AUTOMATIC TEST, CONFIGURATION, AND REPAIR OF CELLULAR ARRAYS

by

Frank Blase Manning

S.B., S.M., MASSACHUSETTS INSTITUTE OF TECHNOLOGY
(1972)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1975

Signature of Author.....
Department of Electrical Engineering, May 22, 1975

Certified by.....
Thesis Supervisor

Accepted by.....
Chairman, Departmental Committee on Graduate Students

AUTOMATIC TEST, CONFIGURATION, AND REPAIR OF CELLULAR ARRAYS

by
Frank Blase Manning

Submitted to the Department of Electrical Engineering on May 22, 1975, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

A cellular array is an iterative array of identical information processing machines, cells. The arrays discussed are rectangular arrays of programmable logic, in which information stored in a working cell tells the cell how to behave. No signal line connects more than a few cells. A loading mechanism in each cell allows a computer directly connected to one cell to load any good cell that is not walled off by flawed cells. A loading arm is grown by programming cells to form a path that carries loading information. Cell mechanisms allow a computer to monitor the growth of a loading arm, and to change the arm's route to avoid faulty cells. Properly programmed cells carry test signals between a tested cell and a testing computer directly connected to only a few cells. The computer may discover the faulty cells in an array; and repair the array by loading the array's good cells to embed a desired machine.

Terminology and network models are developed to describe the characteristics of a machine that are important to the test and repair of an array embedding that machine. Important machine classes are defined, and their test and repair requirements are compared. Computer simulations of repair aid this comparison.

Each machine class is represented by a particular cellular machine design. Arrays are presented for realizing highly-integrated, computer-maintained memories, such as variable-length shift-registers, random-access memories, and track-addressed sequential-access memories. One flawed array of simple cells may perform like any digital machine, within limits set by the size of the array, its number of input-output leads, and the speed of its components. One such machine can test, configure, and repair its cellular environment. Applications for these cellular arrays are discussed.

The thesis' approach is oriented toward the realities and trends in large-scale integrated circuit production; and has potential integration level, reliability, maintainability, and flexibility advantages.

THESIS SUPERVISOR: Edward Fredkin
TITLE: Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

Professor Edward Fredkin helped me throughout my graduate schooling; he suggested this thesis topic and guided my work. Professor Peter Elias was my chief advisor in the writing of this thesis document. Dr. E. Roger Banks and I engaged in productive discussions about cellular machines throughout the research. Roger and Professor Jack Dennis read the thesis and provided useful suggestions.

Many people at MIT and in industry helped relate my work to integrated circuit considerations. MIT's John Moussouris and IBM's Dr. Rick Dill were particularly helpful in this regard.

Bruce Kramer and Mary Jo Keller suggested ways to develop my previously untutored drawing techniques.

My wife, Lynn Nina, and my parents, Thomas Patrick and Mary Anne Boschert Manning, helped and encouraged me in many ways.

Many people and facilities at Project MAC assisted this work.

This research was supported in part by a National Science Foundation Fellowship; and in part by the Advanced Research Projects Agency of the Department of Defense under ARPA No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

TABLE OF CONTENTS
(1st of 2 pages)

TABLE OF CONTENTS
(2nd of 2 pages)

LIST OF TABLES

LIST OF ILLUSTRATIONS
(1st of 3 pages)

LIST OF ILLUSTRATIONS
(2nd of 3 pages)

LIST OF ILLUSTRATIONS
(3d of 3 pages)

CHAPTER 1: OVERVIEW

Section 1.0: Introduction

A cellular array is an iterative array of identical information processing machines, cells. Test of an array discovers its flawed cells. Configuration of an array programs it to behave like some machine. Repair of an array programs it to behave like a desired machine in spite of faulty array cells. This thesis develops a practical systems approach to highly integrated, computer-maintained cellular machines. The structural simplicity of cellular machines gives them many advantages, especially now when large-scale integrated circuits (LSI) are proliferating. We specify cell mechanisms and outline associated support programs for an arbitrarily large, two-dimensional, rectangular array. While we focus on two-dimensional rectangular arrays, our approach has obvious extensions to arrays with different interconnection geometries and more dimensions. This approach allows a digital machine to electronically test, configure, and repair an array by direct communication with only a few cells in the array. The fact that a computer can test and repair an array implies that the array need not be perfect. All the cells of the array may be simultaneously produced as a very large, integrated array device. Such a device usually has faulty cells. After the array is first fabricated, a computer can find the defective cells in the array and load a perfect machine, which incorporates only good cells, into the flawed array. Thus the same mass-produced device may be program-customized by a mass-produced device,

the computer, to behave like a desired machine. If this array-embedded machine develops a new flaw during its operation, and if this flaw causes a noticed performance degradation, the array may be partly or completely re-tested and re-customized by a computer. Thus an array-embedded machine may be electronically tested, and repaired to incorporate only good cells. This means that the array may be maintained by a digital machine. Furthermore, the array may be re-customized at any time. This approach is tailored to the realities and trends in design, manufacture, distribution, and maintenance of digital systems, particularly those composed of LSI components.

We discuss arrays of "programmable logic", where information loaded into memory elements in a working cell tells the cell how to behave. No signal line connects many cells. A loading mechanism is developed for each cell in an array; this allows a computer directly connected to only one cell to load any good cell that is not walled off by flawed cells. The loading information which the computer sends to the array may select one of a large set of possible paths for a loading arm that carries loading information to a cell. We develop cell mechanisms that allow the computer to monitor the growth of a loading arm to a cell, and to change the route of the arm to avoid faulty cells. A method is described for testing cells in an array by using a test machine directly connected to only a few cells in the array. Properly programmed cells carry test signals between some newly tested cell and the test machine. A loading arm may be used to vary the state of the tested cell.

Programming an array to behave like a given machine is called embedding that machine. When a machine is embedded in an array, it should not allow faulty cells to affect its behavior. Therefore an embedded machine is programmed to ignore signals sent from faulty cells. We find that the communication paths required between the essential cells of an embedded machine affect test and repair of an array for embedding that machine. Development of terminology and network models allows us to describe embedded machines more precisely. Important embedded machine classes are defined, and their associated test and repair requirements are detailed. Computer simulations of repair facilitate this comparison.

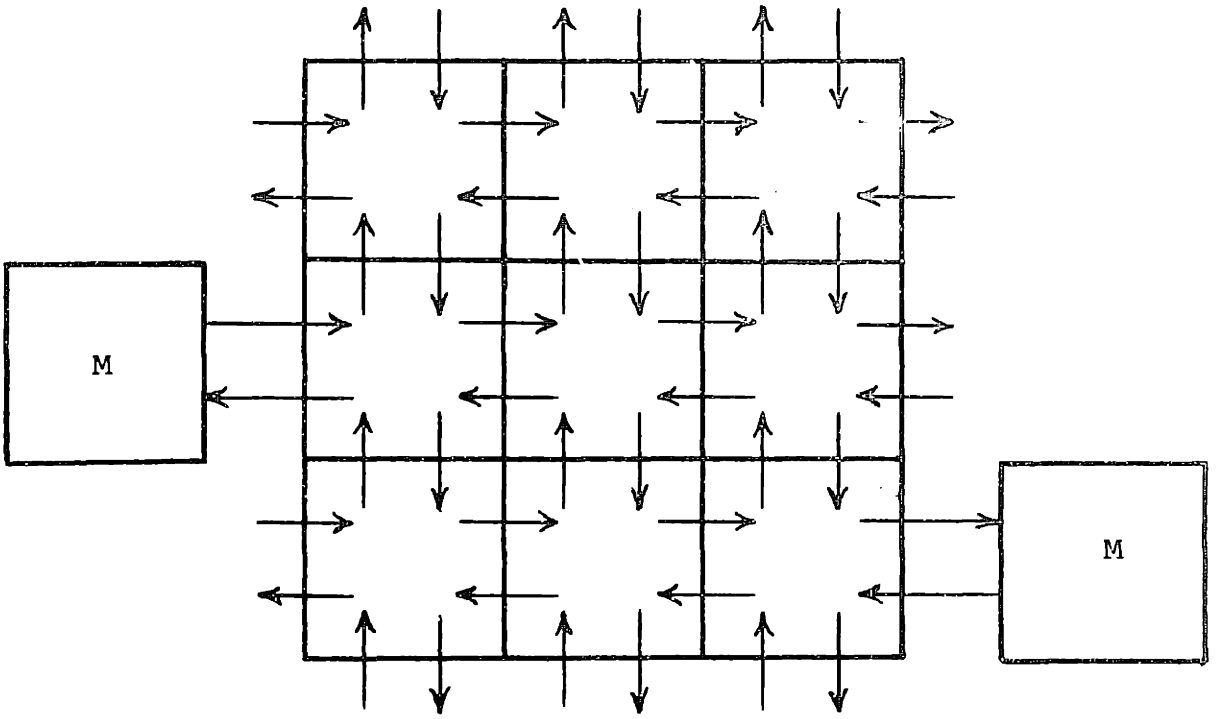
For each class of machine that's described, a particular, potentially useful representative of that class is detailed. All arrays' cells contain our loading mechanism. Arrays are presented for realizing highly integrated, computer-maintained memories. These include arrays for realizing variable-length shift-registers, random-access memories, and track-addressed sequential-access memories. One array of simple cells may be programmed to embed an arbitrary digital machine, within limits set by the size of the array, its number of input-output leads, and the speed of its components. An array-embedded computer can test, configure, and repair its cellular environment using techniques we develop. Indeed, two or more array-embedded computers can test and maintain each other.

Section 1.1: Arrays And Embedded Machines

A more detailed description of our approach requires introduction of some key terms.

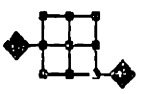
A *cellular array* is an array of functionally identical information processing machines, *cells*, interconnected in an iterative way. Each cell of an n-dimensional array occupies a lattice point in an n-dimensional space. Each cell communicates directly with other machines through a finite set of signal lines. Figure 1.1 shows a possible layout of a cellular array. Each cell in a given array has a fixed number of signal line *side-sets*, each corresponding to potential direct communication with another cell, a *neighbor*. If any member of a side-set connects to a neighbor, all members of the side-set connect to that neighbor. If a side-set doesn't connect to a neighbor, some or all of its members may connect to an extra-array machine. Unconnected inputs act as if they are connected to a binary 0; this is easily implemented. We concentrate on *checkerboard* cellular arrays, two-dimensional arrays like that shown in figure 1.1, where each cell has four side-sets, with input and output signal lines at each side-set. We use the term "checkerboard" to suggest an obvious layout for our arrays, with each checkerboard square standing for an identical cell. Some have proposed arrays in which signal busses run through many cells. Since an array is often catastrophically damaged when a signal bus is damaged, we require that there be no signal busses in checkerboard arrays; at most, a signal line connects a cell to its four neighbors. Checkerboard arrays are well-suited to the step-and-repeat nature of current integrated circuit (IC)

Fig. 1.1 Layout Of A Checkerboard Array Connected To Two Extra-array Machines



Key: An arrow indicates one or more of a machine's inputs or outputs, with the arrow's direction showing the direction of information flow. The "M boxes" represent extra-array Machines. The small, unlettered boxes represent cells.

Fig. 1.2 Interconnection Network For Checkerboard Array Of Figure 1.1



production. These arrays also offer testing, configuration, and repair advantages.

An *interconnection network*, such as that of figure 1.2, partially describes an array's layout by showing how each cell directly communicates with its cellular neighbors or extra-array machines. In an interconnection network, each node represents a cell, and each diamond represents an extra-array machine. A node is linked to another node or diamond if and only if one or more signal lines directly link the associated machines. A node has *degree n* if n links connect to the node.

It's obvious that a sub-array of an array is also an array. Consequently, it's valid to isolate the activity in a sub-array as array activity, and treat the cells outside this array as extra-(sub)array machines.

We focus on *programmable logic* checkerboard arrays, where each cell contains *function-specification state bits* affecting which of several operations the cell performs. Both cells and arrays are viewed as having two functional layers - a *loading layer* and a *processing layer* - with distinct inputs, outputs, and memory elements for each layer. Of course, these layers may be physically intertwined. At any given time instant, only one of a cell's layers is activated. The processing layer of an array is used to provide the functions of the array that are immediately useful to an array's user. The processing layer's output and state are a function of the processing layer's input and state, and of the *function state* - the state of the function-specification state bits. The function-specification state bits may enter a particular function state when an array is powered on; after this, they may only be loaded through use of the array's loading inputs. The sole function of the

loading layer is to load these bits, and thereby affect the function performed in the processing layer of the array. Thus the function-specification bits act as intermediaries between the loading layer and the processing layer. The function-specification bits are the only cell logic elements that are not in either layer. Typical use of an array involves loading the function-specification state bits, thereby specifying some function state affecting processing outputs. Then the loader is quiescent while the processing layer performs its function. Re-use of the loader may re-program the processing layer to provide some new function.

For many actual array applications, a user expects certain variables of an array to remain fixed during a given time interval. For instance, an application might dictate that an array of programmable logic, once loaded, keep the same processing input-output leads and function state. The user of this array would justifiably think of his array as an environment for a machine embedded in the processing layer, with the fixed attributes of the array specifying the embedded machine. Similarly, a user might only use an array's loading inputs during an interval devoted to loading function-specification state bits. The user would then think of the array as an environment for a machine embedded in the loading layer during this interval. Finally, a user might intertwine the processes of loading, using loading inputs, and testing, using processing inputs and outputs, during an interval. The user could think of an embedded machine as occupying both loading and processing layers during the interval. As we might expect, the nature of the embedded machine profoundly affects the testability and repairability of an array.

Knowledge of the constraints on an array during an interval affects testing and repair so profoundly that we develop a language to describe these constraints. We consider definitions relative to use of an array during a given time interval. The user identifies which side-set lines and memory elements directly interest him during an interval; these are the interesting elements. The input-output leads of an array that connected to a user's machine might be the interesting lines for that user. Similarly, function-specification bits which would affect the function an array performed for a user might be interesting, while function-specification bits in a remote section of an array might be uninteresting. Thus interest is defined in terms of a user's intended application. The state of the array at the beginning of the interval, and the signals it may receive during the interval, affect which of its memory elements and side-set lines are *relevant*; that is, which may affect interesting elements during the interval. An *embedded machine* for a given array is described by a list of those relevant memory elements and side-set input lines whose values are known to be fixed during the interval, and their associated values. The embedded machine's input-output lines are the relevant input-output lines of the array that are variable during the interval. A programmable logic loading mechanism may set function-specification state bits, and thereby partially or completely specify the machine embedded in the processing layer of an array. Figure 1.3 gives a characterization of one such embedded machine. Irrelevant inputs and outputs are not shown, since they don't affect the performance of the embedded machine.

Fig. 1.3 Machine Embedded In A Processing Layer

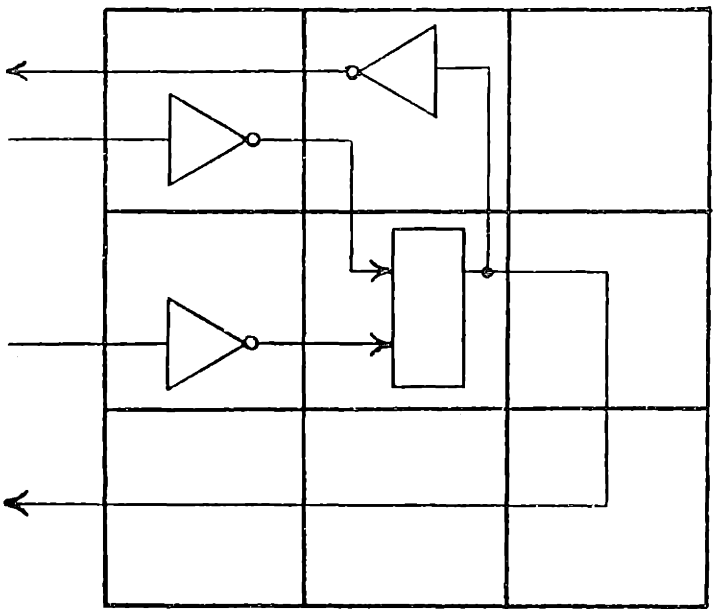
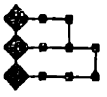


Fig. 1.4 Relcon Network For The Embedded Machine Of Figure 1.3



Different embedded machines may be equivalent. If a flawed array is configured to embed a machine that is equivalent to a perfect array's embedded machine, we say the flawed array has been *repaired* to embed a perfect machine. Our array repair is therefore an information process setting the states of cells, and not a mechanical alteration of an array.

A *relevant connection network*, or *relcon network*, is a subnetwork of the interconnection network that describes communication in an embedded machine (see figure 1.4). As in the interconnection network, dots correspond to cells, and diamonds correspond to extra-array machines. If and only if at least one relevant connection directly links a cell with another cell or extra-array machine, a link connects the cell's dot to the appropriate dot or diamond in the relcon network. A cell's *relcon neighbors* are the entities - cells or extra-array machines - whose representatives are directly linked to the cell's dot in the relcon network. A cell with n relcon neighbors is a cell of *relcon degree n*, called a *relcon-n* cell. An embedded machine's relcon is the highest relcon degree of any of its cells.

A qualification of our definition of an embedded machine makes it more consistent with our intuitive understanding of a machine. We require that two cells in the same embedded machine be connected by some path of relcon neighbors; that is, an embedded machine's relcon network must have some path between their representative dots. Thus two or more machines may be embedded in the same array.

An array constrains the relcon network of machines embedded in that

array. Two cells can be relcon neighbors only if they are linked in the array's interconnection network. Furthermore, the set of allowed cell states may further constrain relcon networks. Although all the arrays that we present have a checkerboard interconnection network, the maximum relcon degree of their associated processing-layer-embedded machines varies. An array is only interesting if this maximum relcon degree is at least 2; otherwise cells in an embedded machine don't communicate with each other.

An array with flawed cells further constrains the relcon network of its embedded machines. In repairing an array, we embed in a flawed array a machine equivalent to one specified for a perfect array. The equivalent machine cannot incorporate flawed cells. We typically use the simplest, harshest model for the flawed cells in an array: these cells are so bad that no embedded machine should have a relevant side-set connected to one of them.

Section 1.2: The Loader And Related Concepts

Transmisston states are fundamental to many of our testing, configuration, and repair operations. In a transmission state, some inputs to a cell are transmitted unchanged as outputs after a delay; that is, the cell acts like one or more wires connecting an input to an output. The cells we discuss all have inputs I and outputs O that can be described by the following indexed sets.

$$\{I_{U1},..I_{UN},I_{R1},..I_{RN},I_{D1},..I_{DN},I_{L1},..I_{LN}\}$$

$$\{O_{U1},..O_{UN},O_{R1},..O_{RN},O_{D1},..O_{DN},O_{L1},..O_{LN}\}$$

The first subscript - U, R, D, or L - denotes one of a cell's four side-sets - Up, Right, Down, or Left. Each side-set of a given cell has the same number of loading inputs and outputs, M, and the same number of processing inputs and outputs, N-M. For all $1 \le K \le N$, it's true that I_{UK}, I_{RK}, I_{DK}, I_{LK}, O_{UK}, O_{RK}, O_{DK}, and O_{LK} are associated, and given the same name. Thus we might speak of the Select loading input and output of each of a cell's side-sets. In a *loading transmisston state*, each loading line of one side-set is transmitted to an associated loading output at one other side-set after a delay not longer than about one gate-delay. Thus a loading transmission state busses the loading inputs of one side-set to associated loading outputs at another side-set. *Processing transmisston states* effectively connect a bus to every side-set's processing outputs. Each bus connects the processing outputs of a side-set to the associated processing inputs of any one of the cell's side-sets.

Transmisston links are important to our testing and repair processes in some arrays. A transmission link is a processing layer's chain of cells in

transmission states that acts as a two-way signal bus, connecting each processing input at one of its ends to an associated processing output at its opposite end. A transmission link performs the same bussing function in an array independent of the link's path or length.

Our loading approach uses cells in loading transmission states to transmit loading signals to the inputs of a cell being loaded. In testing arrays for embedding some relcon-3 and relcon-4 machines, transmission links conduct test signals from a test machine, such as a computer, to a tested cell. These same links concurrently return a tested cell's response back to the test machine. Arrays tested in this way are repaired by linking clusters of good cells via transmission links.

An embedded machine *arm* is a chain of relcon neighbors (see figure 1.5). The arm's *tip* has relcon-1, and all other cells in the arm have relcon-2. The relcon-2 cells are the arm's *body*. The *base* of the arm is the cell farthest from the tip in the relcon network's chain. A loading arm is used to load the cells in an array; loading signals flow from the loading arm's base to its tip, where a cell is loaded. We discuss mahines which are easily embedded as arms in the processing layer of a machine.

We develop a loading mechanism that can be coupled with any programmable logic processing mechanism in an array of two or more dimensions. This loading mechanism allows the loading of any cell in a perfect, arbitrarily large array by signals input to one cell anywhere in the array. This is possible because a *loading arm* may be grown to the loaded cell. The loading arm is an arm

Fig. 1.5 Relcon Network For Two Embedded Arms

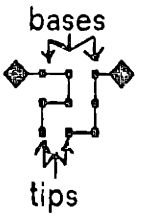
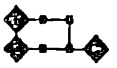
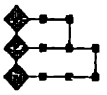


Fig. 1.6 Relation Between Essential Network And Associated Relcon Networks

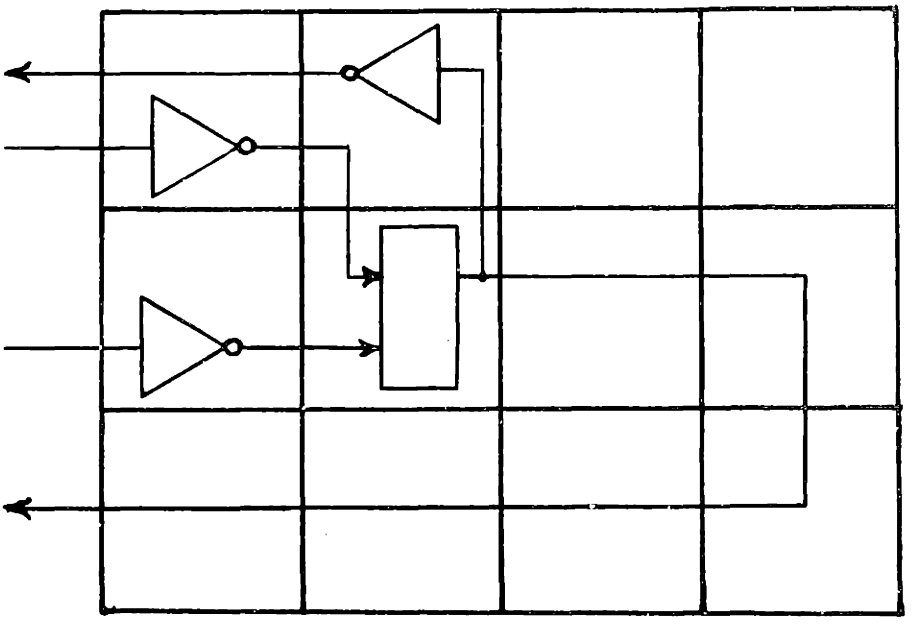
A) Essential network for figure 1.3's embedded machine



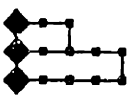
B) Relcon network for figure 1.3's embedded machine



C) An embedded machine equivalent to 1.3's embedded machine



D) Relcon network for the embedded machine in C



embedded in the loading layer of an array. Cells in the body of the arm are in loading transmission states, carrying loading signals from the base of an arm to its tip. The loading mechanism in each cell is *mono-active*: loading inputs from any one side-set are sufficient to affect any desired loading behavior of a cell in a perfect array, and for most cells in a flawed array. When a loading side-set is activated, a cell prepares to accept loading information. This information is then clocked into the cell through the active loading lines, setting the cell's loader and function states. If these loading inputs remain active, the loader state determines how the cell's loader subsequently behaves. The cell may enter a loading transmission state, in which it transmits its loading inputs to loading outputs at one of its side-sets. That is, it may become part of the body of a longer arm loading some new tip cell. Thus a cell may load one of its neighbors, which loads one of its neighbors, ...; so a flexible arm is extended into an array. This process is called *arm growth*; it is used to grow a path that carries loading information from one cell's loader inputs to other cells in an array. A tip cell may be loaded with a loader state preparing it to be re-loaded; this is useful when a cell is tested in various function states. A loaded tip cell may also cause its relcon neighbor in the loading arm to be the new tip; the former tip's loading inputs are de-activated, and the loading arm is incrementally retracted. A signal to the base of an arm can also cause the arm to totally retract; that is, the signal can de-activate all the loader inputs of all the cells in the arm. Thus signals to the base of a loading arm can extend the arm in one of several directions, retract the arm, or repeatedly

change the state of the arm's tip. Because only a loading arm's tip cell can have its function state changed, a cell's temporary role as an arm's tip is sufficient to permanently set its function state. A loading arm may also re-load cells in an array, in order to re-customize or repair the array. In a flawed or irregularly shaped array, the arm's ability to snake through alternate paths, twisting around flaws and retracting when necessary, gives it advantages. Chapter 3 details the loader, and discusses how loader options extend the loading arm's capability.

Use of a balanced loader and balanced function states facilitates machine-embedding in a flawed array. A loading arm's flexibility in routing loading signals derives from the mono-active, balanced nature of the loader. The mono-active property of the loader implies that a working cell can be activated, loaded, and de-activated by a loading arm linked to the loader inputs of one of the cell's side-sets. The *loader's balance* means that ANY side-set's loader inputs may be used to activate a cell for loading, load the cell with a desired function state, subsequently send loading signals through the cell to ANY side-set's loader outputs, and de-activate the cell's loader. The loader's balance allows a loading arm to funnel the same loading command to an arm-tip cell independent of the arm's path through an array.

The term "balance" is used to indicate a cell mechanism's functional symmetry with respect to its side-sets. A cell's processing mechanism may be balanced for some or all of the cell's function states. Consider some function state S_A of the processing mechanism. This state can be completely described by a set

of statements relating each subscripted processing output - $O_{U(M-1)}$ . . . O_{LN} - and each processing state (if applicable; that is, if the processing state can affect some processing output in function state F_A) to subscripted processing inputs and aplicable processing states. If each permutation on the side-set subscripts in this set of statements - such as the permutation interchanging L and U, but keeping R and D where they are - yields a set of statements that completely describes some allowed function state, the processing mechanism is *balanced* for state S_A and the *balance-related* function states generated by the side-set permutations. If a cell is balanced in every allowed function-state, the *cell* is *balanced*. Sometimes the construction of a cell requires disallowed function states. One might, for instance, use four function-specification state bits for thirteen allowed function states. Three function states might be incidentally generated, useless, and therefore disallowed.

An example clarifies the concept of a processing mechanism's balance. Consider some processing mechanism with one processing input and one processing output at each of its four side-sets. One transmission function state F_A of this mechanism is described by the set of statements below.

For F_A: $\{I_D \rightarrow O_U, I_U \rightarrow O_D, I_L \rightarrow O_R, I_R \rightarrow O_L\}$

An arrow indicates an input is transmitted to an output. This processing mechanism is balanced in state F_A if and only if the cell has function states F_B and F_C such that the following statements are true.

For F_B: $\{I_D \rightarrow O_R, I_R \rightarrow O_D, I_L \rightarrow O_U, I_U \rightarrow O_L\}$

For F_C: $\{I_D \rightarrow O_L, I_L \rightarrow O_D, I_U \rightarrow O_R, I_R \rightarrow O_U\}$

If the mechanism is balanced in state F_A, it is obviously balanced in states F_B and F_C. We then say that F_A, F_B, and F_C are balance-related. If the mechanism is balanced for every allowed function state, the cell is balanced.

Section 1.3: Basic Fault Assumptions

We review other work relating to testing, loading, or repair of cellular arrays. Our approach is the first one we've seen that specifies modules that allow a computer to test, configure, and repair an arbitrarily large, flawed array via leads connected to a few cells in the array. This approach requires assumptions about faulty behavior. One basic assumption is that a good cell is loaded under a test machine's control, and not by signals caused by faulty cells. In any approach that allows appropriate signals into a finite set of cells to affect loading of a cell, there is some chance that faulty cells will provide those signals to load a cell without a test machine's control, and therefore contradict this assumption. We describe design techniques for making this arbitrarily unlikely; this involves making the set of valid loading commands smaller than the set of possible loading commands, so that fault-generated commands are likely to be disobeyed. Another basic assumption is that the behavior of a cell depends on that cell's state and inputs, and not on the state of other cells. Our checkerboard arrays help assure the validity of this assumption, because no signal line connects distant cells. A third basic assumption is that a faulty cell is somewhat consistent in its faulty behavior: if a cell is good whenever it or its neighbors are tested, the cell must be good in the intervals between tests. If a good cell becomes flawed, it may not pretend to be good whenever it is tested. The first and third assumptions are met if a faulty cell's outputs all remain stuck at some value. Another assumption, which is made to reduce test time, states independence of certain mechanisms in a cell. For

instance, it's assumed that the state of a shift-register stage does not affect the performance of a remote stage in the shift-register. We describe array-design techniques which help assure the validity of all our assumptions. These assumptions all seem reasonable, and are all similar to assumptions made in testing conventional large-scale integrated circuits and other digital systems. However, ultimate justification of these assumptions requires experimentation with carefully designed arrays embodying our approach.

Section 1.4: Processing-layer Machines

Because the relcon-2 machines embedded in the arrays of chapter 3 are *arm machines*, which are machines that are always embedded in the processing layer as arms composed of balanced cells, these arrays are particularly easy to test and repair. Proper communication of a digital machine with one side-set of one working base cell in a flawed, arbitrarily large array allows test and repair of the array. A loading arm gradually extends an embedded arm machine into an array. After each extension, the arm is tested via processing signals between the Array Programmer and the base of the arm. If growth is unsuccessful, the arm is retracted and then grown through a new path in the flawed array. The balance of the cells in the arm machine imply that all arms with the same number of good cells may perform the same function, independent of their path through an array. For consider some embedded arm machine M1 whose T function states are described by the list (F_{11} F_{21} . . . F_{T1}). F_{11} is the function state of the arm's base, which is the first cell in the arm, and F_{N1} is the function state of the Nth cell in the arm. F_{T1} is the function state of the arm's tip. Because all the function states are balanced, an embedded machine M2 is equivalent to M1 if and only if M2 is an arm machine that has T function states (F_{12} F_{22} . . . F_{T2}), and F_{N1} is balance-related to F_{N2} for all $1 \leq N \leq T$.

Chapter 3 discusses programs that govern embedding of arm machines, given reasonable models of flawed behavior. Repair of arm arrays is studied through a program that simulates that repair (see figure 3.17 and 3.18 for pictures

of a repaired array). In most flawed arrays with N percent flawed cells, $0 \leq N \leq 25$, this program embeds an arm machine containing (100 - 2.2 N) percent of the total cells in the array. This performance can be slightly improved. When N is greater than approximately 35, only very small arm machines can be embedded in a checkerboard array.

Chapter 3 also discusses other issues relevant to practical realization of arm machines. In particular, it presents a fairly detailed plan for realization of enormous computer-repairable, variable-length shift-registers in a single IC package. Arm machine realizations are appropriate to many machines which are realized as a chain of modules, with each module communicating with at most two other modules, and only the modules at the end of the chain directly connected to the machine's inputs and outputs. Many one-dimensional cellular arrays have this characteristic, so they could be appropriately realized as arm machines in a flawed checkerboard array.

An *essential machine* is a perfect machine embedded in a processing layer that is described as a machine composed of essential cells that are wired together in some way. The *essential cells* of an embedded machine are those cells that are not in non-branching transmission states. The four cells in the upper-left square of figure 1.3 are the essential cells of that embedded machine. A *wire* carries relevant information between an essential cell and its *essential neighbor*, which is another essential cell or extra-array machine. A wire is a directed signal path from an output; this path is either direct (between interconnection neighbors) or

indirect (via cells in transmission states). The output of the element in the middle cell of figure 1.3 is wired directly to the essential cell above it, and indirectly to an extra-array machine.

An essential machine is associated with a class of equivalent embedded machines. The arm machines described in chapter 3 are composed entirely of essential cells. If an essential arm machine is an arm with T cells whose Nth cell is in function state F_A, $1 \leq N \leq T$, then an embedded machine is equivalent if and only if it is also an arm with T cells whose Nth cell is balance-related to F_A.

Just as a relcon network describes an embedded machine, an *essential network* describes an essential machine. Figure 1.6.A shows an essential network for the essential machine in figure 1.3. A square in the network stands for an essential cell, and sides of the square stand for side-sets of the essential cell in the obvious way. Diamonds stand for extra-array machines. If and only if a wire connects an essential cell's side-set to an essential neighbor's side-set, an associated link appears in the essential network in the expected way. An essential network is obviously related to the relcon networks of the embedded machines in its associated equivalence class (see figure 1.6). Each square in an essential network must have one and only one corresponding *essential node* in each associated relcon network. If two cells are essential neighbors in an essential network, each relcon network must have a path between corresponding essential nodes. This path may be a link (between essential cells that are interconnection neighbors), or a line of links (corresponding to cells in transmission states carrying

one or more wires).

Chapter 4 discusses arrays embedding processing-layer machines called *high-relcon machines*. All embeddings of a high-relcon machine contain some cells with three or four relcon neighbors. Our high-relcon repair procedure only recognizes embedded machines as equivalent if they have the same essential cells, and differ only in the length of wires connected to relevant inputs and outputs of these essential cells. For every essential cell in one high-relcon embedded machine, there is one and only one corresponding cell in an embedded machine that our high-relcon repair procedure judges to be equivalent. Similarly, for every wire in one high-relcon embedded machine, there is one and only one wire in an equivalent machine. Since essential states of our high-relcon machines are unbalanced, corresponding essential cells must be in the same function state. Corresponding essential cells are wired to other corresponding cells and extra-array machines in the same way. Only the length of associated wires may differ in equivalent embedded machine; chapter 4 discusses the timing implications of this fact. Our repair procedure ignores the equivalence of high-relcon machines built of different essential cells for simplicity's sake. This is reasonable, since a designer typically specifies the simplest essential machine that will perform a desired function.

Our repair of high-relcon machines requires the assumption that the length of associated wires may differ in equivalent embedded machines. While we mention techniques which help assure this assumption is valid, we do not study the

most appropriate high-relcon machine architectures. The assumption that the length of associated wires may differ in equivalent embedded machines is not required for repair of arrays embedding the other machines we detail.

Like chapter 3, chapter 4 focuses on testing, configuration, and repair. We show that the mechanisms that provide these facilities are very close to comparable mechanisms presented for chapter 3's arrays. The loaders are functionally identical. Testing is accomplished by growth of transmission links between a test machine and a tested cell. A *test link* is a transmission link between a test machine and some cell in an array. The base of the link connects to the test machine, and the tip of the link is the cell on the opposite end of the test link. Each cell in the test link conducts signals to and from the tip end of the link, where a tested cell may be located. The test link is grown as the body of a *test arm*, which is a test link terminated on a cell in a "U-turn" function state. Signals from a test machine into the base of this test arm flow down the arm's body to its tip, turn, return to the base of the arm, and end at the test machine. The test machine uses such signals to monitor the growth of a test arm. The balanced states of cells in the arm allow it to flexibly snake around flawed cells as it grows from a test machine to the side-set of a tested cell. Test links are grown to all the accessible side-sets of a tested cell. These links allow a test machine to monitor the tested cell's behavior in various function states, which are set by the loader.

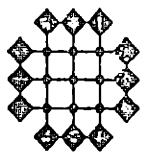
Repair of arrays embedding high-relcon machines is accomplished by use

of transmission states to wire together essential neighbors which aren't interconnection neighbors. Experiments with a repair simulator allow us to begin to compare the repair costs involved with different essential machines. The most difficult checkerboard-array repair involves embedding a high-relcon machine whose essential network is a grid (see figure 1.7.A). We call such a machine a *grid machine*, and call its essential cells *grid cells*. The most general repair of a checkerboard array assumes that a grid will be embedded in the array. Many high-relcon machines have essential networks with squares and links missing (see figure 1.7.B). We'll see that these machines are embedded in a flawed array more easily than grids machines. This shows that knowledge of the constraints on relevant inter-cell communication paths during an interval facilitates repair, but may require re-repair when the interval ends.

Chapter 4 also details an array of simple cells designed for the realization of arbitrary digital machines. Others have described infinite arrays that may contain initially-finite machines capable of performing any computation and of constructing other machines that can perform any computation. Our array is the first one we've seen capable of embedding a universal computer-constructor-repairer. A computer may be embedded in a finite portion of the processing layer of the array. A function state that transmits processing inputs as loading outputs provides this computer with a loading arm. (This is the only time that our previous description of programmable logic is slightly incorrect, in that processing inputs may affect loading outputs.) Under this embedded computer's control, the loading

Fig. 1.7 Essential Networks For Two High-relcon Machines

A) Grid



B) Non-grid

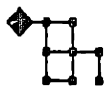
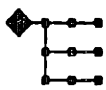
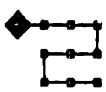


Fig. 1.8 Relcon Networks For Two Equivalent Tree Machines

A) A tree with several branches



B) A tree that is also an arm



arm works with four test arms to test, program, and repair the computer's environment. The machine may construct more memory for itself by using its loading and test arms. Furthermore, two or more computers embedded in an array may test and repair each other. We briefly describe an embedded machine we've designed as the processor of a universal computer-constructor-repairer.

Chapter 4 also discusses practical production issues and application areas relevant to high-relcon machines.

Chapter 5 discusses processing-layer machines called *tree machines*. Random-access and track-addressed sequential-access memories may be efficiently realized as tree machines in flawed arrays. This is true because tree machine realizations are appropriate to machines which may be viewed as a small set of modules with a common input bus and common output bus, with the output bus accessed by only one active module at a given time. Each cell in a tree machine is a balanced, essential cell whose function state includes a unique name. All embedded tree machines have tree-like relcon networks - relcon networks in which a tree trunk, which may or may not have offshoot branches, extends from the tree's base cell (see figure 1.8). A tree's *base cell* is the only cell that is directly connected to the input-output lines of the tree machine. Two embedded tree machines are equivalent if and only if they have the same set of cell names; the particular shapes of their tree-like relcon networks are irrelevant. Thus an embedded tree machine whose relcon network is an arm may be equivalent to an embedded tree machine whose relcon network has several branches. Tree

machines are embedded in flawed arrays more easily than arm or high-relcon machines. If there is any path of good cells between two good cells in a tree array, those good cells may be incorporated in the same tree machine. Interwoven test and repair processes for tree machines are like those for arm machines.

Section 1.4: Array Repair

Chapter 2 reviews particularly relevant work involved with cellular arrays. Many have presented particular array designs. Some have presented methods for testing and repairing particular arrays. Most of these methods use custom metallization, but some use programmed repair. Some have concentrated on necessary and sufficient conditions for testability or diagnosability of a particular type of array. We design cell modules which are incorporated into an array to enable testing, loading, and repair. We also present the first systematic treatment we've seen of the affect an embedded machine's communication structure has on the testability and repairability of an array for embedding that machine.

We describe how constraints on the wiring between essential cells of a machine affect testing and repair of an array embedding that machine. Chapters 3, 4, and 5 consider this question by focusing on three related classes of machine - the arm, the grid, and the tree. Figure 1.9 indicates how these three classes relate. Given a flawed array is to embed a given type of machine, we model the repair process in the following way. The flawed array is viewed as a *flaw pattern* (see figure 1.10.A), with a dot corresponding to a good cell and an X corresponding to a flawed cell. The machine to be embedded in the processing layer is associated with an essential machine and a class of equivalent embedded machines. In considering repair of an array, this class is restricted to embedded machines whose dimensions allow them to fit into the flawed array. The nature of this

equivalence class partly depends on the function states available to an embedded machine. A particular embedded machine is chosen from this equivalence class so that the machine's relcon network fits into the flaw pattern without touching any X. The array is repaired to embed that machine. For instance, consider embedding a desired arm machine with 13 cells; all associated embedded machines are arms containing 13 cells. Only some of these embedded machines have a relcon network that fits into the flaw pattern of figure 1.10.A. Figure 1.10.B shows one such relcon network superposed over the flaw pattern. The associated embedded machine may be embedded in the flawed array.

We noted that the nature of an equivalence class depends on the function states associated with a given array. For instance, balanced states may expand the size of an equivalence class and therefore facilitate repair. Arms and trees use balanced cells to facilitate testing and repair. The balance of cells in transmission links facilitates repair of arrays embedding high-relcon machines. Figure 1.10.C shows a 3 x 2 grid machine embedded in a flawed array. The relcon-2 nodes in the network correspond to transmission links connected to grid cells. In grid-embedding, cells used as links are overhead associated with repair.

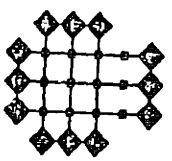
For every flaw pattern and class of essential machine, there's an associated *optimum repair efficiency*, which is the highest attainable ratio of the number of embedded essential nodes to the number of dots in the flaw pattern. In figure 1.10 the optimum repair efficiency is 6/14 for grids, 13/14 for arms, and 14/14 for trees. Let ORE_G, ORE_T, and ORE_A be the optimum repair efficiencies for

Fig. 1.9 Relation Between Grids, Trees, And Arms

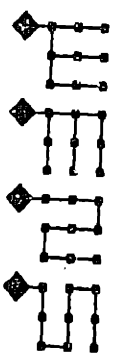
If a grid's essential network has a certain number S of squares, all the grid's associated relcon networks have at least S nodes. For each of a grid relcon network's N nodes, there are one or more tree subnetworks with N nodes. N is greater than or equal to S. One or more of the tree subnetworks of a grid's relcon network are arms; each has n or fewer nodes. At least one arm has S nodes.

A) The simplest relcon network of a grid - its essential network

B) Another relcon network for the same grid



C) Tree subnetworks of A's relcon network



D) Tree subnetworks of B's relcon network

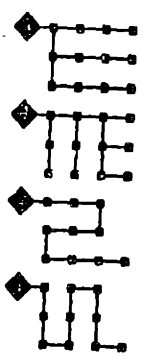
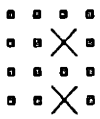
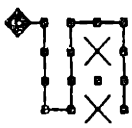


Fig. 1.10 Repair Of Arrays With The Same Flaw Pattern

A) Flaw pattern

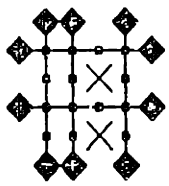


B) 13-node arm in flawed array



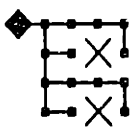
The repair efficiency is 13/14.

C) 6-square grid in flawed array



The flaw pattern has 14 dots. The embedded relcon network has 14 used nodes, but 6 of these are essential nodes and 8 of these are relcon-2 overhead nodes associated with transmission links. The repair efficiency is therefore 6/14.

D) 14-node tree in flawed array



The repair efficiency is 14/14.

grids, trees, and arms for a given flaw pattern. Because of the relation between grid, tree, and arm machines noted in figure 1.9, $ORE_G \leq ORE_A \leq ORE_T$ for any flaw pattern. Chapters 3 and 4 explore repair efficiency attained by programs that simulate repair for arms, and for grids and other high-relcon machines. Chapter 4 compares the results of these experiments. Experimental and theoretical exploration of testing and repair argue for designs oriented, when possible, toward limited requirements on the communication paths between a machine's essential cells.

Chapter 6 summarizes the thesis, and suggests further production-oriented and theoretical projects.

The next chapter provides context by exploring other systems approaches, comparing them to this one, and considering evolutionary trends which suggest that this approach will become increasingly attractive.

CHAPTER 2: CONTEXT

Section 2.0: Introduction

This chapter puts this work in context with respect to relevant system approaches and evolutionary trends. Key parameters of cellular arrays are discussed, and the relation of our approach to these parameters is detailed. Cellular arrays and conventional IC systems are compared. Fabrication of cellular arrays on a silicon slice is shown to be similar to conventional fabrication of IC circuit "chips" on a slice. Four evolutionary trends are discussed: rapidly increasing capability of integrated circuits, increased reliance on electronic machines, mass-production of a few high-volume components, and increasing regularity. Our approach is viewed as a systems approach tailored to the realities and trends in digital system design, manufacture, distribution, and maintenance. Other efforts toward very high integration, testing and repair, and cellular machines are reviewed, and they are compared to our approach.

Section 2.1: Cellular Arrays

2.1.A Introduction

This section locates our cellular approach in the domain of cellular arrays. We focus on distinctions in array interconnection, customization, size, and function. We briefly consider the current state of proposed array systems.

The behavior of a cellular array depends on the functional capability of its cells, and their interconnection. Since electronics is currently most suited to implementation of a cell's function, we describe cells using corresponding terminology. However, the approach applies to functionally equivalent arrays realized in other technologies.

2.1.B Array Interconnection

Arrays with many different types of interconnection have been studied, but 1- and 2-dimensional arrays are most common. In a checkerboard array, a cell may send signals to and from at most four neighbors. The cutpoint array, and other arrays with the same type of signal flow, have been extensively studied. These *cutpoint-connected* arrays have the same interconnection network as a checkerboard array, but signals may only enter a cell from its left and upper neighbors, and leave the cell to enter its lower and right neighbors. We chose a richer interconnection structure, with its slightly higher associated cost, for several reasons.

1) Most machines require fewer cells and less associated delays in checkerboard arrays. Cutpoint-connected arrays are limited by the fact that an operation on the outputs of some cells cannot be performed above the lowest of these cells, or left of the rightmost of these cells, without external connections for this purpose. Checkerboard arrays don't have this limitation, because each cell outputs in all four directions. This means, for instance, that the feedback connections of an embedded sequential machine may be formed inside a checkerboard array.

2) Signals from an arbitrary cell in a perfect, arbitrarily large array can cause loading of an arbitrary cell in the array only if there is an interconnection path from the loading cell to the loaded cell. This important capability is therefore impossible in cutpoint-connected arrays.

3) Repair is more flexible in checkerboard arrays, due to the larger set of possible processing transmission states.

The checkerboard array's interconnection structure is highly compatible with the two-dimensional, step-and-repeat nature of IC production. Furthermore, this structure is relatively easy to understand and manipulate compared to, for instance, hexagonal two-dimensional structures.

2.1.C Customization Techniques

Another aspect of cellular arrays is their customization technique. All but the simplest arrays have the property that each cell can be customized via memory elements to one of a set of function states corresponding to various output functions. Thus an array can be customized to realize a particular embedded machine by one of several customization techniques.

Unalterable customization late in IC production is a common, extreme form of array customization. A common technique uses selective metallization via a mask, fusible metal links, laser, or mechanical scribe. Polycell, Micromatrix, Read-Only Memory (ROM), and Programmable Logic Arrays (PLA) provide well-known examples of this approach. Because such customization is unalterable, design or customization errors can be particularly disastrous.

Programmable ROMs (PROMs) achieve greater flexibility by allowing customization that is alterable, albeit currently difficult. One such technique uses FAMOS transistors, which can be put in 1 of 2 conduction states by appropriate electric signals (see <Feeney 72>.). Intel guarantees each transistor to hold its state for 10 years. High-energy ultraviolet light or x-rays can erase these memory elements for subsequent re-programming. Difficulties include the high voltages, long write-times, and difficult erasing associated with the FAMOS transistor. Happily, Stanley Mazor of Intel expects that logic-programmable, logic-eraseable FAMOS transistors will be developed soon. This would provide the great advantage of a logic-compatible, read-mostly, nonvolatile semiconductor

memory.

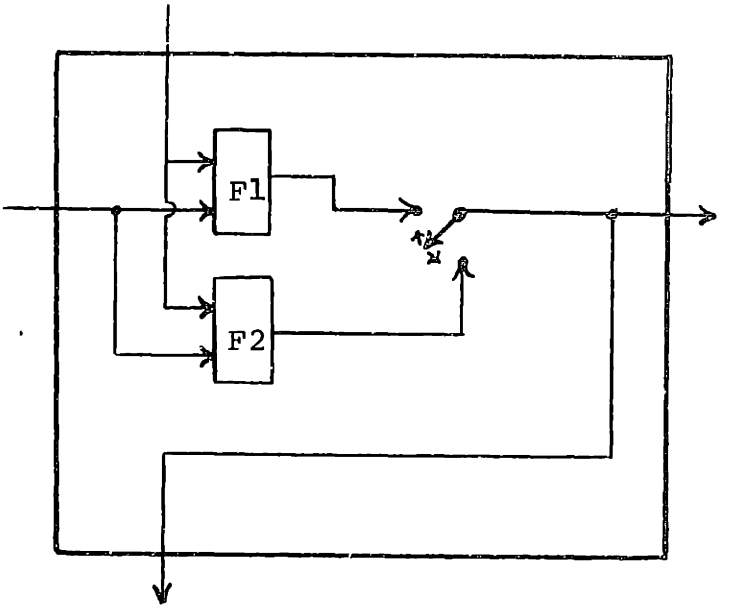
Programmable Logic (not to be confused with Programmable Logic Arrays) provides the ultimate in customization flexibility, but currently suffers from volatility. The arrays presented in this thesis are a type of programmable logic. Re-customization of programmable logic is as easy as loading its function-specification state bits. We develop an approach which facilitates test and repair of arrays of programmable logic. Building test and repair mechanisms into a programmable array can provide lower system test and repair costs than those associated with less flexibly customized integrated circuits.

Because a practical implementation of programmable logic would probably be realized via semiconductor technology, and because semiconductor memories are currently volatile, programmable logic is currently volatile. Development of logic-compatible nonvolatile semiconductor memory, such as a modified form of FAMOS gate, would offer big advantages for programmable logic.

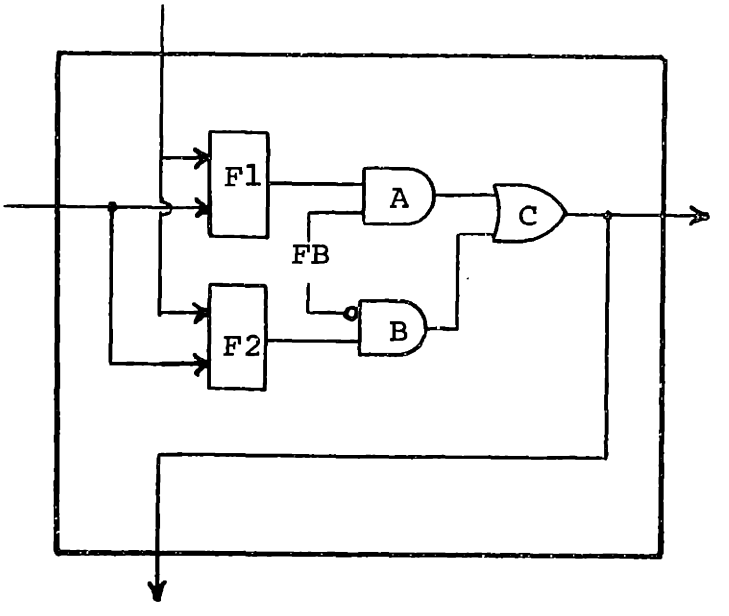
A further difficulty with programmable logic is its increased cell-delays compared to metal-customized arrays. This is true because there are delays associated with the selection of function via logic gates (see figure 2.1). While there is no denying this difficulty, two facts ameliorate the situation. The first is that the delays through gates A and B in the figure can be made very small because these elements can be designed assuming FB, a function-specification state bit, will not change state during normal operation of an embedded machine. This means that a cell-delay can be about one gate-delay. The second is that logic

Fig. 2.1 Two Customization Techniques

A) Metal-customized



B) Programmable logic



gates are becoming increasingly fast, especially when designed for a known, simple, low-load environment. If Josephson junction gates become practical, the expected delay through a gate of about .1 nsec. is the same delay as that through 3 cm. of wire.

A final problem with programmable logic is its demand for extra gates for loading function-specification state bits and selecting a particular output function. These gates consume an integrated circuit's area and power. The fact that extra area is required for these gates is offset somewhat by the fact that programmable logic minimizes non-circuit programming facilities, such as the many area-consuming bond pads required by some custom metallization techniques. This becomes more significant as shrinking transistor geometries make bond pads and other mechanical customization components occupy a relatively higher gate area. The power consumption problem is alleviated by the fact that function-specification state bits change state infrequently; in some technologies, an element's power dissipation is very low when the element is not changing state.

Because of the pin constraints on ICs, most proposals for loading programmable logic attempt loading via electric signals through leads at the edge of an array. Chapter 3 reviews the most attractive methods that have been suggested for programmable logic loaders, and gives a loading approach with advantages achieved by adding a small amount of circuitry to each cell in the array.

2.1.D Size

Size is another distinguishing attribute of cellular arrays. A particular array's size depends on the size of each cell and the number of cells. The best measure of an IC cell's size is the amount of area it occupies, but this measure is too dependent on technology, designer, design time, and design aids to be useful in preliminary estimation of the size of cells. Consequently, the normal measure of size is gate-count of the cell. This measure has limited value because of the variable types, number of inputs, and density of gates, and because of the tradeoff between input-output lines and gates for a cell performing a particular function. Nevertheless, several authors have used gate count as a means for roughly classifying arrays (see <Minnick 67> and <Mukhopadhyay 71>). They distinguish between microcellular arrays, in which each cell contains only a few gates, and macrocellular arrays, in which each cell contains a large number of gates.

The cells presented in this thesis use few logic elements and few function states. The loading mechanism, the only mechanism common to all the cells we discuss, is shown in figure 3.5. It has a minimum of about twenty gates and five memory bits, with slightly more if loading options are incorporated. A processing mechanism of any size and complexity may be combined with the loader. The actual complexity chosen for a cell depends on the envisioned application, and on a tradeoff between yield and overhead circuitry. In the memory arrays we've designed, this tradeoff is the main consideration in determining how large a memory to put in each cell. The universal cell presented in chapter 4 uses less than one-

hundred gates and memory bits, and only fourteen function states. This simplicity increases cell yield and reduces test time.

<Mukhopadhyay 71>, <Kautz 71>, and <Minnick 67> have compared the number of cells of different types required to perform various functions. We make no such comparison here, for many of the functions we perform cannot be performed in other proposed arrays. All our techniques are applicable to arbitrarily large arrays.

2.1.E Function

Various functional categorizations of arrays have been made. These include consideration of the functional capabilities and the time behavior of cellular arrays.

The most common functional classification views an array according to its ability to do combinational logic, memory, or more general sequential machine functions. <Shoup 70> discusses this in terms of "generality" of the array. Our testing and repair techniques work for any cell generality. We discuss some memory cells in chapters 3 and 5, and a sequential machine cell in chapter 4.

The chapter 4 array is able to realize an arbitrary digital machine. In particular, the array can support a finite-configuration universal computer-constructor-repairer. That is, a finite number of cells can be programmed out of their initial quiescent states into an embedded machine able to perform any computation, to create a new, disjoint embedded machine able to perform any

computation, and to do these things in a faulty array. The embedded machine's use of a loading arm and test arms allows it to test and program its environment. It can, for instance, enlarge its memory by proper loading of cells. <Rowan 73> describes a cellular array, of more complicated cells, that is computation-universal, but not capable of construction or repair.

In a synchronous cellular array, all cell states are re-calculated simultaneously. Several synchronous arrays capable of supporting universal computer-constructors have been presented. Von Neumann's 1952 pioneering work, *Theory of Self-reproducing Automata*, presented such a 29-state automaton (see <Von Neumann 66>). <Codd 68>, <Gardner 70>, and <Banks 71> followed with simpler cells. While theoretically interesting, synchronous arrays are peripheral to this thesis because they are currently impractical. Since state changes must be synchronized, many technologies require long clock lines linking all cells to a common clock. Signal transmission is severely limited by the clock frequency, since a signal takes at least one clock interval to propagate from a cell to its neighbor. Thus the transmission delay through a cell in a synchronous array is at most the reciprocal of the toggle frequency of its memory elements, which is much slower than the transmission delay of one gate-delay associated with our asynchronous arrays. The overhead circuitry for all proposed synchronous arrays is high. Testing, loading, and repair appear to be more difficult for these arrays.

Asynchronous cellular arrays are far too numerous for extensive consideration here. <Minnick 67> provides an excellent early review. In a more

recent presentation of a theory of logic design with cellular arrays, <Mukhopadhyay 71> summarizes and analyzes some of the major cellular arrays. <Kautz 71> discusses various arrays for arbitrary logic, including sequential machines and special-purpose arrays; many of the designs are his own.

2.1.F Current State

Cellular arrays are already widely used. Popular IC arrays need no customization (Random-Access Memories), Shift-Registers), or only a simple customization step (Read-Only Memories, Programmable Logic Arrays) (see <Luecke 73>). There are also a few systems using many ICs, such as the Illiac IV.

However, many proposed arrays remain paper-studies for various reasons, including current impracticality and IC industry inertia. The characteristics of some of these arrays have been discussed in this section, as background for our approach. This approach overcomes the difficulties of many proposed cellular arrays. Its loading, test, and repair circuits, and their associated programs, are compatible with many arrays that have been proposed.

Section 2.2: Array Fabrication

Fabrication of LSI checkerboard arrays is similar to fabrication of conventional integrated circuits. In the conventional approach, hundreds of identical ICs are batch processed by selective doping and metallization of a wafer that is usually 2" to 3" in diameter. Typically, a key element in this complicated process is use of masks to selectively expose photosensitive material on the wafer to light. Each mask is formed by photographic reduction of a pattern. That pattern is formed by use of a step-and-repeat process which iterates a basic sub-pattern throughout an array. Each sub-pattern corresponds to one of the iterated IC's masks.

Each of a wafer's identical circuits contains bonding pads, which are used for probe-testing and possible connection to the IC package. After a wafer has been batch-processed, each of the identical IC "chips" is tested via electric communication with a test machine connected through probes to the chip. Those chips that are defective are inked. The wafer is diced along horizontal and vertical scribe lines into component chips. Those chips that have been inked are discarded. The other chips are packaged and retested. Those that pass these final tests are ready for use.

For a checkerboard cellular array, a basic circuit is similarly step-and-repeated to form an array of identical circuits. However, the patterns of edge-sharing neighbors overlap slightly to allow lines to interconnect neighbors. Because most of the identical circuits, cells, communicate only with their edge

neighbors, most do not need bond pads. Only lines that may be used for extra-array communication need bond pads. Scribe lines between cells are unnecessary, for the array need not be diced. Scribe lines are only necessary between parts of a wafer that are intended to be parts of different arrays.

Most conventional IC packages are eventually connected to other, similar packages. Thus a given chip's life-cycle usually includes batch-processing with identical chips, separation from them, and eventual re-connection to other chips. We'll see that there are many advantages to a systems approach that doesn't require separate handling of each chip. This separation is now required for two principle reasons. First, conventional ICs cannot function properly if any of their components are faulty; this necessitates making a chip small enough so that there's a reasonable chance it will be perfect. Second, a system of small IC chips requires a variety of chips; a slice contains only one type of chip.

This thesis' cellular approach eliminates the need for separation of chips in many cases. A slice is designed so that electric communication with a digital machine allows testing and repair of the slice; faulty regions of the slice can be tolerated. Chapter 4 discusses such an array of identical, simple cells that are so flexible that a large enough array can perform any computation, and can test and repair its cellular environment.

This thesis focuses on checkerboard arrays designed to tolerate defective cells. Cells are programmable logic cells; in each working cell, programmable function-specification state bits specify the function that the cell is

to perform. Once an array has been formed, electric communication of a test machine with a small number of cells anywhere in the array tests the entire, arbitrarily large array. The testing machine uses the same communication links to program the array to embed a perfect machine, by appropriate setting of function-specification state bits. The same and/or other communication links then provide the inputs and outputs of an array-embedded machine, realized as cells in proper function states. Re-customizing the array is as simple as re-loading its function-specification state bits. Should an array machine become defective because of a malfunction of its circuitry, it need not be discarded. Its links can be used for testing and repairing the array. Because this repair can be done electrically by a digital machine, repair can be automatic, standardized, and quick. Repair can even occur through communication between an array and a remote test machine. Indeed, the universal array of chapter 4 can accept embedded machines that test and repair each other.

The array can be viewed as a bin of spare parts, cells. For a large bin, there is a high probability that a certain percentage of parts will be good. Thus an array is fabricated to have more than enough parts for a particular envisioned application. The availiability of spare parts which can be electrically switched into active status allows the realization of IC packages with more functional power. That is, higher integration is attainable through automatic repair. Furthermore, this spare-part capability facilitates re-customization, reliability, and maintainability. For many types of circuit failure, an array can be re-programmed to resume

performance like a perfect array. This allows graceful degradation of an array.

Simplification of system production and maintenance are achieved via basic, powerful capabilities designed into a simple, standard part. All circuitry that allows program-controlled testing, customization, and repair is in a standard, modular, mass-produced part, the cell. A single, mass-produced, commonly available general-purpose computer can use standard program modules to cause array testing, customization, and repair via electric communication with the cellular array. The standard, modular nature of an array and its *Array Programmer* implies low-cost, high-reliability realization of many systems.

In either conventional or cellular IC fabrication, each process step has an associated yield. That is, each process step tends to introduce loss of some of the product. Wafer-processing yield losses occur because of two basic types of defects - area or line defects, and spot defects.

Area or line defects involve the clustering of faulty components. Although they can occur anywhere, they occur most frequently at a slice's edge, due to handling, misalignment, and other factors. <Camenzind 72> states that they're usually caused by human errors and that, in a well-controlled process, they are rare and of little significance.

Spot defects, characterized by a random sprinkling of small flaws throughout the wafer, are more important and unavoidable in the foreseeable future. Their most common cause is photoengraving (see <Noyce 71>). They can result from, for instance, a dust particle between a mask and a wafer during

contact printing of the wafer.

If a wafer contained only spot defects, one would expect an exponential decrease of chip yield, as measured by the percentage of perfect chips, as a function of active (component-containing) chip area. <Hodges 72> notes that the yield model proposed by Dingwall is most rigorously documented by manufacturing experience.

$$Y = (1 + D_0A/3)^{-3}$$

Y is yield, the ratio of good chips to total chips. D_0 is the number of defects per square inch of slice. A is the active area of a chip, in square inches. In 1972, Hodges said that "a Do value of 200 per square inch is quite typical for normal operations by efficient producers of both bipolar and silicon-gate devices."

Because there are other process steps besides those for wafer fabrication, there are other yield losses in fabrication of a conventional IC system. Wafers are dropped and broken as they are moved between process sites. Perfect chips are packaged or bonded incorrectly. Test programs and testing machines fail, causing erroneous rejection of working ICs. Packages are mislabelled. (One engineer told us of a packaging site putting read-only memories with different contents in the same container, before labelling - they were all read-only memories.) Perfect ICs are miswired or broken.

Great expense is incurred in attempting to minimize yield losses. Manufacturers update their assembly lines, attempting to achieve a clean, efficient flow of materials. Circuits are designed under many difficult constraints. A prime

use of computer-determined wiring patterns which preserve the two-dimensional topology of the array. Custom metallization, which may pass over flawed cells, is used to convert a flawed array into a smaller, perfect array by proper connection of perfect cells. This approach assumes the interconnection network of a perfect array must be preserved; it does not consider the class of a machine embedded in the flawed array.

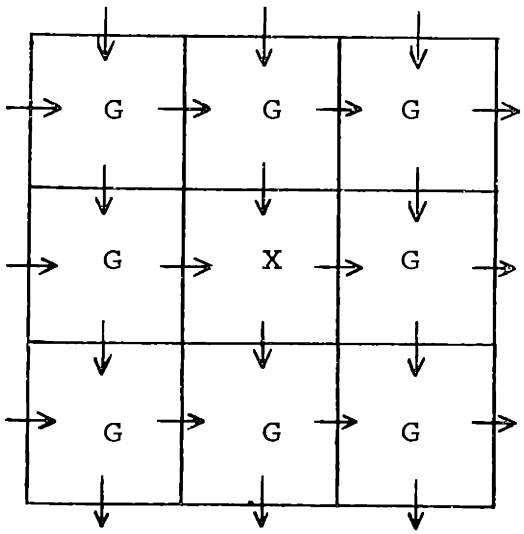
<Minnick 66> describes use of custom metallization for repair of various arrays. He also discusses the efficiency of associated repair strategies for cutpoint arrays.

The closest precursor of our testing and repair approach seems to be <Kukreja 73>. Testing a cell in a particular state involves including each of its inputs and outputs in a signal path to an edge input or output. Test signals to and from a tested cell are carried by cells in transmission states. Repair comes from programming cells in a row or column containing a faulty cell to behave like wires linking good cells (see figure 2.2).

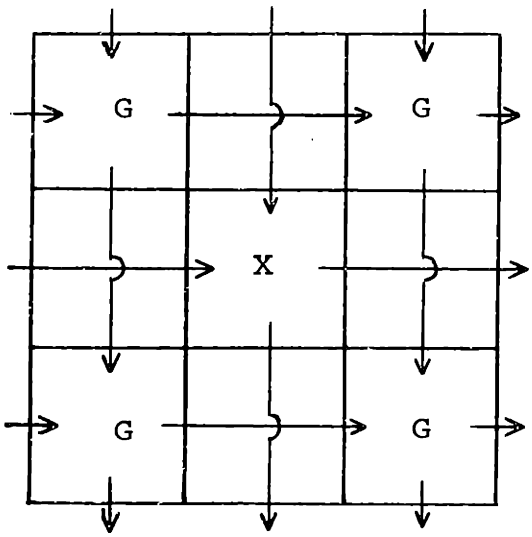
Our test and repair of arrays embedding high-reicon machines similarly involves use of cells in transmission states. However, there are key differences between our arrays and Kukreja's. Kukreja's 2-dimensional array is a cutpoint-connected array of simpler cells, in which each cell receives control variables on lines from a third dimension. Cells are not programmable logic cells, so there is no loader. Sequential machines are realized by a 3-dimensional stack of 2-dimensional arrays. Testing requires direct connection between a test machine and

Fig. 2.2 Programmed Array-Repair

Flawed 3 x 3 array



Perfect 2 x 2 array



X indicates a flawed cell
G indicates a good cell in an arbitrary state

all the edge cells of an array. Kukreja's approach requires far more cells and extra-array connections for testing of arrays, and for realization of most machines. Kukreja's repair approach is what we call "simple repair"; our repair procedure is more complicated, but also more efficient for most checkerboard arrays. Kukreja's emphasis is quite different. He does not address many of the design, testing, and repair issues we address.

In sum, our approach is the first one we've seen that details LSI-oriented circuit modules and describes associated software for low-cost, automated, electrical testing, repair, and customization of cellular arrays. It is a systems approach whose advantages will become clearer as its description becomes more specific in the following chapters.

CHAPTER 3: ARRAY-EMBEDDED ARMS

Section 3.0: Introduction

This chapter presents several examples of machines that are embedded as arms. Since any one of a large set of loading arms may be grown and retracted by loading signals input to one side-set of one arm base cell, flexible loading can occur in a flawed array. Processing-layer arm machines, which are composed of balanced cells, can be gradually grown and tested, and snaked around flaws in an array. We focus on the cell mechanisms and support programs that provide these capabilities. For specificity and practicality, we concentrate on the realization of highly integrated, length-programmable, computer-repairable shift-registers. However, our techniques apply to other arm machines. These techniques are easily generalized to tree and high-relcon machines.

We present a mono-active, balanced loading mechanism for growth of loading arms. The loading inputs of any side-set S of a cell are sufficient to load that cell's loader and function-specification state bits. After the cell is loaded, its loader state bits may specify which of the cell's neighbors, if any, receives loading information funneled through the cell from side-set S. The loader's balance allows an arm to funnel the same command to a cell independent of the path the arm takes in reaching the cell. Optional cell modules extend the loader's capability. The loader state may specify that an arm's tip be re-loaded, or than an arm incrementally retract. A brief signal to the base of an arm can cause the arm to

totally retract. We use this same loading mechanism throughout our work. A mechanism with its capabilities is easily incorporated in arrays of two or more dimensions.

The most interesting machines that can be embedded in the processing layer of this chapter's arrays are arm machines. Embedding is particularly easy for these machines. Typically, the loading and processing arms grow together through the same cells. The Array Programmer adds one cell at a time to these arms; and tests the new, extended arms after each extension. The Array Programmer only communicates directly with the processing inputs and outputs of one side-set of an arm's base cell. When the arms encounter a flawed cell, they may be partially or completely retracted, and grown through different cells in the array. An arm may be grown in any direction, avoiding flawed cells, because of the loading arm's flexibility and the fact that an arm of balanced cells is grown in the processing layer. Our description of the testing and repair processes depends on a model of flawed cells' behavior, which we state and analyze. We study repair efficiency through a program that simulates repair, and suggest techniques to improve efficiency. We consider other issues relevant to the practical implementation of our arrays.

Repair through the interwoven processes of arm growth and testing contrasts to repair of high-relcon arrays. Since the requirements on the communication paths between essential cells of a high-relcon embedded machine are more stringent, repair efficiency is enhanced by the location of all the flaws in

an array before a global determination of a good way to repair the array. Repair efficiency therefore dictates that a test procedure is limited in its ability to predict the role a given cell will play in an embedded machine. No matter how large an arm machine grows, its inputs and outputs are always at one side-set of its base cell. Interwoven processes of machine growth and testing are hampered when a machine's growth implies a sometimes-growing number of inputs and outputs: the number of connections between a test machine and a partially grown embedded machine is variable, and may be large. These considerations encourage us to test all the cells in a high-relcon array before repairing that array. However, the test and repair processes for high-relcon machines use the same loader described in this chapter, and balanced processing transmission states similar to the balanced states in this chapter. Many practical implementation issues are similar. Thus this chapter is useful in itself, and as a bridge to the next chapter.

Since a tree machine may be embedded as an arm, it's not surprising that the approach described for arm machines is readily adapted to tree machines. Because a tree machine may be realized by an embedded machine with any tree-like relcon network, including an arm network, an array embedding a tree machine is particularly easy to test and repair. Embedding a tree or arm machine is considerably easier than embedding a high-relcon machine.

Section 3.1: The Loader

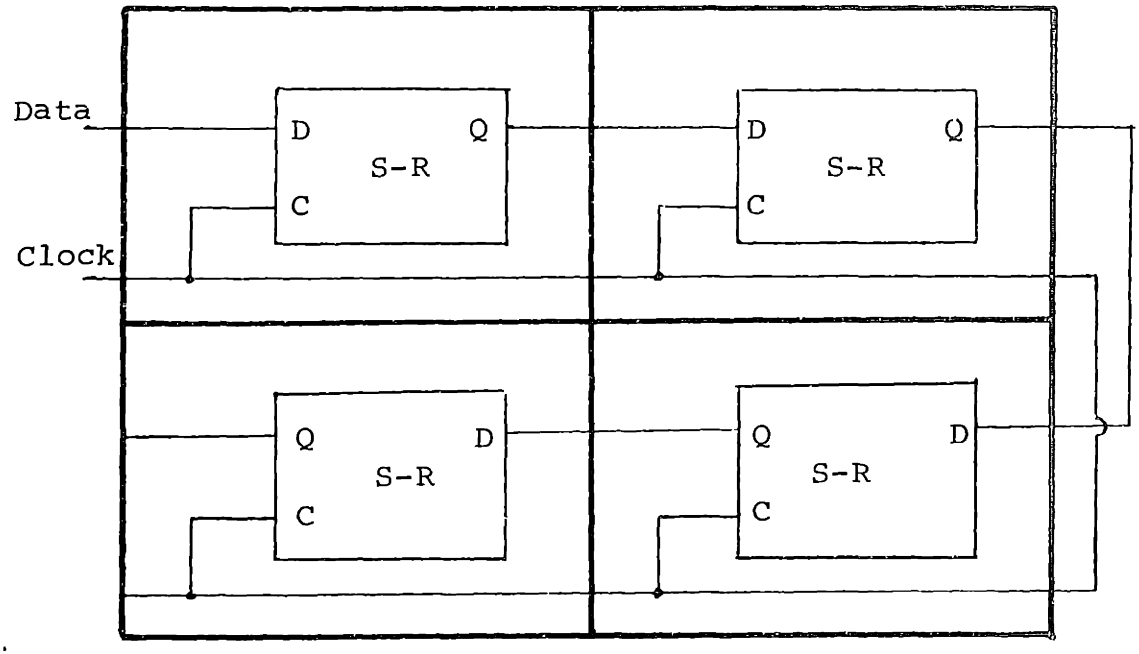
In programmable logic, a loading mechanism is used to load the function-specification state bits in a cell. Others have proposed loading mechanisms incorporating long, fixed, irredundant signal paths routing loading information to a given cell. These loading mechanism have major limitations, including susceptibility to catastrophic failure due to destruction of a long, critical loading line. We propose a method which incorporates extra logic elements in each cell to allow the flexible growth of a loading arm in an array. A loading arm is composed of cells in proper loading states, and not long lines. Since an arm may be grown from any cell, an entire array can be loaded via inputs to one cell in the array.

Figure 3.1 illustrates two common methods for loading the function-specification state bits contained in a shift-register in each cell. Figure 3.6 describes a shift-register's operation. Snake and Crisscross use parallel-out shift-registers, with each output acting as a function-specification state bit connected to the processing layer. The shift-registers in Snake and Crisscross are parallel-out shift-registers, with each output acting as a function-specification state bit connected to the processing layer. These connections are not shown in figure 3.1. In Snake, each shift-register is part of a long shift-register that snakes through all the cells of the array (see <Spandorfer 65>). (If each function register is associated with one cell, cells in different rows have different loading inputs and outputs; the array is strictly cellular only if we conceptually group mini "cells" into a macro cell.). In Crisscross, one of Wahlstrom's methods, each cell is associated

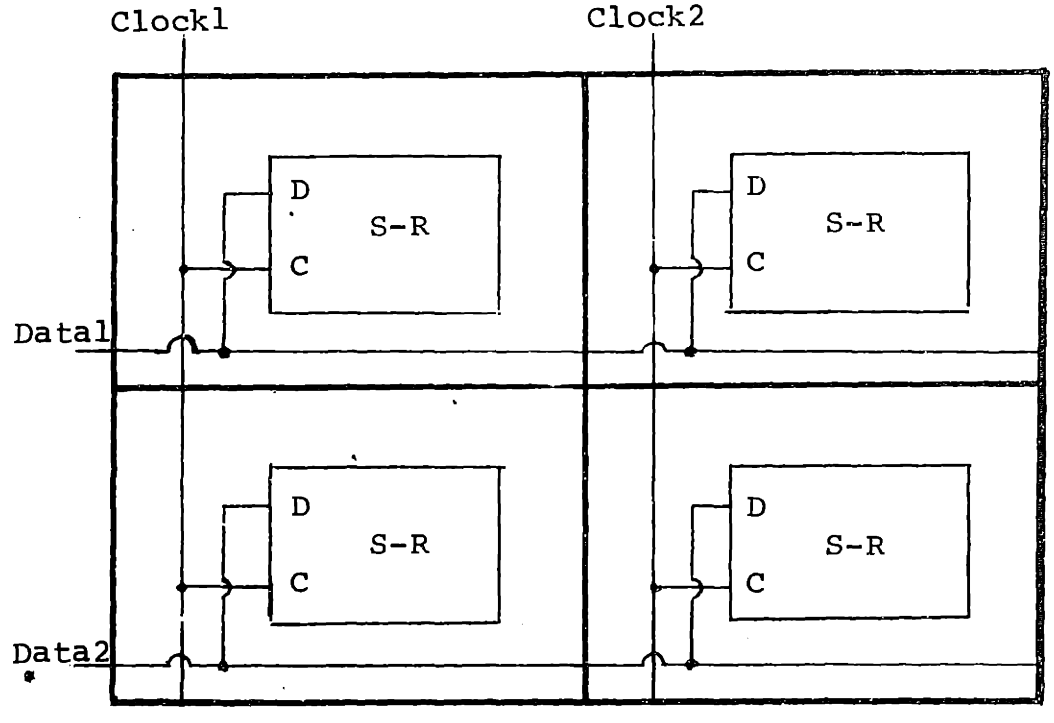
Fig. 3.1 Two Common Programmable Logic Loading Mechanisms

(The function registers are shown without their outputs to the processing layer.)

A) Snake



B) Crisscross



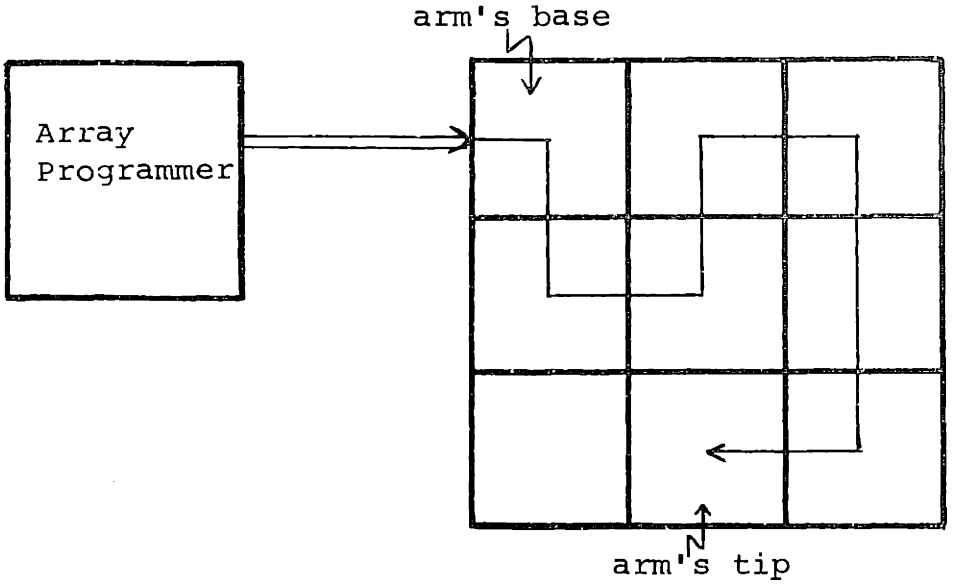
with a unique clockline, dataline pair (see <Shoup 70> and <Wahlstrom 69>). Each clockline extends through a column of cells, and each dataline extends through a row of cells. Co-column cells must therefore be loaded simultaneously.

Note that these methods could operate on a more general type of array. For instance, function shift-registers in different cells could have different lengths, drive different circuitry, etc. That is, the key idea is that a loading mechanism is iterated through the array. The loading techniques we describe are also useful in this type of array, if it has two or more dimensions.

Like the loading methods of figure 3.1, our loading method loads function-specification state bits into a shift-register. However, our loading method uses logic elements in each cell to allow loading information input to any cell to be routed to any other array cell that is not walled off by faulty cells. Loading inputs set a cell's function-specification state bits and loader bits. The loader bits specify how subsequent loading information input to the cell is to be handled. They may, for instance, specify that it is to be routed to some neighbor-cell. Consequently, loading information to a cell may be routed to any cell in the array by a loading path, or arm, of cells in appropriate loader states (see figure 3.2). Loading signals input to the base of the arm can load the tip of the arm, extend the arm from its tip, or retract the arm. Proper use of a perfect array's loading mechanism only allows the embedding of arms in the loader layer of the array.

Our flexible loading arm has several advantages compared to the loaders of Figure 3.1:

Fig. 3.2 A Loading Arm Grown By Array Programmer Signals



1) The methods of Figure 3.1 depend on long, inflexible signal paths. Each cell can be loaded in only one way, so the cell is useless if that way doesn't work. A signal path connecting many cells is a weak link in terms of repairability; its destruction severely limits the usefulness of the array. Furthermore load impedance, noise, and delay considerations make long lines undesirable. Our loading method does not require any long signal lines. In some technologies, such as magnetic bubbles, all long lines, including power supply lines, can be eliminated from the array.

2) The other methods require that many cells be loaded simultaneously, even if one only wants to load one. For Snake, this requires the reloading of an entire array even when one wants to change the state of just one shift-register in the array. For Crisscross, this requires the reloading of an entire column. In our approach, loading cell A from cell B requires a loading arm from A to B. If no arm already exists, only the cells on some path between A and B need reloading. Once a loading arm is formed, its tip can be easily moved around. This is particularly attractive because two successively loaded cells are usually close to each other.

3) Crisscross requires a loader input to the array for each row and each column of the array. Large arrays consequently require a large number of input pins and associated connections. Recognizing this

deficiency, <Wahlstrom 69> describes an extension of Crisscross. In this extension, a cell can enter a state in which a processing input is transmitted to a dataline above it or a clockline to its right. This allows loading of an arbitrary cell in an array via processing and loader inputs to a cell in the lower-left corner of the array. Wahlstrom admits that such loading is indirect and slow. Its utility is severely restricted if the lower-left corner of the array is faulty. Our method allows speedy loading of an arbitrary cell with the three loader inputs of a side-set of any one cell in the array. This implies that connecting the loading outputs of some cell in one array to the loading inputs of some cell in another array allows a loading arm to be extended from the first array into the second array. This minimizes the number of pins required for testing, loading, and repair in systems composed of several ICs.

4) For all the loading methods, one can envision a function state in which a cell's processing inputs control loading lines near the cell. (A machine in Chapter 4 uses such a state to allow a machine embedded in an array to test its environment, and to construct and repair machines in that environment.) For the other methods, there are harsh limits on the position and number of cells that can be loaded from such a cell, even in a perfect array. With our method, any cell can be loaded from any other cell that is not walled off by flawed cells.

5) A loading arm's flexibility allows it to avoid flaws in a faulty array.

6) Our method allows use of several loading arms simultaneously loading unrelated cells. Because the other methods involve loading lines extending through many cells, they do not allow this.

Our loader demands a small number of additional logic elements in each cell to achieve its advantages, but the cost of logic elements is declining rapidly compared to other system costs.

A cell's loading mechanism allows the loading of function-specification state bits in the cell. This mechanism consists of a Basic Loader, which is usually combined with one or more loader options. The site of loading activity in an array is the tip of a loading arm. The Basic Loader allows the extension of an arm to include any of the tip's neighbors which aren't already in a loading arm. A Total Retractor option allows the rapid destruction of an arm by a single signal to the base of the arm. An Incremental Retractor option causes a tip's relcon neighbor in a loading arm to be the new tip; the loading arm is then incrementally retracted. The Tip Changer option allows a tip cell to be repetitively loaded. The loader options demand extra logic elements, but extend the power of the loading arm.

Each cell in a checkerboard array has Select, Clock, and Data loader inputs and outputs at each of the cell's four side-sets. When an array's power is turned on, its working cells are inititalized so that all their Select output lines are low. Raising a side-set SS_i's Select input activates SS_i: Data and Clock inputs at SS_i may load the cell's register containing its function-specification and loader state bits. The newly Selected cell is the tip of some loading arm. A counter in

this tip cell counts the number of bits shifted into the cell's register after activation; so the cell knows when its register has been loaded. The loader state then specifies the new arm tip. With the Basic Loader, one of the loaded cell's interconnection neighbors that isn't in an arm may have its Select input at side-set SS_2 raised. Then Clock and Data information from the base of the loading arm flow through the arm, through the former tip, to the new tip. This process may iterate. Loader options extend an Array Programmer's ability to control a loading arm.

Figures 3.3 through 3.5 give an embodiment of our loading mechanism in a checkerboard cellular array. Most discussion of the loading mechanism is on a functional level. The loader can therefore be understood without reference to these diagrams, but they are included for specificity. Figure 3.3 shows the mnemonically-initialized names of the loading inputs and outputs of a cell. The loader lines are Select, Clock, and Data. Up, Right, Down, and Left refer to the cell's four side-sets. Figure 3.4 shows one possible realization for the Pulser in the lower-left corner of figure 3.5. After power is supplied to an array, the pulser's OUT line in each working cell remains low long enough to assure that all appropriate memory elements are simultaneously reset. Intel's microprocessors have a circuit with this effect. The other elements in figure 3.5 - a complete functional diagram of the loader for one cell - are familiar standard logic elements like those in a Texas Instruments TTL catalog. The function of each logic element is summarized in the symbol table in figure 3.6. These elements could be realized in many forms and technologies.

Fig. 3.3 Input-output Lines Of A Cell's Loading Mechanism

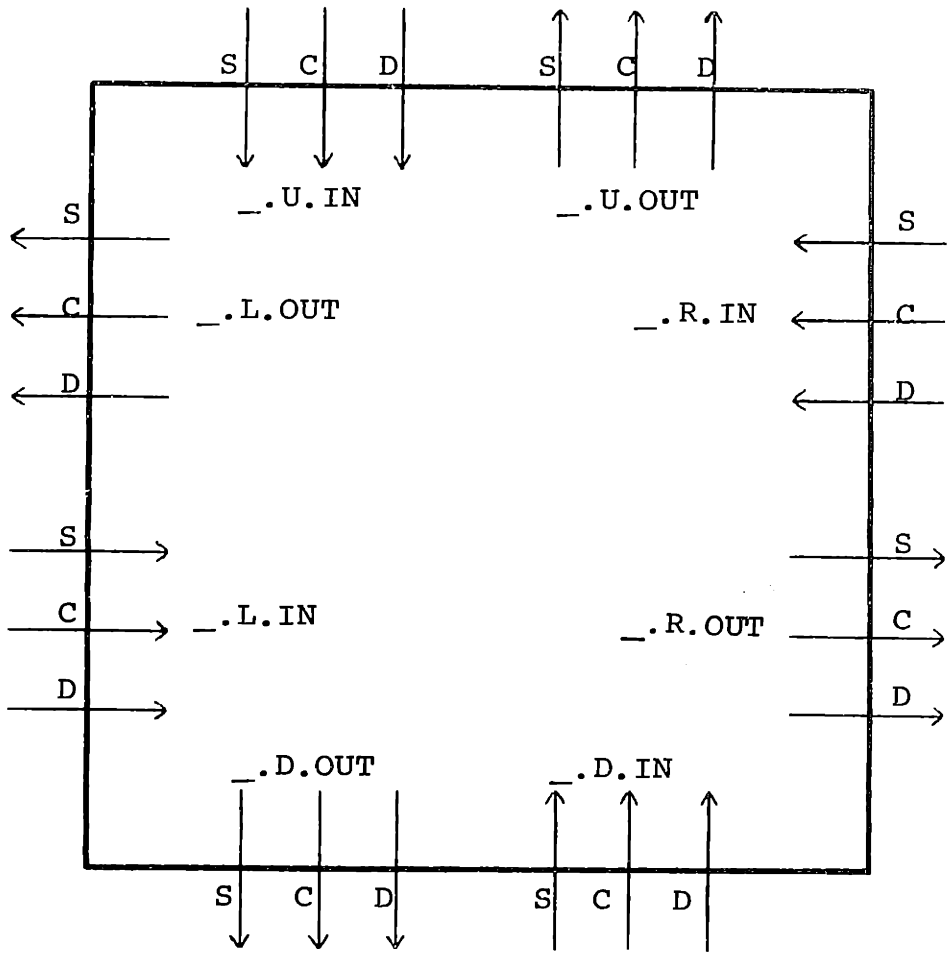


Fig. 3.4 The Loading Mechanism's Pulser

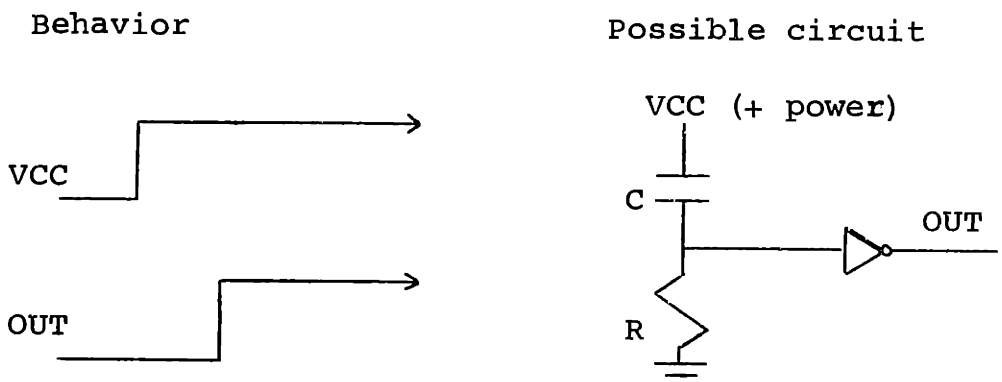
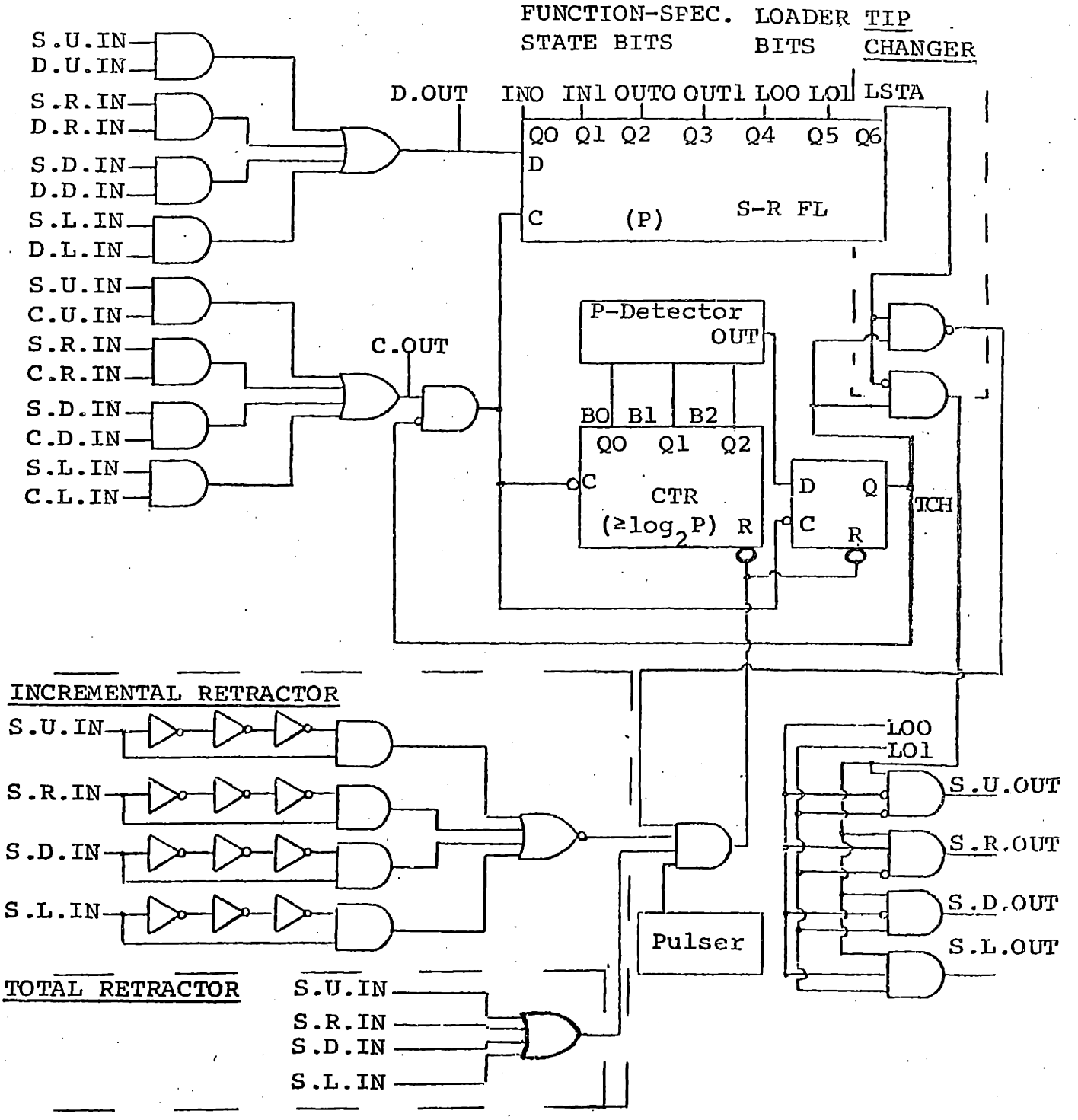


Fig. 3.5 Loading Mechanism With Options

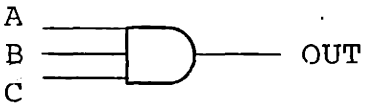


Options are marked by - - - boxes. Not using an option implies elimination of associated elements. If the resultant circuit has an AND gate with only 1 input, that gate is replaced by a wire connecting that input and the output.

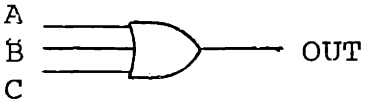
Fig. 3.6 Symbol Table

(1st of 2 pages)

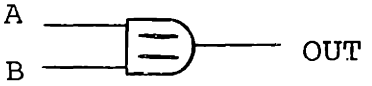
A) Combinational logic elements



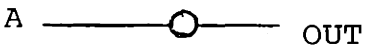
AND GATE: OUT = A AND B AND C. OUT is a logical high, or 1, if and only if A and B and C are all 1.



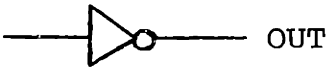
OR GATE: OUT = A OR B OR C.



EQUAL GATE: OUT = (A = B).



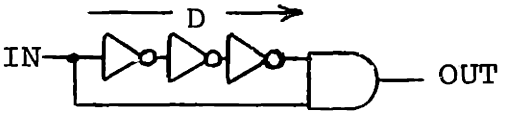
CIRCLE: OUT = ~A. OUT is the complement of A.



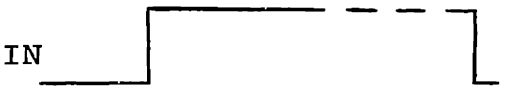
INVERTER: OUT = ~A.

B) Pulse-maker

Although combinational logic elements ideally act instantaneously, actual devices involve a slight delay before a change in the inputs is reflected at the outputs. This delay is used in the pulse-maker.



PULSE-MAKER symbol.



PULSE-MAKER behavior. We always use the Pulse-maker for inputs that remain high or low at least D time-units.

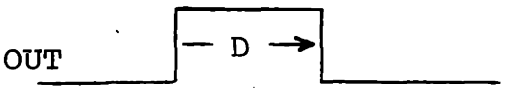
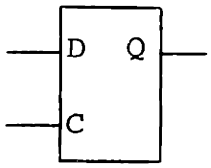


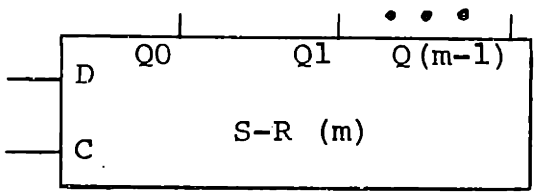
Fig. 3.6 Symbol Table (2nd of 2 pages)

C) Memory elements

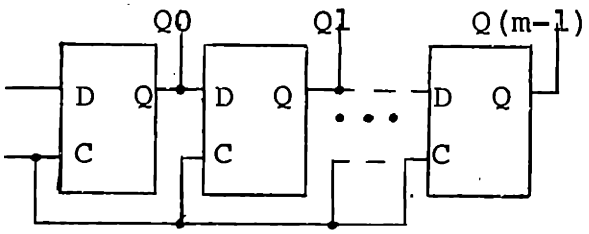
D always indicates a DATA input.
C always indicates a CLOCK input for a memory element. The memory element responds to a positive-going transition on C's input.
R always indicates a RESET input; all outputs of the memory element are 0 if R = 1.
Q always indicates an OUTPUT of a memory-element.



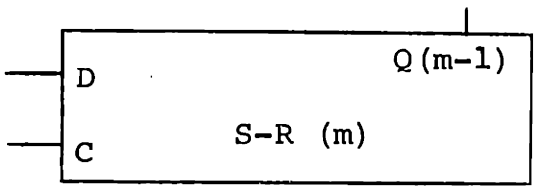
D FLIP-FLOP. Q takes the value of D when a positive-going C transition occurs.



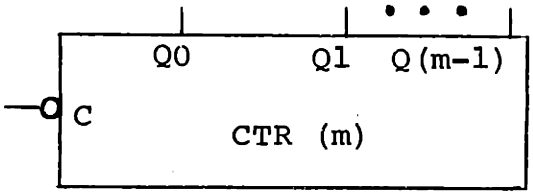
SHIFT-REGISTER (m-bit, serial-in, parallel-out). Outputs are Q0 through Q(m-1).



This is equivalent to the shift-register above.



SHIFT-REGISTER (m-bit, serial-in, serial-out). Only Q(m-1), the last bit of the shift-register, is output.



COUNTER (m-bit). This counts in binary, changing state on positive-going transitions of C. If m = 2, the counter has the state-transitions for (Q0 Q1) of: (0 0) to (1 0) to (0 1) to (1 1) to (0 0).

We first detail the Basic Loader, with none of the loader options. S-R FL is the shift-register containing the function-specification and loader state bits. LO0 and LO1 are the loader state bits that specify the loader's output side-set. LSTA is only included when the Tip Changer is used; this loader state bit specifies whether a tip cell is to be re-loaded. S-r FL may have any number of function-specification state bits. Here we assume four such bits - IN0, IN1, OUT0, and OUT1. CTR, a counter, counts the number of bits shifted into FL after a cell is activated. Since CTR must be able to count to P, the number of bits in FL, CTR is $\log_2 P$ bits. The P-detector outputs a 1 when CTR's count is P. For P = 6, the P-detector performs the function OUT = B2 AND B1 AND NOT B0 = 6, in binary. TCH, the "touch" flip-flop, signals that a cell has been loaded.

The Basic Loader is used to load a perfect array in the following way. When power is supplied to the array, the Pulser resets CTR and TCH to 0. The CTR, the P-detector, and the TCH flip-flop are used to determine when a cell's shift-register FL has been loaded. S-r FL is in an indeterminate state (although some processing layers require it to be pulser-resettable; this forces all cells into the same function state when power is turned on). All extra-array inputs to the array are 0. Assume that S.L.IN for some cell A goes from 0 to 1. Cell A has been TOUCHED from the left, and now its left loading inputs are ACTIVATED. C.L.IN and D.L.IN may now affect the cell's function-specification and loader state bits and its loader outputs. This prepares shift-register FL of cell A to be loaded via D.L.IN and C.L.IN. Since all other S.INs are 0, all other side-set's loader inputs are not

activated. D.L.IN is relayed to D.OUT and C.L.IN is relayed to C.OUT. Besides being the D outputs of the cell, D.OUT is the D input to shift-register FL. Since TCH=0, C.OUT is the C input to shift-register FL and CTR. The first positive transition of C.L.IN causes

1) D.L.IN to be shifted into shift-register FL, and

2) CTR to be incremented to (B0 B1 B2)=(1 0 0).

During loading of the cell, CTR functions to count the number of positive C.IN transitions since the cell was touched. That is, it counts the number of bits shifted into shift-register FL. Succeeding C.L.IN positive-transitions will similarly shift information into shift-register FL and increment CTR. The "p"th such transition (6th in this example) causes

1) the 6th D.L.IN bit to be shifted into shift-register FL, so that all the information in shift-register FL has been loaded from D.L.IN since S.L.IN went high; and

2) CTR=(0 1 1); this causes the output of the P-detector to go high.

Thus shift-register FL has been loaded with function-specification and loader state bits; the P-detector signals this fact by sending a high signal to the input to the D flip-flop. When C.L.IN next goes from high to low, TCH goes high. This causes

1) the C inputs of shift-register FL and CTR to remain low; C.OUT is no longer transferred to them; and

2) one and only one S.OUT to go high, thereby activating inputs at the side-set of some "touched" neighbor cell. The one selected is

determined by LOO and LO1.

The loading arm is a loading-signal path starting with some base cell with a high S.IN, and possibly extending from that cell to other cells, with the arm's path marked by high S lines linking neighboring cells. Figure 3.2 showed one such loading arm. With the Basic Loader we restrict a cell from touching a cell that is in a loading arm. In this case, this means cell A should not touch left, the source of loading information. It may touch cells that are up, right, or down neighbors. Assume cell A touches cell B above cell A. We then say that the loading arm's TIP has been moved up from A to B. This is caused by loading cell A with LOO=LO1=0; when cell A's TCH goes high, its S.U.OUT goes high. That is, cell B is then touched by cell A. Because S.L.IN is still cell A's only high S.IN, it's still true for cell A that C.OUT=C.L.IN, and D.OUT=D.L.IN. Because cell B is the only cell that A is touching, cell B is the only neighbor of cell A to accept C and D information from A. B can now be loaded in the same way that A was. B may then touch some new neighbor FL, funnel loading information to this new tip, etc. That is, this process of a cell's being loaded, touching a neighbor, and funneling loading information to that neighbor may iterate. In this way a loading arm may be snaked through an array, with its length only limited by the size of the perfect array. This growth of a loading arm to any cell from any other is facilitated by the loading mechanism's mono-active, balanced nature.

A brief example will illustrate loading via growth of a flexible loading arm. Assume a perfect array was to be loaded with function states (IN0 IN1 OUT0

OUT1) equal (0 0 1 0) for cell (0 0), (1 1 1 0) for cell (0 1), (0 1 0 1) for cell (1 0), and (1 0 1 0) for cell (1 1). The Array Programmer may connect to cell (0 0) in the manner shown in figure 3.8. After the array is turned on, all cells have CTR=TCH=0. The Array Programmer raises S.L.IN of cell (0 0), the base of the loading arm. The Array Programmer uses the C and D lines to clock out the sequence {0,0,0,1,0,0} in the manner indicated in figure 3.7. This loads shift-register FL with (IN0 IN1 OUT0 OUT1 LO0 LO1)=(0 0 1 0 0 0). That is, the function-specification state bits have been properly loaded and the loader bits LO0 and LO1 tell cell (0 0) to touch up. At T0, the first downward transition of C.L.IN after the loading of shift-register FL, S.R.OUT of cell (0 0) is raised. Cell (0 1) is now ready to receive C and D information from the Array Programmer, routed through cell (0 0).

The subsequent sequence clocked out of the Array Programmer via the C and D lines is {0,1,0,1,1,1}, {1,0,0,1,0,1}, {1,0,1,0}. Thus all the cells of the array have been loaded by a loading arm snaking through the array in the manner indicated in Figure 3.8. A different-shaped loading arm could have been used to accomplish an equivalent loading of function-specification state bits.

We now consider the various options available to enhance the capability of the Basic Loader. The Total Retractor allows a loading arm to be grown and later totally retracted by a signal to the base of the loading arm. This allows, for instance, reloading of cells and rerouting of a loading arm to new cells from the same arm base. With the Total Retractor, a perfect array is loaded just as

Fig. 3.7 Clocking Out The Loading Sequence 0, 0, 0, 1, 0, 0

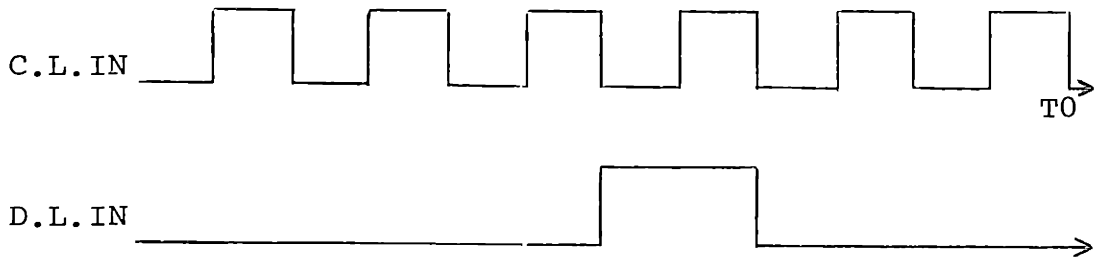


Fig. 3.8 A Loading Arm Formed By Touching Cells

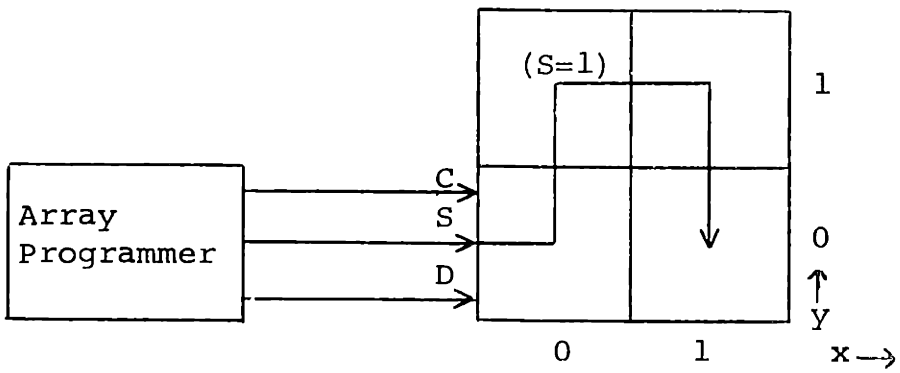
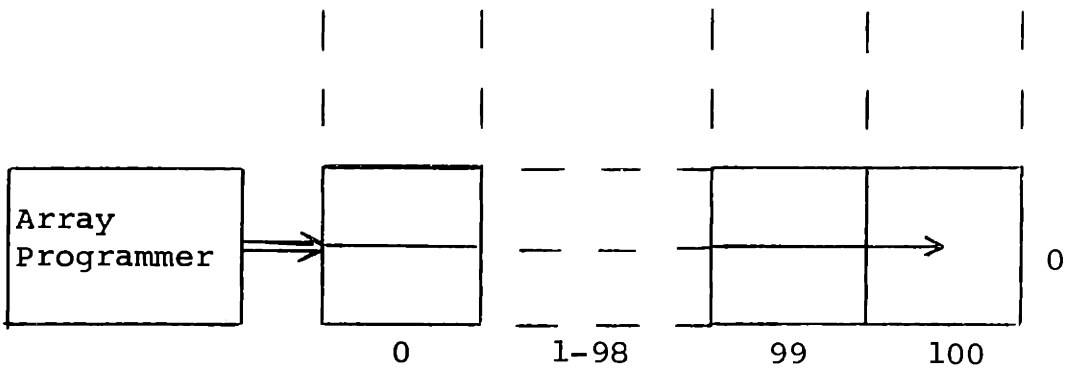


Fig. 3.9 Loading Arm With Tip At (100 0)



described above. Assume the loading arm of figure 3.8 exists. If the Array Programmer lowers its S line, no S.IN of cell (0 0) is high; the Total Retractor of cell (0 0) causes that cell to be reset to CTR=TCH=0. When TCH goes low, all S.OUTs of cell (0 0) go low. This resets (1 0), which resets (1 1), which resets (0 1). The function-specification state bits of these cells are unaffected. The Total Retractor thus allows the resetting of all the cells in a loading arm by lowering the S input to the base of the arm. These reset cells are then ready to be re-loaded by some new loader arm.

The Incremental Retractor allows a loading arm to be shortened cell-by-cell, instead of all-at-once as with the Total Retractor. The Incremental Retractor shown in figure 3.5 includes the Total Retractor circuit, so this Incremental Retractor is always used with the Total Retractor. The Incremental Retractor can save time when, for instance, one wants to change the state of a cell that is near the tip of a long loading arm. Consider the long loading arm of figure 3.9. If the Array Programmer wanted to reload cell (99 0), as it might on the basis of some test on cell (100 0), cell (100 0) could be loaded with information telling it to touch left. When S.L.IN of cell (99 0) went high, the incremental retractor of cell (99 0) would create a reset pulse. This would reset cell (99 0) for subsequent loading from (98 0). Resetting of (99 0) would lower S.R.In of (100 0), thereby causing (100 0)'s Total Retractor to remove (100 0) from the loading arm while leaving (100 0)'s function state the same. This incremental retraction is much faster than the equivalent action of total retraction and subsequent growth of the

loader arm from (0 0) to (99 0).

In loader realizations in which the Incremental Retractor does not include Total Retractor circuitry, the Incremental Retractor may be used for a type of total retraction. If all the other S inputs of a cell are low, lowering its S line and then raising it prepares the cell to be loaded from that S line's side-set. Assume an Array Programmer wanted to grow a new loading arm from the base of an existing, unnecessary arm. By lowering the cell's high S line, then raising it, the Array Programmer would prepare the cell to be loaded, forcing all the cell's S.Out lines low. The cell could re-touch the cell it last touched, or touch some other neighbor, and a new arm could be grown from the old base. Of course, part of the old arm might remain in the array. Under certain conditions this is intolerable; loading a cell in an old arm from some new side-set involves special considerations, as we'll see. Nevertheless, many applications make fast retraction feasible through exclusive use of the Incremental Retractor. When fast retraction is unfeasible, an arm may be totally retracted cell-by-cell via the Incremental Retractor, as we've discussed. If even this is impossible, due to a growth failure at the loading arm's tip, the Incremental Retractor allows a new arm to grow through the working cells of an old arm; subsequent incremental retraction frees these working cells from the loading arm.

The Tip Changer allows a tip cell to be repetitively loaded by the same loading arm. This is another time-saving device, particularly helpful when one wants to test the same cell in various states. It involves adding an extra bit to

shift-register FL, and consequently requires one more clock pulse for the loading of a cell. If a tip cell is loaded with LSTA=1, the downward C.IN transition at T0 (directly after the P-detector goes high) causes resetting of CTR and TCH to CTR=TCH=0. The fact that LSTA is high also prevents the cell from touching any other cell. Thus the cell is loaded with a function state, remains the loading tip, and is therefore ready to be immediately reloaded. If the cell is loaded with LSTA=0, it may touch another cell as if no Tip Changer existed.

Thus the Basic Loader can combine with a combination of Total Retractor, Incremental Retractor, and Tip Changer. The particular combination used in an array depends on the specific objectives for that array. In arrays designed for infrequent loading, minimization of circuitry by exclusive use of one Retractor option might be appropriate. The rest of this chapter details growth of shift-register arms. If program-variable shift-register length was important to an array, variation-speed considerations might encourage use of all loader options.

In summary, the fundamental loading mechanism allows loading inputs from one of several sides to control loading of a cell. The cell may learn that, and how, subsequent information input from its active loading side-set should be passed to loading outputs of some other side-set. If the set of loading mechanism neighbors is properly chosen, inputs to any cell may cause the loading of a cell anywhere in a perfect array, and loading of most cells in a flawed array. The loading mechanism may be incorporated into arrays with diverse processing layers.

Section 3.2: A Perfect Array Of Shift-register Cells

We now examine one realization of a complete shift-register cell, shown in Figure 3.10. We'll eventually show how an array of such cells can provide large, highly integrated, variable-length, automatically testable and repairable memories. For clarity, we begin by considering an array of such cells which contains no flawed cells. The shift-register cell's loading mechanism is almost identical to the loader of Figure 3.5. For simplicity, we assume that all the loader's options are included in the shift-register cell. In fact, the approach we describe can be adjusted to work with just a retractor option.

Each side-set has Select, Clock, and Data loader inputs and ouptuts, whose function has been described. In addition, each side-set has distinct Klock, iNput, and Return processing input-output lines; there is one set of K.IN, K.OUT, N.IN, N.OUT, R.IN, and R.OUT lines in each side-set. The shift-register cell could have been realized by disjoint loading and processing mechanisms. However, the cell shown in Figure 3.10 reduces circuitry and loading time by using bits in shift-register FL in a dual role as function-specification and loader bits. The loader of Figure 3.10 corresponds to that of figure 3.5 with the following mapping:

Figure 3.5: IN0 IN1 OUT0 OUT1 LO0 LO1 LSTA

Figure 3.10: IN0 IN1 OUT0 OUT1 OUT0 OUT1 STA

S-R FL is reset when power is turned on to limit processing layer complications of certain faulty cells.

Figures 3.11 and 3.12 give alternate functional descriptions for the

Fig. 3.10 A Complete Shift-register Cell

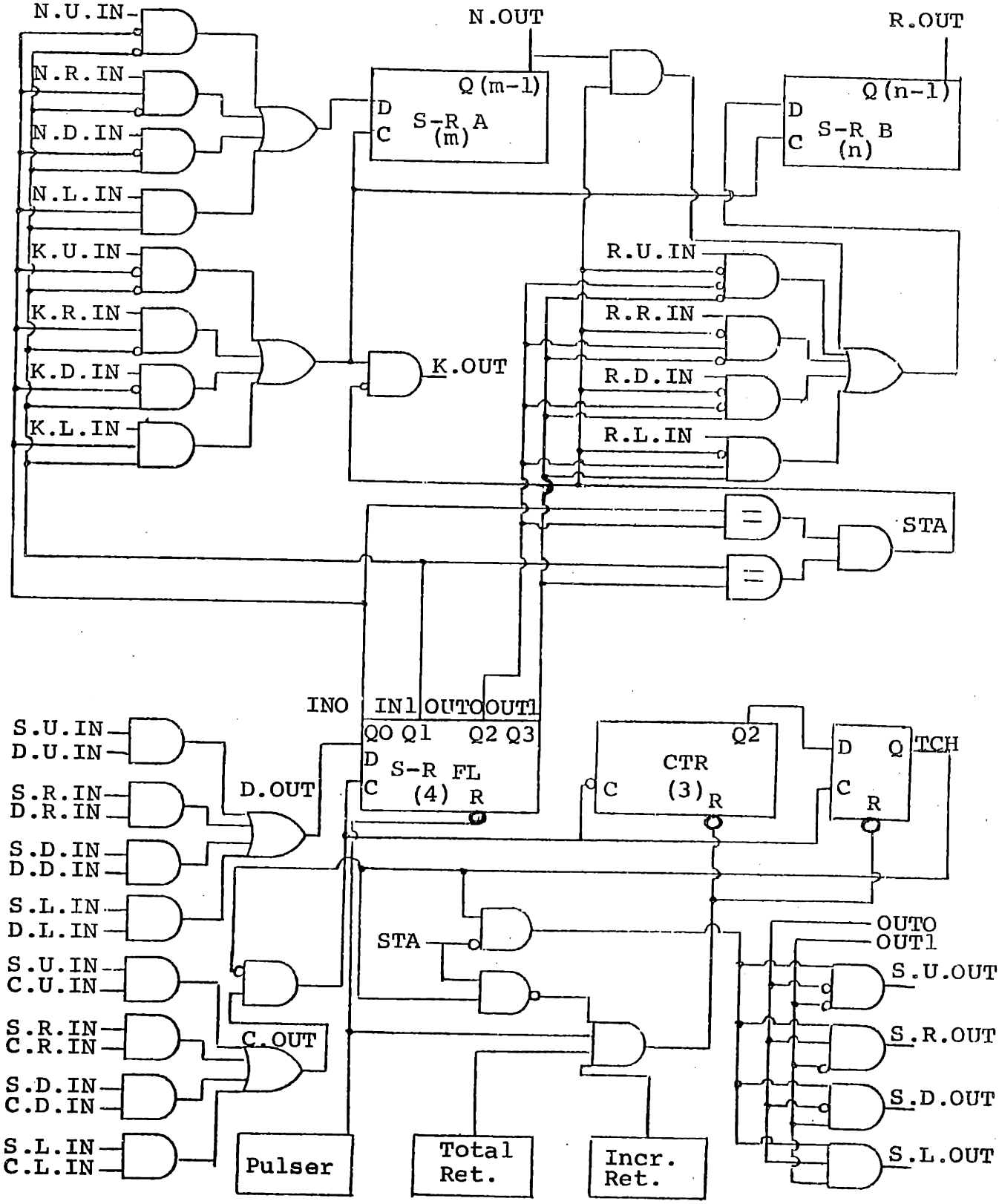
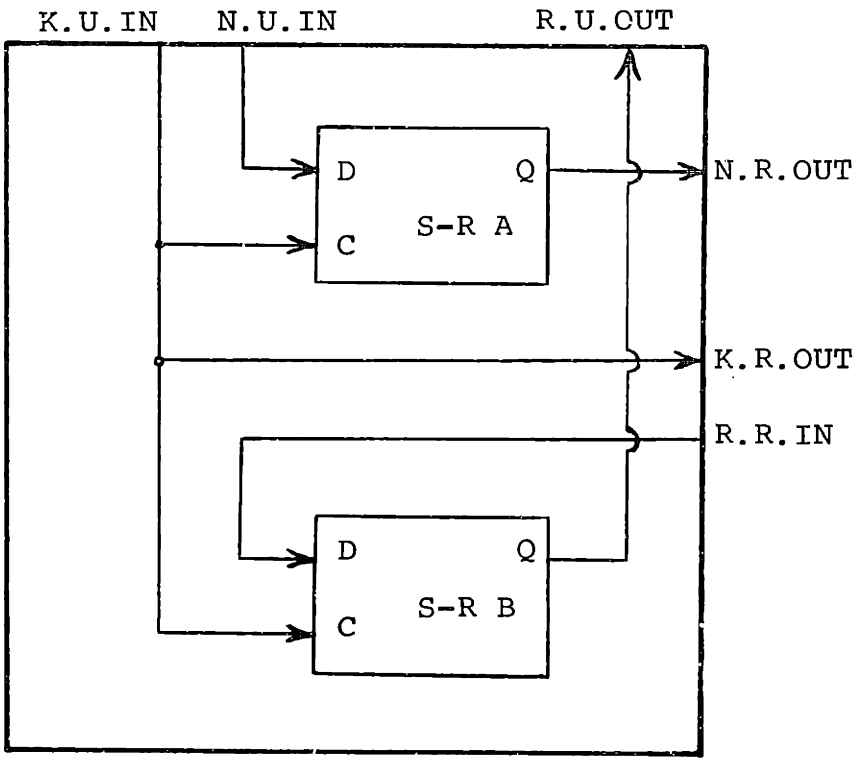
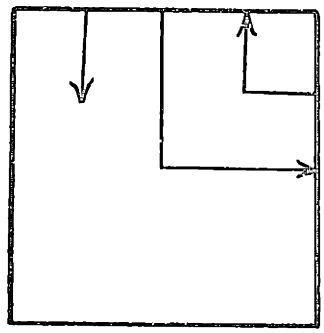


Fig. 3.11 Abbreviating A Shift-register Cell's Function State

Function state



Abbreviation

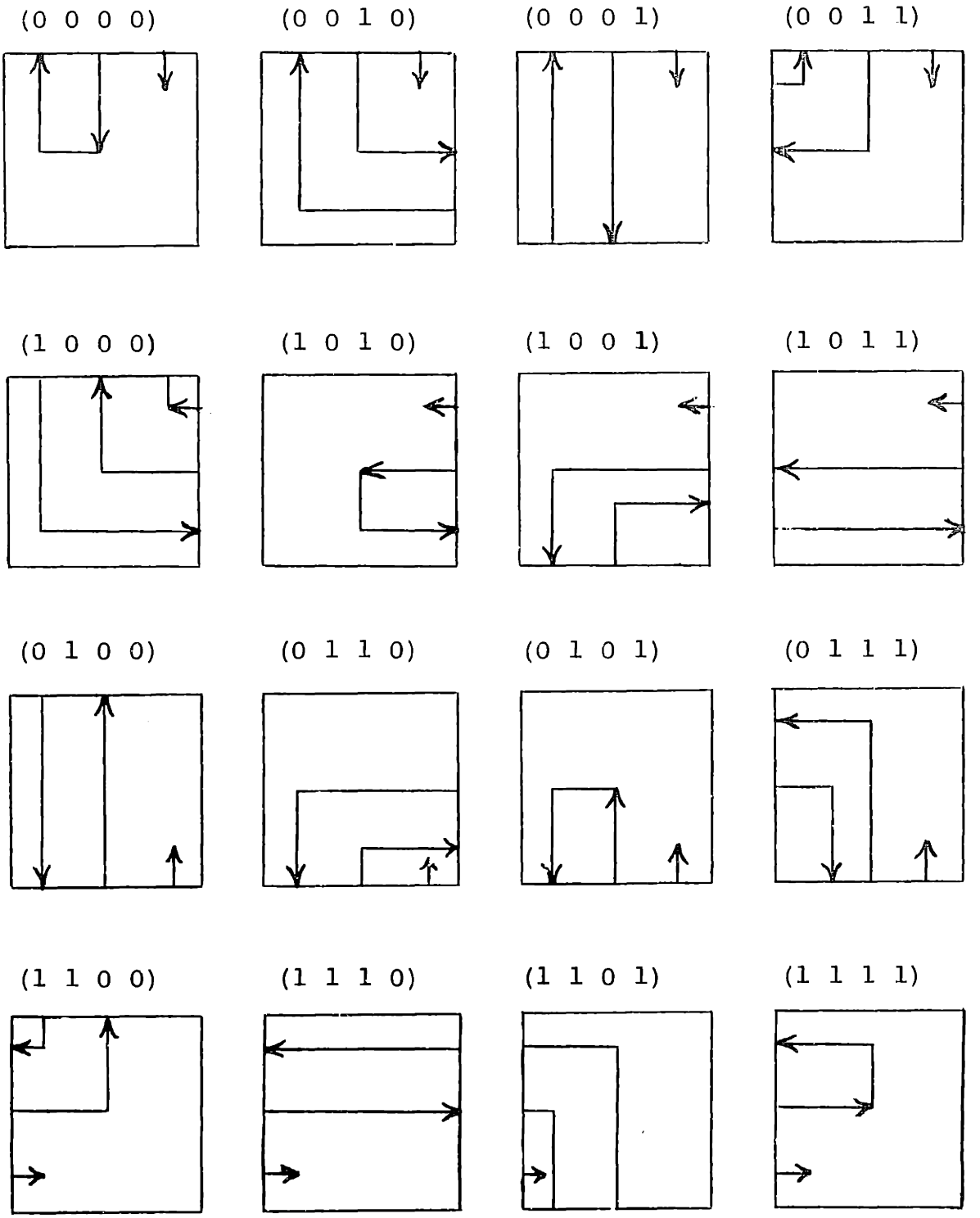


The FUNCTION STATE diagram indicates the important processing inputs and outputs for a particular function state.

The short arrow in the ABBREVIATION diagram indicates the active Klock input. Its side-set is nearest the base of a shift-register arm.

Fig. 3.12 Shift-register Cell's Function States

(Shown for all values of (IN0 IN1 OUT0 OUT1))



processing outputs in various function states. It's apparent that the processing outputs depend only on the function-specification state bits, the processing state (shift-register A and shift-register B), and the processing inputs. Shift-registers A and B are of arbitrary length, with the particular practical length chosen by integration-level considerations discussed later. A cell has one relcon neighbor when STA=1; a Klock input from a side-set E clocks N.E.IN information through shift-register A, then through shift-register B, and finally out N.E.OUT. A cell has two relcon neighbors when STA=0; while K.E.IN clocks shift-register A and shift-register B, N.E.IN flows through shift-register A and then out some side-set F, and N.F.IN flows through S.R B and then out N.E.OUT.

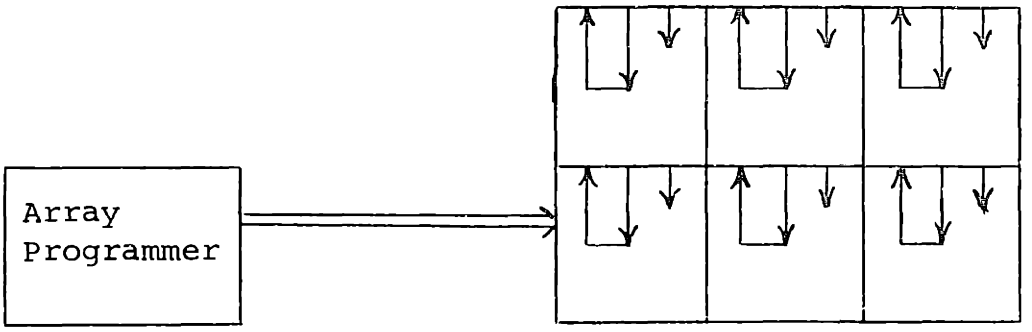
Cells are used to form shift-register arms. Information in an arm flows from the base of the arm via K and N lines to the arm's tip, turns, and flows back to the base via the R lines. The cell at the tip of the arm has STA=1. This cell acts as a loop; it forms shift-register A and shift-register B into one shift-register, with the same relcon neighbor providing K and N inputs to this shift-register, and receiving the Return output of this shift-register. All non-tip cells in the arm have STA=0. Each of these cells receives K.IN and N.IN information from a relcon neighbor nearer the arm base, and transmits R.OUT information to that cell. Each of these cells also outputs N.OUT and K.OUT to a relcon neighbor nearer the arm tip, and receives R.IN information from that cell.

A simple example illustrates how the Basic Loader and Total Retractor allow the loading of more than one shift-register into a perfect array by use of the

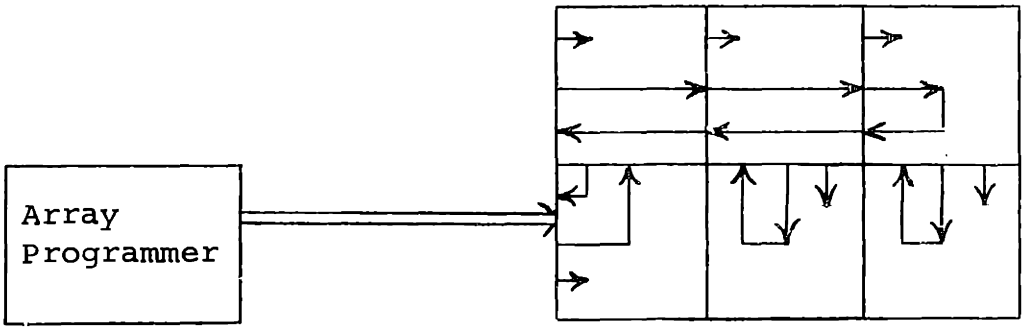
Fig. 3.13 Loading Two Shift-registers Into Perfect Array

(Extra-array processing lines are not shown.)

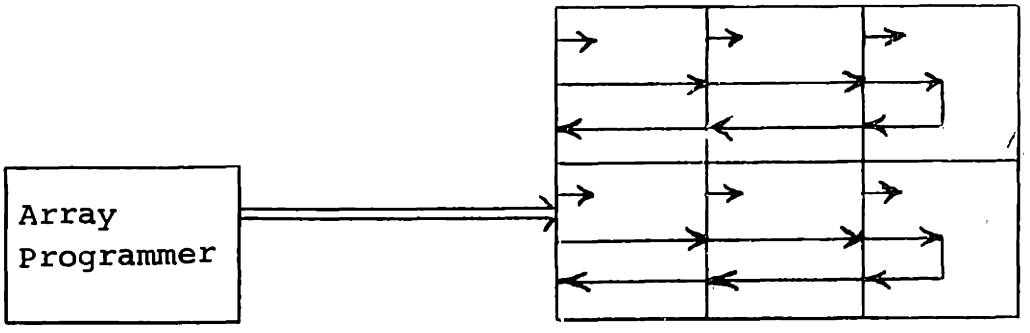
Power on; cells reset



First shift-register complete



Both shift-registers complete



loader inputs to one cell. Consider the perfect array of Figure 3.13.A, with all cells reset because power has just been turned on. The array's only connections with the outside world, other than power lines, are

1) (0 0)'s loader inputs, which connect to the Array Programmer; and

2) (0 0)'s and (0 1)'s N.L.IN, K.L.IN, and R.L.OUT lines (not shown in the figure), which will provide the inputs and outputs of two shift-registers embedded in the array.

After the Array Programmer raises S, the sequence {0,0,1,1}, {0,1,1,1}, {0,1,1,1,}, {1,1,1,1} is clocked via C and D inputs into cell (0 0). When S is lowered, the loading arm totally retracts. This leaves the array in the processing state shown in Figure 3.13.B. S is again raised, and the sequence {0,1,1,1}, {0,1,1,1}, {1,1,1,1} is clocked out. S is lowered, and the array is left in the processing state of Figure 3.13.C. The processing lines shown are made available through some type of wire. Loader lines may also be made available; so that the array may be repaired if it develops a flaw, or an embedded shift-register's length may be varied.

In estimating the time to load a cell's p-bit shift-register FL, we consider two extremes:

1) If the cell being loaded is the base of the arm, the minimum delay between C.IN transitions is tmin = 1/fmax, where fmax is the maximum clock-frequency of a shift-register.

2) If the cell being loaded is many cells away from the arm's base, tmin is determined by 2 factors:

a) After a new D.IN bit has been sent to the loaded cell, C.IN cannot go high until we're sure the D.IN bit will arrive at the loaded cell before C.IN's new transition.

b) After this C.IN transition, D.IN cannot be changed until we're sure the C.IN transition will definitely arrive at the loaded cell before the new D.IN.

Thus the time to load a cell n cells from the base of its loading arm is the greater of 2 numbers.

$$tload = p \times \max(1/fmax, n \times (dmax + cmax - dmin - cmin))$$

Here dmax is the maximum delay of a D signal through a cell, and the other d and c symbols above are similarly defined. Recall that a logic gate can have a very small delay if it's known that only one of its inputs changes frequently. Noting that the C and D delays come solely from an AND-OR function, where the ANDs have only one input that changes fast, we observe that dmax for a cell is approximately equal to dmax for a logic gate with a load of four input-loads.

In estimating the maximum frequency at which an embedded shift-register may be clocked, we make two assumptions:

1) All bits of shift-registers A and B of a particular cell are clocked simultaneously.

2) A clockpulse remains a pulse as it travels down an arm.

The rate-limiting delay then comes from the delay path schematized in Figure 3.14.

$$tmin = andmax + 2 \times andormax + s\text{-}rmax + s\text{-}rsetup$$

Andmax is the maximum delay through an AND gate, where only one input to the gate changes often. Andormax is the maximum delay through an AND-OR gate, where only one input to an AND gate changes often. S-rmax is the maximum time between a clock transition to a shift-register and the subsequent stabilization of its output at its proper value. S-rsetup is the time the D input to a shift-register must be stable before a clock transition. The Klock input to a shift-register arm must have a low enough frequency that, for the longest possible arm, a pulse remains a pulse as it travels down the arm; and two pulses are never less than tmin apart.

The method we've shown for relaying clock signals down loader and shift-register arms has two major disadvantages:

1) A clockpulse may expand or contract indefinitely if it's passed down a long enough arm. This limits the clock's frequency.

2) The frequency at which Data can be sent down the loading arm is limited by the uncertain delay involved in sending a Clock or Data signal down a long arm. Ideally a Clockpulse and its associated Data bit flow through an arm at the same speed.

Figure 3.15 shows a simple circuit which eliminates these difficulties. The circuit

Fig. 3.14 Shift-register's Rate-limiting Delay

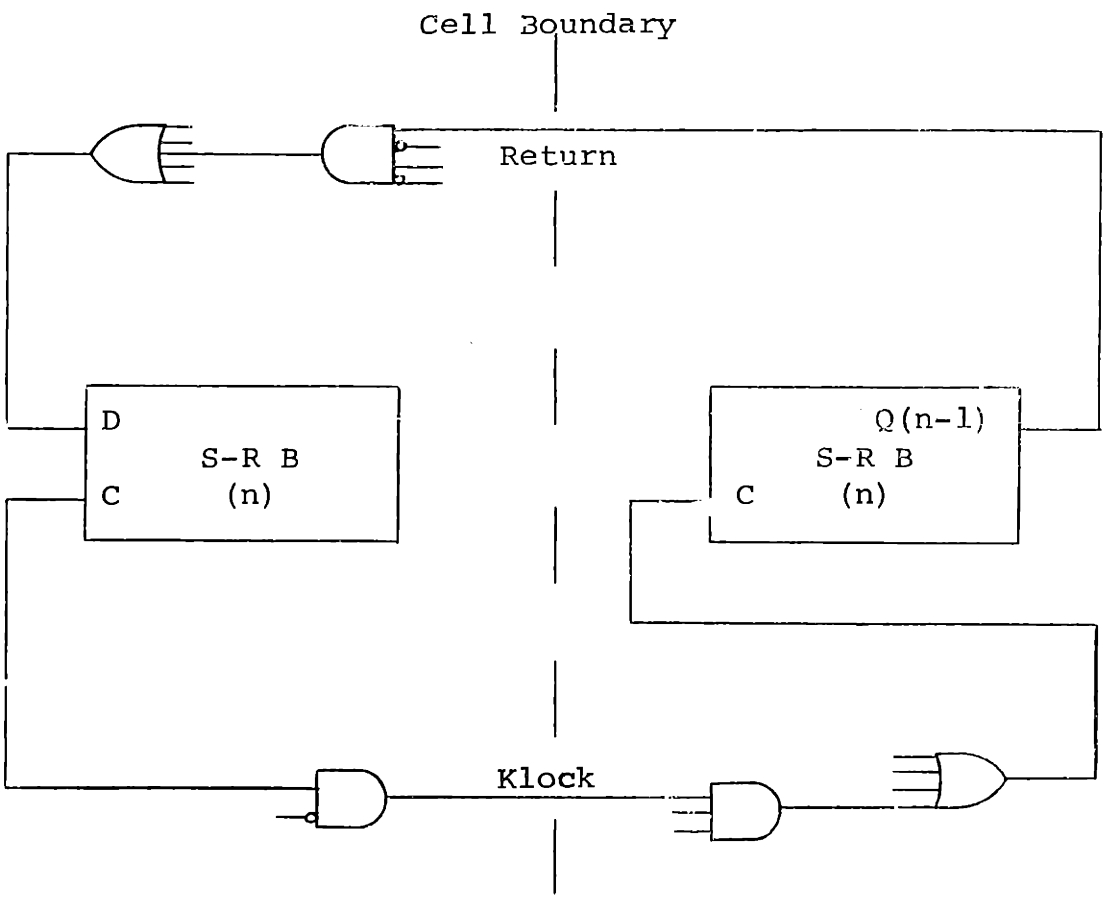
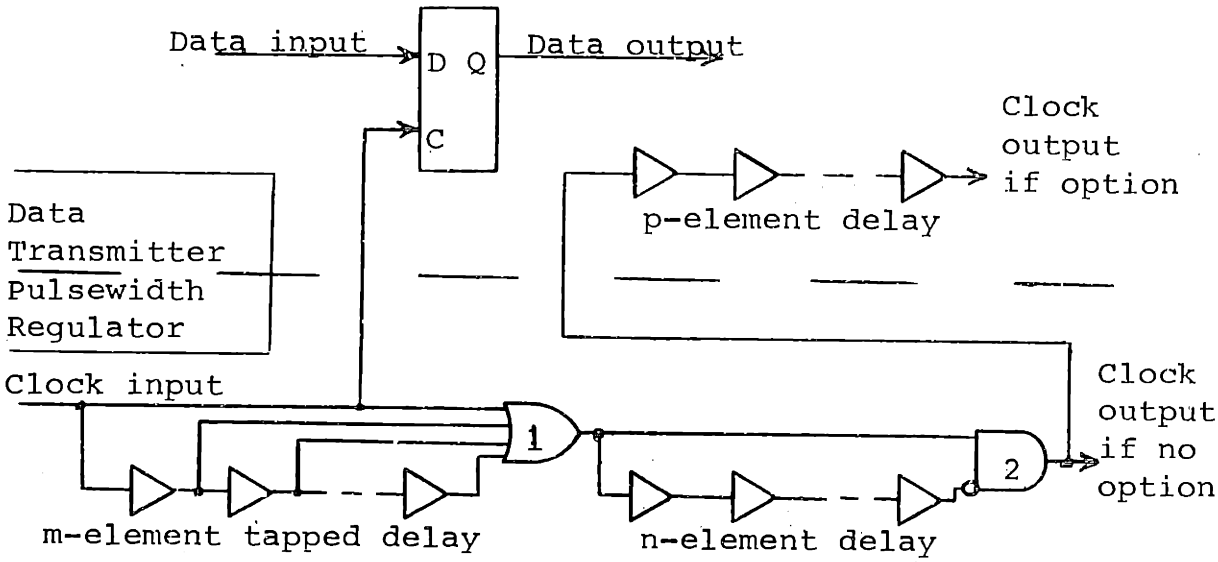


Fig. 3.15 Pulsewidth Regulator With Data Transmitter Option



The combination of the n-element delay with gate 2 constitutes a pulser responsible for outputting a pulse long enough to trigger a neighboring cell's flip-flops. The m-element tapped delay in combination with gate 1 lengthens a clock input pulse enough to assure that the pulser acts properly.

Assume that the delay of a signal-transition through any gate is D plus or minus t. Assume that the clock input to the Pulsewidth Regulator has been 0 long enough so that the outputs of all gates are 0. First consider the Pulsewidth Regulator with no option. Clock input receives a positive pulse of minimum length P sufficient to trigger any of a cell's attached memory elements. The m-element tapped delay is tapped at enough places that a clockpulse P long causes one longer pulse out gate 1. This longer pulse has a minimum width W such that

$$W \geq [P + (m + 1)(D - t) - (D + t)] = [P + mD - (m + 2)t].$$

If $W \geq n(D - t)$, the clock output is a pulse X with

$$X \geq nD - (n + 2)t.$$

Assume that the pulse at the clock output is reduced by at most R as it travels through gates to the clock input of a neighboring cell. Then the following conditions assure that the neighboring cell receives a pulse of minimum width P.

$$nD - (n + 2)t \geq P \quad \text{AND} \quad P + mD - (m + 2)t \geq n(D - t).$$

Longer clock input pulses obviously work fine.

A similar analysis calculates the maximum clock frequency.

If the Data Transmitter option is used, the clock output pulse must be delayed the right amount to assure that a Data bit and its associated clockpulse flow together from one cell to the next.

assures that a clockpulse of minimum width P which is input to a loading arm will travel down the arm with a tightly-bounded width. The Data Transmitter option assures that a Data bit and its associated clockpulse flow together at the same speed down a loading arm. Placement of the Pulsewidth Regulator before the broadcast Klock output of a shift-register cell would increase the maximum clocking frequency of a long shift-register arm. The loader's use of the Pulsewidth Regulator and Data Transmitter option would speed loading for long loading arms at the cost of slower loading for short arms and increased cell overhead.

Richard Shoup's method for forming an embedded shift-register is quite different from ours. In Shoup's method, a cell contains only 1 processing layer shift-register; function state bits control which iNput goes to the shift-register, and the output of the shift-register is broadcast to all its neighbors. Clocks to the shift-register cells come down the Data control lines, the same lines used for loading function-specification state bits. This of course means that all co-column cells must be clocked simultaneously; they can't, for instance, be used for different registers with different clock frequencies. Shoup's arrays are relatively hard to test, especially for large arrays. Testing requires "building up shift-register paths of increasing length between opposite edges of the array." (See <Shoup 73>.) Every cell is tested in all 4 directions; we'll see that this is an unnecessarily large amount of testing. The tester accesses the processing inputs and outputs of all edge cells; this requires excessive use of probes and bonding pads. Our loading method gives our shift-register arrays many advantages. We'll

see that the fact that all communication with an embedded shift-register arm is through its base also facilitates testing. The major drawback of this bi-directional capability of our shift-register cell is that it slightly reduces a shift-register's maximum clock frequency.

Section 3.3: Testing And Repair

In this section we consider the concurrent processes of testing and repair involved in embedding a shift-register arm machine in a flawed array. The shift-register cell is the one we've been considering, that of figure 3.10. We focus on growth of one arm from a base cell with loading and processing connections to an Array Programmer, but the techniques discussed are easily generalized. The Array Programmer uses a loading arm to grow longer and longer shift-register arms, like the two in Figure 3.13. The growing shift-register arm extends through the same cells as the loading arm. The arm is tested as it grows. Failure to pass a test indicates that the arm must twist through the array in a slightly different way, so that it includes only good cells. If an attempt is made to produce an arm of a certain length in a given flawed array with inputs and outputs at a given side-set, several things may happen. Such an arm may be realized, the array may be found incapable of producing such an arm, or testing may take too long.

Embedding an arm of balanced cells is particularly easy. The arm is extended cell-by-cell into an array. When an arm is in a given position, the arm is tested under the temporary assumption that its non-tip cells will remain in their current function states. The relcon neighbors of each body cell are therefore known, and information flowing in the arm to and from its base tests the cell's communication with its relcon neighbors. As long as a cell's interconnection, non-relcon neighbors aren't loaded, it's assumed that their inputs to an arm cell don't change. Consequently, it's sufficient to test an embedded arm via the inputs and

outputs at the base of the arm. The cells' balance allows an arm to move in any direction as it snakes through good cells in an array. Sometimes extension of an arm in an intended direction is prevented by a flawed cell. Then the arm is retracted, and the arm's growth proceeds in some new direction from some stump of the unsuccessfully extended arm. Since the cells in the stump of the arm stay in the same function state, they need not be re-tested. A cell is only tested in its role in an embedded shift-register arm. Thus growth of an arm through balanced cells facilitates the interwoven processes of testing and repair.

Description of the testing process is much clearer if we use an example simplified by some assumptions:

1) Good cells are only loaded under the Array Programmer's control, and not by signals caused by faulty cells. This assumption is satisfied if no faulty cell outputs a high S at the same side-set where it outputs an alternating C, since this is the only way a faulty cell can load a good cell. This assumption is also satisfied if any cell that is not loaded under the Array Programmer's control is defined as a bad cell, even if it is not the cell's fault that it is improperly loaded. Since we'd like properly formed cells to be used as good cells, we specify cell mechanisms that help guarantee that a good cell is not falsely loaded. This involves making the set of valid loading commands smaller than the set of possible loading commands, so that fault-generated commands are likely to be disobeyed.

2) A cell's performance depends only on that cell's mechanism, state, and

input signals. It does not directly depend, for instance, on the state of some other cell in the array. Like the fourth assumption, this saves testing time; it's used in most IC testing programs. The assumption is reasonable because the only lines connecting different cells are the side-set lines and the power lines. This assumption accounts for side-set lines. In some technologies, such as magnetic bubbles, coupling could not occur through cell-connecting power lines because there are no such lines. With other technologies, such as conventional semiconductor technology, it's true that such coupling could occur. However, the regularity of an array is useful in minimizing this possibility. Each cell could contain a simple regulator circuit optimized for the highly predictable characteristics of a working cell.

3) Cells that are faulty during array testing must be somewhat consistent in their faulty behavior; that is,

A) a successfully tested cell doesn't develop new faults during array testing; and

B) a processing input that doesn't alternate during the testing of a cell may not alternate during other array testing, unless the Array Programmer commands it to alternate.

Assumption 3 makes it easier to localize the cause of a test failure. Assumption 3A is reasonable because the time to test a realizable array is very short compared to its mean-time-between-failure. Assumption 3B allows a cell in an embedded arm to be tested solely for proper

communication with its relcon neighbors; it allows the Array Programmer to assume that a cell passing its tests won't misbehave during further array testing due to a previously unencountered input signal. (Most cells wouldn't misbehave anyway, since they're programmed to ignore irrelevant inputs.) A cell may have side-sets which are inaccessible to an Array Programmer, due to the cell's position near a flawed cell or at an array's edge. All the cells of figure 3.16 have at least one inaccessible side-set. Assumption 3 allows such a cell to be embedded in an arm in spite of the inaccessibility of its irrelevant side-sets. Like assumption 1, assumption 3 is valid if all the inputs and outputs of faulty cells are assumped to be stuck at some value. If assumption 3 is invalid for a particular array, the Array Programmer may become confused during testing. In this case the Array Programmer can start testing the array again. Repeated confusion indicates that faults are forming at a pathologically high rate; then the Array Programmer signals that the array should be rejected.

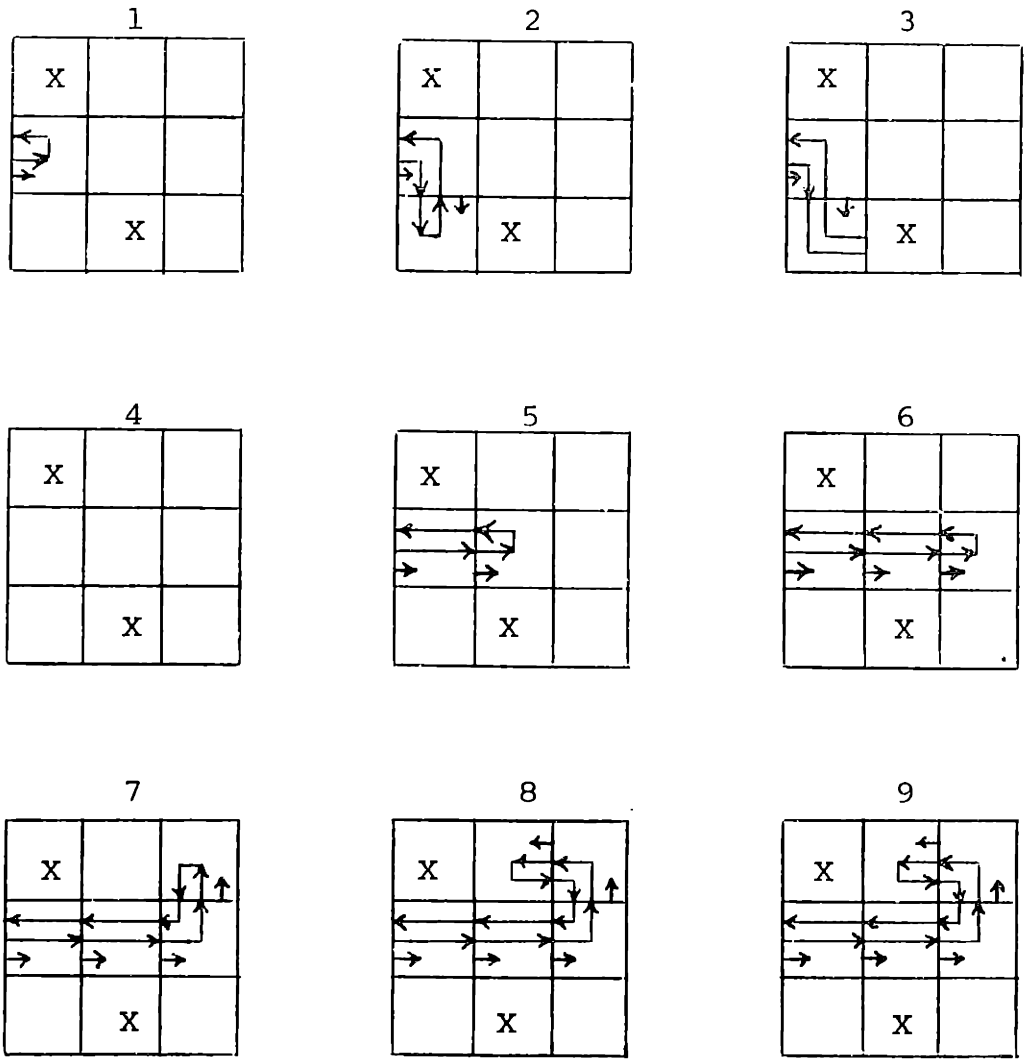
4) The behavior of certain mechanisms in a cell is independent of the state of other mechanisms. We assume that a shift-register bit works if it successfully accepts new information when the bit and its 2 neighboring bits are in any of their $2^3 = 8$ states. This assumption implies that a shift-register bit's function is independent of the state of non-adjacent bits in an array. This allows the testing of a length-n shift-register by testing its ability to shift a (10 + n)-bit sequence (0 0 0 1 0 1 1 1 0 0 ---), in which

the n bits are used to push the first ten bits through the shift-register. This common, reasonable assumption is necessary to save testing time; testing a 40-bit shift-register in every state would take a sequence of approximately 1,000,000,000,000 bits, and we expect a cell's shift-register to be considerably longer than 40 bits. The unspecified bits in the sequence above could be selected to drain maximum current from the power supply. Similarly, we assume that the processing mechanism's behavior is independent of the loader's state. This assumption saves test time.

The validity of these assumptions, which are like those made in testing conventional digital systems, can be made very likely by proper array design and layout. The ultimate test of the validity of these assumptions for a particular array is experimentation with that array.

We now consider a testing process operating under these assumptions. Consider the array of Figure 3.16, shown in successive stages of testing. The only extra-array connections are the Array Programmer's processing and loader connections to cell (0 1), which aren't shown in the figure. Here we assume the shift-register arm is to be 5 cells long; m = 5. When a new cell B is to be tested for possible addition to a shift-register arm currently extending to its tip at cell A, several things happen. Cell A is put in a state so that shift-register arm information is routed to and from B. B is put into a loop state - (0 0 0 0), (0 1 0

Fig. 3.16 Growth Of Perfect Shift-register Into Flawed Array



The connections of the Array Programmer to cell (0 1)'s left loading and processing lines are not shown.

Unmarked cells are good cells in the (0 0 0 0) state.

1), (1 0 1 0), or (1 1 1 1) -with the loop starting and ending at A's side-set shared with B. Assume N is the number of bits shifted completely through the processing shift-registers of cell B and passed down the arm for monitoring. Also assume the Array Programmer knows the contents of all the A shift-registers in the arm up through cell A. (The Array Programmer should know this; it's loaded these registers.) Testing cell B in an arm of length m then requires (N + alength + m x blength) Klock inputs to the arm, where alength and blength are the lengths of shift-register A and shift-register B. Passing the test means that the shift-register arm works properly; a new tip has been properly added. If cell B is the last, "m"th cell of the arm, the Array Programmer is then satisfied that a shift-register arm has been properly formed in the array. (See stages 8 and 9 in the figure.) The Array Programmer doesn't care whether the cells of the arm could have been loaded from other directions, or would have worked in other function states. It doesn't care if some cells of the array haven't been tested at all. (See cell (0 2) in the figure.) The Array Programmer simply cares that its objective has been realized. This pragmatic approach allows substantial reduction of testing time.

If cell B is meant to be part of a longer arm, it must be connected to an interconnection neighbor cell, other than A, just as A was connected to B. The testing of this new, longer arm then proceeds as above. Growth is a recursive procedure.

Failure of the arm after its extension from cell A to cell B indicates that growth from cell A to cell B is impossible. A may be loading B incorrectly, B may

be flawed, A or B may be outputting Klock information to a neighbor elsewhere in the arm, etc. The Array Programmer doesn't worry about the specific nature of the problem. It simply uses one of two reasonable flaw-models. Cell B may be judged as a flawed cell never to be tested again, as in the example. This simplification might be appropriate if the area of shift-registers A and B dominated the area of a cell; failure was probably due to a failure in this area. A second alternative is to just consider the boundary between cells A and B impassable in the attempted direction. Cell B might be approached later from one of its other neighbors.

If cell B can't be approached from cell A, some new arm-path is tried if there is still one to be tried. Cell A considers touching its neighbor cells in some established order. When a neighbor is considered for touching, the touch is attempted if the cell exists (isn't out-of bounds), isn't known to be unloadable from A, and isn't already part of an arm. Furthermore, extension of the arm through that cell must, at least potentially, eventually yield an arm of the desired length. This last provision explains why no attempt is made to include cell (2 0) in the arm in the example; at best a length-4 arm would result.

If all A's neighbors have been rejected, the arm is forced to try some new path that includes all arm cells up to A. In the example, (0 0) of stage 3 is cell A. Since there's no cell compatible with the existing arm that can be loaded from (0 0), the arm is retracted to cell (0 1), where new paths are considered.

A program simulates the method described above for loading a shift-

register arm into a flawed, rectangular array. The simplest fault model is used; a cell is either perfect or hopelessly flawed. A program forms a flaw pattern of specified dimensions with randomly sprinkled flaws. Another program tries to realize the longest arm possible in the flawed array, growing from a specified, good base cell. A time limit is used because the program would eventually consider all possible arm paths extending from the base cell.

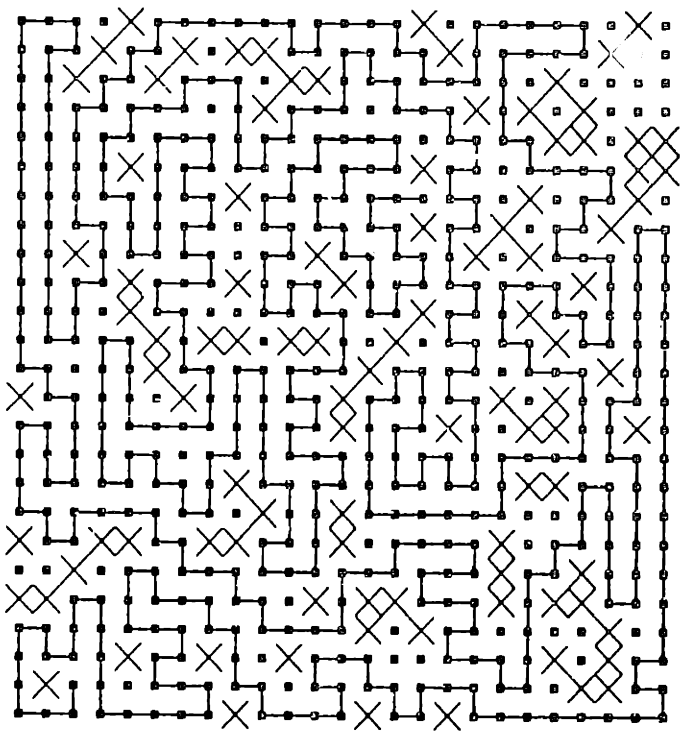
The repair program is short and simple. When an arm has grown to a certain tip, it tries to extend itself toward the nearest array edge. Thus an arm spirals toward the center of an array in a perfect array. If no improvement in the maximum discovered arm is made in one-fourth of the time limit, the program looks at adjacent cells that are not included in this longest arm and are not known to be flawed. The program tries simple jogging of the arm to include these cells. The program returns with a picture of the resulting arm in the flawed array, and some statistics concerning the arm growth.

Figures 3.17 and 3.18 show arms snaking through two different 25 x 25 arrays. Statistics for these and other, similar experiments, appear in table 3.1. Figure 3.19 shows graphs derived from table 3.1.

The experiments suggest several conclusions:

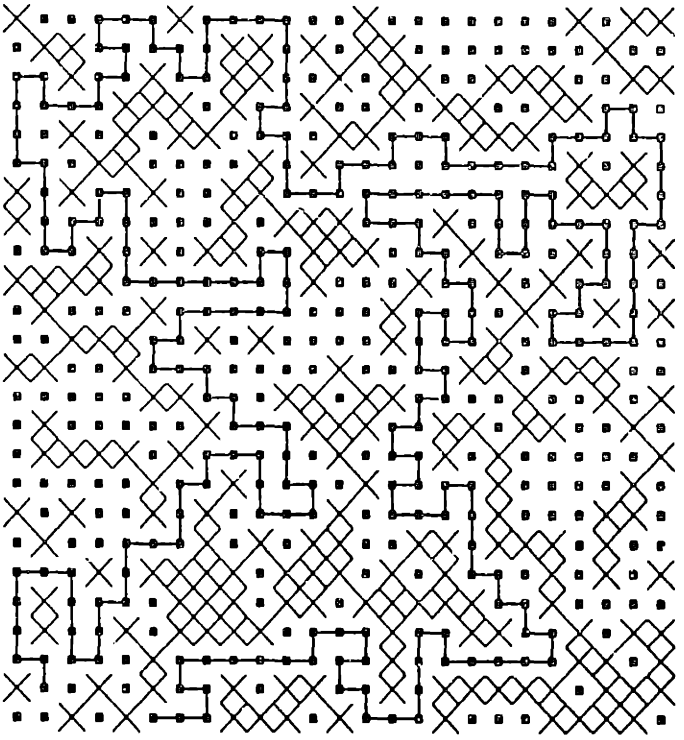
1) For %flawed under about 25, %oftotal drops about 2.2% when %flawed increases 1%. This is nearly independent of the size of the array, with larger arrays doing slightly better. Repair efficiency also drops steadily. For instance, for the unstarred 400-cell array in table

Fig. 3.17 Result Of An Arm-growth Experiment



The relcon network above shows the path of an arm after an arm-growth experiment. One can follow the arm's path from its base, at (2 1), to its tip, at (15 18). There are 625 cells, 100 flawed cells, and 463 arm cells. We were able to repair the array to embed an arm with 495 cells. This suggests that our program's repair efficiency can be improved.

Fig. 3.18 Result Of An Arm-growth Experiment



The base cell is (1 1). There are 625 cells, 225 flawed cells, and 216 arm cells.

Table 3.1 Results Of Arm-growth Experiments
(1st of 2 pages)

Key:

%flawed - flawed cells as percent of all cells
Cells - total cells in square array
#flaws - total flawed cells in array
max-arm - the longest arm our program grew
%oftotal - cells in longest arm as percent of all cells
timelim - the time limit, in seconds
Time - the time the program ran
%oftimelim - time as percent of timelim
* - For two starred (or unstarred) arrays of the same size, one set of flaw coordinates is a subset of the other.

Table:

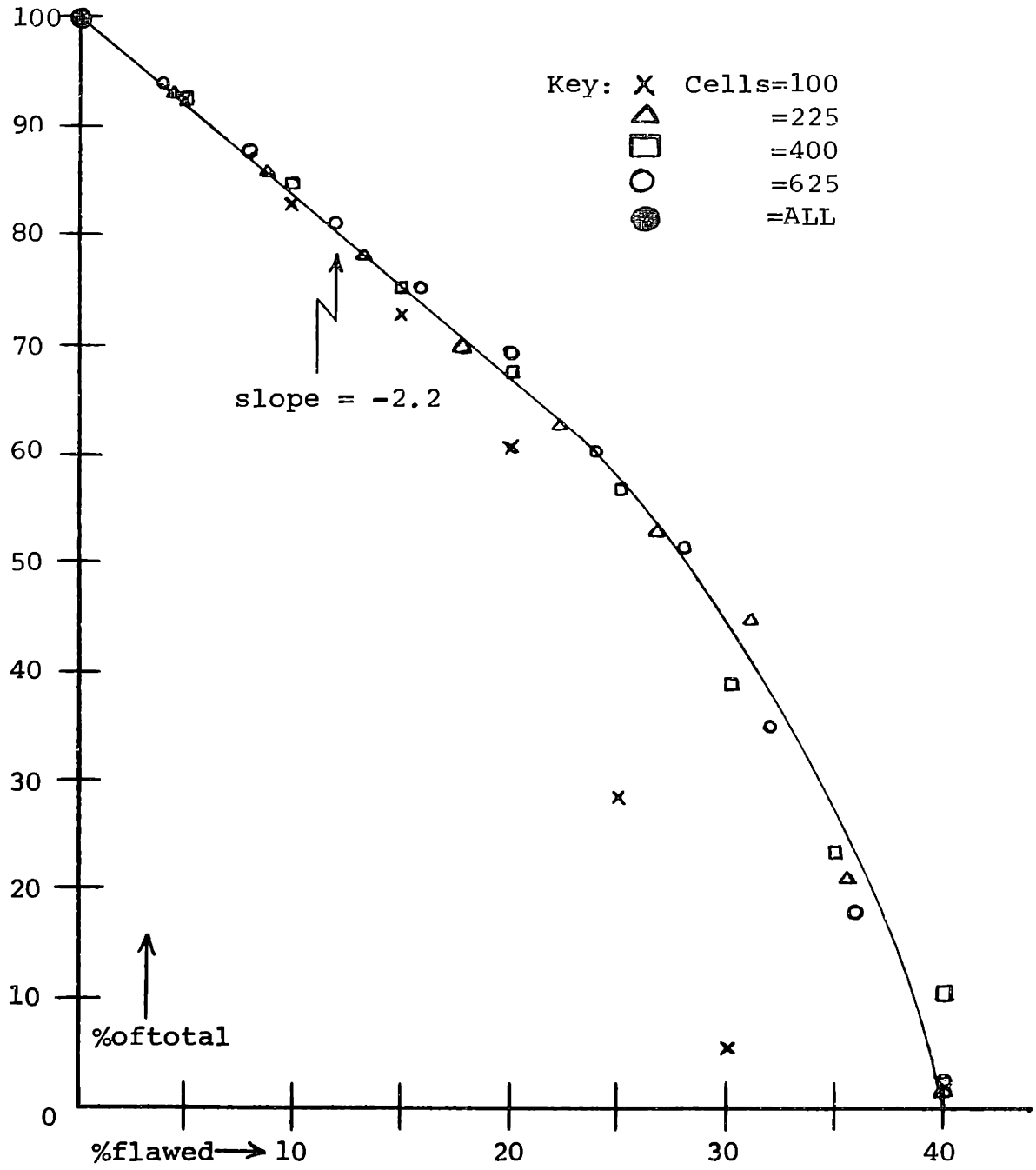
%flawed	Cells	#flaws	max-arm	%oftotal	timelim	Time	%oftimelim
0	100	0	100	100	100	1	1
0	225	0	225	100	225	3	1
0	400	0	400	100	400	5	1
0	625	0	625	100	625	8	1
4	625	25	590	94	625	206	33
* 4	625	25	586	94	625	186	30
4.44	225	10	208	92	225	60	27
* 4.44	225	10	211	94	225	74	33
5	100	5	93	93	100	27	27
* 5	100	5	92	92	100	27	27
5	400	20	367	92	400	141	35
* 5	400	20	370	93	400	106	27
8	625	50	551	88	625	216	35
* 8	625	50	548	88	625	175	28
8.89	225	20	187	83	225	63	28
* 8.89	225	20	199	89	225	65	29
10	100	10	85	85	100	26	26
*10	100	10	81	81	100	27	27
10	400	40	335	84	400	109	27
*10	400	40	344	86	400	136	34
12	625	75	505	81	625	265	42
*12	625	75	506	81	625	189	30
13.33	225	30	168	75	225	73	33
*13.33	225	30	182	81	225	72	32
15	100	15	72	72	100	27	27
*15	100	15	74	74	100	29	29
15	400	60	301	75	400	112	28
*15	400	60	304	76	400	160	40

Table 3.1 Results Of Arm-growth Experiments
(2nd of 2 pages)

%flawed	Cells	#flaws	max-arm	%oftotal	timelim	Time	%oftimelim
16	625	100	463	74	625	241	39
*16	625	100	476	76	625	181	29
17.78	225	40	152	68	225	69	31
*17.78	225	40	163	72	225	69	31
20	100	20	60	60	100	27	27
*20	100	20	61	61	100	27	27
20	400	80	265	63	400	292	73
*20	400	80	287	72	400	114	28
20	625	125	440	70	625	220	35
*20	625	125	434	69	625	187	30
22.22	225	50	144	64	225	64	29
*22.22	225	50	140	62	225	106	47
24	625	150	366	59	625	264	42
*24	625	150	374	60	625	341	55
25	100	25	53	53	100	27	27
*25	100	25	4	4	100	-	-
25	400	100	225	56	400	119	30
*25	400	100	233	58	400	158	39
26.67	225	60	125	56	225	73	32
*26.67	225	60	113	50	225	114	51
28	625	175	346	55	625	332	53
*28	625	175	300	48	625	317	51
30	100	30	11	11	100	-	-
30	400	120	158	40	400	130	33
*30	400	120	150	38	400	355	89
31.11	225	70	115	51	225	181	80
*31.11	225	70	87	39	225	117	52
32	625	200	194	31	625	344	55
*32	625	200	246	39	625	429	69
35	400	140	108	27	400	235	59
*35	400	140	79	20	400	154	39
35.56	225	80	93	41	225	101	45
*35.56	225	80	2	1	225	-	-
36	625	225	7	1	625	-	-
*36	625	225	216	35	625	483	77
40	225	90	7	3	225	-	-
40	400	160	78	20	400	115	29
*40	400	160	4	1	400	-	-
*40	625	250	32	5	625	165	26
45	400	180	58	15	400	275	69
50	400	200	3	1	400	-	-

Fig. 3.19 Graphs For Experiments Embedding Balanced Arms

%oftotal is averaged for a given value of Cells and %flawed.



3.1, the repair efficiency drops from .98 at %flawed = 5 to .415 at %flawed = 35.

2) As %flawed increases, a cutoff point is reached where %oftotal drops precipitously. In our experiments, this occured for %flawed between 25 and 45. This cutoff occurs when an array is so flawed that the arm is trapped. Very small arrays, such as the 100-cell arrays in our experiments, tend to have lower repair efficiencies and lower cutoff points; because a higher percentage of cells are edge cells. An edge is a barrier that restricts the growth of an arm.

3) The time taken to embed an arm varies widely for a fixed %flawed. It is roughly proportional to the number of cells in an array, and tends to increase as %flawed increases. When the cutoff point is reached, the time to embed an arm plummets. This is an example where testing and repair time is far from growing astronomically with an array'ssize, even though very few input leads connect the Array Programmer and the array.

4) If the active area of a cell is fixed, statistical considerations state that %flawed varies less as slice size (and number of cells) increases. This fact, the near-independence of %oftotal on the number of cells in an array, the proportionality of the time to test and repair an array to its number of cells, and the desirability of large memories in one integrated circuit package all argue for fabrication of the largest possible slices.

5) How large should cells on a large slice be? Assume that the dominant consideration is the number of bits in the largest embedded shift-register arm. The total number of bits in an arm embedded on a slice is proportional to the product of two factors:

a) The fraction Y_E of total cells embedded in the arm. Our experiments show that for a given cell yield $Y_C > 3/4$, Y_E is approximately $1 - 2.2(1 - Y_C)$. Y_C is a technology-dependent function of defect density and cell area.

b) The fraction of a cell's area containing processing shift-registers. If a cell has P area devoted to processing shift-registers and V area devoted to other circuitry, this fraction is $P/(P + V)$.

We can express this product as a function of P and technology-related parameters. Finding the maximum of this function via differentiation tells us the value of P that yields the highest expected number of bits in a shift-register arm. At one extreme, a large slice has nothing but overhead circuitry. At the other extreme, it has one large, flawed cell. Note that a minimum condition is that a cell be small enough to make %flawed below the cutoff point. This condition is now met in most technologies.

Though the repair simulation program is simple, its performance is

encouraging. There are several ways it can be improved. In a production line using large slices, the program would know an expected, minimum size of an embedded arm for a slice of a given size. To save computer time, it could be satisfied when it attained that minimum-sized arm, or one slightly larger. At this point, use of much more computer time to maximize the arm would probably not be worth the cost. Our simulation ran in compiled Lisp, and no effort was made to improve speed. A production-oriented repair program would be carefully written in assembly language. More computer time could be used to improve repair efficiency.

We've repaired figure 3.17's array to realize an arm 495 cells long. Our ability to improve the repair efficiency from 88% to 94% suggests our repair program's performance can also improve. A more complicated and/or heuristic program could improve repair efficiency. A simple extension of our program would be more sophisticated about jogging an arm to include unused, good cells. Even the current jogging procedure could be called several times, instead of only once at the end of the main arm-growth procedure.

Now consider our assumption that no faulty cell outputs a high S at the same side-set where it outputs an alternating C. If a faulty cell outputs only constant signals, this assumption is obviously valid. However, this assumption is not valid if our assumptions are relaxed to say that all FAULTY outputs of a cell are stuck outputs. In particular, it is not valid if a cell A's only fault is a high S.OUT -- say S.L.OUT. In this case incoming loading signals may be routed to the

falsely-touched neighbor B at the same time they are routed to the appropriate Array Programmer-intended cell. In this case we call cell A a *branch cell*, and cell B the *branch arm's base*. This branching is particularly vexing because its effects might not be felt until much later in the testing. Consider the array of figure 3.20, whose only fault is a high S.R.In to cell (2 2). That is, (1 2) is a branch cell. In such an array the indicated state could occur. The Array Programmer would only know of the existence of the intended arm. When the intended arm tried to touch (0 0), the branch arm would touch (1 0), causing the subsequent test failure of the extended version of the intended arm that included (0 0). This failure could be caused by a faulty (0 0) cell, but in this case it wouldn't have been. Even if the Array Programmer knew the failure was due to a loading branch, it wouldn't know where the branch occurred; here cells (1 0), (1 1), (1 2), or (1 3) could have been branch cells. The problem is heightened by the fact that total retraction of the intended arm via lowering (1 3)'s S.U.In does not affect the branch arm. Indeed it may grow further if more loading information is clocked into (1 2). Happily, a working cell's Incremental and Total Retractor circuitry implies that attempting to load a good cell in a branch arm results in the freeing of all the cells from the loaded cell to the tip cell in the branch arm.

There is a wide range of possible approaches to the loading branch. On one extreme, the Array Programmer could assume that this branching problem does not exist. If this assumption is invalid for a particular array, the Array Programmer may find itself hopelessly confused. Then it quits its testing attempts, and signals

Fig. 3.20 Growth Of A Branch Arm

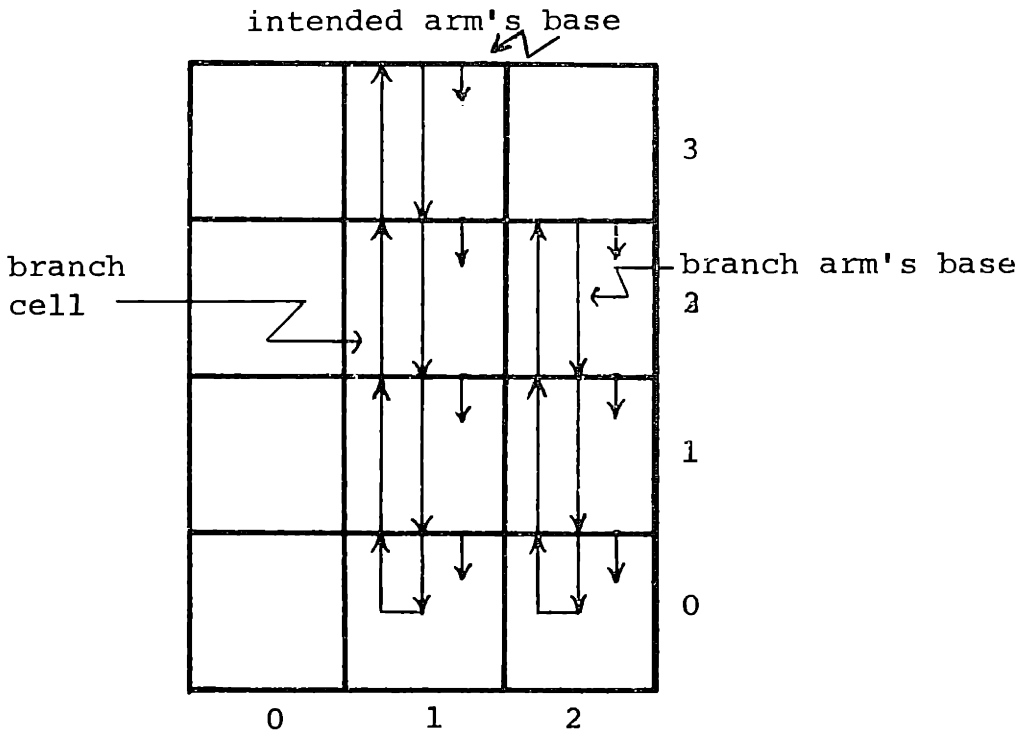
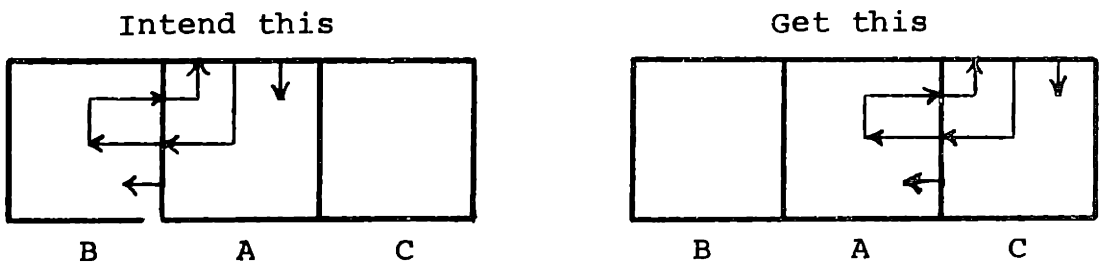


Fig. 3.21 Branch Arm Touching Intended Arm



that the total array should be discarded. This is a fast approach that might be reasonable if the probability of a branch cell was low; for instance, if cells had many elements or arrays had few cells.

An array can be successfully loaded even if it has a branch cell, if one is willing to accept the extra testing involved. By our assumption that all faulty outputs are stuck at some value, a branch cell can only transmit loading information to a branch base if the branch cell's C output to the branch base works. Then D of that side-set is either:

1) an alternating signal transferred by the branch cell, as in the example above; or

2) a fixed D signal, which causes the branch base to be continually reset due to its being programmed into the STA=1 state.

(2) is no problem; it's (1) we're considering.

The Array Programmer can use several facts to generate a list of possible branch cells. When (1) holds, some of the tip end of the branch arm is a translated version of the intended arm. This is true because the branch base receives the same C and D information that the branch cell receives. The Array Programmer can use this information, and its knowledge of the position of the intended arm, to generate a list of possible branch cells. Knowledge of which cell of the intended arm failed helps reduce the size of this list. This knowledge may come from noting that all cells of an intended arm from its base through some cell C properly transmitted their shift-register B; the cell touched by the branch arm

touched cell C or the cell to the tip side of cell C.

Assume cell A tried to touch cell B, and the subsequent test failed. The Array Programmer might suspect a branch cell if not even shift-register B of cell A ((1 3) above) outputs properly during the test. A neighboring cell C, part of a branch arm, may have touched cell A immediately after the simultaneous loading of cells A and C, thereby displacing arm A's tip to cell A. Cell A would then be loaded with information intended for cell B (see figure 3.21).

When the Array Programmer suspects a test failure occurred because of a branch cell, it retracts the intended arm. The Array Programmer then regrows the arm through cell A, and tries to terminate the arm with a loop at cell C, the possible branch cell closest to the former intended arm's tip. This new arm is tested. An unsuccessful test suggests that the potential branch arm's base, cell C, was the branch arm's base; cell A, the branch cell, is marked as totally flawed. If the test is successful, and there are other potential branch cells closer to the base of the intended arm, these cells are tested in the same way cell A was. That is, arm A is retracted and then hooked into a potential branch base cell. This process repeats until all potential branch cells are tested, or a branch cell is found. If all tests are successful, there was no branch cell. Testing and repair continue as if cell A was merely unable to include cell B in the shift-register arm. In any event, this process assures that no branch arm remains to clutter up the array. (If such an arm never affects intended arm growth, we don't care about it anyway.)

Note how the Incremental Retractor circuitry helps in the example

above. It allows the intended arm to touch and load a cell that's been part of a branch arm. (Of course, loading must be slow enough to make negligible the slightly different delays of C and D information traveling through arms A and B.) Furthermore, it allows quick incremental retraction of arm A when a potential branch cell is found to be good.

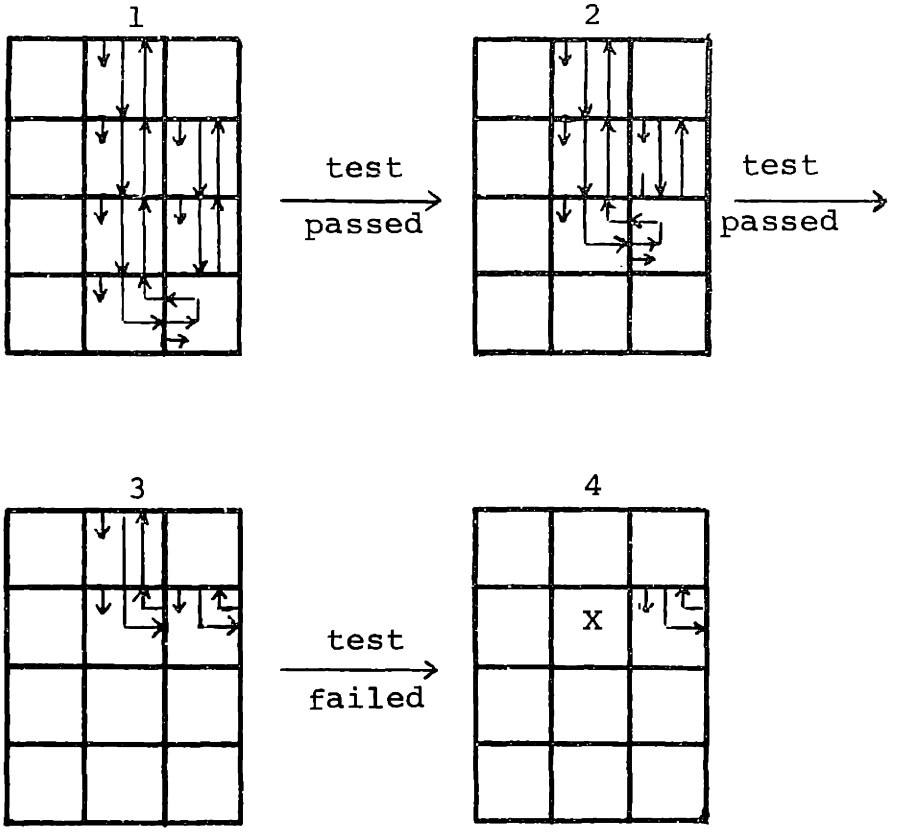
These branch location steps are illustrated in figure 3.22 for the array of figure 3.20. incremental retraction is used between all the stages shown.

In another type of possible branching, a branch cell transmits high S.OUTs to more than one cell AFTER the branch cell has been loaded. This type of branching, which is much less likely than the other, can be handled in a very similar way.

Of course, various steps can be taken to reduce the probability of a branch cell. Instead of one S line for selection, a cell could have a larger set of such lines. Only the proper combination of inputs to these lines would cause a cell to accept loading information. This approach could make chance selection, and consequent branching, arbitrarily unlikely by sufficiently increasing the number of selection lines.

The Array Programmer could send to a cell loading information stating loading-input-direction, which the cell would compare to its Select inputs to decide whether to accept a command. This technique would also help reduce the effects of a branch cell by reducing the ratio of valid loading commands to total loading commands. These techniques, and others like them, would only be employed after

Fig. 3.22 Location Of A Branch Cell



Arm growth proceeds through cell (1 3).

a more thorough analysis of the probability of a branch cell for a specific cell implemented in a specific technology.

In loading more than one shift-register arm into an array, one must worry that a branch arm will destroy a shift-register arm that has already been formed and tested. If this possibility is sufficiently probable, it's a good idea to continue testing a completed shift-register arm while a new arm is being formed. Effects of a branch arm can then be detected and countered before extensive damage to the completed arm machine occurs. Besides monitoring the integrity of the completed arm, this approach helps limit the confusion caused by a branch arm.

In limiting our consideration of possible failure modes to those above, we are encouraged by a quote from <Von Neumann 66>:

> "The axiomatization of automata for the completely defined situation is a very nice exercise for one who faces the problem for the first time, but everybody who has had experience with it knows that it's only a very preliminary stage of the problem." . . .
>
> "There can be no question of eliminating failures or of completely paralyzing the effects of failures. All we can do is to try to arrange an automaton so that in the vast majority of failures it can continue to operate."

Our discussion of testing and repair shows we can achieve Von Neumann's goal simply and efficiently by incorporating our loading, testing, and repair mechanisms into a cellular array. The major limitation of our discussion - the uncertainty of an appropriate flaw model - will be reduced when a particular technology and cell layout are considered for the shift-register array.

Section 3.4: Production And Marketing Considerations

In previous sections we've considered the basic question of array architecture, testing, and arm growth. In this section we consider less fundamental, but important, points relating to specifics of production and marketing.

Once an arm is extended slightly into an array, the arm has many alternate paths; the curves of figure 3.19 then apply. However, it's critical that the Array Programmer be able to penetrate the array via an arm base cell. If the Array Programmer can only access one such cell in a flawed array, there's a probability pflaw that that cell will be flawed, and the array will consequently be unloadable. One way to ameliorate this situation is to fabricate an array with Array Programmer-accessible bond pads to more than one cell - each a potential arm base. If there are m such cells, the probability that no arm can be extended into the array diminishes to about $pflaw^m$. Quick tests would establish which base cells worked. The Array Programmer would then use one or all of these cells as base cells for testing and arm growth. The base cells should probably be away from the edge of the array. One reason is that the edge is more subject to flaws. A second reason is that there are more directions for arm growth away from the edge. Another ameliorating solution would put a circuit on a slice that accepted extra-array inputs which told it which of several cell edges to logically connect to the slice's leads. For instance, one "cell" would replace shift-registers A and B of a cell by wires. This non-cellular part of a slice would be less likely to be flawed than a cell.

Another important question relates to array size. How big should an array be? We know that all the procedures described so far work for arbitrarily large arrays. We've also seen many arguments for large arrays. One constraint on the size of arrays is manufacturing capability, which is geared toward dicing a wafer of maximum 3" diameter into much smaller chips. The current 100% yield approach has limited development of support machinery and techniques for the realization of very large ICs. However, Texas Instruments did use a 3/2" diameter slice for discretionary wiring (see <Spandorfer 68>). We've also heard that Hughes developed a 50-watt package for a 3" slice as part of the Navy's All Applications Digital Computer program; unfortunately, we haven't learned any details about this yet. While many of TI's and Hughes' techniques for mounting, packaging, cooling, etc. can probably be carried over to large cellular arrays, that process may demand considerable investment. However, that process will inevitably occur, spurred by improvements in IC yields. We are not even close to a fundamental limit here.

For technologies that require power lines connecting many cells, increases in array size increase the probability of array-destroying power problems. The probability of a power bus being open-circuited can be made very small by making the bus wide. Layout care can lower the chance of shorts between a power bus and a signal line; most such shorts would probably not be catastrophic anyway. Nevertheless very large arrays should perhaps include protection devices in each cell or block of cells. This circuitry could cut a shorted,

or even overheated, cell off from its power source, before the malfunction blew the power line's fuse or sucked down the power line. The protection devices could be a fuse, or could be semiconductor circuitry, such as common transistor-SCR protection circuitry.

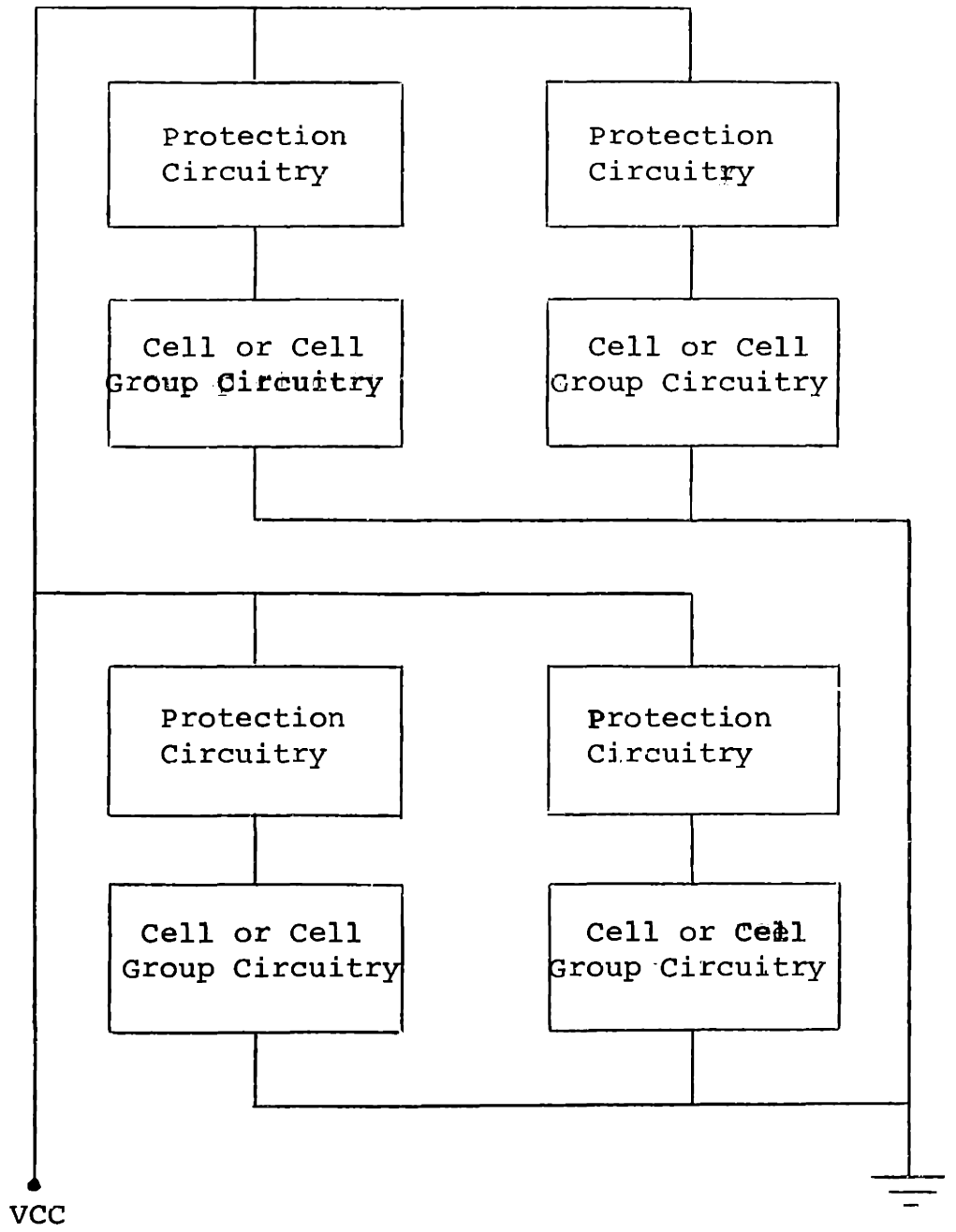
In any case, the well-defined nature of the protection circuitry's expected load would enable it to be very simple. Figure 3.23 schematizes a possible layout for power lines and protection circuits.

Another power-handling approach would make a cell's supply of power controllable by the cell's neighbors. For instance, any of a cell's neighbors could command that the cell's power supply be switched on or off. This could save power in an array, and reduce the danger of faulty cells, by channeling power only to the cells in an embedded machine. Indeed a "power arm" could be "grown" in parallel with a processing arm into an initially quiescent array of cells.

Another question relates to the size of shift-registers A and B. Having shift-register B longer than 1 bit helps in the monitoring of arm growth; if each shift-register B in an arm contains a known pattern of 0s and 1s, the Array Programmer can monitor the position of a faulty cell by noting the location of faulty shift-register B output. On the other hand, a longer shift-register B demands a corresponding longer time to test an arm. Consequently a good length for shift-register B is 2 bits. Shift-register A should probably be a length consistent with maximum expected number of bits in a shift-register arm.

An array yielding a maximum shift-register arm of a certain length can

Fig. 3.23 Possible Layout Of Power Lines And Circuitry



Communication lines between cell groups are not shown.

be used to provide arms shorter than that length. This means an IC producer could customize the same array to various customer needs. An unusually flawed array could provide a small shift-register, and its package could be marked accordingly. Customers could even be given an IC with a variable-length shift-register whose length was controlled via a side-set's loader inputs.

If function-specification state bits are nonvolatile, a shift-register arm can be loaded into an array before it's shipped to a customer. The customer has the option of access to loading lines, which allow him to re-program or repair an array.

If the function-specification state bits are volatile, there are several customer-manufacturer interface options:

1) If a customer has a computer or other appropriate digital machine, he has the capability for testing and programming an array. He can use these capabilities, and a manufacturer-supplied program, on untested or slightly tested (e.g., for functioning arm base cells) arrays.

2) The customer can receive a pre-tested array and a description of the loading sequence required to form a specified arm in the array. This description could be in some non-volatile form, such as read-only memory, paper-tape, or paper. Loading an already-tested array is as easy as loading a shift-register. Power is turned on, an S line is raised, and [4 x (number of cells to be loaded)] bits are clocked via C and D lines into the array.

3) A communication link, terminated by logic-interface machines on each end, could connect the manufacturer and customer. (The link might be a phone line or cable.) This link could be used for loading, and even testing and repairing, of a customer's machine by a manufacturer's or system house's computer.

4) An array requiring very low power (such as a CMOS array) could be shipped around with a battery-supply.

In any event, a volatile array must be backed up, either by a machine capable of re-loading or by a power-supply insuring preservation of the function state of the array.

It's obvious that the techniques we've described for the shift-register arm machine apply to any arm machine. Arm machine realizations are appropriate to many machines which are realized as a chain of modules, with each module communicating with at most two other modules, and only the modules at the end of the chain directly connected to the machine's inputs and outputs. Many one-dimensional cellular arrays have this characteristic, so they could be appropriately realized as arm machines in a flawed checkerboard array. The techniques for arm machines easily generalize to the high-relcon and tree machines discussed in the next two chapters.

CHAPTER 4: HIGH-RELCON MACHINES

Section 4.0: Introduction

This chapter discusses arrays embedding high-relcon machines. High-relcon machines have fewer restrictions on communication between their essential cells than arm and tree machines. In an arm machine, no cell may have more than two essential neighbors. In a tree machine, only one cell may actively output information at a given time. All the essential cells in a high-relcon machine may have four neighbors, and all essential cells may be actively communicating different information at the same time. High-relcon machines may therefore have speed and flexibility advantages. However, high-relcon machines are harder to test and repair because a cell may have up to four essential neighbors, and because essential neighbors in one high-relcon machine must be essential neighbors in all equivalent embedded machines. Powerful mechanisms - the loader, and balanced processing transmission states - allow test and repair of arrays embedding high-relcon machines. The description of a machine as an essential network facilitates repair by abstractly describing the machine in a repair-oriented way.

High-relcon machines are conducive to a sequence in which the array is tested, a plan for repairing the array is developed, and the array is repaired through proper loading of good cells. This contrasts to the interwoven processes of testing and repair appropriate to arm and tree machines. However, this chapter's methods still use a loading arm for loading cells during testing and

subsequent repair of an array. Transmission links form test links for testing an array. These same transmission links may wire together essential neighbors in a machine embedded in a flawed array. We detail the test and repair procedures that use these function states. Experiments with repair procedures we've written help us compare repair difficulties for arm and high-relcon machines, and suggest ways to improve our repair procedures.

Application areas most appropriate to high-relcon machines are considered. We present a simple cell, *General*, which enables realization of the benefits of high-relcon machines. General may be used to realize highly parallel, arbitrary sequential machines, within limits set only by the size of a General array, its number of input-output leads, and the speed of its components. General embodies the mechanisms we use to test, load, and repair high-relcon machines. A General array may embed a universal computer-constructor-repairer that uses the test and repair procedures we describe. General's loading mechanism may be controlled by an extra-array Array Programmer. Moreover, a machine embedded in a General array may be an Array Programmer; it can control the loading mechanism of cells in its environment via a function state that transmits processing inputs to one side's loader outputs. This enables a machine embedded in a General array to test, manipulate, and repair its cellular environment.

For specificity, we begin by detailing the General cell. Then we consider a general testing and repair approach for embedding high-relcon machines, and compare this approach to the one used for arm machines. We discuss

realization issues peculiar to high-relcon machines. A comparison of the properties of high-relcon machines to the properties of arm, tree, and non-array machines reveals applications most suited to high-relcon arrays.

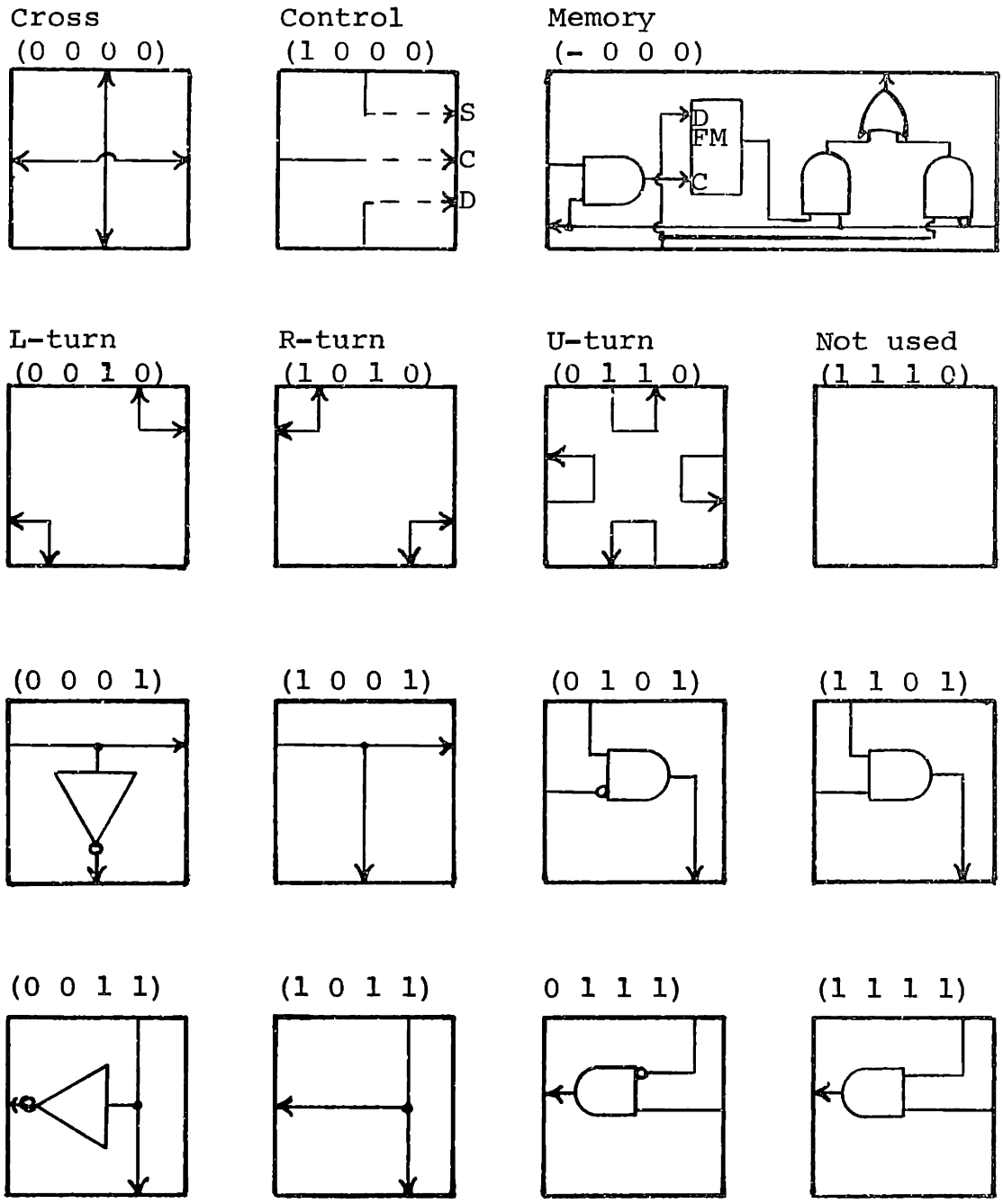
Section 4.1 The General Cell

The General cell is amenable to realization of highly parallel sequential machines. This cell incorporates the mechanisms essential to our testing and repair approaches for high-relcon machines. Function states for processing, transmission, and memorization of information allow realization of an arbitrary sequential machine in the processing layer of an arbitrarily large checkerboard array. A Control function state that transmits processing inputs as loading outputs enables an embedded high-relcon machine to load cells in its environment. Such a machine may control a loading arm and four test links to test, program, and repair its cellular environment. Two or more such machines may monitor and repair each other.

Figure 4.1 gives symbols for the General cell's function states. Like the cells of the last chapter, each General cell only communicates directly with its neighbors or the extra-array world. There are no signal busses extending through a General array. We've discussed the testing and repair advantages of this type of cellular design. The loaders of the Shift-register and General cells are identical, except that processing inputs can control loading outputs when a General cell is in the Control function state. Each of a cell's four sides has S, L, and D loader inputs and outputs (as in figure 3.5), and a Processing input and output. Like the Shift-register cell, the General cell incorporates all the loader's options. The shift-register loaded by a loading arm has four function-specification state bits - FM, F0, F1, and F2 - and three loader state bits - L00, L01, and LSTA. This shift-register

Fig. 4.1 General's Function States

Function states are shown for all values of (FM F0 F1 F2).



Note: An unshown processing output connects to the opposite side's processing input. Loader outputs are only affected by processing inputs in the Control state.

is reset when power is turned on. In all but one function state, only loader inputs affect loader outputs. However, in the Control state each Processing input from one of three sides affects a different loader output at the right side: P.U.IN = S.R.OUT, P.L.IN = C.R.OUT, and P.D.IN = D.R.OUT. This state allows a machine, embedded in an array as a collection of function states, to re-program its cellular environment by appropriate processing signals transferred to some cell's loader outputs.

The Cross, L-turn, R-turn, and U-turn states are types of balanced, non-branching transmission states. Cross is a crossover; the others are bends. We'll see that Cross, L-turn, and R-turn are very useful for testing and fault-avoidance; note their similarity to the shift-register cell's non-tip states (see figure 3.12). Cross, L-turn, and R-turn may combine to form a transmission link arm that snakes through an array. Such a link may act as a two-way wire bus, or simply as a wire carrying information in one direction. U-turn is useful in testing; note its similarity to the shift-register cell's tip states.

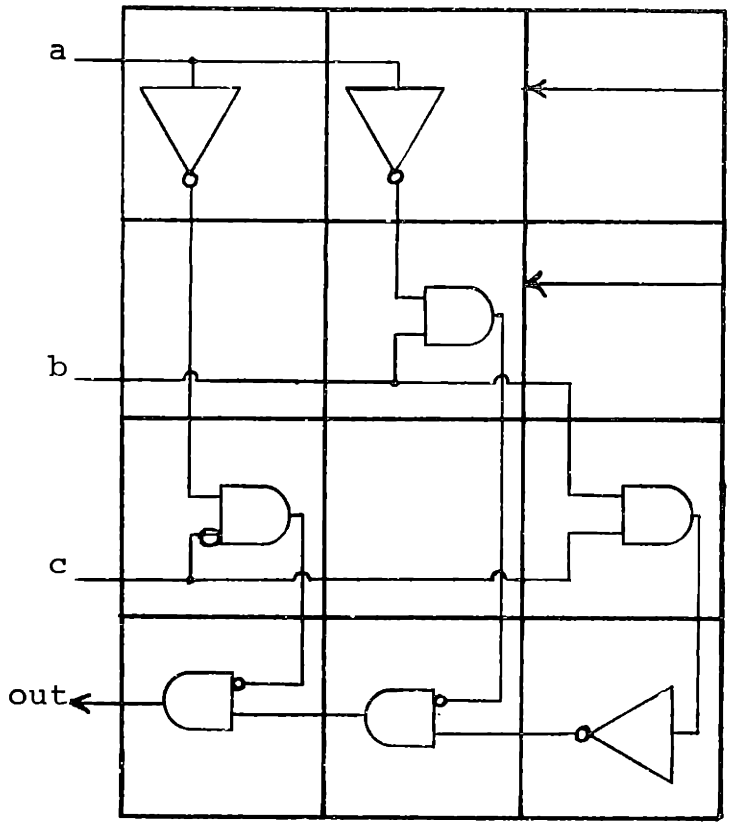
State (- 1 0 0) is a memory state. In this state, FM is not used in its customary function-specification state bit role; instead it's a processing layer P.R.IN-selectable D flip-flop. A Reset input for this flip-flop is not provided, but this function is easily simulated by proper manipulation of P.R.IN and P.D.IN. This memory state is very convenient for realization of registers, addressable read-write memories, and other common memory modules.

The states associated with F2 = 1 allow convenient realization of a

Fig. 4.2 A Function Performed In Different Orientations
(first of 2 pages)

Function F: out=(a + c)(a + $\overline{b}$)($\overline{b}$ + $\overline{c}$)
Some busses between opposite sides are not shown.

A) Array A has inputs and output at its left.



B) Array A has its inputs and output at its right.

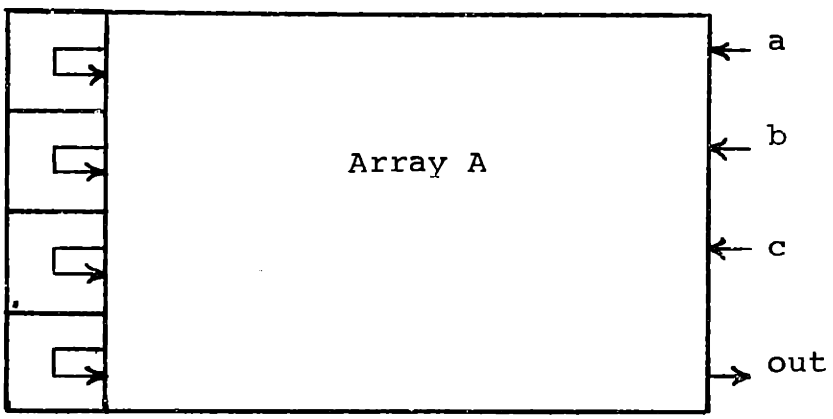
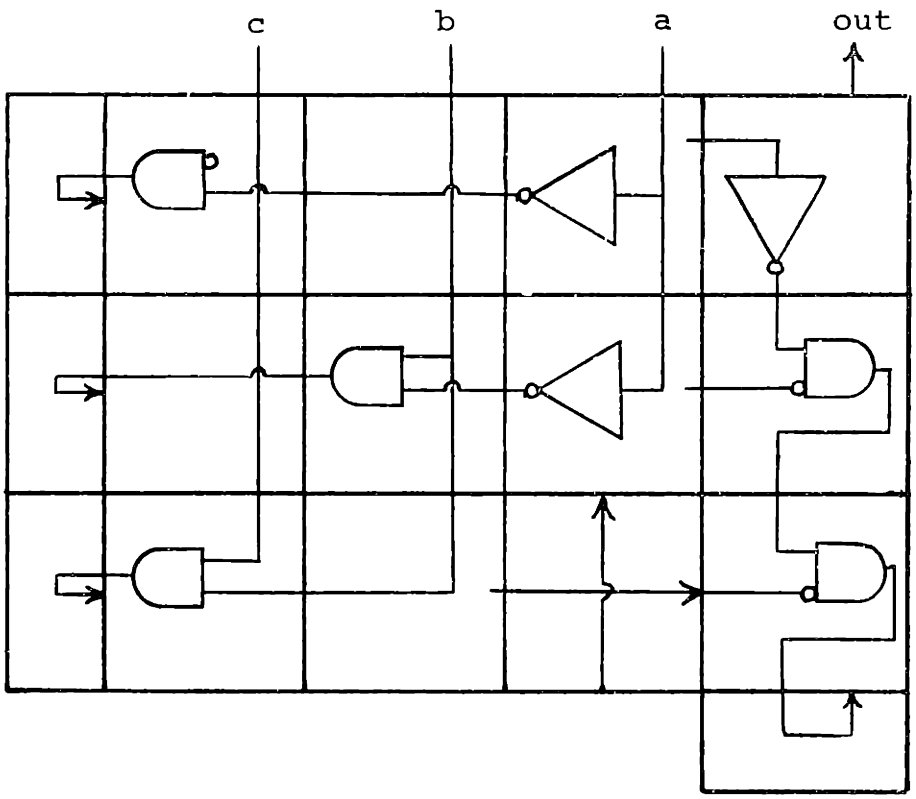


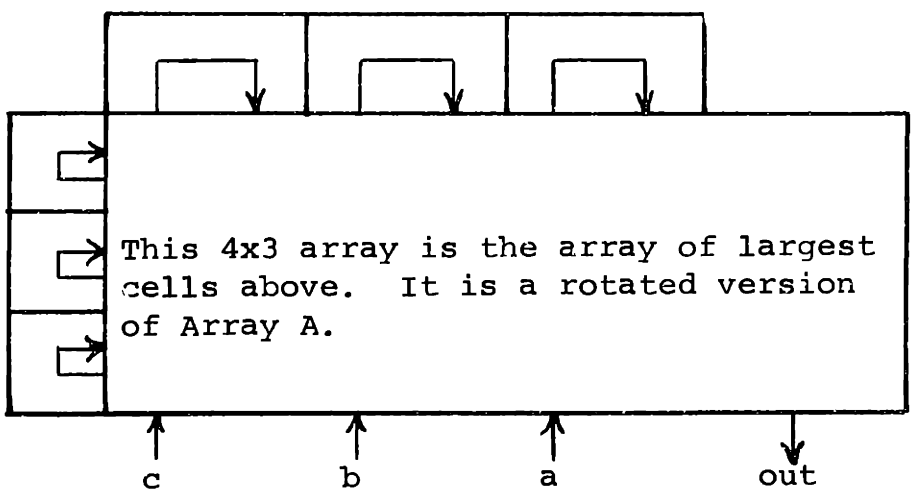
Fig. 4.2 A Function Performed In Different Orientations
(second of 2 pages)

Function F: $out = (a + c)(a + \overline{b})(\overline{b} + \overline{c})$

C) A rotated version of Array A, aided by U-turns, performs F with its inputs and outputs above.



D) A rotated version of Array A, aided by U-turns, performs F with its inputs and outputs below.



combinational logic function expressed, for instance, as a minimum product of sums or sum of products. Figure 4.2 shows that these states function and combine very much like the states in programmable logic arrays. These General states, coupled with U-turn, were designed to eliminate the severe waste of cells that often results from cell designs that only operate on signals coming from a given, preferred direction. Those designs demand the use of many cells to turn an input signal into an appropriate orientation. Figure 4.2 presents sample realizations of a logic function, and indicates the ease with which General arrays operate on signals to or from various directions. This is particularly important for functions with many input-output lines.

The fact that digital machines usually require extensive signal-routing explains the cell's emphasis on bussing signals from one side to an opposite side. This allows a cell to perform bussing operations at some output while simultaneously performing a branch, combinational logic, or memory function at another output.

It's easy to see that arbitrarily large, properly programmed General arrays can perform any time-independent, effectively computable computation. It's been demonstrated that today's general-purpose computers can perform such a computation if their memory capacity is unlimited (see <Minsky 67>). Like <Banks 71>, we therefore need only show the ability to realize an extensible general-purpose computer in the General array. The ability to realize a general-purpose computer comes from the availability of its basic components - Nand gates, wires,

and memory elements. Extensibility comes from the Control state and loading mechanism. An array-embedded computer can be constructed to control the processing inputs, and consequently the right side's loader outputs, of a Control cell on a right side of the computer's periphery. We've seen that appropriate loader signals to an arbitrarily cell allow the growth of a loading arm to an arbitrary cell in a perfect array. The array-embedded computer can consequently send signals to increase its memory as needed.

Since such a machine has a moveable construction arm, it can construct arbitrary digital machines in an arbitrarily large array. For instance, it can construct a copy of itself. It is therefore also a universal constructor.

We'll see that, for array faults of a certain assumed nature, an Array Programmer can test an array and embed a perfect machine in a flawed array. Since the Array Programmer can be realized in a flawed array, the General cell allows universal repair for faults of an assumed nature.

Thus the General array can support a universal computer-constructor-repairer.

General is universal, but simple. A processing mechanism's complexity results in advantages and disadvantages whose importance depends on the cell's use. The need for a low proportion of flawed cells in an array embedding high-relcon machines currently requires that only simple cells be fabricated on a slice containing many cells. Basic, universal cells allow an embedded machine's designer to exploit the parallelism in a given algorithm. Testing, repair, and signal-routing

require cells to assume transmission states; using a very complicated cell in such a simple state wastes most of its complicated mechanism. On the other hand, a simpler cell has a smaller ratio of processing circuitry to loading circuitry; the simpler cell suffers from a higher associated overhead when the loading circuitry is quiescent. When a cell's simplicity requires more cells for a given machine, the function-selection in each cell slows the machine.

One component of a cell's complexity is its number of processing lines. If a cell has many processing lines in a side-set, routing each of the lines to or from a different part of an array requires many cells to break the lines from the side-set's bundle of lines. Furthermore, unless independence of different parts of a cell's processing mechanism is assumed, test time per cell rises exponentially with its number of processing inputs.

An array designer considers these general considerations and specific design goals when designing a high-relcon array.

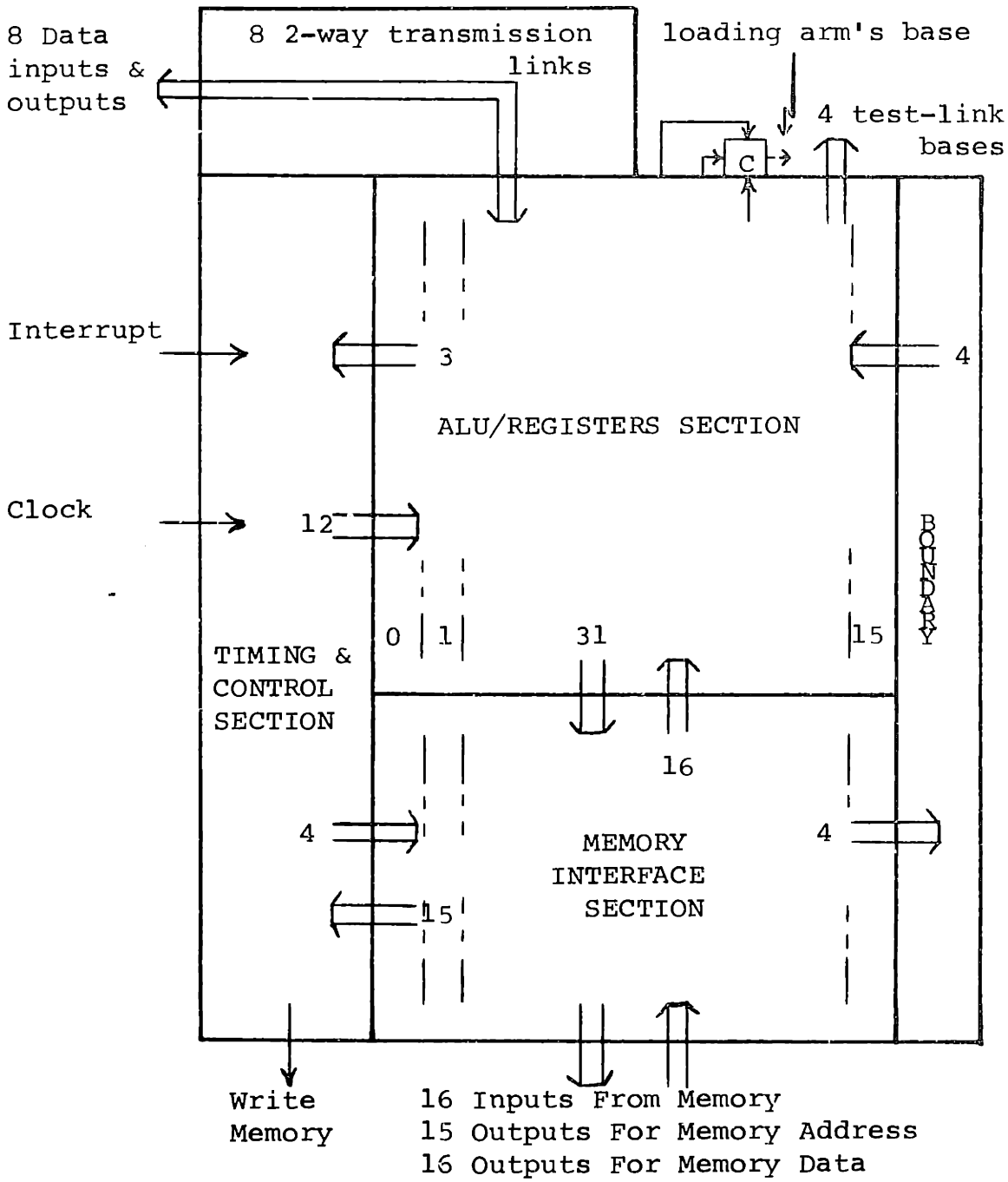
General's processing mechanism is one consistent with efficient implementation of our testing, repair, and computation goals. The Cross, L-turn, R-turn, and U-turn states are important components of test arms and transmission links in testing and repair. Although General cells perform wiring operations in many states, signal-routing is so important that expanding General's signal-routing capabilities might be worthwhile. Some variation of the Control state is necessary for realization of our goal of array-embedded array manipulators. The sequential machines we envision for General would use enough memory to support a memory

state; construction of memory elements from gates would require a greater proportion of cells in an array than is justified by the resultant simplification of a cell. Indeed, actual applications might argue for more memory elements in a cell and/or more memory-oriented function states. It's true that cells with (F1 F2) = (0 1) are rotated versions of cells with (F1 F2) = (1 1), and that cell states can be eliminated by clever use of the (0 1 0 1) cell. Again, these cell simplifications would probably result in disproportionate numbers of cells for most applications.

We briefly digress to give a little information about a familiar machine, a miniprocessor, unique only because we designed it as a machine embedded in a General array, and because a special feature allows it to test and repair its cellular environment. This miniprocessor could be the processor of a universal computer-constructor-repairer. This digression is intended to give some specific information about our cellular realization of a machine like one many readers are familiar with; those who aren't will not lose continuity by jumping to the next section. We don't think the General cell is particularly suited to realization of conventional processors, because processors are already mass-produced ICs. However, we do want to demonstrate the General cell's power. Furthermore, this design gives some insight into the number of cells of various types needed to implement a somewhat familiar machine.

The miniprocessor we designed is a 16-bit parallel, synchronous, single-sequence machine with conventional A-B-C bus structure. Figure 4.3 gives a map of the miniprocessor. The machine has 66 extra-array lines: 1 clock, 1 interrupt,

Fig. 4.3 Map Of Miniprocessor-Tester-Repairer



8 data inputs, 8 data outputs, 15 memory address, 1 "write memory", 16 memory data inputs, and 16 memory data output lines. The machine also has four test links, and one loader arm for testing and repair; we discuss use of these easily implemented features in later sections. The machine's main sections are a Timing and Control section, a Memory Interface section, and an Arithmetic-Logic Unit/Registers section. Both the Memory Interface and ALU/Registers sections have 16 similar modules, one for each bit-slice. The Memory Interface Section contains the 14-bit instruction register, and many transmission links. The ALU/Registers section contains six large (15 or 16-bit) registers; these are the Accumulator, Program Counter, Instruction, Subroutine Return, Interrupt Return, and Input-Output/Test & Load registers. This section's 16 blocks are identical, except that the block interfacing with the Timing and Control Section is slightly different. The miniprocessor has fairly conventional arithmetic, logical, subroutine, interrupt, and input-output capabilities. Instructions are processed in a conventional, single-sequence way.

We specified this machine as one embedded in a perfect, rectangular General array with about 9,000 cells. Its non-writing indirect memory reference instruction takes three cycles, with about 700 cell-delays for each cycle. Since most cells introduce about one gate-delay, a cycle takes about seven microseconds for a technology with a gate-delay of 10 nanoseconds. Each rectangular ALU/Register slice gives an example of a mix of cell types; each has 18 unused cells, 147 transmission cells, 53 combinational logic cells, and 6 memory cells. Each

bit-slice has 88 essential cells: 53 combinational logic cells, 6 memory cells, 16 U-turns, and 13 branches. There are 118 non-branching transmission cells used as wires. Other parts of our processor-tester-repairer had an even higher ratio of wire cells to essential cells. This emphasizes the importance of good signal-routing capabilities in high-relcon arrays.

Testing and repair techniques using the General cell depend only on the loader and processing transmission states, so the testing and repair approach for General can be applied to other high-relcon arrays with loader and processing transmission capabilities analogous to General's.

Section 4.2: Introduction To Testing, Construction, And Repair

Testing, configuration, and repair for high-relcon machines is similar to those processes for balanced arm machines, although there are important differences. The chief differences are that high-relcon machines are not conducive to the interwoven processes of test and repair; and test and repair are more difficult and less efficient for high-relcon machines. We consider an approach applicable to any high-relcon checkerboard array with our loading arm and transmission link facilities. We mention how a Control state like General's may be tested, but this state is not essential to our testing and repair approach.

In considering embedding an arm in an array, we made certain reasonable assumptions concerning failure modes of the array. Then the interwoven processes of testing and repair were considered. These processes occurred by the gradual snaking of an arm into an array. A cell was tested only insofar as necessary to establish its successful incorporation into a desired arm; this usually meant a cell wasn't tested in all of its states. Testing of a new arm-tip cell required using a partially tested cell, but this presented no difficulty.

In considering embedding high-relcon machines, we make assumptions very close to those made in the last chapter. However, most high-relcon machines are poorly suited to gradual growth and testing for two main reasons:

1) In growing an arm, the number of relevant extra-array processing inputs and outputs remains fixed. However, high-relcon machines usually have a variable, sometimes large number of relevant side-sets

as they're grown. Most generally, this requires test arms linking a test machine to the relevant side-sets at a partially grown machine's periphery. This requires an Array Programmer to have a variable and often large number of test arms and associated links. We'd much prefer to have a low, fixed number of such links. Consequently, we test cells individually, relying on independence assumptions about cells' behavior.

2) Embedding an arm in a flawed array can be done efficiently by gradual growth of the arm, followed by local jogging of the arm to include clumps of good cells. High-relcon machines benefit greatly from a global repair approach that begins with a description of all the flaws in an array. This means that repair efficiency is improved by separation of the test and repair procedures.

These considerations explain why the test and repair processes for high-relcon machines are segmented into a series of several distinct procedures.

First the Array Programmer's *Test* procedure tests an array, noting the location of faulty cells. This testing is independent of the essential machine that is eventually embedded in the array, so Test's results are valid until an array develops a new flaw.

A *Repair* procedure determines how to embed a perfect machine in the faulty array. Repair accepts a flaw pattern description of a flawed array from Test. Repair also accepts an essential network model of the desired essential

machine. Repair's output is a description of the repaired array that places each of an array's cells into one of the following four categories:

1) The cell is flawed.

2) The cell is an essential cell.

3) The cell may assume an arbitrary non-Control function state. None of its outputs is relevant to the embedded machine's output.

4) The cell is in a Cross, L-turn, or U-turn transmission state. The cell is part of one or more wires associated with relevant inputs and outputs of essential cells.

The *Construct* procedure constructs a perfect machine in a flawed array. Construct modifies Repair's output by mapping each of an essential machine's essential cell states into a properly located essential cell. Repair has arranged that essential cells be wired together in the proper way. Construct accepts from Test a model of the flawed array stating which side-sets may definitely be used for loading. Test develops this model as it tests an array. Every cell that Test finds to be good has some side-set that can be used for loading the cell. Construct only activates the side-sets specified by Test as it extends a loading arm into an array. Construct's loading arm may touch any good cell, but it always touches and loads essential cells (category 2) and wire cells (category 4). When Construct completes its loading task, a perfect machine is embedded in the array. The embedded machine is ready for further test or use.

Our high-relcon repair procedure assumes that the length of wires

between essential cells is irrelevant to the proper functioning of an embedded machine. Possible techniques for assuring the validity of this assumption are suggested at the end of this chapter.

Section 4.3: Testing

Testing an array embedding a high-relcon machine involves the one-by-one testing of the cells in that array via test links between the tested cell and an Array Programmer. This procedure is relatively difficult, compared to testing of an array embedding an arm, because Test doesn't know how Repair will map a perfect machine into the faulty array. This implies that most cells must be tested in all their function states. Because all of a cell's accessible processing inputs and outputs may affect an embedded machine's output, Test must vary the accessible processing inputs to the cell, and monitor the accessible processing outputs. Consequently testing a cell usually involves linking each accessible side-set with the Array Programmer via a test link. Figure 4.4 shows that the processing transmission states are ideally suited for this task.

Test makes the assumptions listed below. Each assumption is analogous to the corresponding assumption made for shift-register cells.

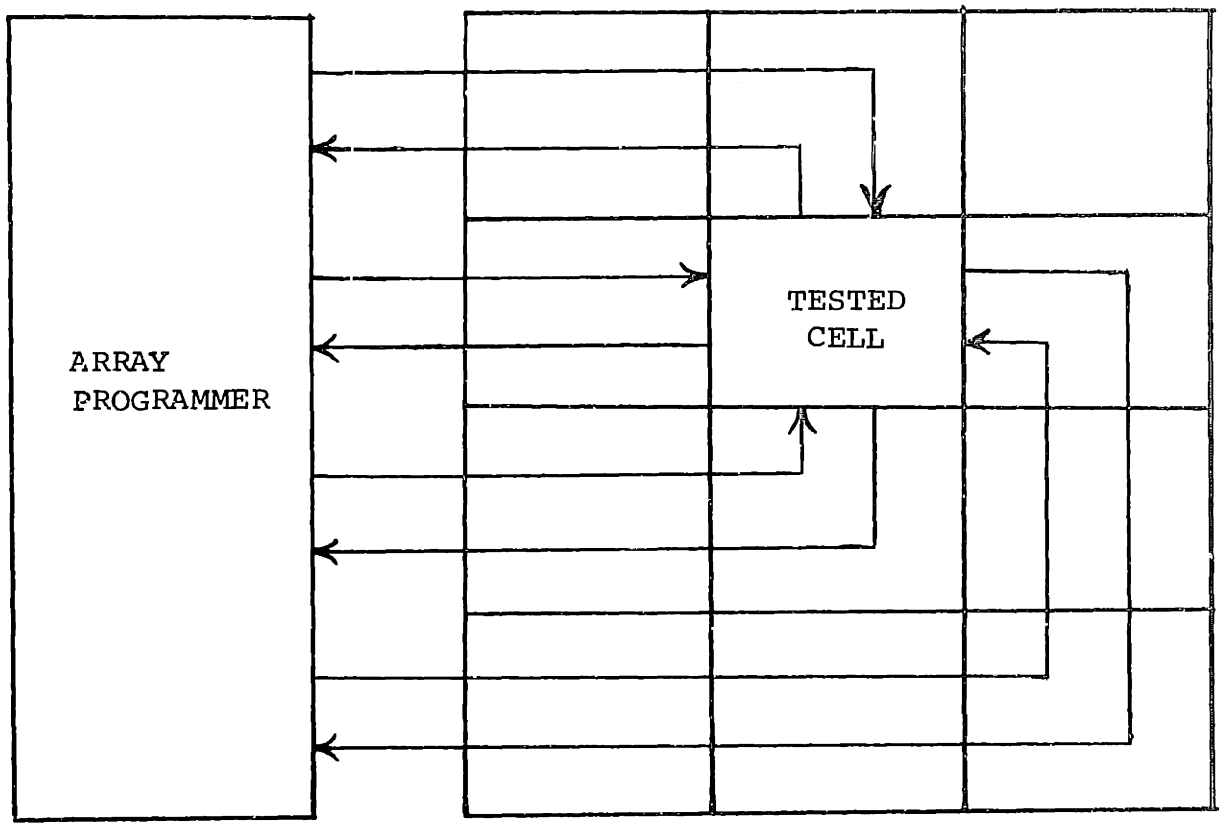
1) Good cell's are only loaded under Test's control, or because of a branch cell, and not by signals caused by faulty cells.

2) A cell's performance depends only on that cell's mechanism, state, and input signals.

3) A successfully tested cell does not develop a fault before the Construct process is over.

4) A cell's processing outputs don't depend on its loader state; and, unless the function state is the Control state, loader performance

Fig. 4.4 Test Links To Processing Lines Of Tested Cell



doesn't depend on the function state. This non-essential, reasonable independence assumption allows a reduction in testing time. Test need not, for instance, test a function state for all loader states.

In considering testing, we first focus on the test stages that occur when all tests are passed. We then address implications of test failures, and possible flaw models. The modelling question is pursued in the subsequent description of Repair.

Testing a cell requires explicit tests of its permissible function states, and concurrent implicit tests of its loader. Tests of a typical cell involve two types of communication between the Array Programmer and the cells at the test site. Test links connect the Array Programmer to the processing inputs and outputs at the test site, as in figure 4.4. The test links are composed only of cells in the Cross, L-turn, or R-turn transmission states. The Array Programmer requires one test link to each accessible side-set. The Array Programmer communicates to a tested cell through signals to and from the base of each test link. Besides the test links, a loading arm extending to the tested region links the Array Programmer with loader inputs. This arm may pass through cells that are also in a test link, or even through the tested cell. (However, the Array Programmer should not relay high processing signals down a test link connected to the up, left, or down side-set of a cell being loaded, and temporarily in the Control function state.) The Array Programmer may change the state of cells, such as the

tested cell, either by sending signals into the base of the loading arm, or by sending processing signals down three test links that converge on a Control cell.

Testing a cell's non-Control function states involves cycling it through those function states the cell may assume in an embedded machine. For each such state, appropriate stimulus signals, and responses to these signals, flow through the test links. We'll see that Repair always specifies that a good cell adjacent to a hopelessly flawed cell assume a Cross, L-turn, or R-turn state; this is an example of the tested function states being a subset of the set of all non-Control function states. In this case, only some of the tested cell's side-sets are accessible. A functional test of a non-Control, non-Memory function state involves at most $2^4 = 16$ input combinations. Fewer input combinations may be appropriate if some side-sets are inaccessible, or if independence of certain outputs and certain inputs is validly assumed. For instance, the left Processing input might be experimentally found to never affect the right Processing output in the U-turn state, even for a faulty cell; this would allow simplified testing of the U-turn state.

Testing a cell's response time in a given state is possible, if the Array Programmer can accurately time a test link's output response to an input. Differential techniques then allow the calculation of the delay associated with each test link. Additional delay comes from delay through the tested cell. Unfortunately, accurate timing requires time resolution of less than one gate-delay, which is difficult to achieve.

If the Array Programmer knows the delay through each test link, test time for a given function state depends on how quickly the Array Programmer can change the input to a test link. This is limited by the Array Programmer's speed or the bandwidth of a cell. Any inaccuracy in the estimate of the delay through a link may also limit test speed by effectively reducing the bandwidth of the link.

In testing a Control function state, test links connect the Array Programmer to all four of the side-sets of the cell in the Control state. First the Array Programmer verifies that the right side-set's test link is not a test arm, by ascertaining that a signal into the base of the test link doesn't return to the base after an appropriate delay. Then signals into the up, left, and down processing inputs command the tested cell to load the cell to its right into a U-turn state. The Array Programmer again tests the right test link. If it's now a test arm, the Control state is good; otherwise the Control state is bad.

Testing a cell's loading mechanism is implicit in the tests of the cell's permissible function states. If a cell fails its function tests, Construct doesn't try to load it. If a cell passes its function tests, a loading arm has successfully loaded the cell and retracted from the cell. Therefore Construct's loading arm can also load the cell from some side-set. Test keeps a map of which side-sets the loader uses to successfully activate and de-activate working cells. Construct uses this map to determine the path of its loading arm.

After a cell has been tested, test links must be moved to a new test site, if there is any remaining. The new test site is usually a cell adjacent to the

last tested cell. Thus only the tip ends of the loading arm and test links need be moved. This is fairly simple, since a loading tip is at the test site. Each test link is gradually extended as part of a test arm, just as arms were extended in chapter 3. After each incremental extension, all links are tested to assure growth is proceeding satisfactorally. Since a test arm only incorporates cells in transmission states, faulty cells are discovered and avoided as in chapter 3. This gradual extension is particularly appropriate in an array with a high fault density. In an array with a very low fault density, the speedup from non-gradual growth could offset the slowdown from a faulty cell's confusion factor.

The process of moving the test site terminates with each of the new test cell's accessible side-sets connected to a test link. The new test cell, in the U-turn state, is the tip of one or more test arms. The test process is repeated for this cell.

In the last chapter we noted that failure after an incremental arm extension could mean several things. For instance, the new tip cell might be hopelessly flawed, or it might just be incapable of receiving information from the indicated direction. We noted that various flaw models might be appropriate, depending on the cell layout and the sophistication of the Array Programmer.

This modelling difficulty again rises with the high-relcon array. Growth of test links is analogous to growth of shift-register arms, so the same comments apply. A similar difficulty arises when a cell is in the process of being fully tested. The cell may produce nonsense in all states; modelling that cell as hopelessly

flawed is then definitely appropriate. However, it may also happen that an output value is only wrong when it's a function of a particular input coming from a cell otherwise considered good. Modelling a side-set, or even a particular input or output, as unuseable might be valid. Choice of sophistication level in the Repair procedure's treatment of slightly flawed cells depends on whether the sophistication is worth the computational cost. In our discussion of Repair, we assume an array may be modelled by a flaw pattern in which every flawed cell is represented by an X.

Section 4.4: Repair

The Repair procedure determines how to embed a perfect, high-relcon machine in a flawed array. Test passes Repair a flaw pattern description of the flawed array. Most generally, Repair may embed the largest grid machine it can. Construct may then construct in the flawed array any machine with an essential network that fits into this largest grid. We consider grid-embedding first. Most actual embedded machines have essential cells with irrelevant side-sets; their essential networks are grids with squares and links missing. The most general Repair method is then less efficient than a method which notices an incomplete grid. We eventually consider such a less general, more efficient Repair procedure. Its main drawback comes when a new machine must be embedded in the flawed array; if the new machine's essential network isn't a subnetwork of the original essential network, new repair of the array is necessary. Repair decides how to locate and wire together good essential cells, using only good cells in transmission states as wires, to embed a perfect machine in a flawed array. This allows Construct to associate the proper function state with each essential cell, and to wire together essential cells with transmission states dictated by Repair.

The Repair procedures that we have written assume the simplest fault model: a cell is either good or hopelessly flawed. This conservative assumption is most questionable, because of its harshness, when Test finds a side between cells A and B is impassable. This condition can be safely modelled by saying that either cell A or cell B is hopelessly flawed. Repair knows that an unflawed cell should

not allow a faulty cell's ouput to affect an embedded machine's output. Thus no output-affecting signal will be transmitted across the faulty side. In all other cases where a cell displays some faulty behavior, it's modelled as a hopelessly flawed cell.

Consider two checkerboard arrays with the same distribution of good and bad cells, and consequently the same flaw pattern. The first array is for embedding a grid machine, and the second is for embedding an arm machine. Since there are many ways an arm can wind through all the essential nodes of any of the grid's relcon networks, the longest embedded arm in the second array contains at least as many essential cells as the largest grid embedded in the first array. Embedding a grid in a flawed array involves using some good cells purely as links between essential neighbors. Cells in the corresponding position in the second array can be used as arm cells, because cells in links have relcon as high as cells in arms. Thus optimum repair efficiency for the arm machine is at least as high as optimum repair efficiency for the grid machine, given the same flaw pattern.

How do the optimum efficiencies compare? Answering this question from a non-experimental, purely mathematical perspective appears very difficult. An analytic, tractable expression for optimum repair efficiency, given a particular flaw pattern, appears impossible for most cases. An expression for average efficiency, averaged over all flaw distributions for a given number of flawed cells in an array of a certain size, also appears impossible for both arms and grids. Although one might find some lower bounds for repair efficiency, it's likely that the

bounds would not be close enough to the optimum to be practically interesting. Furthermore, one would still have little knowledge of the difficulty of attaining or surpassing a lower bound in an actual Repair procedure.

Consequently, our approach has been to write promising repair procedures, observe their behavior, and use our observations to suggest improvements in the procedures. Some of these suggestions are implemented, and the process repeats.

Many actual essential machines contain a mixture of low-relcon and high-relcon essential cells. Figure 4.5 gives the relcon network for our embedding of one bit-slice of the ALU/register section of our processor-tester-repairer in a perfect array. The upper-right region of the bit-slice has many high-relcon essential cells, and has few links to nodes outside the region. On the other hand, the bit-slice has many relcon-2 chains, balanced arms, and even relcon-0 cells. Many relcon-2 and relcon-4 cells are used as a wire or crossover.

In embedding the bit-slice in a flawed array, we could approximate its essential network by a grid. Adding constraints to Repair in this way would have three major effects:

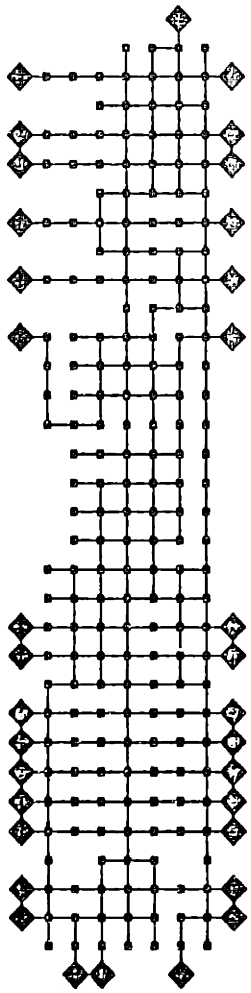
1) It would simplify the description of the slice's essential network.

2) It would make Repair's results valid for any machine that fit into a perfect 7 x 32 array.

4) It would diminish Repair's efficiency.

In this section, we first consider grid-embedding - the most difficult, general repair

Fig. 4.5 Relcon Network For One ALU/Register Bit-slice



In the rectangular bit-slice there are 224 total cells: 18 relcon-0 cells, 16 relcon-1 cells, 78 relcon-2 cells, 28 relcon-3 cells, and 84 relcon-4 cells. The bit-slice's relcon network represents a compact embedding of a machine with 88 essential cells: 53 combinational logic cells, 6 memory cells, 16 U-turn cells, and 13 branch cells. 118 of the relcon-2 and relcon-4 cells are non-branching transmission cells, which are used as wires. Other parts of our processor-tester-repairer had an even higher ratio of wire cells to essential cells.

in a checkerboard array. We compare grid-embedding to arm-embedding. We then suggest an approach which improves embedding efficiency by noticing missing links between nodes in a high-relcon machine's essential network. This type of approach is the most feasible for most embedded machines.

It's usually difficult to optimally embed a grid in a flawed array. If the array has very few flaws, *Grid Repair* is easy because there are few reasonable ways to interconnect good cells to form a large grid. As the number of flaws in the array increases, the number of reasonable ways to form a large grid explodes. Repair cannot consider all possible embeddings; this would take too much computation. The obvious, simple repair methods we've applied to these arrays don't work well. Eventually there are so many flaws in an array that the embedding problem is easy, because it's obvious that no grid can be embedded in the array.

We focus on the most difficult grid-embedding flaw region. We present a reasonable approach which is considerably more sophisticated than the only similar approach we've seen, which is Kukreja's repair of cutpoint-connected arrays.

The nucleus of Grid Repair is a *Twist Repair* procedure. This procedure, which we'll detail, is very efficient at embedding grids in moderately large rectangular arrays of flawed cells. Another procedure, *Blockoff*, accepts as inputs:

1) an essential network for a machine embedded in a perfect array. This network is described as interconnected rectangular grids.

2) a flaw pattern for a flawed array, where each cell is either perfect or flawed.

We temporarily assume that Repair need not consider the location of a flawed array's input-output lines; we assume these are attached after an array is repaired. We'll see that Blockoff is easily modified to consider the location of input-output lines. Blockoff partitions the flawed array into rectangular blocks separated by interconnection strips; each block is intended to hold a grid. Blockoff then asks Twist Repair to determine how to put a proper-sized grid into each of the blocks. If Twist Repair cannot perform its task for one of the blocks, Blockoff fails. Otherwise Blockoff decides whether it can interconnect the proper grid links extending from each block. If it succeeds, Blockoff passes the resulting description of the repaired array to Construct. If Blockoff can't interconnect the grids, it asks Twist Repair for an alternate embedding for at least one block. In re-repairing any block, Twist Repair continues its repair attempts from the point of its last success. The process iterates, until Blockoff succeeds or fails.

Repair is oriented toward rectangular blocks for several reasons. First, this is the most natural, tractable structure in a checkerboard array. Second, the General cell is suited to rectangular machines. Finally, any checkerboard machine can be viewed as a composite of rectangles of various sizes.

We first detail Twist Repair, and then Blockoff. We examine their response to actual embedding problems, compare their performance to Arm Repair, and note their limitations. We also suggest reasonable extensions of the Repair

procedures we've written.

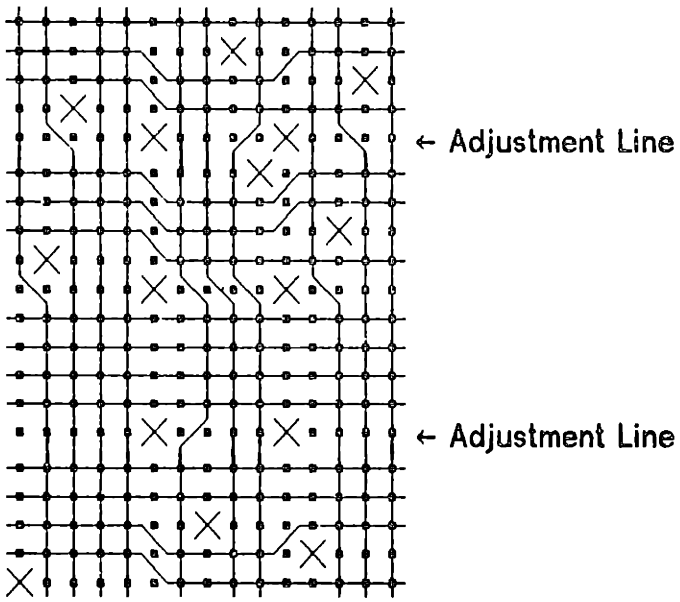
The simplest, most obvious way to embed a grid in a flawed array only uses a cell as an essential node in the grid if the cell's row and column contain no flawed cells. A good cell in a flawed *line* - a row or column - enters the Cross state, so it interconnects essential neighbors. We'll call this repair technique *Simple Repair*. Simple repair of checkerboard arrays is analogous to Kukreja's repair of cutpoint-connected arrays.

Note that Simple Repair is the best possible grid-embedding repair when an array has few flawed cells. If an array has only one flawed cell, an embedded grid must have at least one less row and one less column than the flawed array; the flawed cell's row and column are bottlenecks.

Unfortunately, this Simple Repair is very inefficient as the number of flaws in an array increases. For such an array, we'd like an approach that is able to twist a grid's lines through an array, so that some cells in flawed lines can still be used as essential cells. The L-turn and R-turn, cooperating with the Cross, are ideal for this purpose. Because of the way repaired blocks must interface, we assume a grid's lines must extend from one side of a block to its opposite side.

The Twist Repair approach, which includes Simple Repair, uses horizontal and vertical *adjustment lines* extending completely through a flawed array (see figure 4.6). Any flaw on an adjustment line must be at the junction of a horizontal and vertical adjustment line. Adjustment lines break the array into *boxes* - rectangular regions of cells. At most one flawed cell is allowed in each box. If a

Fig. 4.6 Flawed 15x20 Array Twist-Repaired Into A Perfect 10x14 Array



Explanation:

The relcon network above indicates the states of good cells and flawed cells in a grid-repaired array. Flawed cells are indicated by an X. Good, unused cells in an arbitrary state are indicated by •. Good cells that are essential cells in the grid are indicated by +. Other cells are used to interconnect essential grid cells. The L-turn state is indicated by ⌜, ⌝, or ⌞, depending on the context. Similarly, the R-turn state is indicated by ⌜, ⌞, or ⌝; and the Cross state is indicated by ─ or │. Note that jogging a wire requires the use of at least two L-turn or R-turn states.

line of boxes is free of faults, adjacent boxes in the line interconnect across adjustment lines via Cross cells. If a box is in a row (or column) of boxes, some of which contain flawed cells, one row of the box is not used for essential cells. If the box contains a flaw, the flaw's row is the row with no essential cells; all unflawed cells in that row of the box assume the Cross state. If a box is in a row of boxes with flaws, and the box contains no flaw, an arbitrary row may be put into the Cross state. Thus all the boxes in a row have the same number R of rows useable as rows of essential cells. Cross, L-turn, and R-turn states are used in adjustment lines between boxes in a row to yield R embedded grid rows extending through all the row's boxes.

Several considerations make the Twist Repair approach a reasonable one. Because exhaustive consideration of all repair possibilities is computationally excesssive, a reasonable, heuristic approach is necessary. Simple Repair is inadequate for most arrays with more than a few flaws. Twist Repair recognizes the equivalence of many specific embeddings. For instance, an adjustment line that doesn't include any flawed cell may occupy any line of cells between two flawed cells; all such lines are equivalent. Recognition of equivalence limits computational difficulty. Furthermore, this allows Blockoff more flexibility in interconnecting blocks repaired by Twist Repair. We found that forcing L-turn and R-turn links onto adjustment lines results in far less repair confusion and inefficiency than less restrained use of these states. Consider snaking an embedded grid's row through a flawed array of unbalanced cells, such as a General array. The only possible

essential cells in the snaking path through the flawed array are those cells which the path links to cells on the same row in the flawed array. This suggests that jogging of the line and movement of the line in the vertical direction should be limited. Twist Repair often uses all the side-sets of cells in the L-turn and R-turn states; this efficiency helps minimimize the number of cells used as repair links. Twist Repair also attempts to place essential neighbors close to each other in a flawed array. This is helpful for two reasons. First, since wires between essential neighbors are useless as essential cells, it's important to minimize the number of cells in each wire. Second, an embedded machine's maximum speed is limited by delays through wires; intended processing is only done at essential cells. Our ultimate justification for Twist Repair is that it is better than any other methods we've considered for repairing small rectangular arrays to embed grids.

The Twist Repair program's inputs are a flaw pattern and a request for a minimum acceptable number of grid rows and columns. As in the arm experiments, a square array's flaws are randomly generated. Starting with a good guess of where to draw adjustment lines, Twist Repair considers alternative adjustment line placements exhaustively - ignoring equivalent placements - until it succeeds. Table 4.1 is analogous to a table given for balanced arms, showing the best square grid Twist Repair embedded in experiments varying the number and distribution of flaws in the square array.

Figure 4.7 shows curves based on the information in the table. The curves show the average of %oftotal for a given %flawed, for various array sizes.

Table 4.1 Results Of Twist-Repair Grid-embedding Experiments
(1st of 2 pages)

Key: %flawed - flawed cells as percent of all cells
cells - total cells in square array
flaws - total flawed cells in array
max-grid - the largest square grid our program embedded
%oftotal - max-grid as percent of cells
timelim - time limit, in seconds.
time - the time the program ran
%oftimelim - time as percent of timelim
* - For two starred (or unstarred) arrays of the same size, one set of flaw coordinates is a subset of the other.

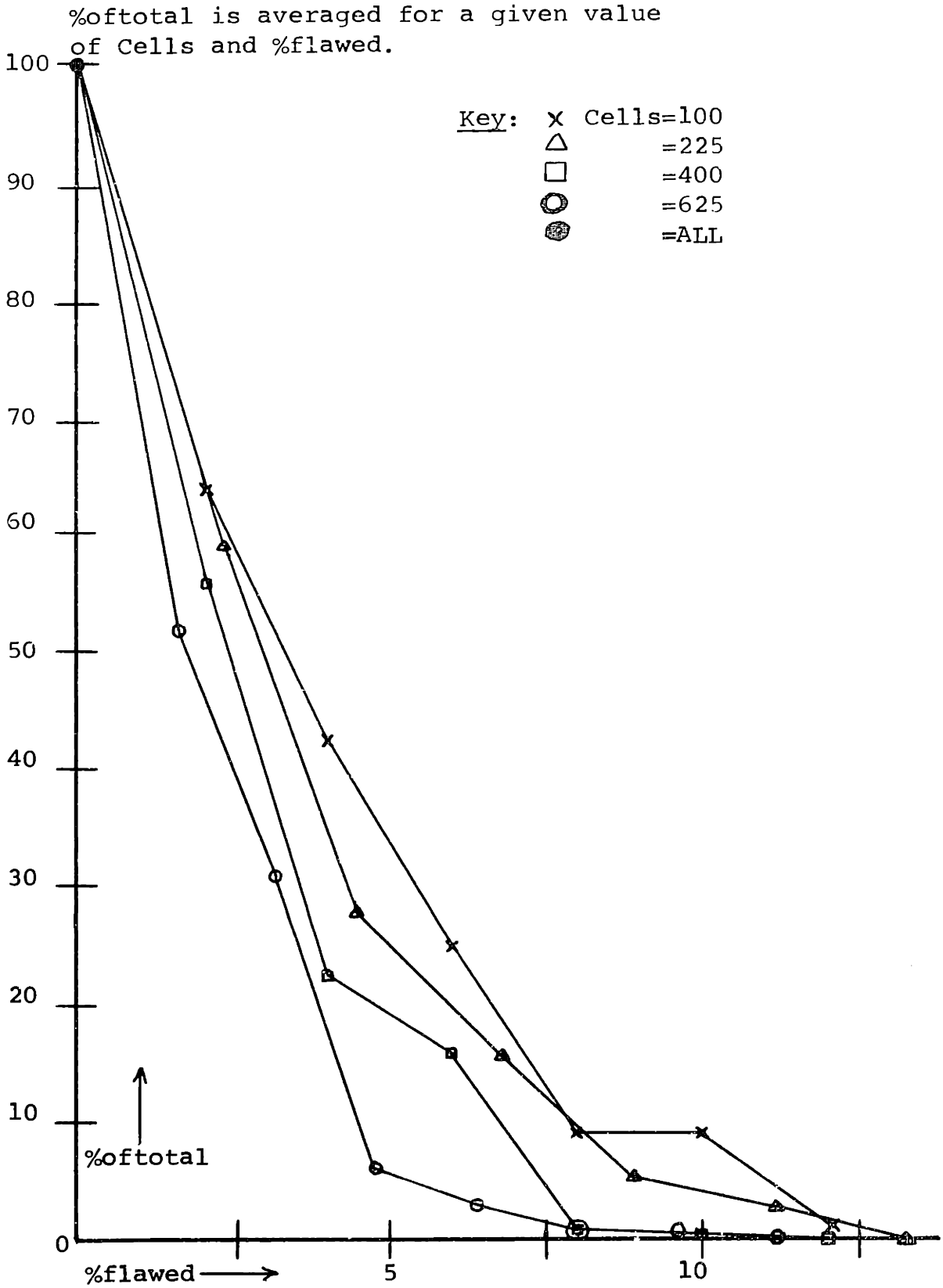
Table:

	%flawed	cells	flaws	max-grid	%oftotal	timelim	time	%oftimelim
	0	100	0	100	100	100	.005	0
	0	225	0	225	100	225	.005	0
	0	400	0	400	100	400	.005	0
	0	625	0	625	100	625	.005	0
	1.6	625	10	324	52	625	26	4
*	1.6	625	10	324	52	625	95	15
	2	100	2	64	64	100	.11	0
*	2	100	2	64	64	100	.05	0
	2	400	8	225	56	400	2.6	1
*	2	400	8	225	56	400	5	1
	2.22	225	5	121	54	225	.17	0
*	2.22	225	5	144	64	225	.16	0
	3.2	625	20	196	31	625	117	19
*	3.2	625	20	Answer not found in timelim				
	4	100	4	49	49	100	.07	0
*	4	100	4	36	36	100	.03	0
	4	400	16	81	20	400	263	66
*	4	400	16	100	25	400	231	58
	4.44	225	10	64	28	225	7.1	3
*	4.44	225	10	64	28	225	2.4	1
	4.8	625	30	36	6	625	198	32
*	4.8	625	30	Answer not found in timelim				
	6	100	6	25	25	100	.88	1
*	6	100	6	25	25	100	.85	1
	6	400	24	Answer not found in timelim				
*	6	400	24	64	16	400	24	6
	6.4	625	40	Answer not found in timelim				
*	6.4	625	40	16	3	625	233	37

Table 4.1 Results Of Twist-Repair Grid-embedding Experiments
(2nd of 2 pages)

	%flawed	cells	flaws	max-grid	%oftotal	timelim	time	%oftimelim
	6.67	225	15	36	16	225	75	33
*	6.67	225	15	36	16	225	8.7	4
	8	100	8	9	9	100	2.2	2
*	8	100	8	9	9	100	3.5	4
	8	400	32	Answer not found in timelim				
*	8	400	32	4	1	400	308	77
	8	625	50	9	1	625	406	65
*	8	625	50	Answer not found in timelim				
	8.89	225	20	9	4	225	99	44
*	8.89	225	20	16	7	225	58	26
	9.6	625	60	4	1	625	129	21
*	9.6	625	60	1	0	625	136	22
	10	100	10	9	9	100	.62	1
*	10	100	10	9	9	100	4.9	5
	10	400	40	4	1	400	307	77
*	10	400	40	1	0	400	148	37
	11.11	225	25	4	2	225	96	43
*	11.11	225	25	9	4	225	57	25
	11.2	625	70	1	0	625	64	10
*	11.2	625	70	0	0	625	7	1
	12	100	12	1	1	100	2.2	2
*	12	100	12	1	1	100	7.0	7
	12	400	48	0	0	400	84	21
*	12	400	48	0	0	400	138	35
	12.8	625	80	0	0	625	64	10
	13.34	225	30	0	0	225	81	36
*	13.34	225	30	1	0	225	32	14
	14	100	14	1	1	100	2.6	3
*	14	100	14	1	1	100	4.2	4
*	15.56	225	35	1	0	225	21	9
	16	100	16	0	0	100	3.0	3
*	16	100	16	0	0	100	4.0	4
*	17.78	225	40	0	0	225	26	12

Fig. 4.7 Graphs For Twist Repair Experiments



The smooth, consistent nature of these curves suggests the conclusions listed below:

1) For a given array size, %oftotal drops with increases in %flawed. This drop tends to be greatest for small %flawed, milder as %flawed increases, and non-existent after %oftotal reaches 0.

Consider the curve of %oftotal as a function of %flawed, for a given square array. Let E be the number of cells in a line of the array. The first flaw introduced into the array forces %oftotal to drop from 100 to $<100(E-1)^2>/E^2$, while %flawed increases from 0 to $100/E^2$. Thus the slope of the curve is 1-2E for %flawed near 0. This explains why %oftotal drops faster for larger arrays in this region.

Consider an array with several flaws. Introduction of a new flaw may not cause a decrease in %oftotal. For instance, the flaw may fall at the intersection of two adjustment lines, or in a box where a flaw had been assumed (to allow the box to interface with adjacent flawed boxes, as discussed earlier). Over the set of all flaw distributions for an array, the probability that a new flaw will not cause a decrease in %oftotal tends to increase with the number of flaws in the array. At worst, a new flaw will eliminate one row and one column of the former repaired array. If the former repaired array is smaller than the original array, i.e., if the repaired array has any flaws, at worst the new flaw decreases %oftotal less than previous "worst possible" flaws. These

considerations help explain the fact that %oftotal drops less rapidly as %flawed increases.

2) %oftotal drops faster with %flawed for larger arrays, because a given %flawed implies a higher percentage of flawed lines for a larger array.

We've already analyzed this situation for %flawed near 0. We found the negative slope of %oftotal versus %flawed was directly proportional to a square array's side length, E. As %flawed increases, the particular distribution of flaws influences %oftotal. However, it's easy to see why %oftotal tends to be smaller for larger arrays, for a given %flawed.

Given a fixed %flawed, large arrays tend to have a higher percentage of flawed lines: the number of flaws is proportional to the area, but the number of lines is proportional to the square root of the area. Consider two arrays, one with E=10 and one with E=100, at %flawed = 1. For E=10, the one flaw implies %oftotal=81. For E=100, the best possible distribution of 100 flaws puts each at one of the 100 nodes associated with 10 horizontal and 10 vertical adjustment lines. %oftotal is then 81. Most other distributions require the jogging of grid lines, and %oftotal is then usually significantly smaller than 81. One extreme occurs in the unlikely event that all 100 flaws occupy the same row or column. The array is effectively cut, so %oftotal=0.

A more general perspective provides a strong argument that moves in the direction of a proof. The flaw distributions in two arrays are *equivalent* if there's a one-to-one mapping between the flaws in the two arrays such that the following is true. If an arbitrary flaw in one array has a certain relative position with respect to the other flaws in that array, the corresponding flaw in the second array has the same relative position with respect to corresponding flaws in the second array. If one of a flaw's coordinates is X, then the relative position, with respect to that coordinate, of a flaw whose corresponding coordinate is Y depends on which of the five following, mutually exclusive, collectively exhaustive statements is true: $X+1<Y$, $X+1=Y$, $X=Y$, $X=Y+1$, $X>Y+1$.

Now consider two square arrays with different sizes, but equivalent flaw distributions. The first array has E rows, n flaws, and E-F grid rows in an optimally embedded square grid. As %flawed has climbed from 0 to $100n/E^2$, %oftotal has dropped from 100 to $100(E-F)^2/E^2$. The second, larger array has K.E rows. Since Twist Repair notices its equivalent flaw distribution, the second array's optimum square grid has K.E-F grid rows. Here %oftotal has climbed from 0 to $(K.E)^2$ as %oftotal has dropped from 100 to $100(K.E-F)^2/(K.E)^2$. The ratio of the change in %oftotal to the change in %flawed is $(2E.F-F^2)/n$ for the first array, and $(2K.E.F-F^2)/n$ for the second array; an equivalent flaw

distribution is more costly in the larger array. Any flaw distribution in an array has a corresponding, equivalent distribution in a larger array. However, the fact that not all flaw distributions in an array have an equivalent distribution in a smaller array precludes simply extension of our reasoning to a proof that, for larger arrays, %oftotal drops faster as %flawed increases. It might be possible to make such a proof by defining some sort of loosely equivalent flaw distributions.

3) For a given array size, %oftotal drops from 100 to 0 fairly smoothly as %flawed increases from 0 to a number N dependent on array size and specific flaw distribution. (For our experiments, $11.2 \leq N \leq 17.78$.) This contrasts with growth of arms, where %flawed decreases gradually and smoothly until it reaches a point where it plummets, usually for %flawed approximately equal to 28.

4) Repair efficiency is much smaller for grids than for arms.

5) For arrays with more than a few (approximately five) flaws, Twist Repair is far superior to Simple Repair. For instance, in the unstarred array with 625 total cells and 20 flawed cells, Twist Repair embedded a 14 X 14 square grid. Simple Repair embedded a 4 X 4 square grid for the same array.

6) The time to repair an array varies widely, even for a constant array-size and %flawed. The ratio of the time to repair an array to the number of cells in the array tends to be higher for larger arrays. For a

particular array, the time for repair is relatively low when there are very few flaws. As flaws are introduced, repair time tends to climb gradually, reach a peak, and then descend rapidly. This is because repair time is roughly proportional to the number of non-equivalent adjustment line placements. If an array has very few flaws, there are few non-equivalent adjustment lines. As flaws are introduced, the number of non-equivalent adjustment lines increases. Eventually an array becomes so crowded with flaws that it's difficult to find an adjustment line that doesn't include a flaw. If an adjustment line contains more than one flaw, several associated lines are required to satisfy the constraint that every flaw on an adjustment line be at the intersection of a horizontal and vertical adjustment line. This reduces the number of non-equivalent adjustment lines for very flawed arrays.

Experiments with Twist Repair suggest a new grid-embedding strategy. We notice that for a given %flawed, %oftotal tends to be substantially higher and %oftimelim significantly lower for smaller arrays. This difference becomes more significant as %flawed increases, until %flawed is so large that all grid-embedding attempts are futile. This suggests that embedding a grid in a large array should be done by breaking the array into blocks of optimum size, separated by interconnection strips. Each block is repaired via Twist Repair, and its grid outputs are connected across the interconnection strips to the grid outputs of its

neighboring blocks. In fact, experiments show that such a procedure is superior to Twist Repair for large arrays with many flaws.

A block's optimum size is determined by a tradeoff. Decreasing block-size tends to increase %ototal within each block, but it also decreases the total area devoted to blocks by increasing the number of interconnection strips. If %flawed is 0, it's pointless to waste any cells on interconnection strips; there should be one maximum-sized block. As %flawed increases, the optimum block-size decreases. Assume that the overriding factor in embedding success is %oftotal in each block. For large enough arrays, the fraction of cells used in blocks, given each block has E cells in a line, is about $(E/E+1)^2$. This number is 100/121 for E=10, and 400/441 for E=20. Using the curves of figure 4.7, this indicates that E=10 is superior to E=20 for %flawed greater than about 1.5, given our assumption. This indicates how the curves and the value of $(E/E+1)^2$ may be used to suggest an optimum block-size for a given %flawed. Experiments with Blockoff have confirmed that there is a fairly predictable, optimum block-size for a given flaw density. This fact of an optimum block-size suggests improved grid-embedding can come from breaking an array into blocks whose approximately equal size is determined by the array's flaw density. Then the simplest approach assigns identical sub-grids to all blocks of the same size. This approach is limited when some blocks have a disproportionately high number of flaws. This situation often arises with current IC slices, where flaws tend to cluster. Since a very flawed block can only contain a small grid, that block is unable to link up with all the grid

outputs of its neighboring, less flawed blocks. This limits the number of grid-rows in its row of blocks.

Before considering how Grid Repair should handle blocking off an array containing flaw clusters, it's useful to examine the repair problem more generally. It's quite clear that a heuristic approach is necessary if Repair is to efficiently repair arrays with many flaws. Twist Repair is time-consuming, especially when one wants to place a near-largest grid into a flawed array. We'd therefore like to be able to determine a priori the feasibility of a certain repair, in terms of computational difficulty and probability of success. This is particularly true if Blockoff is used to interconnect many blocks. Assume Blockoff operates on m blocks, and there are g_m satisfactory, non-equivalent sub-grids that can be embedded in block m. Let P be the product of g_n, as n varies from 1 to m. There are P combinations of sub-grids which Blockoff may try to interconnect to form an embedded grid. If a high percentage of these P combinations are consistent with the desired grid, Blockoff may quickly succeed. At the other extreme, Blockoff would spend a time proportional to P in vainly considering each of the combinations.

Happily, Repair may use a rather simple heuristic approach to reduce repair time. Let F be a success function which estimates the grid-size that Repair can reasonably expect to embed in a given array. Most simply, F is a function of a square array's size and its flaw density. F can be refined in various ways we'll consider. For instance, an input to F could state the probability F's estimate is not

an over-estimate. If non-square grids and blocks are considered, F can depend on their specified shapes. It's reasonable to obtain F experimentally, because of the monotonic nature of F. For instance, we've observed that F's output decreases as an array's dimension or flaw density increases. This monotonicity enables us to estimate F by experimentally determining some of its key values, and interpolating to find its other values. F is Repair's heuristic guide.

Now consider the following procedure adapted to embedding a grid containing R rows and C columns in an array that may contain flaw-clusters. Repair uses F to break the flawed array into approximately equal blocks whose size depends on the array's dimensions and average flaw density. F suggests the block size that is expected to yield the maximum embedded grid. Repair then considers each line of blocks, associating with each line a number equal to the number of lines F associates with the most flawed block in the line. Thus Repair recognizes the difficulty of snaking a grid's rows or columns through a cluster of faulty cells. Repair finds the sum S of all the numbers associated with the row lines. If $S < R$, embedding the specified grid will be difficult or impossible; Repair's action depends on whether it's willing to spend a lot of computation on what is probably a vain effort. (This decision can be made implicit if a success-probability parameter, like the one we've discussed, is passed to F.) If $S = K.R$, where K is greater than or equal to 1, Repair multiplies each row number by about $1/K$; so that all the row numbers sum to R. An analogous procedure is applied to the columns of blocks. If Repair decides to call Blockoff, Repair has heuristically

determined the size of the sub-grid assigned to each block. Thus F facilitates a heuristic for guessing whether a given repair is feasible and, if so, the assignment given to Blockoff.

We've noted that grid-embedding is the most difficult repair problem in a checkerboard array. The major practical importance of a repair procedure geared toward grid-embedding is its generality; if Repair passes Construct a flawed array specified as a large grid, Construct may embed in the flawed array any machine whose essential network is a sub-network of the grid's essential network. However, this generality diminishes the efficiency with which a non-grid machine is embedded in a flawed array. We've seen this for the extreme case of arm-embedding.

The Blockoff procedure we've written is general enough to accept an essential network containing rectangular sub-grids with specified wires between adjacent sub-grids. It's easy to see why noticing limited communication paths between a machine's high-relcon regions promotes efficiency: Blockoff operates under fewer constraints.

Table 4.2 and figures 4.8 and 4.9 summarize a series of experiments that begins to explore how block-size and missing grid links affect embedding high-relcon machines. Table 4.2 summarizes the data from the experiments, and figure 4.8 and 4.9 give Blockoff-produced pictures of repaired arrays.

The experiments all used %flawed = 5, which figure 4.7's curves suggest is a region where block-size of 10x10 is better than block-size of 20x20.

Table 4.2 Experiments With Three Different Blockoff Goals

Blockoff operated on arrays with 5% flawed cells. For a given flawed array, at most three different Blockoff experiments were performed. In each experiment, Blockoff was asked to place the same-sized square grid in each of an array's approximately equal-sized blocks. Blockoff's best result is shown for each entry. Unless otherwise noted, Blockoff found it impossible to achieve a better result, given the constraints. In 10-connect, Blockoff allocated 10 x 10 blocks for each grid, and tried to connect the small grids into one large grid. In 10-noconnect, Blockoff allocated 10 x 10 blocks for each grid, but did not interconnect the small grids. In 20-connect, Blockoff allocated 20 x 20 blocks for each grid, and interconnected block grids into one large grid. Subsequent figures contain the repaired 20 x 20 and 40 x 40 arrays produced by Blockoff.

	Array	Experiment	Best Blockoff Result	Time (seconds)
	10x10	10-connect	1 6x6 grid	2.3
	20x20	10-connect	4 4x4 grids = 8x8	47.0
	20x20	10-noconnect	4 4x4 grids	9.2
	20x20	20-connect	1 8x8 grid	63.1
+	40x40	10-connect	16 3x3 grids = 12x12	58.7
	40x40	10-noconnect	16 4x4 grids	32.2
	40x40	20-connect	4 5x5 grids = 10x10	832.
*	80x80	10-connect	64 2x2 grids = 16x16	963.
	80x80	10-noconnect	64 3x3 grids	188.
!	80x80	20-connect	-	

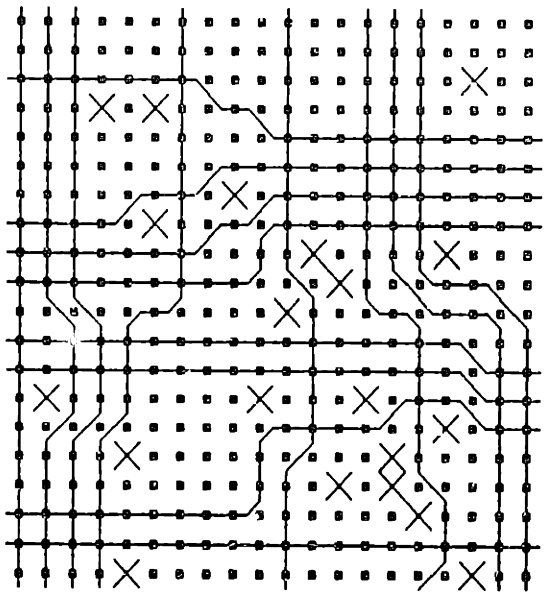
+ When asked to put 16 4x4 grids in this flawed array, Blockoff was still thinking after 45 minutes. Then we interrupted and terminated Blockoff.

* When asked to put 64 3x3 grids in this flawed array, Blockoff was still thinking after 27 minutes. Then we interrupted and terminated Blockoff.

! When asked to put 16 3x3 or 4x4 grids in this flawed array, Blockoff was still thinking after 22 minutes and 9.5 minutes, respectively. Then we interrupted and terminated Blockoff.

Fig. 4.8 Blockoff's Repair Of 20x20 Array With 5% Flawed Cells
(1st of 2 pages)

A) 10-connect embeds four 4x4 interconnected grids



B) 10-noconnect embeds four 4x4 unconnected grids

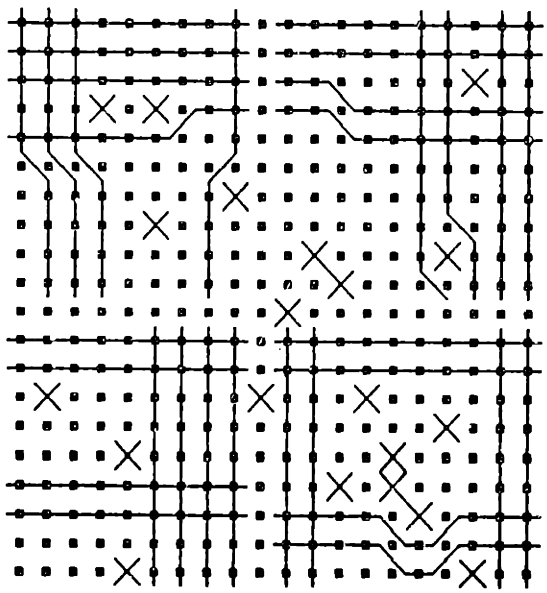


Fig. 4.8 Blockoff's Repair Of 20x20 Array With 5% Flawed Cells
(2nd of 2 pages)

C) 20-connect embeds one 8x8 grid

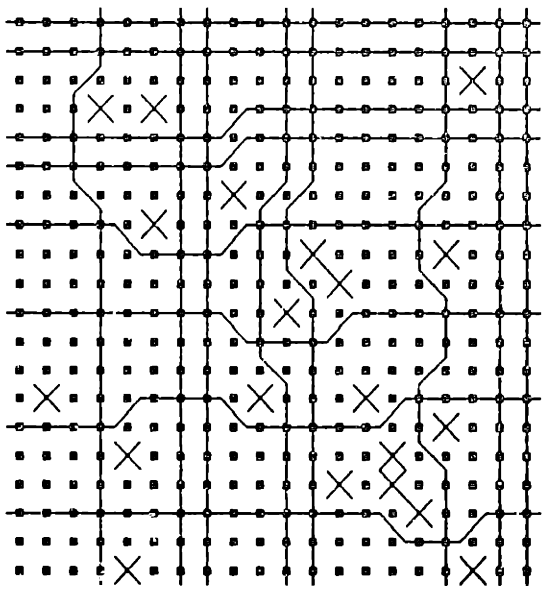


Fig. 4.9 Blockoff's Repair Of A 40x40 Array With 5% Flawed Cells
(1st of 4 pages)

A) 10-connect embeds sixteen 3x3 interconnected grids

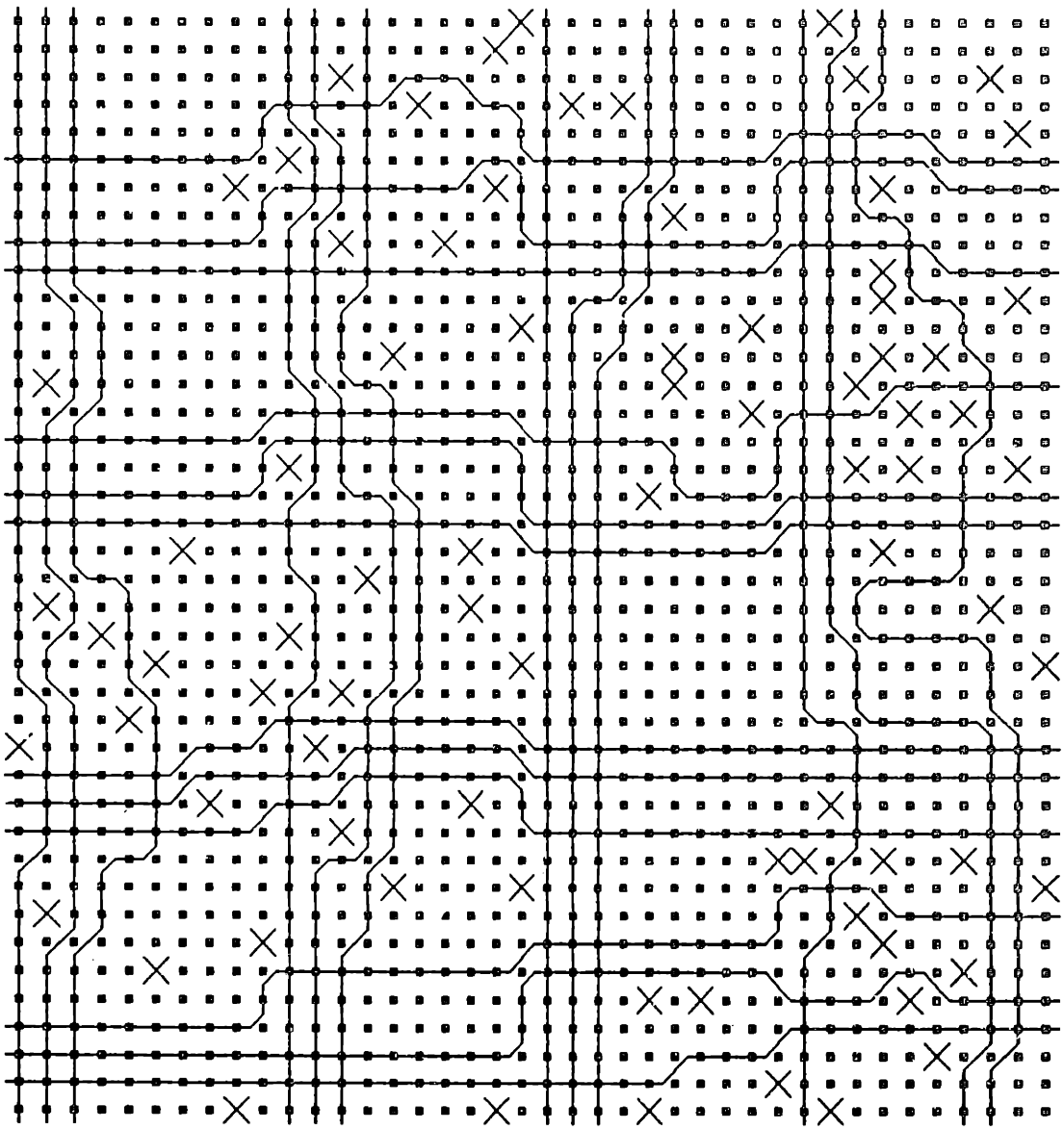


Fig. 4.9 Blockoff's Repair Of 40x40 Array With 5% Flawed Cells
(2nd of 4 pages)

B) 10-noconnect embeds sixteen 4x4 unconnected grids

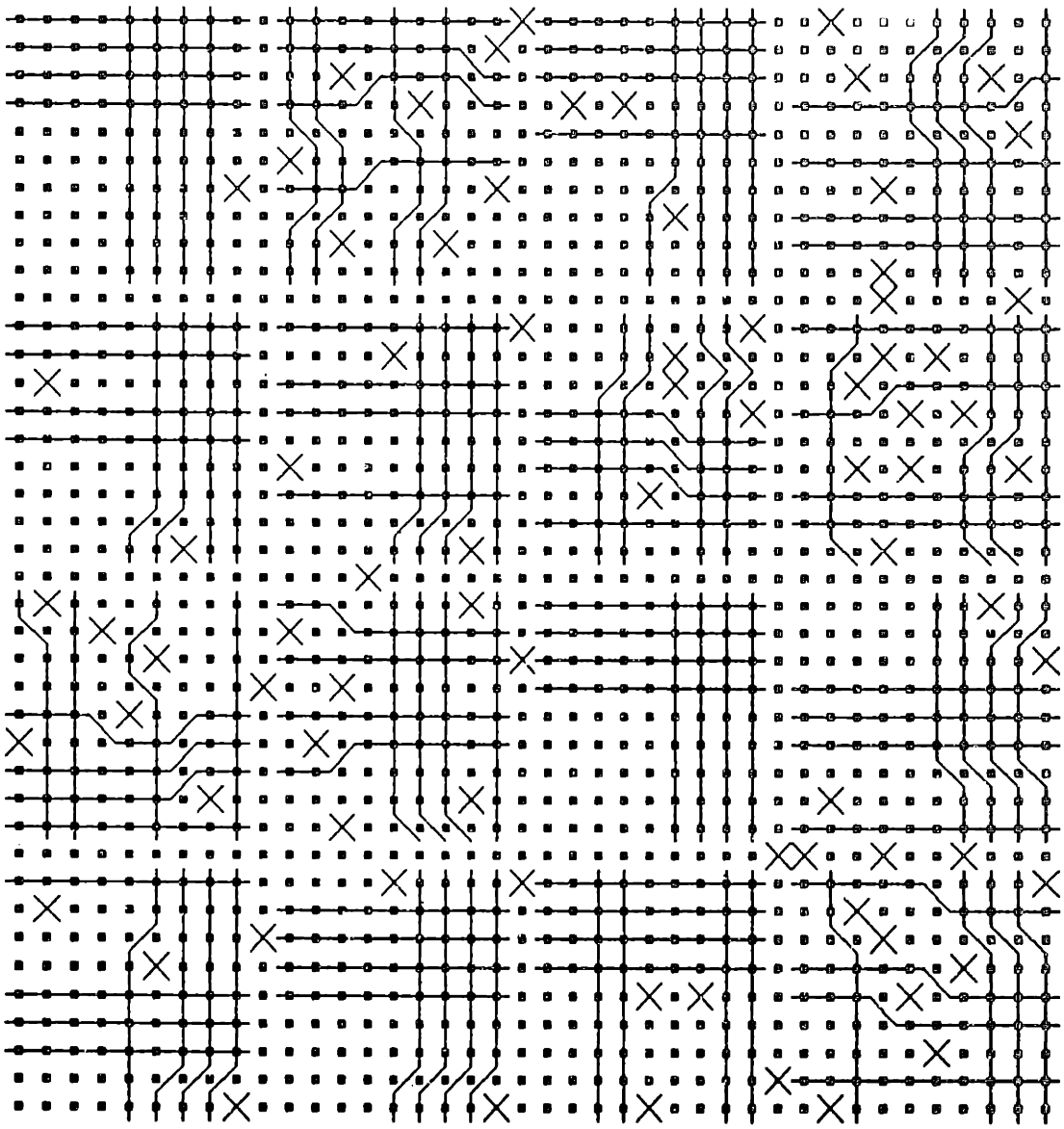


Fig. 4.9 Blockoff's Repair Of 40x40 Array With 5% Flawed Cells
(3rd of 4 pages)

C) 20-connect embeds four 5x5 interconnected grids

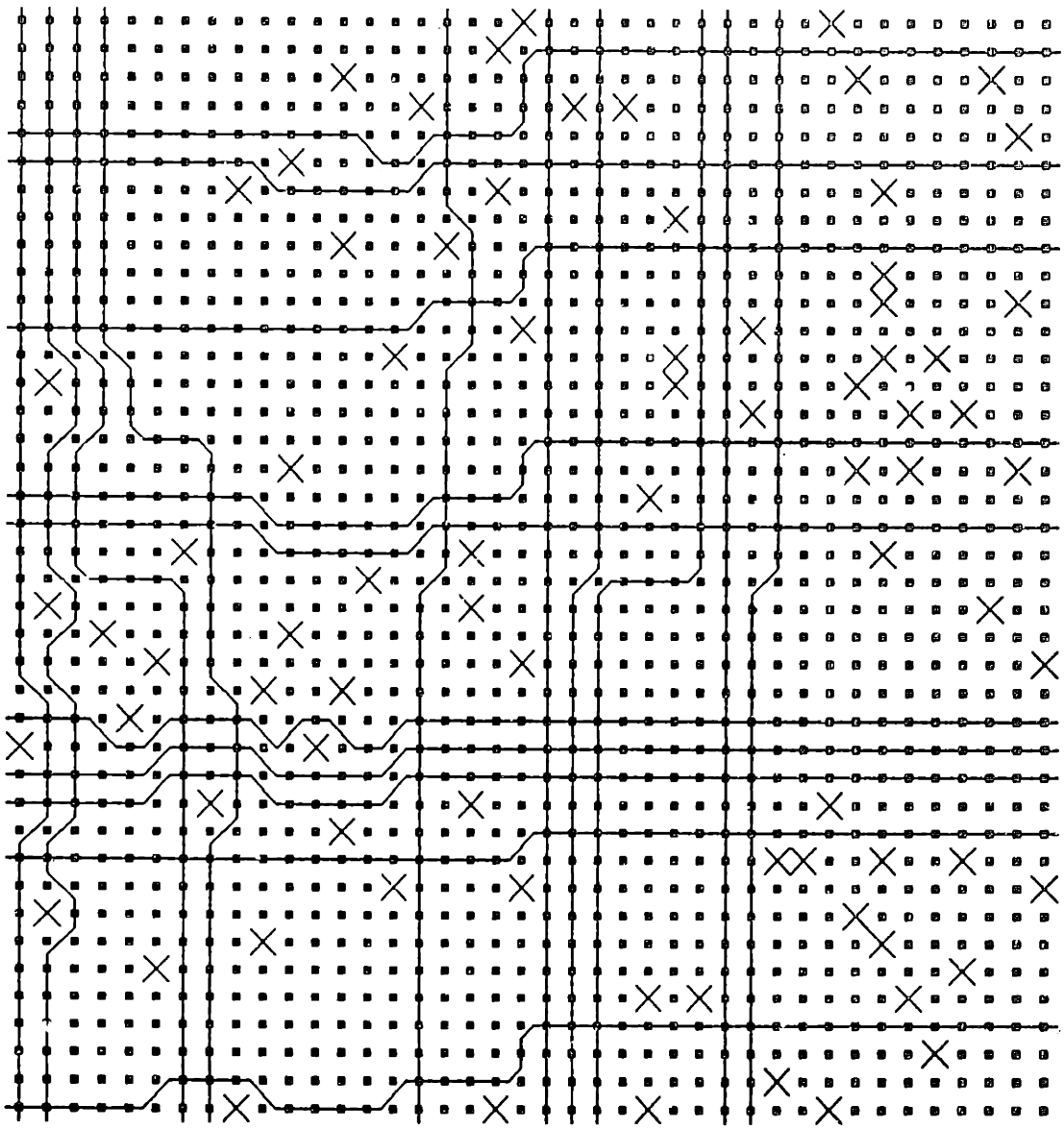
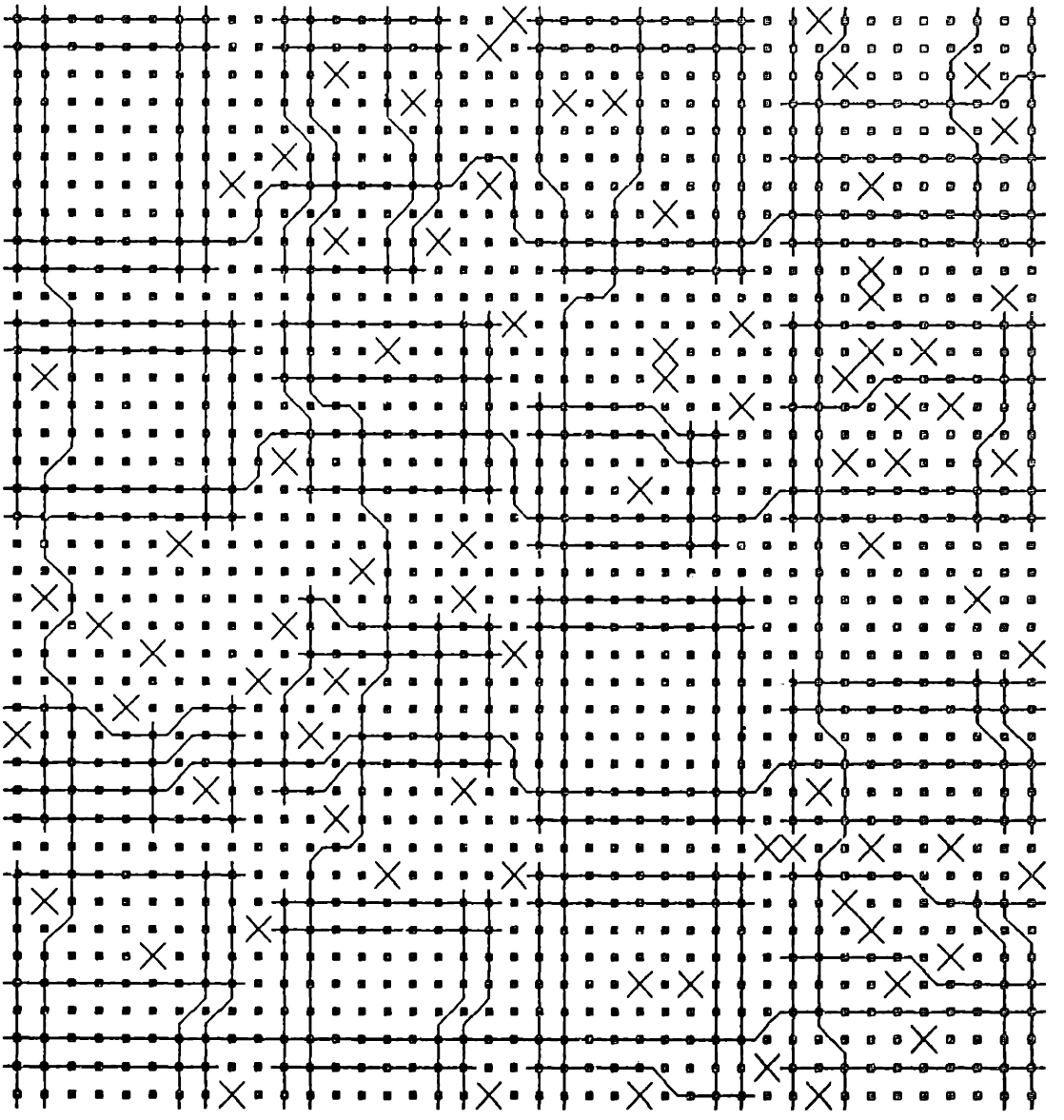


Fig. 4.9 Blockoff's Repair of 40x40 Array With 5% Flawed Cells
(4th of 4 pages)

D) Only one link between adjacent 4x4 grids



Indeed, 20-connect always took substantially longer to repair an array than did 10-connect. Furthermore, 20-connect never embedded a larger machine than 10-connect, and sometimes embedded a smaller machine. 10-noconnect always embedded at least as many essential nodes as the others, because 10-noconnect works on a grid with links missing. We commanded Blockoff to place the same square grid in each of an array's blocks, because we didn't want to help Blockoff by implicitly telling it the location of flaw clusters. This constraint on Blockoff limited its performance; this explains why we can see ways to snake extra grid rows and columns through the flawed arrays. Figure 4.8.A indicates that the lower-right block of the 20 x 20 flawed array limited the performance of 10-connect and 10-noconnect. Similarly, figure 4.9.B shows that certain very flawed blocks limited Blockoff's performance. This argues for use of the success heuristic suggested earlier. Figure 4.9 indicates that Blockoff's performance diminished as more links were introduced between sub-grids.

Comparing the graphs for Twist Repair experiments with table 4.2 shows Blockoff's superiority to Twist Repair as a flawed array's size increases. For %flawed equal 5, Twist Repair achieved a %oftotal of 6 for a 25 x 25 array. This indicates that for 40 x 40 and 80 x 80 arrays, Twist Repair would have achieved %oftotal substantially under 6. For %flawed equal 5, Blockoff used 10-connect to achieve a %oftotal of 9 for a 40 x 40 array, and %oftotal greater than or equal to 4 for a 80 x 80 array. This and other comparisons we've made of Blockoff and Twist Repair indicate Blockoff is superior when %flawed remains

constant as an array's size increases.

We wish we could offer more experimental results from Repair. However, Repair's large computation-time demands have made further experiments unfeasible.

We now re-consider the improved Repair procedure. We see that it is oriented toward embedding a machine abstractly described as interconnected rectangular sub-grids. Repair may use a heuristic approach to decide what part of a flawed array should accept each sub-grid. After making such an assignment, Repair calls Blockoff. Blockoff may use a heuristic like Repair's to decide how to embed each sub-grid. If the sub-grid is sufficiently small, Twist Repair is appropriate. Otherwise Blockoff may break the sub-grid into blocks, and present Repair with each sub-grid. That is, Repair may be recursive. In any case, an array is eventually broken into blocks repaired by Twist Repair, and interconnected by Blockoff.

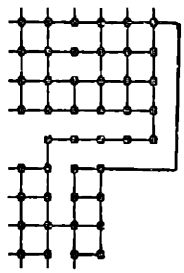
We've purposely ignored discussing an embedded machine's interconnections to other machines, either in or out of its array. Tradeoffs relating to this question are analogous to those for arm-embedding. Blockoff may be easily adapted to accepting inputs describing which of the cells at a machine's periphery carry the machine's inputs and outputs. Handling this is like handling the interface between linked sub-grids. In each case, a particular cell (for instance, one with a lead to the extra-array world) should connect to a particular essential cell.

One can envision further levels of Repair sophistication, whose goal is

increased embedding efficiency. Choice of sophistication level depends on the character of expected repair problems. For instance, very large arrays might benefit by interconnection strips wider than one line between large blocks. If substantial sections of an essential machine were likely to have essential cells with few essential neighbors (as the General processor-tester-repairer did), efficiencies would result from special handling of these sections. Indeed, perfect machines should probably be designed in a modular fashion, with relatively few communication paths between modules. The need to limit inter-module communication paths is already recognized in the design of conventional systems.

We briefly sketch a promising repair technique for such high-relcon machines. An essential network is categorized in the following way. Each essential node with three or four essential neighbors is associated with some rectangular *high-relcon block* in a compact Blockoff-compatible way that's been discussed. Those wires and essential cells with one or two essential neighbors that are not in a high-relcon block are associated with *low-relcon blocks* (see figure 4.10). A straight horizontal or vertical line through an essential network passes through at least one high-relcon or low-relcon block. That block which the Success Heuristic F estimates as least efficiently repaired, given the flawed array's average flaw density, determines how many flawed array lines should be allocated for an essential network line. For instance, the expected embedding efficiency for the large high-relcon block dictates the number of flawed array columns devoted to perfect array columns 0 through 5. The relatively high

Fig. 4.11 Blocking Off A High-relcon Essential Network



The lower-left corner of the essential network is node (0 0). A reasonable way to block off the network makes three high-relcon blocks with corner-lists of <(0 5)(5 5)(0 8)(5 8)>, <(0 0)(1 0)(0 3)(1 3)>, and <(2 0)(3 0)(2 3)(3 3)>. The first high-relcon block dominates horizontal allocation of nodes whose horizontal coordinates are 0 through 5, and dominates vertical allocation of nodes whose vertical coordinates are 5 through 8. The other two high-relcon blocks dominate vertical allocation of nodes whose vertical coordinates are 0 through 3. Low-relcon blocks dominate allocation for other nodes.

embedding efficiency of transmission links dictates a lower multiple of flawed array columns devoted to column 6. Thus Repair uses the success heuristic to estimate whether a repair will succeed, and to get a rough estimate of how to allocate flawed array space. Repair may then adjust its initial estimates by considering the actual number of flaws in each allocated block. Repair then uses Blockoff to repair the high-relcon blocks. Given a success here, Repair calls a procedure devoted to all the low-relcon essential cells and wires that aren't in a high-relcon block. The main point of this approach is to carefully associate essential cells with flawed array regions. The small amount of time devoted to heuristic care guards against vain, exhaustive, computationally-expensive attempts by Blockoff to repair an array blocked off in an unrepairable way.

If Repair uses details of a machine's essential network to increase embedding efficiency, Repair needs a description of that network. In the least sophisticated case, a designer could specify that network to Repair; we've done this in our experiments with Blockoff. However, it is fairly easy to write a procedure which abstracts a machine's essential network from its description as an embedded machine. The procedure "works back" from the embedded machine's outputs to find the essential cells and wires of the machine. The resulting essential network could be blocked off by analysis of the location of high-relcon regions. Straight lines through the network that yielded a low density of links would indicate reasonable boundaries between high-relcon regions.

We've discussed an effective Repair procedure, and actually written

and analyzed fundamental components of this procedure. Nevertheless, repair of high-relcon machines remains a largely unexplored area. Some programs we've sketched remain to be implemented. Perhaps better methods of repair can be found. More experiments would enable a better understanding of the heuristic success-function's nature. Interesting theoretical questions remain. Consider the curve of the expected width of a square embedded grid versus the width of a square flawed array, for some low, non-zero flaw density. Is there a repair procedure such that this curve is monotonically increasing? Is there a repair procedure such that the curve is above some positive-sloped straight line for very large arrays? Can you produce such a procedure, or prove there isn't one? This is an important question, because its answer tells us the expected size limits on grid machines embedded in arrays of a given flaw density. This helps us determine the expected size limits of high-relcon machines that aren't grids.

Section 4.5: Construct

Construct accepts three information inputs which dictate how Construct loads cells in a flawed array. These inputs are:

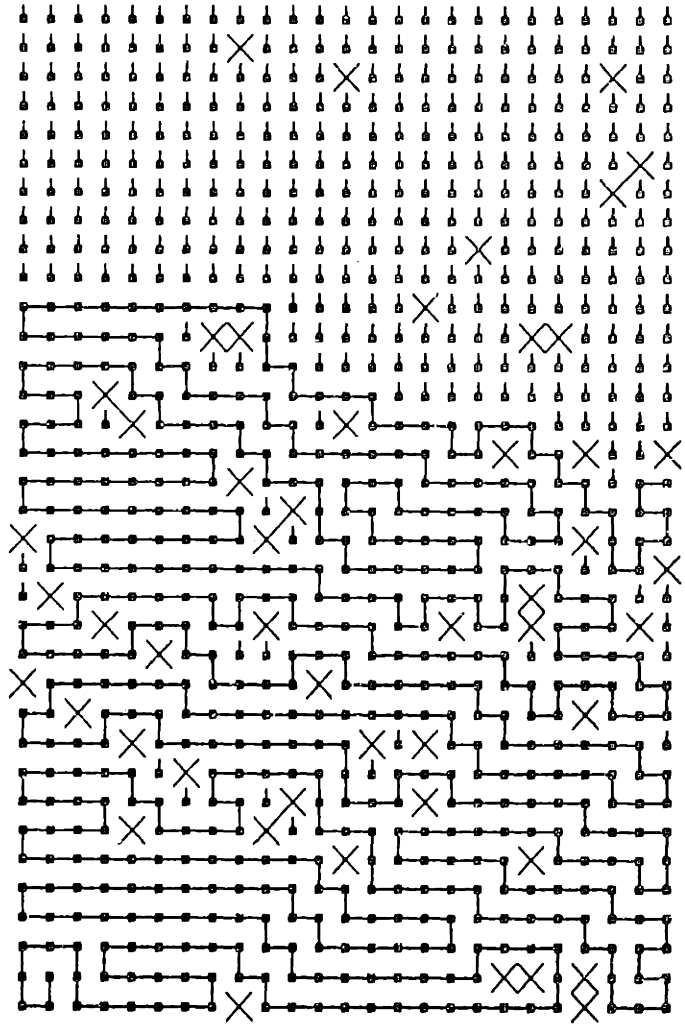
1) a description of an essential machine stating essential states and associated wiring;

2) a description of a repaired array, in which each cell's processing layer function is in one of the four categories we've mentioned - flawed, essential cell, particular non-branching transmission state, or unused good cell in arbitrary state; and

3) a description of the repaired array stating the side-sets successfully activated and de-activated by Test's loader.

We've noted that Construct's precise nature depends on Repair's generality. In any case, Construct is very simple. First Construct "mentally" maps a machine's essential cells into a repaired array's essential nodes. Then Construct extends a loading arm into the flawed array, possibly touching all good cells and at least touching and properly setting all the cells acting as essential cells or wires between essential cells. The loading arm's base may be any cell with access to the cells that must be set. For instance, any of the cells of an embedded machine would be an acceptable base. Setting the proper cells is even easier than growing a long arm into an array. Construct knows the location of flawed cells, and may extend, retract, or move its arm through side-sets Test successfully activated in

Fig. 4.10 Result Of An Experiment Showing Construct's Capabilty



Construct was asked to touch all unflawed cells, using a flexible loading arm with its base at (1 1). The picture shows the state of Construct's arm when Construct completed its task, after 65 seconds. All ↓ cells are cells that have been touched by Construct's loading arm, but are no longer part of that arm. Construct finished in 65 seconds for this 25 x 35 array with 50 flaws.

any way consistent with touching all the proper cells. Figure 4.11 shows the result of a simulation demonstrating Construct's ability to perform its task. The simulating procedure moved its arm in a flawed array. All cells were initially in either the X (flawed) or G (good) state. For simplicity, it was assumed that all accessible side-sets of good cells could be successfully activated and de-activated. The arm moved around in the array until all touchable cells were touched. (This is doing more than is necessary.) The figure shows the state of the loading arm when Construct succeeded. Of course, Construct could completely plan its loading strategy via such a simulation before actually extending its arm into an array.

Section 4.6: Other Considerations In Realizing High-relcon Machines

We've considered the basic issues of testing, construction, and repair for high-relcon machines in previous sectons. Now we turn to less fundamental, but important, aspects of high-relcon machines. We considered production and marketing issues for arm machines in section 3.4. We suggested ways to satisfy constraints imposed by the need for adequate array-access ports (chapter 3 called them "arm bases"), the need for proper handling of shared power lines, and volatility. These constraints have obvious analogs in high-relcon machines. Because satisfaction of these constraints is also obviously analogous, we need not consider these constraints further. Instead we concentrate on considerations peculiar to high-relcon machines.

All the testing procedures we've discussed assume independence of cell behavior. Gradual growth of an arm machine involves concurrent tests of an individual cell and its associated machine. As soon as the last cell of an arm has been tested, the arm is complete and tested. On the other hand, high-relcon cells are independently tested before they're included in an embedded machine. Testing an embedded machine, or its modules, checks our independence assumptions. An embedded machine may be tested like any digital machine, via its inputs and outputs. Furthermore, test-link capability provides testability to high-relcon array machines that's not available in ordinary digital machines. Test links may connect an embedded machine's module with a test machine, to allow independent testing of that module. A test link, terminated by a transmission-branch cell, may be used

as a probe which, at a given time, monitors the signals on a wire in an embedded machine. After test links are used for module testing or probing, they can be withdrawn. Of course, this assumes that the test arms do not affect the operation of the eventually embedded machine; this is a safer assumption than a simple cell-independence assumption.

The use of cells as wires in high-relcon machines necessitates special considerations. In most hard-wired machines, it's safe to disregard the delay through wires; but this assumption is usually not valid in high-essential machines because the delay through a wire cell is close to the delay through some other cell. A pair of essential neighbors may be linked by different-length wires in different flawed arrays. Wire delays consequently decrease an embedded machine's maximum operation speed. Furthermore, they compound the "critical race" problem, thereby making array machine designs more constrained than non-array designs. A synchronous high-relcon machine must be clocked slowly enough to allow for the delay through embedded wires. Other conventional techniques for solving timing problems, such as ready-acknowledge signalling, may be employed where needed for communication between modules in embedded machines.

Array machines compensate for inherent limitations by providing added capabilities, including automation-compatibility. We've seen that a simple array facilitates testing and repair by its iterative nature, and by the fact that test and repair facilities are built into a cell. An array's simple structure also facilitates computer-aided design. A designer could specify a machine as a perfect

embedded machine with timing constraints on its cells. A simple program could check that an envisioned embedding satisfied these constraints. A more sophisticated program could "compile" a machine's high-level-language specification into an acceptable array-embedded machine; this is difficult, but easier than analogous computer-aided design in a less regular environment.

Section 4.7: High-relcon Machine Applications

We've discussed our approach to the most difficult testing and repair processes for a checkerboard array - treatment of high-relcon machines. Abstract description of an essential machine as an essential network focuses on the properties of a machine that are important to test and repair. We've shown that high-relcon machines have higher testing and repair costs than arm machines. We've also shown that, even for high-relcon machines, our cellular approach offers major integration, test, and maintenance advantages relative to other methods for system implementation. In this section we consider applications merits of high-relcon machines, relative to arm machines and non-array machines. We discuss the General cell as one which enables realization of the benefits of high-relcon machines.

Chapter 2 discussed the general advantages of cellular arrays, and argued for our array approach. This approach attempts to meet system design, production, and maintenance needs through standard, high-volume, flexible, automation-oriented modules - cells and associated programs. We compared our approach to other, less constrained approaches. Chapter 3 discussed balanced arm machines using our approach. Earlier sections of this chapter compared testing and repair processes for arm and high-relcon machines. This section highlights performance features that haven't been sufficiently covered.

Because the communication paths between cells in a high-relcon machine are less constrained than those in an arm machine, a high-relcon machine provides

speed and flexibility advantages in certain applications. The suitability of a particular type of essential machine depends on the utility of a given degree of direct, simultaneous inter-cell communication for that machine. A serial-in, serial-out shift-register is well-suited to arm machines, because each stage of the register communicates directly with at most two other stages. Tree machines are well-suited to machines which have only one section addressed at a given time; therefore random-access and some other memories are well-suited to realization as tree machines. In an arm machine, essential neighbors are always in adjacent cells. In a grid embedded in a flawed array, essential neighbors aren't necessarily in adjacent cells; this diminishes the speed advantage of the embedded grid machine. High-relcon machine realization is particularly suited to machines composed of modules which communicate different information to three or more other modules at the same time. Such machines might require complex cells to even awkwardly share the communication paths available in tree or arm machines. For instance, building a processor as an arm or tree machine would probably require complex cells, and suffer from low parallelism. Forcing all a machine's extra-array leads to connect to one cell makes realization of certain machines very difficult. Thus high-relcon arrays provide additional information paths, but require higher testing and repair costs when they are used for high-relcon machines. A high-relcon array is most suited to machines which exploit the array's available information paths, such as the processor-tester-repairer built of General cells.

High-relcon arrays, such as General arrays, offer major advantages as

peripheral equipment in a computer system. The computer system offers the array a non-volatile Array Programmer. The array offers a reliable, inexpensive, programmable, high-speed processing capability.

For many machine tasks, the General array is a correct compromise between a single-sequence computer and a special hard-wired machine. A typical computer's performance advantages include computational power, flexibility, and easy programmability. Its major disadvantage is slow performance relative to hard-wired machines. The single-sequence computer processes only one instruction at a time, with each instruction taking many gate-delays. The ameliorating parallelism in some instructions is often wasted. For instance, an algorithm that only operates on 1-bit words still uses an AND that operates on larger words. The conventional computer is particularly ill-suited to irregular or high-frequency real-time applications; handling incoming signals through interrupts is particularly time-consuming and tricky. Computers are so clumsy at real-time applications that they often rely on a hard-wired machine to buffer incoming signals; this machine continuously monitors, collects, and pre-processes incoming data. Many applications are more suited to a special-purpose machine, which offers higher speed. Disadvantages of such a machine include high setup times and high setup costs, especially if these costs are not distributed over a large number of machines. Testing and repair of these machines can be particularly difficult and costly.

A peripheral array, such as the General array, is a compromise between

the performances of these two most common machine approaches. The array may be quickly and easily programmed to one of a large set of embedded machines. For instance, a processing-intensive problem could be solved in an array which interrupted the computer's processor when it had solved the problem. Alternately, a General array could be used as a processor component tailored to the requirements of a particular processing task. The array provides a high degree of potential parallelism. Basic cell operations, those that occur in the cell's function states, are faster than basic computer operations, but slower than the basic operations of a special-purpose hard-wired machine. Like a hard-wired machine, a General machine can continuously monitor and process incoming signals. Furthermore, our arrays have the added advantages of low cost and easy, automatic maintenance.

Of course, an array's suitability depends on its intended application area. The General array is oriented toward narrow data words; there is only one processing input in each of a cell's side-sets. Parallel algorithms, especially those amenable to two-dimensional array solution, are particularly appropriate for the General array. Many physical problems, such as temperature distribution on a plate, are consistent with such an array solution. Such an array might benefit from larger processing side-sets to accomodate numbers representing one of a wide range of temperatures. However, such a macro cell with large side-sets could be built of General cells. The General array is good at logic simulation. Real-time applications which would otherwise require an expensive, low-volume special

machine are often suited to a high-relcon array.

An array's utility depends on its size. This partially explains our interest in array repair. One way to increase an array's size is to interconnect arrays. If one's objective is a large array with a checkerboard array's interconnection network, one must currently make many interconnections between neighboring arrays. This is fairly costly, even if one uses a special interconnect-array circuit board, because of the many IC leads involved. Our approach reduces the need for many leads between sub-arrays by relying on testing and loading arms, and by Repair's block orientation. This block orientation recognizes that most machines are composed of modules, and have few communication paths between the modules.

A high-relcon array may also replace special-purpose machines in a computer system. Here the array is most appropriate when computer-maintainability is important.

The ability of an array-embedded machine to test, program, and repair its cellular environment is particularly attractive. Such a machine can form its cellular environment into machines appropriate to a given application at a given time. Two or more machines like the one we've designed can achieve high relibability by monitoring and repairing each other. Each machine is embedded in a sea of spare parts, cells, with enough cells to support many cell failures. When one machine notices that the other has failed, it re-tests the other's environment before embedding a new, perfect machine. With three array-embedded machines,

a first good machine may continue normal operation while the second good one repairs the faulty machine.

It's amusing to consider the unlikely event of a form of "array cancer", in which a faulty machine attempted to wipe out a properly working machine. Each embedded machine could guard against inappropriate attack with test arms for noticing attack, and a loader arm for fighting the attack. A machine could be programmed so that both its defense and attack programs required proper use of all the machine's processor sections. With the right attack and defense programs, a perfect machine should then be able to dominate a malicious, faulty machine.

If the General array is non-volatile or easily backed up by a power supply or loading source, it may be mass-produced and program-customized to provide inexpensive, low-volume machines inappropriate to microprocessor realization. Sometimes added advantages come from the machine's nature as a standard part that can be tested, programmed, and repaired through limited communication with a standard machine. Our arrays can even be repaired by a remote machine connected to an array via communication links.

CHAPTER 5: TREE MACHINES

This chapter discusses embedded machines with a particularly simple nature. All cells of a tree machine are effectively linked to a common input bus and common output bus. Each cell is a balanced, essential cell whose function state includes a unique name. At any given time, only one cell may transmit its information out of the embedded machine. Examples of such a machine are paged random-access and track-addressed sequential-access memories, with one cell per page or track. The embedded machine's simplicity means that a cell's processing layer can be designed so that all tree-like relcon networks with a given number of nodes may correspond to equivalent embedded machines. This allows efficient use of good cells in a flawed array, because an embedded machine can incorporate any good cell linked to its input-output (tree base) cell by some path of good cells. Because one form of tree is an arm, a flawed array embedding a tree machine can be repaired at least as efficiently as a corresponding array embedding an arm machine. Furthermore most large, flawed arrays may be repaired to embed random-access memories or other tree machines with average access time proportional to the square-root of the number of cells in the array.

For specificity, we consider a paged random-access memory implementation with the following characteristics. The RAM has 2^p pages, or cells, with 2^w words of length L in a RAM on each page. Command and output words are handled serially. The RAM has two input lines called Klock and Command, and one output line called Return. When the RAM is ready to receive a command

specifying a "Read" or "Write" operation, the Command stream Klocked into the RAM specifies the following:

1) a p-bit page address which selects the one cell of the embedded machine with the identical name stored as p function-specification state bits;

2) a w-bit address selecting a particular word within the page;

3) a Read/write bit specifying either a "Read" or "Write" operation; and

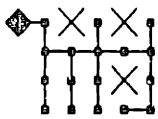
4) if the command is a "Write", the L-bit word to be written.

If the command is "Read", the L Klock pulses after the command Klock the selected word out of the embedded machine.

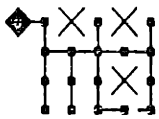
Since the paged-RAM cell's loader is the same loader detailed previously, we focus on a balanced processing mechanism and associated function-specification state bits for cells in a checkerboard array. Each of a cell's side-sets has one Insel input line specifiying whether that side-set is selected to send Klock and Command information directly into the cell. A cell in a working embedded machine has only one of its Insel inputs high. The Insel-selected Klock and Command information is broadcast to the cell's neighbors via the cell's Klock and Command Outputs. A cell's broadcast Return output is that cell's RAM Output line if the cell has been addressed; otherwise the Return output is the OR of from zero to three Return inputs selected by four Retsel function-specification state bits. Each Retsel state bit corresponds to one of a cell's side-sets. Besides

Fig. 5.1 Relcon Networks For RAMs In Identical Flawed Arrays

A) Relcon network for one embedded RAM



B) An embedded RAM with better access time than A



Commands input to a tree's base flow to the tips of the tree. Every link that carries an input command in one direction carries a Return in the opposite direction. An addressed cell's Output information successfully reaches the embedded machine's output because the Output is ORed with 0s as it flows to the embedded machine's output. Maximum access time is minimized by minimizing the longest path between a tree-tip and the tree's base. Machine B's access time is better than A's because A has a circuitous path to node (3 0). The best expected access time results from placement of a tree's base at the center of its associated array.

determining whether a cell's left Return input is selected to be ORed, the "left" Retsel state bit is also the cell's left Insel output. A corresponding statement is true for a cell's "right", "up", and "down" Retsel state bits. Thus an embedded machine is organized so that cell A accepts a Return input from cell B if and only if cell B accepts Klock and Command inputs from cell A. Input command information enters a cell from one of its neighbors, and is accepted by up to three of its other neighbors. Hence a given RAM with c cells can be realized by any tree-like relcon network of c good cells consistent with the limits imposed by an array's interconnection network. Figure 5.1 shows relcon networks for two equivalent machines in identical flawed arrays. The machines differ only in their access time.

In a checkerboard array, access time is minimized by placing a tree's base cell at the center of a square region of cells; one diagonal of the square is a row, the other is a column, and the diagonals cross at the tree base cell. When such a strategy is used, the expected time required to send information to or from the tip of a tree embedded in a flawed array is proportional to the square-root of the number of tree cells. Expected access time is therefore proportional to the square-root of the number of cells in large tree machines. In an n-dimensional array, this expected access time is proportional to the "n"th root of the number of tree cells when the tree's base is at the center of a cube or hyper-cube.

Since the cell we've discussed handles information serially, it needs a counter and associated circuitry to coordinate activity. This counter is initialized by the loader.

Techniques used for improving performance of conventional RAMs, such as use of parity bits, are applicable to this approach. The RAM in each cell is identical to conventional RAMs. The fact that a loading arm can rapidly shuffle the names of cells in a machine without disturbing their RAM contents may be useful for some systems' memory management. If a simple paging system is willing to effectively construct a page table by shuffling the names of memory cells, a special page table and its associated delays are not required.

Test and repair of flawed arrays embedding tree machines is similar to test and repair for arm machines. A tree machine is grown cell-by-cell into the area around its base, and each extension is monitored by communication between the tree's base and the Array Programmer. A cell in an embedded machine is always linked to the base cell by the shortest possible relcon path, and given a unique name. A working cell in an embedded machine ignores inputs from flawed cells and dangling array inputs.

Packaged memories are easily formed into larger memories by providing a few links between packages to allow growth of the tree through all the packages. The number of cells in the tree is only constrained by the required access time and the number p of page-address bits in each cell.

Overhead circuitry could be reduced by using triangular arrays instead of checkerboard arrays, if this was compatible with the production process.

It's obvious that this approach is applicable to any machine which may be realized as a tree machine. Inputs and outputs to such a machine could be

parallel rather than serial. One could implement a many-tracked sequential-access memory, with one cell for each track. Associative memories and even some multi-processor systems (similar to the ETHER system) are compatible with this approach.

These tree machines further evidence the fact that relaxing the requirements on the communication paths between essential cells in an embedded machine facilitates repair efficiency.

CHAPTER 6: CONCLUSION

This thesis has presented an LSI-oriented systems approach to test, configuration, and repair of cellular arrays. We've specified standard modules that are built into the cells of a machine to facilitate testing, loading, and repair. Thus the mechanisms for testing and customizing a flawed array are built into a simple, iterated part. A computer may access these mechanisms via a few direct connections to an array. Programs allow the computer to maintain or re-customize the array. We've been careful to note our assumptions, and to discuss design approaches that help insure the validity of these assumptions in actual arrays.

Development of terminology and models for programmable logic machines has helped us analyze important machine classes; these are arm, high-relcon, grid, and tree machines. A particular class of machine is characterized by the requirements placed on the communication paths between essential cells of any embedded machine in the class. A particular embedded machine is associated with a set of equivalent embedded machines. The nature of this set affects the testability and repairability of an array. Properties of a cell, such as balance, affect an embedded machine's structure and associated equivalence class; and therefore affect the repairability of an array.

There are reasonable practical and theoretical extensions of this work. We believe that tying further theoretical inquiry to actual machine realization goals will be most productive.

Tree and arm machines seem particularly suited to immediate array realization. These machines are relatively simple to automatically test, repair, and re-customize. Furthermore, their relatively low cell-circuitry overhead and high repair-efficiency give them major integration-level advantages.

Arm or tree machines may first be constructed in a system containing many ICs. Such a system would enable further exploration and demonstration of the feasibility of our approach. The major advantages of a many-IC system, compared to a system integrated on one slice, are its low development cost and high component accessibility. Such a system should be able to function when some of its ICs are removed or destroyed, and some of its wires are cut. The many-IC system would enable us to refine our designs and our test and repair programs. The major limitation of a many-IC research vehicle is that it doesn't precisely model our ultimate goal, a cellular system integrated on one flawed slice.

Besides its obvious value as a system component, a single-slice tree or arm machine would help answer important questions relevant to other arrays. How accurately do our assumptions model actual conditions on a flawed slice? How significant is the branch cell problem? How do power supply, heat dissipation, array size, and other practical considerations affect array implementation? Answers to these questions will depend partly on the engineering skill and production care of array developers. Since arm and tree machines are simpler to implement than many other programmable logic machines, ability to implement these arrays is a sine qua non for practicality of many other proposed arrays.

Arrays like General are the most exciting, because of their use of simple cells acting in parallel to provide universal computation-construction-repair capabilities. These arrays offer speed, reliability, and flexibility advantages in a low-cost integrated circuit. Current IC densities and yields probably don't allow practical realization of these large arrays. However, densities and yields are improving so rapidly that these arrays should be feasible before 1980. By then, many questions pertinent to these arrays should have been solved for tree and arm arrays. Continued work on testing and repair, and development of computer-aided design facilities for these arrays, will be important to their commercial success. Consideration should be given to the machine organizations most suited to high-relcon machines.

The first use of our approach to high-relcon machines may be in many-IC arrays of fairly complex machines, such as microprocessors. This is true because these arrays are closer to conventional digital systems. Unfortunately, the fact that such arrays have relatively low basic operation speed compared to General means they don't use high-relcon arrays to full advantage. Nevertheless we've seen the advantages of building simple test and repair mechanisms into an iterated component.

Since our test, configuration, and repair techniques may be adapted to existing arrays, it would be useful to categorize these arrays according to their realizability as an arm, tree, high-relcon, or other class of embedded machine.

Other inquiries may take numerous directions. A more rigorous

treatment of our testing assumptions and approach would be useful. Many questions remain concerning the repair of checkerboard arrays that embed high-relcon machines. These questions concern the best way to repair these arrays, and the limits of this repair. A better understanding of repair will allow better estimates of the reliability and maintainability of high-relcon machines. The use of a plurality of high-relcon machines in a self-repairing system should be explored. The reliability and maintainability levels that can be achieved by our various machines should be compared to the levels achieved by other machines. The network models and terminology we've developed for embedded machines can be refined, and new machine classes can be identified and studied. Our treatment of testing and repair for checkerboard arrays can be extended to arrays with other interconnection networks.

BIBLIOGRAPHY

<Altman 74>, L. Altman, "A new day for logic design", ELECTRONICS, vol. 7, #4, 2/21/74

<Banks 71>, E.R. Banks, INFORMATION PROCESSING AND TRANSMISSION IN CELLULAR AUTOMATA, Technical Report 81, MIT Project MAC, Cambridge, Mass. Jan. 1971

<Bell 72>, C.G. Bell, "The effect of technology on near term computer structures", COMPUTER, vol. 5, #3, pp. 29-38, March 1972

<Camenzind 72>, H.R. Camenzind, ELECTRONIC INTEGRATED SYSTEMS DESIGN, Van Nostrand Reinhold Co., New York, 1972

<Carr 72>, W. Carr and J. Mize, "The economics of MOS/LSI", MOS/LSI DESIGN AND APPLICATIONS, Texas Instruments, pp. 305-323, 1972

<Carter 70>, W.C. Carter and W.G. Bouricius, "A survey of fault-tolerant architecture and its evaluation", Report #RC 3154, IBM Watson Research, Nov. 1970

<Carter 73>, W.C. Carter, "Fault-tolerant computing: an introduction and a viewpoint", IEEE TRANSACTIONS ON COMPUTERS, vol. C-22, #3, pp. 225-229, March 1973

<Chien 73>, R.T. Chien, "Memory error control: beyond parity", IEEE SPECTRUM, vol. 10, #7, pp. 18-23, July 1973

<Codd 68>, E.F. Codd, CELLULAR AUTOMATA, Academic Press, New York and London, 1968

<Colbourne 74>, E.D. Colbourne, G.P. Coverley, and S.K Behera, "Reliability of MOS LSI circuits", PROCEEDINGS OF THE IEEE, vol. 62, #2, pp. 244-259, Feb. 1974

<Feeney 72>, H.V. Feeney, "Micro computer applications of electrically alterable ROMs", WESCON TECHNICAL PAPERS, 4th session, p. 4/4.1, 1972

<Feth 73>, G.C. Feth, "Memories are bigger, faster - and cheaper", IEEE SPECTRUM, vol. 10, #12, pp. 28-35, Nov. 1973

<Foss 70>, R.C. Foss, "Economic Considerations in LSI design", LARGE SCALE INTEGRATION IN MICROELECTRONICS, AGARD Lecture Series 40, July 1970

<Franson 74>, P. Franson, "Need custom design? Do it yourself", ELECTRONICS, vol. 47, #2, pp. 67-68, 1/24/74

<Furlow 73>, W. Furlow, "Today's hardest design decision: an overview of custom CMOS/LSI", EDN, vol. 18, #11, pp. 42-47, 6/5/73

<Gardner 70>, M. Gardner, "The fantastic combinations of John Conay's new solitaire game 'life'", SCIENTIFIC AMERICAN, vol. 223, #4, pp. 120-123, Oct. 1970

<Hodges 72>, D.A. Hodges, "Chip yield and manufacturing costs", SEMICONDUCTOR MEMORIES, IEEE Press, page 175, 1972

<Hu 73>, S.C. Hu, "Cellular synthesis of a synchronous sequential machine", IEEE TRANSACTIONS ON COMPUTERS, vol. C-21, #12, pp. 1399-1405, Dec. 1972

<Kautz 67>, W.H.Kautz, "Testing for faults in combinational cellular logic arrays", PROCEEDING OF THE 8TH ANNUAL SYMPOSIUM ON SWITCHING AND AUTOMATA THEORY, pp. 161-174, Oct. 1967

<Kautz 68>, W.H. Kautz, "Fault testing and diagnosis in combinational digital circuits", IEEE TRANSACTIONS ON COMPUTERS, vol. C-17, #4, pp. 352-366, April 1968

<Kautz 69>, W.H. Kautz, "Cellular logic-in-memory arrays, IEEE TRANSACTIONS ON COMPUTERS, vol. C-18, #8, pp. 719-727, Aug. 1969

<Kautz 71>, W.H. Kautz, "Programmable cellular logic", RECENT DEVELOPMENTS IN SWITCHING THEORY, Academic Press, 1971

<Kosy 72>, D.W. Kosy and J.A. Farquhar, AIR FORCE COMMAND AND CONTROL INFORMATION PROCESSING IN THE 1980S: TRENDS IN SOFTWARE TECHNOLOGY, Report R-1012-PR, Rand Corp., Santa Monica, Ca., Oct. 1972

<Kukreja 73>, S.N. Kukreja and I. Chen, "Combinational and sequential cellular structures", IEEE TRANSACTIONS ON COMPUTERS, vol. C-22, #9, pp. 813-823, Sept. 1973

<Landgraff 71>, R.W. Landgraff and S.S. Yau, "Design of diagnosable iterative arrays", IEEE TRANSACTIONS ON COMPUTERS, vol. C-20, #8, pp. 867-877, Aug. 1971

<Lathrop 70>, J.W. Lathrop, "Evolution of LSI", LARGE SCALE INTEGRATION IN MICROELECTRONICS, AGARD lecture series 40, July 1970

<Luecke 73>, J. Luecke, J.P. Mize, and W.N. Carr, SEMICONDUCTOR MEMORY DESIGN AND APPLICATIONS, McGraw-Hill, New York, 1973

<Marinos 71>, P.N. Marinos, "Fault diagnosis in digital systems - an overview", 5TH ANNUAL IEEE CONFERENCE ON HARDWARE, SOFTWARE, FIRMWARE, AND TRADEOFFS, pp. 71-72, Sept. 1971

<Marvin 67>, C.E. Marvin and R. Walker, "Plan ahead for LSI", Fairchild Semiconductor, Mountain View, Ca., Jan. 1967

<McLuhan 64>, M. McLuhan, UNDERSTANDING MEDIA: THE EXTENSIONS OF MAN, McGraw-Hill, New York, 1964

<Menon 71>, P.R. Menon and A.D. Friedman, "Fault detection in iterative logic arrays", IEEE TRANSACTIONS ON COMPUTERS, vol. C-20, #5, pp. 524-535, May 1971

<Minnick 66>, Minnick et al, CELLULAR ARRAYS FOR LOGIC AND STORAGE, Report #AF-CRL-66-613, Stanford Research Institute, Menlo Park, Ca., April 1966

<Minnick 67>, R.C. Minnick, "A survey of microcellular research", JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY, vol. 14, #2, pp. 203-241, April 1967

<Minsky 67>, M.L. Minsky, COMPUTATION: FINITE AND INFINITE MACHINES, Prentice-Hall Inc., Englewood Cliffs, N.J., 1967

<Moore 73>, G.E. Moore, "How large-scale is large enough?", ELECTRONICS, vol. 46, #22, p. 105, 10/25/73

<Moore 74>, G.E. Moore, "1974 award for achievement - Gordon Moore", ELECTRONICS, vol. 47, #21, p. 64, 10/17/74

<Mostek 73>, Mostek Corporation, THE MOSTEK LINE, Oct. 1973

<Mukhopadhyay 71>, A. Mukhopadhyay and H.S. Stone, "Cellular logic", RECENT DEVELOPMENTS IN SWITCHING THEORY, Academic Press, 1971

<Murphy 64>, B.T. Murphy, "Cost-size optima of monolithic integrated circuits", PROCEEDINGS OF THE IEEE, vol. 62, #12, pp. 1537-1545, Dec. 1964

<Noyce 71>, R. Noyce, "The integrated circuits industry", EDN/EEE, Sept. 15, 1971, pp. 28-32

<Peattie 74>, C.G. Peattie et al, "Elements of semiconductor device reliability", PROCEEDINGS OF THE IEEE, Feb. 1974, pp. 149-168

<Rowan 73>, J.H. Rowan and R.S. Kashef, "A universal programmable cellular array", PROCEEDINGS OF THE 6TH ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCE, Jan. 1973, pp. 264-267

<Sander 72>, W.B. Sander, "Yield-enhancement techniques in semiconductor memories", IEEE JOURNAL OF SOLID-STATE CIRCUITS, vol. SC-7, #4, pp. 298-300, Aug. 1972

<Seeds 67>, R.B. Seeds, "Yield and cost analysis of bipolar LSI", 1967 INTERNATIONAL ELECTRON DEVICES MEETING TECHNICAL DIGEST, IEEE, New York, 1967

<Seth 69>, S.C. Seth, "Fault diagnosis of combinational cellular arrays", PROCEEDINGS OF THE 7TH ANNUAL ALBERTON CONFERENCE ON CIRCUIT AND SYSTEM THEORY, (Monticello, Ill.), pp. 272-283, Oct. 1969

<Seth 70>, S.C. Seth, FAULT TESTING IN COMBINATIONAL CELLULAR ARRAYS, Report R-470, U. of Ill., Urbana, Ill., May 1970

<Shoup 70>, R.G. Shoup, PROGRAMMABLE CELLULAR LOGIC ARRAYS, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pa., March 1970

<Shoup 73>, R.G. Shoup, "Programmable cellular logic", 5TH ANNUAL IEEE CONFERENCE ON HARDWARE, SOFTWARE, FIRMWARE, AND TRADEOFFS, pp. 27-28, Sept. 1971

<Spandorfer 65>, L.M. Spandorfer, SYNTHESIS OF LOGIC FUNCTIONS ON AN ARRAY OF INTEGRATED CIRCUITS, contract #AF 19(628)2907, Nov. 1965

<Spandorfer 68>, L.M. Spandorfer, "Large-scale integration - an appraisal", ADVANCES IN COMPUTERS, pp. 179-237, 1967

<Tammaru 67>, E. Tammaru and J.B. Angell, "Redundancy for LSI yield enhancement", IEEE JOURNAL OF SOLID-STATE CIRCUITS, vol. SC-2, #4, pp. 172-182, Dec. 1967

<Tammaru 69>, E. Tammaru, "Testing of combinational cells in large-scale arrays", submitted to IEEE COMPUTER GROUP REPOSITORY, May 1969

<Thurber 69>, K.J. Thurber, "Fault location in cellular arrays", 1969 FALL JOINT COMPUTER CONFERENCE, pp. 81-88, 1969

<Toffler 70>, A. Toffler, FUTURE SHOCK, Bantam Books, New York, 1970

<Turn 72>, R. Turn, AIR FORCE COMMAND AND CONTROL INFORMATION PROCESSING IN THE 1980S: TRENDS IN HARDWARE TECHNOLOGY, Report R-1011-PR, Rand Corp., Santa Monica, Ca., Oct. 1972

<Vaccaro 74>, J. Vaccaro, "Semiconductor reliability within the Department of Defense", PROCEEDINGS OF THE IEEE, vol. 62, #2, pp. 169-184, Feb. 1974

<Vinson 74>, N. Vinson, "On today's real world of testing", EDN, vol. 19, #3, pp. 50-54, 2/5/74

<Vischi 72>, M. Vischi, "State of the art of reliability practice in the European computer market", PROCEEDINGS OF THE 1972 ANNUAL RELIABILITY AND MAINTENANCE SYMPOSIUM, pp. 329-335, Jan. 1972

<von Neumann 66>, J. von Neumann, THEORY OF SELF-REPRODUCING AUTOMATA, edited and completed by A.W. Burks, U. of Ill. Press, Urbana and London, 1966

<Wahlstrom 67>, S.E. Wahlstrom, "Programmable logic arrays - cheaper by the millions", ELECTRONICS, vol. 40, #25, pp. 90-95, 12/11/67

<Wahlstrom 69>, S.E. Wahlstrom, "Electronically controlled microelectronic cellular logic array", US PATENT 3,473,160, Patented OCT. 1969

<Yau 70>, S. S. Yau and M. Orsic, "Fault diagnosis and repair of cutpoint cellular arrays", IEEE TRANSACTIONS ON COMPUTERS, vol. C-19, #3, pp. 259-262, March 1970

BIOGRAPHY

Frank Blase Manning was born in Saint Louis, Missouri, on September 13, 1948. He is the eldest of five brothers and a sister. He graduated from Saint Louis University High School in 1966, and entered MIT the same year.

As an undergraduate he held various positions in Sigma Phi Epsilon social fraternity, and was elected a house officer. He lettered in soccer in his first three undergraduate years. He also ran several college mixers sponsored by his fraternity.

He received bachelor's and master's degrees jointly from the MIT Electrical Engineering Department in 1972. His thesis was published as a technical report titled *Autonomous, Synchronous Counters Constructed Only Of J-K Flip-flops.*

The author's graduate research was at MIT Project MAC. He has filed a patent relating to designs presented in his doctoral thesis, and plans to continue working toward the goal of mass-production of some form of the cellular arrays described in that thesis.

He is a member of Eta Kappa Nu and Sigma Xi honoraries, and has been nominated to Tau Beta Pi.

Frank has held numerous jobs in the short intervals between school attendance. He has worked as a paperboy, filling-station attendant, caddie, fireworks-stand proprietor, shipping clerk, technician, and digital music machine designer. He ran a McGovern storefront in the 1972 Presidential campaign, and worked in the Rhode Island primary that year.

Frank and his wife, Lynn Nina, were married on May 5, 1974.

His chief delights are his family, friends, and racquet sports. Other interests include innovation, politics, and other games. He likes designing hardware and software systems, and is interested in communication systems.