations of survivability equations for algorithm 4-12, and Figure 80 on page 193 is a plot comparing the data for an 8 by 8 active array with and without the algorithm installed.

The MTBF for this basic 8 by 8 array is $\int_0^\infty e^{-64(.0001)t} = 156$. The basic array has a probability of surviving that long of approximately 37 percent. The array augmented with 36 spare cells has a survival probability of 99 percent for that same period.

5.8 *Cost of Implementation*

The number of additional bits which must be transmitted by each cell to its logical neighbors to enable implementation of this algorithm is 41, eight single latch bits and 33 fault register bits. Logic estimates, again based on the simulation code, indicates the need of 139 'and' gates, 70 inverters and 159 'or' gates in the decoder, and 430 'and' gates, 252 'or' gates and 47 inverters in the fault register logic. Total new logic is approximately 1100 gates.

5.9 *Time Complexity*

In an 'm' row by 'n' column array, the worst-case reconfiguration would be the existence of two faults which were not adjacent on one side of the array, with a fault in the northern most row in the same column as the western fault. If a single fault occurred in the same row on the opposite boundary of the array, it would take approximately 'm' clock cycles for the information about the new fault to propagate sufficiently to begin the row deformation. Another 'n' cycles might be required to reach the point where the knowledge that the northern deformation is blocked and another 'n' cycles to deform to the south. Total time in a reconfiguration is m + 2n.

Table 5. Coverage Analysis Results

Number of Faults	Possibilities	Covered	Percentage
1	100	100	100
2	4950	4950	100
3	161,700	161340	99.7
4	3.92×10^6	3.86×10^6	98.4
5	7.53×10^7	7.24×10^7	95.9

64 Cell Active Array
2 Spare Columns 2 Spare Rows
lambda = 0.0001

t	W/O SPARES	W/SPARES	t	W/O SPARES	W/SPARES
0	1	1	1	0.938005	0.999999
20	0.879853	0.999994	30	0.825307	0.999981
40	0.774142	0.999954	50	0.726149	0.999905
60	0.681131	0.999826	70	0.638905	0.999703
80	0.599296	0.999518	90	0.562142	0.999252
100	0.527292	0.998879	110	0.494603	0.99837
120	0.46394	0.997694	130	0.435178	0.996814
140	0.408199	0.995694	150	0.382893	0.994294
160	0.359155	0.992575	170	0.33689	0.990496
180	0.316004	0.988017	190	0.296413	0.985101
200	0.278037	0.981711	210	0.2608	0.977814
220	0.244632	0.973379	230	0.229466	0.968379
240	0.21524	0.962791	250	0.201897	0.956595

Figure 54. Algorithm 4-12 Survivability Probability.

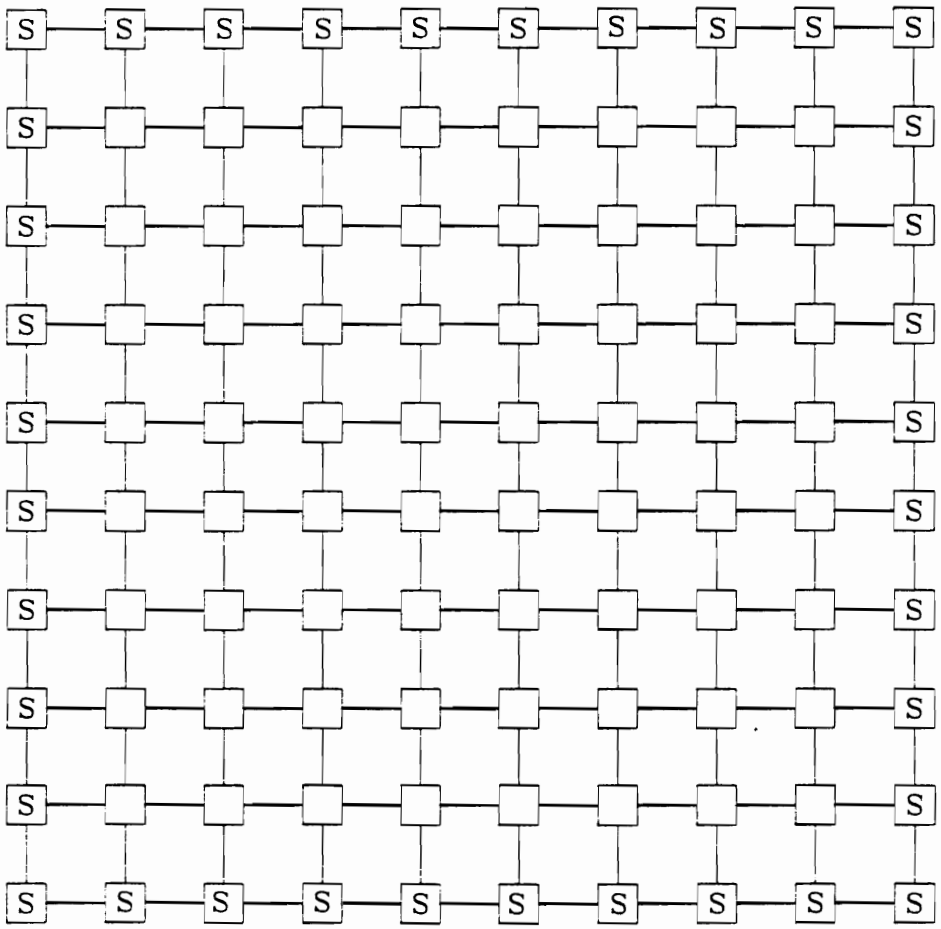


Figure 55. Ten by Ten array: Array with two spare rows and two spare columns.

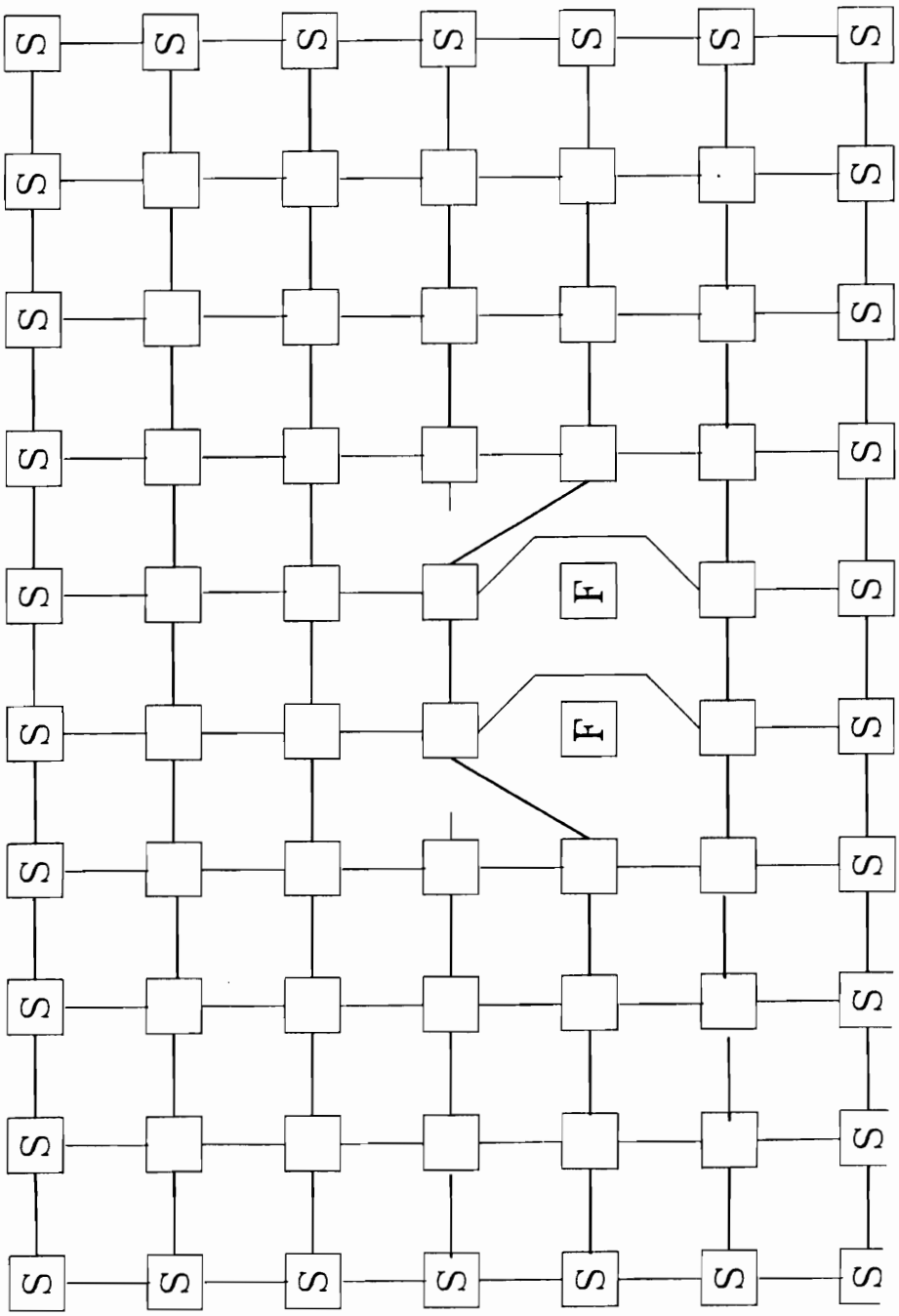


Figure 56. Two Faults Case 2-3: Array with two faults in adjacent cells, t = 1.

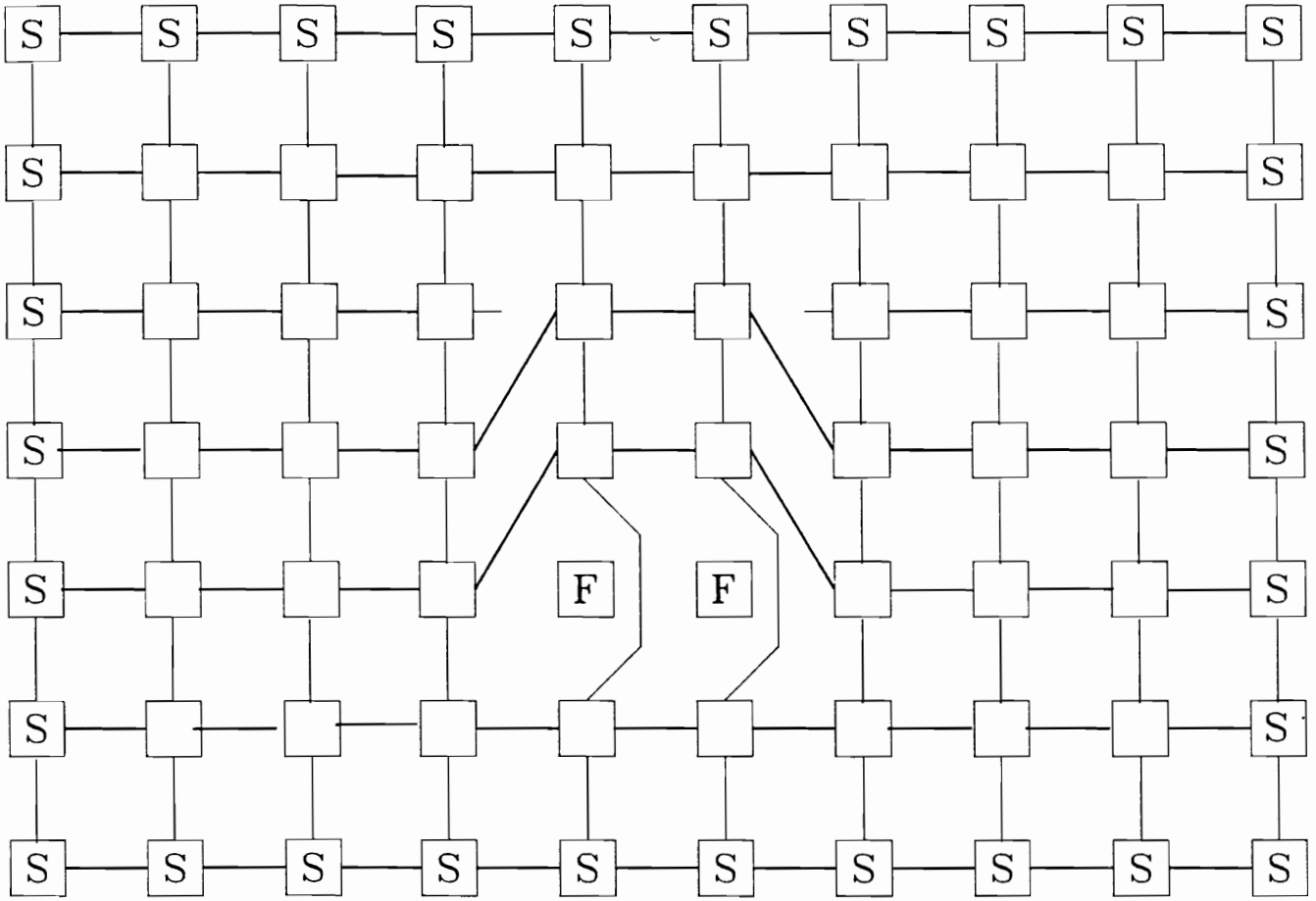


Figure 57. Two Faults Case 2-3: Array with two faults in adjacent cells, t = 2.

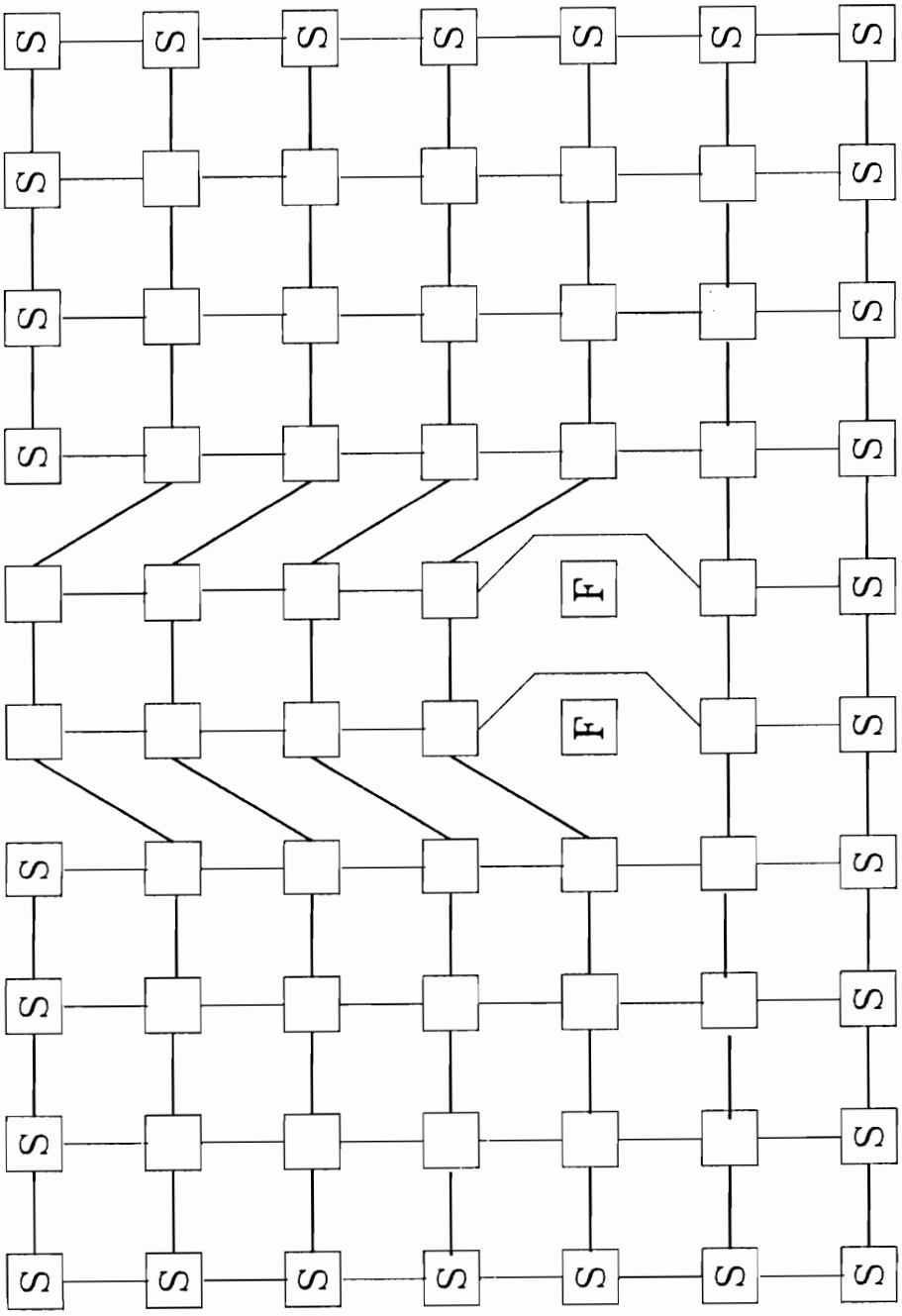


Figure 58. Two Faults Case 2-3: Array with two faults in adjacent cells, t = final.

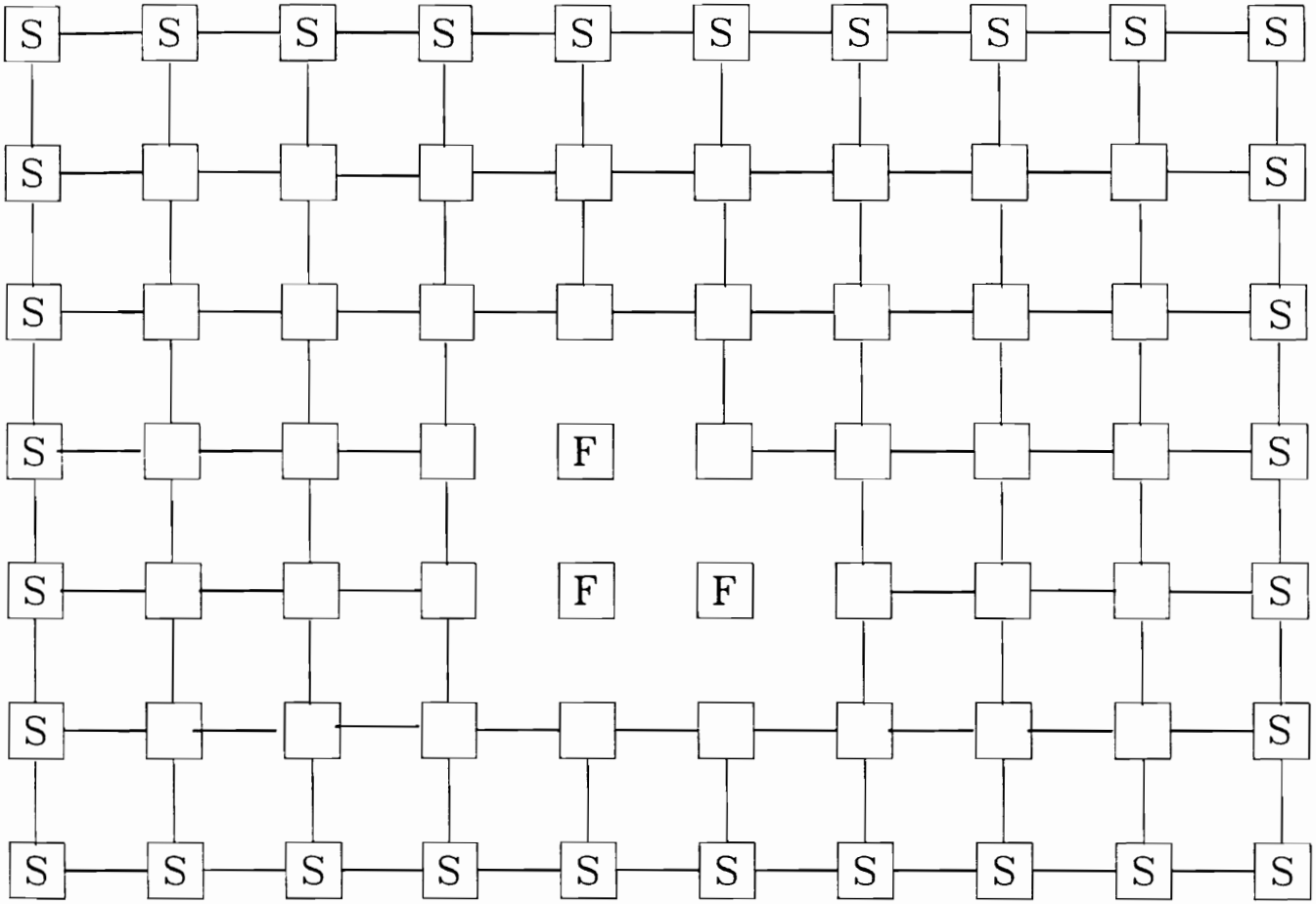


Figure 59. Three Faults Case 3-2b: Array with two faults in adjacent cells, one single fault.

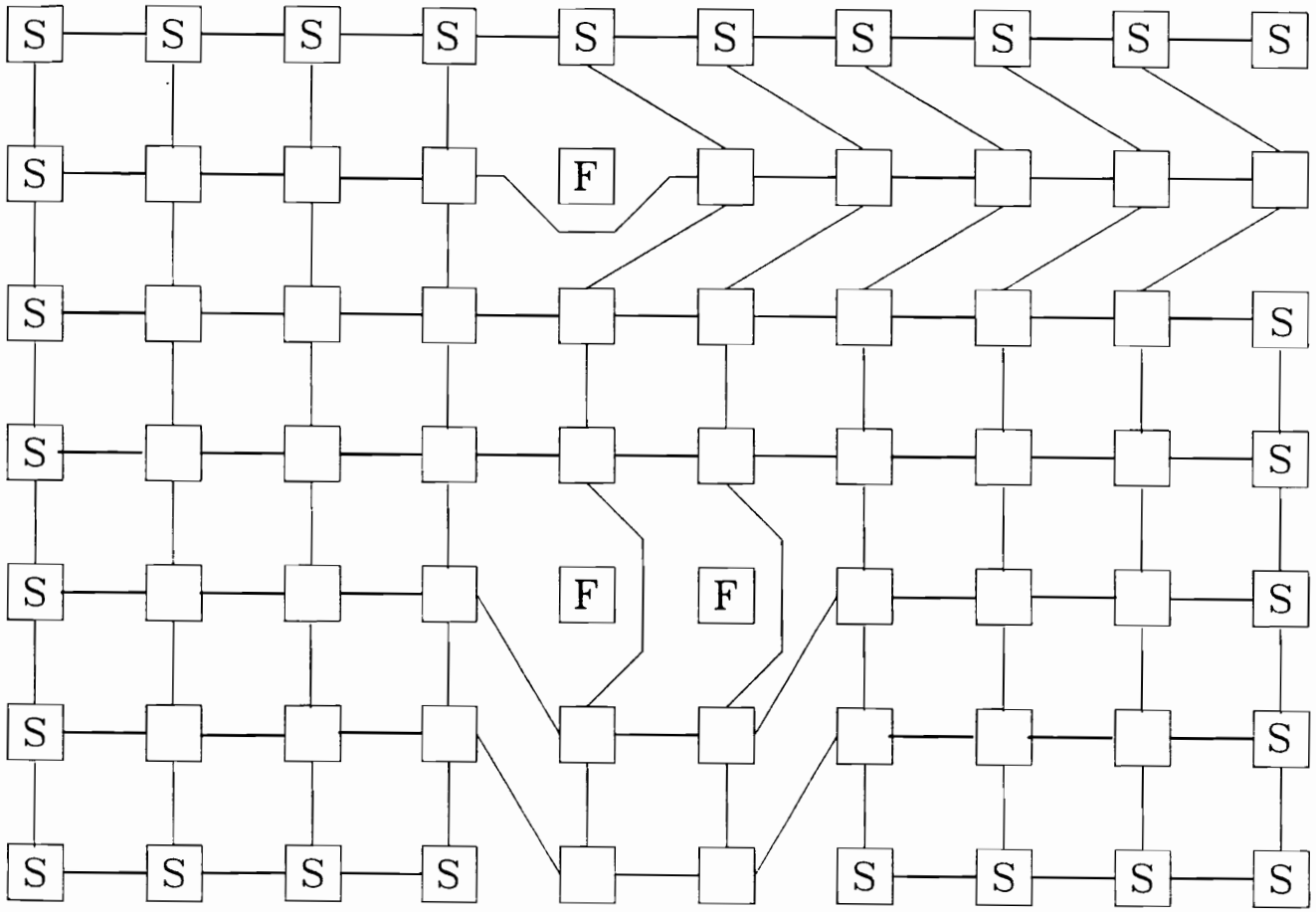


Figure 60. Three Faults Case 3-2b: Array with two faults in adjacent cells, one single fault.

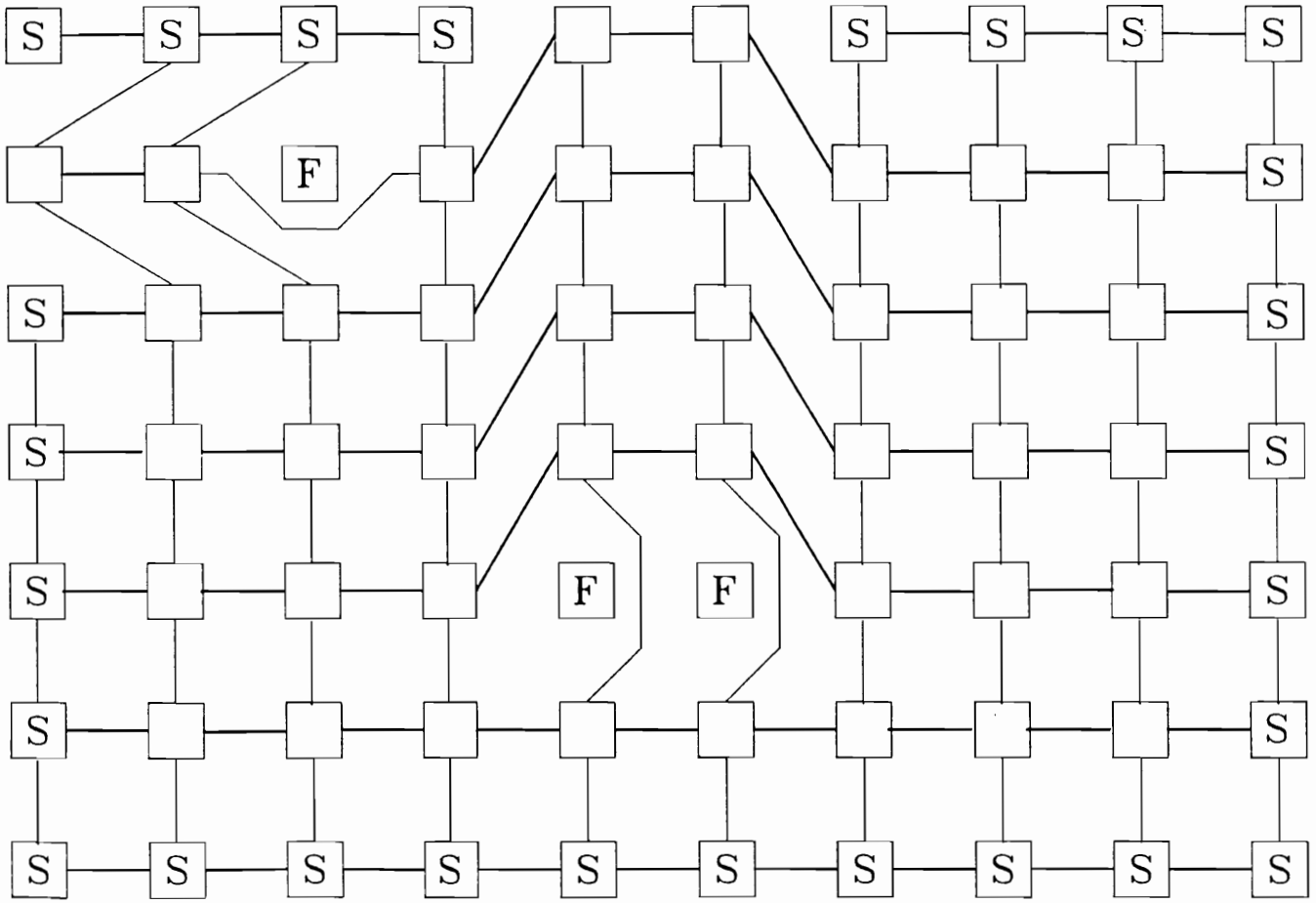


Figure 61. Three Faults Case 3-2b: Array with two faults in adjacent cells, one single fault.

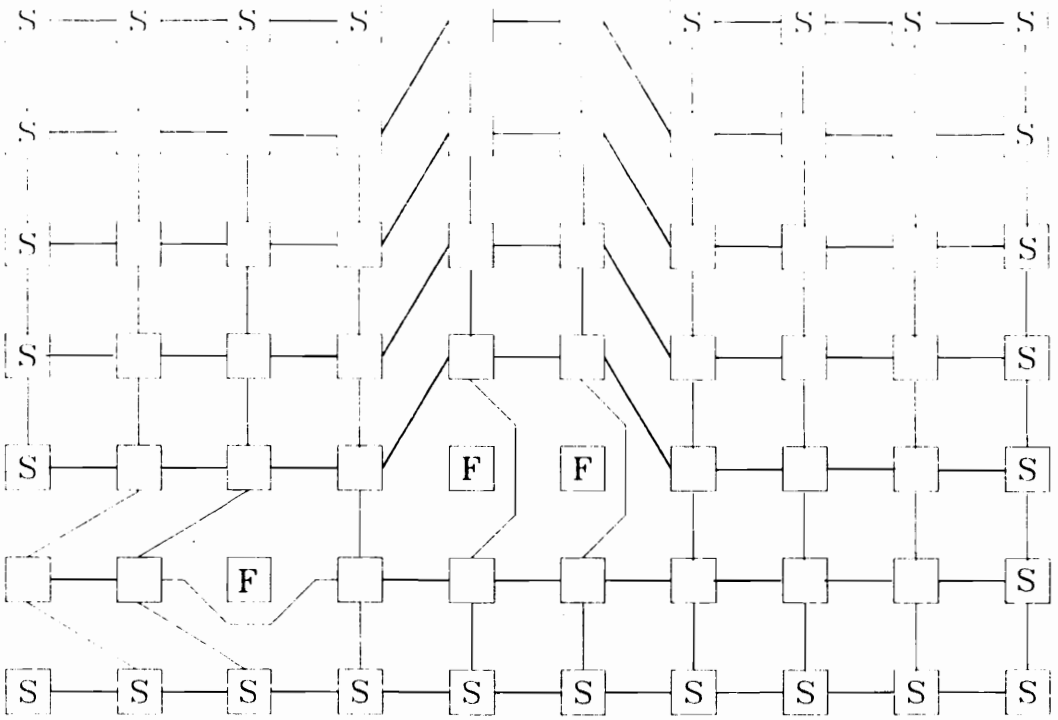


Figure 62. Three Faults Case 3-2b: Array with two faults in adjacent cells, one single fault.

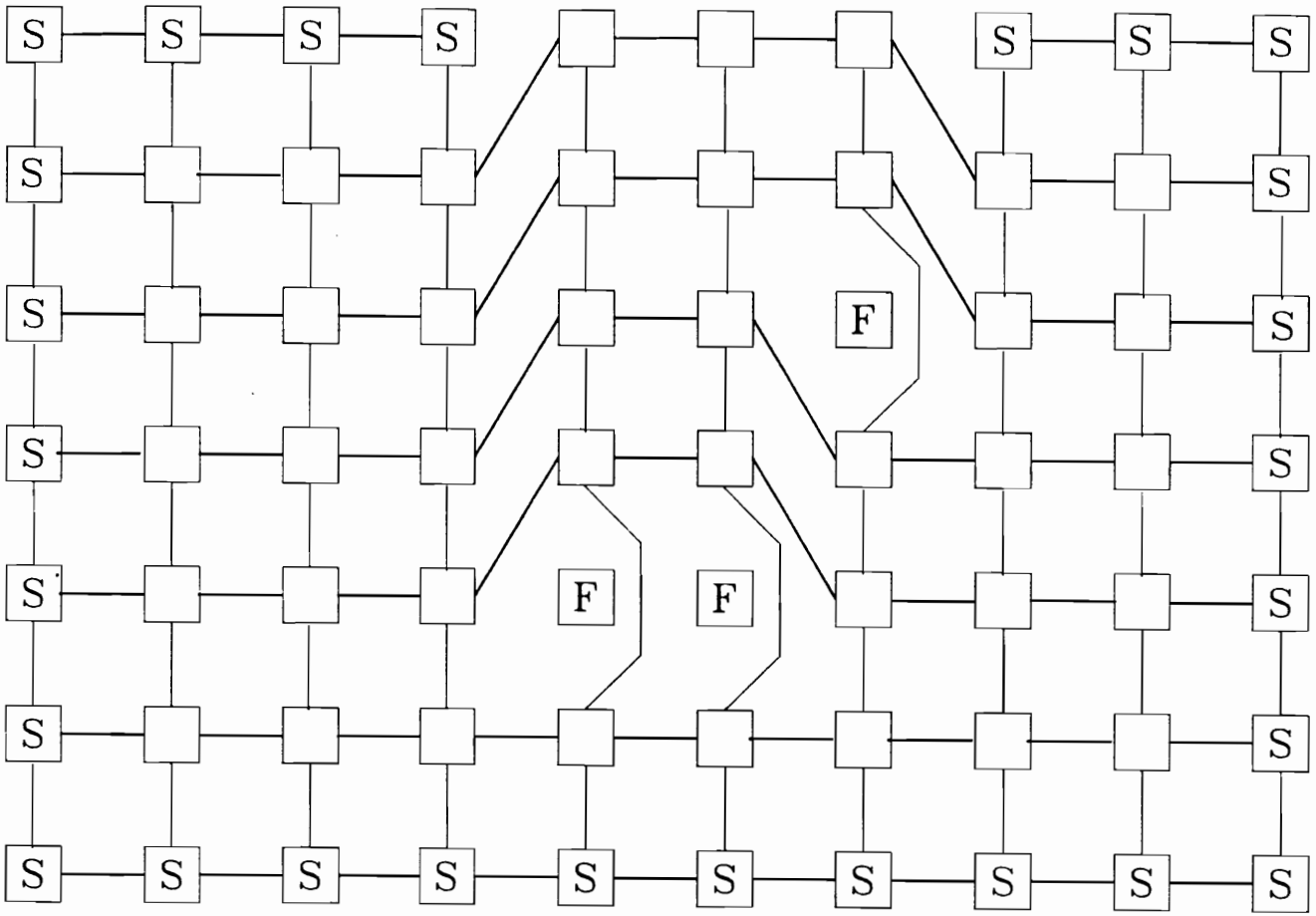


Figure 63. Three Faults Case 3-2b: Array with two faults in adjacent cells, one single fault.

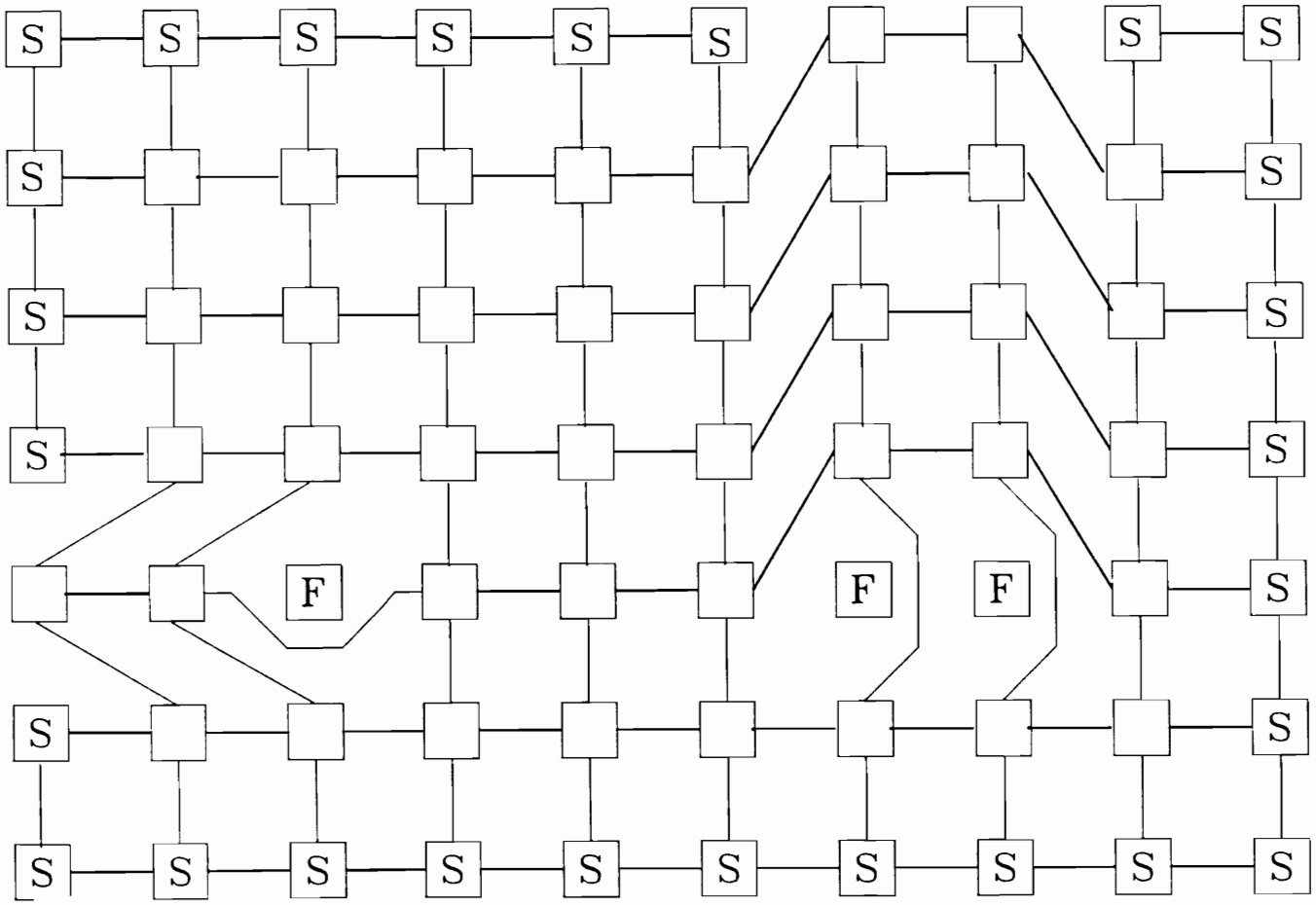


Figure 64. Three Faults Case 3-3: Array with three faults in one row.

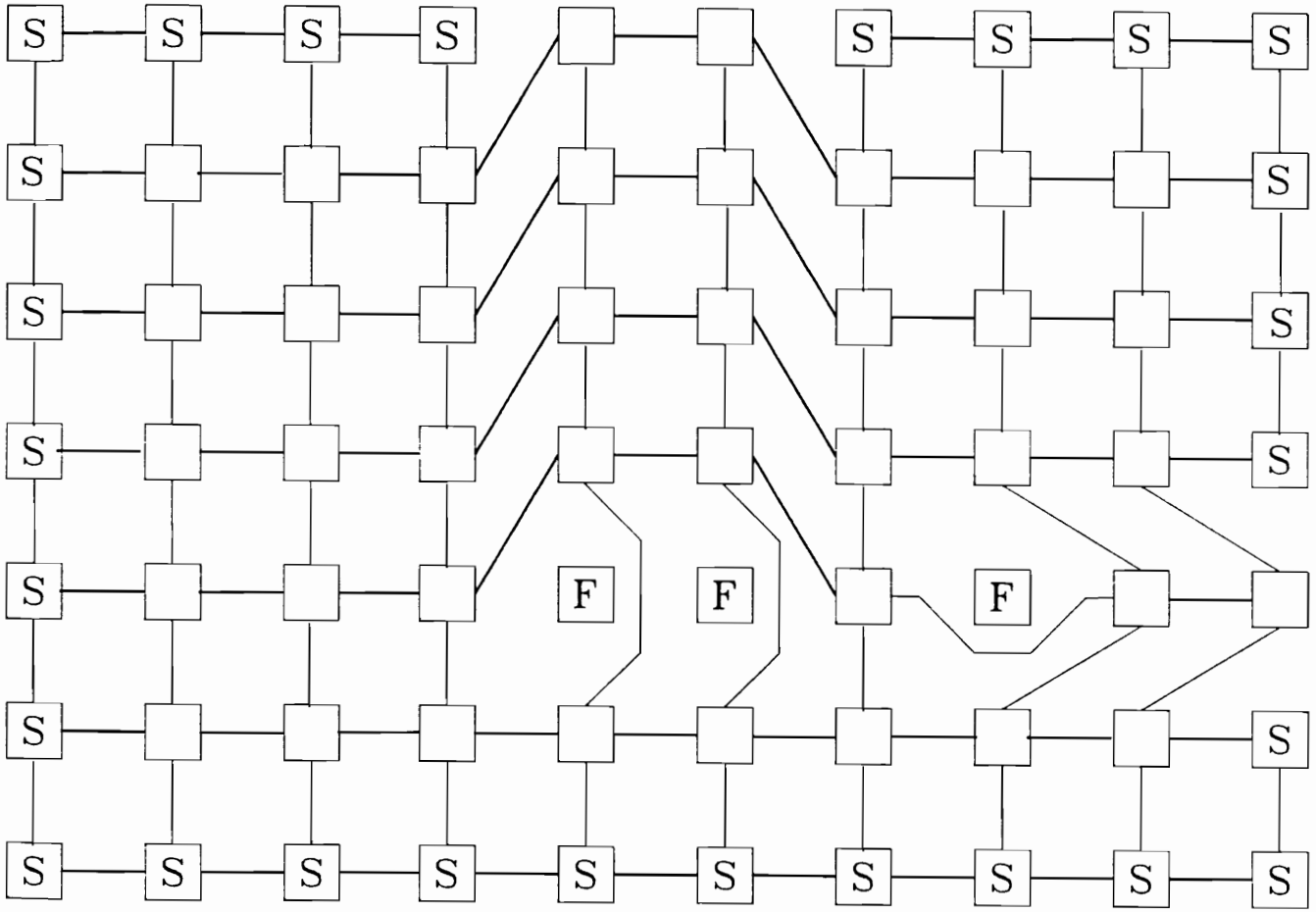


Figure 65. Three Faults Case 3-3: Array with three faults in one row.

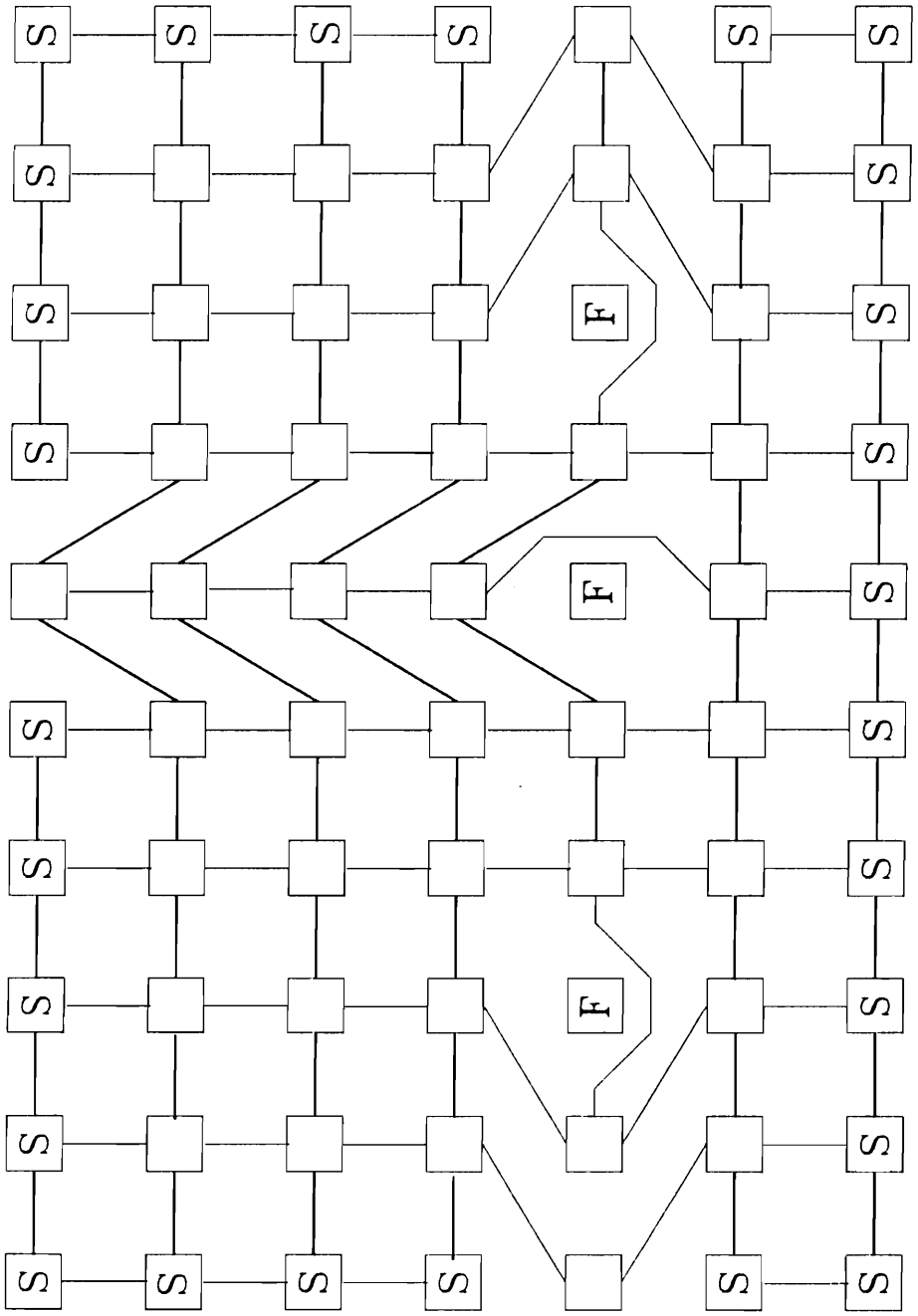


Figure 66. Three Faults Case 3-3c: Array with three faults in one row.

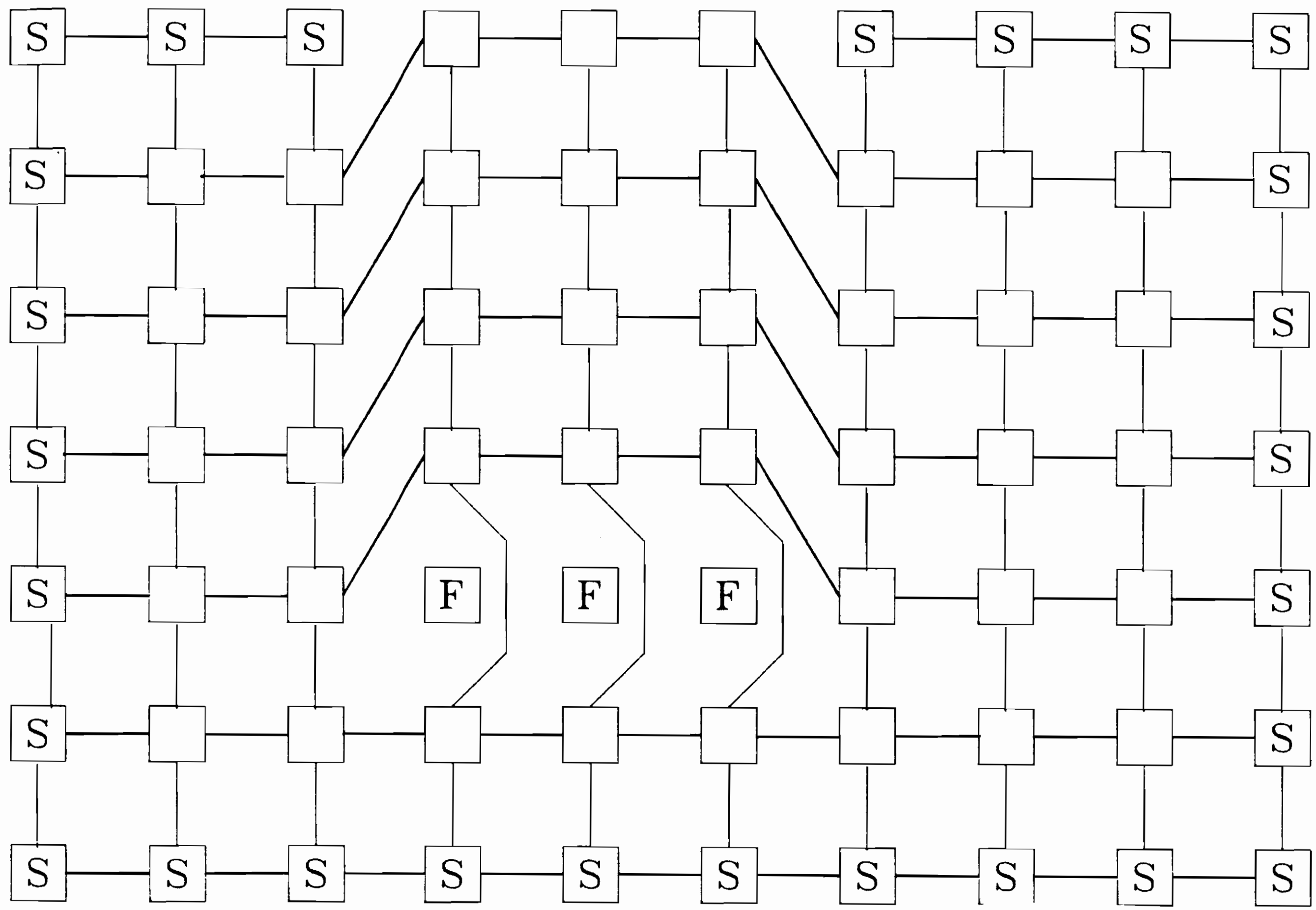


Figure 67. Three Faults Case 3-3a: Array with three faults in one row.

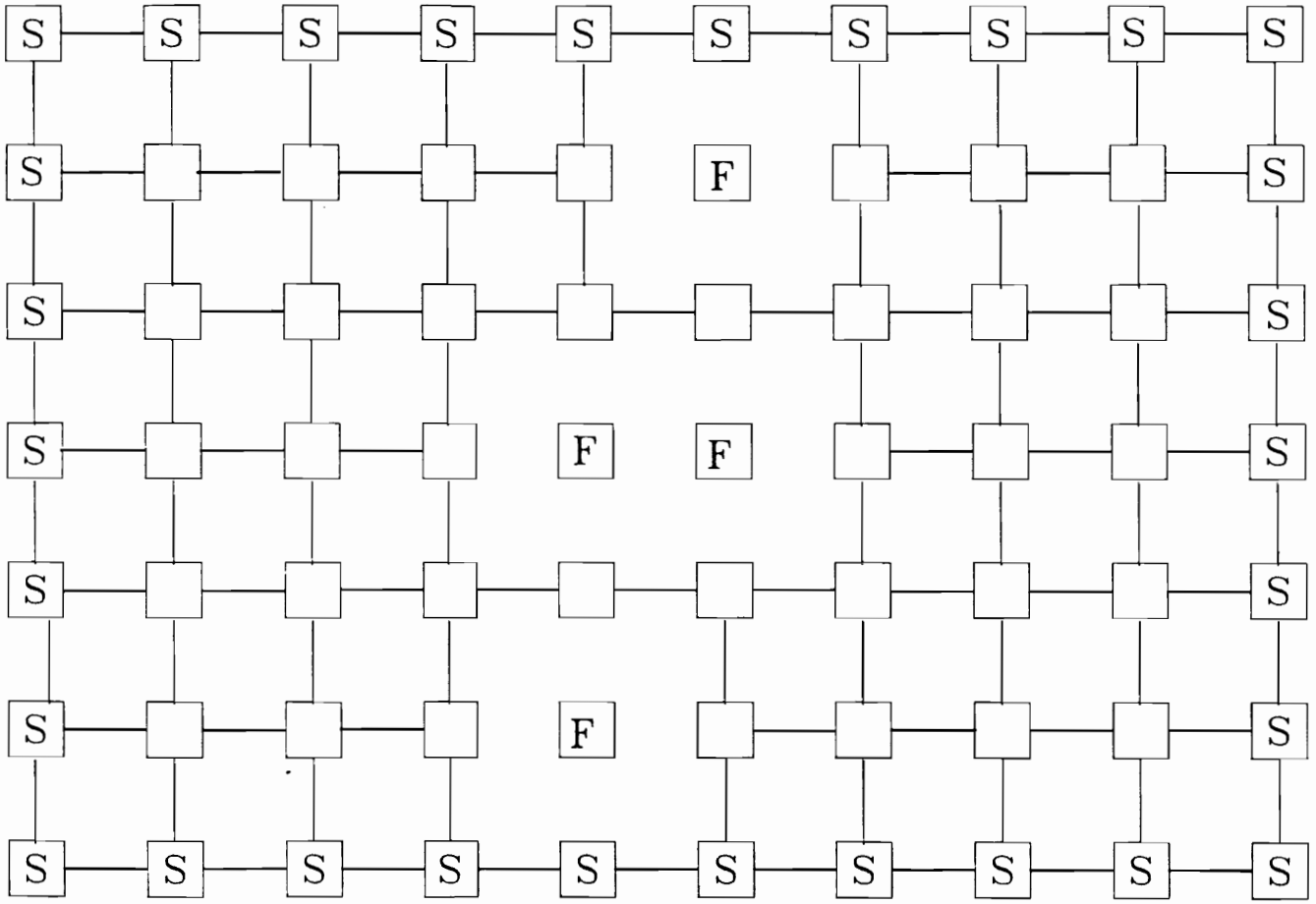


Figure 68. Four Faults Case 4-2: Array with four faults which cannot be tolerated.

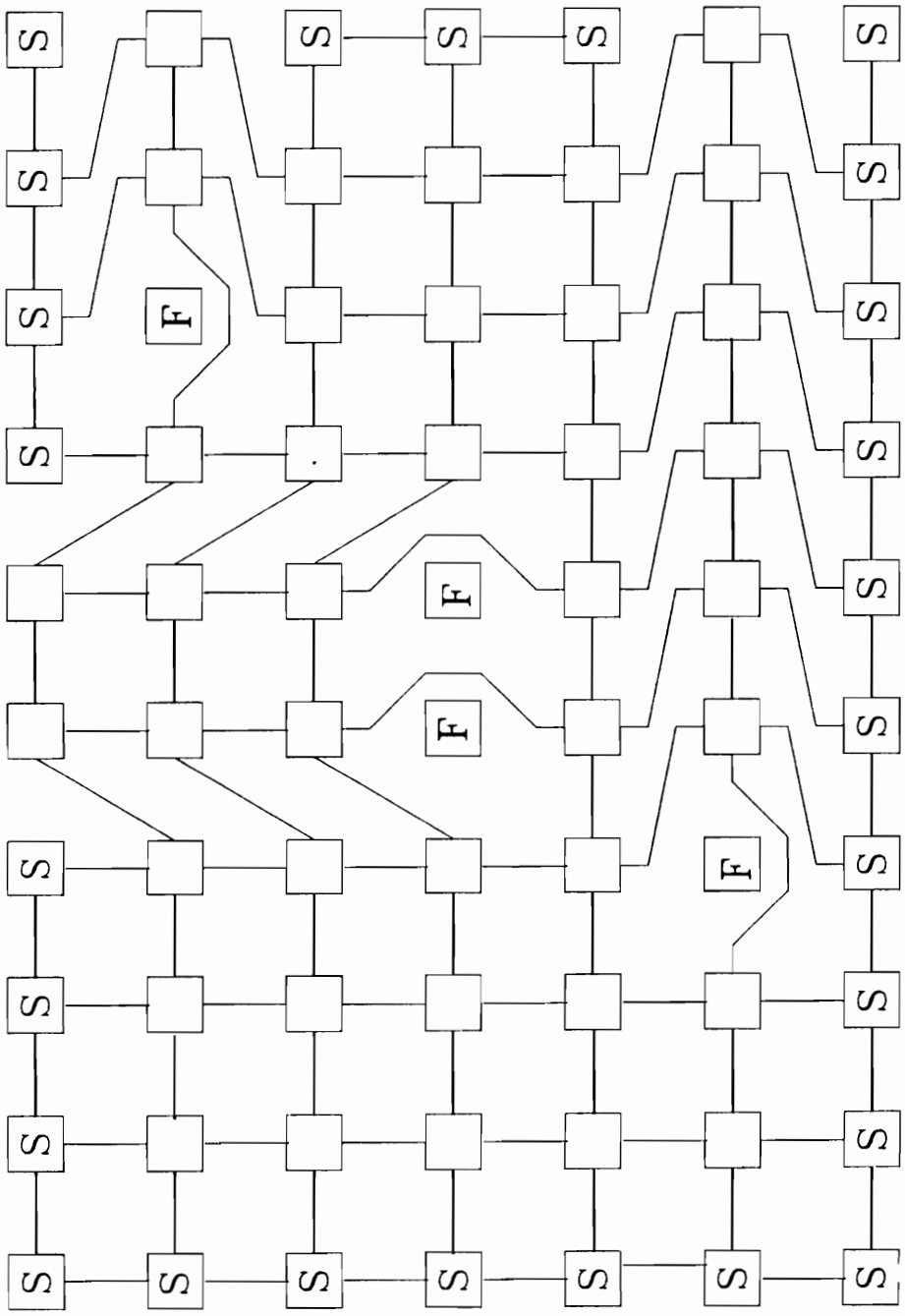


Figure 69. **Four Faults Case 4-2:** Array with two faults in one row and single faults in two other rows.

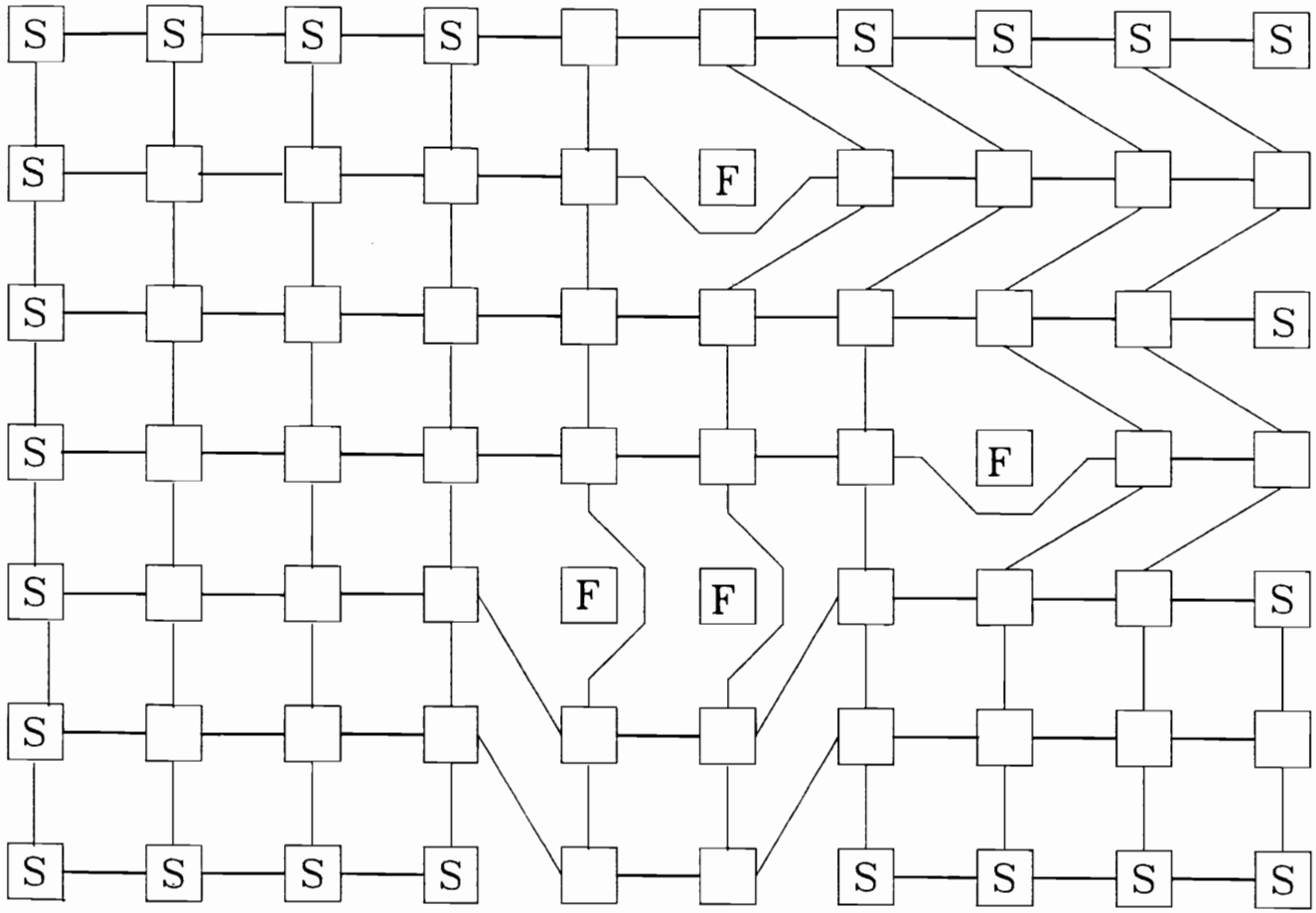


Figure 70. Four Faults Case 4-2: Array with two faults in one row and single faults in two other rows.

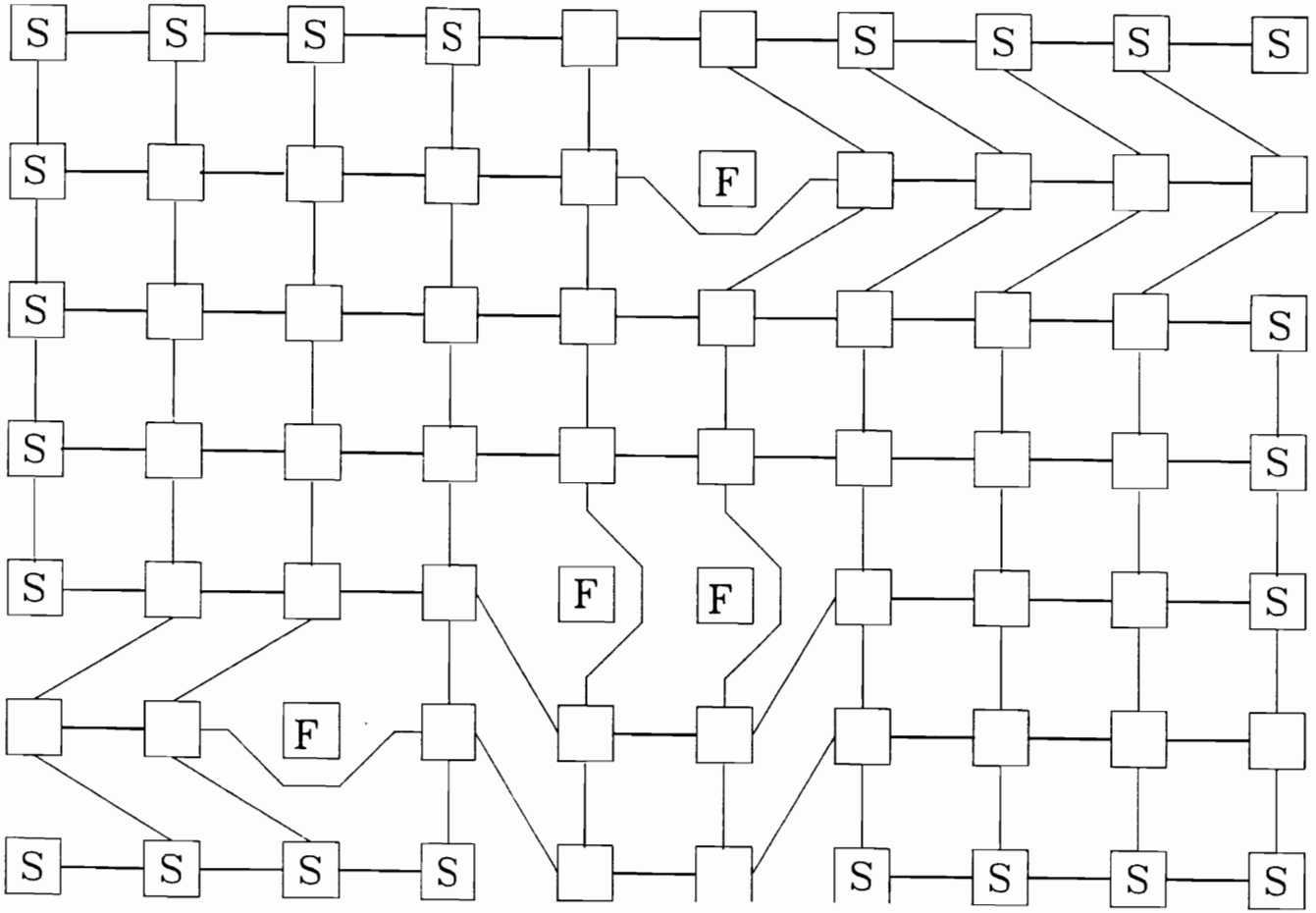


Figure 71. Four Faults Case 4-2: Array with two faults in one row and single faults in two other rows.

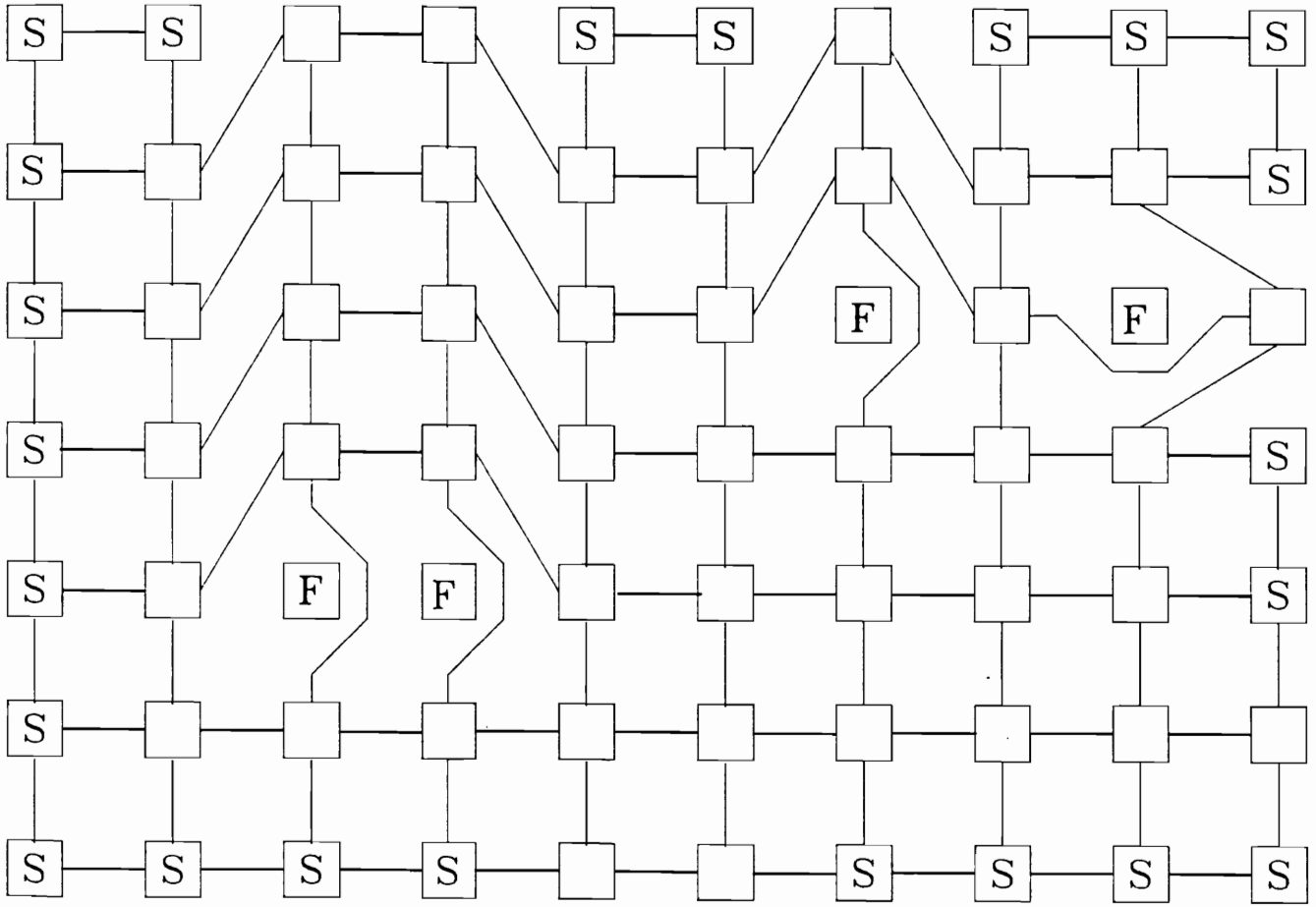


Figure 72. Four Faults Case 4-3: Array with two faults in each of two rows.

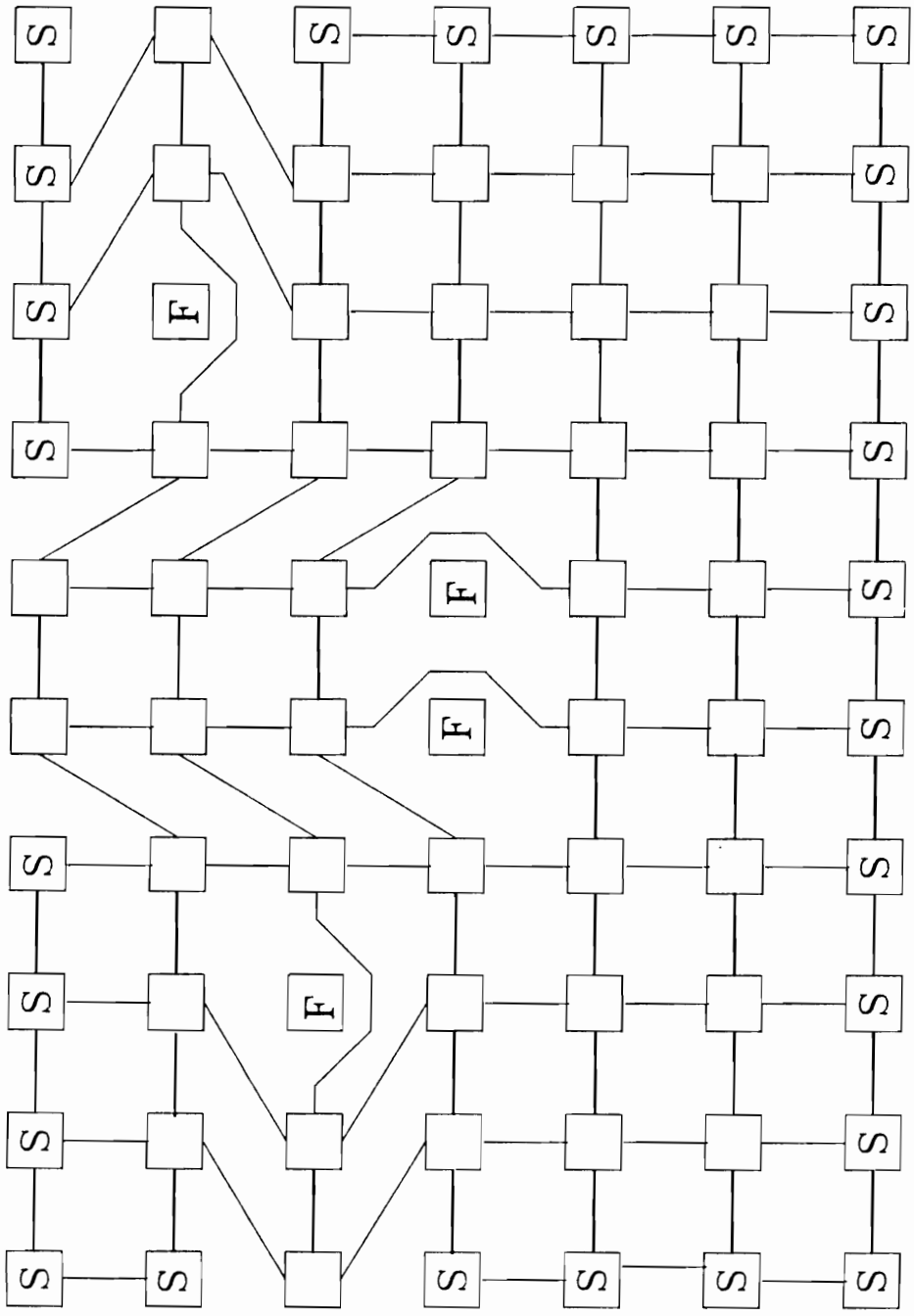


Figure 73. Four Faults Case 4-3: Array with two faults in two rows.

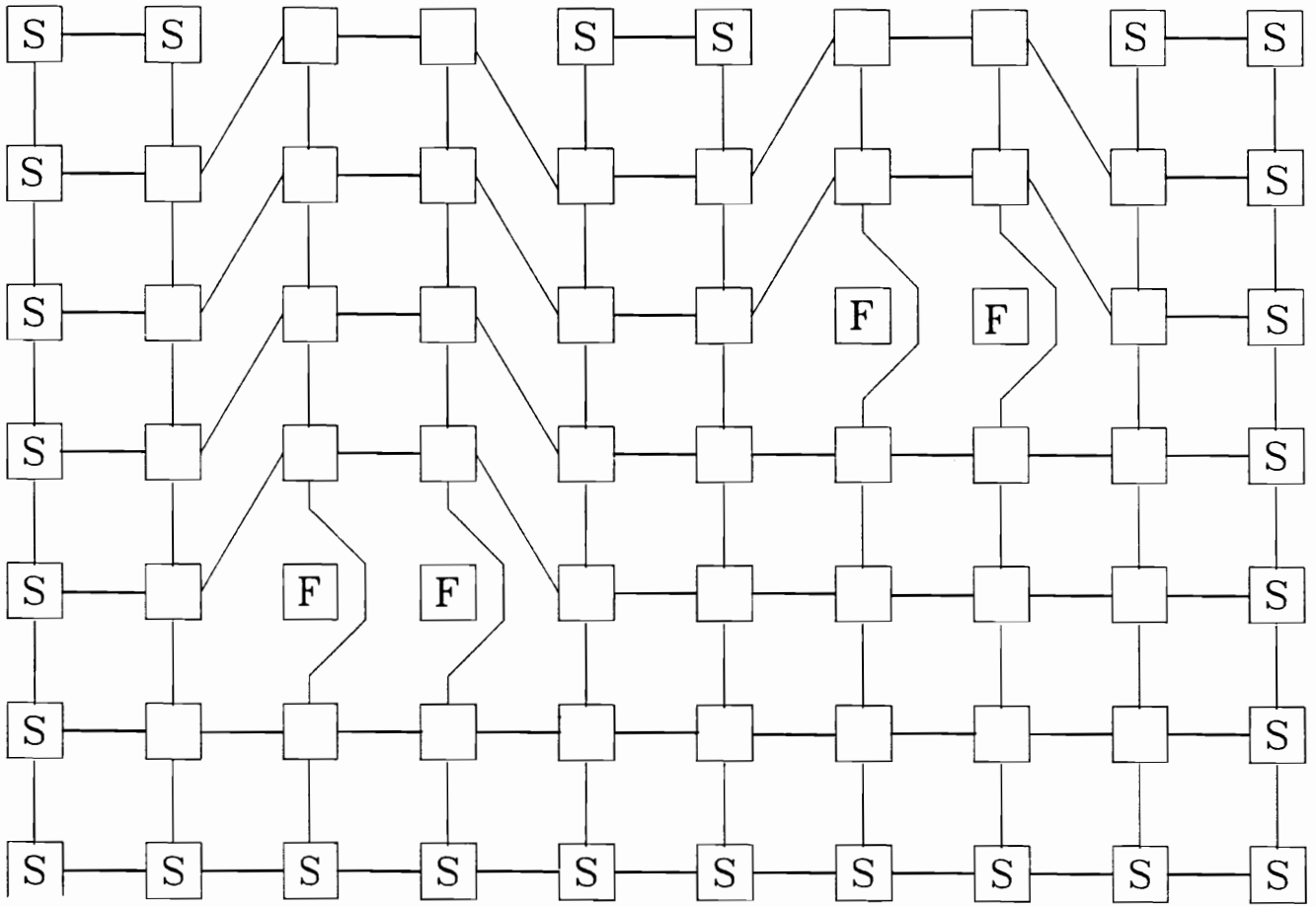


Figure 74. Four Faults Case 4-3: Array with two faults in each of two rows.

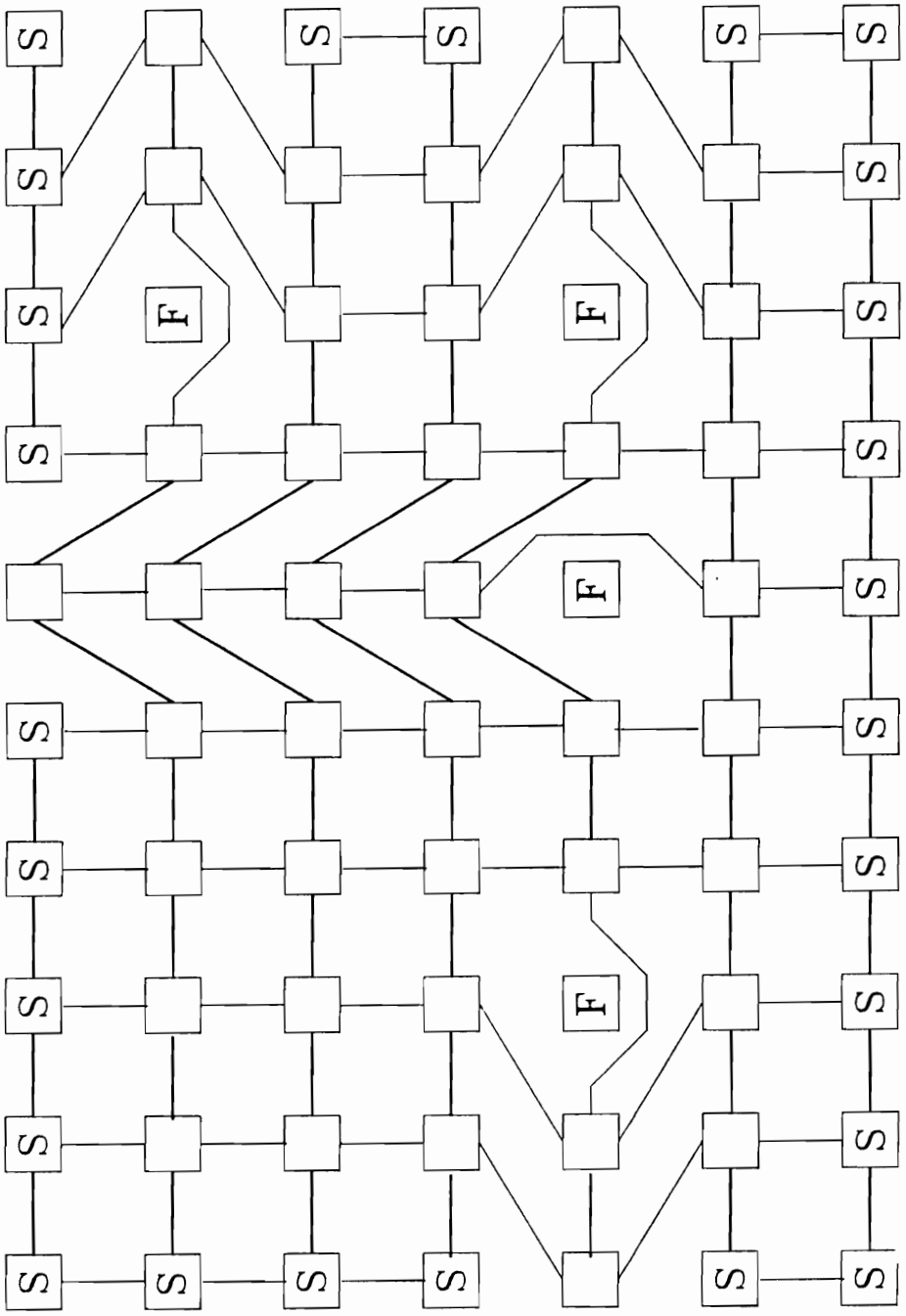


Figure 75. Four Faults Case 4-4: Array with three faults in one row.

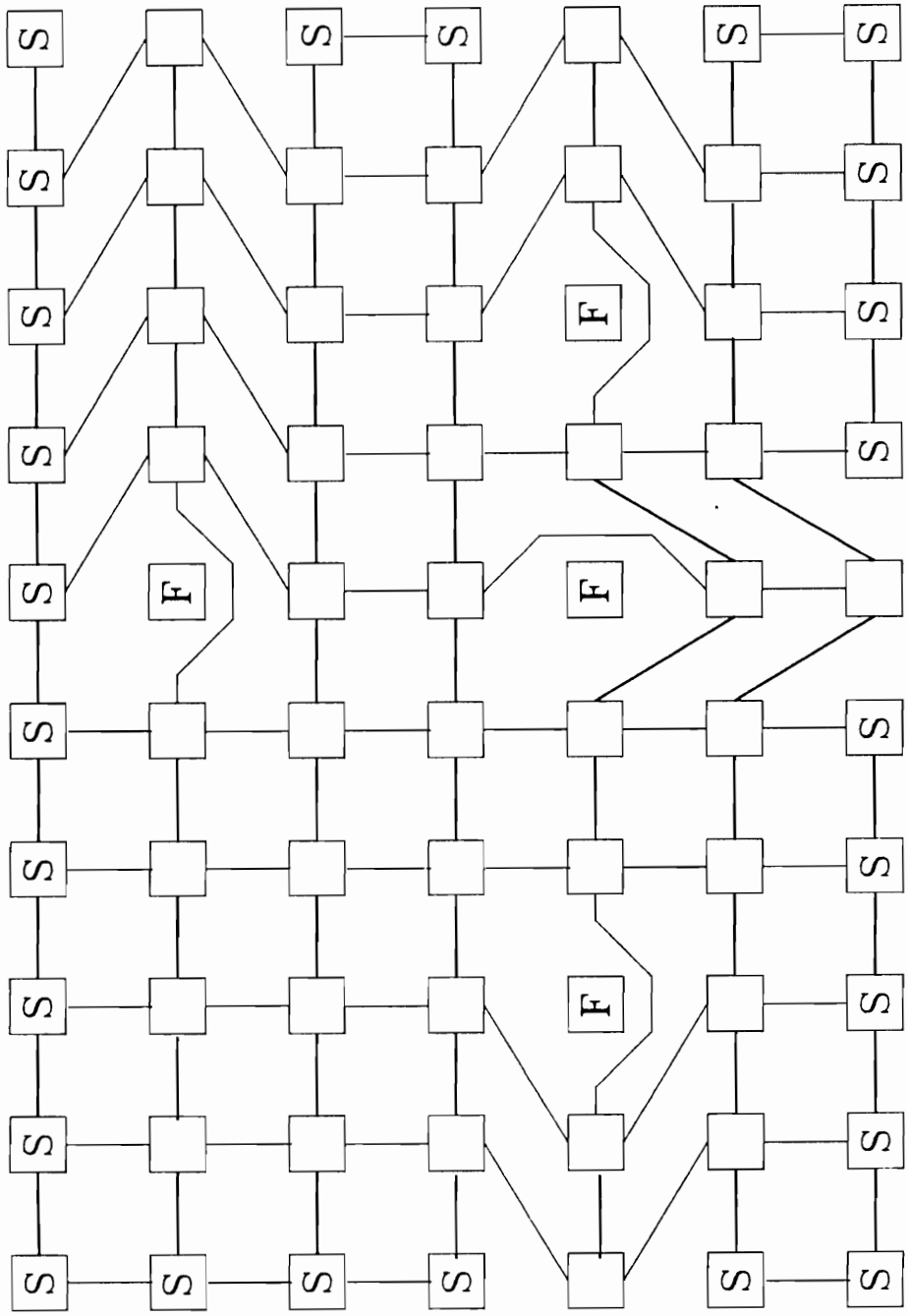


Figure 76. Four Faults Case 4-4: Array with three faults in one row.

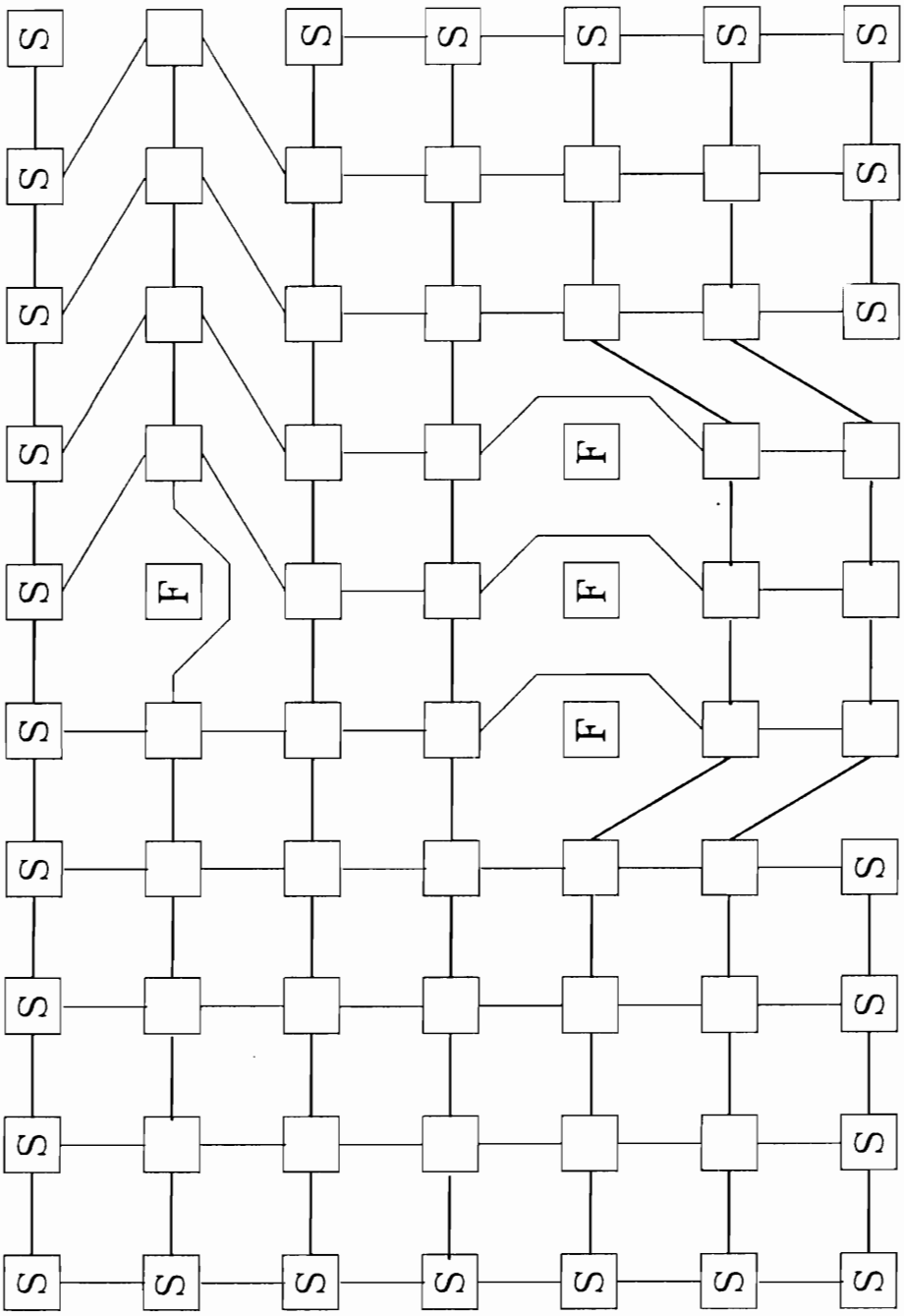


Figure 77. Four Faults Case 4-5: Array with three faults in one row.

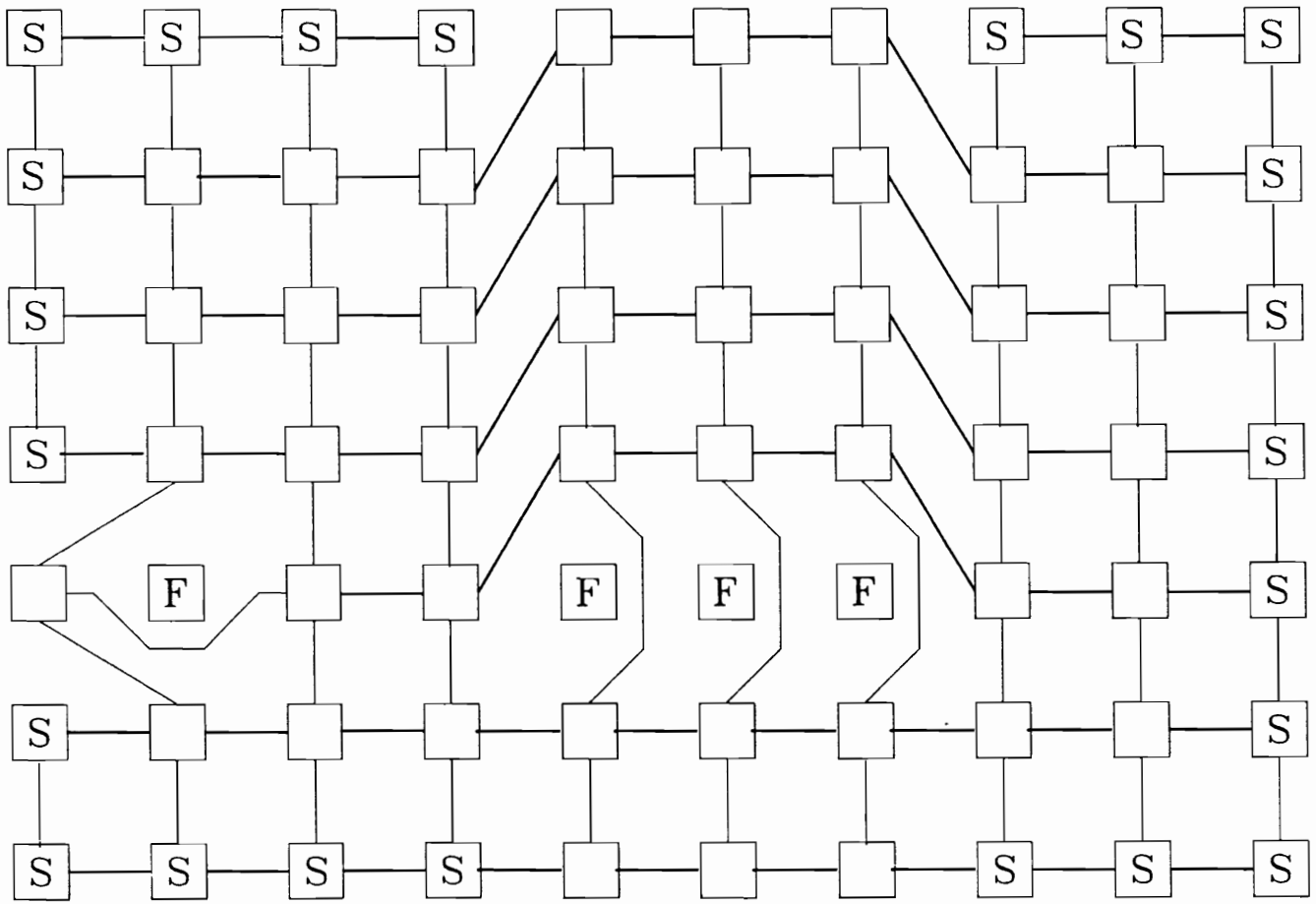


Figure 78. Four Faults Case 4-5: Array with four faults in one row.

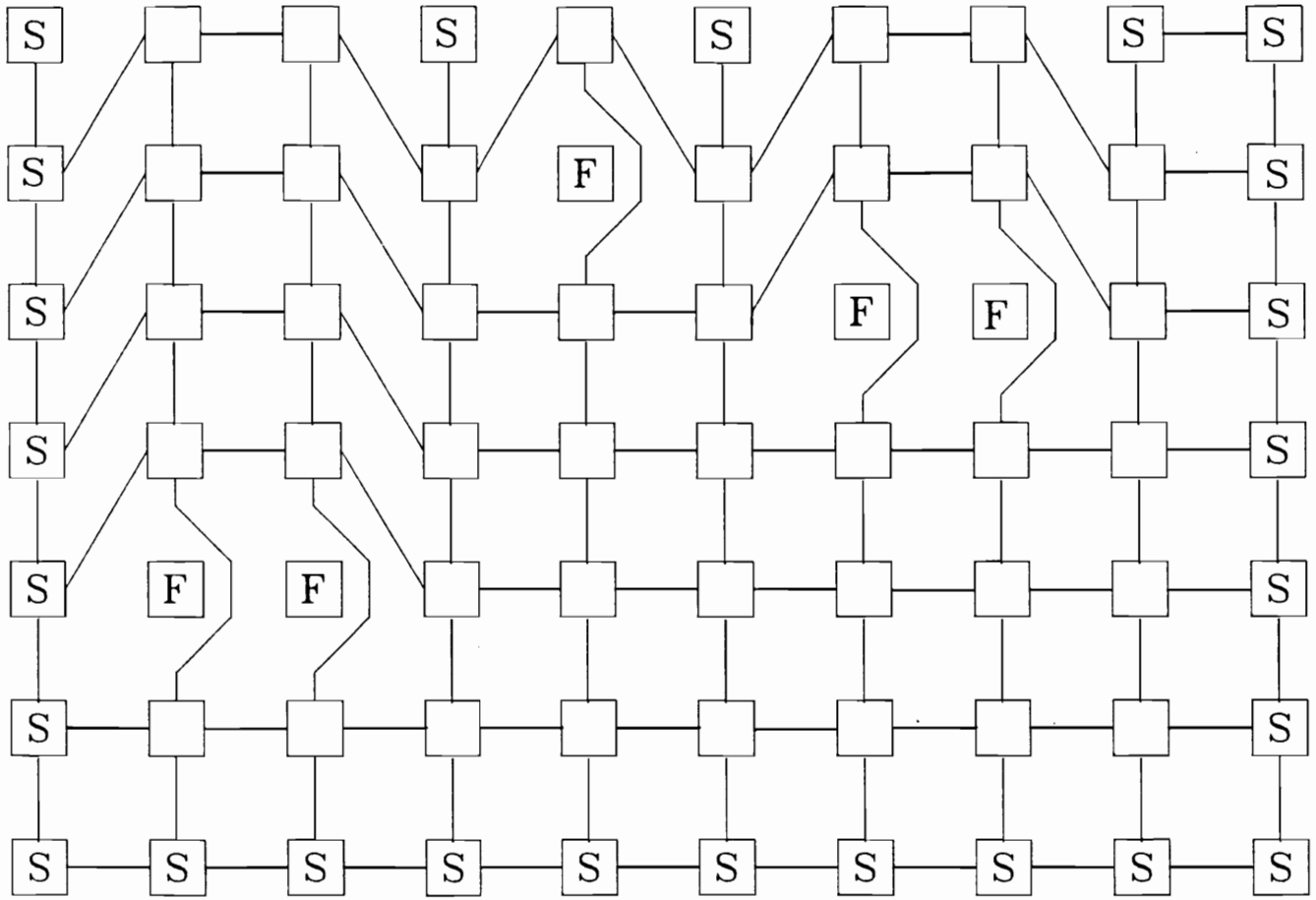


Figure 79. Five Faults Case 5-3: Array with two faults in each of two rows and a single fault in a third row.

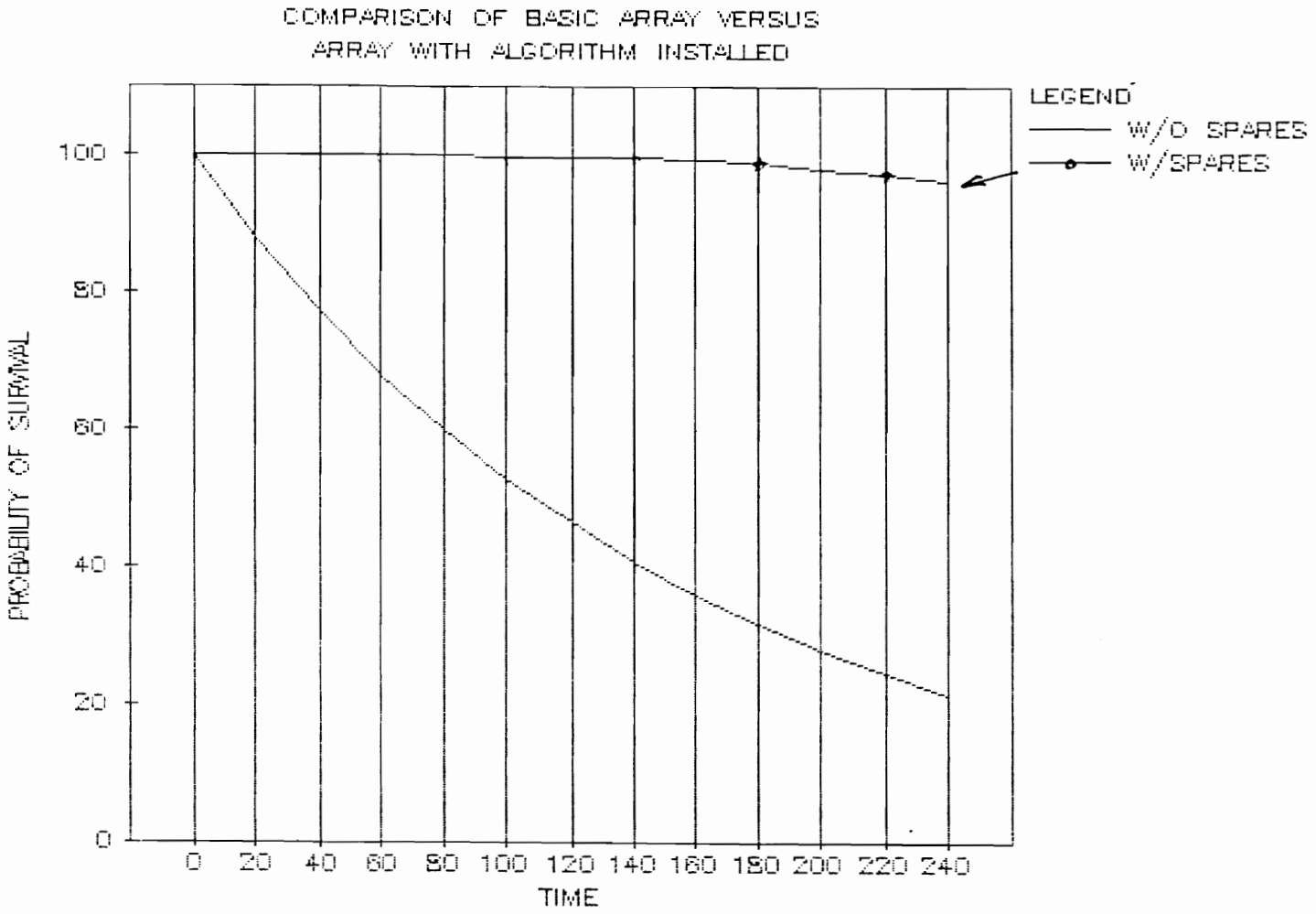


Figure 80. Plot of Probability of Survival vs Time: For an array using algorithm 4-12

6.0 Algorithm 4-24

Algorithm 4-24 is implemented using two rows and two columns of spare cells identical to the spares complement of algorithm 4-12. The complement of neighboring cells with which a cell can communicate has been increased to 24, or all cells which have a cell-distance of three or less. Figure 82 on page 219 shows the neighborhood of a cell.

The cells which can be configured as cell[i,j]'s NORTH neighbor are:

- Cell[i-1,j]
- Cell[i-2,j]
- Cell[i-3,j]

The possible SOUTH neighbors of cell[i,j] are:

- Cell[i + 1,j]
- Cell[i + 2,j]
- Cell[i + 3,j]

Cells which can be WEST neighbors are:

- Cell[i,j-1]
- Cell[i,j-2] /* not used */
- Cell[i-1,j-1]
- Cell[i + 1,j-1]

- Cell[i + 2,j-1]
- Cell[i-2,j-1] /* not used */

Those that are designated as possible EAST neighbors are:

- Cell[i,j + 1]
- Cell[i,j + 2] /* not used */
- Cell[i-1,j + 1]
- Cell[i + 1,j + 1]
- Cell[i + 2,j + 1] /* not used */

6.1 Spares Complement

Two spare rows of cells, one on the northern boundary and one on the southern boundary, are included, in addition to the two spare columns of spare cells used in algorithm 2-10 and 2-12. This is the same spares complement used in algorithm 4-12. Figure 55 on page 168 shows an 8 by 8 active array with two rows and two columns of spare cells.

6.2 Fault Register.

The fault register for this algorithm contains the same bits as the fault register used in algorithm 4-12, with the addition of three of the auxillary bits, the row-deformation-blocker bits, with these three bits also added to the decoder, plus NN and SS, for a total of 30 bits decoded.

ALGORITHM 4-12 FAULT REGISTER

Bit Position	Definition	Fault Condition
0	N	Cell[i-1,j]
1	NW	Cell[i-1,k]|k < j
2	NE	Cell[i-1,k]|k > j
3	S	Cell[i + 1,j]
4	SW	Cell[i + 1,k]|k < j
5	SE	Cell[i + 1,k]|k > j
6	E	Cell[i,j + 1]
7	FE	Cell[i,k]|k > j + 1
8	W	Cell[i,j-1]
9	FW	Cell[i,k]|k < j-1
10	NN	Cell[i-2,j]
11	NNW	Cell[i-2,k]|k < j
12	NNE	Cell[i-2,k]|k > j
13	MFNN	Two or more faults, row [i-2]
14	SS	Cell[i + 2,j]
15	SSW	Cell[i + 2,k]|k < j
16	SSE	Cell[i + 2,k]|k > j
17	MFSS	Two or more faults, row [i + 2]
18	MF	Two or more faults, row [i]
19	MFN	Two or more faults, row [i-1]
20	MFS	Two or more faults, row [i + 1]
21	RDN	Row deformation north
22	RDNE	Row deformation northeast
23	RDNW	Row deformation northwest
24	FCR	Fault in row [i] and west of a multiple fault in row [i + 1] or [i-1]
25	FCRN	Fault in row [i-1] which is west of a multiple fault in row [i-2] or [i]
26	FCRS	Fault in row[i + 1] which is west of a multiple fault in row [i] or [i + 2]
27	RDFS	Row deformation begins in this row A condition exists in row[i + 1] which requires Row deformation
28	RDFT	Condition exists in this row which requires row deformation
29	RDFN	Condition exists in row[i-1] which requires row deformation
30	RDS	Force row deformation to the south instead of the north. This bit is set when a fault occurs north of a deformed row
31	RDE	Row deformation to the east. This bit is set whenever a fault exists in a row with a deformation to its east

32	RDW	Row deformation to the west.
33	RDNB	Row deformation to the north blocked.
34	RDNEB	Row deformation to the northeast blocked.
35	RDNWB	Row deformation to the northwest blocked.

6.2.1 Setting the Fault Register

The bits in the fault register are set identically to the bits in the fault register of algorithm 4-12. Two additional bits are used for decoding in this algorithm, the NN and the SS bits. For a description of the logic used in determining the state of the fault register, see the previous chapter.

The three new bits, row-deformation blockers to the north, northeast and northwest are set whenever a deformation encounters a faulty cell in its path. This is indicated by the occurrence of the row-deformation-north bit set in a cell which has a faulty cell to its immediate north. When this occurs, the RDNB bit, row-deformation- north-blocker is set in that cell with the RDNEB bit set whenever the RDNE bit is set and the northeastern neighbor is faulty. The RDNWB is set if RDNW is set and the northwestern neighbor is faulty.

6.2.2 The Algorithm

Algorithm 4-24 is identical to algorithm 4-12 with two exceptions. This Algorithm will tolerate two classes of faults which could not be tolerated by algorithm 4-12. Those are the faults which occur as adjacent faults in one row, with a cell which is adjacent to one of the row faults, but in another row also being faulty. This would occur if cell[i,j], cell[i,j + 1] and cell[i-1,j] were faulty. The second class of faults tolerated by this algorithm is that of adjacent faults in one row with faults one of the same columns to the north of the adjacent faults and one to the south in one of the same columns.

In the second class of fault configurations, the faults to the north and south of the adjacent faults could be in any column which contained one of the adjacent faults. In this algorithm, these faults can be tolerated as long as the northern and southern faults do not occur in the same column. This condition cannot be reconfigured.

More significant benefits cannot be achieved with the additional interconnections because cells may have different states. Reconfiguration cannot be done to cells on a diagonal because of the four-cell neighborhood. This problem can be demonstrated by an example. Figure 83 on page 220 shows a fault pattern which cannot be tolerated even though the cell interconnection will support reconfiguration.

Assume three cells are faulty in row[i] with the center fault in cell[i,j] and a faulty cell in cell[i + n,j] south of the center fault. Reconfiguration would proceed with row deformation above the center fault in cell[i,j]. If cell[i,j] had a fault-free state of "d", cell[i-1,j] takes on state "d". The state pattern will be shifted up as row deformation proceeds. A fatal condition will exist in this case, if cell[i-n,j] with a fault-free state of 'a' becomes faulty during the cycle in which cell[i-n + 1,j] takes on state 'c'. At this time, two states are missing from the state pattern, 'a' and 'b'. Only two cells exist which can take on these states and provide a path towards a spare cell. Those cells are cell[i-n,j-1] and cell[i-1,j + 1]. Neither of these cells has any knowledge of the previous state of cell[i-n + 1,j]. Row deformation must be blocked and reversed to the south. This deformation is blocked by the faulty cell[i + n,j]. The additional interconnections used to implement this algorithm provide little additional fault coverage. Figure 84 on page 221 shows another instance of the same fault configuration. In this case the fault in cell[i-n,j] occurred some time before the fault in cell[i,j]. In this case the cell with state 'a', cell[i-n,j + 1] could possibly take on state 'b', with 'a' shifted to cell[i-n-1,j], which would contain this state if cell[i-n,j] were not faulty. If the cell[i-n,j] were the western fault of two faults in row[i-n], then cell[i-n,j + 1] would not be available to take on state 'b' because it would not be connected to that cell. In this case cell[i-n,j-1] is the only one available.

If the fault in cell[i-n,j] occurs after the row deformation, then no cell in row[i-n] has knowledge of the state of cell[i-n,j] because its eastern and western neighbors would then be cell[i-n + 1,j-1] and cell[i-n + 1,j + 1]. The set of cells available to assume the function of cell[i-n + 1,j] contains only one cell which is common in all cases, and that cell is cell[i-n + 1,j]. This cell must maintain the same state if reconfiguration is to be successful, meaning configuration must proceed to the south. If a cell[i + n,j] is also faulty, a fatal conditions exists for the same reasons given above.

```
                                        { /* begin Algorithm 4-24 */
                                          /* begin WEST code  */
Switch(RD) /* any row deformation bit */
   {
Case (null) {

1. if(W)        connect far-west

2. else         connect west
   break; }
   Case(RD){

3. if(((RDS and RDN and not RDNW) or(RDNW and not(RDS or RDN))
                and not NW)
                connect north_west

4. if(((RDS and RDN and not RDNW) or(RDNW and not(RDS or RDN)) and NW)
                connect far_north_west

5. else if(not RDS and RDN and not RDNW and not SW)
                connect south-west

6. else if(not RDS and RDN and not RDNW and SW)
                connect far-south-west

7. else if((RDE and not RDW and W and not FW)
          or(RDW and not RDE and W and not(E or FE))
                connect far-west

8. else         connect west
   break; }
    } /* end WEST code   */

                                        /* begin EAST code */
Switch(RD)  { /* and row-deformation bit */
 Case(null){

9. if(E)        connect far-east

10. else        connect east
    break;  }
     case(RD){

11. if(((RDS and RDN and NOT RDNE) or (RDNE and NOT(RDN or RDS))
        and not NE)
         connect north-east
```

```
12. if(((RDS and RDN and NOT RDNE) or (RDNE and NOT(RDN or RDS))
          and  NE)
           connect far-north-east

13. else if(((RDS and RDNE and NOT RDN) or (RDN and NOT(RDNE or RDS))
             and not SE)
            connect south-east

14. else if(((RDS and RDNE and NOT RDN) or (RDN and NOT(RDNE or RDS))
             and SE)
            connect south-east

15. else if((RDW and not RDE and E and not FE)
             or (RDE and not RDW and E and not(W or FW)))
                 connect far-east

16. else         connect east
    break }
     } /* end EAST code   */

    /* begin NORTH link code */

    switch(MF and MFN and FCR and FCRN and RD)
     {
    case (null){

17. if(
      (not(W or FW) and not(N or NW))
        or((W or FW) and  NW)
        )
       connect north

18. else if(
      ((N or NW) and not(W or FW))
        )
       connect north-east

19. else if(
         ((W or FW) and not( NW))
           )
       connect north-west
       break; } /* end case  */
    case (FCR){

20. if(
       ((W or FW) and not(N or NW))
         )
       connect north

21. else if(
       ((W or FW) and (N or NW))
       or(not(W or FW))
             )
           connect north-east

22. else  /* not possible in this configuration */
           connect north-west
```

```
    break; }
  case (FCRN){
23. if(
      (( NW) and not(W or FW))
            )
          connect north

24. else if(
      (not( NW)) or(( NW) and (W or FW))
            )
          connect north-west

25. else     connect north-east
       break; }
  case (FCR and FCRN){

26. if(
      (not(W or FW) and not(N or NW))
      or((W or FW) and ( NW))
          )
          connect north

27. else if(
       (not(W or FW) and (N or NW))
            )
          connect north-east

28. else if(
      ((W or FW) and not( NW))
            )
          connect north-west
      break; }
  case (MF){

29. if(
      ((W or FW) and ( E or FE) and not(N or NW))
      or(NW and not( E or FE))
          )connect north

30. else if(
      (not(W or FW))
      or((W or FW) and ( E or FE) and (N or NW))
          )connect north-east

31. else if(
      (not( E or FE) and not( NW))
          )connect north-west
      break; }

  case (MF and FCRN){

32. if(
      ( NE)  or((W or FW) and ( E or FE))
          )connect north

33. else if(
```

```
      (not(W or FW) and (N or NW))
           )connect north-east
34.  else if(
      (not( E or FE))
           )connect north-west
      break; }

     case (MFN){
35.  if(N and NW and NE)
      connect far-north
36.  else
37.  if(
      (not(W or FW) and (NE and NW)
       or((W or FW) and (not( N or NE)))
        )connect north
38.  else if(
      (not( W or FW or NE))
      )connect north-east
39.  else if(
      (not( NW))  or((W or FW) and ( N or NE) and ( NW))
           )connect north-west
      break; }

     case (MFN and FCR){
40.  if(N and NE and NW)
       connect far-north
41.  else if(
      (not(W or FW))   or(NE and NW)
      )connect north
42.  else if(not( NE))
      connect north-east
43.  else if(
      ((W or FW) and not( NW))
           )connect north-west
      break; }

     case (MF and MFN){
44.  if( (not(W or FW) and not(N or NW))
      or((W or FW) and ( E or FE) and (NE and NW)
      or(not( E or FE)   and not( N or NE))
           )connect north
45.  else if(
      (not(W or FW) and (N or NW))
      or(( E or FE) and (W or FW) and not( NE))
           )connect north-east
46.  else if(
```

```
    ((W or FW) and not( NW))
    or(not( E or FE) and ( N or NE))
        )connect north-west
    break; }
  case(RD){
47. if(RDS and N and not(RDE or RDW) and not NN)
        connect far-north
48. if(RDS and N and not(RDE or RDW) and NN)
        connect far-north-north
49. else if(RDE and not RDW and( not(E or NE)  or (E and NE)))
        connect north
50. else if(RDE and not RDW and E and not NE)
        connect north-east
51. else if(RDE and not RDW and NE and not E)
        connect north-west
52. else if(RDW and not RDE and( not(W or NW)  or (W and NW)))
        connect north
53. else if(RDW and not RDE and W and not NW)
        connect north-west
54. else if(RDW and not RDE and NW and not W)
        connect north-east
55. else if(RDW and RDE and not N)
        connect north
56. else if(RDW and RDE and N)
        connect far-north
57. else connect north
        }
   } /* end switch NORTH link code */

   /*  begin SOUTH link code  */
   switch(MF and MFS and FCR and FCRS and RD)
   { /* south link switch code */
   case (null)
     {
58. if(
     (not(W or FW) and not( S or SW))
     or((W or FW) and ( SW))
        )connect south
59. else if(
     ((W or FW) and not( SW))
        )connect south-west
60. else if(
     (not(W or FW) and ( S or SW))
        )connect south-east
     break; }
```

```
    case (FCR)    {
61. if(
      ((W or FW)
         and (not( S or SW)))
      )connect south

62. else if(
      ((W or FW) and (( S or SW)))
      or(not(W or FW))
      )connect south-east

63. else    connect south-west
      break; }

    case (FCRS)   {

64. if(
      (not(W or FW) and  SW)
      )connect south

65. else if(
      (not( SW))
      or(( SW) and (W or FW))
           )connect south-west

66. else     connect south-east/* not possible this configuration */
      break; }

    case (FCR and FCRS)   {

67. if(
      (not(W or FW))
      or((W or FW) and  SW)
           )connect south

68. else if(
      ((W or FW) and not( SW))
           )connect south-west

69. else if(
      (not(W or FW) and( S or SW))
           )connect south-east
      break; }

    case (MF)             {

70. if(
      ((W or FW) and ( E or FE) and not( S or SW))
             )connect south

71. else if(
      (not( E or FE or SW))
           )connect south-west

72. else if(
      (not(W or FW))
```

```
      or((W or FW) and ( E or FE) and( S or SW))
            )connect south-east
      break}

    case (MF and FCRS)     {

73. if(
      ( SE )
      or((W or FW) and ( E or FE))
           )connect south

74. else if(
      (not( E or FE))
           )connect south-west

75. else if(
      (( S or SW) and not(W or FW))
           )connect south-east
      break; }

    case (MFS)          {

76. if(
       (S and SE and SW) or(RDFS and S)
       )
        south_link[cell_number] = 18;

77. else

78. if(
      ((SE and SW) and  not(W or FW))
           )connect south

79. else if(
      (not( SW))  or(( S or SE ) and ( SW) and (W or FW))
           )connect south-west

80. else if(
      (not( W or FW or SE ))
      )connect south-east
      break; }

    case (MFS and FCR)   {

81. if(
      (SE and SW) or(not(W or FW))
           )connect south

82. else if(
      ((W or FW) and not( SW))
           )connect south-west

83. else if(
      (not( SE ))
           )connect south-east
      break; }

    case (MF and MFS)
```

```
      {
84.  if(
       ((W or FW)
       and ( E or FE) and (SE and SW)
       or(not( E or FE) and not( S or SE ))
       or(not(W or FW) and not( S or SW))
       )connect south

85.  else if(
       ((W or FW)  and not( SW))
       or(not( E or FE) and ( S or SE ))
       )connect south-west

86.  else if(
       ((W or FW)and (E or FE) and not( SE ))
        or(not( E or FE) and ( S or SE ))
        or(not(W or FW) and( S or SW))
        )connect south-east
       break; }
     case(RD) {

87.  if((RDN and S and not(RDE or RDW) and not SS)
        connect far-south

88.  if((RDN and S and not(RDE or RDW) and SS)
        connect far-south-south

89.  else if(RDE and not RDW and( not(E or SE)  or (E and SE)))
           connect south

90.  else if(RDE and not RDW and E and not SE)
           connect south-east

91.  else if(RDE and not RDW and SE and not E)
           connect south-west

92.  else if(RDW and not RDE and( not(W or SW)  or (W and SW)))
           connect south

93.  else if(RDW and not RDE and W and not SW)
           connect south-west

94.  else if(RDW and not RDE and SW and not W)
           connect south-east

95.  else if(RDW and RDE and not S)
           connect south

96.  else if(RDW and RDE and N)
           connect far-south

97.  else connect south
       break}

                        /* end SOUTH connection algorithm  */
```

6.3 *Fault Coverage*

6.3.1 Single and Double Faults

Theorem 4-1: An array equipped to implement algorithm 4-24 will tolerate all single and double fault occurrences.

Proof: An array equipped to use algorithm 4-24 will configure identically to an array equipped to use algorithm 4-12, as described in the previous chapter.

■

Theorem 4-2: An array equipped to implement algorithm 4-24 will tolerate all triple faults.

Proof: Three faults may occur in the following configurations:

1. 1-1-1
2. 1-2
   a. 1-2, two not adjacent
   b. 1-2, two in adjacent cells
3. 3
   a. all three adjacent
   b. two adjacent, one non-adjacent
   c. none in adjacent cells.

All triple faults are configured identically to the same fault conditions in an array equipped to implement algorithm 4-12 with the exception of the instances of case 3-2b not tolerated by algorithm 4-12. Figure 85 on page 222 shows one instance of that fault condition. With this fault pattern, assuming the faults are in cell[i,j], cell[i,j + 1] and cell[i + 1,j], at the end of the reconfiguration process cell[i-ℓ,j-1] | 0 < = ℓ < = i will have the same configuration as the same cells in case 2-3 of

algorithm 4-12. Cell[i-2-ℓ,j] | 0 < = ℓ < = i-2 , cell[i-2-&l.,j + 1] | 0 < = &l. < = i-2 and cell[i-ℓ,j + 2] | 0 < = ℓ < = i will also configure identically to case 2-3 in algorithm 4-12. Cell[i + 1 + ℓ,j-1] |0 < = ℓ, cell[i + 1 + ℓ,j + 1] |0 < = ℓ and cell[i + 3 + &l.,j] | 0 < = &l. will have the same configurations as the same cells in algorithm 4-12 case 4-4 as illustrated by Figure 76 on page 189, with the triple fault in that figure located in row[i + 1]. Only two cells will have different configurations. Cell[i-1,j] has a fault to its south(S) and farsouth(SS) and connects to cell[i + 2,j] as its southern neighbor. This places cell[i-1,j] in configuration 88. Cell[i + 2,j] has faults to its north(N) and farnorth(NN) and connects to cell[i-1,j] as its northern neighbor. This cell is in configuration 48.

This fault configuration can occur in four ways, the one just described and with the fault in cell[i + 1,j] shifted to either cell[i + 1,j + 1], cell[i-1,j] or cell[i-1,j + 1]. The reconfiguration process is similar in all four instances.

We have shown that an array equipped to implement algorithm 4-24 will tolerate all triple fault occurrences.

■

6.3.2 Four faults.

Four faults can be occur in the following configurations.

1. 1-1-1-1
2. 1-1-2
3. 2-2
4. 1-3
5. 4

Theorem 4-3: An array equipped to implement algorithm 4-24 will tolerate all four-fault configurations except instances of case 4-2, with adjacent faults in one row, and the single faults located

one to the north of the adjacent faults and one to the south, both single faults in the same column as one of the adjacent faults.

Proof: Case 4-1 with four single faults is handled identically to similar fault patterns in algorithm 2-10 and and 4-12. Case 4-2 is also identical to algorithm 4-12 with the exception of the instances in which either the three fault pattern described in the previous section or instances in which the two faults are adjacent and the two single faults lie one to the north and one to the south in the same columns.

Figure 86 on page 223 shows one configuration which can be tolerated with the triple fault pattern. In this example, the faults exist in cell[i,j], cell[i,j + 1], cell[i + 1,j] and cell[i-n,j + 1]. As in case 3-2b described previously, the row deformation to the north originally included both columns [i] and [i + 1]. When the deformation reached cell[i-n,j + 1], the blocker bits were generated and deformation in this column was reversed and sent to the south.

Cell[i-n-ℓ,j + m] | (ℓ < = n and -1 < = m < = 2) have the same configurations they would have had if cell[i-n,j + 1] were not faulty, with the exception of the state of the southern communications link on cell[i-n-1,j + 1], which now has a faulty cell to its south. In this algorithm, when a fault occurs in any column involving a northern deformation, the deformation bits, RDN, RDNE and RDNW are latched and not cleared as was the case in algorithm 4-12. This is also true in southern deformations in which only the RDNE or RDNW bits are set. This allows the state of cell[i-n-1,j + 1] to assume the previous state of the faulty cell[i-n,j + 1] and continue the deformation north of that fault, and in turn to allow the deformations in column[j] to remain in place while forcing the deformation of column[j + 1] south of cell[i-n,j] to be moved to the south.

Figure 87 on page 224 shows a similar fault pattern with the single fault in cell[i + n,j + 1]. In this instance, only column[j] originally had a deformation to the south. When cell[i + n,j + 1] became faulty, fault register bits RDN, RDS and RDNW were latched into cell[i + n + 1,j + 1], and the RDNW and RDS bits were latched in cell[i + n,j + 1].

It has been proven that an array equipped to implement algorithm 4-24 will tolerate all four-fault occurrences except instances in which two adjacent faults are accompanied by two single faults, one to the north of the adjacent faults and one to the south, and both the single faults are in the same column.

■

6.3.3 Five Faults.

Five faults can occur as:

1. 1-1-1-1-1
2. 1-1-1-2
3. 1-2-2
4. 1-1-3
5. 2-3
6. 1-4
7. 5

Theorem 4-4: An array equipped to implement algorithm 4-24 will tolerate all occurrences of five faults except those in which a row deformation is necessary, and which have faults north and south of the fault requiring deformation in same column. These may occur as cases 5-2, 5-3, 5-4 or 5-5.

Proof: Case 5-1 in algorithm 4-24 is identical to case 5-1 in algorithm 4-12. Figure 88 on page 225 shows one example of case 5-2 which can be tolerated by hardware supporting algorithm 4-24, but was not tolerated by algorithm 4-12. The row deformation around the faults in cell[i,j], cell[i,j + 1], cell[i-n,j + 1] and cell[i + m,j] is identical to the previously described example of four faults shown in Figure 86 on page 223 The value of 'm' and 'n' is not significant as long as they are positive values. The fifth fault may be located in any cell as long as it is not in columns containing the other four faults.

Figure 89 on page 226 shows an occurrence of case 5-3, with two faults in each of two rows, and the fifth fault in any row, as long as it is not in one of the columns containing the other four faults. If the four adjacent faults occur in cell[i,j], cell[i,j + 1], cell[i + 1,j] and cell[i + 1,j + 1], then the configurations of cell[i-n,j-1] | 0 < = n < = i, cell[i-n,j + 1] |0 < = n < = i, cell[i-n,j] | 2 < = n < = i and cell[i-n,j + 1] | 2 < = n < = i is identical to the state of the same cells in case 4-2, Figure 87 on page 224. Cell[i-1,j] and cell[i-1,j + 1] both have adjacent faults to their immediate south and have the same southern configuration and connection as cell[i-1,j] in the example of case 4-2 referenced above.

Cells to the south of the four adjacent faults have configurations which are the mirror image of those to the north of the faults. The fifth fault may occur in any row and in any column except column[j] and column[j + 1].

Figure 90 on page 227 and Figure 91 on page 228 show instances case 5-4, with three faults in one row and two single faults. This fault pattern can be tolerated if the two single faults are not located north and south of a fault requiring row deformation, with all three faults in the same column.

The number of occurrences of five faults tolerated by this algorithm makes it tedious process to prove all of them. Basically, an array with hardware to support algorithm 4-24 can tolerate any number of faults as long as there is a fault-free path to a spare cell.

■

6.4 Coverage Analysis

An array with algorithm 4-24 installed covers all single, double and triple fault occurrences. The only occurrence of four faults not covered is adjacent faults with single faults to the north and the

south in the same column as one of the adjacent faults. There are nine ways to get adjacent faults in one row, with the number of choices for the other two faults dependent upon in which row the adjacent faults occur. This yields $2\times9\times(2\times8 + 4\times7 + 6\times6 + 8\times5) = 2160$ four-fault occurrences which cannot be tolerated.

Five fault occurrences which cannot be reconfigured consist of the four fault configuration which cannot be tolerated or three faults in one row with the single faults north and south of one of the three faults which requires row deformation. To be conservative, there are $\binom{10}{3} = 120$ ways to choose 3 out of 10 cells in a row. The occurrence in which the three faults are all adjacent was covered in the above calculation, as well as the case in which two were adjacent and one not adjacent. This leaves $120\times2\times(7 + 12 + 15 + 16) = 12000$ ways to place the single faults north and south of the center of the three faults, or a total of $12000 + 205200 = 217{,}200$ five fault conditions which cannot be tolerated.

These data are shown in Table 6 on page 218 and probability of survival for different lengths of time compared to the array without algorithm 4-24 installed is given in Figure 81 on page 218. A plot of the information given in Table 6 is given in Figure 92 on page 229.

6.5 Cost of Implementation

The number of additional bits which must be transmitted by each cell to its logical neighbors to enable implementation of this algorithm is 42, nine single latch bits and 33 fault register bits. Logic estimates, again based on the simulation code, indicates the need of 149 'and' gates, 76 inverters and 169 'or' gates in the decoder, and 461 'and' gates, 276 'or' gates and 47 inverters in the fault register logic. Total new logic is approximately 1200 gates. This estimate is not considered to be as reliable

as the estimates in the previous algorithms because simulation was not as extensive on this algorithm as on the previous algorithms.

6.6 Time Complexity

In an 'm' row by 'n' column array, the worst-case reconfiguration would be the existence of two faults which were not adjacent on one side of the array, with a fault in the northern most row in the same column as the western fault. If a single fault occurred in the same row on the opposite boundary of the array, it would take approximately 'm' clock cycles for the information about the new fault to propagate sufficiently to begin the row deformation. Another 'n' cycles might be required to reach the point where the knowledge that the northern deformation is blocked and another 'n' cycles to deform to the south. Total time in a reconfiguration is m + 2n.

6.7 Embedding the Array Within a Processor/Switch Lattice

Implementation of algorithm 4-24 would require seven vertical channels and two horizontal channels in the layout. To decrease this number of conductors, investigation was done on methods of embedding this array into a processor-switch lattice, with a reduced number of vertical channels.

There are two major problems when using a switch lattice for the communications network. First a method of isolating the faulty cells must be found. In the scheme using direct-switched links, this was done by having all cells with links to the faulty cell disable that link by opening a switch. When

using a switch lattice, either the switches in the lattice which connect to the faulty cell must be opened or the power must be removed from the cell.

The second problem which must be solved is distributed control for the switches of the lattice to eliminate the single-point failure mode which is a feature of control schemes which use external controllers to set the state of the communications network.

Kumar[Kuma84b] solved these problems by making control of the switch-lattice in the same plane as the control cells for the computational plane. Faulty cells or switches were isolated by placing the control cells surrounding the faulty component in a quiescent state. In this state, a cell would not transfer any data from faulty components to fault-free components. A cell in the quiescent state could not become a member of an active array. This was accomplished by assigning an s-value to the quiescent cells of 0. Reconfiguration around faulty cells is accomplished by moving the active array to a fault-free area of the array.

The reconfiguration algorithms discussed in this and previous chapters provide mechanisms for isolating faulty cells in either the control or computation plane, but provide no control for the communications network. Zaidi's path growth algorithm provided a distributed control mechanism for the communications plane, but has a speed problem and does not address fault isolation.

The first method examined was an architecture which combines the control of the computation plane and the communications network into one cell. The cell in the control plane has control of the computation cell and the switches of the communications network in its area. To prevent the loss of the communications network in the vicinity of a faulty cell, switch control must be shared by at least two cells.

In this architecture, two serious problems exist. One is still the isolation of faulty cells. If control of the switches is limited to two cells, and one becomes faulty, the fault must still be isolated. This can be done by placing additional switches into the control lines from a cell, with these switches

under the control of neighboring cells, or by devising a method of disconnecting power from the faulty cell, thus eliminating all signals from the faulty cell. Either method requires a voting method be used for allowing the fault-free neighbors to remove power from the faulty cell or for disconnecting its control output. Because only the four physical neighbors of a cell would know of its faulty state, the voters must consist of these four cells. A majority voter would allow one of these cells to become faulty and still operate properly. If two of a cells neighbors become faulty, the voter would not be functional.

A substitute for the voter might be a simple 'or' function of the enable terms, except two faulty cells which were adjacent would be able to enable each other. Also, if a cells four neighbors become faulty, the cell would be disabled, leaving a block of five adjacent cells which were disabled, and therefore eliminating a large portion of the communications network. This would also take a fault configuration of four faults which is tolerated by all four reconfiguration algorithms discussed and turn it into a block of five faults which is not tolerated by any of the algorithms.

Therefore, if a distributed control function for the communications network is to be devised, the control of the communications cannot be combined with control of the computation function.

The control plane envisioned by Kumar[Kuma84b] can be used to control the communications network. This separate plane would have control of the communications network, and could be used to isolate faulty cells in either the control or computation plane. It was proven in [Kuma84b] that this architecture could control both the state of the functional cells and the state of the communications path. If provisions are made in the state pattern algorithm to treat a fault as a valid state, then when a fault occurred, the condition is communicated to the cells controlling the communication network. This change in state of one of the cells in the active array would then trigger a new wave of state changes which would leave the communications network in the proper configuration to allow the cells which do not control switches to shift operations.

An example of how this procedure would work is the single fault case. If a fault occurs in functional cell (as opposed to switch cell) it is detected by its four functional neighbors. Instead of changing the internal switch settings to achieve the proper connections for reconfiguration, the functional cells pass the fault information to cells which control the communications network. These cells would then change the switch settings in the switch lattice to make the proper connections to start the reconfiguration process. The fault information is then passed to the neighboring functional cells over the new network and they repeat the process of communicating the fault condition to switch control cells.

Another possibility for setting the communications network would be imbedding the testing algorithm within the switch control plane. This would allow the reconfiguration process to be achieved entirely in the switch plane. Switch control cells would start a state pattern growth process when they detected a fault which would then force the proper functional cells to change to the proper state for reconfiguration.

6.7.1 Reconfiguration Around Faulty Switches or Switch Control Cells

Reconfiguration around faulty switches can be accommodated by adding redundant switches and communications links to the switch lattice. However, the problem of faulty switches or switch control cells re-introduces the original problem with the architecture. Global reconfiguration is not possible because the functions controlled by adjacent cells within the control plane is not necessarily identical. Some cells control switches, while their neighbors control functional cells. The operation of a switch might be transferred to a functional cell, but the operations of a functional cell can not be passed to a switch.

To embed the functional control plane within a switch-lattice, local reconfiguration must be introduced into the communications network. The most simple scheme for implementing this would

be interstitial reconfiguration [Sing88], with spare cells placed within the network which are not used unless a neighboring cell or switch is found to be faulty. Figure 93 on page 230 shows a simple example of how this system would work. Shown on the left of the figure is a small section of a processor switch lattice with no redundancy. The control plane for this configuration is the 2-dimensional mesh shown below the lattice On the right side of the figure is a modified lattice with spare switches added to each pair of switches in the original lattice. The control plane for this architecture is shown below it. Control cells marked with an "S" control the spare switches. Their function in a fault- free system is to monitor the two neighboring control cells to detect a fault in either the control cells or their associated switches. If a fault occurs in a switch, its control cell will set a status bit which would notify its four neighbors and the spare cell of the condition. These five cells would then disconnect communications to both the faulty switch and to its control cell. Communications would be enabled to the spare cell and its switch. Figure 93 on page 230 shows enabled communications links in solid lines and disabled links in dashed lines. The right side of the figure shows the configuration of the lattice and its control cells after the reconfiguration process. The spare cell has taken on the fault-free state of the faulty cell, an connected to the same four cells to which the faulty cell was previously connected. The state pattern of the communication plane control cells is not changed, and global reconfiguration is not necessary.

Additional fault coverage can be handled by adding additional redundant cells. Singh[Sing88] shows schemes of redundancy from 100% to 25%.

Table 6. Coverage Analysis Results Algorithm 4-24

Faults	Possibile	Covered	Percentage
1	100	100	100
2	4950	4950	100
3	161,700	161700	100
4	3.92×10^6	3.92×10^6	99.9
5	7.53×10^7	7.51×10^7	99.7

64 Cell Active Array
2 Spare Columns 2 Spare Rows
lambda = 0.0001

t	W/O SPARES	W/SPARES	t	W/O SPARES	W/SPARES
0	1	1	10	0.938005	1
20	0.879853	1	30	0.825307	0.999999
40	0.774142	0.999996	50	0.726149	0.999986
60	0.681131	0.999963	70	0.638905	0.999916
80	0.599296	0.99983	90	0.562142	0.999686
100	0.527292	0.999459	110	0.494603	0.999123
120	0.46394	0.998643	130	0.435178	0.997986
140	0.408199	0.997112	150	0.382893	0.995981
160	0.359155	0.994551	170	0.33689	0.992782
180	0.316004	0.99063	190	0.296413	0.988056
200	0.278037	0.985021	210	0.2608	0.981488
220	0.244632	0.977425	230	0.229466	0.972801
240	0.21524	0.967591	250	0.201897	0.961772

Figure 81. Algorithm 4-12 Survivability Probability.

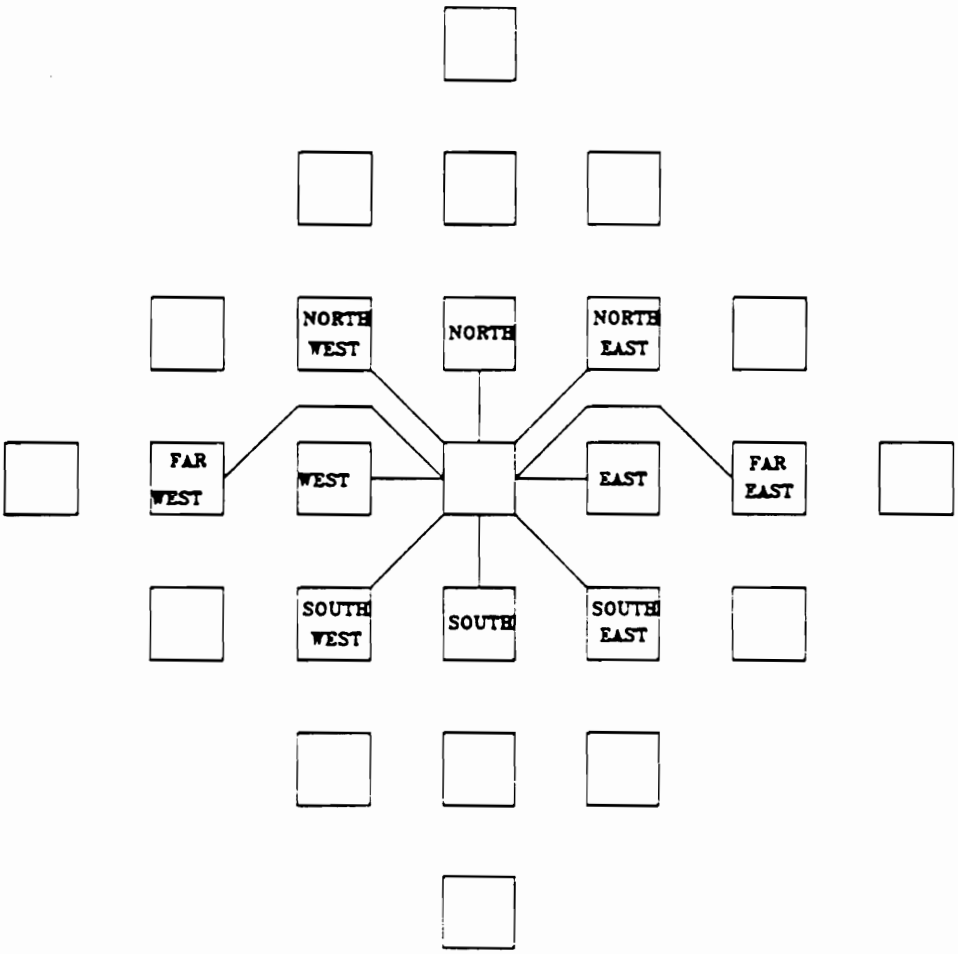


Figure 82. 24 Neighbor Interconnection: Cell[i,j] and its possible neighbors.

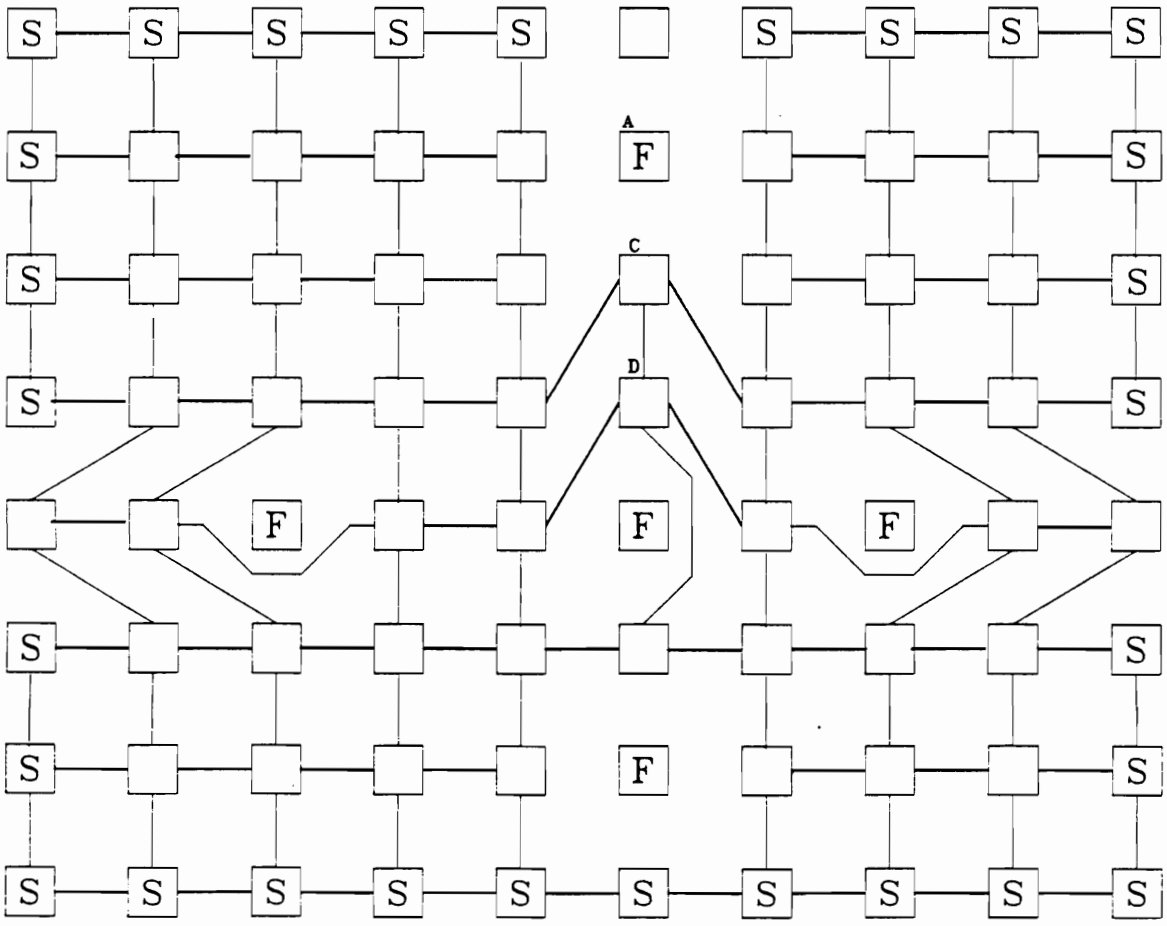


Figure 83. Example of unrecoverable pattern.

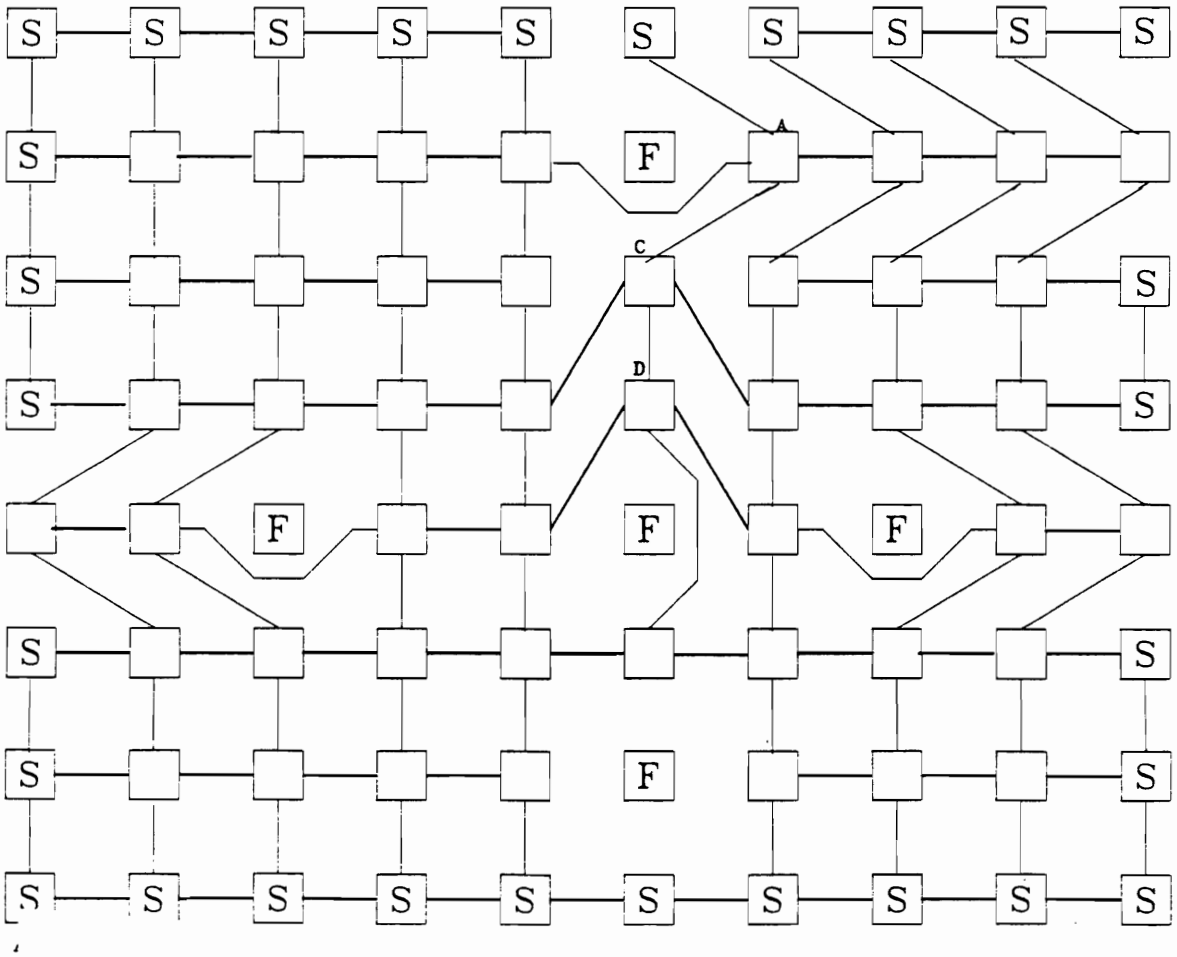


Figure 84. Example of unrecoverable pattern.

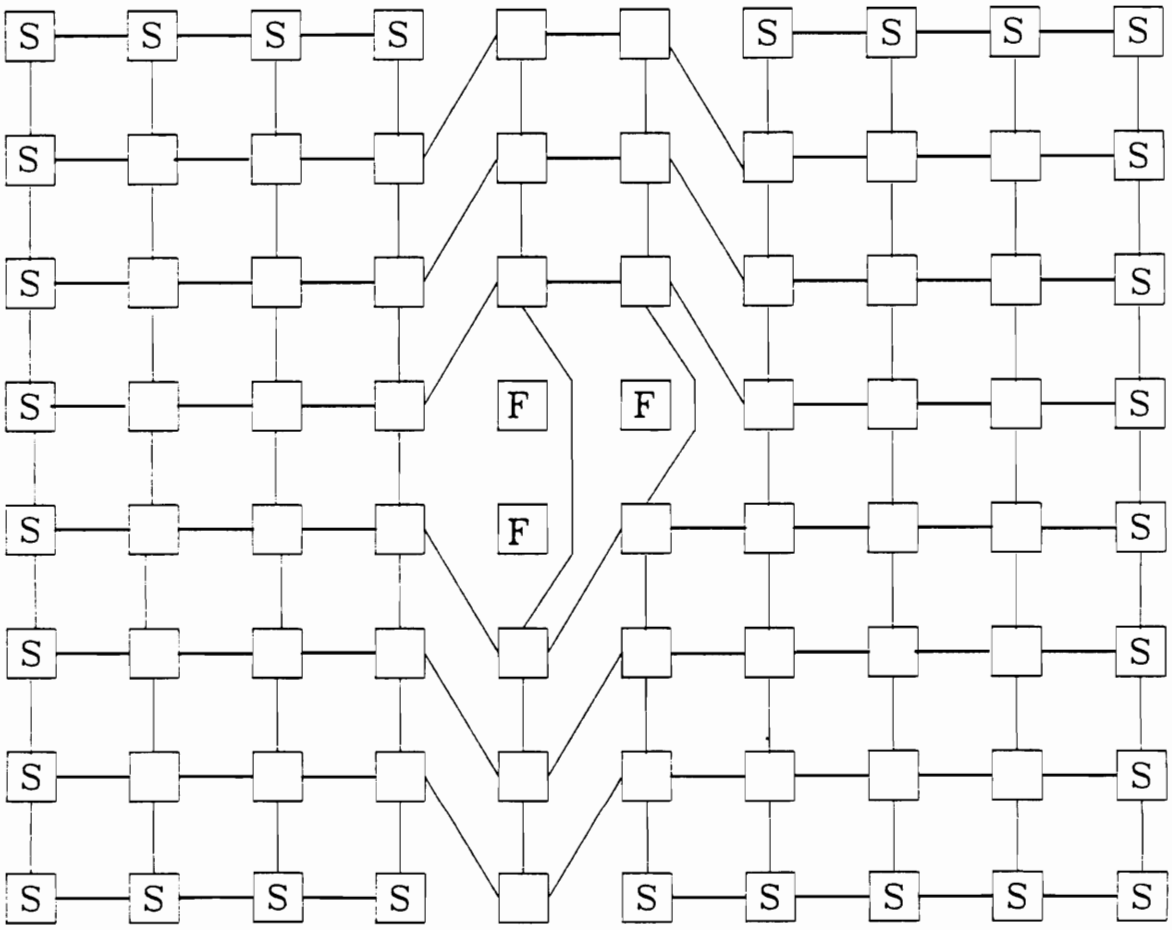


Figure 85. Three Faults Case 3-2: Array with two faults in one row with an adjacent cell faulty in another row.

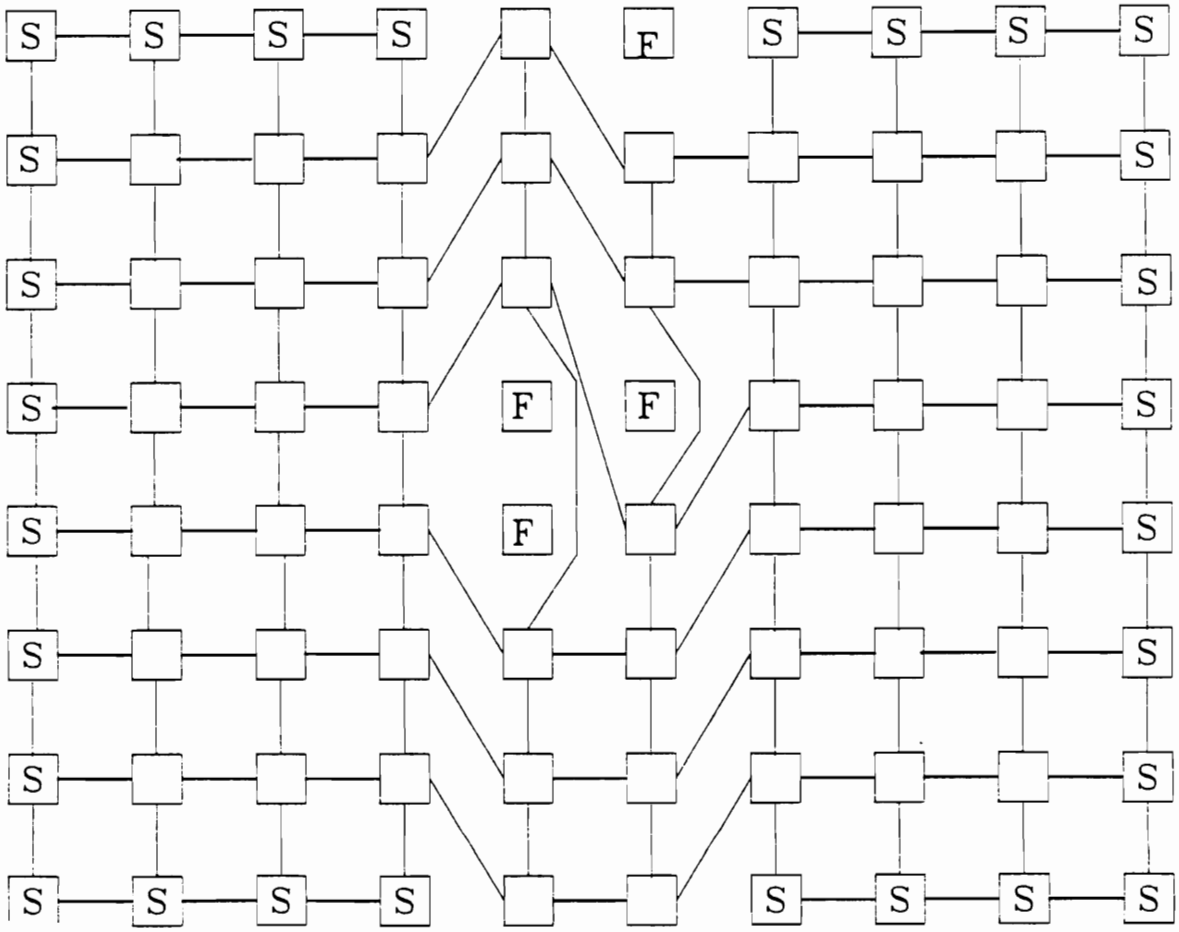


Figure 86. Four Faults Case 4-2: Array with two faults in one row with an adjacent cell faulty in another row.

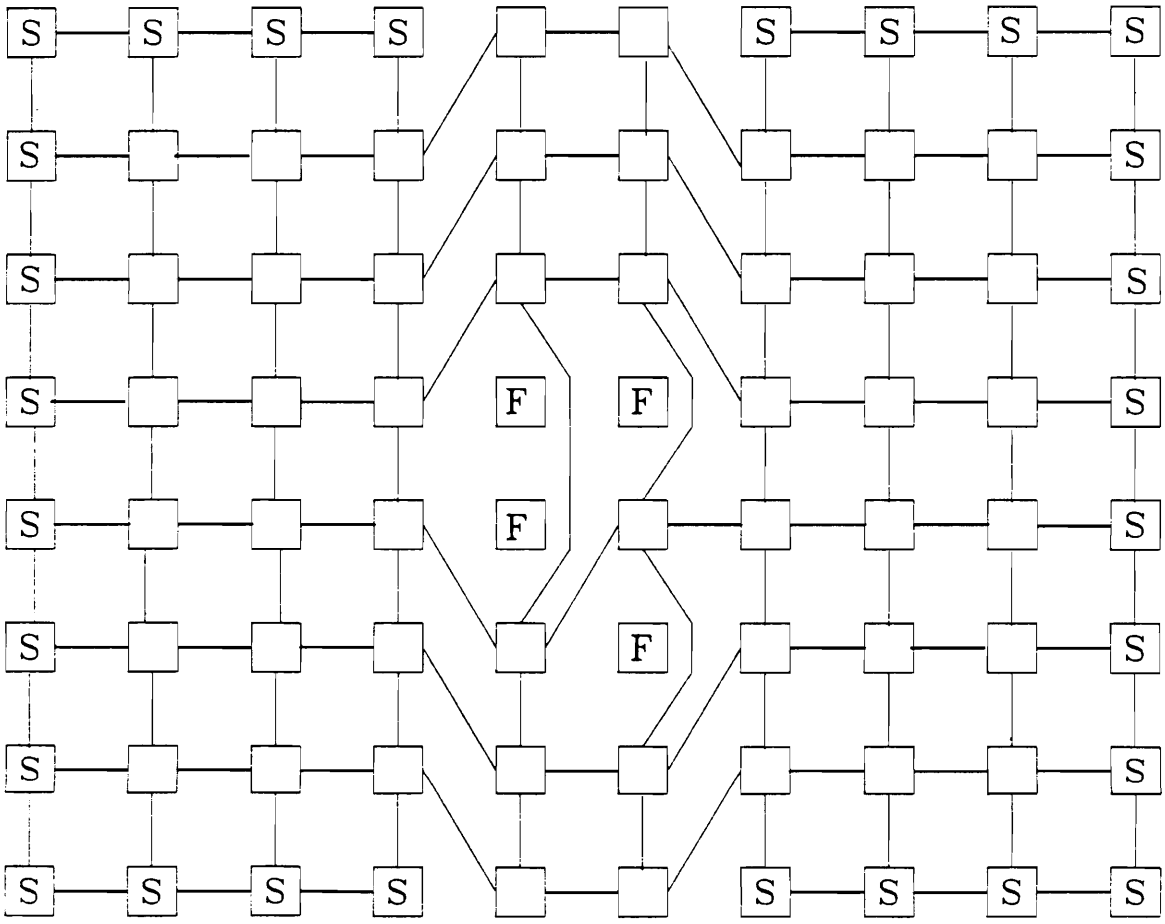


Figure 87. Four Faults Case 4-2: Array with two faults in one row with an adjacent cell faulty in another row.

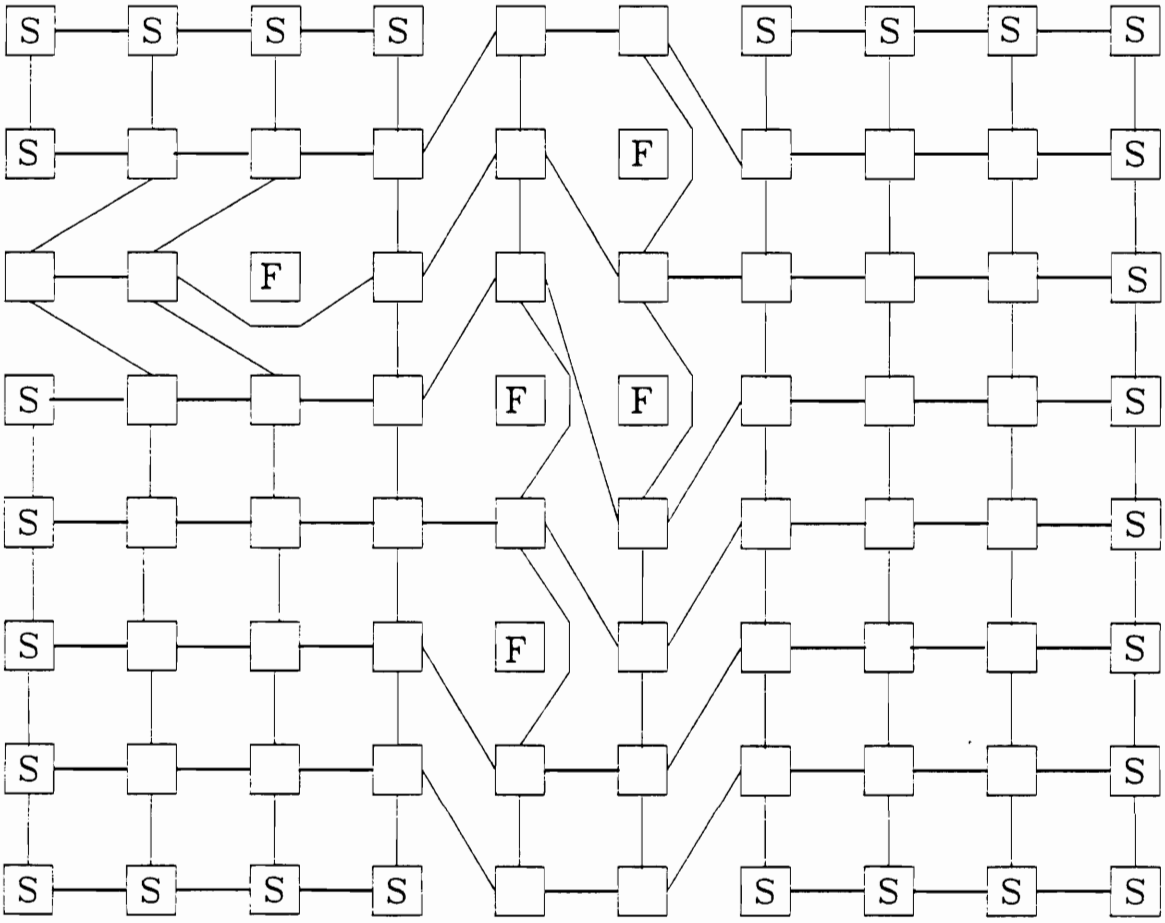


Figure 88. Five Faults Case 5-2: Array with two faults in adjacent cells and a fault to the north and south in the same columns.

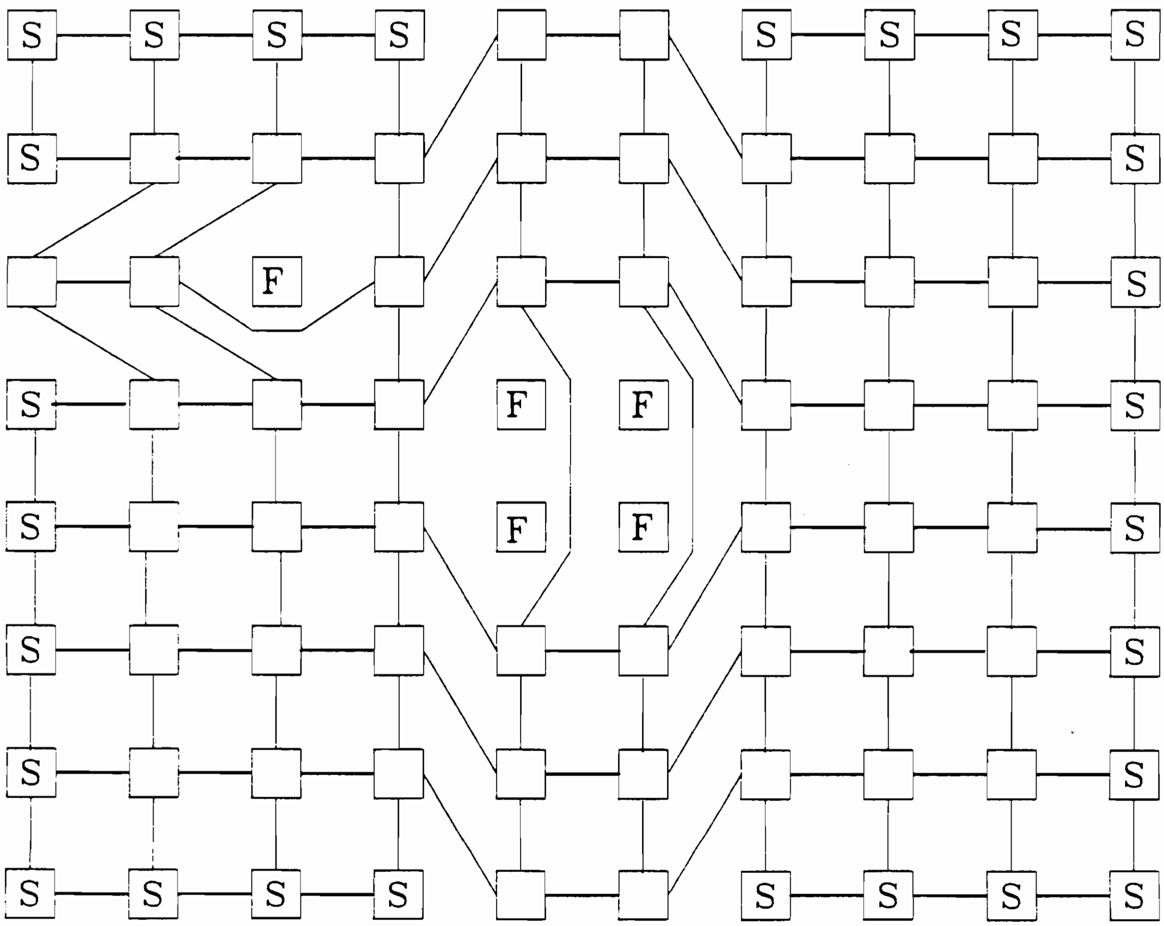


Figure 89. Five Faults Case 5-5: Array with two faults in two rows and a single fault.

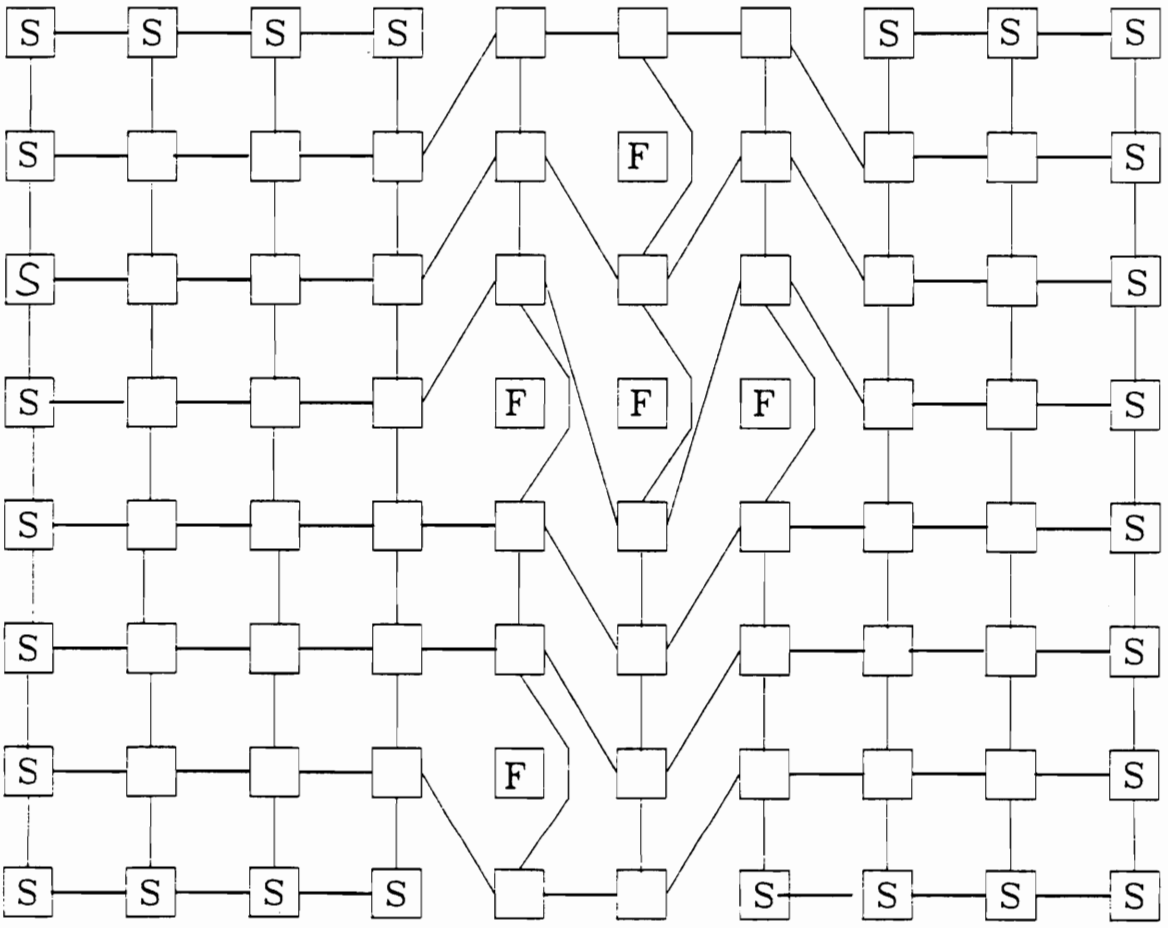


Figure 90. Five Faults Case 5-4: Array with three faults in one row and two single faults.

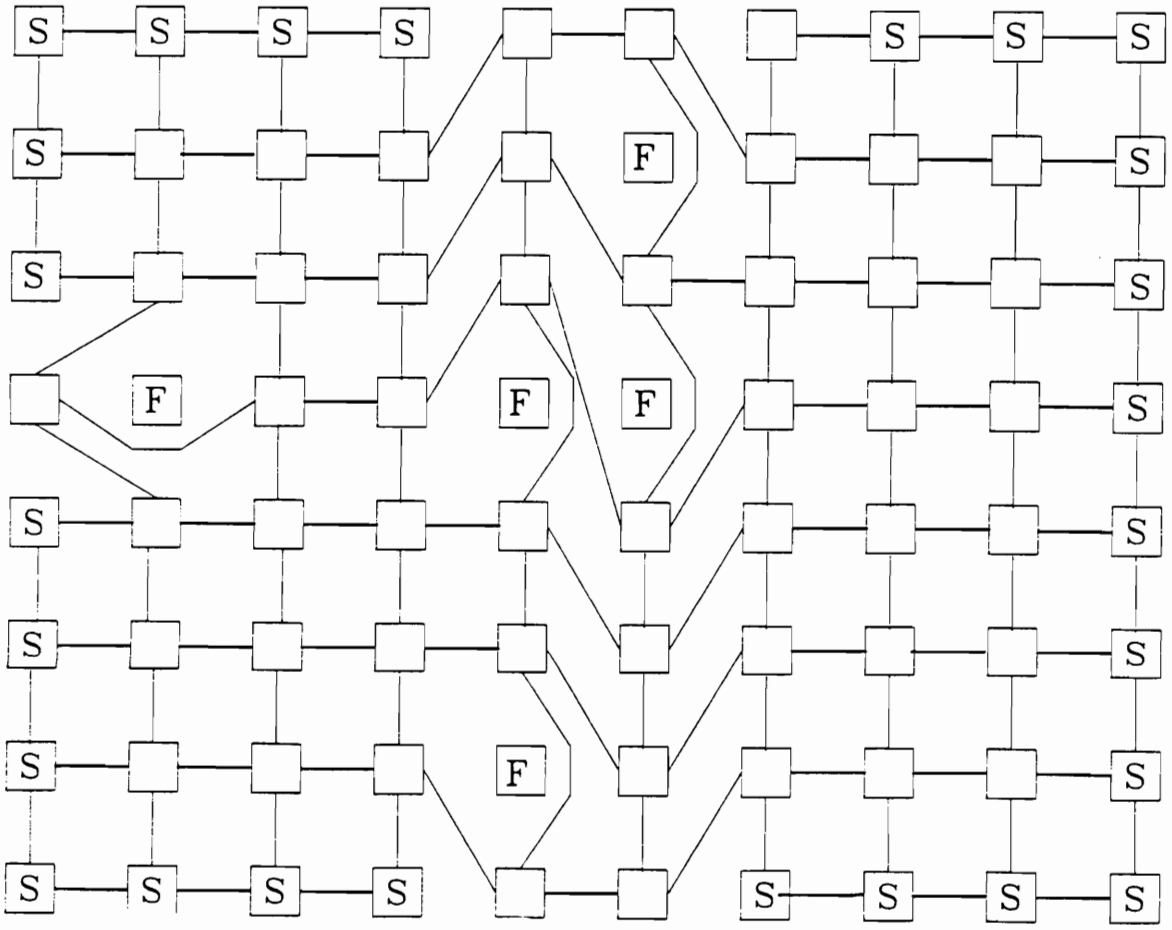


Figure 91. Five Faults Case 5-4: Array with three faults in one row and two single faults.

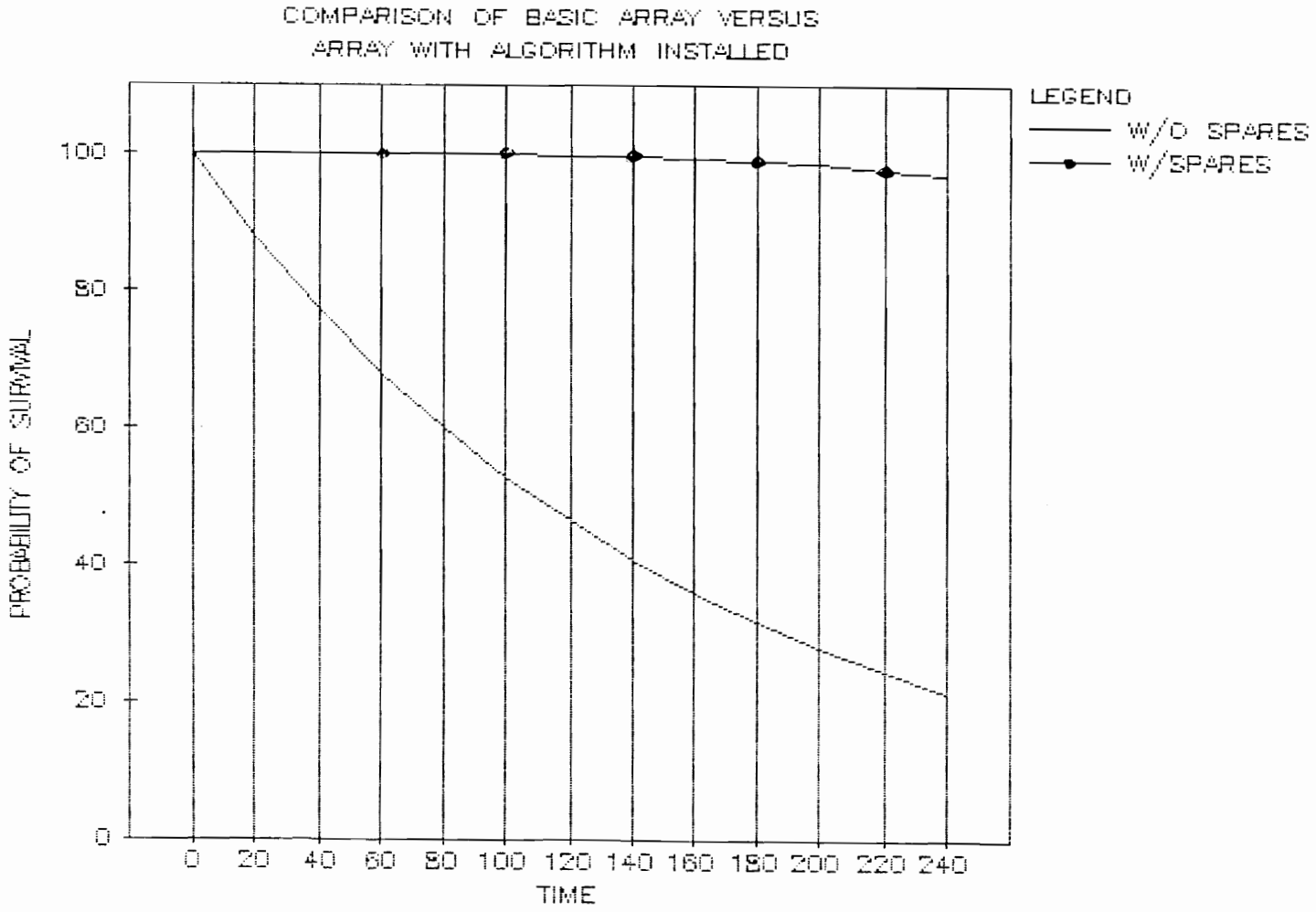


Figure 92. Plot of Probability of Survival vs Time

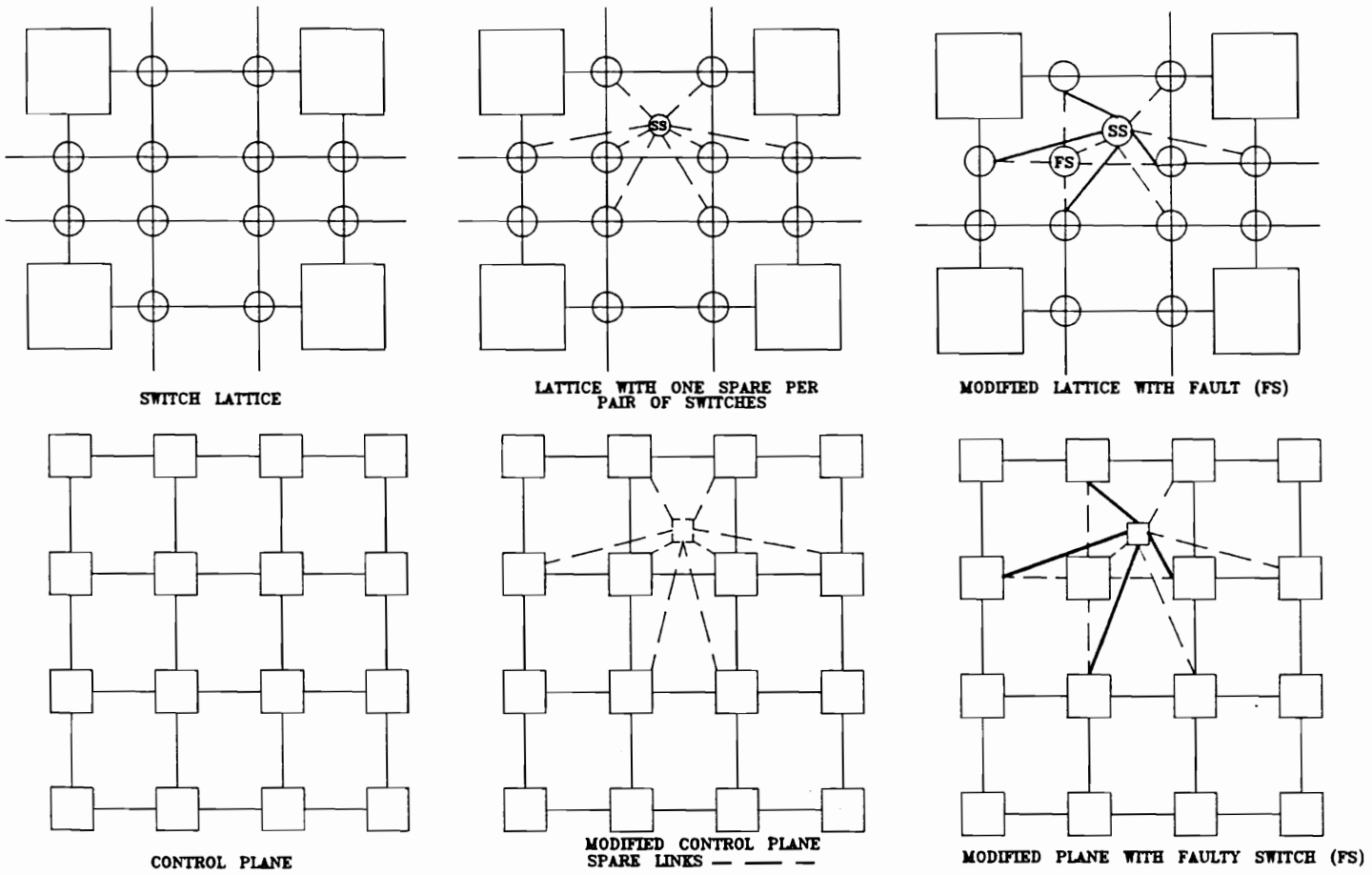


Figure 93. Switch Lattice: Normal Lattice, w/Local spares, w/Faulty Switch

7.0 Simulation

The stated results for this algorithm, and algorithm 4-12 and 4-24 were verified with a program written in the "C" language. The simulator is broken into ten files.

1. algor_x.c This file contains the code to simulate the reconfiguration algorithm. This code determines which of the communications links is enabled by each cell in the array.

2. cell.c This file contains the code to simulate the high-level control of each cell. This subroutine takes the input from sim_x.c that consists of the cell number and the clock cycle being simulated, and makes the calls to the appropriate function to be executed during this particular cycle.

3. faultrg_x.c This file contains the code for the logic which is used to set bits in the fault register and the auxiliary bits, which are simple latches, and not two phased latches as is the fault register.

4. graphics.c The code in this subroutine runs at the end of each major clock cycle and at the end of simulation. It displays on the console of a PC the communications links which are

enabled and the cells of the array. Faulty cells are displayed as filled white circles and fault-free cells as unfilled circles.

5. init.c This code runs on the initial pass and clears all cell parameters.

6. newstate.c This file would represent the code necessary to simulate the state pattern growth algorithm. To speed the simulation of the reconfiguration process, this is an empty file at present and the growth algorithm call is disabled.

7. newsval.c This file contains code which computes the s-value of each cell according to Kumar's algorithm. This code is functional but is not called to speed simulation.

8. transmit.c This file contains code which simulates the transmission of data between cells. Simulated transmission is accomplished by writing the output data into an array called cell_output. The number of elements in this array is 24 to allow the code to be used in all algorithms. In algorithm 2-10, only output values 0 thru 9 are used. The definitions of the output array elements is at the top of the file transmit.c. Elements are assigned 0 thru 23 to represent the communications link to a particular neighbor. Assignments are made to insure that the sum of the array element used to transmit data by one cell to another is 23 -(the array element used to receive data by the other cell). Example, the connection between cell[i,j] and cell[i,j-1] is '0' in cell[i,j] and '23' in cell[i,j-1]. If data is not written into an output element, it is cleared.

9. sim_x.c This is the main program. It simulates operation of the master clock and transmits cell numbers and clock cycles to cell.c.

10. receive.c This code simulates the reception of data by each cell. The code examines the values of the communications links to the north, south, east and west and writes the data from the appropriate output array into an input register. For example, if the north link is specified to

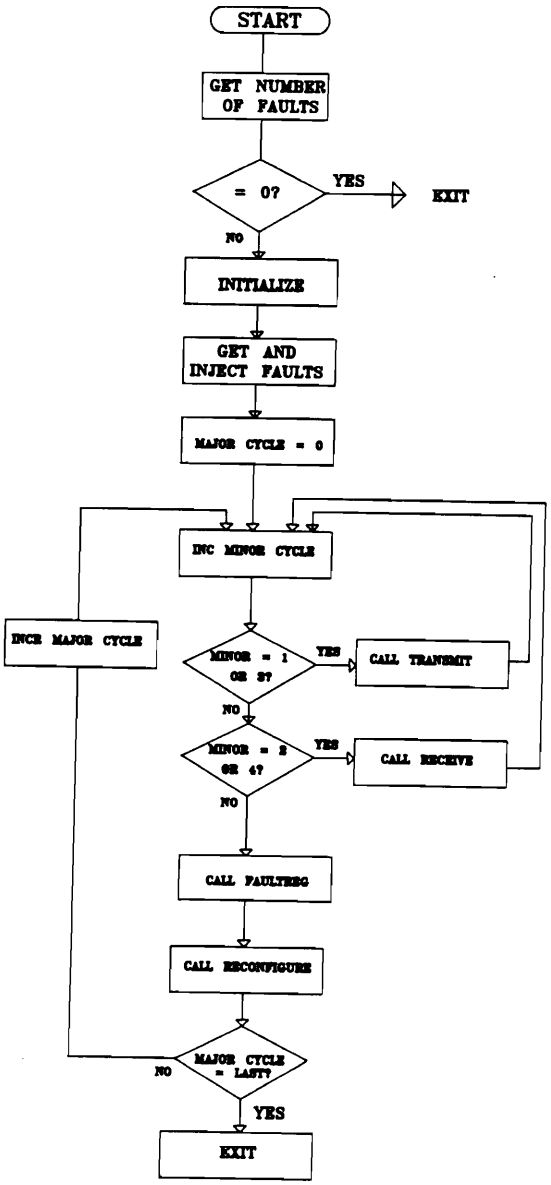
be north-east, which is assigned the logical element 4 in the output array, receive.c will take the contents of cell_output[cell_number - 9][19] and write it into cell[i,j]'s input registers.

Simulation is done assuming all faults occur simultaneously. This creates the most difficulty in transmitting fault data to neighboring cells. The main program, sim_x, cycles through the minor clock cycle and updates the term major_clock_cycle each complete pass through the minor clock cycle chain. For each minor clock cycle, a call is issued to subroutine 'cell' which takes the input clock cycle, determines which function should be called, and issues the call. In minor cycle 1, transmit to the north and west is done, and receive from the south and east. Cycle 2 causes transmit south and east and receive from north and west. Cycle 3 is a compute s-value cycle. Cycle 4 is used to transmit fault register and state data to the north and west and receive the same from the south and east and cycle 5 causes data to be transmitted to the south and east and to be received from the north and west. Cycle 6 is the compute-new-state, which is not coded in this simulator, and reconfigure cycle.

Faults are injected by reading a file called 'simulation.in' which contains the number of faults to be injected and their cell numbers, 0 thru 99, on each line. The main program reads one line of this file, initializes all variables in the simulation, and then sets the state of cells in which faults are to be injected to 0xf. Any cell receiving a state value of 0xf from a neighbor will set the bit in its fault register which corresponds to that cell being faulty. Normal state conditions are that all cells have a state of '1'. If a cell receives any state other than 0xf or '1', it will ignore the input and determine that the other cell is not connected to the same link.

The simulator is set to run for a period of 25 major clock cycles and then halt, displaying a graphical representation of the state of the array on the personal computer screen.

Source code for each of the simulator files is included in Appendix A. Figure 94 on page 234 shows a flow-chart of the simulator.



Simulator Flow Chart.

8.0 Conclusions and Future Research Opportunities

Reconfiguration algorithms offering various advantages and costs have been provided for the control cells in Kumar's [Kumar84b] architecture which determine the state of the execution cells. However, these algorithms assume separate control of the communications network. One possible method of implementing local reconfiguration into the control of the communications network was presented, although it was not proven that it would function as desired.

In using these algorithms, the problems associated with other reconfiguration schemes are eliminated. Others assume that the communications section of a faulty cell is always fault-free[Royc90]. Faulty cells are isolated in the algorithms presented in this work, and no part of the faulty cell is assigned any duty. Control of these reconfigurations is distributed, and control has been confirmed with simulation. Other algorithms have claimed to be distributable, but are described as if the control is centralized [Royc90] [Chea90]. Speed of the systolic array is maintained by implementing the algorithms with direct connections, rather than switch lattices as in [Patr89] or large switch networks. This allows each connection to be minimum length. [Yann86] cannot tolerate concurrent faults, which presents no problem to hardware implementing these algorithms, and the final configuration of the array depends only upon the fault pattern, and not on the order in which the faults occurred. Although these algorithms do not achieve the fault-tolerance of a "FUSS" type

algorithm [Chea90], they are much more simple to implement, and achieve their results without resorting to extremely long communications paths.

8.1 Open Problems

Although not a problem, investigation into allowing reconfiguration on a diagonal should be made. A great amount of logic was added to force the final configuration of the array to depend only upon the final fault pattern, and not upon the order in which the faults occurred. Some advantages may be gained if the algorithms are changed to allow the algorithms to take into account the current configuration of the array when configuring around a new fault. An example of this is in the final two algorithms presented. If a fault exists west of a new column deformation, the algorithms force the column deformation to reverse itself. Logic could be reduced if the row deformation is reversed when a column deformation is encountered. It may also be possible to gain some advantages in algorithm 4-24 by allowing configuration on the diagonal when a row deformation encounters a column deformation that is stable. Connections exist in the 24-cell neighborhood that could support the diagonal reconfiguration, but this would change the final configuration based on order of faults.

Future research should investigate the possibility of implementing a control scheme for the communications network which offers local reconfiguration. Work can also be done investigating new methods of implementing the global reconfiguration algorithms presented here, with emphasis on reducing the amount of logic needed to implement these algorithms. No effort was made in this research to use minimum logic to execute the algorithms.

Another research project should also include the capabilities of real-time reconfiguration. The high speed with which these arrays handle data makes it important to recover from faulty conditions as quickly as possible, and to recover without losing data already generated.

9.0 Bibliography

[Abar87] J. A. Abraham, P. Banerjee, C.Y. Chen, W. K. Fuchs, S. Y. Kuo, A. L. Reddy, "Fault Tolerance Techniques for Systolic Arrays," Computer, Vol. 20, No. 7, pp. 65-75, July 1987.

[Baer84] J. L. Baer, "Computer Architecture" Computer, Vol. 17, No. 10, pp. 77-86, October 1984.

[Brig87] B. A. Brighton, Improved Pattern Growth and Reconfiguration Methods for a Fault-Tolerant Cellular Architecture, Masters Thesis, Virginia Polytechnic Institute and State University, August, 1987.

[Chea89] M. Chean and J.A.B. Fortes, "FUSS: A Reconfiguration Scheme for Fault Tolerant Processor Arrays," Abstracts of International Workshop on Hardware Fault Tolerance in Multiprocessors, pp. 30-32, June 19-20, 1989.

[Chea90] M. Chean and J.A.B. Fortes, "The Full-Use-of-Suitable-Spares (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerant Processor Arrays," IEEE Trans. on Computers, Vol. 39, No. 4, pp. 564-571, April 1990.

[Chun83] F.R.K Chung, F.T. Leighton and A.L Rosenberg, "Diogenes: A Methodology for Designing Fault-Tolerant VLSI Processor Arrays," FTCS Digest of Papers, Milan, Italy, pp. 26-32, June, 1983.

[Cimi87] L. Cinimiera, C. Demartini and A. Valenzano, "Defect-Tolerant Array Structures for VLSI and WSI Architectures," Proc. 20th Annual Hawaii Int. Conf. on System Sciences, pp. 21-30, 1987.

[Conn85] K. Connel, I/O Algorithm and a Test Algorithm for a Reconfigurable Cellular Array Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, June, 1985.

[Degr87] A.J. DeGroot, E.M. Hohansson, J.P. Fitch and C.W. Grant, "SPRINT- The Systolic Processor with a Reconfigurable Interconnection Network of Transputers," IEEE Trans. on Nuclear Science, Vol. 34, No. 4, pp. 873-877, August 1897.

[Dist89] F. Distante, M.G. Sami, R. Steffanelli, "Reconfiguration Techniques in the Presence of Faulty Interconnections," Proc. of the International Conf. on Wafer Scale Integration, San Francisco, California, pp. 379-388, January 1989.

[Doni84] V.N. Doniants, S. Lori, M. Pellegrino, E. I. Pi'il, and R. Stefanelli, "Fault-Tolerant Reconfigurable Processing Arrays Using Bi-Directional Switches," Microprocessing and Microprogramming, Vol. 14, pp. 109-115, October-November 1984.

[Evan86] R.A. Evans, J.V. McCanny, and K.W. Wood, "Wafer Scale Integration Based on Self Organisation," Wafer Scale Integration, IOP Publishing Limited, Bristol, England, pp. 101-112, 1986.

[Feng81] T.Y. Feng, "A Survey of Interconnection Networks," Computer, Vol. 14, No. 12, pp. 12-27, January 1981.

[Fort84] J.A.B. Fortes and C.S. Raghavendra, "Dynamically Reconfigurable Fault-Tolerant Array Processor," Proc. of the 9th Annual Symp. on Computer Architecture, May 1982.

[Fort85] J.A.B. Fortes and C.S. Raghavendra, "Gracefully Degradable Processor Arrays," IEEE Trans. on Computers, Vol. 34, No. 11, pp. 1033-1044, November 1985.

[Fous87] D. E. Fousler and R. Schreiber, "The SAXPY-1, A General-Purpose Systolic Computer," Computer, Vol. 20, No. 7, pp. 35-43, July 1987.

[Gent84] G. Gentile, M.G. Sami and M. Terzoli, "Design of Switches for Self-Reconfiguring VLSI Array Structures," Microprocessing and Microprogramming, Vol. 14, pp. 99-108, 1984.

[Goll84] N. Gollakota and F.G. Gray, "Reconfigurable Cellular Architecture," Proc. of the 1984 Conf. on Parallel Processing, Bellaire, Michigan, pp. 472-483, August 21-24, 1984.

[Hwan85] K. Hwang, "Multiprocessor Supercomputers For Scientific/Engineering Applications," Computer, Vol. 18, No. 6, pp. 57-73, June 1985.

[Hoss89] S.H.Hosseini, "On Fault Tolerant Structure, Distributed Fault Diagnosis, Reconfiguration, and Recovery of the Array Processor," IEEE Trans. on Computers, Vol. 38, No. 6, pp. 932-942, July, 1989.

[Jerv88] L. Jervis, F. Lombardi, and D. Sciuto, "Orthogonal mapping: A Reconfiguration Strategy for fault tolerant VLSI/WSI 2D arrays," Proc. Intl. Workshop on Defect Tolerance in VLSI Systems, October 1988.

[Kim89] J.H. Kim and S.M. Reddy, "On the Design of Fault Tolerant Two Dimensional Systolic Arrays for Yield Enhancement," IEEE Trans. on Computers, Vol. 38, No. 4, pp. 515-525, April 1989.

[Kore86] I. Koren and D.K. Pradhan, "Yield and Performance Enhancement through Redundancy in VLSI and WSI Multiprocesor Systems," Proc. of the IEEE, Vol. 74, No. 5, pp. 699-711, 1986.

[Kore87] I. Koren and I. Pomeranz, "Distributed Structuring of Processor Arrays in the Presence of Faulty Processors," Systolic Arrays, IOP Publishing Limited, Bristol, England, pp. 239-248, 1987.

[Kuma84a] R. Kumar and F.Gail Gray, "A Fault Tolerant One-Dimensional Cellular Structure," Int. Conf. on Distributed Computing Systems, San Francisco, California, May 14-18, 1984.

[Kuma84b] R. Kumar, A Fault-Tolerant Cellular Architecture, Ph.D Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, July 1984.

[Kung82] H.T. Kung, "Why Systolic Architectures?," Computer, Vol. 15, No. 1, pp. 37-46, January 1982.

[Kung86] S.Y. Kung, C.W. Chang and C.W. Jen, "Real-Time Reconfiguration for Fault-Tolerant VLSI Array Processors, Proc. Real-Time Syst. Symp, pp. 46-64, December 1986.

[Kung87] S.Y. Kung, C.W. Chang and C.W. Jen, "Fault-Tolerance Design in Real-Time VLSI Array Processors," Proc. 1987 Intl. Phoenix Conf. on Comput. and Comm., pp. 110-115, 1987.

[Kung89] S.Y. Kung, S.N. Jean, C.W. Chang, "Fault Tolerant Array Processors Using Single Track Switches," IEEE Trans. on Computers, Vol. 38, No. 4, pp. 501-514, April 1988.

[Leig85] T. Leighton and C.E. Leiserson, "Wafer Scale Integration of Systolic Arrays," IEEE Trans. on Computers, Vol. 34, No. 5, pp. 448-462, May 1985.

[Li89] H.F. Li, R. Jayakumar, and C. Lam, "Restructuring for Fault Tolerant Systolic Arrays," IEEE Trans. on Computers, Vol. 38, No. 2, pp. 307-311, February 1989.

[Lomb87] F. Lombardi, R. Negrini, M.G. Sami and R. Stefanelli, "Reconfiguration Of VLSI Arrays: A Covering Approach," Proc. 17th Int. Symp. on Fault Tolerant Computing, pp. 251-256, July 1987.

[Mart80] H.L. Martin, "A Self-Reconfigurable Cellular Structure," Ph.D Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1980.

[Mead80] C.Mead and L.Conway, Introduction to VLSI Systems, Reading, Ma., Addison-Wesley, 1980.

[Moor85] W.R. Moore and R. Mahat, "Fault-Tolerant Communications for Wafer-Scale Integration of a Processor Array," Microelectronics and Reliability, Vol. 25, No. 2, pp. 291-294, 1985.

[Negr85a] R. Negrini, M. Sami and R. Stefanelli, "Fault-Tolerance Approaches for VLSI/WSI Arrays," Proc. IEEE Conf. on Comp. and Comm. Phoenix, Az. pp. 460-468, 1985.

[Negr85b] R. Negrini and R. Stefanelli, "Time Redundancy in WSI Array of Processing Elements," Proc. Intl. Conf. Supercomputing Systems, pp. 429-438, December 1985.

[Negr85c] R. Negrini, and R. Stefanelli, "Algorithms for Self Reconfiguration of Wafer-Scale Regular Arrays," Proc. Intl. Conf. on Circuits and Systems, (ICCas-85), pp. 190-196, October 1985.

[Negr86] R. Negrini, M. Sami and R. Stefanelli, "Fault Tolerance Techniques for Array Structures Used in Supercomputing, " Computer, Vol. 19, No. 2, pp. 78-87, February 1986.

[Patr89] J.L. Patry and G. Saucier, "Design of an Universal Switching Network for Reconfigurable 2D Arrays on Silicon," Abstracts of Int. Workshop on Hardware Fault Tolerance in Multiprocessors, Urbana, Illinois, pp. 33-41, June 19-20, 1989.

[Popl88] S.P. Popli and M.A. Bayoumi, "A Reconfigurable VLSI Array for Reliability and Yield Enhancement," Proc. of Int. Conf. on Systolic Arrays, San Diego, California, pp. 631-642, May 1988.

[Rose83] A.L. Rosenberg, "The Diogenes Approach to Testable Fault Tolerant Arrays of Processors," IEEE Trans. on Computers, Vol. 32, No. 10, pp. 902-910, October 1983.

[Royc90] V.P.Roychowdhury, J. Burck and T. Kailath, "Efficient Algorithms for Reconfiguration in VLSI/SWI Arrays," IEEE Trans. on Computers, Vol. 39, No. 4, pp. 480-489, April, 1990.

[Sami83] M. Sami, "Reconfigurable Architectures For VLSI Processing Arrays," Proc National Comp. Conf. AFIPS Los Angeles, CA. 1983, pp. 567-577.

[Sami84] M.G. Sami and R. Stefanelli, "Fault Tolerance of VLSI Processing Arrays: the Time-Redundancy Approach," Proc. Real-Time Systems Symp., pp. 200-207, 1984.

[Sami85] M. Sami and R. Stefanelli, "Fault-Tolerance of VLSI Array Structures," Proc. Intl. Conf. on Circuits and Systems, pp. 205-210, June 1985.

[Sami86a] M.G. Sami and R. Stefanelli, "Fault-tolerance and functional reconfiguration in VLSI proessing arrays," Proc. ISCAS86, 1986, pp. 643-648.

[Sami86b] M. Sami and R. Stefanelli, "Reconfigurable Architectures for VLSI Processing Arrays," Proc. of the IEEE, Vol. 74, No. 5, pp. 712-722, 1986.

[Sing88] A.D. Singh, "Interstitial Redundancy: An Area Efficient Fault Tolerant Scheme for Large Area VLSI Arrays," IEEE Trans. on Computers, Vol. 37, No. 11, pp. 1398-1410, November 1988.

[Snyd82] L.Snyder, "Introduction to the Configurable, Highly Parallel Computer," Computer, Vol. 15, No. 1, pp. 47-64, January 1982.

[Ston87] H.S. Stone, High-Performance Computer Architecture, Reading, Ma., Addison-Wesley, 1987.

[Uyar88] M.U. Uyar and A.P. Reeves, "Dynamic Fault Reconfiguration in a Mesh Connected MIMD Environment," IEEE Trans. on Computers, Vol. 37, No. 10, pp. 1191-1205, October 1988.

[Wang89] M. Wang, "Reconfiguration of VLSI/WSI Mesh Array Processors with Two Level Redundancy," IEEE Trans. on Computers, Vol. 38, NO. 4, pp. 547-554, April 1989.

[West85] N. Weste and K. Eshraghian, Principles of CMOS VLSI Design, A System Perspective, Reading, Ma. Addison-Wesley Publishing Co., 1985.

[Whit88] T.S. White, "A Distributed Control Reconfiguration Algorithm for 2-Dimensional Mesh Architectures which Tolerates Single Faults per Row," Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1988.

[Whit89] T.S. White, and F.G. Gray, "Summary of a Distributed Control Algorithm for a Dynamically Reconfigurable Array Architecture," Proc. of the Int. Conf. on Wafer Scale Integration, San Francisco, California, pp. 131-140, January, 1989.

[Yann86] R.M. Yanney and J.P. Hayes, "Distributed Recovery in Fault-Tolerant Multiprocessor Networks," IEEE Trans. on Computers, Vol. 35, No. 10, pp.871-879, October 1986.

[Yala85] S. Yalamanchili and J.K. Aggarwal, "Reconfiguration Strategies for Parallel Architectures," Computer, Vol. 18, No. 12, pp. 44-61, December 1985.

Appendix A. Simulation Source Code

A.1 Simulate.c

```
/*****************************************************************
                File  simulate.c
This is a generic file, some of the variables are not used
in all algorithms.
This is the main program of the simulation.  This file
generates the clock pulses which trigger the cells
*****************************************************************/
#include  < graphics.h >
#include  < stdlib.h >
#include  < conio.h >
#include  < stdio.h >
FILE *output_file
FILE *fopen()
FILE *input_file
/*************************************************
GLOBAL VARIABLE DEFINITIONS
*************************************************/
int north_link[100],          /* connection to northern neighbor*/
 south_link[100],
 east_link[100],
 west_link[100],
 row_deformation_north_blocker[100],
 row_deformation_northeast_blocker[100],
 row_deformation_northwest_blocker[100],
 row_deformation_northwest_blocker[100],
 row_deformation_east[100][2],
 row_deformation_west[100][2],
 clock_cycle,
 cell_number,
 s_value[100],
 state[100],
 connection_valid[100][24], /* used to disable connections to
                               non existant cells   */
 north_west_is_faulty[100], /* immediate neighbor is faulty */
 south_west_is_faulty[100],
 south_east_is_faulty[100],
 north_east_is_faulty[100],
 east_is_faulty[100],west_is_faulty[100],
 fareast_is_faulty[100],
 farwest_is_faulty[100],
 s_value_register[100][4]
```

```
lnpgt_from_north[100][2],
input_from_east[100][2],
input_from_south[100][2],
cell_output[100][24][2],
input_from_west[100][2],
fault_register[100],
fault_register_delayed[100]
/*********************************************
FUNCTION PROTOTYPE DEFINITIONS
*********************************************/
extern void run_cell(int cell_number, int clock_cycle)
extern void initialize(int cell_number)
extern void reconfigure(int cell_number)
extern void transmit(int cycle, int operation, int cell_number)
  /* cycle = 1 or 2 for first or second*/
extern void receive(int cycle, int operation, int cell_number)
   /* operation = 1 or 2 for svalue or state */
extern void compute_svalue(int cell_number)
extern void compute_state(int cell_number)
extern void graph(void)
extern void set_fault_register(int cell_number)

 int main()
{ /*   begin main */
/*   local variable definitions    */
int master_clock_cycles = 0   /* indicates the clock cycle
         currently  being simulated   */
int i, j, k, number_of_faults
char display = '0'  /* local variables */

output_file = fopen("simulate.out","w") /* clears output file */

fclose(output_file)

output_file = fopen("simulate.out","a") /* opens file to write */

input_file = fopen("simulate.in","r")
fscanf(input_file,"%d",&number_of_faults)
      /* get number of faults first test*/

while(number_of_faults != 0)
{ /* begin while */

 for(cell_number = 0 cell_number < = 99; cell_number + + )
  initialize(cell_number)
  /* call initialization function to set up the
  initial communications paramaters     */

 for(j = 1 j < = number_of_faults; j + + )
  { fscanf(input_file,"%d",&cell_number)
  state[cell_number] = 0xf
  }  /* get list of faulty cells and set states */
while((master_clock_cycles != 25)  )
 { /* begin simulation cycle, runs for 25 major clock cycles */

 for(clock_cycle = 1 clock_cycle < = 11; + + clock_cycle)
  { /* begin clock pulse "for" loop    */
 /* minor clock cycles
/* set pulse to cycle from beginning */

   if(clock_cycle = = 1)
   {  /* begin if  */
   + + master_clock_cycles
   clock_cycle = 6

/*  bypass the s_value portion of the code
cells are not currently programed to
calculate the s_value for reasons of speed */

   } /* end if clock = = 1 */
```

```
 for(cell_number = 0 cell_number < =99; + +cell_number)
   run_cell(cell_number,clock_cycle)

  /* cycle through cell operations each clock cycle  */

  } /* end clock pulse "for" loop */
/* print fault register at the end of each cycle,
 comment out these lines if only final registers are needed */
  for(i=0 i < =99; i+ +)
 {
  if( !(i%10)) fprintf(output_file,"\n")
  if(state[i] != 0xf) fprintf(output_file,"%8lx "
                          ,fault_register_delayed[i])
   fprintf(output_file,"  ---  ")
  }
 fprintf(output_file,"\n")

} /* end while master_clock_cycles != 25  */

master_clock_cycles = 0 /* reset master clock for next run */

if(display = = '0') graph()
/* display array configuration on screen */
getch() /* halt at end of display */

fscanf(input_file,"%d",&number_of_faults)

 } /* end while  number of faults != 0   */

 return (0)

} /* end main */
```

A.2 Init.c

```
/*************************************************************
 Initialize.c   This file will initialize all parameters to
 the initial conditions,  communications will be
 north = 1, south = 3, east = 2 and west = 1.
 The code is expandable to a neighborhood of all cells
 with distance 3 or less.
 This is a generic file, not all variables are used in
 all algorithms.
 the scheme is        9
                    8   5  10
                  7  3   1   4  11
               6  2  0    X  23  21 17
                 12 19  22  20  16
                   13  18  15
                      14
The value of north_link, ect is set to the value of the connetion
shown in the above diamond whenever a connection is to be made. Both
cells must want to connect. If cell (x) has a connection "0" enabled,
then (x-1) must have "2", or some other cell have a connection set
which corresponds to cell =(x)'s connection 0. The relative position of
the numbers in the above diamond correspond to the physical position of
the cell to which that number is associated. Initial conditions are
for each cell to enable 0,1,23 and 22. This produces a 2-dimensional
lattice connection. These numbers were selected because adding
an enabled port within one cell to the associated port in an
adjoining cell produces the value 23 and makes coding easier. If a
cell outputs to its northern neighbor on port 3, it would expect to
receive data from that same cells port 20.

*************************************************************/
extern long input_from_north[100][2],input_from_south[100][2],
   input_from_east[100][2],input_from_west[100][2];
```

```
extern int north_link[100], south_link[100],
                    east_link[100], west_link[100];
extern long fault_register[100],fault_register_delayed[100];
extern int connection_valid[100][24],
   s_value[100],
   north_west_is_faulty[100],
   north_east_is_faulty[100],
   east_is_faulty[100],
   west_is_faulty[100],
   state[100],
   south_west_is_faulty[100],
   south_east_is_faulty[100],
   fareast_is_faulty[100],
   farwest_is_faulty[100],
   row_deformation_north_blocker[100],
   row_deformation_northeast_blocker[100],
   row_deformation_northwest_blocker[100],
   row_deformation_east[100][2],
   row_deformation_west[100][2];

void initialize(cell_number)
{ /* begin initialize  */

int i;  /* indexing variable */

north_link[cell_number] = 1;
south_link[cell_number] = 22;
east_link[cell_number] = 23;
west_link[cell_number] = 0;
row_deformation_north_blocker[cell_number] = 0;
row_deformation_northeast_blocker[cell_number] = 0;
row_deformation_northwest_blocker[cell_number] = 0;
row_deformation_northwest_blocker[cell_number] = 0;

row_deformation_east[cell_number][0] = 0;
row_deformation_west[cell_number][0] = 0;
row_deformation_east[cell_number][1] = 0;
row_deformation_west[cell_number][1] = 0;
s_value[cell_number] = 1;
state[cell_number] = 1;
north_west_is_faulty[cell_number] = 0;
north_east_is_faulty[cell_number] = 0;
south_west_is_faulty[cell_number] = 0;
south_east_is_faulty[cell_number] = 0;
east_is_faulty[cell_number] = 0;
west_is_faulty[cell_number] = 0;
farwest_is_faulty[cell_number] = 0;
fareast_is_faulty[cell_number] = 0;

input_from_north[cell_number][0] = 0;
input_from_south[cell_number][0] = 0;
input_from_west[cell_number][0] = 0;
input_from_east[cell_number][0] = 0;
input_from_north[cell_number][1] = 0;
input_from_south[cell_number][1] = 0;
input_from_west[cell_number][1] = 0;
input_from_east[cell_number][1] = 0;

fault_register[cell_number] = 0;
fault_register_delayed[cell_number] = 0;

for(i=0; i<24; i++)
   connection_valid[cell_number][i] = 1;  /* set all ports valid */
   /* next code disables ports to non-existent cells */
   if(cell_number < 10) {
  connection_valid[cell_number][1] = 0;
  connection_valid[cell_number][7] = 0;
  connection_valid[cell_number][3] = 0;
  connection_valid[cell_number][4] = 0;
  connection_valid[cell_number][11] = 0;
  } /* these cells have no northern neighbor */
   if(cell_number < 20) {
```

```
 connection_valid[cell_number][8] = 0;
 connection_valid[cell_number][5] = 0;
 connection_valid[cell_number][10] = 0;
    }
 if(cell_number < 30) connection_valid[cell_number][9] = 0;
 if(cell_number > 69) connection_valid[cell_number][14] = 0;
 if(cell_number > 79) {
connection_valid[cell_number][13] = 0;
connection_valid[cell_number][18] = 0;
connection_valid[cell_number][15] = 0;
    }
 if(cell_number > 89) {
connection_valid[cell_number][12] = 0;
connection_valid[cell_number][19] = 0;
connection_valid[cell_number][22] = 0;
connection_valid[cell_number][20] = 0;
connection_valid[cell_number][16] = 0;
   }
 if((cell_number %10)= =0) {
connection_valid[cell_number][13] = 0;
connection_valid[cell_number][19] = 0;
connection_valid[cell_number][0] = 0;
connection_valid[cell_number][3] = 0;
connection_valid[cell_number][8] = 0;
connection_valid[cell_number][12] = 0;
connection_valid[cell_number][7] = 0;
connection_valid[cell_number][2] = 0;
connection_valid[cell_number][6] = 0;
   }
 if(((cell_number -1) %10)= =0) {
connection_valid[cell_number][12] = 0;
connection_valid[cell_number][2] = 0;
connection_valid[cell_number][7] = 0;
connection_valid[cell_number][6] = 0;
      }
 if(((cell_number -2)% 10)= =0) connection_valid[cell_number][6] = 0;
 if(((cell_number +1) %10)= =0) {
connection_valid[cell_number][10] = 0;
connection_valid[cell_number][4] = 0;
connection_valid[cell_number][23] = 0;
connection_valid[cell_number][20] = 0;
connection_valid[cell_number][15] = 0;
connection_valid[cell_number][11] = 0;
connection_valid[cell_number][21] = 0;
connection_valid[cell_number][16] = 0;
connection_valid[cell_number][17] = 0;
   }
 if(((cell_number +2) %10)= = 0) {
connection_valid[cell_number][11] = 0;
connection_valid[cell_number][21] = 0;
connection_valid[cell_number][16] = 0;
connection_valid[cell_number][17] = 0;
      }
 if(((cell_number +3)%10)= = 0) connection_valid[cell_number][17] = 0;


} /* end initialize */
```

A.3 Cell.c

```
/*********************************************************************
```

FILE cell.c This file represents the normal control
functions of the control logic. When called, it will
call the proper operation function, depending upon the

```
clock cycle
*************************************************************************/
#include < stdio.h >

extern state[100];

void run_cell(cell_number, clock_cycle)

{ /* begin cell operations */

   switch ( clock_cycle)
   {
   case 1: transmit(1,1,cell_number); break;
   case 2: receive(1,1,cell_number); break;
   case 3: transmit(2,1,cell_number); break;
   case 4: receive(2,1,cell_number); break;
case 5: compute_svalue(cell_number); break;
case 6: transmit(1,2,cell_number); break;
case 7: receive(1,2,cell_number); break;
case 8: transmit(2,2,cell_number); break;
case 9: receive(2,2,cell_number); break;
case 10: {
  set_fault_register(cell_number);
/*compute_state(cell_number);
* reconfigure(cell_number);
  break;
  }
   } /* end switch */

} /* end cell operations */
```

A.4 Transmit.c

```
/*************************************************************

File transmit.c  This function transmits the proper output
from each cell.  It takes two input values, both integers.
if input 1 is 1, then the transmit is an s_value, it it is
2, then the transmit is a state value and reconfiguration
information

*************************************************************************/

#include < stdio.h >

extern int
s_value[100],
state[100],
multiple_faults[100],north_west_is_faulty[100],
south_west_is_faulty[100],north_east_is_faulty[100],
south_east_is_faulty[100],
fault_in_critical_region[100],
fault_in_critical_region_north[100],
fault_in_critical_region_south[100],
north_link[100],
east_link[100],
west_link[100],
south_link[100],
row_deformation_north_blocker[100],
row_deformation_northeast_blocker[100],
row_deformation_northwest_blocker[100],
row_deformation_east[100][2],
row_deformation_west[100][2];
extern long
fault_register[100],
cell_output[100][24][2];
```

```
void transmit(cycle,operation,cell_number)
{ /* begin transmit function */
long int output[2];   /* local temporary holding register */
int i,j;

if(state[cell_number] != 0xf)  {
 if(operation == 1 ) output[0] = s_value[cell_number];
 else   {

 output[0] = state[cell_number];
if(row_deformation_north_blocker[cell_number])
  output[0] = output[0] |     0x20000;
if(row_deformation_northwest_blocker[cell_number])
  output[0] = output[0] |     0x40000;
if(row_deformation_northeast_blocker[cell_number])
  output[0] = output[0] |     0x800000;
if(row_deformation_east[cell_number][0])
  output[0] = output[0] |   0x1000000;
if(row_deformation_west[cell_number][0])
  output[0] = output[0] |  0x2000000;
if(north_west_is_faulty[cell_number])
  output[0] = output[0] | 0x10000000;
if(south_west_is_faulty[cell_number])
  output[0] = output[0] | 0x20000000;
if(north_east_is_faulty[cell_number])
  output[0] = output[0] | 0x40000000;
if(south_east_is_faulty[cell_number])
  output[0] = output[0] | 0x80000000;
output[1] =
 fault_register[cell_number];
  }
  } else {  output[0] = 0xf; output[1] = 0; }
if(cycle == 1 && operation == 1)
{
   for(i = 0; i<24; i++)
 {
 cell_output[cell_number][i][0] = 0;
 cell_output[cell_number][i][1] = 0;

 } /* this code sets up the output connections- any cell attempting to
   read an input which is all zeroes will ignore the input
   since it is attempting to complete a connection while
   the cell on the other end is not  **********************/

}
if(cycle == 1 )
{
cell_output[cell_number][north_link[cell_number]][0] = output[0];
cell_output[cell_number][north_link[cell_number]][1] = output[1];
cell_output[cell_number][west_link[cell_number]][0] = output[0];
cell_output[cell_number][west_link[cell_number]][1] = output[1];
}
else if(cycle == 2)
{
cell_output[cell_number][east_link[cell_number]][0] = output[0];
cell_output[cell_number][east_link[cell_number]][1] = output[1];
cell_output[cell_number][south_link[cell_number]][0] = output[0];
cell_output[cell_number][south_link[cell_number]][1] = output[1];
}
/*************************************************************
special code for faulty cells , all outputs = 0xf
*************************************************************/
if(operation == 2 && state[cell_number] == 0xf)
 for(i=0; i<= 23; i++)
  cell_output[cell_number][i][0] = 0xf;

} /* end transmit function */
```

A.5 Receive.c

```
/*****************************************************************
File receive.c  This function receives data from other cells.
It requires two parameters, cycle and operation.  if cycle is a
1, the receive is from cells which are the logical south and
east neighbors.  If cycle is a 2, the reception is from north and
west.
If operation is a 1, the data is s_value, a 2 indicates state and
fault information

THIS VERSION OF RECEIVE IS CODED ONLY FOR THE 10 NEIGHBOR
VERSION OF THE CONNECTIONS, WITH SPARE COLUMNS ON EITHER END

*****************************************************************/

#include < stdio.h >


extern long
  input_from_north[100][2],
  input_from_south[100][2],
  cell_output[100][24][2],
  input_from_east[100][2],
  input_from_west[100][2];
extern connection_valid[100][24],
  s_value_register[100][4],
  north_link[100],
  south_link[100],
  east_link[100],
  west_link[100];



void receive(cycle, operation,cell_number)
{ /* begin reception function */
int i;



if(cycle = = 1 && operation = = 1)
  { /* read s-value from south and eastern neighbors */
  if(connection_valid[cell_number][south_link[cell_number]])
  {
switch(south_link[cell_number] )
{
case 22: s_value_register[cell_number][0] =
     cell_output[cell_number + 10][1][0]; break;
case 19: s_value_register[cell_number][0] =
     cell_output[cell_number + 9][4][0]; break;
case 20: s_value_register[cell_number][0] =
     cell_output[cell_number + 11][3][0]; break;
case 18: s_value_register[cell_number][0] =
     cell_output[cell_number + 20][5][0]; break;
} /* end south link switch */
  } /* end if connection south valid */
  else   s_value_register[cell_number][0] =  0;

  if(connection_valid[cell_number][east_link[cell_number]])
  { /* read east neighbor output */
switch(east_link[cell_number] )
{
case 23: s_value_register[cell_number][1] =
     cell_output[cell_number + 1][0][0]; break;
case 21: s_value_register[cell_number][1] =
     cell_output[cell_number + 2][2][0]; break;
} /* end east switch */
  } /* end east validation code */
```

```
  else    s_value_register[cell_number][1] =  0;
  } /* end if cycle and oper = 1 */
if(cycle = = 2 && operation = = 1)
  {
  if(connection_valid[cell_number][north_link[cell_number]])
  { /* read north neighbor output */
  switch(north_link[cell_number])
  {
  case 3: s_value_register[cell_number][2] =
    cell_output[cell_number - 11][20][0]; break;
  case 1: s_value_register[cell_number][2] =
    cell_output[cell_number - 10][22][0]; break;
  case 4: s_value_register[cell_number][2] =
    cell_output[cell_number - 9][19][0];  break;


  } /* end north switch */
  } /* end if valid north connection */
  else    s_value_register[cell_number][2] =  0;
if(connection_valid[cell_number][west_link[cell_number]])
   switch(west_link[cell_number])
   {
   case 0: s_value_register[cell_number][3] =
   cell_output[cell_number -1][23][0]; break;
   case 2: s_value_register[cell_number][3] =
   cell_output[cell_number - 2][21][0]; break;
   } /* end west input */
   else s_value_register[cell_number][3] = 0;

  } /* end cycle 2 operation 1 input */


  if(operation = = 2 && cycle = = 1)
  { /* read state and fault information from south and
  eastern neighbors */
  if(connection_valid[cell_number][south_link[cell_number]])
  {
switch(south_link[cell_number] )
{
case 22:{
  input_from_south[cell_number][0] =
      cell_output[cell_number + 10][1][0];
  input_from_south[cell_number][1] =
      cell_output[cell_number + 10][1][1]; break; }
case 19:{
 input_from_south[cell_number][0] =
      cell_output[cell_number + 9][4][0];
 input_from_south[cell_number][1] =
      cell_output[cell_number + 9][4][1]; break; }
case 20:{
 input_from_south[cell_number][0] =
      cell_output[cell_number + 11][3][0];
 input_from_south[cell_number][1] =
      cell_output[cell_number + 11][3][1]; break;}
case 18:{
 input_from_south[cell_number][0] =
      cell_output[cell_number + 20][5][0];
 input_from_south[cell_number][1] =
      cell_output[cell_number + 20][5][1]; break;}
case 14:{
 input_from_south[cell_number][0] =
      cell_output[cell_number + 30][9][0];
 input_from_south[cell_number][1] =
      cell_output[cell_number + 30][9][1]; break;}
} /* end south link switch */
  } /* end if connection south valid */
  else {   input_from_south[cell_number][0] =  0;
    input_from_south[cell_number][1] = 0;   }

  if(connection_valid[cell_number][east_link[cell_number]])
  { /* read east neighbor output */
```

```
switch(east_link[cell_number] )
{
case 23:{
 input_from_east[cell_number][0] =
      cell_output[cell_number + 1][0][0];
 input_from_east[cell_number][1] =
      cell_output[cell_number + 1][0][1];
       break; }
case 21:{
  input_from_east[cell_number][0] =
      cell_output[cell_number + 2][2][0];
 input_from_east[cell_number][1] =
      cell_output[cell_number + 2][2][1]; break;}
case 20:{
 input_from_east[cell_number][0] =
      cell_output[cell_number + 11][3][0];
 input_from_east[cell_number][1] =
      cell_output[cell_number + 11][3][1]; break;}
  case 4:{
 input_from_east[cell_number][0] =
     cell_output[cell_number - 9][19][0];
 input_from_east[cell_number][1] =
     cell_output[cell_number - 9][19][1];
  break; }
} /* end east switch */
  } /* end east validation code */
  else  {
input_from_east[cell_number][0] =  0;
input_from_east[cell_number][1] =  0;
} /* end else */
  } /* end if cycle and oper = 1 */
if(cycle = = 2 && operation = = 2)
  {
  if(connection_valid[cell_number][north_link[cell_number]])
  { /* read north neighbor output */
  switch(north_link[cell_number])
  {
  case 3:{
   input_from_north[cell_number][0] =
     cell_output[cell_number - 11][20][0];
   input_from_north[cell_number][1] =
     cell_output[cell_number - 11][20][1];
      break; }
  case 1:{
 input_from_north[cell_number][0] =
     cell_output[cell_number - 10][22][0];
   input_from_north[cell_number][1] =
     cell_output[cell_number - 10][22][1];
     break;}
  case 4:{
 input_from_north[cell_number][0] =
     cell_output[cell_number - 9][19][0];
 input_from_north[cell_number][1] =
     cell_output[cell_number - 9][19][1];
  break; }
  case 5:{
 input_from_north[cell_number][0] =
     cell_output[cell_number - 20][18][0];
 input_from_north[cell_number][1] =
     cell_output[cell_number - 20][18][1];
  break; }
  case 9:{
 input_from_north[cell_number][0] =
     cell_output[cell_number - 30][14][0];
 input_from_north[cell_number][1] =
     cell_output[cell_number - 30][14][1];
  break; }

  } /* end north switch */
  } /* end if valid north connection */
  else {
 input_from_north[cell_number][0] =  0;
```

```
 input_from_north[cell_number][1] =  0;
 }
if(connection_valid[cell_number][west_link[cell_number]])
   switch(west_link[cell_number])
   {
   case 0:{
input_from_west[cell_number][0] =
   cell_output[cell_number -1][23][0];
input_from_west[cell_number][1] =
   cell_output[cell_number -1][23][1]; break;}
   case 2:{
input_from_west[cell_number][0] =
   cell_output[cell_number - 2][21][0];
input_from_west[cell_number][1] =
   cell_output[cell_number - 2][21][1]; break; }
case 3:{
input_from_west[cell_number][0] =
   cell_output[cell_number - 11][20][0];
input_from_west[cell_number][1] =
   cell_output[cell_number - 11][20][1]; break; }
case 19:{
 input_from_west[cell_number][0] =
      cell_output[cell_number +9][4][0];
 input_from_west[cell_number][1] =
      cell_output[cell_number +9][4][1]; break; }
} /* end west input */
else{
input_from_west[cell_number][0] = 0;
input_from_west[cell_number][1] = 0;
   }

  } /* end cycle 2 operation 2 input */


} /* end reception function */
```

A.6 Newsval.c

```
/*********************************************************************
File newsval.c  This function computes the new s_value
of a cell.  Input parameter is the cell_number

*********************************************************************/
#include <stdio.h>
extern s_value[100], s_value_register[100][4];
extern state[100];
void compute_svalue(cell_number)
{ /* begin new s value computation */
int j,i; /* indexing variable */

s_value[cell_number] = s_value_register[cell_number][0];

for(i = 1; i < 4; i++)  /* find minimum */
  if(s_value[cell_number] > s_value_register[cell_number][i])
 s_value[cell_number] = s_value_register[cell_number][i];

  ++ s_value[cell_number];

} /* end new s function */
```

A.7 Graphics.c

```
#include <graphics.h>
#include <stdlib.h>
```

```
#include < conio.h >
#include < stdio.h >

extern int north_link[100],south_link[100],
east_link[100],west_link[100];
extern int connection_valid[100][24],state[100];

void graph()
{ /* begin graphics routine */
   /* request auto detection */
   int g_driver = DETECT, g_mode, errorcode;
   int x_cord, y_cord, temp;
   int radius = 12;
   int i,j,cell;

   int south[100][2] = {20,32 ,80,32, 140,32, 200,32,
   260,32, 320,32, 380,32, 440,32, 500,32, 560,32,
     20,80, 80,80, 140,80, 200,80, 260,80, 320,80,
380,80, 440,80, 500,80, 560,80,
     20,128,80,128,140,128,200,128,260,128,320,128,380,128,
440,128,500,128,560,128,
     20,176,80,176,140,176,200,176,260,176,320,176,380,176,
440,176,500,176,560,176,
     20,224,80,224,140,224,200,224,260,224,320,224,380,224,
440,224,500,224,560,224,
     20,272,80,272,140,272,200,272,260,272,320,272,380,272,
440,272,500,272,560,272,
     20,320,80,320,140,320,200,320,260,320,320,320,380,320,
440,320,500,320,560,320,
     20,368,80,368,140,368,200,368,260,368,320,368,380,368,
440,368,500,368,560,368,
     20,416,80,416,140,416,200,416,260,416,320,416,380,416,
440,416,500,416,560,416,
     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
   int north[100][2] = {
     0,0 ,0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0,
     20,56, 80,56, 140,56, 200,56, 260,56, 320,56, 380,56,
440,56, 500,56, 560,56,
     20,104,80,104,140,104,200,104,260,104,320,104,380,104,440,104,
500,104,560,104,
     20,152,80,152,140,152,200,152,260,152,320,152,380,152,440,152,
500,152,560,152,
     20,200,80,200,140,200,200,200,260,200,320,200,380,200,440,200,
500,200,560,200,
     20,248,80,248,140,248,200,248,260,248,320,248,380,248,440,248,
500,248,560,248,
     20,296,80,296,140,296,200,296,260,296,320,296,380,296,440,296,
500,296,560,296,
     20,344,80,344,140,344,200,344,260,344,320,344,380,344,440,344,
500,344,560,344,
     20,392,80,392,140,392,200,392,260,392,320,392,380,392,440,392,
500,392,560,392,
     20,440,80,440,140,440,200,440,260,440,320,440,380,440,440,440,
500,440,560,440
     };
   int east[100][2] = {32,20 ,92,20, 152,20, 212,20, 272,20, 332,
   20, 392,20, 452,20, 512,20, 0,0,
     32,68 ,92,68, 152,68, 212,68, 272,68, 332,68, 392,68, 452,68,
512,68, 0,0,
     32,116 ,92,116, 152,116, 212,116, 272,116, 332,116, 392,116,
452,116, 512,116, 0,0,
     32,164 ,92,164, 152,164, 212,164, 272,164, 332,164, 392,164,
452,164, 512,164, 0,0,
     32,212 ,92,212, 152,212, 212,212, 272,212, 332,212, 392,212,
452,212, 512,212, 0,0,
     32,260 ,92,260, 152,260, 212,260, 272,260, 332,260, 392,260,
452,260, 512,260, 0,0,
     32,308 ,92,308, 152,308, 212,308, 272,308, 332,308, 392,308,
452,308, 512,308, 0,0,
     32,356 ,92,356, 152,356, 212,356, 272,356, 332,356, 392,356,
452,356, 512,356, 0,0,
```

```
    32,404 ,92,404, 152,404, 212,404, 272,404, 332,404, 392,404,
452,404, 512,404, 0,0,
    32,452 ,92,452, 152,452, 212,452, 272,452, 332,452, 392,452,
452,452, 512,452, 0,0,
        };

  int west[100][2] = {0,0 ,68,20, 128,20,188,20,248,20,308,20,368,
  20,428,20,488,20,548,20,
    0,0 ,68,68, 128,68,188,68,248,68,308,68,368,68,428,68,488,68,
548,68,
    0,0 ,68,116, 128,116,188,116,248,116,308,116,368,116,428,116,
488,116,548,116,
    0,0 ,68,164, 128,164,188,164,248,164,308,164,368,164,428,164,
488,164,548,164,
    0,0 ,68,212, 128,212,188,212,248,212,308,212,368,212,428,212,
488,212,548,212,
    0,0 ,68,260, 128,260,188,260,248,260,308,260,368,260,428,260,
488,260,548,260,
    0,0 ,68,308, 128,308,188,308,248,308,308,308,368,308,428,308,
488,308,548,308,
    0,0 ,68,356, 128,356,188,356,248,356,308,356,368,356,428,356,
488,356,548,356,
    0,0 ,68,404, 128,404,188,404,248,404,308,404,368,404,428,404,
488,404,548,404,
    0,0 ,68,452, 128,452,188,452,248,452,308,452,368,452,428,452,
488,452,548,452,
     };

  initgraph(&g_driver, &g_mode, "bgi");
  err
  orcode = graphresult();
          setcolor(4);
for(y_cord = 20; y_cord < =480; y_cord = y_cord + 48)
 for(x_cord =20; x_cord < 600; x_cord = x_cord + 60)

{
if(x_cord = = 20 && y_cord = = 20) cell = 0;
else + + cell;
if(state[cell] = = 0xf) {
setcolor(9);
fillellipse(x_cord,y_cord,12,12);
}
else {
setcolor(4);
circle(x_cord,y_cord,radius);
}
 }


for(i = 10;i < =99; i+ + )  /* draw north_south links */
if(state[i] != 0xf)
  {
  switch(north_link[i] )
  {
  case 0: {
  if(connection_valid[i][0])
  {
  if((south_link[i-1] != 23) || (state[i-1] = = 0xf))  setcolor(9);
  line(north[i][0],north[i][1],north[i][0],north[i][1]-10);
  line(north[i][0],north[i][1]-10,north[i][0]-10,north[i][1]-10);
  line(north[i][0]-10,north[i][1]-10,south[i-1][0]+ 10,south[i-1][1]+ 10);
  line(south[i-1][0]+ 10,south[i-1][1]+ 10,south[i-1][0],
south[i-1][1]+ 10);
  line(south[i-1][0],south[i-1][1]+ 10,south[i-1][0],south[i-1][1]);
  }  break;
    }

  case 3: {
  if(connection_valid[i][3])
  {
  if(south_link[i-11] != 20) setcolor(9);
  line(north[i][0],north[i][1],south[i-11][0],south[i-11][1]);
```

```
  } break;
    }
case 1: {
  if(connection_valid[i][1])
  {
   if(south_link[i-10] != 22) setcolor(9);
   line(north[i][0],north[i][1],south[i-10][0],south[i-10][1]);
  }
    break;
    }
case 4: {
  if(connection_valid[i][4])
    {
   if(south_link[i-9] != 19) setcolor(9);
   line(north[i][0],north[i][1],south[i-9][0],south[i-9][1]);
   } break;
    }
  case 5: {
  if(connection_valid[i][5])
  {
  if((south_link[i-20] != 18) || (state[i-20] == 0xf))setcolor(9);
  line(north[i][0],north[i][1],north[i][0] + 20,north[i][1]-20);
  line(north[i][0] + 20,north[i][1]-20,north[i][0] + 20,south[i-20][1] + 20);
  line(north[i][0] + 20,south[i-20][1] + 20,south[i-20][0],south[i-20][1]);

  } break;
    }
  case 9: {
  if(connection_valid[i][9])
  {
  if((south_link[i-30] != 14) || (state[i-30] == 0xf))setcolor(9);
  line(north[i][0],north[i][1],north[i][0] + 20,north[i][1]-20);
  line(north[i][0] + 20,north[i][1]-20,north[i][0] + 20,south[i-30][1] + 20);
  line(north[i][0] + 20,south[i-30][1] + 20,south[i-30][0],south[i-30][1]);

  } break;
    }

  }/* end switch */
  setcolor(4);
} /* end north link drawing */

for(i = 0;i < = 99;i + + )  /* draw east -west links */
if(state[i] != 0xf)
   {
 setcolor(3);

switch(west_link[i])
{
case 0:
   {
   if(connection_valid[i][0])
  {
  if(east_link[i-1] != 23) setcolor(9);
  line(west[i][0],west[i][1],east[i-1][0],east[i-1][1]);
  } break;
 }
case 2:
   {
   if(connection_valid[i][2])
 {
  if(east_link[i-2] != 21) setcolor(9);
 line(west[i][0],west[i][1],west[i][0]-30,west[i][1] + 20);
 line(west[i][0]-30,west[i][1] + 20,east[i-2][0] + 30,east[i-2][1] + 20);
 line(east[i-2][0] + 30,east[i-2][1] + 20,east[i-2][0],east[i-2][1]);
 } break;
 }
case 3:
 {
 if(connection_valid[i][3])
  {
  if((east_link[i-11] != 20) || (state[i-11] == 0xf))
```

```
    setcolor(9);
   line(west[i][0],west[i][1],east[i-11][0],east[i-11][1]);
   } break;
  }
 case 8:
  {
  if(connection_valid[i][8])
   {
   if((east_link[i-21] != 15) || (state[i-21] == 0xf))
   setcolor(9);
/*line(west[i][0],west[i][1],west[i][0]-20,west[i][1]-20);
   line(west[i][0]-20,west[i][1]-20,east[i-21][0]+20,east[i-21][1]+20);
   line(east[i-21][0]+20,east[i-21][1]+20,east[i-21][0],east[i-21][1]);
*/line(west[i][0],west[i][1],east[i-21][0],east[i-21][1]);
   } break;
  }
 case 13:
  {
  if(connection_valid[i][13])
   {
   if((east_link[i+19] != 10) || (state[i+19] == 0xf))
   setcolor(9);
/*line(west[i][0],west[i][1],west[i][0]-20,west[i][1]-20);
   line(west[i][0]-20,west[i][1]-20,east[i-21][0]+20,east[i-21][1]+20);
   line(east[i-21][0]+20,east[i-21][1]+20,east[i-21][0],east[i-21][1]);
*/line(west[i][0],west[i][1],east[i+19][0],east[i+19][1]);
   } break;
  }
 case 19:
  {
  if(connection_valid[i][19])
   {
   if((east_link[i+9] != 4)||(state[i+9] == 0xf))
     setcolor(9);
   line(west[i][0],west[i][1],east[i+9][0],east[i+9][1]);
   } break;
  }

 }/* end switch */


    } /* end west draw */

 getch();
  closegraph();



 } /* end graphics routine */
```

A.8 Algor1.c

```
/*********************************************************

File: algor1b.c  This file contains code to
set the inter-cell communications links

*********************************************************/
#include < stdio.h >
/******************* GLOBAL VARIABLES  ************************/
extern long fault_register_delayed[100];

extern int north_link[100], south_link[100], east_link[100],
 west_link[100],state[100];
void reconfigure(cell_number)
```

```
{ /* begin reconfigure */

if(fault_register_delayed[cell_number] & 0x100)
  west_link[cell_number] = 2;
else west_link[cell_number] = 0;

if(fault_register_delayed[cell_number] & 0x40)
  east_link[cell_number] = 21;
else east_link[cell_number] = 23;

/* begin algorithm 2-10 code  */
/* begin north link code */

switch(fault_register_delayed[cell_number] & 0x30c0000)
 {
case 0x0:{
  if(
   (!(fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x3))
   ||((fault_register_delayed[cell_number] & 0x300)
     && fault_register_delayed[cell_number] & 0x2)
    )
  north_link[cell_number] = 1;
else    if(
    ((fault_register_delayed[cell_number] & 0x3)
     && !(fault_register_delayed[cell_number] & 0x300))
    )
  north_link[cell_number] = 4;
else    if(
    ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x2))
    )
  north_link[cell_number] = 3;
  break; } /* end case  */
case 0x1000000:{                    /* fcr */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x3))
    )
  north_link[cell_number] = 1;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
    && (fault_register_delayed[cell_number] & 0x3))
  ||(!(fault_register_delayed[cell_number] & 0x300))
          )
        north_link[cell_number] = 4;
else  /* not possible in this configuration */
        north_link[cell_number] = 3;

  break; } /* end case 1000000  fcr   */
case 0x2000000:{
        if(
  ((fault_register_delayed[cell_number] & 0x2) && !
(fault_register_delayed[cell_number] & 0x300))
          )
        north_link[cell_number] = 1;

else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x2) &&
(fault_register_delayed[cell_number] & 0x300))
          )
        north_link[cell_number] = 3;
else    north_link[cell_number] = 4;
    break; } /* end case 2000000 fcrn   */
case 0x3000000:{
  if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
    !(fault_register_delayed[cell_number] & 0x3))
  ||((fault_register_delayed[cell_number] & 0x300)
      && (fault_register_delayed[cell_number] & 0x2))
        )
```

```
        north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
         )
        north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x2))
         )
        north_link[cell_number] = 3;
  break; } /* end case 3000000 fcr and fcrn */
case 0x40000:
case 0x1040000:{   /* mf or mf and fcr */
        if(
  ((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x3))
  ||(fault_register_delayed[cell_number] & 0x2 &&
!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
(fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0) &&
!(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;



  break; } /* end case 40000 multiple fault mf */
case 0x2040000:
case 0x3040000:{
  if(
  (fault_register_delayed[cell_number] & 0x4)
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 3;
  break; } /* end case 2040000 mf and fcrn */
case 0x80000:
case 0x2080000:{ /* mfn or mfn and fcrn */
  if((fault_register_delayed[cell_number] & 0x7)  = = 0x7)
  north_link[cell_number] = 5;
else if(
  (!(fault_register_delayed[cell_number] & 0x300)
     && ((fault_register_delayed[cell_number] & 0x6)= = 0x6))
   ||((fault_register_delayed[cell_number] & 0x300)
     && (!(fault_register_delayed[cell_number] & 0x5)))

        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x304))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x5) &&
(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
```

```
  break; } /* end case 80000  mfn  */
 case 0x1080000:
 case 0x3080000:{ /* mfn and fcr */
        if(fault_register_delayed[cell_number] = = 0x80007)
     north_link[cell_number] = 5;

 else     if(
   (!(fault_register_delayed[cell_number] & 0x300))
   ||((fault_register_delayed[cell_number] & 0x6) = = 0x6)
   )north_link[cell_number] = 1;
 else     if(
   (!(fault_register_delayed[cell_number] & 0x4))
   )north_link[cell_number] = 4;
 else     if(
   ((fault_register_delayed[cell_number] & 0x300)
      && !(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
   break; } /* end case 1080000 mfn and fcr */
 case 0xc0000:
 case 0x10c0000:
 case 0x20c0000:
 case 0x30c0000:{/* mf and mfn, fcr and fcrn not critical */
        if(
   (!(fault_register_delayed[cell_number] & 0x300)
   && !(fault_register_delayed[cell_number] & 0x3))
   ||((fault_register_delayed[cell_number] & 0x300)
   && (fault_register_delayed[cell_number] & 0xc0) &&
      ((fault_register_delayed[cell_number] & 0x6) = = 0x6))
   ||(!(fault_register_delayed[cell_number] & 0xc0)
   && !(fault_register_delayed[cell_number] & 0x5))
        )north_link[cell_number] = 1;
 else     if(
   (!(fault_register_delayed[cell_number] & 0x300)
   && (fault_register_delayed[cell_number] & 0x3))
 /*||(!(fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x3))   */
   ||((fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x300) &&
      !(fault_register_delayed[cell_number] & 0x4))
        )north_link[cell_number] = 4;
 else     if(
   ((fault_register_delayed[cell_number] & 0x300)
   && !(fault_register_delayed[cell_number] & 0x2))
   ||(!(fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x5))
        )north_link[cell_number] = 3;
   break; } /* end case c0000 mf and mfn  */
  } /* end switch north link code */

 /*  begin south link code  */
 switch(fault_register_delayed[cell_number] & 0x5140000)
 { /* south link switch code */
 case 0:
   { /* begin case 0 */
    if(
   (!(fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x18))
   ||((fault_register_delayed[cell_number] & 0x300)
     && (fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 22;
 else     if(
   ((fault_register_delayed[cell_number] & 0x300) &&
 !(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
 else     if(
   (!(fault_register_delayed[cell_number] & 0x300) &&
 (fault_register_delayed[cell_number] & 0x18))
        )south_link[cell_number] = 20;
   break; } /* end case 0, single faults, no critical area faults */

 case 0x1000000:
   { /* begin case  */
```

```
  if(
  ((fault_register_delayed[cell_number] & 0x300)
     && (!(fault_register_delayed[cell_number] & 0x18)))
  )south_link[cell_number] = 22;

else    if(
  ((fault_register_delayed[cell_number] & 0x300)
    && ((fault_register_delayed[cell_number] & 0x18)))
  ||(!(fault_register_delayed[cell_number] & 0x300))
  )south_link[cell_number] = 20;
else    south_link[cell_number] = 19;
  break; } /* end case 0x1000000, fcr */

case 0x4000000:
  { /* begin case  */

  if(
  (!(fault_register_delayed[cell_number] & 0x300)
      && fault_register_delayed[cell_number] & 0x10)
  )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x10)
    && (fault_register_delayed[cell_number] & 0x300))
       )south_link[cell_number] = 19;
else    south_link[cell_number] = 20;
/* not possible this configuration */
  break; } /* end case fcrs */

case 0x5000000:
  { /* begin case  */
       if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300)
&& fault_register_delayed[cell_number] & 0x10)
       )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
 !(fault_register_delayed[cell_number] & 0x10))
       )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x18))
       )south_link[cell_number] = 20;
  break; } /* end case fcr and fcrs      */

case 0x40000:
case 0x1040000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x18))
         )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xd0))
       )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     (fault_register_delayed[cell_number] & 0x18))
         )south_link[cell_number] = 20;
  break; } /* end case mf */

case 0x4040000:
    { /* begin case  */
       if(
  (fault_register_delayed[cell_number] & 0x20)
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0))
```

```
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )south_link[cell_number] = 19;
else    if(
  ((fault_register_delayed[cell_number] & 0x18) &&
!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 20;
  break; } /* end case mf and fcrs                          */

case 0x100000:
case 0x4100000:
  { /* begin case  */
  if(
   ((fault_register_delayed[cell_number] & 0x38) = = 0x38)
    ||((fault_register_delayed[cell_number] & 0x8000008) = = 0x8000008)
    )
    south_link[cell_number] = 18;
else
  (((fault_register_delayed[cell_number] & 0x30)= = 0x30) &&
      !(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x28) &&
(fault_register_delayed[cell_number] & 0x10) &&
     (fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x320))
        )south_link[cell_number] = 20;
  break; } /* end case mfs */


case 0x1100000:
case 0x5100000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x30) = = 0x30)
  ||(!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x20))
        )south_link[cell_number] = 20;
  break; } /* end case mfs and fcr */

case 0x140000:
case 0x1140000:
case 0x4140000:
case 0x5140000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0)
  && ((fault_register_delayed[cell_number] & 0x30) = = 0x30))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && !(fault_register_delayed[cell_number] & 0x28))
  ||(!(fault_register_delayed[cell_number] & 0x300)
   && !(fault_register_delayed[cell_number] & 0x18))
  )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x10))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x28))
  )south_link[cell_number] = 19;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
```

```
  && (fault_register_delayed[cell_number] & 0xc0)
   && !(fault_register_delayed[cell_number] & 0x20))
   ||(!(fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x28))
   ||(!(fault_register_delayed[cell_number] & 0x300)
   &&(fault_register_delayed[cell_number] & 0x18))
   )south_link[cell_number] = 20;
  break; } /* end case mf and mfs */


} /* end south link switch code */

} /* end reconfigure */
```

A.9 Faultrg1.c

```
/*************************************************************

file: set_f_rg.c set fault registers, this routine takes the input
from the receive routine and sets the appropriate fault register
bits

*********************************************************************/
#include < stdio.h >
extern long
input_from_north[100][2],
input_from_south[100][2],
input_from_east[100][2],
input_from_west[100][2],
fault_register[100],
fault_register_delay[100];
extern
north_link[100],
south_link[100],
east_link[100],
west_link[100],
state[100],
north_west_is_faulty[100],
north_east_is_faulty[100],
south_west_is_faulty[100],
south_east_is_faulty[100],
fareast_is_faulty[100],
farwest_is_faulty[100];

void set_fault_register(cell_number)

{
int i;
long bit[34];

 fault_register_delay[cell_number] = fault_register[cell_number];

for(i=0; i< = 33; i++)
bit[i] = 0;

/*********************************************************************
SET AUXILLARY BITS, NE IS FAULTY, SE IS FAULTY, NW IS FAULTY ANS SW IS
FAULTY THESE BITS ARE TRANSMITTED, BUT ARE NOT USED IN DECODING
*********************************************************************/
if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf)
   && (north_link[cell_number] = = 4))
  ||((input_from_east[cell_number][1] & 0x1) &&
  east_link[cell_number] = = 23)
  )
  north_east_is_faulty[cell_number] = 1;

if(
```

```
(((input_from_north[cell_number][0] & 0xf) = = 0xf)
  && (north_link[cell_number] = = 3))
  ||((input_from_west[cell_number][1] & 0x1) &&
 west_link[cell_number] = = 0)
  )
  north_west_is_faulty[cell_number] = 1;
if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf)
  && (south_link[cell_number] = = 20))
  ||((input_from_east[cell_number][1] & 0x8) &&
 east_link[cell_number] = = 23)
  )
  south_east_is_faulty[cell_number] = 1;
if(
(((input_from_south[cell_number][0] & 0xf) = = 0xf)
 && (south_link[cell_number] = = 19))
  ||((input_from_west[cell_number][1] & 0x8)&&
 west_link[cell_number] = = 0)
  )
south_west_is_faulty[cell_number] = 1;
if(
(((input_from_east[cell_number][0] & 0xf) = = 0xf)
  &&(east_link[cell_number] = = 21))
  ||((input_from_south[cell_number][0] & 0x40000000) &&
 (south_link[cell_number]= = 20))
  ||((input_from_east[cell_number][0] & 0x80000000)
 && east_link[cell_number] = = 4)
  )
  fareast_is_faulty[cell_number] = 1;

if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf)
  &&(west_link[cell_number] = = 2))
  )
  farwest_is_faulty[cell_number] = 1;



/*****************************************************************
RECONFIGURATION IS DONE BASED ON THE CONTENTS OF THE delayED
FAULT REGISTER, BUT INPUT IS USED TO SET THE CONTENTS OF THE
FAULT REGISTER.  THIS ALLOWS A CELL TO COMMUNICATE TO ITS
CURRENT NEIGHBORS ALL THE INPUT IT RECEIVED FROM ALL ITS
NEIGHBORS BEFORE DISCONNECTING FROM A CELL
*****************************************************************/

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 1))
  || ((input_from_west[cell_number][0] & 0x40000000)  &&
  (west_link[cell_number] = = 0) )
|| ( (input_from_east[cell_number][0] & 0x10000000)  &&
  (east_link[cell_number] = = 23) )
  )
  {
  fault_register[cell_number] = fault_register[cell_number] | 0x1;
  } /* immediate cell to the north is faulty */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = =  3))
  || ((input_from_north[cell_number][1] & 0x300) &&
  (( north_link[cell_number] = = 1) || (north_link[cell_number] = = 3)))
  || (( input_from_north[cell_number][1] & 0x200) &&
 north_link[cell_number] = = 4)
  ||((input_from_west[cell_number][1] & 0x3)&&
((input_from_west[cell_number][1] & 0xe00000) != 0x800000)&&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][0] & 0x40000000) &&
      (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
```

```
  west_link[cell_number] = = 3)
  || north_west_is_faulty[cell_number]
  )
  bit[1] = 1;
  /* fault north and west of this cell */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x5)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ((input_from_north[cell_number][1] & 0xc0) &&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4))
||((input_from_east[cell_number][1] & 0xc0) &&
   east_link[cell_number] = = 4)

)
  bit[2] = 1;
/* fault north and east of this cell */

if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 22))
 || ((input_from_west[cell_number][0] & 0x80000000) &&
   (west_link[cell_number] = = 0))
 || ((input_from_east[cell_number][0] & 0x20000000) &&
   (east_link[cell_number] = = 23))
  )
  fault_register[cell_number] = fault_register[cell_number] | 0x8;
  /* immediate southern neighbor is faulty */

if(

  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 19))
||((input_from_west[cell_number][1] & 0x18) &&
!((input_from_west[cell_number][1] & 0xe00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
||((input_from_west[cell_number][0] & 0x80000000) &&
  west_link[cell_number] = = 2)
||((input_from_south[cell_number][1] & 0x300) &&
  south_link[cell_number] = = 22 )
||((input_from_west[cell_number][1] & 0x300) &&
 west_link[cell_number] = = 19)
|| ((input_from_west[cell_number][1] & 0x8) &&
   west_link[cell_number] = = 0)
||((input_from_south[cell_number][1] & 0x2) &&
 south_link[cell_number] = = 18)
 )
bit[4] = 1;
  /* fault to the southwest of this cell */

/*if(cell_number = = 55 && (fault_register_delay[55] & 0x300038)){
  printf("55 has %lx input %d\n",input_from_south[55][1],south_link[55]); getch(); }
*/
if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 20))
||((input_from_east[cell_number][1] & 0x28)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
  (east_link[cell_number] = = 21 || east_link[cell_number] = = 23))
|| ((input_from_south[cell_number][1] & 0xc0)  &&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20)  )
||((input_from_east[cell_number][1] & 0xc0) &&
 east_link[cell_number] = = 20)
||((input_from_south[cell_number][1] & 0x4) &&
 south_link[cell_number] = = 18)

)

bit[5] = 1;
```

```
 /* fault to the southeast of this cell */


if(
  (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (east_link[cell_number] = = 23))
  )

  fault_register[cell_number] = fault_register[cell_number] | 0x40;
  /* immediate eastern neighbor is faulty */


if(
 (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (east_link[cell_number] = = 21))
  || ((input_from_east[cell_number][1] & 0xc0)&&
  !((input_from_east[cell_number][1] & 0x600000) = = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
  || ((input_from_east[cell_number][1] & 0x20) &&
 east_link[cell_number] = = 4)
  || ((input_from_east[cell_number][1] & 0x4) &&
  east_link[cell_number] = = 20)
  ||fareast_is_faulty[cell_number]
)
  bit[7] = 1;
  /* faulty cell to  the far east in this row */


if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 0))
  )
 {
  fault_register[cell_number] = fault_register[cell_number] | 0x100;
  /* immediate western neighbor is faulty */
}
if(
  ((((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
  !((input_from_west[cell_number][1] & 0xe00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x10) &&
  west_link[cell_number] = = 3)
  || ((input_from_west[cell_number][1] & 0x2) &&
  west_link[cell_number] = = 19))
/*  &&!(input_from_west[cell_number][1] & 0x20000)*/
  ||farwest_is_faulty[cell_number]
  )
bit[9] = 1;
 /* fault to the far west of this cell */


 if(
((input_from_north[cell_number][1] & 0x1) &&
  (north_link[cell_number] = = 1) )
|((input_from_north[cell_number][0] & 0x10000000) &&
  north_link[cell_number] = = 4)
||((input_from_north[cell_number][0] & 0x40000000) &&
  north_link[cell_number] = = 3)
 )
  bit[10] = 1;
/* nn-  fault 2 rows to the north and north of this cell*/


if(
((input_from_north[cell_number][1] & 0x2) &&
(north_link[cell_number] = = 1))
||((input_from_north[cell_number][1] & 0x3) &&
 (north_link[cell_number] = = 0x3))
|| ((input_from_west[cell_number][1] & 0xc00)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
```

```
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
)
bit[11] = 1;
/* nnw- fault 2 rows north and west of this cell*/


if(
(input_from_north[cell_number][1] & 0x4)
&& (north_link[cell_number] = = 1)
||((input_from_north[cell_number][1] & 0x5) &&
 (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x1400)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
  bit[12] = 1;
/* nne- fault 2 rows north and east of this cell*/


if(
(
((input_from_north[cell_number][1] & 0x80000) &&
 ((north_link[cell_number] = = 1) || (north_link[cell_number] = = 4) ) )
|| ((input_from_east[cell_number][1] & 0x2000)&&
  !((input_from_west[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||((input_from_west[cell_number][1] & 0x2000)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  )
 && fault_register[cell_number] & 0x1c00
)
bit[13] = 1;
 /*mfnne- multiple faults 2 rows north and east of this cell*/


if(
(input_from_south[cell_number][1] & 0x8)  &&
  (south_link[cell_number] = = 22)
|| ((input_from_south[cell_number][0] & 0x20000000) &&
  (south_link[cell_number] = = 20))
|| ((input_from_south[cell_number][0] & 0x80000000) &&
 (south_link[cell_number] = = 19))
)
bit[14] = 1;
/* ss- fault 2 rows to the south and in the same column */


if(
 ((input_from_south[cell_number][1] & 0x10) &&
 (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x18) &&
 (south_link[cell_number] = = 19))
||((input_from_west[cell_number][1] &  0xc000) &&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
){
bit[15] = 1;
 }
/* ssw- single fault 2 rows south and west of this cell*/


if(
((input_from_south[cell_number][1] & 0x20)  &&
  (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x28) &&
  (south_link[cell_number] = = 20))
||(((input_from_east[cell_number][1] & 0x14000))&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
bit[16] = 1;
```

```
/* sse- single fault 2 rows south and east of this cell*/
else bit[16] = 0;

 if(
 (
 ((input_from_south[cell_number][1] & 0x100000)&&
  ((south_link[cell_number] = = 22) || (south_link[cell_number] = = 20) ))
 ||((input_from_east[cell_number][1] & 0x20000)&&
  !(fault_register_delay[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
 ||(((input_from_west[cell_number][1] & 0x20000))&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
   (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  )&& fault_register[cell_number] & 0x1c000
  )
bit[17] = 1;
 /* mfss- multiple faults 2 rows to the south of cell*/


if(
(
((fault_register[cell_number] & 0xc0) = = 0xc0)
||((fault_register[cell_number] & 0x300) = = 0x300)
||( ((input_from_east[cell_number][1] & 0x400c0) > = 0x40000)&&/*added = for test*/
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||( (input_from_west[cell_number][1] & 0x40000)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
) && fault_register[cell_number] & 0x3c0
||((fault_register[cell_number] & 0x280) = = 0x280)
||((fault_register[cell_number] & 0x340) = = 0x340)
)
  bit[18] = 1;
  /* mf multiple faults in this row  */


if(
  (
  ((fault_register[cell_number] & 0x5) = = 0x5)
  ||( (input_from_east[cell_number][1] & 0x80000)&&
  !(fault_register_delay[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
 ||(((input_from_north[cell_number][1] & 0x40000)&&
 (input_from_north[cell_number][1] & 0x3c0))&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
   north_link[cell_number] = = 3))
 ||  ((input_from_west[cell_number][1] & 0x80000)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 || ((input_from_south[cell_number][1] & 0x2000) &&
 south_link[cell_number] != 18)
 ) && fault_register[cell_number] & 0x7
||((fault_register[cell_number] & 0x3) = = 0x3)
||((fault_register[cell_number] & 0x5) = = 0x5)
||((fault_register[cell_number] & 0x6) = = 0x6)
  )
  bit[19] = 1;
 /* mfn multiple fault in the row to the north */


if(
(
((fault_register[cell_number] & 0x28) = = 0x28)
|| ( (input_from_east[cell_number][1] & 0x100000)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
(east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ( (input_from_south[cell_number][1] & 0x40000)&&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20 ||
   south_link[cell_number] = = 19)  )
|| ((input_from_west[cell_number][1] & 0x100000)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
```

```
)&& fault_register[cell_number] & 0x38
||((fault_register[cell_number] & 0x18) == 0x18)
||((fault_register[cell_number] & 0x28) == 0x28)
||((fault_register[cell_number] & 0x30) == 0x30)
)
bit[20] = 1;
/* mfs multiple fault in the row to the south */

if(
  (((fault_register[cell_number] & 0x100010)== 0x100000)&&
  (fault_register[cell_number] & 0x300)) /* mfs & !sw & w+fw */
  || (((fault_register[cell_number] & 0x80002)==0x80000)  &&
  (fault_register[cell_number] & 0x300)) /*  mfn & !nw ww+fw not*/
  ||((input_from_west[cell_number][1] & 0x1000000)&&
  !(fault_register_delay[cell_number] & 0x400000) &&
 (west_link[cell_number] == 0 || west_link[cell_number] == 2))
  ||( (input_from_east[cell_number][1] & 0x1000000)&&
 (east_link[cell_number] == 23 || east_link[cell_number] == 21))
  || (((fault_register[cell_number] & 0x80044) == 0x80044)&&
 (!(fault_register[cell_number] & 0x3)))
  ||((input_from_north[cell_number][1] & 0x4000000)&&
  ( north_link[cell_number] == 1 || north_link[cell_number] == 4 ||
   north_link[cell_number] == 3))
  || (((fault_register[cell_number] & 0x2000040) == 0x2000040) &&
 !(fault_register[cell_number] & 0x3))
  ||((input_from_south[cell_number][1] & 0x2000000)&&
 ( south_link[cell_number] == 22 || south_link[cell_number] == 20 ||
   south_link[cell_number] == 19) )
  ||(((fault_register_delay[cell_number] & 0x4000010)== 0x4000000) &&
  (fault_register_delay[cell_number] & 0x300))
  ){
 /* fcr fault in the critical region of this row */
  bit[24] = 1;
 }
if(

  (((fault_register[cell_number] & 0x40300)==0x40000) /* mfe */
&&(fault_register[cell_number] & 0x3))
|| ((input_from_east[cell_number][1] & 0x2000000)&&
  !(fault_register_delay[cell_number] & 0x800000) &&
 (east_link[cell_number] == 23 || east_link[cell_number] == 21))
|| ((input_from_west[cell_number][1] & 0x2000000)&&
  !(fault_register_delay[cell_number] & 0x400000) &&
 (west_link[cell_number] == 0 || west_link[cell_number] == 2))
|| ((input_from_north[cell_number][1] & 0x1000000)&&
  ( north_link[cell_number] == 1 || north_link[cell_number] == 4 ||
    north_link[cell_number] == 3))
)    /* fcrn fault in the critical region of the row to the north*/
  bit[25] = 1;

if(
  (((fault_register[cell_number] & 0x40300)==0x40000)
  && (fault_register[cell_number] & 0x18)) /* mfe and not w or fw */
  ||((input_from_west[cell_number][1] & 0x4000000)&&
  !(fault_register_delay[cell_number] & 0x400000) &&
 (west_link[cell_number] == 0 || west_link[cell_number] == 2))
  ||(((input_from_east[cell_number][1] & 0x4000000)
  ||((input_from_east[cell_number][1] & 0x190) == 0x190))&&
  !(fault_register_delay[cell_number] & 0x400000) &&
 (east_link[cell_number] == 23 || east_link[cell_number] == 21))
  ||((input_from_south[cell_number][1] & 0x1000000) &&
  (south_link[cell_number] == 22))
  ||(((fault_register[cell_number] & 0x1080040)==0x1080040) &&
  (( fault_register[cell_number] & 0x18) ||
  (south_east_is_faulty[cell_number]) )&&
  !(fault_register[cell_number] & 0x300) )

/*}fcrs fault in the critical region of the row to the south*/
bit[26] = 1;
}

fault_register[cell_number] = fault_register[cell_number] & 0xff1fffff;
```

```
fault_register[cell_number] = fault_register[cell_number] & 0x00149; /* e00149 */

 for(i = 0; i< = 31; i+ +)
  fault_register[cell_number] = fault_register[cell_number] |
        (bit[i] < < i);


} /* end set registers routine */
```

A.10 Algor3.c

```
/*********************************************************

   File: algor3.c  This file contains code to
set the inter-cell communications links

*********************************************************/
#include < stdio.h >
/******************* GLOBAL VARIABLES  ************************/
extern long fault_register_delayed[100];

extern int north_link[100], south_link[100], east_link[100],
 west_link[100],state[100], row_deformation_east[100][2],
 row_deformation_west[100][2];

void reconfigure(cell_number)

{ /* begin reconfigure */

/* bypass algorithm 2-10 code if row deformation exists in this cell */
if(!(fault_register_delayed[cell_number] & 0xb8e00000) &&
 !(row_deformation_east[cell_number][0]) &&
 !(row_deformation_west[cell_number][0]))
{ /*    no row deformation bits set     */

{   /* algor3b changes rde bits to end of algor1 code */

if(fault_register_delayed[cell_number] & 0x100)
  west_link[cell_number] = 2;
else west_link[cell_number] = 0;

if(fault_register_delayed[cell_number] & 0x40)
  east_link[cell_number] = 21;
else east_link[cell_number] = 23;

/* begin algorithm 2-10 code  */
/* begin north link code */

switch(fault_register_delayed[cell_number] & 0x30c0000)
 {
case 0x0:{
  if(
   (!(fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x3))
   ||((fault_register_delayed[cell_number] & 0x300)
     && fault_register_delayed[cell_number] & 0x2)
    )
  north_link[cell_number] = 1;
else    if(
    ((fault_register_delayed[cell_number] & 0x3)
     && !(fault_register_delayed[cell_number] & 0x300))
    )
  north_link[cell_number] = 4;
else    if(
    ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x2))
```

```
    )
  north_link[cell_number] = 3;
  break; } /* end case */
case 0x1000000:{                    /* fcr */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
    && !(fault_register_delayed[cell_number] & 0x3))
    )
  north_link[cell_number] = 1;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
   && (fault_register_delayed[cell_number] & 0x3))
  ||(!(fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 4;
else  /* not possible in this configuration */
        north_link[cell_number] = 3;

  break; } /* end case 1000000  fcr   */
case 0x2000000:{
        if(
  ((fault_register_delayed[cell_number] & 0x2) &&
!(fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 1;

else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x2) &&
(fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 3;
else    north_link[cell_number] = 4;
   break; } /* end case 2000000 fcrn   */
case 0x3000000:{
  if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
    !(fault_register_delayed[cell_number] & 0x3))
  ||((fault_register_delayed[cell_number] & 0x300)
      && (fault_register_delayed[cell_number] & 0x2))
       )
        north_link[cell_number] = 1;
else    if(
   (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
         )
        north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x2))
         )
        north_link[cell_number] = 3;
  break; } /* end case 3000000 fcr and fcrn */
case 0x40000:
case 0x1040000:{   /* mf or mf and fcr */
        if(
  ((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x3))
  ||(fault_register_delayed[cell_number] & 0x2 &&
!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
(fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0) &&
!(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
```

```
  break; } /* end case 40000 multiple fault mf */
case 0x2040000:
case 0x3040000:{
  if(
  (fault_register_delayed[cell_number] & 0x4)
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 3;
  break; } /* end case 2040000 mf and fcrn */
case 0x80000:
case 0x2080000:{ /* mfn or mfn and fcrn */
  if((fault_register_delayed[cell_number] & 0x7)  = = 0x7)
  north_link[cell_number] = 5;
else
  (!(fault_register_delayed[cell_number] & 0x300)
     && ((fault_register_delayed[cell_number] & 0x6) = = 0x6))
   ||((fault_register_delayed[cell_number] & 0x300)
     && (!(fault_register_delayed[cell_number] & 0x5)))

        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x304))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x5) &&
(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
  break; } /* end case 80000  mfn  */
case 0x1080000:
case 0x3080000:{ /* mfn and fcr */
        if(fault_register_delayed[cell_number] = = 0x80007)
    north_link[cell_number] = 5;

else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x6) = = 0x6)
  )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x4))
  )north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
  break; } /* end case 1080000 mfn and fcr */
case 0xc0000:
case 0x10c0000:
case 0x20c0000:
case 0x30c0000:{/* mf and mfn, fcr and fcrn not critical */
        if(
  (!(fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x3))
  ||((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0) &&
     ((fault_register_delayed[cell_number] & 0x6) = = 0x6))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && !(fault_register_delayed[cell_number] & 0x5))
        )north_link[cell_number] = 1;
else    if(
```

```
  (!(fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0x3))
/*||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x3))   */
  ||((fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x300) &&
      !(fault_register_delayed[cell_number] & 0x4))
        )north_link[cell_number] = 4;
else     if(
  ((fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x2))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x5))
        )north_link[cell_number] = 3;
  break; } /* end case c0000 mf and mfn  */
 } /* end switch north link code */

/*  begin south link code  */
switch(fault_register_delayed[cell_number] & 0x5140000)
{ /* south link switch code */
case 0:
  { /* begin case 0 */
   if(
  (!(fault_register_delayed[cell_number] & 0x300)
    && !(fault_register_delayed[cell_number] & 0x18))
  ||((fault_register_delayed[cell_number] & 0x300)
    && (fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 22;
else     if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else     if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x18))
        )south_link[cell_number] = 20;
  break; } /* end case 0, single faults, no critical area faults */

case 0x1000000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
      && (!(fault_register_delayed[cell_number] & 0x18)))
  )south_link[cell_number] = 22;

else     if(
  ((fault_register_delayed[cell_number] & 0x300)
     && ((fault_register_delayed[cell_number] & 0x18)))
  ||(!(fault_register_delayed[cell_number] & 0x300))
  )south_link[cell_number] = 20;
else    south_link[cell_number] = 19;
  break; } /* end case 0x1000000, fcr */

case 0x4000000:
  { /* begin case  */

  if(
  (!(fault_register_delayed[cell_number] & 0x300)
      && fault_register_delayed[cell_number] & 0x10)
  )south_link[cell_number] = 22;
else     if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x10)
     && (fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 19;
else     south_link[cell_number] = 20;
/* not possible this configuration */
  break; } /* end case fcrs */

case 0x5000000:
  { /* begin case  */
        if(
```

```
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
fault_register_delayed[cell_number] & 0x10)
        )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300)
&&(fault_register_delayed[cell_number] & 0x18))
        )south_link[cell_number] = 20;
  break; } /* end case fcr and fcrs      */

case 0x40000:
case 0x1040000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
&& (fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x18))
          )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xd0))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300)
&& (fault_register_delayed[cell_number] & 0xc0) &&
     (fault_register_delayed[cell_number] & 0x18))
          )south_link[cell_number] = 20;
  break; } /* end case mf */

case 0x4040000:
    { /* begin case  */
        if(
  (fault_register_delayed[cell_number] & 0x20)
  ||((fault_register_delayed[cell_number] & 0x300)
&& (fault_register_delayed[cell_number] & 0xc0))
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )south_link[cell_number] = 19;
else    if(
  ((fault_register_delayed[cell_number] & 0x18) &&
!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 20;
  break; } /* end case mf and fcrs                        */

case 0x100000:
case 0x4100000:
  { /* begin case  */
  if(
   ((fault_register_delayed[cell_number] & 0x38) = = 0x38)
    ||((fault_register_delayed[cell_number] & 0x8000008) = = 0x8000008)
    )
    south_link[cell_number] = 18;
else
  (((fault_register_delayed[cell_number] & 0x30)= = 0x30) &&
     !(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x28) &&
(fault_register_delayed[cell_number] & 0x10) &&
     (fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x320))
        )south_link[cell_number] = 20;
  break; } /* end case mfs */
```

```
case 0x1100000:
case 0x5100000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x30) = = 0x30)
  ||(!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else     if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else     if(
  (!(fault_register_delayed[cell_number] & 0x20))
        )south_link[cell_number] = 20;
  break; } /* end case mfs and fcr */

case 0x140000:
case 0x1140000:
case 0x4140000:
case 0x5140000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0)
  && ((fault_register_delayed[cell_number] & 0x30) = = 0x30))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && !(fault_register_delayed[cell_number] & 0x28))
  ||(!(fault_register_delayed[cell_number] & 0x300)
   && !(fault_register_delayed[cell_number] & 0x18))
  )south_link[cell_number] = 22;
else     if(
  ((fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x10))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x28))
  )south_link[cell_number] = 19;
else     if(
  ((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0)
   && !(fault_register_delayed[cell_number] & 0x20))
   ||(!(fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x28))
   ||(!(fault_register_delayed[cell_number] & 0x300)
   &&(fault_register_delayed[cell_number] & 0x18))
   )south_link[cell_number] = 20;
  break; } /* end case mf and mfs */


} /* end south link switch code */


} /* end code for row-deformation east or west not */
/*****************************************************************************
   CODE BELOW THIS POINT ADDED TO ALGORITHM 1
         TO ENABLE ALGORITHM 3

*****************************************************************************/
} /*  end code for no deformation bits set    */
else {

if((fault_register_delayed[cell_number] & 0x38000000)&&
!(fault_register_delayed[cell_number] & 0xe00000) &&
 !(row_deformation_east[cell_number][0]) &&
 !(row_deformation_west[cell_number][0]))
  {  north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0;  }
/*****   the above code is not complete    not all possibilities
   are simulated                  ******************************/
else if(row_deformation_east[cell_number][0]  &&
```

```
  !row_deformation_west[cell_number][0]){

if(fault_register_delayed[cell_number] & 0x312)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
        }
else if(fault_register_delayed[cell_number] & 0x28)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 19;
                        }
else if(fault_register_delayed[cell_number] & 0x80)
 {  north_link[cell_number] = 4;
  south_link[cell_number] = 20;
   }
else if(fault_register_delayed[cell_number] & 0x40)
 {  north_link[cell_number] = 4;
  south_link[cell_number] = 20;
  }
else if(fault_register_delayed[cell_number] & 0x5)
 {  north_link[cell_number] = 3;
  south_link[cell_number] = 22;
 }
if(fault_register_delayed[cell_number] & 0x100)
  { east_link[cell_number] = 23;
  west_link[cell_number] = 2;  }
if(fault_register_delayed[cell_number] & 0x40)
  { east_link[cell_number] = 21;
  west_link[cell_number] = 0;  }
if(fault_register_delayed[cell_number] = = 0x80003)
 { north_link[cell_number] = 5;
        }

/* end row deformation east */ }

else if(row_deformation_west[cell_number][0]  &&
!row_deformation_east[cell_number][0]){

if(fault_register_delayed[cell_number] & 0xe4)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
        }
else if(fault_register_delayed[cell_number] & 0x18)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 20;
        }
else if(fault_register_delayed[cell_number] & 0x3)
 {  north_link[cell_number] = 3;
  south_link[cell_number] = 19;
   }
else if(fault_register_delayed[cell_number] & 0x300)
 {  north_link[cell_number] = 3;
  south_link[cell_number] = 19;
  }
if(fault_register_delayed[cell_number] & 0x100)
  { east_link[cell_number] = 23;
  west_link[cell_number] = 2;  }
if(fault_register_delayed[cell_number] & 0x40)
  { east_link[cell_number] = 21;
  west_link[cell_number] = 0;  }

 /* end row deformation west */ }
else if(row_deformation_east[cell_number][0] &&
 row_deformation_west[cell_number][0] )
  { if(fault_register_delayed[cell_number] & 0x1)
 north_link[cell_number] = 5;
else north_link[cell_number] = 1;
 south_link[cell_number] = 22;
 east_link[cell_number] = 23;
 west_link[cell_number] = 0;
  }

if(
```

```
  ((fault_register_delayed[cell_number] & 0x7) = = 0x7)
||((fault_register_delayed[cell_number] & 0x20000001) = = 0x20000001)
  )
 {  north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0;  }
if(
  ((fault_register_delayed[cell_number] & 0x8e00008) = = 0x8000008)
  )
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0;  }

/*****************************************************************************

BEGIN CODE FOR ROW DEFORMATIONS

*****************************************************************************/
 if(
  ((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600001)
  ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00001)
  ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200001)
  ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00001)
) north_link[cell_number] = 5;

else if(
 ((fault_register_delayed[cell_number] & 0x80e00008) = = 0x800000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x400000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00000)
||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80800000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80400000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00000)
||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200000)
) north_link[cell_number] = 1;


if(
((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200008)
 )south_link[cell_number] = 18;

else if(
((fault_register_delayed[cell_number] & 0x80e00008) = = 0x400000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x800000)
||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80400000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200001)
||((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80800000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600001)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00001)
 ||((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00001)
```

```
) south_link[cell_number] = 22;

if(
((fault_register_delayed[cell_number] & 0x80e00001) == 0x80200000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80200001)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x400000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80a00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80a00001)
) east_link[cell_number] = 4;

else if(
((fault_register_delayed[cell_number] & 0x80e00008) == 0xa00000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x200008)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80400000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xa00008)
||((fault_register_delayed[cell_number] & 0x80e00008) == 0x200000)
) east_link[cell_number] = 20;

else if(
 ((fault_register_delayed[cell_number] & 0x80e00008) == 0x600000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x600008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xe00000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xe00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x800000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80600000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80800000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80e00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80600001)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80e00001)
  east_link[cell_number] = 23;


if(
  ((fault_register_delayed[cell_number] & 0x80e00001) == 0x80200000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80200001)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x800000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80600000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80800000)
 west_link[cell_number] = 3;

else if(
  ((fault_register_delayed[cell_number] & 0x80e00008) == 0x200008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x600000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x600008)
||((fault_register_delayed[cell_number] & 0x80e00008) == 0x200000)
  west_link[cell_number] = 19;

else if(
((fault_register_delayed[cell_number] & 0x80e00008) == 0xe00000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xa00000)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xe00008)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0xa00008)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80400000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80e00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80600001)
 ||((fault_register_delayed[cell_number] & 0x80e00008) == 0x400000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80a00000)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80a00001)
 ||((fault_register_delayed[cell_number] & 0x80e00001) == 0x80e00001)
 west_link[cell_number] = 0;

 } /* end deformation code */

} /* end reconfigure */
```

A.11 Faultrg3.c

```
/*************************************************************
file: set_f_rg.c set fault registers, this routine takes the input
from the receive routine and sets the appropriate fault register
bits

*************************************************************************/
#include < stdio.h >
extern long
input_from_north[100][2],
input_from_south[100][2],
input_from_east[100][2],
input_from_west[100][2],
fault_register[100],
fault_register_delayed[100];
extern
north_link[100],
south_link[100],
east_link[100],
west_link[100],
state[100],
north_west_is_faulty[100],
north_east_is_faulty[100],
south_west_is_faulty[100],
south_east_is_faulty[100],
fareast_is_faulty[100],
farwest_is_faulty[100],
row_deformation_east[100][2],
row_deformation_west[100][2],
row_deformation_northeast_blocker[100],
row_deformation_northwest_blocker[100];
extern long fault_register1[100];
void set_fault_register(cell_number)

{
int i;
long bit[34];

 fault_register_delayed[cell_number] = fault_register[cell_number];
 row_deformation_east[cell_number][1] =
 row_deformation_east[cell_number][0];
 row_deformation_west[cell_number][1] =
 row_deformation_west[cell_number][0];

for(i=0; i< = 33; i++)
bit[i] = 0;

/*************************************************************************
SET AUXILLARY BITS, NE IS FAULTY, SE IS FAULTY, NW IS FAULTY ANS SW IS
FAULTY THESE BITS ARE TRANSMITTED, BUT ARE NOT USED IN DECODING
*************************************************************************/
if(
  ((fault_register[cell_number] & 0x80200001)= = 0x200001)
  ||(input_from_north[cell_number][0] & 0x4000000)
  ||(((input_from_east[cell_number][0] & 0x4000000)
  ||(input_from_west[cell_number][0] & 0x4000000))
&&
  (((fault_register[cell_number] & 0x200008)= = 0x200008)||
 ((fault_register[cell_number] & 0x400040) = = 0x400040)))

)
 row_deformation_northeast_blocker[cell_number] = 1;
if(
  ((fault_register[cell_number] & 0x80200001)= = 0x200001)
  ||(input_from_north[cell_number][0] & 0x8000000)
  ||(((input_from_east[cell_number][0] & 0x8000000) ||
  (input_from_west[cell_number][0] & 0x8000000)) &&
  (((fault_register[cell_number] & 0x200008)= = 0x200008)||
```

```
    ((fault_register[cell_number] & 0x800100) == 0x800100)))
   )
   row_deformation_northwest_blocker[cell_number] = 1;

 if(

 ((input_from_east[cell_number][1] & 0x400000)&&
   !(fault_register[cell_number] & 0x400000)
  &&  (fault_register[cell_number] & 0x312)) /* was 3ff */
   ||((input_from_east[cell_number][0] & 0x1000000)
  &&  (fault_register[cell_number] & 0x3ff))
 ||((input_from_south[cell_number][0] & 0x1000000) &&
  fault_register[cell_number] & 0x140)
   ||((input_from_south[cell_number][0] & 0x1000000) &&
   south_link[cell_number] == 18)
  ||((row_deformation_east[cell_number][1] &&
  row_deformation_west[cell_number][1]))
  ){
 bit[32] = 1; }
 if(

 ((input_from_west[cell_number][1] & 0x800000) &&
    !(fault_register[cell_number] & 0x800000)
  && (fault_register[cell_number] & 0xe4)) /* was 3ff*/
 || ((input_from_west[cell_number][0] & 0x2000000)&&
   (fault_register_delayed[cell_number] & 0x3ff) )
 ||((input_from_south[cell_number][0] & 0x2000000) &&
  fault_register[cell_number] & 0x140)

 ||((input_from_south[cell_number][0] & 0x2000000) &&
  south_link[cell_number] == 18)
  ||((row_deformation_east[cell_number][1] &&
  row_deformation_west[cell_number][1]))
   ||(((input_from_north[cell_number][1] & 0x80800000) == 0x800000)  &&
   (fault_register[cell_number] & 0x80) &&
   (north_link[cell_number] == 1))

 )
 bit[33] = 1;
 if(
   (((input_from_north[cell_number][0] & 0xf) == 0xf)
    && (north_link[cell_number] == 4))
   ||((input_from_east[cell_number][1] & 0x1) &&
   east_link[cell_number] == 23)
   )
   north_east_is_faulty[cell_number] = 1;

 if(
 (((input_from_north[cell_number][0] & 0xf) == 0xf)
   && (north_link[cell_number] == 3))
   ||((input_from_west[cell_number][1] & 0x1) &&
  west_link[cell_number] == 0)
   )
   north_west_is_faulty[cell_number] = 1;
 if(
   (((input_from_south[cell_number][0] & 0xf) == 0xf)
   && (south_link[cell_number] == 20))
   ||((input_from_east[cell_number][1] & 0x8) &&
  east_link[cell_number] == 23)
   )
   south_east_is_faulty[cell_number] = 1;
 if(
 (((input_from_south[cell_number][0] & 0xf) == 0xf)
  && (south_link[cell_number] == 19))
   ||((input_from_west[cell_number][1] & 0x8)&&
  west_link[cell_number] == 0)
   )
 south_west_is_faulty[cell_number] = 1;
 if(
 (((input_from_east[cell_number][0] & 0xf) == 0xf)
   &&(east_link[cell_number] == 21))
   ||((input_from_south[cell_number][0] & 0x40000000) &&
```

```
 (south_link[cell_number]= = 20))
  ||((input_from_east[cell_number][0] & 0x80000000)
 && east_link[cell_number] = = 4)
  )
  fareast_is_faulty[cell_number] = 1;

if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf)
  &&(west_link[cell_number] = = 2))
  )
  farwest_is_faulty[cell_number] = 1;



/*****************************************************************
RECONFIGURATION IS DONE BASED ON THE CONTENTS OF THE delayedED
FAULT REGISTER, BUT INPUT IS USED TO SET THE CONTENTS OF THE
FAULT REGISTER. THIS ALLOWS A CELL TO COMMUNICATE TO ITS
CURRENT NEIGHBORS ALL THE INPUT IT RECEIVED FROM ALL ITS
NEIGHBORS BEFORE DISCONNECTING FROM A CELL
*****************************************************************/

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 1))
  || ((input_from_west[cell_number][0] & 0x40000000)  &&
  (west_link[cell_number] = = 0) )
|| ( (input_from_east[cell_number][0] & 0x10000000)  &&
  (east_link[cell_number] = = 23) )
  )
  {
  fault_register[cell_number] = fault_register[cell_number] | 0x1;
  } /* immediate cell to the north is faulty */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = =  3))
  || ((input_from_north[cell_number][1] & 0x300) &&
  (( north_link[cell_number] = = 1) || (north_link[cell_number] = = 3)))
  || (( input_from_north[cell_number][1] & 0x200) &&
 north_link[cell_number] = = 4)
  ||((input_from_west[cell_number][1] & 0x3)&&
((input_from_west[cell_number][1] & 0xe00000) != 0x800000)&&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][0] & 0x40000000) &&
      (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
  west_link[cell_number] = = 3)
  || north_west_is_faulty[cell_number]
  )
  bit[1] = 1;
  /* fault north and west of this cell */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x5)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ((input_from_north[cell_number][1] & 0xc0) &&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4))
||((input_from_east[cell_number][1] & 0xc0) &&
   east_link[cell_number] = = 4)

)
  bit[2] = 1;
/* fault north and east of this cell */

if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 22))
```

```
  || ((input_from_west[cell_number][0] & 0x80000000) &&
    (west_link[cell_number] = = 0))
  || ((input_from_east[cell_number][0] & 0x20000000) &&
    (east_link[cell_number] = = 23))
   )
   fault_register[cell_number] = fault_register[cell_number] | 0x8;
   /* immediate southern neighbor is faulty */

 if(

   (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
  && (south_link[cell_number] = = 19))
 ||((input_from_west[cell_number][1] & 0x18) &&
 !((input_from_west[cell_number][1] & 0xe00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 ||((input_from_west[cell_number][0] & 0x80000000) &&
   west_link[cell_number] = = 2)
 ||((input_from_south[cell_number][1] & 0x300) &&
   south_link[cell_number] = = 22 )
 ||((input_from_west[cell_number][1] & 0x300) &&
  west_link[cell_number] = = 19)
 || ((input_from_west[cell_number][1] & 0x8) &&
    west_link[cell_number] = = 0)
 ||((input_from_south[cell_number][1] & 0x2) &&
  south_link[cell_number] = = 18)
  )
 bit[4] = 1;
   /* fault to the southwest of this cell */

 if(
   (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
  && (south_link[cell_number] = = 20))
 ||((input_from_east[cell_number][1] & 0x28)&&
   !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
   (east_link[cell_number] = = 21 || east_link[cell_number] = = 23))
 || ((input_from_south[cell_number][1] & 0xc0)  &&
  ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20)  )
 ||((input_from_east[cell_number][1] & 0xc0) &&
  east_link[cell_number] = = 20)
 ||((input_from_south[cell_number][1] & 0x4) &&
  south_link[cell_number] = = 18)

 )

 bit[5] = 1;
  /* fault to the southeast of this cell */


 if(
   (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
  && (east_link[cell_number] = = 23))
   )

   fault_register[cell_number] = fault_register[cell_number] | 0x40;
   /* immediate eastern neighbor is faulty */


 if(
  (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
  && (east_link[cell_number] = = 21))
   || ((input_from_east[cell_number][1] & 0xc0)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
   || ((input_from_east[cell_number][1] & 0x20) &&
 east_link[cell_number] = = 4)
   || ((input_from_east[cell_number][1] & 0x4) &&
   east_link[cell_number] = = 20)
   ||fareast_is_faulty[cell_number]
 )
   bit[7] = 1;
   /* faulty cell to  the far east in this row */
```

```
if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 0))
  )
 {
  fault_register[cell_number] = fault_register[cell_number] | 0x100;
  /* immediate western neighbor is faulty */
}
if(
  ((((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
  !((input_from_west[cell_number][1] & 0xe00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x10) &&
  west_link[cell_number] = = 3)
  || ((input_from_west[cell_number][1] & 0x2) &&
  west_link[cell_number] = = 19))
/*  &&!(input_from_west[cell_number][1] & 0x20000)*/
  ||farwest_is_faulty[cell_number]
  )
bit[9] = 1;
 /* fault to the far west of this cell */


 if(
((input_from_north[cell_number][1] & 0x1) &&
  (north_link[cell_number] = = 1) )
|((input_from_north[cell_number][0] & 0x10000000) &&
  north_link[cell_number] = = 4)
||((input_from_north[cell_number][0] & 0x40000000) &&
  north_link[cell_number] = = 3)
 )
  bit[10] = 1;
/* nn-  fault 2 rows to the north and north of this cell*/


if(
((input_from_north[cell_number][1] & 0x2) &&
(north_link[cell_number] = = 1))
||((input_from_north[cell_number][1] & 0x3) &&
 (north_link[cell_number] = = 0x3))
|| ((input_from_west[cell_number][1] & 0xc00)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
)
bit[11] = 1;
/* nnw- fault 2 rows north and west of this cell*/


if(
(input_from_north[cell_number][1] & 0x4)
&& (north_link[cell_number] = = 1)
||((input_from_north[cell_number][1] & 0x5) &&
 (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x1400)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
  bit[12] = 1;
/* nne- fault 2 rows north and east of this cell*/


if(
(
((input_from_north[cell_number][1] & 0x80000) &&
 ((north_link[cell_number] = = 1) || (north_link[cell_number] = = 4) ) )
|| ((input_from_east[cell_number][1] & 0x2000)&&
  !((input_from_west[cell_number][1] & 0x600000) = = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||((input_from_west[cell_number][1] & 0x2000)&&
```

```
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
   )
 && fault_register[cell_number] & 0x1c00
)
bit[13] = 1;
 /*mfnne- multiple faults 2 rows north and east of this cell*/


if(
(input_from_south[cell_number][1] & 0x8)  &&
  (south_link[cell_number] = = 22)
|| ((input_from_south[cell_number][0] & 0x20000000) &&
  (south_link[cell_number] = = 20))
|| ((input_from_south[cell_number][0] & 0x80000000) &&
 (south_link[cell_number] = = 19))
)
bit[14] = 1;
/* ss- fault 2 rows to the south and in the same column */


if(
 ((input_from_south[cell_number][1] & 0x10) &&
 (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x18) &&
 (south_link[cell_number] = = 19))
||((input_from_west[cell_number][1] &  0xc000) &&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
){
bit[15] = 1;
 }
/* ssw- single fault 2 rows south and west of this cell*/


if(
((input_from_south[cell_number][1] & 0x20)  &&
  (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x28) &&
  (south_link[cell_number] = = 20))
||(((input_from_east[cell_number][1] & 0x14000))&&
  !((input_from_east[cell_number][1] & 0x600000) = = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
bit[16] = 1;
/* sse- single fault 2 rows south and east of this cell*/
else bit[16] = 0;

 if(
 (
 ((input_from_south[cell_number][1] & 0x100000)&&
  ((south_link[cell_number] = = 22) || (south_link[cell_number] = = 20) ))
||((input_from_east[cell_number][1] & 0x20000)&&
 !(fault_register_delayed[cell_number] & 0x800000) &&
(east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||(((input_from_west[cell_number][1] & 0x20000))&&
 !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 )&& fault_register[cell_number] & 0x1c000
 )
bit[17] = 1;
 /* mfss- multiple faults 2 rows to the south of cell*/


if(
(
((fault_register[cell_number] & 0xc0) = = 0xc0)
||((fault_register[cell_number] & 0x300) = = 0x300)
||( ((input_from_east[cell_number][1] & 0x400c0) > = 0x40000)&&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||( (input_from_west[cell_number][1] & 0x40000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
```

```
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
) && fault_register[cell_number] & 0x3c0
||((fault_register[cell_number] & 0x280) = = 0x280)
||((fault_register[cell_number] & 0x340) = = 0x340)
)
  bit[18] = 1;
  /* mf multiple faults in this row */


if(
  (
  ((fault_register[cell_number] & 0x5) = = 0x5)
  ||( (input_from_east[cell_number][1] & 0x80000)&&
  !(fault_register_delayed[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
 ||(((input_from_north[cell_number][1] & 0x40000)&&
 (input_from_north[cell_number][1] & 0x3c0))&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
    north_link[cell_number] = = 3))
 ||  ((input_from_west[cell_number][1] & 0x80000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 || ((input_from_south[cell_number][1] & 0x2000) &&
 south_link[cell_number] != 18)
 ) && fault_register[cell_number] & 0x7
||((fault_register[cell_number] & 0x3) = = 0x3)
||((fault_register[cell_number] & 0x5) = = 0x5)
||((fault_register[cell_number] & 0x6) = = 0x6)
  )
  bit[19] = 1;
 /* mfn multiple fault in the row to the north */


if(
(
((fault_register[cell_number] & 0x28) = = 0x28)
|| ( (input_from_east[cell_number][1] & 0x100000)&&
  !((input_from_east[cell_number][1] & 0x600000) = = 0x400000) &&
(east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ( (input_from_south[cell_number][1] & 0x40000)&&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20 ||
   south_link[cell_number] = = 19) )
|| ((input_from_west[cell_number][1] & 0x100000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
)&& fault_register[cell_number] & 0x38
||((fault_register[cell_number] & 0x18) = = 0x18)
||((fault_register[cell_number] & 0x28) = = 0x28)
||((fault_register[cell_number] & 0x30) = = 0x30)
)
bit[20] = 1;
/* mfs multiple fault in the row to the south */

if(
  (((fault_register[cell_number] & 0x100010) = = 0x100000)&&
  (fault_register[cell_number] & 0x300)) /* mfs & !sw & w+fw */
  || (((fault_register[cell_number] & 0x80002) = =0x80000)  &&
  (fault_register[cell_number] & 0x300)) /*  mfn & !nw ww+fw not*/
  ||((input_from_west[cell_number][1] & 0x1000000)&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  ||( (input_from_east[cell_number][1] & 0x1000000)&&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
  || (((fault_register[cell_number] & 0x80044) = = 0x80044)&&
 (!(fault_register[cell_number] & 0x3)))
  ||((input_from_north[cell_number][1] & 0x4000000)&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
    north_link[cell_number] = = 3))
  || (((fault_register[cell_number] & 0x2000040) = = 0x2000040) &&
 !(fault_register[cell_number] & 0x3))
  ||((input_from_south[cell_number][1] & 0x2000000)&&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20 ||
```

```
   south_link[cell_number] = = 19) )
  ||(((fault_register_delayed[cell_number] & 0x4000010)= = 0x4000000) &&
     (fault_register_delayed[cell_number] & 0x300))
  ){
  /* fcr fault in the critical region of this row */
   bit[24] = 1;
  }
if(

   (((fault_register[cell_number] & 0x40300)= =0x40000) /* mfe */
&&(fault_register[cell_number] & 0x3))
|| ((input_from_east[cell_number][1] & 0x2000000)&&
  !(fault_register_delayed[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ((input_from_west[cell_number][1] & 0x2000000)&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
|| ((input_from_north[cell_number][1] & 0x1000000)&&
   ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
    north_link[cell_number] = = 3))
)    /* fcrn fault in the critical region of the row to the north*/
   bit[25] = 1;

if(
   (((fault_register[cell_number] & 0x40300)= =0x40000)
   && (fault_register[cell_number] & 0x18)) /* mfe and not w or fw */
   ||((input_from_west[cell_number][1] & 0x4000000)&&
   !(fault_register_delayed[cell_number] & 0x400000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
   ||(((input_from_east[cell_number][1] & 0x4000000)
   ||((input_from_east[cell_number][1] & 0x190) = = 0x190))&&
   !(fault_register_delayed[cell_number] & 0x400000) &&
  (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
   ||((input_from_south[cell_number][1] & 0x1000000) &&
   (south_link[cell_number] = = 22))
   ||(((fault_register[cell_number] & 0x1080040)= =0x1080040) &&
   (( fault_register[cell_number] & 0x18) ||
   (south_east_is_faulty[cell_number]) )&&
   !(fault_register[cell_number] & 0x300) )

/*){fcrs fault in the critical region of the row to the south*/
bit[26] = 1;
}

/*************************************************************************

CODE TO FORCE ROW DEFORMATION.  ROW DEFORMATION TAKES PLACE WITH ALL
INTERIOR FAULT OCCURENCES, THAT IS ALL BUT THE FARTHEST EAST AND WEST
FAULT IN MULTIPLE FAULT OCCURENCES.  ROW DEFORMATION TO THE NORTH
IS ATTEMPTED FIRST, AND IF IT DOES NOT ENCOUNTER A DOUBLE FAULT EITHER
TO THE EAST OR WEST OF THE COLUMN AFFECTED, AND IT DOES NOT ENCOUNTER
ANOTHER FAULT IN THE SAME COLUMN, IT PROCEEDES UNTIL A SPARE CELL IN THE
TOP ROW IS USED.  IF EITHER OF THE ABOVE CASES EXISTS, THE ROW
DEFORMATION TO THE NORTH IS CANCELLED AND DEFORMATION TO THE SOUTH
IS ATTEMPTED.  IF THIS IS UNSUCCESSFUL, AN IRRECOVERABLE CONDITION EXISTS
AND THE GLOBAL PATTERN REGROWTH ALGORITHM MUST BE CALLED
*************************************************************************/
if(
   ((((fault_register[cell_number] & 0x100038) = = 0x100038)/* center of three faults */
   ||((fault_register[cell_number] & 0x8) &&       /* adjacent faults*/
   ((input_from_west[cell_number][1] & 0x8) ||
  (input_from_east[cell_number][1] & 0x8)||
  (south_east_is_faulty[cell_number])||
  south_west_is_faulty[cell_number])))
&&   /* adjacent faulty cells */
  ( !row_deformation_northeast_blocker[cell_number] &&
   !row_deformation_northwest_blocker[cell_number]))
   ||((((fault_register[cell_number] & 0x80007) = = 0x80007)/* center of three faults */
   ||((fault_register[cell_number] & 0x1) &&       /* adjacent faults*/
    ( (input_from_west[cell_number][1] & 0x1) ||
     (input_from_east[cell_number][1] & 0x1)||
     (north_east_is_faulty[cell_number])||
```

```
    north_west_is_faulty[cell_number] ) ))
 &&( row_deformation_northeast_blocker[cell_number] &&
  row_deformation_northwest_blocker[cell_number]))


  || ((input_from_south[cell_number][1] & 0x200000)&&
  (south_link[cell_number] = = 22) &&
  !row_deformation_northeast_blocker[cell_number] &&
  !row_deformation_northwest_blocker[cell_number])

 &&((input_from_north[cell_number][1] & 0x80200000) = = 0x80200000)
  (north_link[cell_number] = = 1))
  ||((input_from_south[cell_number][1] & 0x400000) &&
 south_link[cell_number] = = 19)
  ||((input_from_south[cell_number][1] & 0x800000) &&
 south_link[cell_number] = = 20)
  ||(((fault_register[cell_number] & 0x18) = = 0x18) &&
  row_deformation_east[cell_number][1]&&
  !(row_deformation_northeast_blocker[cell_number]) &&
  !(row_deformation_northwest_blocker[cell_number])
  )
  ||((fault_register[cell_number] & 0x8) &&
  row_deformation_east[cell_number][1] &&
  row_deformation_west[cell_number][1])

  )
 bit[21] = 1;    /* row  deform north */


if(
(((fault_register[cell_number] & 0xc0) = = 0xc0)&&/* e & fe */
 (fault_register[cell_number] & 0x300))            /* & (w + fw) */
||(((fault_register[cell_number] & 0xc0) = = 0xc0) &&
  (fareast_is_faulty[cell_number]))
||((input_from_south[cell_number][1] & 0x80400000) = = 0x400000) &&
  (south_link[cell_number] = = 22)
 &&((input_from_north[cell_number][1] & 0x80400000) = = 0x80400000)
  (north_link[cell_number] = = 1))
  ||(((fault_register[cell_number] & 0x80200000) = = 0x80200000) &&
  north_east_is_faulty[cell_number])
  ||((fault_register[cell_number] & 0x8) &&
  south_east_is_faulty[cell_number])
  ||((input_from_east[cell_number][1] & 0x200000) &&
 east_link[cell_number] = = 23)
  ||((fault_register[cell_number] & 0x300) &&
 (fault_register[cell_number] & 0x40) &&
 row_deformation_east[cell_number][0] )
||((fault_register[cell_number] & 0x40) &&
 row_deformation_east[cell_number][0] &&
 row_deformation_west[cell_number][0] )
){
  bit[22] = 1;  /* row deformation north or south east */
 }
if(
  (((fault_register[cell_number] & 0x40300) = = 0x40300)&& /* w & fw */
 (fault_register[cell_number] & 0xc0))       /* and e + fe */
  || ((input_from_south[cell_number][1] & 0x800000)&&
 (south_link[cell_number] = = 22) &&
 !(input_from_south[cell_number][0] & 0x8000000))
  ||
(((input_from_north[cell_number][1] & 0x80800000) = = 0x80800000)  &&
  (north_link[cell_number] = = 1))
  ||((input_from_west[cell_number][1] & 0x200000)/*&&
  (fault_register[cell_number] & 0x50)*/)  /* was 10 */
  ||(south_west_is_faulty[cell_number] &&
  fault_register[cell_number] & 0x200000)
  ||(north_west_is_faulty[cell_number] &&
  ((fault_register[cell_number] & 0x80200000) = = 0x80200000))
||((input_from_north[cell_number][1] & 0x800000) &&
 (fault_register[cell_number] & 0x10000000)  &&
 (north_link[cell_number] = = 1))
  ||(
```

```
  (((fault_register[cell_number] & 0x40300) = = 0x40300) &&
  farwest_is_faulty[cell_number]) ||
   ((( fault_register[cell_number] & 0x80001) = = 0x80001) &&
  north_west_is_faulty[cell_number])
   && row_deformation_northwest_blocker[cell_number])
   ||(((fault_register[cell_number] & 0x40300) = = 0x40300) &&
   (row_deformation_east[cell_number][1]))
   ||((fault_register[cell_number] & 0x100) &&
   row_deformation_east[cell_number][1] &&
   row_deformation_west[cell_number][1])
   ||((input_from_west[cell_number][0] = = 0xf)&&
   west_link[cell_number] = = 2)
   ) {
   bit[23] = 1;  /* row deformation north or southwest */
}

   if(
  ((fault_register[cell_number] & 0x8) &&
  ((input_from_west[cell_number][1] & 0x8) ||
   (input_from_east[cell_number][1] & 0x8)))
||((input_from_east[cell_number][1] & 0x8000000) &&
   (east_link[cell_number] > = 21) &&
   !(fault_register[cell_number] & 0x10))
||((input_from_west[cell_number][1] & 0x8000000) &&
  (west_link[cell_number] < = 2) &&
  !(fault_register[cell_number] & 0x20))/**/
  )
   bit[27] = 1; /* row deformation generated row [i+1] */
/*  this bit used to cancel the multiple fault effects of
     adjacent faults in the  row to the south */
if(
   ((fault_register[cell_number] & 0x100) &&
  farwest_is_faulty[cell_number])
((input_from_west[cell_number][1] & 0xf) = = 0xf) &&
   (west_link[cell_number] = = 2))
||((fault_register[cell_number] & 0x40) &&
  fareast_is_faulty[cell_number])
((input_from_east[cell_number][1] & 0xf) = = 0xf) &&
   (east_link[cell_number] = = 21))
*/  ||((input_from_east[cell_number][1] & 0x10000000) &&
  !(fault_register[cell_number] & 0x300))
  ||((input_from_west[cell_number][1] & 0x10000000) &&
  !(fault_register[cell_number] & 0xc0))
  )
bit[28] = 1;
/* above bit cancels multiple fault effect of
adjacent faults in this row */

if(
  ((fault_register[cell_number] & 0x1) &&
  (((input_from_west[cell_number][1] & 0x1) &&
west_link[cell_number] = = 0)
  ||((input_from_east[cell_number][1] & 0x1) &&
  east_link[cell_number] = = 23)))
||((input_from_east[cell_number][1] & 0x20000000) &&
  (east_link[cell_number] > = 21) &&
  !(fault_register[cell_number] & 0x2))
||((input_from_west[cell_number][1] & 0x20000000) &&
  (west_link[cell_number] < = 2) &&
  !(fault_register[cell_number] & 0x104))
)
bit[29] = 1;
/* this bit used to cancel the effect of adjacent
faults in northern row*/


if(
(((fault_register[cell_number] & 0x80007) = = 0x80007) &&
 row_deformation_northeast_blocker[cell_number] &&
 row_deformation_northwest_blocker[cell_number])
||
(((fault_register[cell_number] & 0x402c0) = = 0x402c0) &&
```

```
  row_deformation_northeast_blocker[cell_number])
||
(((fault_register[cell_number] & 0x40380) = = 0x40380) &&
  row_deformation_northwest_blocker[cell_number])
||
(input_from_north[cell_number][1] & 0x80000000)
|| (((fault_register[cell_number] & 0x40300) = = 0x40300) &&
 row_deformation_northwest_blocker[cell_number])
|| (((fault_register[cell_number] & 0x400c0) = = 0x400c0) &&
 row_deformation_northeast_blocker[cell_number])
|| (((fault_register[cell_number] & 0x20080001) = = 0x20080001)&&
 row_deformation_northeast_blocker[cell_number] &&
 row_deformation_northwest_blocker[cell_number])


  )
bit[31] = 1; /* this bit forces deformation bits to deform to the south*/

/*************************************************************************
The remaining code clears all bits in the fault register execpt those
set by direct testing of neighbors at all times except during
reconfiguration ( indicated by a "0" input on one of the inputs. Since
border cells never have all inputs, this check is disabled for cell
numbers on the top row because these cells would normally have
an external input here. Code also clears the bits used to generate
north deformation whenever a faulty cell is encountered in the
deformation path
*************************************************************************/
fault_register[cell_number] = fault_register[cell_number] & 0xff1fffff;


/*if(
input_from_north[cell_number][1] &&
input_from_south[cell_number][1] && */
input_from_east[cell_number][1] &&
input_from_west[cell_number][1])
  ||(cell_number < 10) || (((cell_number + 1) %10) = = 0)||
 ((cell_number % 10) = = 0)|| (cell_number > 89)
  )*/{
fault_register[cell_number] = fault_register[cell_number] & 0x00149;
row_deformation_east[cell_number][0] = 0;
row_deformation_west[cell_number][0] = 0;
}

 if(
((input_from_south[cell_number][1] & 0x8) )
&& (input_from_south[cell_number][0] & 0x4000000)
)
{
bit[21] = 0; bit[22] = 0;
}
 if(
((input_from_south[cell_number][1] & 0x8) )
&& (input_from_south[cell_number][0] & 0x8000000)
)
{
bit[21] = 0; bit[23] = 0;
}

 for(i = 0; i< = 31; i+ + )
  fault_register[cell_number] = fault_register[cell_number] |
       (bit[i] < < i);
  row_deformation_east[cell_number][0] =
       row_deformation_east[cell_number][0] | bit[32];
  row_deformation_west[cell_number][0] =
  row_deformation_west[cell_number][0] | bit[33];

if((fault_register[cell_number] & 0x200009) = = 0x200000)
 fault_register[cell_number] = fault_register[cell_number] & 0xfffffc00;
if((row_deformation_northeast_blocker[cell_number])&&
fault_register[cell_number] & 0x8)
fault_register[cell_number] = fault_register[cell_number]
```

```
        & 0xff1fffff;

} /* end set registers routine */
```

A.12 Algor4.c

```
/*********************************************************

File: algor4a.c  This file contains code to
set the inter-cell communications links

*********************************************************/
#include < stdio.h >
/******************* GLOBAL VARIABLES  ************************/
extern long fault_register_delayed[100];

extern int north_link[100], south_link[100], east_link[100],
 west_link[100],state[100], row_deformation_east[100][2],
 row_deformation_west[100][2],
 row_deformation_north_blocker[100],
 row_deformation_northeast_blocker[100],
 row_deformation_northwest_blocker[100],
 east_is_faulty[100],west_is_faulty[100],
 north_west_is_faulty[100],north_east_is_faulty[100],
 south_west_is_faulty[100],south_east_is_faulty[100];

void reconfigure(cell_number)

{ /* begin reconfigure */

/* bypass algorithm 2-10 code if row deformation exists in this cell */
if(!(fault_register_delayed[cell_number] & 0xb8e00000))
{ /*    no row deformation bits set     */

if(!(row_deformation_east[cell_number][0]||
row_deformation_west[cell_number][0]))
{  /* logic changes if there is a deformation east or west of cell*/

if(fault_register_delayed[cell_number] & 0x100)
  west_link[cell_number] = 2;
else west_link[cell_number] = 0;

if(fault_register_delayed[cell_number] & 0x40)
  east_link[cell_number] = 21;
else east_link[cell_number] = 23;

/* begin algorithm 2-10 code  */
/* begin north link code */

switch(fault_register_delayed[cell_number] & 0x30c0000)
 {
case 0x0:{
  if(
   (!(fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x3))
   ||((fault_register_delayed[cell_number] & 0x300)
     && fault_register_delayed[cell_number] & 0x2)
    )
  north_link[cell_number] = 1;
else    if(
    ((fault_register_delayed[cell_number] & 0x3)
     && !(fault_register_delayed[cell_number] & 0x300))
    )
  north_link[cell_number] = 4;
else    if(
    ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x2))
    )
```

```
  north_link[cell_number] = 3;
  break; } /* end case */
case 0x1000000:{              `          /* fcr */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
    && !(fault_register_delayed[cell_number] & 0x3))
   )
  north_link[cell_number] = 1;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
   && (fault_register_delayed[cell_number] & 0x3))
  ||(!(fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 4;
else  /* not possible in this configuration */
        north_link[cell_number] = 3;

  break; } /* end case 1000000  fcr  */
case 0x2000000:{
        if(
  ((fault_register_delayed[cell_number] & 0x2) &&
!(fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 1;

else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x2) &&
 (fault_register_delayed[cell_number] & 0x300))
         )
        north_link[cell_number] = 3;
else    north_link[cell_number] = 4;
   break; } /* end case 2000000 fcrn  */
case 0x3000000:{
  if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
    !(fault_register_delayed[cell_number] & 0x3))
  ||((fault_register_delayed[cell_number] & 0x300)
      && (fault_register_delayed[cell_number] & 0x2))
        )
        north_link[cell_number] = 1;
else    if(
   (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
         )
        north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x2))
         )
        north_link[cell_number] = 3;
  break; } /* end case 3000000 fcr and fcrn */
case 0x40000:
case 0x1040000:{   /* mf or mf and fcr */
        if(
  ((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x3))
  ||(fault_register_delayed[cell_number] & 0x2 &&
!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
 (fault_register_delayed[cell_number] & 0xc0) &&
 (fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0) &&
!(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
```

```
  break; } /* end case 40000 multiple fault mf */
case 0x2040000:
case 0x3040000:{
  if(
  (fault_register_delayed[cell_number] & 0x4)
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x3))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )north_link[cell_number] = 3;
  break; } /* end case 2040000 mf and fcrn */
case 0x80000:
case 0x2080000:{ /* mfn or mfn and fcrn */
  if((fault_register_delayed[cell_number] & 0x7)  = = 0x7)
  north_link[cell_number] = 5;
else
  (!(fault_register_delayed[cell_number] & 0x300)
     && ((fault_register_delayed[cell_number] & 0x6) = = 0x6))
   ||((fault_register_delayed[cell_number] & 0x300)
     && (!(fault_register_delayed[cell_number] & 0x5)))

        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x304))
        )north_link[cell_number] = 4;
else    if(
  (!(fault_register_delayed[cell_number] & 0x2))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0x5) &&
(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
  break; } /* end case 80000  mfn  */
case 0x1080000:
case 0x3080000:{ /* mfn and fcr */
        if(fault_register_delayed[cell_number] = = 0x80007)
    north_link[cell_number] = 5;

else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x6)  = = 0x6)
  )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x4))
  )north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
     && !(fault_register_delayed[cell_number] & 0x2))
        )north_link[cell_number] = 3;
  break; } /* end case 1080000 mfn and fcr */
case 0xc0000:
case 0x10c0000:
case 0x20c0000:
case 0x30c0000:{/* mf and mfn, fcr and fcrn not critical */
        if(
  (!(fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x3))
  ||((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0) &&
     ((fault_register_delayed[cell_number] & 0x6)  = = 0x6))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && !(fault_register_delayed[cell_number] & 0x5))
        )north_link[cell_number] = 1;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300)
```

```
  && (fault_register_delayed[cell_number] & 0x3))
 /*||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x3))  */
  ||((fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x300) &&
     !(fault_register_delayed[cell_number] & 0x4))
       )north_link[cell_number] = 4;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x2))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x5))
       )north_link[cell_number] = 3;
  break; } /* end case c0000 mf and mfn  */
 } /* end switch north link code */

/*  begin south link code  */
switch(fault_register_delayed[cell_number] & 0x5140000)
{ /* south link switch code */
case 0:
  { /* begin case 0 */
   if(
  (!(fault_register_delayed[cell_number] & 0x300)
    && !(fault_register_delayed[cell_number] & 0x18))
  ||((fault_register_delayed[cell_number] & 0x300)
    && (fault_register_delayed[cell_number] & 0x10))
       )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
       )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300) &&
 (fault_register_delayed[cell_number] & 0x18))
       )south_link[cell_number] = 20;
  break; } /* end case 0, single faults, no critical area faults */

case 0x1000000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
     && (!(fault_register_delayed[cell_number] & 0x18)))
  )south_link[cell_number] = 22;

else    if(
  ((fault_register_delayed[cell_number] & 0x300)
    && ((fault_register_delayed[cell_number] & 0x18)))
  ||(!(fault_register_delayed[cell_number] & 0x300))
  )south_link[cell_number] = 20;
else   south_link[cell_number] = 19;
  break; } /* end case 0x1000000, fcr */

case 0x4000000:
  { /* begin case  */

  if(
  (!(fault_register_delayed[cell_number] & 0x300)
      && fault_register_delayed[cell_number] & 0x10)
  )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x10)
    && (fault_register_delayed[cell_number] & 0x300))
       )south_link[cell_number] = 19;
else    south_link[cell_number] = 20;
/* not possible this configuration */
  break; } /* end case fcrs */

case 0x5000000:
  { /* begin case  */
       if(
  (!(fault_register_delayed[cell_number] & 0x300))
```

```
  ||((fault_register_delayed[cell_number] & 0x300)
&& fault_register_delayed[cell_number] & 0x10)
        )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300)
&&(fault_register_delayed[cell_number] & 0x18))
        )south_link[cell_number] = 20;
  break; } /* end case fcr and fcrs      */

case 0x40000:
case 0x1040000:
  { /* begin case  */
  if(
  ((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     !(fault_register_delayed[cell_number] & 0x18))
          )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xd0))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x300))
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0) &&
     (fault_register_delayed[cell_number] & 0x18))
          )south_link[cell_number] = 20;
  break; } /* end case mf */

case 0x4040000:
    { /* begin case  */
        if(
  (fault_register_delayed[cell_number] & 0x20)
  ||((fault_register_delayed[cell_number] & 0x300) &&
(fault_register_delayed[cell_number] & 0xc0))
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0xc0))
        )south_link[cell_number] = 19;
else    if(
  ((fault_register_delayed[cell_number] & 0x18) &&
!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 20;
  break; } /* end case mf and fcrs                    */

case 0x100000:
case 0x4100000:
  { /* begin case  */
  if(
   ((fault_register_delayed[cell_number] & 0x38) = = 0x38)
    ||((fault_register_delayed[cell_number] & 0x8000008) = = 0x8000008)
   )
    south_link[cell_number] = 18;
else
  (((fault_register_delayed[cell_number] & 0x30)= = 0x30) &&
     !(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else    if(
  (!(fault_register_delayed[cell_number] & 0x10))
  ||((fault_register_delayed[cell_number] & 0x28) &&
(fault_register_delayed[cell_number] & 0x10) &&
    (fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x320))
        )south_link[cell_number] = 20;
  break; } /* end case mfs */
```

```
case 0x1100000:
case 0x5100000:
  { /* begin case */
  if(
  ((fault_register_delayed[cell_number] & 0x30) = = 0x30)
  ||(!(fault_register_delayed[cell_number] & 0x300))
        )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300) &&
!(fault_register_delayed[cell_number] & 0x10))
        )south_link[cell_number] = 19;
else    if(
  (!(fault_register_delayed[cell_number] & 0x20))
        )south_link[cell_number] = 20;
  break; } /* end case mfs and fcr */

case 0x140000:
case 0x1140000:
case 0x4140000:
case 0x5140000:
  { /* begin case */
  if(
  ((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0)
  && ((fault_register_delayed[cell_number] & 0x30) = = 0x30))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && !(fault_register_delayed[cell_number] & 0x28))
  ||(!(fault_register_delayed[cell_number] & 0x300)
   && !(fault_register_delayed[cell_number] & 0x18))
  )south_link[cell_number] = 22;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
  && !(fault_register_delayed[cell_number] & 0x10))
  ||(!(fault_register_delayed[cell_number] & 0xc0)
  && (fault_register_delayed[cell_number] & 0x28))
  )south_link[cell_number] = 19;
else    if(
  ((fault_register_delayed[cell_number] & 0x300)
  && (fault_register_delayed[cell_number] & 0xc0)
   && !(fault_register_delayed[cell_number] & 0x20))
   ||(!(fault_register_delayed[cell_number] & 0xc0)
   && (fault_register_delayed[cell_number] & 0x28))
   ||(!(fault_register_delayed[cell_number] & 0x300)
   &&(fault_register_delayed[cell_number] & 0x18))
   )south_link[cell_number] = 20;
  break; } /* end case mf and mfs */


} /* end south link switch code */


} /* end code for row-deformation east or west not */
/************************************************
   CODE BELOW THIS POINT ADDED TO ALGORITHM 1
         TO ENABLE ALGORITHM 3

************************************************/
if(row_deformation_east[cell_number][0]  &&
  !row_deformation_west[cell_number][0]){

if(fault_register_delayed[cell_number] & 0x312)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
        }
else if(fault_register_delayed[cell_number] & 0x28)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 19;
                        }
else if(fault_register_delayed[cell_number] & 0x80)
 {  north_link[cell_number] = 4;
  south_link[cell_number] = 20;
```

```
    }
else if(fault_register_delayed[cell_number] & 0x40)
 { north_link[cell_number] = 4;
  south_link[cell_number] = 20;
  }
else if(fault_register_delayed[cell_number] & 0x5)
 { north_link[cell_number] = 3;
  south_link[cell_number] = 22;
 }
if(fault_register_delayed[cell_number] & 0x100)
  { east_link[cell_number] = 23;
  west_link[cell_number] = 2;  }
if(fault_register_delayed[cell_number] & 0x40)
  { east_link[cell_number] = 21;
  west_link[cell_number] = 0;  }
if(fault_register_delayed[cell_number] = = 0x80003)
 { north_link[cell_number] = 5;
          }

/* end row deformation east */ }

if(row_deformation_west[cell_number][0]  &&
!row_deformation_east[cell_number][0]){

if(fault_register_delayed[cell_number] & 0xe4)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
         }
else if(fault_register_delayed[cell_number] & 0x18)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 20;
         }
else if(fault_register_delayed[cell_number] & 0x3)
 { north_link[cell_number] = 4;
  south_link[cell_number] = 22;
   }
else if(fault_register_delayed[cell_number] & 0x300)
 { north_link[cell_number] = 3;
  south_link[cell_number] = 19;
  }
if(fault_register_delayed[cell_number] & 0x100)
  { east_link[cell_number] = 23;
  west_link[cell_number] = 2;  }
if(fault_register_delayed[cell_number] & 0x40)
  { east_link[cell_number] = 21;
  west_link[cell_number] = 0;  }

 /* end row deformation west */ }
if(row_deformation_east[cell_number][0] &&
 row_deformation_west[cell_number][0] )
  { if(fault_register_delayed[cell_number] & 0x1)
 north_link[cell_number] = 5;
else north_link[cell_number] = 1;
 south_link[cell_number] = 22;
 east_link[cell_number] = 23;
 west_link[cell_number] = 0;
  }
} /*  end code for no deformation bits set   */
else {
if((fault_register_delayed[cell_number] & 0x38000000)&&
!(fault_register_delayed[cell_number] & 0xe00000))
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0; }

if(
  ((fault_register_delayed[cell_number] & 0x7) = = 0x7)
||((fault_register_delayed[cell_number] & 0x20000001) = = 0x20000001)
  )
 { north_link[cell_number] = 5;
  south_link[cell_number] = 22;
```

```
  east_link[cell_number] = 23;
  west_link[cell_number] = 0; }
if(
  ((fault_register_delayed[cell_number] & 0x8e00008) = = 0x8000008)
  )
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0; }


if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
/*if(south_east_is_faulty[cell_number])*/
  east_link[cell_number] = 20;
  west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x200008)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 20;
  west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x400000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
  west_link[cell_number] = 0; }

else if((fault_register_delayed[cell_number] & 0x80e00009 ) = = 0x800000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 3; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x800008)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 23;
  west_link[cell_number] = 3; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
 if(!(row_deformation_northeast_blocker[cell_number]) &&
   !(east_is_faulty[cell_number]))
  east_link[cell_number] = 23;
 else east_link[cell_number] = 20;
  west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0x600008)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
 if((row_deformation_northeast_blocker[cell_number]) &&
   south_east_is_faulty[cell_number])
  east_link[cell_number] = 15;
 else east_link[cell_number] = 23;
   west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
 if(!(row_deformation_north_blocker[cell_number]))
 { east_link[cell_number] = 23;
  west_link[cell_number] = 0;}
 else
 { east_link[cell_number] = 4;
  west_link[cell_number] = 3;}
  }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0xe00008)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0; }

else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0xc00008)
```

```
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
if(row_deformation_north_blocker[cell_number])
  east_link[cell_number] = 4;
else east_link[cell_number] = 23;
if(row_deformation_north_blocker[cell_number])
  west_link[cell_number] = 3;
else west_link[cell_number] = 0;
 }


else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 20;
 if(!(row_deformation_northwest_blocker[cell_number])&&
  !(west_is_faulty[cell_number]))
  west_link[cell_number] = 0;
  else west_link[cell_number] = 19;   }
else if((fault_register_delayed[cell_number] & 0x80e00008) = = 0xa00008)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 18;
  east_link[cell_number] = 20;
 if((row_deformation_northwest_blocker[cell_number]) &&
 south_west_is_faulty[cell_number] )
  west_link[cell_number] = 13;
else west_link[cell_number] = 0;
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
  west_link[cell_number] = 3; }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80200001)
 { north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
 if((row_deformation_north_blocker[cell_number]) &&
  north_west_is_faulty[cell_number])
  west_link[cell_number] = 8;
else  west_link[cell_number] = 3; }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80400000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 20;
  west_link[cell_number] = 0; }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80400001)
 { north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 20;
  west_link[cell_number] = 0; }

else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80800000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00001) = =   0x800001)
 { north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 3;
 if(cell_number = = 34) {
 printf("34 is here \n");getch();}   }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80800001)
 { north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 19; }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600000)
 { north_link[cell_number] = 1;
  south_link[cell_number] = 22;
if(!(row_deformation_north_blocker[cell_number]))
```

```
  east_link[cell_number] = 23;
else  east_link[cell_number] = 4;
  west_link[cell_number] = 3;  }

else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80600001)
 {  north_link[cell_number] = 5;
  south_link[cell_number] = 22;
if(row_deformation_north_blocker[cell_number] &&
 north_east_is_faulty[cell_number])
   east_link[cell_number] = 10;
elseast_link[cell_number] = 23;
if(row_deformation_north_blocker[cell_number] &&
 north_west_is_faulty[cell_number])
  west_link[cell_number] = 8;
elsest_link[cell_number] = 3;  }

else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00000)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0;  }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80e00001)
 {  north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 23;
  west_link[cell_number] = 0;  }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00000)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
  west_link[cell_number] = 0;  }
else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0x80a00001)
 {  north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
  west_link[cell_number] = 0;  }

else if((fault_register_delayed[cell_number] & 0x80e00001) = = 0xc00001)
 {  north_link[cell_number] = 5;
  south_link[cell_number] = 22;
  east_link[cell_number] = 4;
  west_link[cell_number] = 3;  }

else if((fault_register_delayed[cell_number] & 0x80e00009) = = 0xc00000)
 {  north_link[cell_number] = 1;
  south_link[cell_number] = 22;
if(row_deformation_north_blocker[cell_number])
  east_link[cell_number] = 4;
else  east_link[cell_number] = 23;
if(row_deformation_north_blocker[cell_number])
  west_link[cell_number] = 3;
else  west_link[cell_number] = 0;  }

if((fault_register_delayed[cell_number] & 0x204008) = = 0x204008)
  south_link[cell_number] = 14;
if((fault_register_delayed[cell_number] & 0x80200401) = = 0x80200401)
  north_link[cell_number] = 9;
 } /* end deformation code */

} /* end reconfigure */
```

A.13 Faultrg4.c

```
/*****************************************************************
file: faultrg4.c set fault registers, this routine takes the input
from the receive routine and sets the appropriate fault register
bits
```

```
**********************************************************************/
#include < stdio.h >
extern long
input_from_north[100][2],
input_from_south[100][2],
input_from_east[100][2],
input_from_west[100][2],
fault_register[100],
fault_register_delayed[100];
extern
north_link[100],
south_link[100],
east_link[100],
west_link[100],
state[100],
north_west_is_faulty[100],
north_east_is_faulty[100],
west_is_faulty[100],
east_is_faulty[100],
south_west_is_faulty[100],
south_east_is_faulty[100],
fareast_is_faulty[100],
farwest_is_faulty[100],
row_deformation_east[100][2],
row_deformation_west[100][2],
row_deformation_north_blocker[100],
row_deformation_northeast_blocker[100],
row_deformation_northwest_blocker[100];
extern long fault_register1[100];
void set_fault_register(cell_number)

{
int i;
long bit[34];

 fault_register_delayed[cell_number] = fault_register[cell_number];
 row_deformation_east[cell_number][1] = row_deformation_east[cell_number][0];
 row_deformation_west[cell_number][1] = row_deformation_west[cell_number][0];

for(i=0; i< = 33; i++)
bit[i] = 0;

/**********************************************************************
SET AUXILLARY BITS, NE IS FAULTY, SE IS FAULTY, NW IS FAULTY ANS SW IS
FAULTY THESE BITS ARE TRANSMITTED, BUT ARE NOT USED IN DECODING
**********************************************************************/
if(
  ((fault_register[cell_number] & 0x80200001)= = 0x200001)
  ||(input_from_north[cell_number][0] & 0x20000)
  ||(row_deformation_north_blocker[cell_number] &&
 fault_register[cell_number] & 0x1)

)
 row_deformation_north_blocker[cell_number] = 1;
if(
  (((fault_register[cell_number] & 0x80400000)= = 0x400000)&&
 north_east_is_faulty[cell_number])
  ||(input_from_north[cell_number][0] & 0x800000)
  ||(row_deformation_northeast_blocker[cell_number] &&
 north_east_is_faulty[cell_number])
)
 row_deformation_northeast_blocker[cell_number] = 1;
if(
  (((fault_register[cell_number] & 0x80800000)= = 0x800000)&&
north_west_is_faulty[cell_number])
  ||(input_from_north[cell_number][0] & 0x40000)
  ||(row_deformation_northwest_blocker[cell_number] &&
  north_west_is_faulty[cell_number])
  )
  row_deformation_northwest_blocker[cell_number] = 1;

if(
```

```
((input_from_east[cell_number][1] & 0x400000)&&
  !(fault_register[cell_number] & 0x400000)
 &&  (fault_register[cell_number] & 0x312)) /* was 3ff */
  ||((input_from_east[cell_number][0] & 0x1000000)
 &&  (fault_register[cell_number] & 0x3ff))
||((input_from_south[cell_number][0] & 0x1000000) &&
 fault_register[cell_number] & 0x140)
  ||((input_from_south[cell_number][0] & 0x1000000) &&
  south_link[cell_number] = = 18)
 ||((row_deformation_east[cell_number][1] &&
 row_deformation_west[cell_number][1]))
 ){
bit[32] = 1; }
if(

((input_from_west[cell_number][1] & 0x800000) &&
   !(fault_register[cell_number] & 0x800000)
 && (fault_register[cell_number] & 0xe4)) /* was 3ff*/
|| ((input_from_west[cell_number][0] & 0x2000000)&&
  (fault_register_delayed[cell_number] & 0x3ff) )
||((input_from_south[cell_number][0] & 0x2000000) &&
 fault_register[cell_number] & 0x140)

||((input_from_south[cell_number][0] & 0x2000000) &&
 south_link[cell_number] = = 18)
 ||((row_deformation_east[cell_number][1] &&
 row_deformation_west[cell_number][1]))
)
bit[33] = 1;
if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf)
   && (north_link[cell_number] = = 4))
  ||((input_from_east[cell_number][1] & 0x1) &&
  east_link[cell_number] = = 23)
  )
  north_east_is_faulty[cell_number] = 1;

if(
(((input_from_north[cell_number][0] & 0xf) = = 0xf)
  && (north_link[cell_number] = = 3))
  ||((input_from_west[cell_number][1] & 0x1) &&
 west_link[cell_number] = = 0)
  )
  north_west_is_faulty[cell_number] = 1;
if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf)
  && (south_link[cell_number] = = 20))
  ||((input_from_east[cell_number][1] & 0x8) &&
 east_link[cell_number] = = 23)
  )
  south_east_is_faulty[cell_number] = 1;
if(
(((input_from_south[cell_number][0] & 0xf) = = 0xf)
 && (south_link[cell_number] = = 19))
  ||((input_from_west[cell_number][1] & 0x8)&&
 west_link[cell_number] = = 0)
  )
south_west_is_faulty[cell_number] = 1;
if(
(((input_from_east[cell_number][0] & 0xf) = = 0xf)
  &&(east_link[cell_number] = = 21))
  ||((input_from_south[cell_number][0] & 0x40000000) &&
 (south_link[cell_number] = = 20))
  ||((input_from_east[cell_number][0] & 0x80000000)
 && east_link[cell_number] = = 4)
  )
  fareast_is_faulty[cell_number] = 1;

if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf)
  &&(west_link[cell_number] = = 2))
```

```
  )
  farwest_is_faulty[cell_number] = 1;



/*****************************************************************
RECONFIGURATION IS DONE BASED ON THE CONTENTS OF THE delayedED
FAULT REGISTER, BUT INPUT IS USED TO SET THE CONTENTS OF THE
FAULT REGISTER. THIS ALLOWS A CELL TO COMMUNICATE TO ITS
CURRENT NEIGHBORS ALL THE INPUT IT RECEIVED FROM ALL ITS
NEIGHBORS BEFORE DISCONNECTING FROM A CELL
*****************************************************************/

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 1))
  || ((input_from_west[cell_number][0] & 0x40000000)  &&
  (west_link[cell_number] = = 0) )
|| ( (input_from_east[cell_number][0] & 0x10000000)  &&
  (east_link[cell_number] = = 23) )
  )
  {
  fault_register[cell_number] = fault_register[cell_number] | 0x1;
  } /* immediate cell to the north is faulty */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = =  3))
  || ((input_from_north[cell_number][1] & 0x300) &&
  (( north_link[cell_number] = = 1) || (north_link[cell_number] = = 3)))
  || (( input_from_north[cell_number][1] & 0x200) &&
 north_link[cell_number] = = 4)
  ||((input_from_west[cell_number][1] & 0x3)&&
((input_from_west[cell_number][1] & 0xe00000) != 0x800000)&&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][0] & 0x40000000) &&
      (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
  west_link[cell_number] = = 3)
  || north_west_is_faulty[cell_number]
  )
  bit[1] = 1;
  /* fault north and west of this cell */

if(
  (((input_from_north[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x5)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ((input_from_north[cell_number][1] & 0xc0) &&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4))
||((input_from_east[cell_number][1] & 0xc0) &&
   east_link[cell_number] = = 4)

)
  bit[2] = 1;
/* fault north and east of this cell */

if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 22))
 || ((input_from_west[cell_number][0] & 0x80000000) &&
   (west_link[cell_number] = = 0))
 || ((input_from_east[cell_number][0] & 0x20000000) &&
   (east_link[cell_number] = = 23))
  )
  fault_register[cell_number] = fault_register[cell_number] | 0x8;
  /* immediate southern neighbor is faulty */

if(
```

```
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 19))
||((input_from_west[cell_number][1] & 0x18) &&
!((input_from_west[cell_number][1] & 0xe00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
||((input_from_west[cell_number][0] & 0x80000000) &&
  west_link[cell_number] = = 2)
||((input_from_south[cell_number][1] & 0x300) &&
  south_link[cell_number] = = 22 )
||((input_from_west[cell_number][1] & 0x300) &&
 west_link[cell_number] = = 19)
|| ((input_from_west[cell_number][1] & 0x8) &&
   west_link[cell_number] = = 0)
||((input_from_south[cell_number][1] & 0x2) &&
 south_link[cell_number] = = 18)
 )
bit[4] = 1;
  /* fault to the southwest of this cell */

if(
  (((input_from_south[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (south_link[cell_number] = = 20))
||((input_from_east[cell_number][1] & 0x28)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
  (east_link[cell_number] = = 21 || east_link[cell_number] = = 23))
|| ((input_from_south[cell_number][1] & 0xc0)  &&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20)  )
||((input_from_east[cell_number][1] & 0xc0) &&
 east_link[cell_number] = = 20)
||((input_from_south[cell_number][1] & 0x4) &&
 south_link[cell_number] = = 18)

)

bit[5] = 1;
 /* fault to the southeast of this cell */


if(
  (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (east_link[cell_number] = = 23))
  ) {
  east_is_faulty[cell_number] = 1;
  fault_register[cell_number] = fault_register[cell_number] | 0x40;}
  /* immediate eastern neighbor is faulty */


if(
 (((input_from_east[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (east_link[cell_number] = = 21))
  || ((input_from_east[cell_number][1] & 0xc0)&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
  || ((input_from_east[cell_number][1] & 0x20) &&
 east_link[cell_number] = = 4)
  || ((input_from_east[cell_number][1] & 0x4) &&
  east_link[cell_number] = = 20)
  ||fareast_is_faulty[cell_number]
)
  bit[7] = 1;
  /* faulty cell to  the far east in this row */


if(
  (((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 0))
  )
 {west_is_faulty[cell_number] = 1;
  fault_register[cell_number] = fault_register[cell_number] | 0x100;
  /* immediate western neighbor is faulty */
```

```
}
if(
  ((((input_from_west[cell_number][0] & 0xf) = = 0xf) /* cell is faulty */
 && (west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x300) &&
  !((input_from_west[cell_number][1] & 0xe00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  || ((input_from_west[cell_number][1] & 0x10) &&
  west_link[cell_number] = = 3)
  || ((input_from_west[cell_number][1] & 0x2) &&
  west_link[cell_number] = = 19))
/*  &&!(input_from_west[cell_number][1] & 0x20000)*/
  ||farwest_is_faulty[cell_number]
  )
bit[9] = 1;
 /* fault to the far west of this cell */


 if(
((input_from_north[cell_number][1] & 0x1) &&
  (north_link[cell_number] = = 1) )
|((input_from_north[cell_number][0] & 0x10000000) &&
  north_link[cell_number] = = 4)
||((input_from_north[cell_number][0] & 0x40000000) &&
  north_link[cell_number] = = 3)
 ||((input_from_east[cell_number][0] & 0x10000000) &&
 east_link[cell_number] = = 4)
 ||((input_from_west[cell_number][0] & 0x40000000) &&
 east_link[cell_number] = = 3)
 ||((input_from_north[cell_number][1] & 0x8) &&
  north_link[cell_number] = = 9)
  )
  bit[10] = 1;
/* nn-  fault 2 rows to the north and north of this cell*/


if(
((input_from_north[cell_number][1] & 0x2) &&
(north_link[cell_number] = = 1))
||((input_from_north[cell_number][1] & 0x3) &&
 (north_link[cell_number] = = 0x3))
|| ((input_from_west[cell_number][1] & 0xc00)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
)
bit[11] = 1;
/* nnw- fault 2 rows north and west of this cell*/


if(
(input_from_north[cell_number][1] & 0x4)
&& (north_link[cell_number] = = 1)
||((input_from_north[cell_number][1] & 0x5) &&
 (north_link[cell_number] = = 4))
|| ((input_from_east[cell_number][1] & 0x1400)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
  bit[12] = 1;
/* nne- fault 2 rows north and east of this cell*/


if(
(
((input_from_north[cell_number][1] & 0x80000) &&
 ((north_link[cell_number] = = 1) || (north_link[cell_number] = = 4) ) )
|| ((input_from_east[cell_number][1] & 0x2000)&&
  !((input_from_west[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||((input_from_west[cell_number][1] & 0x2000)&&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
```

```
  )
 && fault_register[cell_number] & 0x1c00
)
bit[13] = 1;
 /*mfnne- multiple faults 2 rows north and east of this cell*/


if(
(input_from_south[cell_number][1] & 0x8)  &&
  (south_link[cell_number] = = 22)
|| ((input_from_south[cell_number][0] & 0x20000000) &&
  (south_link[cell_number] = = 20))
|| ((input_from_south[cell_number][0] & 0x80000000) &&
 (south_link[cell_number] = = 19))
||((input_from_south[cell_number][1] & 0x1) &&
 south_link[cell_number] = = 14)
||(((input_from_south[cell_number][0] & 0xf) = = 0xf) &&
 south_link[cell_number] = = 18)
||((input_from_west[cell_number][0] & 0x80000000)
 &&(west_link[cell_number] = = 19))
  )
bit[14] = 1;
/* ss- fault 2 rows to the south and in the same column */


if(
 ((input_from_south[cell_number][1] & 0x10) &&
 (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x18) &&
 (south_link[cell_number] = = 19))
||((input_from_west[cell_number][1] &  0xc000) &&
  !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
){
bit[15] = 1;
 }
/* ssw- single fault 2 rows south and west of this cell*/


if(
((input_from_south[cell_number][1] & 0x20)  &&
  (south_link[cell_number] = = 22))
||((input_from_south[cell_number][1] & 0x28) &&
  (south_link[cell_number] = = 20))
||(((input_from_east[cell_number][1] & 0x14000))&&
  !((input_from_east[cell_number][1] & 0x600000)= = 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
)
bit[16] = 1;
/* sse- single fault 2 rows south and east of this cell*/
else bit[16] = 0;

 if(
 (
 ((input_from_south[cell_number][1] & 0x100000)&&
  ((south_link[cell_number] = = 22) || (south_link[cell_number] = = 20) ))
||((input_from_east[cell_number][1] & 0x20000)&&
 !(fault_register_delayed[cell_number] & 0x800000) &&
(east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||(((input_from_west[cell_number][1] & 0x20000))&&
 !((input_from_west[cell_number][1] & 0xa00000)= = 0x800000) &&
  (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 )&& fault_register[cell_number] & 0x1c000
  )
bit[17] = 1;
 /* mfss- multiple faults 2 rows to the south of cell*/


if(
(
(((fault_register[cell_number] & 0xc0) = = 0xc0)&&
  !(fareast_is_faulty[cell_number]))
```

```
||(((fault_register[cell_number] & 0x300) == 0x300)&&
  !(farwest_is_faulty[cell_number]))
||( ((input_from_east[cell_number][1] & 0x400c0) > = 0x40000)&&/*added = for test*/
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
||( (input_from_west[cell_number][1] & 0x40000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
) && fault_register[cell_number] & 0x3c0
)
  bit[18] = 1;
  /* mf multiple faults in this row */


if(
  (
  ((fault_register[cell_number] & 0x5) = = 0x5)
  ||( input_from_east[cell_number][1] & 0x80000)&&
  !(fault_register_delayed[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
 ||(((input_from_north[cell_number][1] & 0x40000)&&
 (input_from_north[cell_number][1] & 0x3c0))&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
   north_link[cell_number] = = 3))
 ||  ((input_from_west[cell_number][1] & 0x80000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
 || ((input_from_south[cell_number][1] & 0x2000) &&
 south_link[cell_number] != 18)
 ) && fault_register[cell_number] & 0x7
  )
  bit[19] = 1;
 /* mfn multiple fault in the row to the north */


if(
(
((fault_register[cell_number] & 0x28) = = 0x28)
|| ( (input_from_east[cell_number][1] & 0x100000)&&
  !((input_from_east[cell_number][1] & 0x600000) = = 0x400000) &&
(east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ( (input_from_south[cell_number][1] & 0x40000)&&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20 ||
   south_link[cell_number] = = 19) )
|| ((input_from_west[cell_number][1] & 0x100000)&&
  !((input_from_west[cell_number][1] & 0xa00000) = = 0x800000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
)&& fault_register[cell_number] & 0x38
)
bit[20] = 1;
/* mfs multiple fault in the row to the south */

if(
  (((fault_register[cell_number] & 0x100010) = = 0x100000)&&
  (fault_register[cell_number] & 0x300)) /* mfs & !sw & w + fw */
  || (((fault_register[cell_number] & 0x80002) = =0x80000)  &&
  (fault_register[cell_number] & 0x300)) /*  mfn & !nw ww + fw not*/
  ||((input_from_west[cell_number][1] & 0x1000000)&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  ||( input_from_east[cell_number][1] & 0x1000000)&&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
  || (((fault_register[cell_number] & 0x80044) = = 0x80044)&&
 (!(fault_register[cell_number] & 0x3)))
  ||((input_from_north[cell_number][1] & 0x4000000)&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
   north_link[cell_number] = = 3))
  || (((fault_register[cell_number] & 0x2000040) = = 0x2000040) &&
 !(fault_register[cell_number] & 0x3))
  ||((input_from_south[cell_number][1] & 0x2000000)&&
 ( south_link[cell_number] = = 22 || south_link[cell_number] = = 20 ||
   south_link[cell_number] = = 19) )
  ||(((fault_register_delayed[cell_number] & 0x4000010) = = 0x4000000) &&
```

```
    (fault_register_delayed[cell_number] & 0x300))
  ){
 /* fcr fault in the critical region of this row */
  bit[24] = 1;
 }
if(

  (((fault_register[cell_number] & 0x40300) = =0x40000) /* mfe */
&&(fault_register[cell_number] & 0x3))
|| ((input_from_east[cell_number][1] & 0x2000000)&&
  !(fault_register_delayed[cell_number] & 0x800000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
|| ((input_from_west[cell_number][1] & 0x2000000)&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
|| ((input_from_north[cell_number][1] & 0x1000000)&&
  ( north_link[cell_number] = = 1 || north_link[cell_number] = = 4 ||
    north_link[cell_number] = = 3))
)    /* fcrn fault in the critical region of the row to the north*/
  bit[25] = 1;

if(
  (((fault_register[cell_number] & 0x40300) = =0x40000)
  && (fault_register[cell_number] & 0x18)) /* mfe and not w or fw */
  ||((input_from_west[cell_number][1] & 0x4000000)&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (west_link[cell_number] = = 0 || west_link[cell_number] = = 2))
  ||(((input_from_east[cell_number][1] & 0x4000000)
  ||((input_from_east[cell_number][1] & 0x190) = = 0x190))&&
  !(fault_register_delayed[cell_number] & 0x400000) &&
 (east_link[cell_number] = = 23 || east_link[cell_number] = = 21))
  ||((input_from_south[cell_number][1] & 0x1000000) &&
  (south_link[cell_number] = = 22))
  ||(((fault_register[cell_number] & 0x1080040) = =0x1080040) &&
  (( fault_register[cell_number] & 0x18) ||
  (south_east_is_faulty[cell_number]) )&&
  !(fault_register[cell_number] & 0x300) )

/*){crs fault in the critical region of the row to the south*/
bit[26] = 1;
}

/*************************************************************************

CODE TO FORCE ROW DEFORMATION.  ROW DEFORMATION TAKES PLACE WITH ALL
INTERIOR FAULT OCCURENCES, THAT IS ALL BUT THE FARTHEST EAST AND WEST
FAULT IN MULTIPLE FAULT OCCURENCES.  ROW DEFORMATION TO THE NORTH
IS ATTEMPTED FIRST, AND IF IT DOES NOT ENCOUNTER A DOUBLE FAULT EITHER
TO THE EAST OR WEST OF THE COLUMN AFFECTED, AND IT DOES NOT ENCOUNTER
ANOTHER FAULT IN THE SAME COLUMN, IT PROCEEDES UNTIL A SPARE CELL IN THE
TOP ROW IS USED.  IF EITHER OF THE ABOVE CASES EXISTS, THE ROW
DEFORMATION TO THE NORTH IS CANCELLED AND DEFORMATION TO THE SOUTH
IS ATTEMPTED.  IF THIS IS UNSUCCESSFUL, AN IRRECOVERABLE CONDITION EXISTS
AND THE GLOBAL PATTERN REGROWTH ALGORITHM MUST BE CALLED
*************************************************************************/
if(((
  ((fault_register[cell_number] & 0x100038) = = 0x100038)/* center of three faults */
  ||(
(fault_register[cell_number] & 0x8) &&       /* adjacent faults*/
  (
  (input_from_west[cell_number][1] & 0x8) ||
 (input_from_east[cell_number][1] & 0x8)||
 (south_east_is_faulty[cell_number])||
 south_west_is_faulty[cell_number])))
&&   /* adjacent faulty cells */
  !(row_deformation_north_blocker[cell_number]))
  ||(
  (
  ((fault_register[cell_number] & 0x80007) = = 0x80007)/* center of three faults */
  ||(
  (fault_register[cell_number] & 0x1) &&       /* adjacent faults*/
   (
```

```
   (input_from_west[cell_number][1] & 0x1) ||
    (input_from_east[cell_number][1] & 0x1)||
    (north_east_is_faulty[cell_number])||
     north_west_is_faulty[cell_number] ) ))
 &&  row_deformation_north_blocker[cell_number])


  || ((input_from_south[cell_number][1] & 0x200000)&&
  (south_link[cell_number] = = 22) &&
  !row_deformation_north_blocker[cell_number])

 &&((input_from_north[cell_number][1] & 0x80200000) = = 0x80200000)
  (north_link[cell_number] = = 1))
  ||((input_from_south[cell_number][1] & 0x400000) &&
 south_link[cell_number] = = 19)
  ||((input_from_south[cell_number][1] & 0x800000) &&
 south_link[cell_number] = = 20)
  ||(((fault_register[cell_number] & 0x18) = = 0x18) &&
  row_deformation_east[cell_number][1]&&
  !(row_deformation_north_blocker[cell_number]))

  ||((fault_register[cell_number] & 0x8) &&
  row_deformation_east[cell_number][1] &&
  row_deformation_west[cell_number][1])
||((fault_register[cell_number] & 0x80000401) = = 0x80000401)
||(((fault_register[cell_number] & 0x200008) = = 0x200008) &&
  !(row_deformation_north_blocker[cell_number]))
||((fault_register[cell_number] & 0x1) &&
 (north_east_is_faulty[cell_number] ||
 north_west_is_faulty[cell_number]) &&
 row_deformation_north_blocker[cell_number])
)
bit[21] = 1;     /* row  deform north */


if(
(((fault_register[cell_number] & 0xc0) = = 0xc0)&&/* e & fe */
 (fault_register[cell_number] & 0x300))              /* & (w+fw) */
||(((fault_register[cell_number] & 0xc0) = = 0xc0) &&
  (fareast_is_faulty[cell_number]))
||(((input_from_south[cell_number][1] & 0x80400000)= = 0x400000) &&
  (south_link[cell_number] = = 22) &&
  !row_deformation_northeast_blocker[cell_number])
 &&((input_from_north[cell_number][1] & 0x80400000) = = 0x80400000)
  (north_link[cell_number] = = 1))
  ||(((fault_register[cell_number] & 0x80200000) = = 0x80200000) &&
  north_east_is_faulty[cell_number])
  ||((fault_register[cell_number] & 0x8) &&
  south_east_is_faulty[cell_number])
  ||((input_from_east[cell_number][1] & 0x200000) &&
 east_link[cell_number] = = 23)
  ||((fault_register[cell_number] & 0x300) &&
 (fault_register[cell_number] & 0x40) &&
 row_deformation_east[cell_number][0] )
||((fault_register[cell_number] & 0x40) &&
 row_deformation_east[cell_number][0] &&
 row_deformation_west[cell_number][0] )
||(((fault_register[cell_number] & 0x400008) = = 0x400008) &&
  !(row_deformation_northeast_blocker[cell_number]))
|| ((fault_register[cell_number] & 0x1) &&
 (north_east_is_faulty[cell_number]) &&
 (input_from_north[cell_number][0] & 0x80000))
||((fault_register[cell_number] & 0x40) &&
 fareast_is_faulty[cell_number] &&
 row_deformation_northeast_blocker[cell_number])
||((fault_register[cell_number] & 0x1) &&
 row_deformation_northeast_blocker[cell_number])
||((fault_register[cell_number] & 0x400000) &&
   east_is_faulty[cell_number] &&
 (input_from_east[cell_number][1] & 0x200000))
  )
  bit[22] = 1;  /* row deformation north or south east */
```

```
if(
  (((fault_register[cell_number] & 0x40300) == 0x40300)&& /* w & fw */
 (fault_register[cell_number] & 0xc0))       /* and e + fe */
  || ((input_from_south[cell_number][1] & 0x800000)&&
 (south_link[cell_number] == 22) &&
 !(row_deformation_northwest_blocker[cell_number]))
 &&((input_from_north[cell_number][1] & 0x80800000) == 0x80800000)
  (north_link[cell_number] == 1))
  ||((input_from_west[cell_number][1] & 0x200000)/*&&
  (fault_register[cell_number] & 0x50)*/)  /* was 10 */
  ||(south_west_is_faulty[cell_number] &&
  fault_register[cell_number] & 0x200000)
  ||(north_west_is_faulty[cell_number] &&
/*!(row_deformation_north_blocker[cell_number]) &&
  ((fault_register[cell_number] & 0x80200000)==0x80200000))
||((input_from_north[cell_number][1] & 0x800000) &&
 (fault_register[cell_number] & 0x10000000)  &&
 (north_link[cell_number] == 1))
  ||(((((fault_register[cell_number] & 0x100) ) &&
 farwest_is_faulty[cell_number]) ||
  ((( fault_register[cell_number] & 0x80001) == 0x80001) &&
 north_west_is_faulty[cell_number])
  && row_deformation_northwest_blocker[cell_number])
  ||(((fault_register[cell_number] & 0x40300) == 0x40300) &&
  (row_deformation_east[cell_number][1]))
  ||((fault_register[cell_number] & 0x100) &&
  row_deformation_east[cell_number][1] &&
  row_deformation_west[cell_number][1])
||(((fault_register[cell_number] & 0x800008) == 0x800008) &&
  !(row_deformation_northwest_blocker[cell_number]))
||((fault_register[cell_number] & 0x1) &&
 north_west_is_faulty[cell_number] &&
 (input_from_north[cell_number][0] & 0x40000))

||((fault_register[cell_number] & 0x100) &&
   farwest_is_faulty[cell_number] &&
 row_deformation_northwest_blocker[cell_number])
||((fault_register[cell_number] & 0x1) &&
 row_deformation_northwest_blocker[cell_number])
||((fault_register[cell_number] & 0x800000) &&
   west_is_faulty[cell_number] &&
 (input_from_west[cell_number][1] & 0x200000))
  )
  bit[23] = 1;  /* row deformation north or southwest */

  if(
 ((fault_register[cell_number] & 0x8) &&
 (
 (input_from_west[cell_number][1] & 0x8) ||
 (input_from_east[cell_number][1] & 0x8) ||
 fault_register[cell_number] & 0xc00000)
 )
||((input_from_east[cell_number][1] & 0x8000000) &&
 east_link[cell_number] > = 21)
||((input_from_west[cell_number][1] & 0x8000000) &&
 west_link[cell_number] < = 2)
 )
  bit[27] = 1; /* row deformation generated row [i+1] */
/*  this bit used to cancel the effects of adjacent faults in the
 row to the south */
if(
  ((fault_register[cell_number] & 0x100) &&
((input_from_west[cell_number][1] & 0xf) == 0xf) &&
  (west_link[cell_number] == 2))
||((fault_register[cell_number] & 0x40) &&
((input_from_east[cell_number][1] & 0xf) == 0xf) &&
  (east_link[cell_number] == 21))
||(input_from_south[cell_number][1] & 0x20000000)
)
bit[28] = 1;
/* above bit cancels effect of adjacent faults in this row */
```

```
if(
 ((fault_register[cell_number] & 0x1) &&
 (((input_from_west[cell_number][1] & 0x1) && west_link[cell_number] = = 0)
 ||((input_from_east[cell_number][1] & 0x1) && east_link[cell_number] = = 23)
 ||(north_west_is_faulty[cell_number])))
||((input_from_east[cell_number][1] & 0x20000000) &&
 east_link[cell_number] > = 21)
||((input_from_west[cell_number][1] & 0x20000000) &&
 west_link[cell_number] < = 2)
||((fault_register[cell_number] & 0x80200401) = = 0x80200401)
||((fault_register[cell_number] & 0x800001) = = 0x800001)
)
bit[29] = 1;
/* this bit used to cancel the effect of adjacent faults in northern row*/

if(
(((fault_register[cell_number] & 0x80007) = = 0x80007) &&
 row_deformation_north_blocker[cell_number])
||
(((fault_register[cell_number] & 0x402c0) = = 0x402c0) &&
  row_deformation_northeast_blocker[cell_number])
||
(((fault_register[cell_number] & 0x40380) = = 0x40380) &&
  row_deformation_northwest_blocker[cell_number])
||
(input_from_north[cell_number][1] & 0x80000000)
|| ((((fault_register[cell_number] & 0x40300) = = 0x40300) ||
   fault_register[cell_number] & 0x1  &&
   north_west_is_faulty[cell_number]) &&
 row_deformation_northwest_blocker[cell_number])
|| ((((fault_register[cell_number] & 0x400c0) = = 0x400c0) ||
 fault_register[cell_number] & 0x1 &&
 north_east_is_faulty[cell_number] ) &&
 row_deformation_northeast_blocker[cell_number])
|| (((fault_register[cell_number] & 0x20080001) = = 0x20080001)&&
 row_deformation_north_blocker[cell_number])
  ||((input_from_north[cell_number][1] & 0x200000) &&
 north_link[cell_number] = = 9)
  ||((fault_register[cell_number] & 0x100) &&
  row_deformation_northwest_blocker[cell_number])
  )
bit[31] = 1; /* this bit forces deformation bits to deform to the south*/

/**************************************************************************
The remaining code clears all bits in the fault register execpt those
set by direct testing of neighbors at all times except during
reconfiguration ( indicated by a "0" input on one of the inputs.  Since
border cells never have all inputs, this check is disabled for cell
numbers on the top row because these cells would normally have
an external input here.  Code also clears the bits used to generate
north deformation whenever a faulty cell is encountered in the
deformation path
**************************************************************************/
fault_register[cell_number] = fault_register[cell_number] & 0xff1fffff;
/* clear fault register 1 and then reset it */

fault_register[cell_number] = fault_register[cell_number] & 0x00149; /* e00149 */
row_deformation_east[cell_number][0] = 0;
row_deformation_west[cell_number][0] = 0;
 if(
((input_from_south[cell_number][1] & 0x8) )
&& (input_from_south[cell_number][0] & 0x4000000)
)
{
bit[21] = 0; bit[22] = 0;
}
 if(
((input_from_south[cell_number][1] & 0x8) )
&& (input_from_south[cell_number][0] & 0x8000000)
)
{
```

```
bit[21] = 0; bit[23] = 0;
}

 for(i = 0; i< = 31; i+ +)
  fault_register[cell_number] = fault_register[cell_number] |
        (bit[i] < < i);
  row_deformation_east[cell_number][0] =
        row_deformation_east[cell_number][0] | bit[32];
  row_deformation_west[cell_number][0] =
  row_deformation_west[cell_number][0] | bit[33];

if((fault_register[cell_number] & 0x200009) = = 0x200000)
 fault_register[cell_number] = fault_register[cell_number] & 0xfffffc00;

if(((fault_register[cell_number -10] & 0x80400040) = = 0x400040) &&
(fault_register[cell_number] & 0x40) )
 fault_register[cell_number] = fault_register[cell_number]
          | 0x80400000;
if((north_west_is_faulty[cell_number] &&
(fault_register[cell_number] & 0x101) = = 0x101))
 fault_register[cell_number] = fault_register[cell_number]
          | 0x80800000; /* force south deform*/
if(
  ((fault_register[cell_number] & 0xe00000)= = 0xa00000) &&
south_east_is_faulty[cell_number] &&
((input_from_south[cell_number][1] & 0xa00000) = = 0xa00000)
)
fault_register[cell_number] = fault_register[cell_number] | 0x400000;

} /* end set registers routine */
```

VITA

The author was born in West Virginia in 1940. He received a B. S. Degree in Electrical Engineering from VPI in 1967. He was then employed in various positions by Control Data Corporation for 18 years before returning to school. He received his M. S. degree in Electrical Engineering at VPI&SU in 1985.