ON-LINE CHECKING
OF
PROGRAMMABLE LOGIC ARRAYS

by

DONALD MARCYNUK

A thesis
presented to the University of Manitoba
in fulfillment of the
thesis requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba

ISBN 0-315-63228-3

ON-LINE CHECKING OF PROGRAMMABLE LOGIC ARRAYS

BY

DONALD MARCYNUK

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Don M. Marcynuk

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Don M. Marcynuk

ABSTRACT

As the density of integrated circuit technology increases, so does the practicality of employing on-line checking schemes, and the necessity for doing so due to the expectation of soft failures which elude off-line testing. This thesis investigates on-line checking strategies for a particular class of circuit, namely the programmable logic array (PLA).

A succinct error model for nonconcurrent PLA's is formulated from a detailed analysis of errors due to a comprehensive fault model. This error model consists of: single-bit errors, one or more rightmost outputs held at zero, and the bitwise OR of two codewords (*i.e.* an error of type ξ).

Two new on-line checking schemes for nonconcurrent PLA's are presented. Both share a common base consisting of parity checked outputs and specifying the rightmost two outputs in a 1-out-of-2 code. One scheme defines extra PLA outputs so that the parity check also detects errors of type ξ. The other scheme, called OR-k, uses a function dependent checker to recognize errors of type ξ. Output codewords must be unordered. The checker realizes a positive (unate) function. Both schemes have the advantage that the design of testable PLA's is based on just the set of output patterns normally produced, rather than the entire PLA specification.

These schemes are compared to existing on-line checking strategies, including those designed to detect all modelled errors (*i.e.* schemes based on unidirectional error detecting codes, and those based on the PLA's physical structure), and others with only partial error coverage (*eg.* low cost residue codes). Quantitative figures of merit are determined for schemes with partial coverage.

The proposed schemes are found superior to schemes which use a checker on all product lines of a nonconcurrent PLA. An OR-k checker is faster, and usually smaller, than Berger and mod 3 code checkers.

The increase in number of product terms to achieve nonconcurrency is found in practice to be nearly the same as that required to encode the PLA with unordered outputs. Algorithms for transforming a PLA into a nonconcurrent form are presented, but "limited concurrency" is shown to be a more useful as well as more easily obtained form. An OR-k testable PLA with limited concurrency is shown to be strongly fault secure.

to Debbie

"I know what you're thinking about," said Tweedle-dum; "but it isn't so, nohow."

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, *Through the Looking Glass*

ACKNOWLEDGMENTS

I am indebted to Dr. D. Michael Miller, my supervisor, for initially suggesting this topic area, and especially for uplifting encouragement at critical junctures. The quality of the final manuscript is a tribute to the extensive advice of Drs. Miller and G.H. John van Rees. I am genuinely appreciative of the arduous task undertaken by the examining committee, and of the kind comments and suggestions by Drs. Gordon Russell and Robert D. McLeod.

The process of writing this thesis posed a severe test of the love and patience of my wife Debbie, happily we passed that test with flying colours. Debbie's unwavering confidence and continual efforts in support of my work were a significant catalyst for its successful completion. The first year with our daughter, Kathryn, coincided with the last year of my programme. Although Kati made it a real challenge to get any work done, the joy she brings to our lives more than compensates.

I wish to gratefully acknowledge the support provided by the Department of Computer Science, and that of an NSERC Postgraduate Scholarship. Thanks to my parents for giving me opportunities which they never had. Moral support from dnk@ibm.com was invaluable.

CONTENTS

List of Algorithms

List of Figures

List of Tables

SYMBOL GLOSSARY

Chapter 1

INTRODUCTION

1.1 THE PROBLEM

The world is increasingly dependent upon things controlled by micro-electronic integrated circuits. Along with the continual push for products with increased functionality, there exists a growing concern for reliability. This concern may be addressed in two ways with respect to digital electronic circuits. One is to periodically subject a circuit to a series of tests to determine if it behaves as originally intended. The other is to design circuits which have the capability to continually check that their own operation is correct. The subject of this dissertation is the analysis and self-checking design of a particular class of digital electronic circuit, the programmable logic array.

The structure of a programmable logic array, or PLA, will be described in detail later. For the moment, consider a PLA as a number of signal lines arranged in rows and columns in a regular fashion. The regular row/column organization of a PLA resembles the structure of memory circuits which have a storage cell situated at each intersection point. Some of a PLA's columns represent inputs to the circuit, while the remaining columns serve as its output lines. There is no electrical connection between a row and column line at the point where they cross, unless a transistor is specifically placed at that crosspoint. A PLA may be configured to realize a set of arbitrary Boolean functions by appropriate placement of such transistors. The minimum number of rows (which are called product terms) required to realize a function depends on the function. Minimization algorithms exist which attempt to obtain a realization requiring minimal number of rows, and possibly using a minimal number of crosspoint transistors. It is relatively straightforward to produce a chip layout for a PLA once one has a minimized

specification for the function, and the layout process is easily automated. The PLA layout is very dense, *i.e.* there is relatively little unused chip area internal to the PLA boundary. However, a PLA realization is often larger in area than a random logic design for the same function.

A PLA is essentially a two-level AND/OR array which directly realizes a Boolean function expressed in sum-of-products normal form. A random logic synthesis of the same function is not restricted to two levels of logic gates, and a variety of logic gate types may be used. The random logic realization of the function typically requires fewer logic gates than does a PLA, and these gates are often hand crafted into a physical implementation where the dimensions, placement and interconnection of all semiconductor devices are chosen to best utilize available chip area, and to obtain acceptable operating performance (*i.e.* speed, power dissipation, acceptable noise margin, *etc.*). The trade-off between PLA and random logic implementations is mainly one of design effort versus chip area.

Although the idea of using regular array structures to realize logic functions had been known for some time, it was not until 1975 that the use of PLA's was considered feasible. Fleisher and Maissel [Flei75] argued that the advent of computerized minimization algorithms made the design process easier, and that improved technology, which allowed increased circuitry per chip, made the area penalty less significant. They also suggested that as the systems being designed were becoming increasingly complex, a systematic design methodology, such as the one for PLA's, was required. Fleisher and Maissel are widely cited for describing various technology implementations and structures for PLA's.

Today, technology and design complexity continue to advance. Random logic circuit design continues to be popular, due in part to sophisticated computer-aided design (CAD) software tools running on powerful workstations. Such tools include programs which fit logical circuits to gate arrays or standard cells, silicon compilers, and automated placement and routing. Despite these, the PLA is still a viable option on a designer's

palette. High profile commercial products such as microprocessors often use PLA's in their implementations. Kuban and Salick [Kuba84] describe the Motorola MC68020 as having several PLA's driving, in total, over 200 outputs. Intel's 80386 also incorporates PLA's, three of which have input/product/output lines numbering 13/350/16, 16/160/18, and 19/175/12 [Gels86]. The BELLMAC-32A microprocessor incorporates eight PLA's ranging in size from 25/42/12 to 50/190/67 [Law82].

PLA's have enjoyed considerable academic attention in the hardware testing community, primarily because practitioners can take advantage of their regular structure in the analysis and design of testing strategies.

Most of the testing techniques developed thus far have largely been ignored for practical commercial chips. Even though current technology permits high density circuitry, most designers would rather not sacrifice increased functionality for enhanced testability. For example, major criticism of the proposed IEEE standard P1149.1, for a boundary scan design for testability scheme, was directed against the area overhead of up to 30 percent [Fitz89]. Low area overhead off-line testing strategies are used for the PLA's of both the MC68020 and the 80386. For the MC68020, microcode firmware cooperates which an external tester to apply deterministic test vectors to most of its PLA's. For the rest, microcode generates an exhaustive test sequence which is compressed by an on-chip signature register. Results of all tests are routed off chip for external verification. For the 80386, exhaustive pseudorandom test sequences (512K cycles) are generated on-chip using linear feedback shift registers (LFSR), and PLA responses are compressed on-chip using multiple input signature registers (MISR). Microcode compares the final signatures to expected values which are stored in ROM.

Lala [Lala86] presents the argument that "conventional (deterministic) test generation techniques cannot cope with circuits [in general] of VLSI complexity," and, for various reasons, built-in self-testing based on pseudorandom test sequences and signature

compression may not be suitable either. On-line checking techniques introduce extra circuitry to continuously monitor the behaviour of functional blocks. Lala invites researchers to pick up the torch for on-line checking. Only on-line checking techniques will be able to cope with transient and intermittent failures which are expected to predominate in high density VLSI. The challenge is to develop on-line testing methods which are appropriate for particular circuit structures while keeping additional chip area low. This dissertation strives to do this within the context of programmable logic arrays.

1.1.1 Dissertation Outline

The remainder of Chapter 1 defines general terms and notation used in this dissertation. Additional technical terms will be introduced when needed in later chapters. Several different nomenclatures appear in the literature. Where possible, this work uses a single term or notation consistently for a given concept, even when referring to details of papers which use different terminology.

Chapter 2 reviews the literature pertaining to all aspects of PLA testing. Chapter 3 presents an analysis of the expected behaviour of normal and nonconcurrent PLA's under a specific fault model. Quantitative measures used to evaluate the effectiveness of partially self-checking schemes are also discussed. Experimental results based on the error analysis and measures are then shown for mod 3 and parity check schemes.

Chapters 4 and 5 introduce two new strategies for the on-line checking of nonconcurrent PLA's. In Chapter 4, it is shown how simple parity can be used to detect errors due to all modelled faults. An algorithm for making a PLA specification nonconcurrent is also proposed. Several variations of a strategy called OR-k testing are developed in Chapter 5.

Chapter 6 contains a comparative evaluation of the new strategies against several existing on-line checking schemes. Finally, some concluding remarks and topics for further research are presented in Chapter 7.

1.2 DEFINITIONS AND NOTATION

A *binary vector* is a tuple of elements from the set {0, 1}. The *Hamming distance* (or simply distance) between two binary vectors, denoted $d(x,y)$, is the number of bit positions in which they differ. The *Hamming weight* (or simply weight) of a binary vector, denoted $wt(x)$, is the number of 1-bits in the vector. Vector a *covers* vector b, denoted $a \supseteq b$, if all positions which are a 1 in b are also a 1 in a. A set of binary vectors is *unordered* if no vector covers another. The logical operations AND, OR, exclusive-OR (EXOR) and complementation, symbolized as $a \wedge b$, $a \vee b$, $a \oplus b$, and $\overline{a}$ respectively, are performed bitwise on binary vectors. The difference of two binary vectors is $a-b=a \wedge \overline{b}$. Notation of the form a' does not represent the complementation operator, as so defined by some authors. Rather a' represents a variable which is distinct from but closely related to the variable a. This notion is likewise intended for other adornments such as $\hat{a}$ and $\tilde{a}$.

An n-bit binary *code* is a subset of the set of all 2^n n-tuples. An n-tuple belonging to a code is called a *codeword*. Codewords often have some distinctive property which is absent in all noncodewords.

When binary n-tuples correspond to signal lines of electronic circuits, there is a possibility that some of the bits are incorrect due to circuit malfunction. Use of a code provides protection against those malfunctions which transform a codeword into a noncodeword; a check is required to determine whether the n-tuple is a codeword or not. For example, a simple odd parity code contains all n-tuples with odd weight; it detects any change affecting an odd number of bits.

If one wants to treat a particular set of r-tuples as a code, and if those tuples have no inherent distinctive property, then one can map the r-tuples onto n-tuples in such a way that there exists some property making it easier to distinguish codewords from noncodewords. For example, a simple odd parity code could be constructed from a set of

arbitrary (n–1)-tuples by adding a check bit so that each n-tuple has odd weight. A code where the original r-tuple appears embedded in the codeword is said to be *systematic.* For a *non-systematic* code, the r-tuples are mapped onto n-tuples in such a way that the original values are not directly accessible. An inverse mapping must be performed to recover them. The term systematic is further intended for the case where all 2^r r-tuples need to be encoded. If only a subset of these values is required, then the code is said to be *separable* [Jha89a]. A code is *unordered* if the set of all codewords comprising the code is unordered.

A variable, x_i, and its complement, $\overline{x}_i$, are called *literals.* A Boolean function $f(x_1, x_2, \ldots, x_m)$ for which there exists a disjunctive normal form expression (sum-of-products) where the variable x_i only appears as an uncomplemented (complemented) literal is said to be *positive (negative) in* x_i. If there exists a normal form expression of f which is either positive in x_i or negative in x_i for each variable, then the function is called *unate.* If a unate function is positive (negative) in x_i for all variables x_i of f, then the function is said to be *positive (negative).*

Let S be a set of objects, and $|S|$ denote the number of elements in the set. A *partition* of set S is a collection of subsets of S such that the union of all subsets equals S, and all pairs of subsets have null intersections. A *grouping* of S is a collection of subsets whose union also covers S, but subsets are allowed to intersect. A relation ~ on a set S ($a,b,c \in S$) is called a *compatibility relation* if it is reflexive ($a \sim a$) and symmetric ($a \sim b \Rightarrow b \sim a$), but not necessarily transitive ($a \sim b$, $b \sim c \not\Rightarrow a \sim c$). A subset T of S is said to be a *maximal compatible* of the relation ~ if all elements of T are mutually compatible ($a \sim b$ for all $a,b \in T$), and it is not possible to enlarge T while maintaining mutual compatibility (for all $c \in S–T$ there exists some $a \in T$ such that $a \not\sim c$). The collection of all maximal compatibles is a grouping of S. Note that this differs from an equivalence relation which is transitive and thus induces a partition of S.

1.2.1 Programmable Logic Arrays

A *programmable logic array* or *PLA* is a regular structure which implements a multi-output combinational logic function†. Conceptually it is a two level AND/OR circuit directly realizing a set of sum-of-products Boolean expressions. Selected literals are logically AND'ed together to form *product terms*, p_j, and selected product terms are OR'ed together to form outputs, z_k.

The sum-of-products expressions implemented by a PLA are usually determined by a computer program which takes a set of function specifications and attempts to find a sum-of-products representation with minimal product terms and literals. Several minimizers have been developed, for example: MINI [Hong74], ESPRESSO-II [Bray84], and McBOOLE [Dage86]. The size of a PLA may be reduced through *folding* which attempts to "determine permutations of the rows (and columns) which permit a maximal set of column pairs (row pairs) to be implemented in the same column (row) of the physical logic array" [Hach82].

In this dissertation, a general PLA realizes a multi-output function F, and has m input variables, p product terms and n output lines. The input variables and function outputs will be denoted: $X=(x_1, x_2, x_3,\ldots, x_m)$ and $Z=(z_1, z_2, z_3,\ldots, z_n)$, respectively, with the behaviour of the PLA specified by $Z=F(X)$. Various technological implementations of a PLA exist, but a one-bit decoder NOR/NOR structure is adopted in this dissertation, because of its common use in recent literature. Each input line, x_i, is decoded into two *bit lines*, $x_{i,0}$ and $x_{i,1}$, representing the true and complemented values of x_i, respectively. Running perpendicular to bit lines are *product term lines* (or simply product lines). Wherever two lines intersect is a potential site for a *crosspoint* or contact, denoted graphically as a dot on the lines. The crosspoint

† A PLA actually realizes a set of Boolean functions, but for the sake of simplicity, we will refer to this set of functions as a multi-output function. Similarly, a PLA with one output will be said to implement a single-output function.

represents a transistor connection between the bit and product term lines as shown on the left hand side of Figure 1.1.

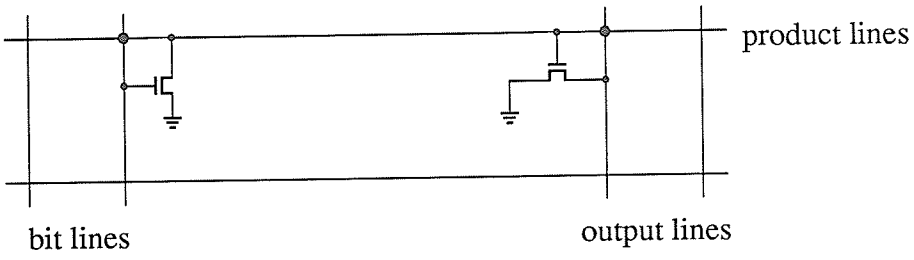
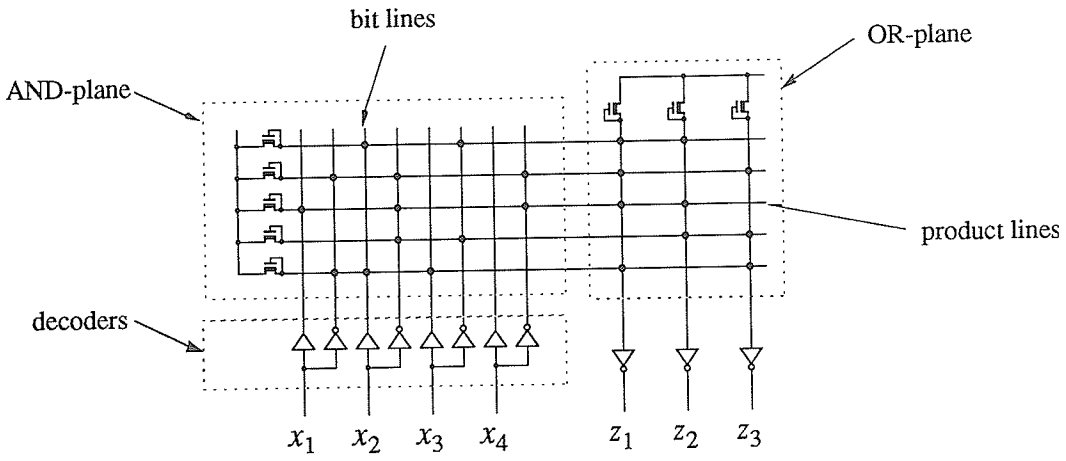


Figure 1.1 Transistor connection detail for PLA crosspoints.

A product term line computes the logical NOR of the connected bit lines; if any crosspoint transistor is turned on, then the product line is pulled to ground through that transistor. Since the dual form of an AND gate is an OR with inverted inputs and output, any product of literals expression can be realized by connecting the complement of each desired literal. A product term line and the product term it realizes will be used synonymously.

Product lines and output lines are similarly configured so that an output line computes the NOR of the connected product lines (as illustrated on the right hand side of Figure 1.1) . Final inverters on each output line produce the sum-of-products desired. The area where bit and product lines cross is called the *AND-plane*, and where product and output lines cross is the *OR-plane*.

An example PLA is shown in Figure 1.2a which implements the set of functions given in Figure 1.2b (this example corresponds to the sample PLA called "bis2" as described in Appendix A). Depletion-mode transistors, which act as pull-up resistors for product and output lines, are shown graphically as  and  . Non-inverting buffers, which are used to help construct the decoders, are depicted as .



a) PLA diagram.

$$z_1 = \bar{x}_2 x_3 + x_1 x_2 x_4 + \bar{x}_1 x_2 x_4 + x_1 \bar{x}_2 \bar{x}_3$$
$$z_2 = \bar{x}_2 x_3 + \bar{x}_1 x_2 x_4 + x_2 x_3$$
$$z_3 = x_1 x_2 x_4 + x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3$$

b) Sum-of-products equations

```
X01X  110
11X1  101
01X1  110
X11X  011
100X  101
```

c) Cubical specification

Figure 1.2 Example PLA.

The 2^m m-bit binary vectors corresponding to all possible distinct assignments to a circuit's m inputs are called *minterms*. Two minterms are *adjacent* if they differ in exactly one bit position. A product term can be represented in cubical notation [Diet78] where a *cube* is a tuple of elements from {0, 1, X} and a 0(1) indicates that the corresponding input variable in its complemented(true) form is part of the product term. An X means that the product term is independent of the corresponding input variable, *i.e.* the variable does not appear in the product term.

An m-coordinate cube, c, will be denoted: $c=(c_1, c_2, c_3,\ldots, c_m)$; and a list of p cubes, $\mathcal{C}$, as: $\mathcal{C}=(c^1, c^2, c^3,\ldots, c^p)$. For index set $I=\{i_1, i_2,\ldots, i_k\}$, c^I denotes the subset of cubes from $\mathcal{C}$ selected according to indices in I. The notation c_j^i represents the j^{th} component of the i^{th} cube; for a list of coordinate indices, $\Omega=(\omega_1, \omega_2,\ldots, \omega_k)$,

the notation c^i_Ω represents the subcube made up of these coordinates, $c^i_\Omega=(c^i_{\omega_1}, c^i_{\omega_2},\ldots, c^i_{\omega_k})$. The notation $c\circ c_i\rightarrow\alpha$ represents the cube which is identical to c except the i^{th} component is changed to α [Smit79].

The cubes corresponding to the product terms implemented in the example PLA in Figure 1.2a are shown in Figure 1.2c. Appended to the cube coordinates representing a product term is an n-bit binary vector which has 1's in only those positions corresponding to the subset of the n outputs which use the product term. The term "cube" will be used to refer to a product term, both alone, and including the appended binary vector. When needed, the two parts of a product term line will be referred to as the *input part*, cube C, and the *output part*, binary vector D. Product term line L is the concatenation of C and D, $L=(C, D)$, where $L_{1..m}=C$ and $L_{m+1..m+n}=D$. A product line and the cube it realizes will also be used synonymously. A PLA specification is represented as the set of cubes, $\boldsymbol{L=(C, D)}$.

The set of minterms belonging to a cube are those found by replacing the X's with 0's and 1's in all possible ways. If a minterm belongs to a cube then it is said that the cube *covers* that minterm. A cube which contains k X's is said to be a *k-cube*. A 0-*cube* which contains no X's represents a single minterm. (Cube c^1 of Figure 1.4a is a 1-cube, and c^2 is a 3-cube.) The *intersection* of cubes c^i and c^j, denoted $c^i\cap c^j$, is the cube representing the set of minterms which are contained in both c^i and c^j. If two cubes have no minterms in common, then their intersection is said to be empty or *null*, and the cubes are said to be *disjoint*. Two disjoint cubes are *adjacent* if some minterm from one is adjacent to some minterm from the other. (All three cubes shown in the example of Figure 1.4c are pairwise adjacent to each other.) Cube c^i *contains* or *covers* cube c^j, denoted $c^i\supseteq c^j$, if all minterms in c^j are also in c^i.

Cube intersection and covering can also be defined coordinate-wise using the tables shown in Figure 1.3. The intersection of cubes c and s, $t=c\cap s$, is obtained coordinate

by coordinate as $t_i = c_i \cap s_i$, and the intersection is null if any $t_i = \phi$. Cube c covers cube s, $c \supseteq s$, only if $c_i \supseteq s_i$ is true (1) for all coordinates.

c_i \ s_i	0	1	X
0	0	ϕ	0
1	ϕ	1	1
X	0	1	X

(intersection, $\cap$)

c_i \ s_i	0	1	X
0	1	0	0
1	0	1	0
X	1	1	1

($\supseteq$)

Figure 1.3 Tables for coordinate-wise $\cap$ operator and $\supseteq$ relation for cubes.

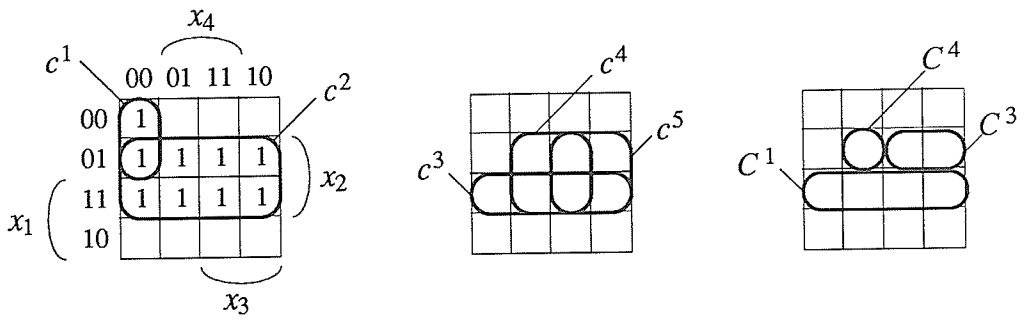
The *sharp* operation on cubes c^i and c^j, denoted $c^i \# c^j$, represents all minterms which are contained in c^i but not in c^j. It may be impossible to represent the result using just a single cube. The *disjoint sharp* operation, $c^i \circledast c^j$, is similar except the result must be expressed as disjoint cubes. A procedural definition for $c^i \circledast c^j$ ([Hong74]) specifies the result as m cubes, C^k for k=1..m, and each C^k defined according to equation 1.1.

$$C^k = (c_1^i \cap c_1^j,\ c_2^i \cap c_2^j,\ \cdots,\ c_{k-1}^i \cap c_{k-1}^j,\ c_k^i \cap \bar{c}_k^j,\ c_{k+1}^i,\ c_{k+2}^i,\ \cdots,\ c_m^i) \tag{1.1}$$

Some of the resultant cubes, C^k, may be null. Also, $c_k^i \cap \bar{c}_k^j$ is considered to be ϕ, in the sense of Figure 1.3, whenever c_k^j = X. The example in Figure 1.4 shows the effect of the two sharp operators. The definition of these operators, as well as the other cubical operations, extends to arguments which are sets of cubes in a straighforward manner. For example, see equations 1.2 and 1.3. Let $\boldsymbol{C}$ and $\boldsymbol{B}$ be arrays of p and q cubes, respectively. For disjoint sharp, equation 1.2 requires that all cubes in $\boldsymbol{B}$ be initially pairwise disjoint.

$$\boldsymbol{B} \# c^j = \bigcup_{i=1}^{q} b^i \# c^j \tag{1.2}$$

$$\boldsymbol{B} \# \boldsymbol{C} = (\ \ldots((\boldsymbol{B} \# c^1) \# c^2) \# \ldots\ ) \# c^p \tag{1.3}$$



a) $c^1 = 0X00$ and $c^2 = X1XX$ b) $c^2 \# c^1$ c) $c^2 \circledcirc c^1$

Figure 1.4 Example of sharp and disjoint sharp operators.

A set of cubes is called a *cubical array* (or simply array). An array is a specification of a PLA, and therefore represents a set of Boolean functions. The output of a PLA, for any given input minterm, can be determined from its cubical array. All product term lines whose cubes cover the minterm are *selected.* The output is the bitwise OR of the output parts of all selected product lines. A *functional array* ([Diet78]) is a cubical array such that if the input parts of two cubes intersect, then their output parts must be identical. The output of a PLA specified by a functional array is simply the output part of any cube which covers the input minterm; all cubes which cover the minterm have identical output parts.

A cubical array is *k-concurrent* if k is the size of a maximal subset of cubes which have a non-null intersection; k is the *concurrency* of the array. If all pairs of cubes from an array are disjoint, then the array is *1-concurrent.* The array of three cubes shown in Figure 1.4b is 3-concurrent because the intersection of all three cubes is non-null; it is the 0-cube 1111. The concurrency of Figure 1.4a is 2, and Figure 1.4c is 1-concurrent. A cubical array is a *complete cover* if the union of all its cubes contains all possible minterms. An array is *nonconcurrent* if it is both 1-concurrent and a complete cover.

1.2.2 On-line Checking

The *design* of a circuit is essentially a mathematical model which defines the intended output for any input applied. A physical implementation of a design as an integrated circuit is subject to *failures* and *defects* so that the intended output may not always be produced. When this happens an *error* may be observed on the circuit's outputs. For the purpose of analysis, a defect is approximated by a *fault* which describes a design transformation. For example, a fault may transform a three input AND gate into a two input AND gate. Let $\mathcal{F}=\{f_1, f_2,..., f_q\}$ represent the set of faults that are modeled. A circuit designed to implement the function $F(X)$ will be said to implement the function $\tilde{F}(X,f)$, if the fault f is present, or simply $\tilde{F}(X)$, if it is not known which, if any, faults are present. With $\varnothing$ representing a fault-free circuit, $F(X)=\tilde{F}(X, \varnothing)$. Define $\mathcal{Z}$ to be the set of output patterns produced by an error-free circuit: $\mathcal{Z}=\{ Z=F(X)\}$. Let $N=|\mathcal{Z}|$.

On-line checking is the process of attempting to detect errors during normal operation of the circuit. The terms "built-in" and "concurrent" are also used in the literature but to avoid confusion with "built-in self-test" (defined later) and concurrency as defined previously, these terms will not be used†. The general arrangement of an on-line checking scheme is depicted in Figure 1.5.

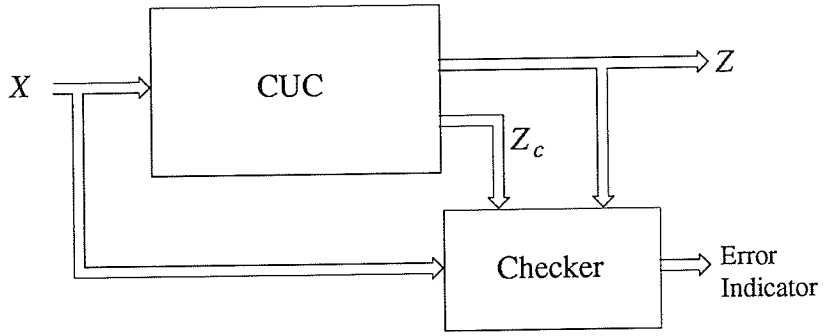


Figure 1.5 On-line checking scheme.

† The terms "self checking" and "implicit testing" also appear in the literature.

The *circuit under check* (CUC, alternatively circuit under test, CUT) performs some function in a digital system, taking in input X and producing output Z. For the purpose of on-line checking, the CUC is extended to also produce extra check bit outputs, Z_c. The checker circuit is designed to detect when a fault causes an error to appear on the outputs. In general, the checker is a function of X, Z and Z_c, but often X is not used.

The combined output of Z and Z_c usually is designed so that all output patterns produced are codewords belonging to some error detecting code which is capable of detecting errors resulting from a predefined set of faults. A checker expects its inputs to be codewords. If a noncodeword is observed by the checker, then an error has been detected, and the checker produces an error indication output. The CUC's inputs may also be encoded in an error detecting code.

Carter and Schneider introduced the concept of *totally self-checking* (TSC) circuits [Cart68], and Anderson and Metze formalized it as follows [Ande73]. Both the circuit's inputs and outputs are assumed to be encoded. The totally self-checking goal is to have a circuit always produce a noncode output as the first erroneous output due to a fault. An alternative statement of the goal is to never produce an incorrect codeword output. A circuit is *self-testing,* if for every fault considered, it produces a noncode output for some code input. A circuit is *fault secure,* if for every fault considered, it never produces an incorrect code output for a code input. A circuit is *TSC* if it is both self-testing and fault secure. A circuit is *code-disjoint* if it produces a noncode output for every noncode input. A circuit is a *TSC checker* if it is self-testing and code disjoint†. A TSC checker outputs an error indicator if it is given a noncodeword input. It is generally assumed that a TSC circuit will detect a fault before a second fault, which might mask the first, occurs. This implies that all input combinations are applied between the occurrence of successive faults.

---

† Some authors require that the circuit be fault secure also.

Smith and Metze enlarge the class of circuits which meet the TSC goal to include *strongly fault secure* (SFS) circuits [Smit78]. Generally, if a circuit is fault secure but not self-testing for some fault, then if that fault is present it is not detectable but the circuit produces no erroneous outputs. However, the circuit transformed by this fault may no longer be fault secure either, and future faults which occur may cause undetectable errors. A circuit is SFS if it remains fault secure for any sequence of fault transformations until a final fault transforms it into a TSC circuit.

In a similar manner, Nicolaidis and Courtois extend the definition of code disjoint to *strongly code disjoint* (SCD) [Nico84, Nico88]. Generally if a circuit is code disjoint, but not self-testing or fault secure, it is possible that a fault will cause it to produce an incorrect codeword output for a code input. If the circuit is a checker, then the value of its output is immaterial, only whether it is a codeword or not is significant because any noncodeword output stands as an error indication. If the checker's input is a codeword and the checker outputs a wrong valued output codeword, it still indicates no error in the CUC. Likewise if the checker's input is a noncodeword and the checker produces a wrong valued noncode output, the error is still reported. As long as no fault transforms the checker into a non-code disjoint circuit, the checker continues to correctly do its job of monitoring the CUC for noncodewords. A circuit is therefore SCD if it remains code disjoint for any sequence of fault transformations until a final fault transforms it into a self-testing code disjoint circuit.

With respect to programmable logic arrays, one ideally strives to develop techniques for designing TSC (or SFS) PLA's and TSC (or SCD) checkers to check them.

1.3 SUMMARY

This chapter posed the problem of on-line checking of programmable logic arrays. The structure of a PLA, cubical operations and terminology, and on-line checking

concepts were defined. Justification and incentive for this line of research were presented.

A glossary of the symbolic notation introduced in this chapter, as well as symbols defined in later chapters, may be found on page xii. Technical term definitions are indexed on page 202. Computer program source code listings for the implementation of algorithms presented in later chapters may be found in [Marc90b].

Chapter 2

LITERATURE REVIEW

This chapter presents a comprehensive review of recent literature pertaining to all aspects of PLA testing. Although the subject of this dissertation is the on-line checking of PLA's, it is important to be aware of the related research results in the area of off-line testing. Analysis of expected faults and methods to detect erroneous circuit output form the basis of all testing strategies. It is also interesting to note the extent to which some off-line schemes go to make a PLA more easily testable, and compare that with the effort required to achieve on-line testing. One should also notice that compared to off-line testing, relatively little work has been done on on-line checking of PLA's.

The material on off-line testing strategies will be presented in an informal manner, so as to gain an intuitive understanding of the testing method and philosophy. Several of the original publications cited, as well as other survey papers [Agar86, Regh86, Some86, Zhu88] provide detailed information, proofs and comparative analysis of each strategy's multidimensional attributes. Such attributes may be used as the basis for evaluation of the various testing schemes, all which address the same basic testing problem but within varying contexts. These attributes include: fault coverage, testing time, test set storage size, function independence, area/pin/delay overhead, technology independence, faults modeled, and computational complexity of test generation or design automation algorithms. Breuer and Zhu [Breu85] have even developed an expert system to help designers select a PLA test methodology suited to their specific goals.

Figure 2.4 (at the end of the chapter) presents a taxonomy of the PLA testing strategies reviewed.

2.1 FAULT MODEL

The possibility of a physical failure of an integrated circuit which prevents if from performing its intended function and yielding correct outputs within specified voltage and timing constraints, is a reality that designers and users of that circuit must deal with. Common causes of failure include manufacturing defect, irreversible damage or wear-out after a period of use, and transient conditions such as supply voltage or temperature fluctuation and α-particle radiation [Dill88, p. 602] which leave no permanent effect in their wake. The actual physical failures that arise are dependent both on the circuit structure and technology, and have been studied extensively [Abra86, Bane82, Bane84, Gali80, Maly86, Maly87, Nick80, Redd84, Wads78]. Rather than working at the lowest level of detail necessary to analyze physical failures, designers of test strategies typically define a *fault model* which describes the failures of interest more abstractly.

The classic *stuck-at* fault model is formulated in terms of the logic gate representation of a circuit. In this model an input or output of a logic gate is permanently held at a logic 0 or logic 1 state regardless of the input pattern applied to the gate. We say that the input or output is stuck-at-α for $\alpha \in \{0,1\}$. Primary input faults, internal decoder faults, and faults for any other logic added to the PLA are usually modeled traditionally. When applied to the AND and OR-planes of a PLA, the classic gate level stuck-at model is usually interpreted as: a bit line, a product term line, or an output line stuck-at-α. Stuck-at faults on individual AND or OR "gate" inputs are equivalent either to a crosspoint fault (described later) or to the gate's output being stuck. Chen, Fuchs and Abraham [Chen85] argue that product line stuck-at-1 faults can be "avoided" by conservative chip layout.

Another standard gate level fault model is the *bridging* fault which refers to a short circuit between two or more normally unconnected circuit nodes. Complex behaviour arises from a physical failure of this sort, and is generally technology dependent. A bridging fault may transform a combinational circuit into a sequential circuit with feedback. Bridging fault models typically restrict potential bridges to two points within close proximity and may or may not admit feedback bridges. Generally the bridged nodes

are modeled to take on a value which is the logical AND or OR of their intended values, depending on the technology used. Some authors consider both types, but many assume only AND bridging. In terms of a PLA, most authors account for bridging faults between adjacent parallel lines in the two planes, and often between orthogonal lines. The latter may require special interpretation if a crosspoint transistor exists between the bridged lines, because the fault represents a short circuit across the gate and drain of that transistor (*cf.* Figure 3.6). Banerjee and Abraham [Bane82] show that, depending on the ohmic value of the bridge, the possible effects of such a fault include: no effect, NOR gate output effectively stuck-at-0, or both lines taking on the same intermediate voltage level which may not be considered a proper logical 0 or 1.

The regular structure of a PLA lends itself to a third type of fault model introduced by Cha [Cha78]. A PLA may be considered a generic device consisting of two perpendicular groups of parallel lines, one group crossing over the other. A specific function is realized by configuring the PLA's *personality*, *i.e.* connecting diodes or transistors between certain pairs of crossing lines. A *crosspoint* or *contact* fault is defined as the erroneous presence or absence of such a connection. While a missing crosspoint is intuitively plausible as a physical failure, some authors believe that the spontaneous creation of an unintended transistor at a vacant crosspoint site is extremely unlikely [Eich80, Kuba84, Some84]. Despite this, extra crosspoint type faults are still worthy of consideration because "they may cover nonmodeled faults and on the other hand, a complete crosspoint test set is useful for design verification" [Some84]. Also Maly [Maly86] presents probability distributions for various faults, including extra crosspoints (see later).

Smith [Smit79] further classified crosspoint faults as follows:

- *S* or *shrinkage* fault for extra crosspoints in the AND-plane,
- *G* or *growth* fault for missing crosspoints in the AND-plane,
- *A* or *appearance* fault for extra crosspoints in the OR-plane,
- *D* or *disappearance* fault for missing crosspoints in the OR-plane.

Ramanatha and Biswas [Rama83] identify the special case of an extra AND-plane crosspoint when a crosspoint already exists on the complementary bit line, and denote it a *V* or *vanishing* fault, since the product term can never be asserted. Biswas and Jacob [Bisw85] suggest alternate associations for *S* and *G*, specifically *surplus* and *gone*. Wei and Sangiovanni-Vincentelli [Wei86] denote an output stuck-at-1 fault as an *E* fault. An alternative crosspoint fault classification introduced by Agarwal [Agar80] denotes *S* and *A* faults as *0-contact* faults, and *G* and *D* faults as *1-contact* faults.

A fourth type of fault, the *line break* fault, which actually corresponds to a common physical defect, is advocated by Tamir and Séquin [Tami84]. For a PLA, lines disconnected from a driving source "float", and any crosspoints controlled by a floating line generally can not be activated. Thus those crosspoints appear missing. Tamir and Séquin also define *weak* 1(0) faults for the situation when a "line is supposed to be a logic 1(0) but is interpreted by at least one of the gates [it is connected to] as logic 0(1)." Weak 1(0) faults can also be thought of as transient stuck-at-0(1) faults on individual gate inputs. A short between orthogonal lines may also behave as a weak 1(0) fault on the shorted lines.

Fault models, especially the stuck-at model, although necessary to make the testing problem tractable, have long been considered poor approximations of observed physical failures in random logic circuits [Gali80, Nick80]. PLA fault models, however, correlate closely to observed defects and thus enjoy greater credibility. Based on empirical observation of actual process parameters, Maly [Maly86] performed extensive probabilistic analysis of spot defects which arise during the manufacture of integrated circuits. Defects are generally observed to be spots on a mask causing missing or extra material to be deposited on some layer of the integrated circuit. These in turn appear as breaks in one or more lines or shorts between two or more lines. This work lends evidential support for all PLA fault models described above: stuck-at, adjacent and orthogonal line bridging†, both missing and extra crosspoints, and line breaks.

† Maly defines a *merger* fault which appears indiscernible from a bridging fault.

Furthermore, multiple faults and feedback bridging are considered likely. Table 2.1 summarizes from [Maly86] a sample set of fault probabilities for a particular PLA, and indicates that some of the faults considered are much more likely than others.

Fault	Probability
2 product lines shorted	2.5×10^{-5}
product line stuck-at-0	1.5×10^{-5}
3 product lines shorted	1.5×10^{-6}
missing crosspoint	1.0×10^{-8}
feedback bridge	1.0×10^{-8}
extra crosspoint	$< 1.0 \times 10^{-8}$
orthogonal bridge	$< 1.0 \times 10^{-8}$
line break	$< 1.0 \times 10^{-8}$

Table 2.1 Probabilities of various PLA faults [Maly86].

Figure 2.1, reproduced from another paper by Maly [Maly87], represents an interesting physical defect. A scratch on a photolithographic mask resulted in several broken traces as well as a short which reconnects two broken pieces from different lines, drastically altering the circuit topology. It is difficult, if not impossible, to represent such a physical failure using any combination of the above fault models.

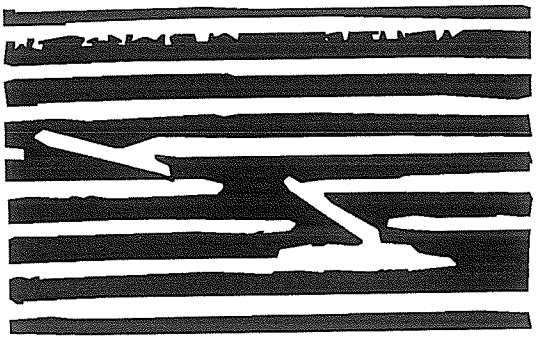


Figure 2.1 Example of multiple metal line shorts and breaks due to a scratch on a photolithographic plate [Maly87].

Often authors make the assumption that only a single fault exists. A testing strategy so designed may then be re-evaluated to determine coverage of multiple faults. Alternatively, various combinations of multiple faults may be considered from the outset.

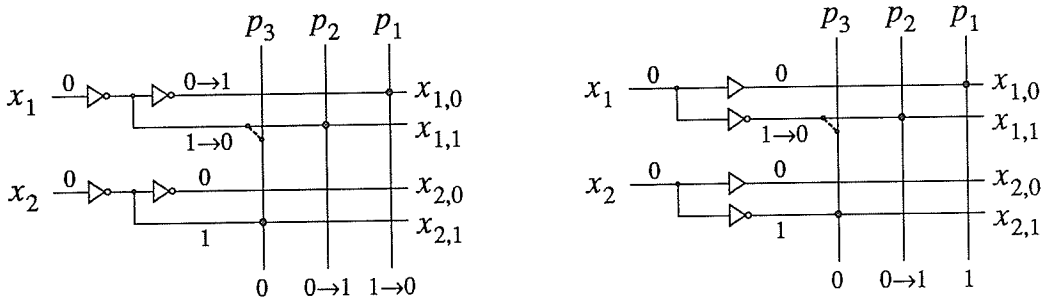
Special models may apply to certain PLA implementations. A CMOS PLA may require transistor stuck-open and stuck-on fault models. Folded PLA's may need to account for missing cuts. Dynamic realizations, and static PLA's which incorporate a driver on the product term lines between the two planes, may need to model product term stuck-at and bridging faults separately in each plane.

2.2 ERROR MODEL

Coding theorists deal with an even more abstract view of circuit testing. To them a circuit is a black box for which failures are characterized as having certain types of effects on the output lines. An *error model* thus describes the ways in which correct output patterns are expected to change. Fault detection schemes operate under the assumption that for any input pattern applied to a circuit, the output is either correct or has been modified in one of the ways specified by the error model. Generally error models specify the type, number and duration of errors considered [Bose86b]. A *binary symmetric* error model allows outputs to change 0→1 and 1→0 in any combination. An *asymmetric* error model permits either 0→1 or 1→0 but not both. In a *unidirectional* error model both 0→1 and 1→0 errors occur, but never together at the same time in an output pattern. *Single bit* error models assume at most one output bit is in error at a time. Double, triple, and t-bit *multiple bit* error models are similarly defined. An *independent* error model assumes each bit is equally likely to be in error. A *burst* error model assumes that the bits in error occur in positions close to each other in the output codeword. It is like a t-bit error, except the bits in error are contained within any field of t adjacent bit positions. The first and last bits of the codeword are often considered adjacent. Errors, as well as faults, can also be classified as *permanent*, *transient* (*i.e.* occurring once then disappearing), and *intermittent* (*i.e.* recurring from time to time). Abraham and Fuchs [Abra86] describe the relationships between various fault and error models.

2.2.1 Unidirectional Error Model

A unidirectional error model with no restriction on the number of errors is usually used for PLA's. This is justified through an analysis of errors arising from permanent single faults of a specific fault model [Mak82, Fuch84]. The regular structure of PLA's permits such an analysis to be done in a generalized manner and have it apply to all specific PLA's thereafter. Banerjee and Abraham [Bane82] point out that a particular design of one-bit decoders is required to prevent certain faults from causing a non-unidirectional error. This is illustrated by a bridging fault between bit line $x_{1,1}$ and product line p_3 in the AND-plane NOR-array shown in Figure 2.2. With the usual decoder design of Figure 2.2a, p_3 is 0 due to bit line $x_{2,1}$, the fault causes bit line $x_{1,1}$ to become 0, and this in turn passes through the inverter causing bit line $x_{1,0}$ to be 1.



a) non-unidirectional error b) modified decoder design

Figure 2.2 Effect of decoder design on error model — bridging fault example.

This situation can not occur in the modified decoder design shown in Figure 2.2b. Note that bit line $x_{1,1}$ in Figure 2.2a, if stuck-at-0, would also produce the same effect. Also note that the error in Figure 2.2a is equivalent to an external error which changes input x_1 from 0 to 1. Thus an internal PLA fault can be indistinguishable from an input error. This too is avoided in the modified design.

Nonconcurrent PLA's allow the use of a more specific error model ([Chen85, Marc88]) consisting of the union of single bit errors, an error resulting in an all zero

output, and errors where some other codeword is bitwise OR'ed with the intended codeword output. This is discussed in detail in section 3.1.2.4.

2.2.2 Unordered Codes

In the coding theory approach to error detection, information bits are encoded into a codeword belonging to an error detecting code capable of detecting errors from an appropriate error model. A code is systematic if the information and check bits exist in distinct bit fields of the codeword. Encoding thus is the process of determining check bits for given information bits, and concatenating them together to form a codeword. It is well known that any code which detects unidirectional errors must be unordered [Bose82c].

The *m-out-of-n* or constant weight codes are well known non-systematic unordered codes where each n-bit codeword contains exactly m 1's. *Two-rail* codes, where the check bits are the complement of the information bits, form a popular subset of the n-out-of-$2n$ code. Freiman [Frei62] showed that (n/2)-out-of-n codes are optimal non-systematic asymmetric error detecting codes; no other n-bit asymmetric code has more codewords. An m-out-of-n code may be used in a systematic manner, which is non-optimal, by appending check bits, equal in number to the difference of maximum and minimum number of 1's in the set of information words, so that each codeword has the same weight.

Berger [Berg61] first described what he called "sum" codes, but which now bear his name, as a separable (systematic) asymmetric error detecting code. Check bits are defined to be either the bitwise complement of the number (in binary) of information bits which are 1's, or as the number (in binary) of information bits which are 0's. If the number of information bits is 2^k–1, then the two forms are identical. Subsequently, Freiman [Frei62] proved that Berger's is an optimal systematic code, *i.e.* no other systematic asymmetric error detecting code with fewer check bits exists.

Asymmetric codes are also *unidirectional error detecting* (UED) codes [Bose81]. A UED code which detects unidirectional errors in at least t bit positions, a t-UED, may require fewer check bits than a UED which detects all unidirectional errors, an AUED. Borden describes an optimal non-systematic t-UED code [Bord82]. It is basically the union of a set of w-out-of-n codes for all $w \equiv \lfloor n/2 \rfloor \bmod (t+1)$. All codewords are either unordered or distance $d>t$ apart.

Many other UED codes have been presented ([Bose82c, Bose81, Bose82b, Bose84b, Bose85, Bose86a, Bose87, Lin88, Blau88, Smit84, Jha87a, Jha89a, Niko86, Tao88]). Each offers various degrees of error detection, and some provide additional properties such as: optimality, error correction, burst error detection, systematic construction, or TSC checker design. TSC checker designs for various unordered codes have also been presented by several authors, including: m-out-of-n codes ([Ande73, Bose84a, Maro77, Son81]), 1-out-of-n codes ([Gola84, Khak82b, Tao87]), two-rail codes ([Son81, Min88]), Berger codes ([Son81, Ashj77, Lo88, Maro78, Pies87]), Borden's code ([Jha89b]), low-cost codes† ([Ashj77, Gait85a]), comparators used in duplication codes ([Tami84, Hugh84]), and in general ([Jha85, Nany88a]). In all on-line checking schemes, the checker's output is updated with each new output codeword checked, and unless the system stops instantly upon detection of an error, some other circuit must keep a persistent indicator that an error has occurred. TSC error indicator circuitry is considered in [Gait85b] and [Nany87].

2.3 TESTING STRATEGIES

In 1975 Fleisher and Massel [Flei75] mentioned the need to consider the testing of PLA's, and several others have since taken up the challenge. Most testing strategies fall into one of three categories: *off-line*, built in self test or *BIST*, and *on-line*. Some hybrid strategies have components falling into two or all three categories.

† Low cost codes are not actually unordered; they are discussed in section 2.3.4.6.

In off-line testing, normal operation of the circuit is suspended and an external tester is used to apply a set of test patterns or vectors to the CUT's inputs while monitoring the CUT's outputs. If the CUT produces the correct output response for all test vectors, then it is considered fault-free. A test set generation procedure is used to determine a set of test vectors which is sufficient to expose all modeled faults; clearly small test sets are preferable. Conventional test set generation methods, such as Roth's D-algorithm [Roth66], where the PLA is treated as its random logic gate equivalent, are not appropriate due both to the large amount of reconvergent fanout and to the inadequacy of the single stuck-at fault model used [Osta79].

For BIST, extra circuitry is added which can generate a sequence of test patterns and collect the responses. The circuit is tested in place. It is put into a special test mode where it runs through its test sequence and outputs a single pass/fail indicator when complete. The normal operation of the circuit is suspended during test mode.

With on-line testing, the CUC's outputs are checked continuously during normal operation. Outputs generally must be encoded in some error detecting or correcting code, and code checking must be performed by extra on-chip circuitry.

Specialized deterministic and probabilistic test set generation algorithms designed for PLA's are discussed in subsection 2.3.1, followed by design for testability (DFT) techniques for off-line, BIST and on-line testing strategies in subsequent subsections.

2.3.1 Test Set Generation

The standard practice for IC testing has long been off-line testing where a test engineer designs a set of test vectors which are applied to the circuit under test. Actual circuit outputs are compared to a dictionary of known good responses. Muehldorf and Williams [Mueh77] described a method for generating optimal test sets for stuck-at faults in PLA's. However, Cha [Cha78] was the first to consider bridging and crosspoint faults as well. Cha suggested the following framework for a test vector generation algorithm:

- enumerate faults
- for each untested fault
  - justify input conditions which set up the fault and propagate an error indicator to some output along a sensitized path
  - assign don't care input conditions to form a test vector in such a way that it might also test some other yet untested faults
  - simulate untested faults to determine additional faults actually tested

Cha observed that if the crosspoint faults are handled first, then most of the stuck-at and bridging faults are also covered. Several choices for optimization exist, including: the order in which untested faults are processed, which of several available propagation paths is used, and the actual assignment of those inputs which may be assigned arbitrarily. Undetectable faults may be encountered. These are faults where the faulty circuit realizes the same function as the good circuit. A single fault test set which is derived from the structure of a good circuit, if applied to a circuit containing an undetectable fault, might no longer expose all detectable faults [Rama83, Agar80, Osta79].

Others devised test set generation algorithms with similar form. Consider a PLA in terms of the cubical representation of its product terms. Any single crosspoint fault involving cube c either enlarges c or removes some or all of c. In either case a single cube, say $\hat{c}$, specifies the affected minterms, and a test for the fault necessarily must be contained within $\hat{c}$. Other cubes of the PLA specification may conspire to mask the effect of the fault by being defined to assert the same outputs that appear or disappear due to $\hat{c}$. The set of all vectors which are tests for this fault is: $T_{\hat{c}} = \hat{c}\,\#(ON - c)$, where ON is the ON-set of the function realized by the PLA. Ostapko and Hong [Osta79] also analyzed the relationship between crosspoint faults and stuck-at and bridging faults, and developed heuristics which make the selection of one test vector from $T_{\hat{c}}$ and fault simulation more efficient. They too found that a single crosspoint fault test set, T_C, would detect most of the other faults. They note that about 7 percent of all crosspoint faults in a "typical" PLA are undetectable. Smith [Smit79] developed another algorithm and also proved that T_C

would additionally detect all single stuck-at faults except those on redundant product term lines and on any output line which is asserted by every product term.

Both [Osta79] and [Smit79] use heuristics to select a single test vector from $T_{\hat{c}}$ for each fault, without actually performing the expensive computation of $T_{\hat{c}}$. Somenzi, Gai, Mezzalama and Prinetto [Some84] present a similar algorithm which they claim produces better results and uses less computer time. The significant distinction of their approach is that they essentially compute one cube of $T_{\hat{c}}$ for each fault and select the test vector from that test cube. Their procedure for computing the test cube is less expensive than finding a complete solution for $T_{\hat{c}}$, and is designed to produce large cubes which potentially contain tests for other faults as well. Their algorithm also handles folded PLA's.

Smith [Smit79] additionally analyzed PLA's to determine coverage of multiple crosspoint faults, finding that T_C also detects:

- any detectable combination of *A* faults,
- any detectable combination of *S* faults,
- any detectable combination of *G* and *D* faults in a *G-D* irredundant circuit,
- any detectable combination of *S* and *A* faults in a *S-A* irredundant circuit.

In a *G-D irredundant* (*S-A irredundant*) PLA, all single *G* and *D* (*S* and *A*) faults are detectable. Each form of irredundancy is achieved at the expense of the other.

Agarwal [Agar80] further analyzed multiple crosspoint faults detectable by T_C. A fault f_1 is said to be *masked* by fault f_2 if all test vectors in a test set which expose f_1 fail to do so if both faults are present. Agarwal points out that the multiple fault $f_1 \cup f_2$ is still detectable via the test for f_2, but he fails to mention that there may also exist a test vector, $t \notin T_C$, which detects f_1 and is not subject to masking.

Agarwal proved the following basic requirements for a detectable crosspoint fault to be masked by another. A fault on a product term line which causes some or all of that terms's outputs to erroneously be asserted can only be masked by other fault(s) on the

same line which either deselect the term or disconnect it from those outputs. A fault on a product term line which causes some or all of that term's outputs to erroneously be unasserted can only be masked by fault(s) on *other* product term lines which reassert the missing outputs. He further shows that a particular arrangement of four-way masking faults is a minimal configuration for the faults to mask each other; thus implying that T_C is guaranteed to detect all combinations of 2 or 3 detectable crosspoint faults. For $r \geq 4$ faults, and $s=m+n/2$, at most $\binom{p}{2} \cdot s^4 \cdot \binom{2ps-4}{r-4}$ faults of multiplicity r out of a total possible $\binom{2ps}{r}$ are undetectable by T_C†.

For a particular example PLA, the author reports "that more than 98 percent of all multiple contact faults of size 8 and less in an irredundant PLA are covered by each T_C of the PLA" [Agar80]. This result, however, is also dependent on the number of product terms p. A PLA, P_{1920}, with $p=12$, $m=50$, and $n=60$ has the same number of crosspoints (1920) as Agarwal's example, but his formula predicts that at most 34 percent of multiple faults of size ≤ 8 are undetectable by T_C; *i.e.* an effective lower bound of only 66 percent detectable multiple faults.

In fact, while Agarwal computed an upper bound on the number of undetectable multiple fault configurations, Rajski and Tyszer [Rajs85] derived a lower bound on the number of detectable multiple fault combinations. While their results are similar to Agarwal's, they clearly show that multiple fault coverage "unexpectedly" decreases with decreasing number of product terms, but not as pessimistically as estimated by Agarwal. For P_{1920} their bound predicts that at least 80 percent of size ≤ 8 faults are detectable by T_C. The same authors further note [Rajs86] that multiple faults which contain a four-way masking cycle may still be detectable due to other unmasked faults. Their new bound for P_{1920} is 95 percent, and in general more than 99 percent of faults of size ≤ 8 are detectable by T_C in PLA's with at least 24 product terms.

---

† [Agar80] and [Rajs85] use the symbols n, m, p in place of m, p, n which are used here.

A single crosspoint fault test set is therefore capable of detecting nearly all crosspoint faults. Both [Agar80] and [Rajs85] refer to a work by Goldstein [Gold77] in which it is shown to be sufficient to consider only faults of size 8 or less for most chips manufactured using current technologies.

Although PLA test set generation algorithms are straightforward to implement, they still require considerable computational effort. In other testing arenas, random test vectors have been found to provide reasonable coverage with low development cost. A PLA with its high fan-in AND array is unsuited to random vector testing. To test a k-input AND gate for stuck-at faults requires one test where all inputs are high, and k tests each where all but one input is high. Eichelberger and Lindbloom [Eich80] call these k+1 tests *embryonic* tests. The likelihood that a random test is useful is thus $\frac{k+1}{2^k}$. They suggest deterministically generating the embryonic portion of a test vector for a particular fault and randomly choosing the remaining inputs. Several random assignments may be required before an actual test is found. Once a test vector is determined, the PLA is simulated for each remaining untested fault to see if this test detects any other faults. As the process continues, each new test generated tends to cover fewer additional faults. Their approach simplifies test vector generation by eliminating the path sensitization and justification steps, at the expense of a larger test set and increased fault simulation.

Wei and Sangiovanni-Vincentelli [Wei86] propose trying a fixed number of random assignments for each embryonic condition, to obtain high fault coverage at low computational expense, then switching to a heuristic driven deterministic algorithm for the remaining faults. For the deterministic portion they use $\hat{c}\ \#(ON{-}c)$ for test generation, and fault simulation to determine additional coverage.

Another test generation procedure which employs the concept of embryonic conditions, and avoids the computationally expensive sharp operator, is Robinson and Rajski's [Robi88] backtracking search algorithm. For each embryonic condition, they select an output line with the fewest OR-plane crosspoints as the target of a sensitized

path. All product terms connected to that output, other than the one being tested, comprise potential masking cubes. An iterative procedure assigns values to the remaining unassigned inputs. If any unassigned input is unate in the remaining masking cubes, then it can immediately be assigned to inhibit those cubes which are dependent on that input. Further, if any masking cube can only be inhibited through one remaining unassigned input, then that input must be assigned to do so. The authors provide experimental evidence that 99.76 percent of the attempts to generate a test vector for a fault were satisfied using just these mandatory operations. If masking cubes remain, the algorithm must make a choice and branch. An input assignment is chosen which would most greatly reduce the total number of minterms still capable of selecting a masking cube (since masking cubes may intersect, only an estimate which is the sum of such minterms for each cube is used). Whenever any masking cube is all X's in the remaining unassigned inputs, backtracking is required. Once all masking cubes are inhibited, any remaining unassigned inputs may be used to detect outstanding shrinkage faults on the same product term, otherwise they are randomly assigned. Fault simulation determines additional faults covered by the test. The tests in the final test set are then re-simulated in reverse order of creation, and any test which does not increase fault coverage is culled.

The approach to generation of T_C considered by Bose and Abraham [Bose82a] is designed to avoid fault simulation and test selection heuristics, while producing a minimal test set. A minimal T_C is found by systematically identifying all potential test cubes for each fault (*i.e.* the sets denoted $T_{\hat{c}}$ earlier), then finding the intersections of these test cubes (*i.e.* cubes which stand as tests for more than one fault) and finally seeking a minimal set of test cubes which cover all the faults. The authors claim the algorithm to be $O(mp^2)$.

2.3.2 Augmented PLA's Designed for Testability

Several authors have presented ways to augment PLA's to achieve various goals of enhanced fault coverage, easier test set generation, function independent test, or to optimize factors such as: testing time, test set storage space, silicon area overhead, number of test pins, and degradation of normal operating speed. *Function independent test* means that one universal test set/sequence may be used to test any PLA regardless of the actual function realized by the PLA.

2.3.2.1 Explicit Control of Bit and Product Lines

Hong and Ostapko [Hong80] and H. Fujiwara and Kinoshta [Fuji80, Fuji81] separately arrived at similar PLA designs where all crosspoints are testable by a fixed test set T_C independent of the function realized by the PLA. The basic premise is to provide individual control over the bit lines and the product term lines. To test AND-plane crosspoints all bit lines except one are disabled. An extra product term line is programmed to ensure that each bit line selects an odd number of product term lines; this is verified using a parity checker. A control decoder driven by extra control inputs, is used to select one of the PLA's input decoders, and this, along with the use of the normal inputs to the selected decoder, selects a single bit line [Hong80].

Instead of using a control decoder, H. Fujiwara and Kinoshta [Fuji80] use a pair† of control lines which can be used to inhibit either all true bit lines or all complemented bit lines. These control lines, along with the appropriate choice of PLA input test vectors, also ensures that only one bit line is selected at a time.

Both schemes introduce a shift register which overrides the AND-plane to allow the selection of exactly one product term line in the OR-plane. An extra output line is programmed to ensure that each product term line drives an odd number of outputs; this is verified using a parity checker. The outputs from the two parity checkers, rather than the PLA outputs themselves, are monitored by the tester.

† If two-bit decoders are used then four control lines are required.

Test set T_C is function independent and detects all single stuck-at and crosspoint faults, as well as a large range of multiple faults in either the PLA or the added circuitry. This philosophy of exercising direct control over individual bit and/or product term lines will be seen to reappear frequently.

Saluja, Kinoshita and H. Fujiwara [Salu81, Salu83] describe a variation of the method of [Fuji80] targeted towards multiple fault detection and elimination of parity checkers. A second shift register is introduced to control the bit lines and permit testing of the decoders. Two test outputs are defined: Z_Π, the AND of all bit lines, and Z_Σ, the OR of all product term lines. Monitoring the test and normal outputs while applying a function independent test set is shown to detect all single or multiple stuck-at and crosspoint faults except for undetectable OR-plane faults. Test responses are function dependent.

Crosspoint testability achieved in [Salu81] may be re-examined in terms of a redefined definition of the inputs. Normally a product term is specified by a cube with elements from {0, 1, X} and an input pattern is a binary vector made up of {0, 1}. Each input is decoded and drives two bit lines in a 1-out-of-2 code, {01, 10}. With individual control over all bit lines, the two bit lines associated with any input may also take on values from {00, 11}. An *effective test pattern* can be expressed as a cube t' made up from {ε, 0, 1, X}. Considering the AND-plane realized as a true AND array, a product term with cube c can be asserted in the presence of 00 on x_i's bit lines only if it has no connection with either bit line; *i.e.* it is independent of x_i, or c_i=X. Likewise 11 on x_i's bit lines, can not inhibit the product term regardless of c_i. Let 00 correspond to t'_i =X and 11 correspond to t'_i =ε.

A product term, with corresponding input cube c, is selected by effective test t' if, and only if, $c \supseteq t'$. An ε component is covered by all other values; only X covers an X. Test patterns which are all ε except for a single 0 or 1 obviously excite the product term line through a single crosspoint, thus the presence or absence of that crosspoint is directly

indicated by the value of the product line. In [Salu81] (as with [Hong80] and [Fuji80]), a shift register permits disabling all product lines except one. The OR of all product terms, Z_Σ, makes the result of a crosspoint test observable externally. With a single product term selected, OR-plane crosspoints for that term are read directly from the PLA outputs.

H. Fujiwara [Fuji84b] subsequently went on to identify several deficiencies with the design of [Salu81], namely: the area penalty for second shift register, operating speed penalties, no consideration of bridging faults, and function dependent test responses. Fujiwara proposed that shift register control over bit lines be replaced by the reintroduction of the two control lines of [Fuji80] which can inhibit either all true or all complemented bit lines. This capability, along with appropriate input patterns, can also produce the effective test patterns which are ε in all coordinate positions except one.

2.3.2.2 Explicit Control of Product Lines Alone

Khakbaz [Khak84a] also addressed the bit line control shift register and speed penalties of [Salu81], doing so without requiring any extra hardware to control individual bit lines. Khakbaz claims that "the key to the design of easily testable PLA's is the ability to control each bit line and each product line individually", as exemplified by previous designs, but his design demonstrates that control over just the product term lines is sufficient. Test output Z_Π is unnecessary, too. His "trick" is to apply a test vector which selects product term p_j while simultaneously using the product term control shift register to disable all product term lines except p_j, then to toggle each input bit one at a time. When the test vector is initially applied, Z_Σ should be 1. As each input x_i is toggled, Z_Σ is expected to change only if there is supposed to be a crosspoint on p_j for x_i.

Biswas and Jacob [Bisw85] refine Khakbaz's scheme by eliminating Z_Σ and observing the PLA outputs directly, and by only toggling individually inputs which are supposed to have a crosspoint on p_j. All independent inputs can be toggled together because any extra crosspoint will cause p_j to be deselected.

Bozorgui-Nesbat and McCluskey [Bozo84, Bozo86] argue that a product term control shift register although fine in theory is impractical to implement because the height of a register cell is larger than the spacing between product lines, creating wasted chip area†. They suggest an alternative method for selecting individual product term lines. It is similar to [Hong80] and [Rama82] (*cf.* section 2.3.2.5) in that extra control inputs are defined. During normal use of the PLA, the control inputs are disabled. The control inputs are defined to ensure that the PLA has the following properties: for each product term p_j there exists a "main" minterm t_j which selects p_j and no other term, and all minterms adjacent to t_j either select term p_j or none at all. In other words, the hamming distance $d(t_j,t_{k_i})\geq 2$ for all minterms $t_{k_i}\subseteq p_k$, $k\neq j$. The test sequence is similar to [Khak84a]: apply a main test vector which selects p_j all by itself even in the presence of a single crosspoint fault, then toggle each input bit one at a time. Test results are obtained by observing the normal PLA outputs.

The problem of defining a minimal number of control inputs to satisfy the above properties (for [Bozo84]) is NP-complete. Heuristics are used to choose, for each p_j, a minterm to promote to "main" status, and to synthesize the connections for each new control line. Control inputs are added incrementally until the testability criteria are met.

Bozorgui-Nesbat and McCluskey [Bozo84] report lower area overhead for their design compared to shift register designs. Control input definition is function dependent but usually results in fewer extra input lines than simply programming the control inputs in a decoder-like fashion. Such a decoder design is presented by Saluja, H. Fujiwara, and Kinoshita [Salu85a, Salu87b]. A PLA with p product terms has $\lceil \log_2 p \rceil$ control inputs programmed so that each product term recognizes exactly one of the input patterns corresponding to binary vectors representing the numbers $0..p-1$. One more control input is

† Others have shown that one register cell which is as tall as two product lines is feasible. The shift register can be placed on both ends of the product term lines [Bosw85a]. Alternatively, since the pattern in the shift register usually is all 0 save a single 1, one register cell can control two product lines with the addition of a little steering logic [Gras83, Hua84, Treu85a, Sali85].

programmed to recognize the parity of the values recognized by the other control inputs. Since the minimum distance between any two elements of a set of distinct binary vectors is 1, and since the parity of any two distance 1 vectors differs, the control inputs themselves, being minimum distance 2 from each other, form the basis of "main" minterms.

Saluja *et al.* call such a configuration a Decoder-Parity-AND-Array, or DPAA. Design simplicity is achieved at the expense of area overhead compared to [Bozo84]. However, Saluja *et al.* introduce a partitioning method (described in the next section) which reduces area overhead.

2.3.2.3 Product Line Partitioning

Saluja *et al.* [Salu85a, Salu87b] describe a design that partitions the product term lines into k blocks of p/k lines (except for perhaps one block which is smaller), and programs each block with a DPAA, all blocks sharing a now reduced number of control inputs. A k-bit shift register is added to select one block at a time during testing. They show how to find an optimal value for k. It turns out to be less dependent on p than on s, the ratio of the size of a shift register cell versus that of a PLA cell. They suggest that their partitioned design has less overhead than [Bozo84] unless the PLA is small or s is large. They also point out that the scheme of [Bozo84] would too benefit from shift register controlled product term partitioning.

With the objective of reducing both area overhead and test length, yet another easily testable PLA design is advanced by Boswell, Saluja and Kinoshita [Bosw85a, Bosw85b, Salu85b]. Most of its elements are familiar: individual control over bit lines (they use a shift register, but any other method would also work), test output Z_{Π}, partitioned product terms and a second shift register to select a partition. For the previously described schemes, all of the bit lines must be exercised for each product term. In this scheme, as bit lines are exercised, all product terms within the selected partition are tested in parallel for multiple stuck-at and crosspoint faults. Extra inputs, product terms or outputs are not

required, but adding inputs or outputs may simplify the task of partitioning or permit larger size partitions. The product terms within a partition must satisfy these criteria:

- each product term must control the sole crosspoint on some output line,
- the (input part of the) product terms must be unordered,
- if more than one product term line have crosspoints on an output:
  - some test vector must select none of these product terms, and
  - some test vector must uniquely select each one of these product terms.

Their proof that these conditions permit parallel testability is by example.

2.3.2.4 Explicit Control of Bit Lines Alone

Rajski and Tyszer [Rajs84] show that control over just the bit lines also suffices to make a PLA easily testable. Additionally the hardware overhead is less since there usually are fewer inputs than product terms in a PLA. They provide a flip-flop for each input; all flip-flops are loaded in parallel via the primary inputs during test mode. Again in terms of an AND-OR array, simple logic based on the input and its flip-flop drives the pair of bit lines for each input so that they either represent the normal true/complemented value of the input, or are both 0. This latter condition corresponds to an effective test vector with an X in that position. Again a product term p_j with input cube c is selected by test t' if, and only if, $c \supseteq t'$. Assuming that no two product terms have the same input cube c, the effective test $t'=c$ will only select p_j, and only if there are no extra crosspoints in the positions where $t'_i = c_i = \mathrm{X}$. Similarly another effective test, which is the same as c except that one $c_i=0$ or 1 component is replaced by $t'_i = \mathrm{X}$, will not select p_j unless the crosspoint associated with x_i is missing. As long as no other product term covers c, both types of tests are decidable by observing the PLA outputs.

Procedure 2 described by Reddy and Ha [Redd85, Redd87] is equivalent to Rajski's scheme, except they use a pass transistor before the inverter on each complemented bit

line. Once the transistor is cut off, the input capacitance of the NMOS inverter acts as a dynamic storage cell. All multiple stuck-at, crosspoint and bridging faults between adjacent product term and output lines are detected.

They also present Procedure 1 to develop a test set which is generally smaller and guarantees to detect all bridging faults. For this procedure, effective tests of form $t'=c$ are still used to detect all extra crosspoints, but missing crosspoints are tested by effective tests derived from implicants of any realization of the complement of the PLA function. If $\tilde{c}$ is such an implicant, then test $t'=\tilde{c}$ except $t'_i=\varepsilon$ wherever $\tilde{c}_i$=X. The same authors [Ha85] also adapt this scheme for use with dynamic CMOS PLA's.

2.3.2.5 Function Redefinition

Pradhan and Son [Prad80] define two extra output lines which ensure that no single bridging fault remains undetectable. They also show that a nonconcurrent PLA's test set, T_C, detects all single stuck-at and bridging faults which are detectable [Son80]. One extra output line may be required, and the product terms must be arranged such that no two adjacent product lines have identical connections in the OR-plane. The increase in product term lines needed to achieve this nonconcurrency is a serious drawback.

Ramanatha and Biswas [Rama83, Rama82] argue that the presence of undetectable faults may invalidate a test set produced by any of the above methods which are based on a single crosspoint fault assumption. They propose a procedure to augment a product term irredundant PLA with extra control inputs to make it crosspoint irredundant. A PLA is *product term irredundant* if every product term p_j contains a distinguished minterm t_j such that when t_j is applied at least one output is asserted by no product term other than p_j. A PLA is *crosspoint irredundant* if all crosspoint faults are detectable. Each product term of a PLA corresponds to a cube in the specification. The control inputs are determined so that for each fault on a product term, there exists a test vector which is not covered by any other product term's cube (considering both input and output parts of the

cube). A candidate test vector, t, is picked from $\hat{c}$ (as defined in section 2.3.1) for an undetectable crosspoint fault on product line p_j, and an output line is chosen to expose the fault (this latter choice is based on a heuristic which determines which available output is masked the least). Then cubes corresponding to all product terms which mask any undetectable fault on p_j are identified. Positive (unate) control inputs are next defined such that the masking terms no longer cover t: p_j is programmed with an X for the new input while all its masking terms are programmed with 1's; t is assigned a 0 for that input, while test vectors for other faults have a 1 in the same position. Two product terms are said to be compatible if neither ever masks an undetectable fault on the other. Maximal compatible sets of product terms can all share the same control input; *i.e.* all have an X programmed for that input. The authors show how to add input, product term and output lines to a crosspoint irredundant PLA to ensure complete testability of stuck-at and bridging faults by test set T_C while preserving crosspoint irredundancy.

A PLA with certain properties is shown by Min [Min84] to be testable for all combinations of stuck-at, bridging and crosspoint faults. Furthermore, the easily generated test set is the same as [Bisw85] described earlier. However Min's method does not use a shift register to control the product terms, rather, like [Bozo84], it prevents fault masking, but does so by redefining product terms and adding outputs rather than by adding inputs as does [Bozo84].

The first property a PLA must satisfy is *pseudo-nonconcurrency* which requires that for any given minterm, each product term selected asserts some output which no other selected product term asserts. The second property is that the output parts of the cubes (which correspond to product terms) must be unordered. Finally, the PLA must be *separate* which means that if $d(c^i \cap c^j, c^k)=1$ then the combined set of outputs asserted by c^i and c^j can not cover those asserted by c^k. In other words, no set of intersecting cubes whose intersection contains a minterm adjacent to some minterm of c^k covers the same outputs as does c^k.

The paper gives a proof showing that a PLA with these properties is testable for all faults by a test simply derived from the cubical specification of the array, and also gives algorithms to transform a normal PLA to one with the needed properties. Product terms may be broken into several cubes to satisfy pseudo-nonconcurrency while extra outputs may be defined to ensure the other properties.

2.3.3 Built In Self Test — BIST

The origin of BIST can be traced back to Sedmak's concept of "self-verification" which sought to capitalize on on-line strategies to help address the ever increasing difficulty of test generation and application for off-line testing [Sedm79, Sedm80]. For random logic, it is often either not possible or too expensive to have on-line checking detect 100 percent of errors during normal device operation. Hence Sedmak suggested adding an internal test sequence generator to fully exercise the circuit in test mode so that the on-line checkers should detect at least one error if a fault is present. Today many BIST techniques replace the on-line checking hardware with simpler circuitry to compress the sequence of responses into a characteristic signature which can be compared to that of a known good circuit. Several built in self test strategies have been designed for use with PLA's. The key components of BIST are economical test sequence generation and response compression which is capable of detecting modeled faults.

2.3.3.1 Universal Test Sequence Techniques

Several BIST designs have been presented which extend the easily testable PLA concept of exercising control over individual bit and product term lines. Daehn and Mucha, [Daeh81], treat a PLA as two separate NOR-arrays for the purpose of testing. The authors show how a nonlinear feedback shift register can be constructed to generate a walking 1's sequence to feed the NOR-array's inputs. Patterns which are all 1(0)'s except

for a single 0(1) are easy to generate using a shift register, and are often called *walking* 0(1)'s. A parallel input signature register compresses the NOR-array's response to the test sequence, and is shifted out for off-line verification.

A built-in logic block observer, or *BILBO* [Köen79], structure combines test generation with compression. As illustrated in Figure 2.3, $BILBO_1$ generates test vectors for the AND plane, $BILBO_2$ collects results from the AND plane and, at a different time, generates tests for the OR plane, and $BILBO_3$ collects results from the OR plane. $BILBO_3$ can generate tests which are fed through the output inverters, back into the input decoders and compressed by $BILBO_1$. Stuck-at, crosspoint, and bridging faults are detected, even in the presence of redundancy.

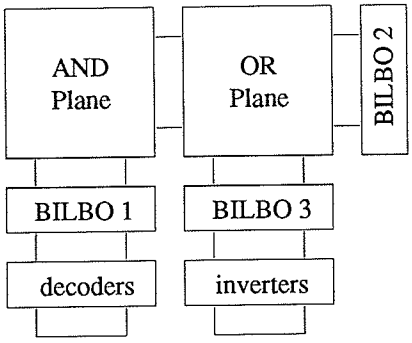


Figure 2.3 BIST architecture of [Daeh81].

Yajima and Aramaki, [Yaji81], designed a clever extension to the function independent augmented PLA of [Hong80] and [Fuji80]. Individual bit line selection is performed by a second shift register as suggested in [Hong80]. A *universal test sequence* is generated which walks 1's and 0's along the bit lines as each product term in turn is uniquely selected. A few extra lines along with the AND and OR-plane parity are combined to determine the next bit to shift into the product term control shift register. Once an error indicator enters the shift register it tends to cause subsequent tests to fail so that at the end of the test sequence the final pattern in the register differs from the error

free terminal state. While Yajima and Aramaki make no claim as to area overhead for this design method, 99 percent is reported in [Hass83].

Another similar design, by Hua, Jou and Abraham, [Hua84], is also based on the augmented PLA of [Hong80] and [Fuji80]. Shift registers control the bit and product lines and the parity checker outputs are used as error indicators. In order to achieve a feasible floorplan, each shift register cell is demultiplexed between two lines it controls. Area overhead for an example PLA is 20 percent. Grassi and Pfleiderer, [Gras82, Gras83], describe the design of an actual PLA chip which uses demultiplexed shift registers to generate the universal test sequence, while outputs are compressed using a multiple input signature register or *MISR* (see [Köen79]). Area overhead of 44 percent was required for the implemented PLA, but overhead reduces as PLA size increases. Counters and decoders to select one bit or product line at a time, for test pattern generation, have been suggested by Salick, Mercer and Underwood [Sali85].

In the same paper where he presented his design for a PLA with an easily generated function independent test, described earlier [Fuji84b], H. Fujiwara suggested that a function independent response indicator could be produced, thus making the design suitable for BIST. The response sequence itself is function dependent, but if it is fed into a parity counter register† which is observed at nine particular points during the test, then the observed values are function independent. These nine bits can easily be collected and compared to their expected value. Recall that this scheme eliminated parity checkers and replaced the shift register which controls the bit lines with two control lines which can disable either all true or all complemented bit lines at once but requiring that the inputs be driven too. This technique is a form of crosspoint counting.

This concept was refined and implemented by Treuer, H. Fujiwara and Agarwal [Treu85a, Treu85b, Treu87]. They essentially put back the shift register on the inputs to

† A *parity counter* register simply computes the parity of a serial stream of bits, EXOR'ing each successive bit with the current register contents.

generate walking test vectors, but only one cell is required for every two bit lines, and output parity is again computed. The PLA is augmented as before with an extra product term and output so that each bit line has an odd number of crosspoints and each product term is connected to an odd number of outputs. The universal test sequence is generated and the output parity for each pattern is accumulated in a one bit parity counter register. The contents of this register are examined at $2m+2p+1$ specific points during the test sequence, and furthermore, the pattern of alternating 0's and 1's which should be observed in a fault-free PLA is easily recognized. Area overheads as low as 15 percent are reported as compared to 20 percent for the same PLA's using [Hua84]. The authors show that all single faults are detected and only $\frac{1}{2^{2n+p}}$ of all multiple faults are missed. Any multiple crosspoint fault which consists of an even number of faults on one bit line or on one product line in the OR-plane is undetectable.

Yet another example of the philosophy of using the universal test sequence to enable one crosspoint at a time is that of Saluja and Upadhyaya, [Salu86, Upad88]. Shift registers are used, but the output lines are controlled along with the bit lines. The two shift registers now uniquely select each crosspoint, and its presence or absence is made visible by OR'ing together all product and output lines. The number of crosspoints on each product line is counted and the sequence of count values is compared to that recorded in another PLA. Product terms are arranged so that successive ones tested have at most one more crosspoint than the previous. The above mentioned PLA simply controls whether a reference counter should be incremented after each successive product term is tested. "Dummy" product terms may be required to satisfy the crosspoint count constraint. Coverage of multiple faults is shown to be nearly 100 percent. The same authors along with C.-Y. Liu later implemented a computer program to generate PLA's using the above design [Liu87]. They reported that a set of 22 PLA's had average overhead of 30 percent while an average of 40 percent was required by [Treu85a] for the same set.

A variation of the crosspoint counting strategy for BIST of CMOS PLA's is presented by D. Liu and McCluskey [Liu88]. Shift registers again generate the universal test sequence, each line is arranged to have an odd number of crosspoints, and Z_Σ is computed. However, a parity counter is placed on each output line, plus one for Z_Σ. This latter counter is used to count AND-plane crosspoints on each of the product lines as they are selected one at a time. The parity counts are shifted out for external verification. They suggest an interesting method of product term control: a $k=\lceil p/4 \rceil$ stage Johnson or twisted ring counter is decoded (along with the toggle flip-flop used to divide its clock by two) to produce a $4k$ bit walking 0's sequence. The authors report lower area overhead and shorter testing time as compared to [Treu85a].

2.3.3.2 Exhaustive and Random BIST Techniques

"An m input combinational network can be tested thoroughly by applying all 2^m input combinations and verifying that the correct output is obtained for each", wrote McCluskey, [McCl82]. Hassan and McCluskey [Hass83] suggest using a maximal length linear feedback shift register, LFSR, to generate an exhaustive test sequence, and to collect responses using three MISR's, one each on the true and complemented bit lines, and one on the outputs. They claim lower area overhead than [Daeh81]; 23 versus 87 percent in one example. Exhaustive testing is rarely practical as the number 2^m is too large.

However, McCluskey [McCl82, McCl84] also suggested an alternative *pseudoexhaustive* method called "verification testing." Suppose an output is dependent on only k inputs, then any 2^k patterns which represent all combinations of these k inputs is a verification test for that output. He claims that such a test has reasonable fault coverage. For any two inputs, if no output is dependent on both, then those inputs are compatible. A maximal set of compatible inputs may all be driven by the same output of a counter during testing. A minimal cover of inputs by maximal compatibles determines the minimum number of counter stages required, and hence the testing time.

Ha and Reddy [Ha86, Ha88] applied pseudoexhaustive techniques for BIST of PLA's. Inputs are partitioned similarly to the way [Salu85a] partitions product lines, *i.e.* inputs are grouped into h blocks of $k=m/h$ input lines each, a single k-bit counter drives all blocks, and a h-bit shift register enables one block at a time. They suggest using $k=2$. The product lines are also partitioned (heuristically), but in such a way that no two terms in a block have intersecting output parts. The response to a pseudoexhaustive test sequence is compressed by a MISR. The sequence consists of an exhaustive test for each combination of input and product line partition. For a sample of 30 PLA's, their method required lower average area overhead, 14.5 versus 24.4 percent, and remarkably had a 69 percent shorter average test length than [Treu85a].

Another approach in avoiding exhaustive testing is to use a pseudorandom sequence of test patterns, which can be easily be generated using LFSR's [Köen79] or cellular automata arrays [Hort89]. Hopefully, a sequence of tests which is considerably less than 2^m in length will contain sufficient test patterns to expose faults in the circuit. Outputs are usually compressed into a signature by a MISR. Random test patterns are ineffective for PLA's as noted earlier [Eich80]. However, Eichelberger and Lindbloom, [Eich83], proposed a design where inputs and product lines are grouped into blocks, and as well as generating a random input pattern, they randomly enable only one input group and one product block. They suggest that product term blocks either be of size 1, or that each term in the block drive some output not driven by any other term in the block.

Ha and Reddy [Ha86] improved on this design by running a weighted random test sequence on all inputs once for each product term partition enabled by a shift register. The random patterns, which drive bit lines, are weighted so that 0's arise more frequently than 1's. Product terms are partitioned so that the average number of AND-plane crosspoints in each block are about the same.

A third design for random pattern testable PLA's is presented by H. Fujiwara [Fuji88a]. He suggests using another programmable NOR array whose outputs have programmed connections on the normal bit lines of the PLA. Control inputs select one or more terms from this "mask" array, and all bit lines connected to these mask terms are disabled (pulled down to logic 0). Random patterns drive both the normal and control inputs, thereby masking random collections of inputs and making the PLA less resistant to random testing. The personality of the mask array should have an influence, but the author provides no theory for its design.

This idea is extended by H. Fujiwara, Fujisawa and Hikone, [Fuji88b], in two ways. Firstly, the NOR array recognizing control input patterns to selectively mask bit lines is eliminated; the control lines themselves now directly control the crosspoints on the bit lines. Secondly, additional control lines mask selective product term lines in a similar manner. Again, both normal and control inputs are driven with random vectors. Determining an optimal set of masking connections is said to be a "hard" problem, so they use a simple heuristic which strives for uniform masking by each mask control line. For eight large example PLA's they report achieving greater than 90 percent fault coverage with low hardware overhead, using 50,000 random patterns and just 2 or 4 masking control lines for each plane.

The *weight* or one's count, $W(F)$, of a single output boolean function is defined to be the number of ones in the truth table. The *syndrome* of an m-input function is defined $S(F)=W(F)/2^m$. Serra and Muzio [Serr85, Serr87] use spectral techniques to show that a weighted sum of syndromes of a PLA's n outputs, $WSS=\sum_{i=1}^{n} w_i W(F_i)$, detects all single stuck-at, bridging and crosspoint faults. An implementation could choose w_i's which are powers of 2 and use an ordinary parallel adder to collect the sum. An exhaustive set of inputs must be applied.

2.3.4 On-line Testing Strategies

The on-line approach to testing introduces additional circuitry to continuously monitor system behaviour during normal operation. During each cycle of a synchronous system's operation, its outputs are checked for errors. Circuits (here PLA's) are designed so that if they are fault-free then they produce outputs possessing some verifiable property, and if they are faulty then it is assumed that they will produce output values which do not have the required property, at least some of the time. Usually the circuit is designed to output n-bit codewords from some error detecting code. A checker determines if each output pattern is a legal codeword. Alternatively, if faults characteristically produce erroneous output patterns with distinctive properties, then it may be possible to define a checker to recognize noncodewords instead, and to assume the output to be correct otherwise. Also, due to its regular structure, a PLA is easily partitioned and separate schemes may be used to check each partition. The problem is to find an efficient encoding and checking scheme which also detects an acceptable proportion of expected errors.

2.3.4.1 Wang and Avizienis [Wang79]

In a landmark paper, Wang and Avizienis [Wang79] set the stage for TSC PLA design research. They begin by showing how inverter-free nonconcurrent PLA's can be used to realize TSC checkers for k-variable two-rail codes and several classes of m-out-of-n codes. Each product term is programmed to exactly recognize one codeword. For the checker to be self-testing for bridging faults, adjacent product lines and adjacent output lines must be arranged to have distinct crosspoint patterns.

They also suggest a general approach for designing a TSC PLA. It must be nonconcurrent and its output patterns must belong to a code for which a TSC checker exists.

The method they propose for removing concurrency from a functional array specification involves the introduction of additional output bits. For example, if minterm m_s is

shared by two product terms, then m_s, and one other minterm which is adjacent to it from either product term, are programmed to assert the new output line. All other minterms adjacent to m_s must not assert the new output. After the PLA is made nonconcurrent, some of the extra outputs may still be unspecified for some minterms. These are fixed to either introduce new codewords (so that the checker gets enough different inputs for it to be self-testing) or to help minimize existing product terms. The introduction of additional outputs to achieve nonconcurrency, however, is not necessary. Moreover, their algorithm produces rather poor results (in terms of percentage increase of product terms). This is discussed further, and an alternative algorithm presented in section 4.2.

Once the PLA is completely specified, one must "examine all output patterns in order to select a checkable code (for which a TSC checker exists)", perhaps dividing the output bits into not necessarily disjoint groups each representing codewords from some smaller code. They suggest no way to automate this last step, and this is a serious drawback of the method.

Note that some of their design constraints may be relaxed. Consider the requirement that a TSC checker be realized by a PLA which is inverter-free and nonconcurrent. The inverter-free restriction may be lifted, as it is only there to avoid "undetectable faults that may occur when an extra crosspoint fault connects an unused inverter output to a product term" [Wang79]. However, such a fault simply causes some or all of the minterms to no longer select that product term, and this is detectable. Alternatively, if there are no crosspoints on a bit line, then that line should not be implemented. The other concern disallowing product terms with identical output patterns from intersecting (or being physically adjacent, for that matter), is excessively restrictive also. Any fault on one product line which is masked by an intersecting (or physically adjacent) product term with an identical output pattern, still produces the correct output. Furthermore, the faulty configuration can no more mask some other detectable fault than either of the affected terms can individually. Concurrency limited to product terms with identical output patterns is discussed further in sections 4.2.1 and 5.2.4.3.

2.3.4.2 Khakbaz and McCluskey [Khak82a]

In this work, a nonconcurrent PLA is partitioned into three sections, two of which naturally produce outputs in a code for which TSC checkers exist. The bit lines naturally form an m-variable two-rail code, while the product terms form a 1-out-of-p code. These two alone are sufficient to detect stuck-at and bridging faults on the bit and product term lines, and crosspoint faults in the AND plane. Under a single fault model, the remaining faults in the OR-plane will cause at worst a single bit error, which is detectable using simple parity if each codeword is arranged to have the same number of 1's. Depending on the function, this last checker may not see enough different codeword patterns to be self-testing. The input decoder design is essentially that of Figure 2.2b.

Khakbaz and McCluskey [Khak82a] claim, at least for their large example PLA, that nonconcurrency "is not necessarily as restrictive as it may appear to be," but their example PLA has little concurrency to begin with. They use the method of [Wang79] to eliminate what concurrency there is. Area overhead for the example is 37 percent relative to the nonconcurrent PLA. Time overhead is 9 gate delays. They also point out that a test set for off-line testing is easily constructed by taking advantage of the on-chip checkers.

Their claim regarding the effect of nonconcurrency is questionable, and perhaps 37 percent overhead is atypical. For any PLA which has a nasty nonconcurrent realization, not only does the PLA itself grow quickly, but so will the 1-out-of-p checker. Otherwise, for PLA's which are nearly nonconcurrent, this is a nice simple solution.

Kling and Banerjee describe a physical implementation of the above testing scheme [Klin86]. Checker design is given in terms of transistors and diodes rather than logic gates. Their equivalents of a two-rail checker on the bit lines, and of a 1-out-of-p checker on the nonconcurrent product lines, both incorporate sense amplifiers which can detect intermediate voltage levels which can be caused by orthogonal line bridge faults. Both checkers also use the idea of bus-like error indicator lines which are pulled down if

an error is sensed. The bit line checker checks all pairs of bit lines in parallel, any error pulls the indicator line low. The product term checker essentially uses a chain of checkers, one each per product line. Two error indicator lines are used, $E_{\geq 1}$ indicates that one or more product terms are asserted, $E_{\geq 2}$ that two or more are. Line $E_{\geq 1}$ forms a chain from term to term, $E_{\geq 2}$ is a bus-like line. If a product line is not asserted, then its checker simply passes its incoming value of $E_{\geq 1}$ further along the chain. If the product line is asserted, then its checker ensures that its outgoing $E_{\geq 1}$ is asserted, furthermore, if its incoming value of $E_{\geq 1}$ is asserted, then $E_{\geq 2}$ is pulled down. If exactly one product term is active, as is expected, then $E_{\geq 1}$ should be asserted and $E_{\geq 2}$ should not be. This product line "checker can be checked itself by applying a few selected 'forbidden' input vectors," and observing $E_{\geq 1}$ and $E_{\geq 2}$.

The output lines are checked using a parity checker. The authors suggest, to reduce delay, that the output be divided into k groups each with a parity check line and a parity checker. The k parity checker outputs are combined together into a single bus-like error indicator line. Their focus is towards a practical design of checkers with low area and delay overhead, rather than addressing TSC goals.

2.3.4.3 Mak, Abraham and Davidson [Mak82]

Under a stuck-at, bridging, line break and crosspoint fault model, it is shown [Mak82] that a single internal NOR-array fault will only cause unidirectional output errors. Further, it is shown that in a NOR-array, a unidirectional error at the input will cause a unidirectional error at the output; thus an AND-plane error propagates through the OR-plane. This, along with the decoder design of Figure 2.2b, ensures that a PLA will have only unidirectional errors. Therefore, if the output codewords belong to a unidirectional error detecting code, then the PLA meets the TSC goal; in fact it is strongly fault secure. Berger and m-out-of-n codes are two well known unidirectional error detecting codes which could be used. SFS checkers exist for both.

The authors introduce a *modified Berger code* where the check bits are defined to be the binary encoding of the difference between the number of 0's in the output pattern and the minimum number of 0's in any codeword. Generally fewer check bits are required.

Dong [Dong82] also defined a modified Berger code. For his design two groups of check bits are specified. The first group of check bits, $C1$, is the binary representation of $I0 \bmod k$, where $I0$ is the number of 0's in the information bits. The second group encodes $C1$ in a unidirectional error detecting code, for example a two-rail code. The advantage of this scheme is that a fixed number of check bits is used independent of the number of information bits. However, not all unidirectional errors are detected; specifically those with weight which is a multiple of k are missed.

Mak *et al.* [Mak82] prove that a complete covering is required for any PLA to be SFS or TSC. This essentially means that a **0** output pattern can not be a codeword in any unidirectional error detecting code. For a normal PLA, a **0** output is simply realized by selecting none of the product terms. For a SFS PLA, however, the **0** information bit pattern must be encoded, and product terms corresponding to the *OFF*-set of the original PLA must be defined to yield the appropriate output codeword. For some PLA's, a significant number of product terms may be required to achieve a complete cover.

A **0** output which is desired should be distinguished from a **0** which arises, due to don't care assignment during minimization, for inputs which really are don't cares, *i.e.* they are not expected to be applied. In the latter case, a **0** output could easily be recognized and treated as the detection of an invalid input.

2.3.4.4 Fuchs and Abraham [Fuch84]

This work extends that of [Mak82] to include certain multiple faults and faults on external inputs. Input patterns must now belong to an error detecting code. The PLA is designed to be code disjoint, which means that a non-codeword input produces a non-

codeword output. Since the failure in Figure 2.2a can be considered equivalent to a fault on the input lines, and since Fuchs and Abraham consider such faults, it is not necessary for them to use the modified decoder design of Figure 2.2b. If X and Z are the unencoded inputs and outputs of a PLA which is to be made on-line testable using this method, then the extended PLA accepts input $X'=(X, X_c)$, where X_c=Berger(X) denotes the Berger check bits for X, and produces output $Z'=(Z, Z_c)$, where Z_c is defined as Z_c=Berger(Z)+$\overline{X}_c$. X_c is not actually input to the PLA, but rather is used by the checker to test if $\overline{Z_c\text{–Berger}(Z)}=X_c$†, where Z_c and Z are taken from the PLA output Z'. This scheme is SFS for external and internal unidirectional errors.

Two adjacent minterms which normally produce the same output have different input check bits hence they will have different output check bits, too. This shattering of product terms will likely result in an explosion in the number required to realize Z' compared to Z. This is in addition to the overhead for check bit outputs. Fuchs and Abraham claim that the nonconcurrent strategies of [Wang79] and [Khak82a] incur too high a cost in extra product terms, but this scheme may even be worse in this regard. Evidential support for this is provided in Chapter 6.

In a subsequent paper, Fuchs, Chen and Abraham, [Fuch87], reiterate this scheme (and that of [Chen85] described in the next subsection) and provide additional implementation detail. They show that the unidirectional error model still holds for folded PLA's, and for PLA's with multiple bit input decoders. They report a 12 percent increase in size due to encoding (using the method of [Fuch84]) for an example PLA. Checking delay is reduced through pipelining; the checker output trails the PLA output by two clock cycles.

2.3.4.5 Chen, Fuchs and Abraham [Chen85]

Chen, Fuchs and Abraham [Chen85] present an on-line checking scheme which strives to reduce the number of check bits required, and to eliminate the adder tree used to

† Actually Z_c–Berger(Z) and X_c form a two-rail code which is checked by a TSC checker.

compute Berger check bits. They claim that product term line stuck-at-1 faults are "most realistically caused by a short between V_{DD} and the gate of a depletion mode transistor," *i.e.* pull-ups. If product term stuck-at-1 faults are "avoided" through conservative chip layout rules, then the error model they use reduces to a single bit error on a bit, product or output line. The bit lines are checked using a two-rail checker, and the product and output lines are checked using parity checkers. An extra output line is required for the expected parity of the output pattern, and two extra outputs predict the even/odd parity of all product term lines except for the extra product term lines required solely to produce these two outputs.

Chen *et al.* also suggest using a similar scheme with nonconcurrent PLA's. The two-rail check on bit lines, and parity check on outputs remain. Two other output lines are defined to generate a 1-out-of-2 code. Pairs of fault adjacent product terms are programmed so that one term outputs 01 and the other term 10 on these lines. Product terms are *fault adjacent* if their cubes conflict (0/1) in only one position, *i.e.* a single fault can cause them to intersect. If cyclic adjacencies prevent the definition of a 1-out-of-2 code in this manner, then another check bit is added. In general, if k check bits are required to resolve all adjacencies, then check bits are defined in a $\lceil k/2 \rceil$-out-of-k code with no two fault adjacent terms having the same check bit pattern. Any error which causes other than exactly one product term to be selected will result in a non-codeword value on the check bit output lines.

2.3.4.6 Sayers and Kinniment [Saye85]

Low cost residue codes are applied to various VLSI structural blocks, including PLA's. The k check bits in a *low cost residue* code for information word Z, treated as a binary number, are defined to be Z mod b, where $b=2^k-1$. With $k=2$, a mod 3 residue code is defined. Sayers and Kinniment [Saye85] describe two methods of using mod 3 residue codes for on-line checking of PLA's. For method 1, check bits become additional

PLA outputs, and for method 2 they are produced by a second PLA based on the same inputs. The check bits are compared to the mod 3 residue recalculated from the PLA's normal output lines. A mod b residue can be computed easily using k-bit adders with end-around carry [Wake78]. Special attention is given to the all zero vector since it is a low cost codeword, and it can potentially arise due to a fault such as a product line stuck-at-0. For method 1, false product terms must be included (*i.e.* the PLA realizes a complete cover), and their check bits are defined as 11 instead of 00. For method 2, 11 is used to represent a zero residue for non-zero outputs. Error analysis (*cf.* section 3.2.2) indicates that coverage can be quite low with many faults causing undetectable errors.

Although unidirectional errors are expected in PLA's, low cost checking can not detect all multi-bit unidirectional errors. Tao, Lala and Hartmann address this concern [Tao86]. They continue to use mod 3 check bits on PLA outputs similar to [Saye85], but like [Chen85] they also use a parity checker on product term lines. Odd parity is expected since they realize the PLA using a nonconcurrent complete covering. They allude to the fact that any single fault in the AND-plane in this configuration results in a single product line error. This is shown later as part of Theorem 3.2.

Single bit output errors are also expected, which means that parity could be used instead of mod 3. Tao *et al.* appear willing to tolerate 66 percent coverage of multiple faults in the OR-plane. This is characteristic of a mod 3 residue code. They suggest a lower bound on the number of multiple faults which are missed by the product term parity checker but are detected by the mod 3 checker. Obviously any fault which causes an even number of terms to be asserted will be detected by parity. The bound predicts that the mod 3 check bits will detect at least 45, 74, 88.3 and 94.8 percent of multiple errors which cause 3, 5, 7 or 9 product terms, respectively, to be asserted.

2.3.4.7 E. Fujiwara and Matsuoka [Fuji85, Fuji87]

A theoretical framework is established for on-line checking of arbitrary multiple output combinational circuits using a Generalized Prediction Checker (GPC). Suppose F and K are both m-input n-output combinational functions, where F is the function required for some system and K is a somewhat arbitrary function used in the checking scheme. Let $Z=F(X)$ represent the desired output for a given input value X. Let $H(x)$ represent a linear transformation (or parity check matrix) which operates on an n-bit vector, x, to produce an r-bit vector (which is a linear combination of the columns of the matrix H selected by x). Three checker circuits are designed in addition to the desired function which computes $Z'=\tilde{F}(X)$, where $\tilde{F}$ is the potentially faulty realization of F. The first, $L(X)=\overline{H(K(X))}$, produces an r-bit predictor which is compared to the output of the second, $R(X,Z')=H(Z')\oplus H(F\oplus K(X))$, by the third which is just a two-rail checker. For fault-free $\tilde{F}$, $H(Z')\oplus H(F(X))=H(Z)\oplus H(Z)=\mathbf{0}$, so R becomes $H(K(X))$. To make this checking system self-testing, and thus TSC, it may be necessary to add a control input, v, which is EXOR'ed with all bits of L and R. If $K=F$, then the case where H is the identity transformation is equivalent to complemented duplication, and the case where $H(x)$ computes the parity of x corresponds to simple parity prediction.

The primary purpose of H is to reduce the number of check bits to compare and determine the error detection ability of the checker. For example H corresponding to a parity check matrix with distinct columns is capable of detecting double bit errors. This checker design is a generalization of several previous parity prediction schemes ([Fuji83, Fuji84a, Khod79, Ko77, Ko78]). Determination of optimal K which minimizes checker overhead is an open problem.

2.3.4.8 Observations

Several issues pertaining to existing on-line testing strategies are addressed later in this dissertation. The first is nonconcurrency. It is necessary to investigate the true cost of nonconcurrency, and in the process, improve upon the algorithm of [Wang79] which converts a concurrent functional array to one which is nonconcurrent. Also, it may be possible to relax the nonconcurrency requirement for some testing strategies without diminishing error coverage.

It is necessary to investigate the cost of extending a PLA to include check bits, when the PLA outputs are encoded in an error detecting code. In addition to the extra output lines required, one must consider the increase in number of product term lines. These lines are introduced both to produce the check bit outputs and to ensure a complete cover. Due to the multiple-output minimization process, some of the extra product terms may combine with terms producing the normal PLA outputs, thus making it difficult to analyze their impact. Parity, mod 3, Berger, and [Fuch84]'s encoding are investigated later in chapters 3 and 6.

Error coverage is another issue related to encoded outputs. Results of empirical studies performed on parity and mod 3 codes are reported in chapter 3. Furthermore, fault models must be considered. This author does not accept the argument (of [Chen85]) that product line stuck-at-1 faults can be ignored, and also believes that the line break fault should be included in the model.

Area and delay overhead due to code checkers is another concern. We contend that a checker which monitors all product lines should be avoided due to the potentially large number of inputs such a checker may have. Checker area and/or delay are generally related to number of inputs. Codes commonly used to encode PLA outputs include: parity, mod 3, and Berger. The design of Berger code checkers (which essentially must count 1's or 0's) and mod 3 checkers typically involve the use of full adder cells, whereas parity checkers only require exclusive-or cells. The latter are clearly smaller and faster.

2.3.5 Hybrid Testing Strategies

The off-line, BIST and on-line strategies may be combined in various ways to produce hybrid strategies. Sedmak, [Sedm79] proposed taking advantage of on-line checking hardware to ease the burden of test generation for off-line testing. Indeed, Khakbaz and McCluskey demonstrate this in their paper on on-line checking of PLA's [Khak82a]. On-chip checkers make test set generation and response evaluation easier. Tao [Tao88] describes a PLA configuration similar to [Khak82a] in that a nonconcurrent PLA is checked by a 1-out-of-p checker on the product lines and a parity checker on the output lines. But rather than observing the checker outputs during off-line testing, Tao suggests that the product term checker have shift register cells incorporated which can disable all but one product line. Thus the PLA becomes easily testable in a manner similar to the schemes described in section 2.3.2.

Sharma and Saluja [Shar88] consider the converse situation. They suggest a general on-line checking strategy where each input applied to a CUT also is looked up in a table of test vectors, and if found, the expected response, also stored in the table, is compared with the actual CUT outputs, and an error indicator is generated. A tag register with one bit per stored test vector is used to record the fact that the test has been applied. When all tags have been set, a complete test set has been applied. The tag bits are reset and the cycle repeats. It is difficult to determine how long it will take for a complete test to be applied, but since a test vector is evaluated each time it appears regardless of whether its tag bit is already set, testing continues and is not invalidated by a few infrequent test patterns.

Combined on-line and BIST strategies have also been proposed. E. Fujiwara and Matsuoka [Fuji87] suggest using their GPC checker to evaluate circuit outputs during BIST exhaustive stimulation.

Saluja, Sharma and Kime [Salu87a] use an on-line scheme similar to [Shar88] except instead of storing a test vector table, a sequence of tests is generated as per usual with BIST, and responses are collected using a MISR. Whenever a normal input matches the next vector in the test sequence, the circuit's outputs are accumulated in the MISR. That a complete test sequence ever occur is unclear. Off-line testing is still performed periodically.

Nicolaidis [Nico89] observes that it is difficult to design self-testing checkers for on-line schemes because the set of outputs produced by the CUC during normal operation is insufficient. He therefore suggests the use of "self-exercising checkers," which is essentially a BIST scheme where the circuit periodically goes off-line and the checkers are exercised fully. Additionally, he proposes that a "unified" BILBO, or *UBILBO*, structure be used analogous to [Köen79]. UBILBO's sandwiched between functional blocks perform the following tasks during different phases of off-line test: test sequence generation for a functional block, response compression of a functional block (on-line checking of that block is still done in parallel), generation of the exercising sequence for the checker of a functional block, and serial scan in/out.

2.4 SUMMARY

This chapter presented a survey of published research on the topic of PLA testing. Stuck-at, bridging, crosspoint and line break fault models were found to be in common use, and most works adopted the unidirectional error model for PLA's. To detect unidirectional errors, an error detecting code must be unordered; several such codes were identified.

Testing strategies may be categorized as off-line, BIST, or on-line. Much has been published in the area of off-line testing. Test generation concepts provide useful analytic tools. Random methods do not work well, while exhaustive methods are only applicable to small PLA's. PLA's designed to be easily tested generally use some technique to

uniquely select individual bit and/or product lines, so that a universal test sequence can then be applied.

BIST methods either extend this so that on-chip circuitry generates the universal test sequence, or, in a more conventional BIST manner, use a MISR to collect a signature from an exhaustive or pseudorandom test sequence. Various ingenious schemes, some function independent, verify the response to a universal test sequence. Chip area overhead for scan-path based BIST designs, in general, tends to be around 30 percent [Serr88]. Reported instances of overhead figures for the PLA BIST schemes range from 15 to 99 percent, but an average of around 30 percent is fairly representative of PLA BIST, also.

The few existing on-line techniques are split between encoding outputs in an error detecting code, mostly unidirectional ones, and using a nonconcurrent PLA with separate simpler code checkers on the bit, product and output lines (*i.e.* capitalizing on the natural partitioning of the PLA's regular structure). Existing strategies may exhibit some of the following unfavourable characteristics: high area overhead, reduced operating speed, poor error coverage, or indefinite design procedures. Figure 2.4 depicts a taxonomy of off-line, BIST and on-line PLA testing strategies.

The achievements in easy testability or BIST generally are of little help to on-line testing, but conversely, on-line checking facilities can be used beneficially during explicit testing.

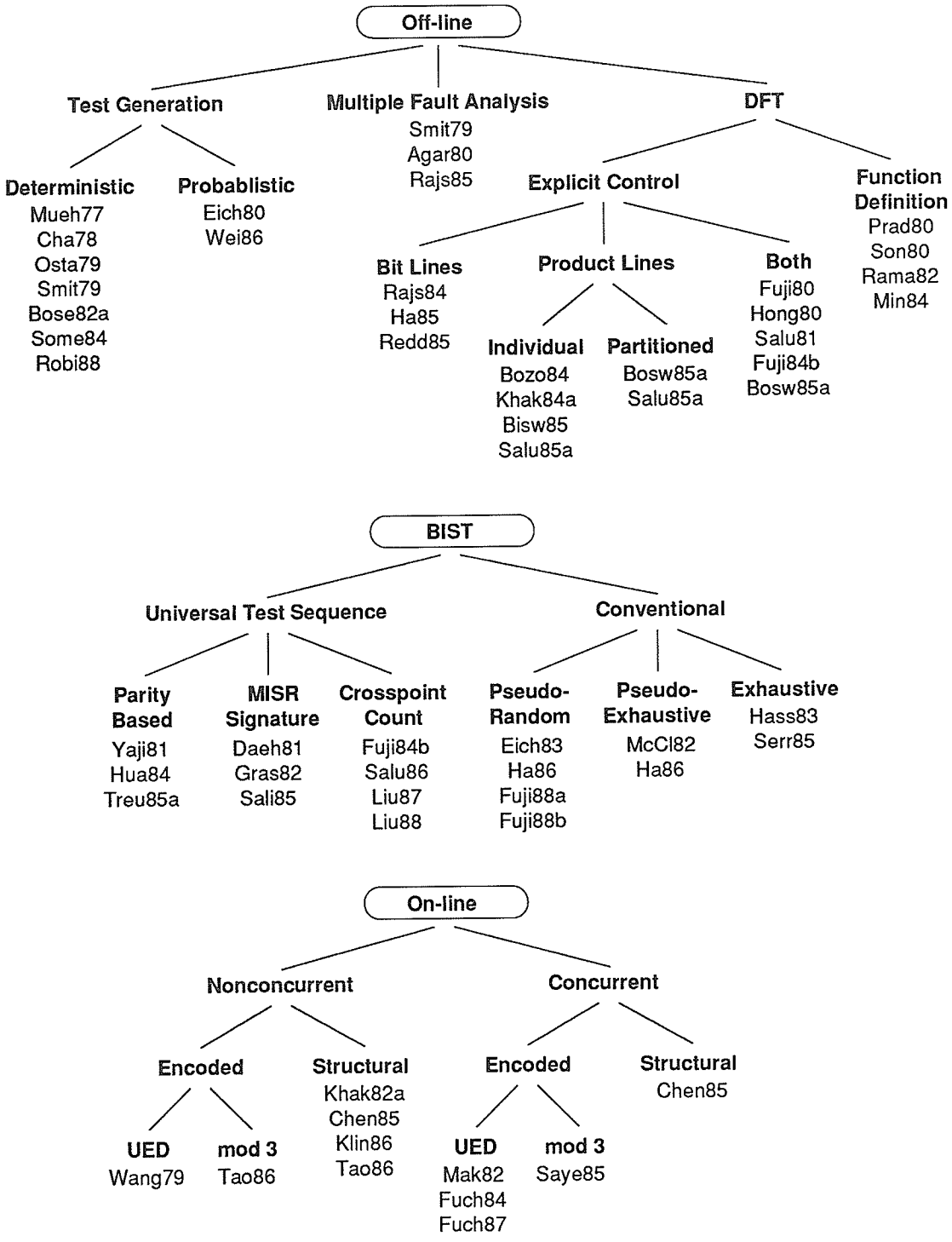


Figure 2.4 A taxonomy of PLA testing strategies.

Chapter 3

ANALYSIS OF PROBLEM

The problem addressed in this dissertation is the design of an effective on-line checking strategy for programmable logic arrays. This chapter deals with two main topics. The first defines the PLA structure and its associated fault model, and determines important properties of that model. The second considers the definition of quantitative measures used to evaluate the effectiveness of partially self-checking strategies. A detailed analysis of a mod 3 residue checking strategy is reported. A final section describes a method to determine the minimal size codeword required to ensure detectability of all modeled errors.

3.1 SCOPE OF STUDY

The objects studied in this dissertation are defined in this section. Specifically the PLA structure and fault model are detailed.

3.1.1 PLA Structure

PLA's considered are restricted to those of the form depicted in Figure 1.2a. Specifically NMOS NOR–NOR PLA's with modified one-bit decoders (*cf.* Figure 2.2b). The implementation details in three areas warrant further attention.

Options exist for the placement of depletion-mode pull-up transistors on the product and output lines. Such transistors may be situated on either end of a line, and in the case of the product term line, pull-ups may also be placed between the AND-plane and the OR-plane. The only fault detection consequence of pull-up placement arises in the behaviour of line break faults. The placement as shown in Figure 1.2a is assumed. It is

also assumed that if a line break occurs, then the portion of the line disconnected from the driving source of the pull-up will behave as a permanent logic 0. If pull-ups were on the opposite end of the product line then, a line break in the OR-plane would behave as a permanent logic 0 on the portion disconnected from the pull-up, and a permanent logic 1 on the portion still connected to the pull-up. Thus a non-unidirectional error may arise. Pull-up placement between the planes is another alternative which avoids such non-unidirectional errors. Output line pull-ups may be placed on either end without affecting the error model.

Drivers on the product term lines, situated between the two planes, are not included in the assumed structure. Such drivers require modelling of product line faults separately in each plane. But no new or different fault behaviour arises.

The decoders shown in Figure 1.2a are more usually implemented as shown in Figure 3.1, where true and complemented bit lines do not alternate. The consequence of this is that different pairs of bit lines are adjacent, and this affects the definition of the bridging faults which are included in the fault model. However, as it will be shown in section 3.1.2.2, bit line bridging faults require no special attention, regardless of which lines are adjacent. In the same way, permuting the input variables has no significant effect with respect to bridging faults.

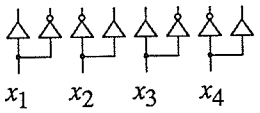


Figure 3.1 Decoder implementation detail.

3.1.2 Fault/Error Model

A transistor level single fault model including the faults listed in Tables 3.1 through 3.4 is adopted. AND bridging is assumed for all bridging faults except where noted (*cf.* Lemma 3.9). The model may be characterized as consisting of a single:

- stuck-at,
- crosspoint,
- AND bridge between two adjacent parallel lines,
- AND bridge between two orthogonal lines, and
- line break fault.

Table 3.1 lists Class I PLA faults which are those which need to be accounted for by any on-line checking scheme. This table also shows the type of error produced by each type of fault; this will be justified in section 3.1.2.1. For each type, either single bit or multiple bit asymmetric errors arise, which means that errors produced by Class I faults are in general unidirectional. Internal decoder stuck-at and bridging faults are not explicitly listed because the buffer and inverter are considered to be part of the bit lines they drive. The effect of any stuck-at or bridging fault situated before these elements propagates through them and appears on the bit lines.

	Fault	Error
I.1	bit line stuck-at-0	Asymmetric (0→1)
I.2	bit line stuck-at-1	Asymmetric (1→0)
I.3	product line stuck-at-0	Asymmetric (1→0)
I.4	product line stuck-at-1	Asymmetric (0→1)
I.5	output stuck-at-0	Asymmetric (1→0) Single Bit
I.6	output stuck-at-1	Asymmetric (0→1) Single Bit
I.7	bit line broken	Asymmetric (0→1)
I.8	product line broken	Asymmetric (1→0)

Table 3.1 Class I faults.

Table 3.2 lists Class II faults. If an on-line checking scheme detects all Class I errors then all Class II errors will also be detectable. This is discussed further in section 3.1.2.2.

	Fault	⊆
II.1	missing AND-plane crosspoint	I.4
II.2	extra AND-plane crosspoint[1]	I.3
II.3	missing OR-plane crosspoint	I.5
II.4	extra OR-plane crosspoint	I.6
II.5	bridge between adjacent bit lines[2]	I.1, I.1
II.6	bridge between adjacent bit lines[3]	I.1, I.1
II.7	bridge between adjacent product lines	I.3, I.3
II.8	bridge between adjacent output lines	I.5, I.6
II.9	bridge between bit and product lines	I.1, I.3, I.4
II.10	bridge between product and output lines	I.3, I.5, I.6

⊆ covering faults, defined in section 3.1.2.2
1 no crosspoint on complementary bit line
2 same variable
3 different variables

Table 3.2 Class II faults.

Class III faults are those which are equivalent to one already considered; these are shown in Table 3.3 along with the fault to which each is equivalent. Only one representative fault from a fault equivalence class need be considered since all faults belonging to the equivalence class produce indistinguishable behaviour.

	Fault	≡
III.1	extra AND-plane crosspoint[1]	I.3
III.2	output line (before inverter) stuck-at-0	I.6
III.3	output line (before inverter) stuck-at-1	I.5
III.4	output line broken	III.2
III.5	any array transistor stuck-on	I.3 or I.6
III.6	any array transistor stuck-open	II.1 or II.3
III.7	gate-to-source short in any array transistor	I.1 or I.3
III.8	gate-to-drain short in any array transistor	II.9 or II.10

1 crosspoint on complementary bit line

Table 3.3 Class III faults.

This dissertation assumes a fault model consisting of all single faults taken from Classes I, II and III. Expressly omitted are faults from Class IV, Table 3.4. Types IV.1

and IV.2 represent faults which map one input vector, X, into another, $\widetilde{X}$, and this is generally indistinguishable from the intended application of $\widetilde{X}$, unless the PLA is code disjoint and $\widetilde{X}$ is a noncode input. It is assumed that PLA inputs are checked elsewhere. Faults of type IV.3 are ignored as they introduce sequential behaviour, thus making fault detection a fundamentally different problem.

	Fault
IV.1	primary input line stuck-at-α
IV.2	bridge between adjacent primary input lines
IV.3	feedback bridge between input and output lines

Table 3.4 Class IV faults.

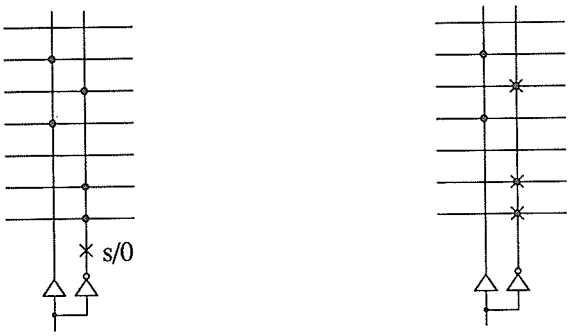
3.1.2.1 Fault Behaviour

In this section the behaviour of Class I faults is described. First, consider the effect of crosspoint faults. If the crosspoint on bit line $x_{j,\alpha}$ and product line p_i is missing (fault type II.1) then the cube, C^i, corresponding to p_i has its j^{th} coordinate changed from α to an X, $C^i \circ c_j^i \rightarrow X$. This implies that the cube $C^i \circ c_j^i \rightarrow \bar{\alpha}$ selects p_i when it should not, and some of p_i's outputs may be asserted in error (if those outputs normally would not have been produced). Therefore a missing AND-plane crosspoint may cause 0→1 type errors.

An extra crosspoint (fault type II.2) between $x_{j,\alpha}$ and p_i causes the j^{th} coordinate of cube C^i to change from an X to α, $C^i \circ c_j^i \rightarrow \alpha$. This implies that the cube $C^i \circ c_j^i \rightarrow \bar{\alpha}$ no longer selects p_i when it should, and some of p_i's outputs may erroneously not be asserted. Therefore an extra AND-plane crosspoint may cause 1→0 type errors.

A missing or extra OR-plane crosspoint (fault types II.3 and II.4) may cause the affected output line to be unasserted or asserted erroneously when the affected product line is selected. Thus a missing (extra) OR-plane crosspoint may cause a single 1→0 (0→1) type error.

A bit line which is stuck-at-0 (fault type I.1) can no longer have an affect on the output of any NOR gate product lines it has crosspoints on, so it is as if all those crosspoints are missing (see Figure 3.2). The multiple fault consisting of multiple missing AND-plane crosspoints, each capable of producing only 0→1 errors, combine to produce only 0→1 errors.



a) I.1 bit line stuck-at-0 b) equivalent multiple missing crosspoints

Figure 3.2 Bit line stuck-at-0 fault in terms of missing crosspoints.

A product line which is stuck-at-0 (fault type I.3) can no longer have an affect on any output lines it has crosspoints on, so it is as if all those crosspoints are missing (see Figure 3.3). The multiple fault consisting of multiple missing OR-plane crosspoints, each capable of producing only single 1→0 errors, combine to produce only 1→0 errors.

a) I.3 product line stuck-at-0

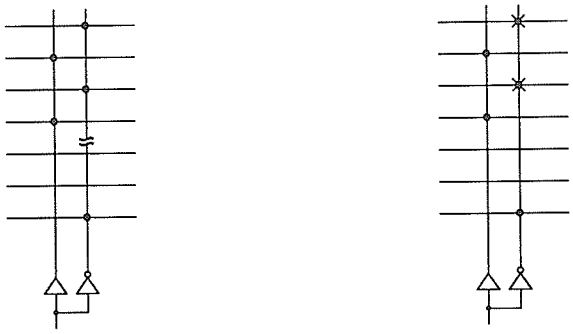
b) equivalent multiple missing crosspoints

Figure 3.3 Product line stuck-at-0 fault in terms of missing crosspoints.

A bit line stuck-at-1 (fault type I.2) causes all product lines it has crosspoints on to be permanently held at 0, so it behaves as a multiple product line stuck-at-0 fault, producing only 1→0 errors.

A product line stuck-at-1 (fault type I.4) causes all NOR-gate output lines it has crosspoints on to be permanently held at 0, so after inversion the outputs appear stuck-at-1; only 0→1 errors arise.

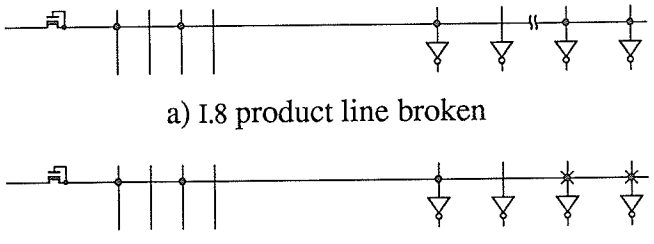
A line break in a bit line (fault type I.7) behaves similarly to a bit line stuck-at-0 since the portion of the bit line disconnected from the decoders is assumed to float to logic 0 and any crosspoints on that portion appear missing (see Figure 3.4). Only 0→1 errors arise.



a) I.7 bit line broken b) equivalent multiple missing crosspoints

Figure 3.4 Break in bit line in terms of missing crosspoints.

A product line break (fault type I.8) behaves like a product line stuck-at-0 as the portion of the product line disconnected from the pull-up is assumed to float to logic 0 and crosspoints on that portion appear missing (see Figure 3.5). Only 1→0 errors arise.



a) I.8 product line broken

b) equivalent multiple missing crosspoints

Figure 3.5 Break in product line in terms of missing crosspoints.

3.1.2.2 Covered and Equivalent Faults

In this section the concept of covered faults is introduced. For on-line checking, covered faults play a role similar to that of fault dominance in off-line testing. It is said that fault f_α dominates f_β if every test vector that detects f_β also detects f_α, thus f_α need not be explicitly considered during test generation [Sche72].

Definition 3.1 An *error event* is an ordered pair $(Z, \widetilde{Z})$, $Z \neq \widetilde{Z}$, where Z represents the fault free output of a combinational circuit for some input X, $Z = \widetilde{F}(X, \varnothing)$, and $\widetilde{Z}$ represents the erroneous output arising due to the presence of some fault f, $\widetilde{Z} = \widetilde{F}(X, f)$.

Definition 3.2 An *error event set* (*EES*) for a fault f is the set of all error events that arise due to f. Hence, $EES(f) = \{(Z, \widetilde{Z}) : \forall X \mid Z = \widetilde{F}(X, \varnothing), \widetilde{Z} = \widetilde{F}(X, f), Z \neq \widetilde{Z}\}$.

Definition 3.3 Fault f_α *covers* fault f_β, denoted $f_\alpha \supseteq f_\beta$, if $EES(f_\alpha) \supseteq EES(f_\beta)$.

Clearly, if an on-line checking strategy detects all errors due to a particular fault, f_α, then any other fault, f_β, covered by f_α need not be explicitly considered†. Note that f_α also dominates f_β but instead of ignoring the dominating fault (f_α), as is the case with test generation, for on-line checking analysis it is the dominated fault (f_β) which need not be considered further. If a testing strategy is unable to detect all errors due to f_α, then further analysis is required to determine error coverage for fault f_β. In several instances, a single Class I fault covers a Class II fault. Sometimes no single Class I fault covers a given Class II fault, but some combination of Class I faults does.

Definition 3.4 Fault set $\{f_{\alpha_1}, \ldots f_{\alpha_k}\}$ *covers* fault f_β if $\bigcup_{i=1}^{k} EES(f_{\alpha_i}) \supseteq EES(f_\beta)$.

† This is true for strategies which observe only CUC outputs, *i.e.* not internal signals. Note that coverage encompasses the notion of fault equivalence, since equivalent faults cover each other.

Next it is shown how each Class II fault is covered by Class I faults.

Lemma 3.1 A fault of type II.1 is covered by a single fault of type I.4.

Proof: A missing AND-plane crosspoint on a product line may result in that line being erroneously selected for certain input vectors. For these input vectors, a stuck-at-1 fault on the product line produces the same error events as does the crosspoint fault. □

Lemma 3.2 A fault of type II.2 is covered by a single fault of type I.3.

Proof: An extra AND-plane crosspoint on a product line may result in that line erroneously not being selected for certain input vectors. For these input vectors, a stuck-at-0 fault on the product line produces the same error events as does the crosspoint fault. □

Lemma 3.3 A fault of type II.3 is covered by a single fault of type I.5.

Proof: A missing OR-plane crosspoint on an output line may result in that output erroneously not being asserted for input vectors which select the affected product line. For these input vectors, a stuck-at-0 fault on the output line produces the same error events as does the crosspoint fault. □

Lemma 3.4 A fault of type II.4 is covered by a single fault of type I.6.

Proof: An extra OR-plane crosspoint on an output line may result in that output being erroneously asserted for input vectors which select the affected product line. For these input vectors, a stuck-at-1 fault on the output line produces the same error events as does the crosspoint fault. □

Lemma 3.5 A fault of type II.5 is covered by two faults of type I.1.

Proof: An AND bridge between bit lines from the same decoder stage, $x_{i,0}$ and $x_{i,1}$, results in both lines being held at 0 since the decoder outputs a 1-out-of-2 code. For input

vectors which normally assert $x_{i,0}(x_{i,1})$, a stuck-at-0 fault on $x_{i,0}(x_{i,1})$ produces the same error events as does the bridging fault. □

Lemma 3.6 A fault of type II.6 is covered by two faults of type I.1.

Proof: An AND bridge between bit lines from different decoders, $x_{i,0}$ and $x_{j,1}$, results in both lines being held at 0 only when $x_{i,0}=\overline{x}_{j,1}$ normally. For input vectors which normally assert only $x_{i,0}(x_{j,1})$, a stuck-at-0 fault on $x_{i,0}(x_{j,1})$ produces the same error events as does the bridging fault. □

If the decoder design of Figure 3.1 is used, then either $x_{i,0}$ and $x_{j,0}$, or $x_{i,1}$ and $x_{j,1}$ will be adjacent bit lines, but the reasoning of Lemma 3.6 still applies.

Lemma 3.7 A fault of type II.7 is covered by two faults of type I.3.

Proof: An AND bridge between product lines, p_i and p_j, results in both lines being held at 0 only when $p_i=\overline{p}_j$ normally. For input vectors which normally assert only $p_i(p_j)$, a stuck-at-0 fault on $p_i(p_j)$ produces the same error events as does the bridging fault. □

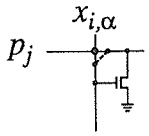
Lemma 3.8 A fault of type II.8 is covered by two faults of type I.5 or I.6.

Proof: An AND bridge between output lines, z_i and z_j, results in both lines being held at 0 only when $z_i=\overline{z}_j$ normally. For input vectors which normally assert only $z_i(z_j)$, a stuck-at-α fault on $z_i(z_j)$, where α=1(0) if the bridge is situated before(after) the final inverter, produces the same error events as does the bridging fault. □

Lemma 3.9 A fault of type II.9 is covered by one fault each of types I.1, I.3 and I.4.

Proof: Two cases are considered. In the first case, no crosspoint exists at the site where the affected bit and product lines cross. An AND bridge between bit and product lines, $x_{i,\alpha}$ and p_j, results in both lines being held at 0 only when $x_{i,\alpha}=\overline{p}_j$ normally. For input

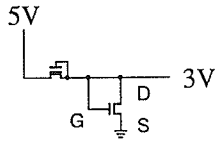
vectors which normally assert only $x_{i,\alpha}(p_j)$, a stuck-at-0 fault on $x_{i,\alpha}(p_j)$ produces the same error events as does the bridging fault. In the second case, a crosspoint transistor exists between $x_{i,\alpha}$ and p_j, and the faulty circuit is shown in Figure 3.6a. When at least one of the two lines is a 0, the circuit behaves as in the first case. This is summarized in Figure 3.6b. Note for the situation $x_{i,\alpha}$=1 and p_j=0, it is assumed that some other bit line is already pulling p_j low. The final situation, where $x_{i,\alpha}$=1 and p_j=1, exists if $x_{i,\alpha}$ is the sole bit line normally supposed to pull the product line low. The fault in this case is essentially a short between the gate and drain of the enhancement mode crosspoint transistor as shown in Figure 3.6c, which is Figure 3.6a redrawn to emphasize the faulty structure. In this situation, the shorted lines both take on a value of approximately 3V, according to [Bane82], which although not a true logic 1 is at least a weak 1. The bit line is thus considered unchanged. For input vectors which establish this situation, a stuck-at-1 fault on p_j produces the same error events as does the bridging fault. □



a) fault II.9

$x_{i,\alpha}$	p_j	short	$\subseteq$
0V	0V	0V	—
0V	5V	0V	I.3
5V	0V	0V	I.1
5V	5V	3V	I.4

b) behaviour



c) effective stuck-at-1

Figure 3.6 Bridge between bit and product lines.

Lemma 3.10 A fault of type II.10 is covered by one fault each of types I.3, I.5 and I.6.

Proof: Similar to Lemma 3.9. □

Theorem 3.1 All Class II faults are covered by the set of single stuck-at faults on bit, product and output lines.

Proof: The proof follows from Lemmata 3.1 through 3.10. Fault type I.2 is not actually required. □

Class III faults are trivially equivalent to the faults listed in the second column of Table 3.3. For fault type III.1, if the true and complemented bit lines associated with the same input variable are both connected to a particular product line, then that product line will always be pulled low regardless of the input value. Fault types III.2 and III.3 simply represent fault propagation through final output inverters. A line break on an output line (III.4) disconnects the pull-up causing the input of the final inverter to float to logic 0, hence the output appears stuck-at-1. A crosspoint transistor stuck-open (III.5) means that transistor never conducts so it appears to be missing. A stuck-on transistor (III.6) always conducts thus holding its NOR line low. Transistor shorts (III.7 and III.8) simply represent specific manifestations of stuck-at-0 and orthogonal line bridging faults.

3.1.2.3 Error Enumeration

It is relatively straightforward, albeit time consuming, to enumerate all error events that arise due to a given PLA fault. Enumeration is equivalent to determining a complete set of test vectors for each fault, but need only be performed on Class I faults. Enumeration is required for two reasons. The first is to assess existing checking schemes which do not provide complete fault coverage, and the second is to obtain an explicit error model to use as the basis for the design of new on-line checking schemes.

Consider $L=(C,D)$ as the cubical array realized by a PLA. In most cases a fault causes one or more product lines, $L^I=\{L^{i1}, L^{i2},..., L^{ir}\}$, to either become selected or deselected in error for some input vectors. Let $\hat{L}=(\hat{C}, \hat{D})$ denote the array of cubes which potentially exhibit erroneous behaviour due to the selection or deselection of some product lines in L^I. For any input vector in $\hat{L}$, if all of the outputs driven by the affected product lines in L^I are also produced by other product lines which are not affected by the fault and which are normally selected for the same input vector, then the effect of the fault is masked and the correct output is produced. Thus $\hat{L}$ #$(L–L^I)$ represents all input

vectors which produce unmasked errors. Algorithm 3.1 determines this set of input cubes, and the corresponding output bits which are in error. Index set $I=\{i_1, i_2,\ldots, i_r\}$ is given as an argument along with L and $\widehat{L}$. $\widehat{L}$ is expected to be 1-concurrent. Additionally, cube S which is the smallest cube which contains $\widehat{L}$, and vector P which is the bitwise OR of D^I, are also given to help optimize computation; any candidate masking cube from L–L^I which does not intersect with S or which has no bits in common with P cannot possibly mask an error.

Algorithm Enum($L, I, \widehat{L}, S, P$):

- **for all** $i\notin I$: $C^i\cap S\neq\varnothing$ **and** $D^i\wedge P\neq 0$ (1)
  - $T\leftarrow\{\}$ (2)
  - **for all** $\widehat{L}^k\in\widehat{L}$: $D^i\wedge\widehat{D}^k\neq 0$ (3)
    - **if** $C^i\supseteq\widehat{C}^k$ **then** (4)
      - $\widehat{D}^k\leftarrow\widehat{D}^k-D^i$ (5)
      - **if** $wt(\widehat{D}^k)\leq 1$ **then** $\widehat{L}\leftarrow\widehat{L}-\widehat{L}^k$ (6)
    - **else** (7)
      - $J\leftarrow C^i\cap\widehat{C}^k$ (8)
      - **if** $J\neq\varnothing$ **then** (9)
        - $T\leftarrow T\cup(\widehat{C}^k\#J, \widehat{D}^k)$ (10)
        - $\widehat{L}^k\leftarrow(J, \widehat{D}^k-D^i)$ (11)
        - **if** $wt(\widehat{D}^k)\leq 1$ **then** $\widehat{L}\leftarrow\widehat{L}-\widehat{L}^k$ (12)
  - $\widehat{L}\leftarrow\widehat{L}\cup T$ (13)
- **return** $\widehat{L}$ (14)

Algorithm 3.1 Enumerate error events.

For example, as discussed in section 3.1.2.1, for missing AND-plane crosspoint on bit line $x_{j,\alpha}$ and product line L^i (fault type II.1), potential error cube $ec=C^i\circ c_j^i\rightarrow\bar{\alpha}$ selects L^i when it should not. Algorithm 3.1 invoked as **Enum**(L, $\{i\}$, $\{(ec,D^i)\}$, ec, D^i) obtains the desired list of error events. As shown earlier, all bits in error are of 0→1 type. An extra AND-plane crosspoint (fault type II.2) is enumerated identically except 1→0

type errors arise. Let e_k denote a binary vector which is all 0's except for the k^{th} bit. For an OR-plane crosspoint fault (either fault type II.3 or II.4) on product line L^i and output line z_k, potential error cube $ec=C^i$ selects L^i but output z_k may be in error. **Enum**$(L,\{i\},\{(C^i,e_k)\},C^i,e_k)$ might be used to determine the error events†. But since only single bit errors arise, and since all possible single bit errors are expected due to fault types I.5 and I.6, there is no point in enumerating OR-plane crosspoint faults in this way. However, this analysis will prove useful in enumerating other faults.

Any Class I fault which was shown to be equivalent to some multiple missing crosspoint fault in section 3.1.2.1 (*i.e.* fault types: I.1, I.2, I.3, I.7 and I.8) can be enumerated on the basis of those missing crosspoints. I simply contains the indices of all product lines affected with missing crosspoints, and $\hat{L}$ is the disjoint array equivalent to the union of all $\hat{L}$'s corresponding to the individual crosspoint faults. For a product line L^i stuck-at-1 (fault type I.4), $\hat{L}=(\overline{C}^i,D^i)$, and $I=\{i\}$. Fault types I.5 and I.6 (output stuck-at faults) can be enumerated directly from a truth table or any other list containing the set of output patterns produced by the PLA. Detailed enumeration analysis of these and the remaining Class II faults can be found in [Marc88].

The stuck-at-0 fault on $x_{3,1}$ in the PLA of Figure 1.2 will be enumerated as an example of the operation of Algorithm 3.1. The cubical specification for this PLA is restated below:

$$L^1 = \text{X01X } 110 \qquad L^4 = \text{X11X } 011$$
$$L^2 = \text{11X1 } 101 \qquad L^5 = \text{100X } 101$$
$$L^3 = \text{01X1 } 110$$

Product terms L^1 and L^4 have crosspoints on bit line $x_{3,1}$. The error cube associated with a missing crosspoint on L^1 is X00X 110, and that for L^4 is X10X 011. The algorithm is therefore invoked: **Enum**$(L, \{1,4\}, \begin{Bmatrix}\text{X00X } 110\\ \text{X10X } 011\end{Bmatrix}, \text{XX0X}, 111)$. A partial trace of the execution of this invocation follows.

† The test condition on lines 6 and 12 of Algorithm 3.1 would have to change from "≤1" to "=0".

i	L^i	$\widehat{L}^k$	stmt	action	comment
2	11X1 101	X00X 110	4, 8	$J = \varnothing$	retain $\widehat{L}^k$ as is
		X10X 011	4, 8	$J \leftarrow 1101$	
			10	$T \leftarrow \begin{Bmatrix} \text{X100 011} \\ \text{0101 011} \end{Bmatrix}$	*i.e.* (X10X ⊕ 1101, 011)
			11	$\widehat{L}^k \leftarrow 1101\ 010$	011 – 101 = 010
			12	$\widehat{L} \leftarrow \widehat{L} - \widehat{L}^k$	$wt(010) = 1$
			13	$\widehat{L} \leftarrow \begin{Bmatrix} \text{X00X 110} \\ \text{X100 011} \\ \text{0101 011} \end{Bmatrix}$	
3	01X1 110	X00X 110	4, 8	$J = \varnothing$	retain $\widehat{L}^k$ as is
		X100 011	4, 8	$J = \varnothing$	retain $\widehat{L}^k$ as is
		0101 011	4, 5	$\widehat{D}^k \leftarrow 001$	011 – 110 = 001
			6	$\widehat{L} \leftarrow \widehat{L} - \widehat{L}^k$	$wt(001) = 1$
			13	$\widehat{L} \leftarrow \begin{Bmatrix} \text{X00X 110} \\ \text{X100 011} \end{Bmatrix}$	
5	100X 101	X00X 110	4, 8	$J \leftarrow 100\text{X}$	
			10	$T \leftarrow 000\text{X}\ 110$	*i.e.* (X00X ⊕ 100X, 110)
			11	$\widehat{L}^k \leftarrow 100\text{X}\ 010$	110 – 101 = 010
			12	$\widehat{L} \leftarrow \widehat{L} - \widehat{L}^k$	$wt(010) = 1$
		X100 011	4, 8	$J = \varnothing$	retain $\widehat{L}^k$ as is
			13	$\widehat{L} \leftarrow \begin{Bmatrix} \text{X100 011} \\ \text{000X 110} \end{Bmatrix}$	

The final value of $\widehat{L}$ indicates that inputs from X100 will result in an error pattern of 011, and inputs from 000X will produce an error pattern of 110 (in all these cases the normal output value is 000). Some single bit errors may also arise, but they are not explicitly enumerated because, when enumerating all faults from the model, all possible single bit errors are expected. Statements 6 and 12 discard potential error cubes when their output parts have weight ≤1.

3.1.2.4 Nonconcurrent PLA

In this section, the behaviour of faulty PLA's, described in section 3.1.2.1, will be re-evaluated in terms of nonconcurrent PLA's, and a simplified error model will be shown to result. Some lemmata are introduced first.

Lemma 3.11 Given $C=\{c^1, c^2,\ldots c^r\}$ an array of disjoint cubes where $c^i_j=\alpha$ for all $c^i\in C$, $\alpha\in\{0,1\}$, the cubical array specified by $C\circ c^i_j\rightarrow$X, for all $c^i\in C'$† for any subset $C'\subseteq C$, is also disjoint.

Proof: Consider any two cubes c^i, $c^k\in C$, since C is disjoint and since $c^i_j=c^k_j=\alpha$ there must exist some variable x_q such that $c^i_q=\bar{c}^k_q$. If either or both c^i_j and c^k_j are changed to X, cubes c^i and c^k remain disjoint due to coordinate x_q. □

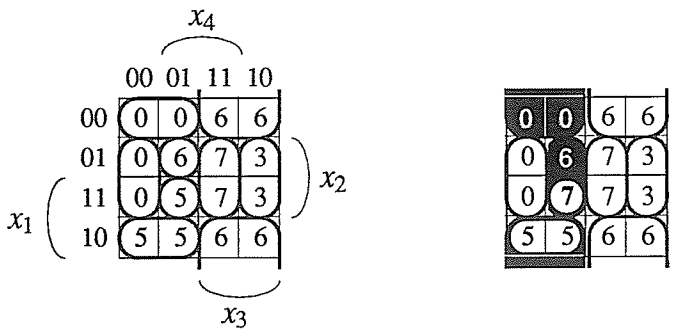
Lemma 3.12 A bit line stuck-at-0 fault (type I.1) will make a nonconcurrent PLA at most 2-concurrent.

Proof: Suppose C is a nonconcurrent cube array corresponding to the fault-free PLA, and $x_{j,\alpha}$ is the affected bit line. C can be partitioned into three subarrays, C_0, C_1 and C_X, each containing only cubes from C with $c^i_j = 0$, 1 and X, respectively. Without loss of generality, consider α=0. $C_1\cup C_X$ remains disjoint since none of the corresponding PLA lines are connected to $x_{j,0}$. Let C'_0 denote $C_0\circ c^i_j\rightarrow$X. From Lemma 3.11 C'_0 is disjoint. Further, $C'_0\cup C_X$ is also clearly disjoint. Therefore $C_X\cap(C'_0\cup C_1)=\varnothing$. Any input vector is covered by either C_X or at most one cube from each of the partitions C'_0 and C_1, thus at most two product lines are selected. □

Consider the PLA of Figure 1.2 in context of Lemma 3.12. A complete cubical specification of a nonconcurrent realization is given later (*cf.* Figure 4.2b), but for now consider its Karnaugh map shown in Figure 3.7a (output values are shown in decimal). If bit line $x_{1,0}$ is stuck-at-0, then two cubes of the specification essentially change:

† The resulting array consists of the complete original array with only cubes from subset C' modified as indicated.

0101→X101 and 000X→X00X. The former now intersects 1101, and the latter intersects 100X, as shown in Figure 3.7b (the enlarged cubes are blackened; the only erroneous output occurs for minterm 1101).



a) nonconcurrent b) 2-concurrent

Figure 3.7 Example of the effect of bit line $x_{1,0}$ stuck-at-0.

Lemma 3.13 A line break in a bit line (fault type I.7) will make a nonconcurrent PLA at most 2-concurrent.

Proof: Similar to Lemma 3.12. □

Definition 3.5 An error in a combinational circuit which consists of the bitwise OR of the intended correct codeword and one other codeword is said to be an *error of type* ξ†.

Theorem 3.2 Any single Class I fault, except for fault type I.8, in a nonconcurrent PLA only produces one of the following kinds of error:

- no product line is selected, the output is all 0's,
- an error of type ξ,
- a single-bit error.

Proof: From Lemmata 3.12 and 3.13 faults of type I.1 and I.7 are shown to make the PLA at most 2-concurrent. Since each product term specifies one output codeword, when two

† [Bane82] uses the notation L_i/L_i+L_j.

terms are selected simultaneously an error of type ξ arises. Fault types I.2 and I.3 cause one or more product lines to appear stuck-at-0, and since only one product line is ever selected by any input vector, for an input which selects any of the stuck-at-0 product lines, an all 0's output occurs. For a fault of type I.4, a single product line is permanently selected. An input vector selects at most one additional product line, hence an error of type ξ arises. Fault types I.5 and I.6 cause only single-bit errors. □

Theorem 3.2 is similar to that shown for NOR-array implementations of decoders in [Bane82], and is a stronger result than that of [Chen85] where an error consisting of any number of codewords OR'ed together is presented.

Corollary 3.1 If a checking scheme detects all errors due to type I.8 faults, then all error events $(Z, \widetilde{Z})$ with $\widetilde{Z} = 0$ (the first kind of error listed in Theorem 3.2) will also be detected.

Proof: In order to detect a type I.8 fault each codeword must have a 1 in at least one of the rightmost two output bit positions, and each of these positions must be a 0 for some codeword. Having a 1 in just the last bit position is insufficient since a stuck-at-1 fault on that line would mask all type I.8 faults. Any error event with $\widetilde{Z} = 0$ would be detected because both final output bits are 0. □

Corollary 3.2 Any code capable of detecting all errors of type ξ in a nonconcurrent PLA must be unordered.

Proof: Suppose D^i and D^k are two codewords from a code capable of detecting all errors of type ξ, and $D^i \supset D^k$. If an error of type ξ occurs involving product terms p_i and p_k, then the output will be D^i, which is a codeword, even if D^k was intended. Thus an undetectable error would occur, and this contradicts the assumption that the code is capable of detecting all errors of type ξ. □

3.2 EVALUATION OF PARTIALLY SELF-CHECKING SCHEMES

Quantitative measures useful in the evaluation of the effectiveness of partially self-checking strategies are considered in this section. A partially self-checking circuit may produce erroneous output patterns which go undetected. Such a circuit may be reasonably effective in exposing faults while being considerably less expensive to realize than a TSC circuit.

3.2.1 Error Checking Effectiveness

In order to evaluate and compare various on-line schemes which are only partially self-checking, some quantitative measure of effectiveness is required. A brief look at effectiveness measures used with off-line testing strategies allows us to conclude that they are inappropriate for on-line schemes. The measures usually used for test vector based off-line testing are: a) *coverage* which is defined to be the percentage of detectable faults from a model which are exposed by a test set, and b) test set size required to obtain a certain level of coverage. The concept of an undetectable error occurring during testing is non-existent since an external tester compares each CUT output to its expected value.

Most BIST schemes use some sort of response compression (such as MISR), and there exists the possibility of unrecognized errors due to *aliasing*, which is when a response sequence containing errors still produces the expected signature. Probabilistic methods are used to derive a level of confidence.

An on-line scheme, however, must check every output pattern produced by the circuit without benefit of knowing the correct value. Other measures of effectiveness are required. One possibility is to estimate the percentage of faults which are detectable (called *fault-detection ability*, or *FDA*, in [Fuji84a]). For any given input vector, the circuit output is either:

- the correct codeword, (*i.e.* C ⇔ Correct),
- an incorrect codeword, (*i.e.* U ⇔ Undetected error), or
- a noncodeword, (*i.e.* T ⇔ a Test which exposes the fault).

Let $N_C(f)$, $N_U(f)$, and $N_T(f)$ denote for a fault f the proportion of $M=2^m$ input vectors producing outputs in each of the three categories, respectively. Simulation or analytical techniques such as given in section 3.1.2.3 are required to obtain these values. *FDA* is just the percentage of faults for which $N_T(f)\neq 0$; it can be seen to convey only a minimal amount of information about the checking scheme. The following 4-tuple measure is proposed, $\Phi^1=(\phi_1^1, \phi_2^1, \phi_3^1, \phi_4^1)$, where $\phi_i^1 = \frac{|\pi_i|}{R}$, $R=|\mathcal{F}|$, and the fault model $\mathcal{F}$ is partitioned:

$$\pi_1 = \{f\in \mathcal{F}:\ N_T(f)=0, N_U(f)=0\}$$

$$\pi_2 = \{f\in \mathcal{F}:\ N_T(f)\neq 0, N_U(f)=0\}$$

$$\pi_3 = \{f\in \mathcal{F}:\ N_T(f)=0, N_U(f)\neq 0\}$$

$$\pi_4 = \{f\in \mathcal{F}:\ N_T(f)\neq 0, N_U(f)\neq 0\}.$$

The question arises of whether all equivalent faults or just one representative of each equivalence class should be counted. Collecting data for all faults will require more processing than that for just the fault equivalence classes, but determining these classes may require significant effort. If the faults in the fault model correlate well with physical defects, then counting all faults should provide a more meaningful measure because distinct defects are taken into account. The fact that faults are equivalent is less significant than the fact that they are caused by distinct defects. Otherwise, if the fault model does not correlate well with physical defects, then counting all faults may yield measures where some of the defects are disproportionately represented. Counting fault equivalence classes only, may thus be fairer because it is less dependent on the details of the fault model. Since a PLA fault model is closely related to underlying defects, to the extent that estimated probabilities have been ascribed to various faults, we have chosen to consider all modeled faults when computing effectiveness measures.

Instead of expressing a figure of merit in terms of faults, another method is to consider error events, $(Z, \widetilde{Z})$ as defined in Definition 3.1, or the more detailed *primary error event*, defined as $(X, \widetilde{Z})$ and representing possibly multiple instances of event $(Z, \widetilde{Z})$ since $F(X)$ may specify the same value for Z for more than one value of X. For a fault f, the proportion of M input vectors producing primary error events is simply $N_T(f)+N_U(f)$. *Error-detection ability* (*EDA*), as introduced in [Fuji84a], is alternatively defined here as:

$$EDA = \frac{\sum_{f \in \mathcal{F}} N_T(f)}{\sum_{f \in \mathcal{F}} N_T(f)+N_U(f)} \tag{3.1}$$

Lu and McCluskey [Lu84] proposed a similar figure of merit based on their *testing input fraction*, (*TIF*), and *secure input fraction*, (*SIF*), essentially defined as:

$$TIF(f) = N_T(f) \tag{3.2}$$

$$SIF(f) = N_T(f) + N_C(f) = 1 - N_U(f). \tag{3.3}$$

The figures of merit they proposed are weighted arithmetic and geometric means defined:

$$\overline{TIF} = \sum_{f \in \mathcal{F}} w_f \cdot TIF(f) \tag{3.4}$$

$$\overline{SIF} = \sum_{f \in \mathcal{F}} w_f \cdot SIF(f) \tag{3.5}$$

$$\widetilde{TIF} = \exp\left(\sum_{f \in \mathcal{F}} w_f \cdot \ln(TIF(f))\right) \tag{3.6}$$

$$\widetilde{SIF} = \exp\left(\sum_{f \in \mathcal{F}} w_f \cdot \ln(SIF(f))\right) \tag{3.7}$$

Generally $w_f = 1/R$, but other weighting factors may be used to reflect the relative probabilities of faults if known. $\overline{TIF}$ can also be interpreted as $Pr_T(\mathcal{F})$, the probability that any given input vector will expose a fault, if one is present, assuming that all input patterns are equally likely†. Likewise the probability of an undetected error is $Pr_U(\mathcal{F})=1-\overline{SIF}$. For $w_f=1/R$, $\widetilde{TIF}$ simplifies to ($\widetilde{SIF}$ is similar):

$$\widetilde{TIF} = \sqrt[R]{\prod_{f\in\mathcal{F}} TIF(f)} \tag{3.8}$$

Geometric means exhibit desired greater sensitivity to low values of *TIF* and *SIF*, but a single instance which is 0 forces the mean to be 0. Lu and McCluskey suggest partitioning the fault set to allow separate mean calculation for each partition and to obtain additional figure of merit $\Phi^2=(\phi_A^2, \phi_B^2, \phi_C^2)$, where $\phi_i^2 = |\pi_i|$, and the partitions are:

$$\pi_A = \{f\in\mathcal{F}:\ TIF(f)>0,\ SIF(f)>0\} = \{f\in\mathcal{F}:\ N_T(f)\neq 0\}$$

$$\pi_B = \{f\in\mathcal{F}:\ TIF(f)=0,\ SIF(f)>0\} = \{f\in\mathcal{F}:\ N_T(f)=0,\ N_U(f)\neq 1\}$$

$$\pi_C = \{f\in\mathcal{F}:\ TIF(f)=0,\ SIF(f)=0\} = \{f\in\mathcal{F}:\ N_U(f)=1\}.$$

It is our contention that Φ^1 more aptly characterizes the effectiveness of a checking scheme than does Φ^2. The latter's partitioning is intended solely to allow computations of $\widetilde{TIF}$ and $\widetilde{SIF}$ which do not degenerate to zero. ϕ_A^2 being essentially *FDA* is of limited value, and important information is lost by ϕ_B^2 which essentially corresponds to $\phi_1^1+\phi_3^1$. ϕ_1^1 represents faults which are undetectable (*i.e.* produce no errors), while ϕ_3^1 represents faults which produce undetectable errors only. While ϕ_3^1 is a valuable figure of merit, ϕ_C^2 is insignificant. Partition π_C represents the faults which produce an undetectable error for every input vector. It is hard to imagine this partition being anything but empty. It is not surprising that all experiments reported in [Lu84] resulted in $\phi_C^2=0$.

† This assumption is rarely justifiable, but actual distributions are nearly impossible to ascertain.

Two other measures may be defined. One is *error event detection ability* (*EEDA*), the proportion of all distinct error events which are detectable, and the other is *error pattern detection ability* (*EPDA*), the proportion of all distinct error patterns which are detectable. An *error pattern* is defined as $Z \oplus \widetilde{Z}$. Since various error events may have identical error patterns, *EPDA* is applicable only to checking schemes where the ability to detect errors is determined by pattern alone.

The collection of measures and figures of merit: Φ^1, *EDA*, $\overline{TIF}$, and *EEDA* provide several perspectives on effectiveness. These may be used to help rank two or more checking schemes on the basis of such measures calculated over a range of sample circuits, or to rank two different circuit realizations under the same checking scheme. The proportion of detectable faults which are totally missed by a checking scheme, ϕ_3^1, is an important indicator, as is ϕ_1^1 which is the proportion of faults which are undetectable and hence may mask other faults. *EDA* and $\overline{TIF}$ are closely related, but $\overline{TIF}$ provides the opportunity for incorporating fault probabilities into the measure (*i.e.* $Pr_T(\mathcal{F})$). *EEDA* is independent of whether all faults or just representatives from each equivalence class are counted; it tends to be more pessimistic than *EDA*. Experimental results (sections 3.2.2.1 and 3.2.3) suggest that ϕ_1^1 and ϕ_3^1 are of comparable magnitude, that $Pr_T(\mathcal{F})$ remains relatively constant over various PLA's, and that *EEDA* exhibits considerably more variability than does *EDA*. Two circuits with comparable *EDA*'s may have significantly different *EEDA*'s.

3.2.2 Mod 3 Residue

The collection of measures described in the previous section are applied against the mod 3 residue encoding scheme presented in [Saye85]. The mod 3 scheme was proposed as a viable on-line checking strategy, but it will be argued here that experimental evidence suggests otherwise. An analysis of the method indicates that the empirical results are not atypical.

Two methods are presented in [Saye85]. *Method 1* extends the PLA so that it realizes a complete cover and two additional check bit outputs are programmed to predict the mod 3 residue of the original outputs considered as a binary integer. The residue of an all 0 output is encoded as 11_2 to avoid accepting an erroneous all 0 output as a valid codeword; faults such as I.3 can produce such an error. This is the reason why Method 1 must realize a complete cover. Figure 3.8b shows the specification of a mod 3 augmented PLA derived for the PLA defined in Figure 3.8a (taken from Figure 1.2).

Method 2 uses a separate two-output check bit PLA to predict the residue from the same inputs as the original PLA which is itself unmodified. An all 0 output would have 00_2 as checkbits, while all other nonzero outputs with 0 residue would have checkbits encoded as 11_2 so that an erroneous all 0 output in either (but not both) PLA can be detected. Note that all errors due to faults located exclusively in the check bit PLA are detectable. An incorrect residue prediction would not match the actual residue calculated from the error-free output of the original PLA. Figure 3.8c shows the definition of a mod 3 check bit PLA for the example of Figure 3.8a.

0	0	6	6
0	6	7	3
0	5	7	3
5	5	6	6

```
X01X 110
01X1 110
X11X 011
100X 101
11X1 101
```

a) map and specification for example PLA

11	11	00	00
11	00	01	00
11	10	01	00
10	10	00	00

```
X01X  110 00
01X1  110 00
X11X  011 00
100X  101 10
1X01  101 10
X111  100 01
000X  000 11
X100  000 11
```

b) Method 1 check bit map and augmented PLA

00	00	11	11
00	11	01	11
00	10	01	11
10	10	11	11

```
X01X 10
0101 11
XX1X 01
10XX 10
1X01 10
XX10 10
```

c) Method 2 check bit PLA map and specification

Figure 3.8 Example of mod 3 encoding for PLA's.

Both methods use the decoder design of Figure 3.9 (redrawn from Figure 2.2a), thus the fault model must be extended to include stuck-at faults on line α internal to the decoder. Such faults are equivalent to type IV.1 faults, except in the case of Method 2 where the check bit PLA's inputs are considered error-free.

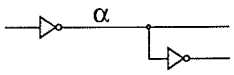


Figure 3.9 Decoder design requiring consideration of internal stuck-at fault.

3.2.2.1 Simulation

A computer program has been written to determine various measures of effectiveness for the mod 3 residue checking. The program processes PLA specifications in cubical form. It enumerates all Class I faults and for each determines the transformed cubical array and the potential error cube(s), as prescribed in section 3.1.2.3. Only minterms from these error cubes are simulated through the transformed circuit model, and output errors are tallied as detectable or not by the mod 3 residue strategy. A program option allows internal decoder faults to be included. They are processed as type IV.1 faults.

Situations corresponding to Class II faults are recognized and separate tallies are maintained thus obviating the need for simulation of these faults. A multiplicity is associated with each considered fault to account for equivalent Class III faults. The program therefore determines the measures: $N_C(f)$, $N_U(f)$, and $N_T(f)$ for all Class I, II, III and IV.1 faults, and from these are derived figures of merit: Φ^1, EDA, and $\overline{TIF}$ ($w_f = 1/R$). The program also collects data allowing determination of $EEDA$.

Appendix A describes the origin of a set of example PLA's used in this work. Simulation results for these PLA's are shown in Tables 3.5 and 3.6 based on Methods 1 and 2 of [Saye85], respectively†. In the case of Method 2, only the original PLA is

† Measures defined earlier as taking on values in range [0..1] will be expressed in these tables, and subsequent ones in this section, as percentages (excluding Pr_T and Pr_U).

considered since all errors produced by the check bit PLA are detectable, and no single fault affects both the original and check bit PLA's. Fault probability estimates based on [Maly86] and Table 2.1 are used to determine $Pr_T(\mathcal{F})$ and $Pr_U(\mathcal{F})$. The fault probabilities used are listed in Table 3.7. Defects affecting long lines (*i.e.* the bit, product and output lines), specifically shorts between adjacent lines and shorts to ground, are expected to be most likely.

PLA	m	p	n	ϕ_1^1	ϕ_2^1	ϕ_3^1	ϕ_4^1	*EDA*	$\overline{TIF}$	$Pr_T(\mathcal{F})$	$Pr_U(\mathcal{F})$	*EEDA*
5xp1	7	127	12	2.4	67.1	8.7	21.8	89.8	16.5	5.5 E-4	1.3 E-4	75.3
alu1	12	618	10									
alu2	10	256	10									
apla	10	80	14	3.1	71.9	2.4	22.7	96.1	10.5	2.9 E-4	3.0 E-5	76.4
bis1	4	8	5	3.4	91.0	0.2	5.5	98.1	22.9	2.6 E-4	5.0 E-6	94.2
bis2	4	8	5	4.2	81.4	0.9	13.5	94.8	22.4	2.3 E-4	1.8 E-5	90.7
bis5	4	11	6	3.4	58.0	8.8	29.8	84.2	20.2	2.5 E-4	7.2 E-5	85.4
bis6	4	10	5	3.0	80.1	4.4	12.6	94.1	19.3	2.2 E-4	1.9 E-5	90.2
bw	5	24	30	1.8	82.2	6.0	10.1	96.2	23.3	6.5 E-4	5.8 E-5	85.6
dc1	4	11	9	3.4	68.1	3.9	24.7	90.5	22.9	3.0 E-4	5.0 E-5	84.6
dc2	8	112	9	3.3	70.8	2.8	23.1	93.7	12.3	4.4 E-4	4.4 E-5	80.0
dist	8	160	7	1.9	62.1	6.7	29.4	89.6	13.3	4.8 E-4	7.8 E-5	83.1
dk17	10	62	13	2.7	73.5	1.4	22.3	97.6	12.0	4.0 E-4	2.0 E-5	80.2
dk27	9	38	11	4.8	78.0	0.5	16.6	92.9	14.0	4.3 E-4	6.4 E-5	83.4
eg	3	6	7	3.3	83.7	5.6	7.5	90.5	28.7	2.9 E-4	3.9 E-5	93.1
f2	4	15	6	2.1	97.5	0.0	0.4	99.7	21.8	3.3 E-4	8.1 E-9	89.8
f51m	8	191	10	4.8	72.9	0.8	21.5	90.4	14.7	6.9 E-4	9.4 E-5	81.3
fsm1	10	26	8	4.4	68.6	9.1	18.0	97.4	10.4	2.5 E-4	1.1 E-5	82.8
fsm2a	6	26	8	2.2	73.7	3.0	21.0	92.9	16.9	3.6 E-4	4.0 E-5	82.9
gary	15	141	13	2.6	58.5	6.4	32.5	94.9	9.7	4.4 E-4	5.3 E-5	73.9
misex1	8	18	9	4.2	77.1	3.2	15.5	97.1	15.8	3.3 E-4	1.4 E-5	84.8
rd53	5	32	5	1.3	71.9	6.4	20.4	85.9	16.0	3.5 E-4	6.3 E-5	91.7
router	11	26	14	2.9	80.8	2.4	13.9	97.7	13.5	2.9 E-4	1.6 E-5	90.7
sao2	10	60	6	3.4	62.7	8.8	25.1	95.9	8.0	2.2 E-4	1.3 E-5	87.4
traffic	5	9	9	3.3	77.7	5.1	13.9	94.9	22.9	3.6 E-4	2.2 E-5	88.1
wim	4	13	9	4.6	70.4	1.9	23.1	91.3	23.7	3.7 E-4	6.1 E-5	81.5
z4	7	123	6	3.0	75.9	3.0	18.1	92.5	14.6	5.3 E-4	5.4 E-5	90.7

Table 3.5 Simulation results: mod 3 residue, Method 1.

For Method 1, simulation of the PLA's "alu1" and "alu2" was not attempted; they are included in Table 3.5 just to indicate the dimensions of the encoded array.

PLA	m	p	n	ϕ_1^1	ϕ_2^1	ϕ_3^1	ϕ_4^1	*EDA*	$\overline{TIF}$	$Pr_T(\mathcal{F})$	$Pr_U(\mathcal{F})$	*EEDA*
5xp1	7	64	10	6.8	67.2	1.5	24.5	86.9	16.4	5.4 E-4	1.2 E-4	81.3
alu1	12	19	8	11.8	81.7	0.0	6.4	97.9	16.8	4.5 E-4	1.4 E-5	80.0
alu2	10	68	8	6.9	62.1	0.0	31.0	92.6	12.0	4.5 E-4	5.4 E-5	72.2
apla	10	25	12	6.3	60.3	1.3	32.2	96.0	11.6	1.7 E-4	2.2 E-5	76.4
bis1	4	5	3	8.2	87.5	0.5	3.8	98.4	22.1	1.6 E-4	2.5 E-6	90.9
bis2	4	5	3	5.9	93.5	0.0	0.5	99.7	23.7	1.7 E-4	2.5 E-9	92.3
bis5	4	10	4	5.3	69.8	5.1	19.8	88.4	19.8	2.1 E-4	3.7 E-5	84.4
bis6	4	6	3	7.2	80.2	7.9	4.7	95.2	18.8	1.3 E-4	1.2 E-5	82.9
bw	5	22	28	2.1	76.8	6.6	14.5	94.9	22.6	5.7 E-4	6.0 E-5	80.7
dc1	4	9	7	3.5	69.4	2.6	24.4	90.8	21.9	2.2 E-4	3.4 E-5	83.1
dc2	8	39	7	5.2	65.6	4.3	24.9	93.4	12.2	2.8 E-4	3.4 E-5	82.9
dist	8	120	5	2.8	71.3	1.8	24.1	91.8	11.6	4.0 E-4	4.0 E-5	75.2
dk17	10	18	11	6.8	65.8	0.8	26.6	98.0	13.5	2.8 E-4	9.2 E-6	77.5
dk27	9	10	9	7.8	76.4	1.3	14.5	95.9	20.5	3.4 E-4	2.2 E-5	84.9
eg	3	6	5	5.7	77.1	5.9	11.3	91.0	29.2	2.5 E-4	3.1 E-5	75.4
f2	4	8	4	2.4	97.2	0.0	0.3	99.7	19.3	1.8 E-4	3.8 E-9	91.3
f51m	8	76	8	7.3	71.2	0.0	21.5	88.9	14.7	5.6 E-4	8.5 E-5	80.9
fsm1	10	15	6	7.6	74.4	5.0	13.1	97.7	9.6	1.6 E-4	5.6 E-6	80.1
fsm2a	6	18	6	5.9	74.6	4.6	14.9	95.0	16.1	2.8 E-4	1.7 E-5	79.8
gary	15	107	11	3.8	62.9	2.4	30.9	96.5	8.3	3.4 E-4	2.3 E-5	72.4
misex1	8	12	7	4.9	80.1	1.5	13.5	97.3	14.3	2.0 E-4	9.8 E-6	79.3
rd53	5	31	3	2.1	84.8	1.3	11.8	91.8	14.3	2.7 E-4	3.5 E-5	96.0
router	11	22	12	5.0	75.9	1.9	17.2	97.1	11.9	1.8 E-4	1.6 E-5	82.4
sao2	10	58	4	5.0	70.6	1.2	23.1	97.1	5.9	9.8 E-5	7.8 E-6	80.2
traffic	5	9	7	7.1	69.6	8.5	14.9	89.9	21.4	2.8 E-4	3.8 E-5	75.2
wim	4	9	7	5.7	59.2	4.0	31.1	91.5	27.5	3.5 E-4	3.6 E-5	78.9
z4	7	59	4	5.8	80.9	0.0	13.3	94.1	13.7	4.1 E-4	2.2 E-5	88.8

Table 3.6 Simulation results: mod 3 residue, Method 2.

Two significant observations can be made. Firstly, for some circuits, a non-negligible (up to 9 percent) proportion of faults, ϕ_3^1, produce only undetectable errors. Secondly, *EEDA* can be quite low (72 percent, in the case of "gary"), even though *EDA* remains fairly high (97 percent). Furthermore, canonic error pattern detection ability or *CEDA*, a mod 3 specific measure which is introduced in the next section, will be shown to range as low as 57 percent. It is also noteworthy that $\overline{TIF}$ ranges from 6 to 29 percent, and that the probability of detecting an error is, on average, 9.3 times greater than that of an undetected error occurring (*i.e.* 8.7 for Method 1, and 10.1 for Method 2).

Methods 1 and 2 are essentially equally effective, each slightly outperforming the other for some circuits. The measures ϕ_3^1, *EDA* and *EEDA* constitute a convenient basis for such quick comparisons.

Probability	Faults
2.5×10^{-5}	II.5, II.6, II.7, II.8
1.5×10^{-5}	I.1, I.3, III.2
1.0×10^{-8}	I.2, I.4, I.5, I.6, II.1, II.3, III.3, III.6
0.5×10^{-8}	I.7, I.8, II.2, II.4, II.9, II.10, III.1, III.4, III.5, III.7, III.8

Table 3.7 Fault probability estimates used to obtain $Pr_T(\mathcal{F})$ and $Pr_U(\mathcal{F})$.

3.2.2.2 Analysis

This section presents the results of analytical investigation into the effectiveness of mod 3 checking for PLA's. This analysis is restricted to values of n which are even integers, and a separate fault-free PLA generates the expected residue (*i.e.* Method 2). The alternate labelling of a PLA's n output lines: $Z = z_{n-1}, z_{n-2}, \ldots, z_1, z_0$, with $z_i \in \{0,1\}$, will be used. Let z denote the value of Z considered as a binary integer, $z=\sum z_i 2^{i\dagger}$. The notation $|a|_3$ will be used as a shorthand for "a mod 3". Let $R=|z|_3$ denote the residue of a PLA's output bit pattern. Residue R may also be considered a weighted sum of the bits of Z: $R=|\sum z_i 2^i|_3=|\sum z_i|2^i|_3|_3=|\sum z_i w_i|_3$, where $w_i=|2^i|_3$.

A faulty PLA produces erroneous output value, $\tilde{z}$, with residue $\tilde{R}$. Define a *directed error pattern* (*DEP*)‡ to be $E = e_{n-1}, e_{n-2}, \ldots, e_1, e_0$, with $e_i \in \{0,1,2\}$, where a $0\rightarrow1$ error is represented by $e_i=1$, and a $1\rightarrow0$ error by $e_i=2$ since $|-1|_3=2$. Therefore we have $\tilde{z}_i=|z_i + e_i|_3$. Let $e=|\sum e_i w_i|_3$. For any error pattern E applied to Z, the residue obtained, $\tilde{R}$, can be expressed as a function of the expected residue R, and e:

† All summations in this section, unless explicitly specified, range over $i = 0..n-1$.

‡ For the remainder of this section the term error pattern will be taken to mean directed error pattern.

$\widetilde{R}=|R+e|_3$. Clearly any error with $e=0$ will produce $\widetilde{R}=R$ and will thus go unnoticed. A single bit error produces erroneous output, $\widetilde{z}=z \pm 2^i$, with residue $\widetilde{R}=R \pm w_i$, and $e=\pm w_i$. Since $e \neq 0$, all single bit errors are detectable.

There are 3^n-1 possible n-bit error patterns, but only 2^n-1 are applicable to any given output Z. An error pattern containing k 0's is applicable to 2^k different output patterns. Since the PLA output Z was redefined as numeric quantity z only to permit arithmetic operations, and since there is no intrinsic meaning associated with this numeric value, the manner in which weights are associated with each z_i, and equivalently how the w_i are assigned in R and e, is totally arbitrary. The natural *bit weight arrangement* (*BWA*), $w_i=|2^i|_3$, is just one of $\binom{n}{n/2}$ ways to obtain a set of weights half of which are 1 and half 2. Only half of these arrangements are distinct in terms of which error patterns produce undetectable errors, since if $e=|\sum e_i w_i|_3=0$, then $2e$ is also 0, and is equivalent to multiplying each w_i by 2 mod 3. Thus complementary *BWA*'s are equivalent (*eg.* 112122 and 221211). Each *BWA* is capable of detecting $2/3$ of all error patterns, but no two non-complementary *BWA*'s detect the same set of errors. See Appendix B for a proof of this statement, as well as derivations of equations 3.13 through 3.17 to follow. It also appears that each *BWA* detects $2/3$ of all error events and $2/3$ of all unidirectional error events.

A given PLA under the assumed fault model will not generate all possible error patterns, hence it seems reasonable that some *BWA*'s will detect more errors than do others. This makes it difficult to determine maximal coverage of a mod 3 checking scheme because an optimal *BWA* must be found first.

Error patterns may be further categorized. Define a *canonic error pattern* (*CEP*) as a vector of n components chosen from $\{a, b, c\}$ constructed such that the first component is always an a, and the first non-a component, if any, is a b. The vector *aabaccba* is one example of a canonic error pattern for n=8. Any error pattern can be

mapped onto a unique canonic error pattern. Each canonic error pattern, other than *aa...a*, corresponds to six error patterns, specifically those found by all possible mappings from $\{a,b,c\}$ onto $\{0,1,2\}$. The set of error patterns corresponding to a canonic error pattern will be called a *canonic error class*. The canonic error class corresponding to *aabaccba* is {00102210, 00201120, 11012201, 11210021, 22021102, 22120012}, and that corresponding to *aaaaaaaa* is {00000000, 11111111, 22222222}. Note that the latter class represents the cases of no error, the error where an all 0 output is changed to all 1's, and the error where an all 1 output is changed to all 0's. Neither of these errors is detectable using mod 3 checking. The ternary number sequence from $100...0_3$ to $111...1_3$ enumerates exactly one representative error pattern from each possible canonic error pattern.

Lemma 3.14 Members of a canonic error class, generated from a canonic error pattern with length n even, either all have zero residue or all have non-zero residue.

Proof: Consider the generating canonic error pattern for the class. Pick any *BWA* with equal numbers of 1's and 2's. Determine w_a, w_b and w_c as the sums of the weights for the positions which contain a, b and c, respectively. Equation 3.9 is immediate.

$$|w_a+w_b+w_c|_3=|1\cdot n/2 + 2\cdot n/2|_3= 0 \tag{3.9}$$

Suppose the mapping $a,b,c \rightarrow 0,1,2$ produces a 0 residue as indicated by equation 3.10.

$$| 0\cdot w_a+ 1\cdot w_b+ 2\cdot w_c|_3= 0 \tag{3.10}$$

Multiplying both sides of equation 3.10 by 2 we obtain equation 3.11 which shows that mapping $a,b,c \rightarrow 0,2,1$ also produces a 0 residue.

$$| 0\cdot w_a+ 2\cdot w_b+ 1\cdot w_c|_3=0 \tag{3.11}$$

Equating 3.9 and 3.10 results in 3.12b showing that mapping $a,b,c \rightarrow 2,0,1$ has 0 residue.

$$| 0\cdot w_a+ 1\cdot w_b+ 2\cdot w_c|_3=|w_a+w_b+w_c|_3 \tag{3.12a}$$

$$| 2\cdot w_a+ 0\cdot w_b+ 1\cdot w_c|_3=0 \tag{3.12b}$$

Multiplying 3.12b by 2 shows that mapping $a,b,c \rightarrow 1,0,2$ also has 0 residue. Similarly,

equating 3.9 and 3.11 and then multiplying by 2 show that mappings $a,b,c \rightarrow 2,1,0$ and $a,b,c \rightarrow 1,2,0$ also have 0 residues. Thus if any of the error patterns has 0 residue, then all other class members must also have 0 residue. □

Lemma 3.14 should not be misinterpreted as saying that a canonic error class has 0 residue for all possible *BWA*'s. Rather, for those *BWA*'s where any class member has 0 residue all members must have 0 residue. The set of *BWA*'s which have non-zero residue for a canonic error class is called the *canonic class covering* set (*CCC*). The size of a *CCC* set depends on the structure of the canonic error pattern. A canonic error pattern is said to belong to *structural class* $\langle n_{min},n_{max}\rangle$, where $n_{min}=\min(n_a,n_b,n_c)$ and $n_{max}=\max(n_a,n_b,n_c)$ and n_a, n_b and n_c denote the number of a's, b's and c's in the canonic error pattern. The number of canonic error classes belonging to a structural class is given by expression 3.13, where r is the maximum number of identical non-zero values contained in the triple $(n_{max}, n_{min}, n-n_{max}-n_{min})$.

$$SCsize = \frac{\binom{n}{n_{max}}\binom{n-n_{max}}{n_{min}}}{r!} \tag{3.13}$$

All member canonic errors of a given structural class have *CCC* sets which are the same size (*i.e.* each error event in that structural class is detectable by the same number, but a different combination, of *BWA*'s). Also all *BWA*'s are capable of detecting the same number of canonic errors (but a different combination) from each structural class, and in general. Since each *BWA* detects $2/3$ of all error patterns, $2/3$ of all canonic error patterns are detectable by each *BWA*. The formula (3.15) for determining the number of *BWA*'s in the *CCC* sets associated with structural class $\langle A,B\rangle$, requires the set of solutions, S, for a linear inequality†. S is given by equation 3.14.

† An equality may be solved instead, and the result for *CCCsize*, from equation 3.15, is then subtracted from the total number of *BWA*'s possible.

$$
\begin{aligned}
S = \{\ (x,y) :\quad & x+2y \neq 2A+B \bmod 3, \\
& 0 \le x \le A, \\
& 0 \le y \le B, \\
& x+y \le n/2, \\
& (A+B) - (x+y) \le n/2\ \}
\end{aligned}
\tag{3.14}
$$

$$
CCCsize\langle A,B\rangle = \frac{1}{2}\sum_{i=1}^{|S|}\binom{A}{x_i}\binom{B}{y_i}\binom{n-A-B}{n/2-x_i-y_i} \tag{3.15}
$$

Equation 3.16 gives a formula for determining the number of structural class members detectable by any *BWA* (where r is as defined in equation 3.13). Structural class properties are related in equation 3.17. See Appendix B for derivations of equations 3.13 through 3.17.

$$
CEPdetected\langle A,B\rangle = \frac{1}{r!}\sum_{i=1}^{|S|}\binom{n/2}{x_i}\binom{n/2-x_i}{y_i}\binom{n/2}{A-x_i}\binom{n/2-(A-x_i)}{B-y_i} \tag{3.16}
$$

$$
SCDA = \frac{CEPdetected}{SCsize} = \frac{CCCsize}{\#BWA} \tag{3.17}
$$

Table 3.8 illustrates the error detection effectiveness of mod 3 checking in terms of percentage of detectable canonic error patterns within each structural class for n=4, 6, 8, 10, 12. One interesting observation is that unidirectional errors fall into structural classes with n_{min}= 0, and the structural classes with the worst coverage tend to be in this category.

Struct. Class	n					Struct. Class
	4	6	8	10	12	
⟨0,12⟩					0	
⟨0,11⟩					100	
⟨0,10⟩				0	45	
⟨0,9⟩				100	82	
⟨0,8⟩			0	44	55	
⟨0,7⟩			100	83	77	
⟨0,6⟩		0	43	52	56	
⟨0,5⟩		100	86	80		
⟨0,4⟩	0	40	49		58	⟨1,10⟩
⟨0,3⟩	100	90			73	⟨1,9⟩
⟨0,2⟩	33			56	64	⟨1,8⟩
				72	68	⟨1,7⟩
			57	64	66	⟨1,6⟩
⟨1,5⟩			71	67		
⟨1,4⟩		60	66		64	⟨2,8⟩
⟨1,3⟩		70			68	⟨2,7⟩
⟨1,2⟩	67			63	66	⟨2,6⟩
				68	68	⟨2,5⟩
⟨2,4⟩			63	65		
⟨2,3⟩			69		66	⟨3,6⟩
⟨2,2⟩		60			67	⟨3,5⟩
				67		⟨3,4⟩
					66	⟨4,4⟩

Table 3.8 Percentage *CEPdetected* by structural class for n=4, 6, 8, 10, 12.

The simulation results of the previous section are extended to investigate the significance of choice of *BWA* and the use of *canonic error detection ability* (*CEDA*), the proportion of all distinct canonic error patterns which are detectable, as an effectiveness measure. For each PLA considered, the error events determined earlier are converted to directed error patterns and then to a set of distinct canonic error patterns. Detectability is determined for all possible *BWA*'s, and the best and worst case results are reported in Table 3.9. The table also contains figures for *DEP* detection ability (*DEDA*). For PLA's with odd n, an additional error-free most significant 0-valued output is assumed. Although tending to be more pessimistic, *CEDA* appears to be as useful a measure as is *EEDA*.

PLA	n	EEDA		DEDA		CEDA		Natural BWA		
		N_{EE}	range	N_{DEP}	range	N_{CEP}	range	EE	DEP	CEP
5xp1	10	3459	67.6—81.3	400	59.5—80.0	242	59.9—78.9	81.3	80.0	78.9
alu1	8	1577	70.2—80.5	180	65.6—68.9	112	66.1—68.8	80.0	68.9	67.9
alu2	8	1996	70.4—74.1	422	65.6—72.5	201	65.2—70.6	72.2	65.6	65.2
apla	12	1231	69.9—82.2	318	61.9—75.8	191	61.8—73.8	76.4	72.3	70.7
bis1	3	33	75.8—90.9	15	60.0—80.0	8	62.5—75.0	90.9	80.0	75.0
bis2	3	26	92.3—100	17	88.2—100	9	88.9—100	92.3	88.2	88.9
bis5	4	77	80.5—85.7	38	65.8—76.3	12	66.7—75.0	84.4	73.7	75.0
bis6	3	35	82.9—88.6	20	70.0—80.0	10	70.0—80.0	82.9	70.0	70.0
bw†	28	1485	75.2—86.6	753	62.9—78.1	565	61.1—75.8	80.0	70.3	69.2
dc1	7	160	78.8—91.9	84	67.9—86.9	50	64.0—84.0	83.1	77.4	74.0
dc2	7	1669	72.6—83.3	205	64.9—77.6	119	67.2—78.2	82.9	77.6	78.2
dist	5	367	71.4—77.1	103	65.0—71.8	52	63.5—71.2	75.2	69.9	69.2
dk17	11	635	69.0—86.3	201	61.7—80.6	110	60.9—77.3	77.5	66.2	65.5
dk27	9	602	76.9—92.4	118	72.0—87.3	74	73.0—85.1	84.9	83.1	82.4
eg	5	61	75.4—86.9	39	61.5—79.5	23	65.2—78.3	75.4	61.5	65.2
f2	4	69	82.6—91.3	21	81.0—90.5	7	71.4—85.7	91.3	90.5	85.7
f51m	8	5371	69.4—80.9	352	62.5—76.4	164	62.2—76.2	80.9	76.4	76.2
fsm1	6	181	75.1—86.7	88	64.8—78.4	44	63.6—77.3	80.1	73.9	72.7
fsm2a	6	198	73.7—84.3	75	58.7—81.3	39	59.0—79.5	79.8	70.7	69.2
gary	11	3551	69.9—75.0	1237	63.9—71.5	688	64.5—71.2	72.4	69.0	68.8
misex1	7	174	77.6—86.2	89	66.3—82.0	57	63.2—78.9	79.3	70.8	68.4
rd53	3	25	84.0—96.0	12	66.7—91.7	6	66.7—83.3	96.0	91.7	83.3
router	12	512	72.5—84.6	230	61.3—77.4	143	59.4—75.5	82.4	72.2	71.3
sao2	4	101	76.2—82.2	46	65.2—76.1	11	63.6—72.7	80.2	76.1	72.7
traffic	7	137	69.3—87.6	81	59.3—85.2	47	57.4—80.9	75.2	69.1	66.0
wim	7	152	76.3—84.2	74	60.8—74.3	48	60.4—72.9	78.9	68.9	68.8
z4	4	134	78.4—88.8	34	67.6—79.4	11	63.6—81.8	88.8	79.4	81.8

N_{EE}, N_{DEP}, N_{CEP} number of distinct error events, directed error patterns and canonic error patterns, respectively
range range of measure (worst — best) obtained by considering all possible *BWA*'s
† only 1% of all *BWA*'s considered

Table 3.9 Simulation results: effect of *BWA* and *CEDA*, Method 2.

Choice of *BWA* does seem to have a significant effect on error detectability. From Table 3.9, the best and worst cases of *EEDA* (*CEDA*) have an average spread of 10.4 (13.2) percentage points, and a maximum observed spread of 18.3 (23.5).

Based on empirical and analytical evidence, the mod 3 checking scheme can be seen to be quite ineffective when applied to PLA's. One relatively simple enhancement which could improve this is to use k distinct *BWA*'s along with k sets of check bits. The overall proportion of error patterns detectable is $1-(1/3)^k$ which reduces to the expected $2/3$

when using just one *BWA*. Computation of additional residues need not involve a k-fold increase in hardware. For example, if R is the residue under BWA 212121, then the residue under 211221 is simply $|R + z_2 + 2z_3|_3$.

For a directed error pattern containing k 0's there are 2^k corresponding output patterns for which the error pattern is applicable. For each structural class one can determine the number of corresponding error events, and by multiplying this by *SCDA*, determine the number of detectable error events. This procedure may be repeated for just those structural classes corresponding to unidirectional errors. For n=6, 67.7 percent of all error events are detectable, but only 58.4 percent of unidirectional errors of size 2..6 are detectable. For n=8, 66.9 percent of all events and 62.4 percent of unidirectional errors of size 2..8 are detectable. As n increases, mod 3 checking is capable of detecting $2/3$ of all error events and $2/3$ of all unidirectional error events. Note that all unidirectional n–1 bit errors are detectable using mod 3, and unidirectional n bit errors are not.

3.2.3 Parity Check Scheme

The fault simulation described in section 3.2.2.1 is repeated here to demonstrate the effectiveness measures on simple parity. The measures obtained are shown in Table 3.10. The simulation is based on Method 2, where check bits are computed in a separate PLA which is assumed fault-free.

The parity scheme results of Table 3.10 can be compared to those obtained for mod 3 checking. Although the mod 3 scheme (Table 3.6) does have better scores for all measures, as is expected, the results for simple parity are surprisingly similar in many instances. For example, the results for "z4" are nearly identical for the two schemes. The "alu1" circuit measures are representative of several PLA's for which mod 3 is definitely superior, but not by very much. Although it may be unfair to make the comparison, it is interesting to note that parity is as effective for "rd53" as mod 3 is for "wim".

Alternatively, "f2" is an example of a substantial difference in effectiveness. The measures in Table 3.10 exhibit greater variability than does Table 3.6.

PLA	m	p	n	ϕ_1^1	ϕ_2^1	ϕ_3^1	ϕ_4^1	*EDA*	$\overline{TIF}$	$Pr_T(\mathcal{F})$	$Pr_U(\mathcal{F})$	*EEDA*
5xp1	7	64	10	6.8	57.9	8.0	27.3	76.1	14.4	4.4 E-4	2.2 E-4	60.5
alu1	12	19	8	11.8	78.0	0.0	10.1	94.8	16.3	4.2 E-4	4.3 E-5	57.4
alu2	10	68	8	6.9	59.0	0.9	33.2	89.7	11.6	4.3 E-4	7.3 E-5	55.5
apla	10	25	12	6.3	48.7	12.8	32.2	87.8	10.6	1.3 E-4	6.1 E-5	60.7
bis1	4	5	3	8.2	68.7	4.4	18.8	89.4	20.0	1.4 E-4	1.8 E-5	60.6
bis2	4	5	3	5.9	37.2	18.6	38.3	69.5	16.5	1.0 E-4	7.2 E-5	61.5
bis5	4	10	4	5.3	49.7	19.5	25.6	79.2	17.8	1.7 E-4	7.1 E-5	63.6
bis6	4	6	3	7.2	62.1	17.0	13.7	85.9	17.0	1.1 E-4	3.0 E-5	62.9
bw	5	22	28	2.1	69.2	15.5	13.3	90.5	21.5	5.1 E-4	1.1 E-4	70.2
dc1	4	9	7	3.5	58.2	8.9	29.4	83.2	20.1	1.9 E-4	5.7 E-5	70.6
dc2	8	39	7	5.2	57.1	8.9	28.8	86.2	11.3	2.4 E-4	7.0 E-5	60.0
dist	8	120	5	2.8	66.5	5.0	25.7	88.4	11.2	3.8 E-4	6.1 E-5	51.2
dk17	10	18	11	6.8	46.1	11.0	36.1	89.2	12.3	2.3 E-4	5.5 E-5	61.3
dk27	9	10	9	7.8	59.6	5.0	27.6	86.5	18.5	2.9 E-4	7.2 E-5	69.8
eg	3	6	5	5.7	61.8	12.3	20.3	78.8	25.3	2.0 E-4	8.1 E-5	68.9
f2	4	8	4	2.4	63.5	27.7	6.3	80.1	15.5	1.2 E-4	5.4 E-5	65.2
f51m	8	76	8	7.3	71.1	0.0	21.6	80.5	13.3	4.9 E-4	1.6 E-4	62.5
fsm1	10	15	6	7.6	59.0	13.0	20.4	93.7	9.2	1.5 E-4	1.5 E-5	63.0
fsm2a	6	18	6	5.9	66.4	4.8	22.9	90.4	15.3	2.6 E-4	3.8 E-5	62.1
gary	15	107	11	3.8	48.9	9.4	37.9	90.5	7.8	3.0 E-4	6.3 E-5	54.0
misex1	8	12	7	4.9	61.7	10.2	23.2	91.5	13.5	1.8 E-4	3.0 E-5	68.4
rd53	5	31	3	2.1	79.6	5.5	12.8	90.2	14.1	2.6 E-4	4.3 E-5	80.0
router	11	22	12	5.0	52.1	18.5	24.4	89.3	10.9	1.4 E-4	5.5 E-5	62.1
sao2	10	58	4	5.0	57.5	8.3	29.2	90.3	5.5	8.5 E-5	2.2 E-5	59.4
traffic	5	9	7	7.1	61.2	13.0	18.7	84.6	20.1	2.6 E-4	6.3 E-5	54.7
wim	4	9	7	5.7	50.7	4.7	38.9	86.9	26.1	3.3 E-4	6.0 E-5	69.1
z4	7	59	4	5.8	80.3	0.0	13.9	93.3	13.5	4.1 E-4	2.3 E-5	56.0

Table 3.10 Simulation results: simple parity check scheme, Method 2.

Table 3.11 shows the averages for several of the measures. The rightmost column indicates how much more likely it is that an error be detected than for an undetected error to occur. The measure which makes the greatest distinction between the two checking strategies is *EEDA*.

Strategy	ϕ_3^1	*EDA*	$\overline{TIF}$	*EEDA*	$Pr_T(\mathcal{F})/Pr_U(\mathcal{F})$
mod 3	2.6	94.4	16.7	81.7	10.1
parity	9.7	86.5	15.2	62.6	4.1

Table 3.11 Averages of measures from Tables 3.6 and 3.10.

3.3 MINIMAL CODEWORD SIZE

Consider the on-line checking architecture depicted in Figure 3.10, where the primary inputs are fed into a check-bit predictor circuit P, and the actual check-bits are computed from the CUC outputs by circuit Q, and the two sets of check-bits are compared by checker C†. This arrangement corresponds to Method 2 of section 3.2.2.

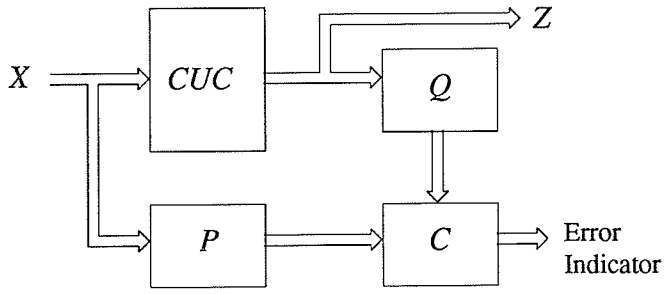


Figure 3.10 Architecture for on-line checking.

The set of error events obtained for a PLA (or any combinational circuit) defines a symmetric relation, ~′, which identifies pairs of output patterns which should not be associated with identical check-bit patterns. The complement of ~′, relation ~, is a symmetric relation identifying pairs of output patterns which may be associated with identical check-bit patterns. Relation ~ is in fact a compatibility relation. The number of subsets in a minimal grouping, obtained from maximal compatibles of ~, represents the minimal number of distinct check-bit patterns required to ensure detectability of all error

† In general check-bit pairs need only be consistent, not necessarily equal. For example the concatenated check-bits from P and Q could be defined to form a two-rail or m-out-of-n code which is checked by C. Relation ~′ would have to be considered in the subsequent analysis.

events considered. Thus the minimal number of check bits required may be determined. Fault enumeration is required to obtain the error events. Constructing ~ is straightforward, there are well-known algorithms for finding maximal compatibles†, and finding a minimal grouping is the same as finding a minimal cover. Assignment of check-bit patterns to output patterns is not considered here.

Let $\mathcal{Z}$ denote the set of output patterns normally produced by the CUC, and let $\tilde{\mathcal{Z}}$ denote the set of erroneous output patterns contained in the error events determined for the CUC. The patterns $\hat{\mathcal{Z}}=\tilde{\mathcal{Z}}-\mathcal{Z}$ represents those error patterns which may arise due to a fault but which are not normally produced by the CUC. Function Q may map all output patterns from $\hat{\mathcal{Z}}$ onto a single check-bit pattern which is distinct from all check-bit patterns associated with members of $\mathcal{Z}$. Thus a near-optimal Q may be obtained by solving the problem on the reduced compatibility relation based on only those error events which produce erroneous outputs which are contained in $\mathcal{Z}$, and then, if necessary, adding one more check-bit pattern to handle the rest. The solution found in this way is at most only 1 larger than the optimal, and this is significant only if the optimal solution size is 2^k-1 check-bit patterns which may be realized using k check bits. Adding one check-bit pattern would then require an additional check bit.

As an example of this procedure, consider the PLA of Figure 1.2. For this PLA, $\mathcal{Z}=\{0, 3, 5, 6, 7\}$, and $\hat{\mathcal{Z}}=\{1, 2, 4\}$. The procedure described in section 3.1.2.3 was used to find the set of error events for all Class I, II and III faults; these are shown in Figure 3.11a. The compatibility relation is shown in Figure 3.11b, and the maximal compatibles in Figure 3.11c. Three of the maximal compatibles form a minimal grouping of $\mathcal{Z}$ (in this case, it is also a partition): {0}, {3, 5, 6} and {1, 2, 4, 7}. Three distinct check bit patterns must be defined, and at least 2 check bits are required. The procedure based on the reduced compatibility relation, corresponding to the upper left submatrix of Figure 3.11b, for this example, produces an identical solution.

† The method we use is an implementation of Prather's algorithm [Prat67, p. 34] enhanced with a data structuring technique which improves computational efficiency.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Z^i	0	0	0	0	0	0	3	3	3	3	5	5	5	5	6	6	6	6	7	7	7	7
$\tilde{Z}^i$	1	2	3	4	5	6	0	1	2	7	0	1	4	7	0	2	4	7	0	3	5	6

a) error events

~	0	3	5	6	7	1	2	4
0	1	0	0	0	0	0	0	0
3	0	1	1	1	0	0	0	1
5	0	1	1	1	0	0	1	0
6	0	1	1	1	0	1	0	0
7	0	0	0	0	1	1	1	1
1	0	0	0	1	1	1	1	1
2	0	0	1	0	1	1	1	1
4	0	1	0	0	1	1	1	1

b) compatibility relation ~

	0	3	5	6	7	1	2	4
{0}	1	0	0	0	0	0	0	0
{3, 4}	0	1	0	0	0	0	0	1
{3, 5, 6}	0	1	1	1	0	0	0	0
{1, 6}	0	0	0	1	0	1	0	0
{2, 5}	0	0	1	0	0	0	1	0
{1, 2, 4, 7}	0	0	0	0	1	1	1	1

c) maximal compatibles of ~

Figure 3.11 Example determination of lower bound on number of check bits.

3.4 SUMMARY

This chapter presented an analysis of PLA faults, and showed how they may be classified into four classes which allows considerable savings when performing error analysis or fault simulation. The important concept of covered faults, within the context of on-line checking, was introduced. It was shown to be sufficient to only consider single stuck-at faults and product line break faults to obtain an enumeration of all error events produced by all faults from the model. An algorithm was presented which plays a central role in the enumeration process. A simplified error model was shown to exist for nonconcurrent PLA's.

Several quantitative measures used to evaluate the effectiveness of on-line checking strategies were reviewed and expressed within a unified framework. New variations of such measures, Φ^1 and *EEDA*, were introduced, and were shown to provide useful new criteria for the evaluation of partially self-checking strategies. The measures were applied against two forms of a scheme based on a mod 3 residue code, and a simple parity check

strategy. The mod 3 schemes yield effectiveness measures which are slightly better than those for the parity scheme.

A detailed analysis of mod 3 checking was presented. It was shown that error detection depends significantly on the way bit weights are assigned to output lines. A canonical form for error patterns was introduced, and a mod 3 specific effectiveness measure (*CEDA*) was suggested. It was argued that the mod 3 checking scheme performs rather poorly relative to this new measure.

Finally, a procedure for determining the minimal number of check bit patterns (and check bits) required to detect all errors due to modelled faults, was described.

Chapter 4

CHECKING NONCONCURRENT PLA'S USING SIMPLE PARITY

This chapter shows how to augment a PLA in such a way that a simple parity check on its outputs is capable of detecting all error events due to single Class I faults, except fault type I.8, as defined in section 3.1.2. Due to Theorem 3.1 and the definition of fault equivalence, all errors resulting from single Class II or III faults will also be detected. Section 4.1 assumes that the PLA is nonconcurrent, and section 4.2 discusses methods for transforming a normal PLA into a nonconcurrent form. Portions of this chapter have appeared as [Marc88] and [Marc89a].

4.1 PROPOSED STRATEGY

It was shown in Theorem 3.2 that, for nonconcurrent PLA's, error events due to Class I faults, except for fault type I.8, may be characterized as producing: an all 0 output, a single bit error, or an error of type ξ. If it is arranged that each output pattern has odd weight, then a simple odd parity checker is capable of detecting both an all 0 output and single bit errors†. It will be shown that parity can also detect errors of type ξ. Errors due to fault type I.8 are considered separately.

4.1.1 Description

The general strategy is to define extra check bit outputs for the nonconcurrent PLA in a way that permits a simple parity check to detect the target errors. The original output patterns of the PLA might not have odd weight. Thus a parity check bit will usually be required to ensure that all output patterns have odd weight. Since the PLA is

† Odd parity is assumed throughout this chapter. The method can be adapted to use even parity, but the all 0 output pattern must be treated as a noncodeword.

nonconcurrent, check bits may be appended to the output patterns without the need to redo minimization, because each product term specifies an output pattern exactly.

4.1.1.1 Fault Type I.8

Fault type I.8 is defined as a product line break (located in the OR-plane). Output bits to the right of the break (from the perspective of Figure 1.2) appear stuck-at-0 whenever that product line is selected. It is sufficient to detect the disappearance of just the rightmost non-zero output. One method might be to append a constant 1 output to all product lines, however that line stuck-at-1 is an undetectable fault which would mask detection of errors due to product line breaks. The proposed method is to define two check bit output lines in the rightmost positions, and program these lines as a 1-out-of-2 code, making sure that both codewords get used. These two lines may either be used as a 1-out-of-2 error indicator, or be combined with other indicators to form a two-rail code. To help reduce overhead, these two check bits could be defined as the parity check bit and its complement. The parity checker need not include the second of these check bits, as it is checked elsewhere anyway. Generally, any PLA output and its complement, in the rightmost positions, may form the 1-out-of-2 code used to detect errors due to fault type I.8. Thus no more than a single check bit is required, independent of the checking strategy used for the remaining faults. From Corollary 3.1, detection of errors due to fault type I.8 ensures that an erroneous all 0 output will also be detected. Hence either odd or even parity may be used to detect the single bit errors and those of type ξ.

The 1-out-of-2 code also detects some errors of type ξ. If the codewords 01 and 10 are assigned to output patterns felicitously (using two check bits), then the remaining errors of type ξ may become significantly easier to detect. This consideration is deferred until section 4.1.6.

4.1.1.2 Errors of Type ξ

As stated in Definition 3.5, an error of type ξ is said to occur if the erroneous output pattern is the bitwise OR of some two codewords. Codewords in this strategy are defined to have odd weight. Therefore an error of type ξ is detectable if the erroneous output pattern, $\widetilde{Z}$, has even weight. Alternatively Lemma 4.1 specifies an *even weight intersection criterion*, *i.e.* the condition the set of codewords must meet in order for all errors of type ξ to be detectable.

Lemma 4.1 In order for a simple odd parity check to detect all type ξ errors in a nonconcurrent PLA, the intersection of any two codewords must have even weight.

Proof: Consider two codewords, Z^i and Z^j; each has odd weight. The error pattern $\widetilde{Z}=Z^i \vee Z^j$ can be expressed as $\widetilde{Z} = \hat{Z}^i \vee I \vee \hat{Z}^j$ where $\hat{Z}^i$, I and $\hat{Z}^j$ are pairwise disjoint and $I= Z^i \cap Z^j$, $\hat{Z}^i = Z^i - I$ and $\hat{Z}^j = Z^j - I$. $\hat{Z}^{i(j)}$ represents the 1 bits which are in $Z^{i(j)}$ but not in $Z^{j(i)}$. $Z^i = \hat{Z}^i \vee I$ and $Z^j = \hat{Z}^j \vee I$. If the weight of I is odd then the weights of $\hat{Z}^i$ and $\hat{Z}^j$ must both be even and thus the weight of $\widetilde{Z}$ will be odd and the error will go undetected. If the weight of I is even then the weights of both $\hat{Z}^i$ and $\hat{Z}^j$ must be odd and the weight of $\widetilde{Z}$ will therefore be even. □

Lemma 4.2 Any set of odd weight codewords satisfying the even weight intersection criterion of Lemma 4.1 is unordered.

Proof: Since the intersection of any two codewords has even weight, and since all codewords have odd weight, each codeword has 1's in an odd number of positions where the other codeword has 0's, hence each pair of codewords is unordered. □

One method of defining check bits to meet the even weight intersection criterion is described in the next section. In practice, however, this criterion is incompatible with having odd weight codewords. If the parity check bit is defined before the intersection

criterion is met, then the check bits added must have even weight. If the parity bit is determined after ensuring even weight intersections, then codewords with a "1" parity bit would then have odd weight intersections. The problem is resolved by using several parity check "pseudo-outputs" instead of a single explicit parity check output line. If a product term has an even weight output part, then a new "pseudo-output" line is introduced and that product term is defined to be the sole term asserting that "pseudo-output." The codeword now has odd weight and the even weight intersection criterion is not affected as the new output is not asserted by any other product term. If there are k such product lines, then it appears that a codeword has k additional "pseudo-output" bits. Rather than realizing a "pseudo-output" as an actual PLA output, the product term line itself may be used as an input to the parity checker. The PLA is nonconcurrent so at most one of the k "pseudo-outputs" is normally a 1. If a fault causes an even number of "pseudo-outputs", each associated with the same normal output pattern, to be erroneously selected together, then this fault is detectable. (If limited concurrency is allowed, as defined in section 4.2.1, then actual parity check output lines would have to be created, but product terms with identical output parts could share a common check bit line.)

4.1.2 Method to Ensure Even Weight Intersection

This section describes a method for augmenting a set of binary vectors with check bits to produce codewords such that the intersection of any two different codewords has even weight. Let $(D^1, D^2, \ldots, D^p)$ denote the output patterns associated with the p product lines of a nonconcurrent PLA before augmentation. Let $\mathcal{Z} = \{Z^1, Z^2, \ldots, Z^N\}$ denote the set of N distinct output patterns produced by a PLA. When a new check bit is introduced, only those codewords which are 1 in the new position have the parity of their intersections with each other changed. The odd weight property for codewords is not addressed until after the even weight intersection criterion is met.

If multiple product terms produce the same output pattern, then they may all be augmented with the same check bit values. The correct output is produced even in the presence of a fault which causes two or more of such product terms to be selected together. Furthermore, such faults do not enable a second fault to produce undetectable errors. Suppose fault f_1 causes product term line p_j to be erroneously selected along with p_i, for some input values, and $D^i = D^j$. The fault is clearly undetectable, but neither does it allow undetectable errors to result from some other fault f_2. There are two cases related to f_2. In the first p_i is not supposed to be selected. If it is not selected then f_1 has no effect on the detectability of f_2. If, due to f_2, p_i is selected (with or without p_j) resulting in an error of type ξ involving p_i, then the error will be detected. In the second case p_i is supposed to be selected. If p_i is selected (but not p_j), or if p_i is not selected, then f_1 has no effect on the detectability of f_2. If both p_i and p_j are always selected whenever f_2 manifests itself, and if an error of type ξ occurs involving p_i, or if one bit of p_i changes from 0 to 1, then it will be detected. If f_2 causes one or more bits of p_i to change from 1 to 0, then f_1 will mask this, but the correct codeword will still be output. Under these conditions, f_1 is capable of masking a sequence of faults each which change bits of p_i or p_j from 1 to 0. If bits only change one at a time, then either the correct codeword continues to be output or the fault is detectable, but if all or multiple rightmost bits change to 0 at once due to a single fault, then an error which is undetectable by simple parity may occur. At least 4 faults, including f_1, must occur for this to happen.

The problem is to augment $\mathcal{Z}$ such that the intersection of any two elements has even weight. It may be restated in terms of a *conflict graph* which is an undirected graph with N vertices, vertex V_j corresponding to Z^j. Vertices V_j and V_k are *adjacent* (*i.e.* they are joined by an edge) if, and only if, Z^j and Z^k have an odd parity intersection, *i.e.* that pair of output patterns conflicts with the testing goal of having even parity intersections. The operation of adding a new check bit output to the PLA corresponds to performing an

edgewise exclusive-OR (local complement) on the conflict graph with the clique on all vertices which have a 1 value for the new check bit. A *clique* is defined as a complete subgraph, *i.e.* a subset of vertices which are all adjacent to each other, and all other vertices are isolated. The restated goal is to reduce the conflict graph to a set of isolated vertices. The solution is thus an edgewise exclusive-OR sum of cliques decomposition of the original conflict graph. Each clique in the decomposition corresponds to a check bit, and the vertices within that clique identify the product terms which output a 1 for that check bit.

Algorithm 4.1 is one solution to the graph decomposition problem which is easily obtained. Consider the graph represented as $G=(V, E)$, where V is the set of vertices and E the set of edges. Let $K(V_i)$ denote the clique formed from vertex V_i and all vertices adjacent to it.

Algorithm EDG(G)

- $S \leftarrow \{\}$ (1)
- **repeat until** $E = \varnothing$ (2)
  - select a V_i with degree > 0 (3)
  - $S \leftarrow S \cup K(V_i)$ (4)
  - $G \leftarrow G \oplus K(V_i)$ (5)
- **return** S (6)

Algorithm 4.1 Edgewise ⊕ decompose conflict graph.

The effect of line 5 of Algorithm 4.1 is to isolate the selected vertex. At least one vertex is isolated each iteration. A graph with N vertices will be decomposed in at most N–1 steps. However, an n step decomposition is trivially obtained by duplicating the n-bit patterns in $\mathcal{Z}$ to form check bits. If the intersection of two patterns has weight t before duplication, then it will have weight $2t$, which is always even, after duplication.

The number of steps required (PLA check bit outputs) is sensitive to the order in which vertices are selected by Algorithm 4.1. A minimal solution is desirable. Figure 4.1 illustrates the decomposition of a 5-vertex, 6-edge conflict graph, using four iterations of the loop in Algorithm 4.1 to produce a four clique solution. For this example, step 3 of the algorithm selects a vertex with maximal degree. This solution is not minimal.

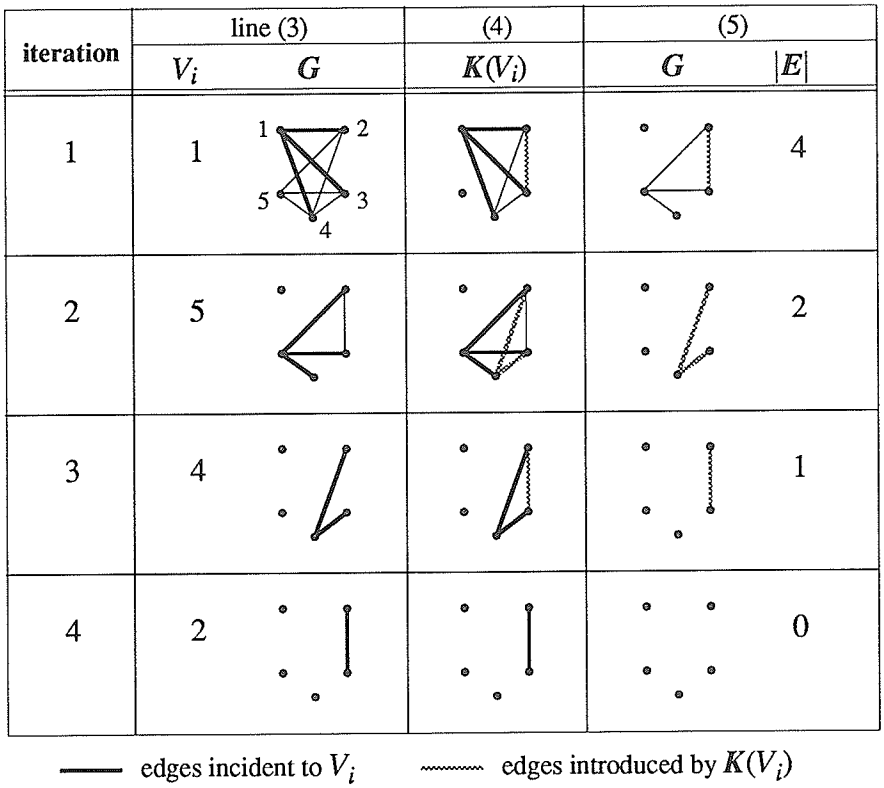
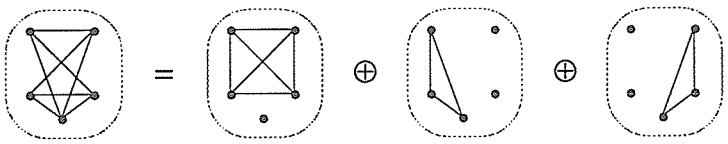


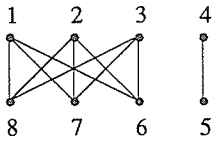
Figure 4.1 Example of Algorithm 4.1.

A minimal solution requiring 3 cliques, found using Algorithm 4.1 with an improved vertex selection heuristic described shortly, is ({1, 2, 3, 4}, {2, 3, 5}, {4, 5}), where the sequence of vertex sets represents the cliques $K(V_i)$, and isolated vertices are underscored. Figure 4.2a shows another decomposition of the example graph from Figure 4.1, demonstrating that it is possible to have a decomposition where no single component clique isolates any vertices; hence it is impossible for Algorithm 4.1 to arrive at this decomposition. For this graph, both solutions are the same size, but for the graph shown

in Figure 4.2b, no vertex selection sequence results in the optimal solution. The best solution by Algorithm 4.1 is ({4, 5}, {1, 6, 7, 8}, {2, 3, 6, 7, 8}, {2, 3}), while a minimal solution is ({1, 2, 3, 4, 5, 6, 7, 8}, {1, 2, 3, 4, 5}, {4, 5, 6, 7, 8}).



a) alternative decomposition to Figure 4.1



b) another graph

Figure 4.2 Conflict graphs with decompositions unattainable by Algorithm 4.1.

Algorithm 4.2 is a heuristic algorithm for selecting a vertex for isolation in line 3 of Algorithm 4.1. Let $G_i=G\oplus K(V_i)$, where $G_i=(V, E_i)$, denotes the conflict graph (with edge set E_i) produced by edge-wise exclusive OR'ing G with $K(V_i)$ to isolate V_i. Let $\overline{G}$ denote the graph complement of G (*i.e.* $\overline{G}$ has the same vertex set as G, and a pair of vertices in $\overline{G}$ are joined by an edge if, and only if, there is no edge between the same vertices in G).

- **if** $\exists\ V_i$ such that G_i has $k_i > 1$ more isolated vertices than G **then** (1)
  - from these select the V_i with maximum k_i (2)
    - break ties by selecting the V_i with minimum $|E_i|$ (3)
      - break ties by selecting the V_i with smallest value of i (4)
- **else if** $\exists\ V_i$ with degree > 1 such that $K(V_i)$ is a subgraph of G **then** (5)
  - from these select the V_i with largest degree (6)
    - break ties by selecting the V_i with smallest value of i (7)
- **else if** $\exists\ V_i$ such that $|E_i| \neq |E|$ and $G_i \neq \overline{G}$ **then** (8)
  - from these select the V_i with maximum $|E_i|$ (9)
    - break ties by selecting the V_i with smallest degree (10)
      - break ties by selecting the V_i with smallest value of i (11)
- **else** select the V_i with smallest value of i (12)

Algorithm 4.2 Heuristic to "select a V_i with degree > 0" for Algorithm 4.1.

Algorithm 4.2 attempts to isolate multiple vertices each iteration of Algorithm 4.1. Failing that, it tries to keep the edge count high since a subgraph which is almost a clique may tend to allow the isolation of multiple vertices in a subsequent iteration.

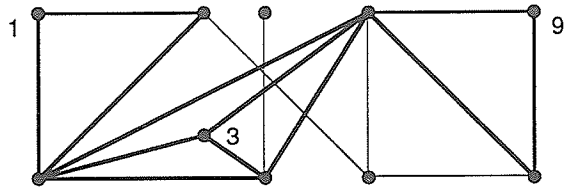


Figure 4.3 Conflict graph, G, for "fsm2b".

As an example of the operation of Algorithm 4.2, consider the conflict graph shown in Figure 4.3. There is no vertex which, if selected, would isolate additional vertices, *i.e.* k_i=1 for all i. However, there are three vertices (1, 3 and 9) for which $K(V_i)$ is a subgraph of G (line 5 of Algorithm 4.2), and since vertex 3 has largest degree, it is selected. G_3 is shown in Figure 4.4.

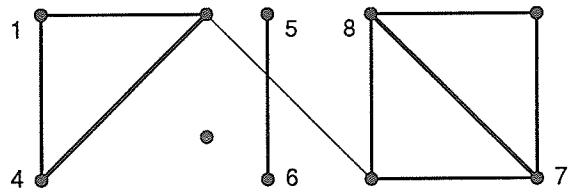


Figure 4.4 Conflict graph G_3.

For G_3, there are three vertices (1, 5 and 7) for each of which $K(V_i)$ isolates two vertices. (When a clique isolates two or more vertices, any of these isolated vertices may be used to generate the clique. We shall use the isolated vertex with smallest vertex number.) Since $|E_i| = 7, 9, 6$ for i=1, 5, 7, respectively, vertex 7 is chosen, according to line 3 of Algorithm 4.2. Figure 4.5 shows G_7.

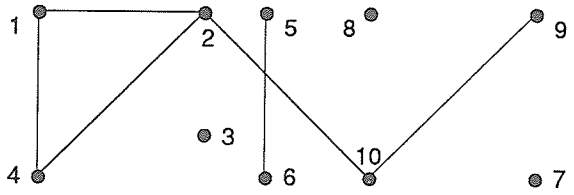


Figure 4.5 Conflict graph G_7.

For this example, four more iterations are required to decompose the original graph. The complete sequence of cliques ultimately obtained is:

{3, 4, 6, 8} — line 6 of Algorithm 4.2,

{7, 8, 9, 10} — line 3,

{1, 2, 4} — line 3,

{5, 6} — line 2,

{2, 10} — line 11,

{9, 10} — line 2.

4.1.3 Example

This section demonstrates the application of the testing strategy on the small example PLA of Figure 1.2. Figure 4.6a shows the PLA specification after minimization by the program McBOOLE [Dage86]. Figure 4.6b shows this specification as converted to a nonconcurrent form by Algorithm 4.4 (*cf.* section 4.2.2). Product terms have also been rearranged so that those which produce the same output pattern are grouped together and labelled with a unique conflict graph vertex number (circled). Figure 4.6c shows the conflict graph. Clearly a single check bit output which is 1 for all product terms associated with vertices ①, ② or ③ will satisfy the testing criteria. Only the two product terms associated with vertex ⑤ need to be extended to ensure odd parity of all output patterns. Instead of actually adding a parity check output column, the two product lines

themselves are fed directly into the parity checker, thus the augmented PLA requires two instead of three extra outputs. A checker verifies that the parity is odd and that the final two output bits are in a 1-out-of-2 code (which detects fault type I.8, *i.e.* broken product term lines). Figure 4.6d shows the final PLA.

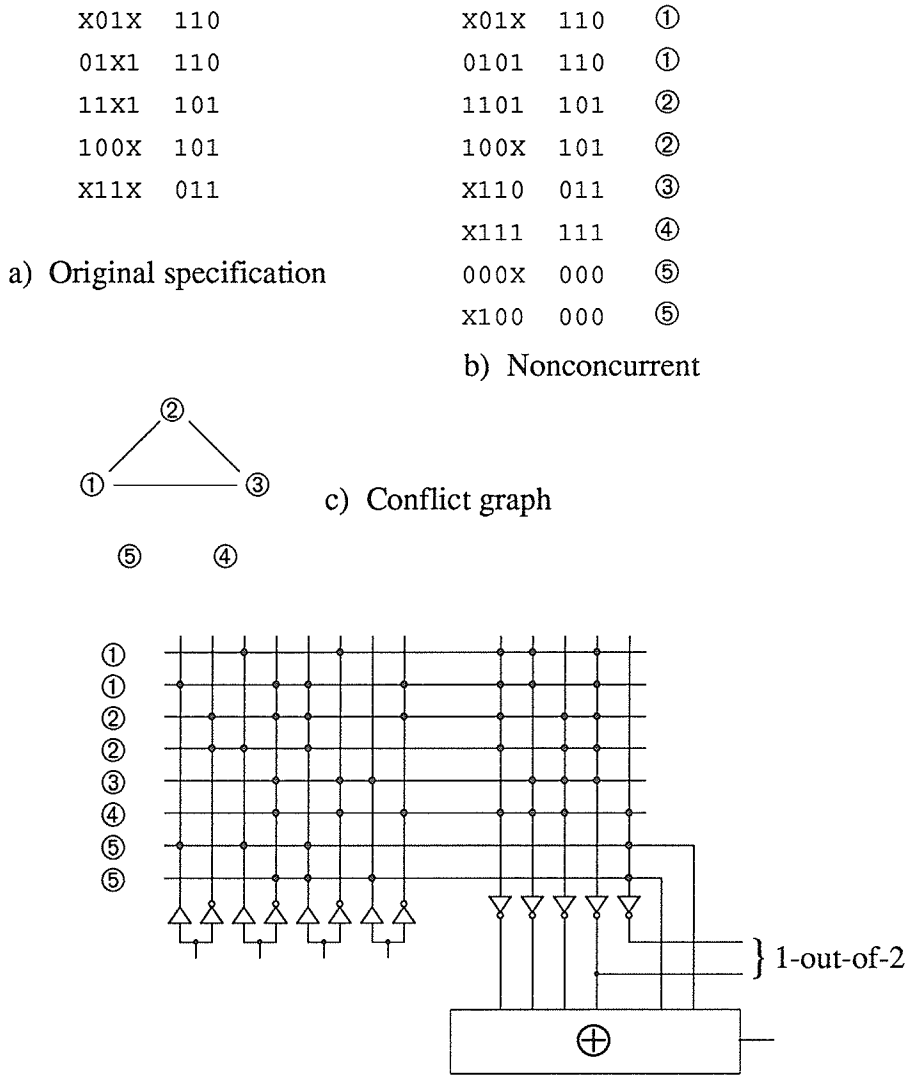


Figure 4.6 Example PLA made on-line testable.

4.1.4 Experimental Results

Both the heuristic of Algorithm 4.2 and exhaustive methods for selection of vertex V_i, in step 3 of Algorithm 4.1, have been applied to some small PLA's. There are too many possible vertex selection sequences in a complete exhaustive search, so this study performs 2000 trials on each PLA. The best solution size found actually occurred for a large proportion of these sequences, and this suggests that a heuristic should stand a fair chance at finding a good solution, and that it may be a minimal solution. Table 4.1 presents the results of this investigation, showing the solution size for conflict graph decomposition resulting from the use of the heuristic of Algorithm 4.2 and minimum/maximum solution sizes obtained when performing 2000 trials from the set of all vertex selection sequences. Normally $|V| = N$ in Table 4.1, except that any vertices which are initially isolated are excluded.

The results in Table 4.1 indicate that the number of check bits required tends to be almost the same as n, the number of outputs of the original PLA. Doubling the number of PLA outputs is still less expensive than duplicating the entire PLA. The example of the previous section, and the "router" PLA in Table 4.1, however, require considerably fewer check bits. This suggests that certain PLA's (*i.e.* those where N is much smaller than 2^n) may be better suited to this type of checking. Previously, it was shown that no more than n check bits are ever required (*i.e.* simply duplicate the outputs). Although the vertices selected by Algorithm 4.2 may exceed this bound (*eg.* "alu1"), Algorithm 4.1 (for some selection sequence) seems to always be able to find a solution with n or fewer check bits. The table also shows that some vertex selection sequences can result in large number of cliques in the graph decomposition (approaching the N–1 bound) thus showing that Algorithm 4.1 is quite sensitive to vertex selection order. The heuristic of Algorithm 4.2 performs quite well.

PLA	m	n	N	$\|V\|$	$\|E\|$	$\|S\|_{\text{Alg 4.2}}$	$\|S\|_{\text{2000 trials}}$
all4	4	4	15	15	56	4	4 — 14
all5	5	5	31	31	240	5	5 — 29
all6	6	6	63	63	992	6	6 — 59
all7	7	7	127	127	4032	7	7 — 122
all8	8	8	255	255	16256	8	8 — 248
alu1	12	8	80	80	1600	10	8 — 64
alu2	10	8	67	67	1120	12	8 — 60
apla	10	12	37	37	285	12	12 — 35
bis1	4	3	5	5	5	3	3 — 3
bis2	4	3	4	3	3	1	1 — 1
bis5	4	4	8	8	14	6	4 — 6
bis6	4	3	5	5	6	4	3 — 4
bw	5	28	22	22	122	20	18 — 21
dc1	4	7	10	10	22	7	6 — 9
dc2	8	7	91	91	2064	7	7 — 83
dist	8	5	21	21	106	5	5 — 18
dk17	10	11	24	24	90	10	10 — 22
dk27	9	9	38	38	299	10	9 — 30
eg	3	5	6	5	5	3	3 — 4
f2	4	4	10	10	24	4	4 — 9
fsm1	10	6	13	13	38	6	6 — 12
fsm2a	6	6	14	13	36	5	5 — 11
fsm2b	5	7	11	10	16	6	6 — 9
fsm2c	8	7	24	24	137	8	7 — 21
gary	15	11	69	69	1087	14	11 — 46
misex1	8	7	10	10	23	7	6 — 9
misex2	25	18	34	28	110	14	12 — 25
rd53	5	3	5	5	5	3	3 — 4
router	11	12	15	15	47	7	7 — 14
sao2	10	4	9	9	17	4	4 — 8
traffic	5	7	8	8	12	6	6 — 7
vg2	25	8	23	23	127	8	8 — 22
wim	4	7	11	11	26	9	7 — 10
z4	7	4	15	15	56	4	4 — 14

Table 4.1 Conflict graph decomposition for some PLA's.

4.1.5 Analysis

It is desirable to have analytically derived results on the number of check bits required. The check bit assignment problem can be expressed as a set of equations, however since these are quadratic, no general analytical method exists to solve them. However, for the case where $N = 2^n$ some analytical results may be obtained.

4.1.5.1 $N = 2^n$ Case

First observe that if n check bits which simply duplicate the n original outputs are defined, then the intersection between any two such codewords will have even weight. Therefore n is an upper bound on the minimal solution size. Ignore for the moment the fact that all codewords have even weight. It will be shown that n is also a lower bound.

Each check bit may be considered a function of the n original outputs. Let $A^i = a_1^i$, a_2^i, ..., a_n^i represent the n check bits associated with output pattern $Z^i = z_1^i, z_2^i, \ldots, z_n^i$. Two codewords, (Z^i, A^i) and (Z^j, A^j) satisfy the even weight intersection criterion if equation 4.1 is satisfied (all operations done in $GF(2)$).

$$\sum_{k=1}^{n} z_k^i z_k^j \oplus a_k^i a_k^j = 0 \tag{4.1}$$

Expressed in terms of inner products this becomes equation 4.2:

$$(Z^i \cdot Z^j) = (A^i \cdot A^j) \tag{4.2}$$

Lemma 4.3 With $N = 2^n$, $n > 2$, and for all $i \neq j$ there exists a Z^l, $i \neq l$ and $j \neq l$, such that $(Z^i \cdot Z^l) \neq (Z^j \cdot Z^l)$.

Proof: For the case where $Z^i \neq \overline{Z}^j$, define Z^l according to equation 4.3:

$$z_k^l = \begin{cases} \overline{z}_k^i, & z_k^i = z_k^j \\ 1, & z_k^i \neq z_k^j,\ z_h^i = z_h^j\ \forall\ h<k \\ 0, & z_k^i \neq z_k^j,\ \exists\ h<k : z_h^i \neq z_h^j \end{cases} \tag{4.3}$$

The first case ensures that $Z^i \neq Z^l$ and $Z^j \neq Z^l$ while the last two cases ensure that $(Z^i \cdot Z^l) \neq (Z^j \cdot Z^l)$. For the case where $Z^i = \overline{Z}^j$, assume, without loss of generality, that $wt(Z^i) \geq wt(Z^j)$ and that $z_k^i = 1$. Define Z^l to be all 0's except for $z_k^l = 1$. (Note that for $n = 2$, the situation $Z^i = 01$ and $Z^j = 10$ would result in $Z^l = Z^i$.) □

Lemma 4.4 For $N = 2^n$ all A^i are distinct.

Proof: From Lemma 4.3, for all $i \neq j$ there exists a codeword (Z^l, A^l), $i \neq l$ and $j \neq l$, such that $(Z^i \cdot Z^l) \neq (Z^j \cdot Z^l)$. From equation 4.2 we have $(A^i \cdot A^l) \neq (A^j \cdot A^l)$ and it follows that $A^i \neq A^j$. □

It can be shown, for $n>2$, that $A^i = Z^i$ for $Z^i = \mathbf{0}$ or $\mathbf{1}$, and that $|wt(Z^i, A^i)|_2 = 0$.

Theorem 4.1 For $N = 2^n$, at least n check bits are required to ensure the even weight intersection criterion.

Proof: From Lemma 4.4, $N = 2^n$ distinct check symbols are required thus at least n check bits are necessary to produce the 2^n distinct symbols. □

As stated above, the n check bits may simply be defined by duplicating the original outputs. Other check bit assignments exist. Check bit vectors may be determined by multiplying output bit vectors by certain $n \times n$ parity check matrices, *i.e.* $A^i = Z^i \cdot \mathbf{H}_n$. Clearly the identity matrix, $\mathbf{I}_n$, and any row/column permutation of it, may be used. It is conjectured that only row/column permutations of those matrices generated from the recursive definition given by equation 4.4 produce acceptable check bit assignments.

$$\mathbf{H}_n = \begin{cases} \mathbf{I}_n, & n \leq 2 \\ \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \mathbf{H}_{n-1} & \\ 0 & & & \end{pmatrix}, & n > 2,\ n \text{ is odd} \\ \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & & & \\ \vdots & \vdots & & \mathbf{H}_{n-2} & \\ 0 & 0 & & & \end{pmatrix} \text{or} \begin{pmatrix} 0 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 1 & \cdots & 1 \\ 1 & 1 & & & \\ \vdots & \vdots & & \mathbf{H}_{n-2} & \\ 1 & 1 & & & \end{pmatrix}, & n > 2,\ n \text{ is even} \end{cases} \tag{4.4}$$

Solutions (to the check bit assignment problem) which are derived from matrices of this form have been observed for n=2..6. All solutions for n=2,3 have been derived and they take on the above form, except for an additional solution for n=2. For n=2, the check bit assignment of form: $a_1^i = z_1^i \oplus z_2^i$ and $a_2^i = z_1^i \vee z_2^i$ also satisfies the even weight intersection criterion.

The fact that for $N = 2^n$ there are multiple solutions to the check bit assignment problem is overshadowed by the result that all such solutions require n check bits. This analysis provides no significant insight into the form or size of solutions for problems where $N < 2^n$. It would be nice to be able a) to determine minimal solutions for given PLA's, and b) to determine maximal sized sets of n-bit conflict-free patterns (so that they may be used for state assignment), which is a problem in the realm of mathematical coding theory.

4.1.5.2 Multiple Fault Coverage

Although the fault model adopted in this dissertation restricts itself to single faults, it is interesting to consider the effectiveness of the checking scheme in the presence of multiple faults. The simple parity check scheme is well known as being able to detect odd weight error patterns. Line breaks in multiple product term lines are independent of each other. Faults which cause up to k product terms† to be selected are of interest here. If k=2 then an error of type ξ arises. For k>2, in order to detect errors resulting from t product terms being selected, $2 \le t \le k$, it is necessary that the weight of the intersection of any t codewords be congruent to t modulo 2. Such criteria are difficult to satisfy. For the case where check bits simply duplicate the normal outputs, it is obvious that the weight of the intersection of t codewords is always even. Thus such a scheme is capable of detecting this type of multiple fault only for those cases where t is even.

† Specifically k product terms associated with distinct output patterns.

4.1.6 A Variation

It was shown in section 4.1.1.1 that all errors due to single faults of type I.8 can be detected if the two rightmost PLA outputs are encoded as a 1-out-of-2 code. Errors of type ξ which happen to be detectable by this code, and which correspond to conflict graph edges, may be treated as "don't care" edges during conflict graph decomposition. A smaller sized decomposition may result. As discussed in section 4.1.1.1, any PLA output and a single extra output line, defined to be its complement, could be used to form this 1-out-of-2 code. However, if two new output lines for the 1-out-of-2 code bits are used, then it may be possible to find some particular assignment (of the patterns 01 and 10 to codewords) which results in an even better conflict graph decomposition.

Consider a conflict graph vertex set V partitioned as disjoint sets V_{01} and V_{10} based on which 1-out-of-2 check bit pattern is assigned to each vertex's corresponding codeword. Any error of type ξ involving one codeword from V_{01} and one from V_{10}, will result in a noncodeword value (*i.e.* 11) on the 1-out-of-2 check outputs. The set of conflict graph edges between V_{01} and V_{10}, $\widehat{E}$, represents errors that are detectable by the 1-out-of-2 encoded check bits, and these edges may be ignored by Algorithm 4.1 (**EDG**).

Rather than redesigning **EDG** to account for "don't care" edges, consider two different ways of applying **EDG** to this type of conflict graph decomposition problem. One is to slightly modify **EDG** so that edges $\widehat{E}$ are removed from G before each iteration (*i.e.* between lines 1 and 2, and after line 5). It is clear from Figure 4.7 that the heuristic Algorithm 4.2, which is unable to select cliques which span both subgraphs, cannot find the minimal solution. The white vertices represent V_{01} and the black ones V_{10}.

a) normal 2 clique decomposition b) single clique solution

Figure 4.7 Simple example of partitioned conflict graph decomposition.

Another approach is to use EDG to solve the conflict graph decomposition problem separately on the two subgraphs based on V_{01} and V_{10}, *i.e.* $G_{01}=(V_{01},E_{01})$ and $G_{10}=(V_{10},E_{10})$, respectively. $E_{01(10)}$ consists of those edges from E which have both endpoints in $V_{01(10)}$. Note that $\widehat{E}$, E_{01} and E_{10} form a partition of E. Check bits for each of the subproblems can share PLA output lines; the number of check bits required is the maximum of the number found for the two subproblems. This variation therefore benefits both from dealing with smaller sized graphs and from check bit sharing. Figure 4.8 represents the partitioned graph for the "fsm2b" PLA. It is clear by inspection that 2 check bits are required (*cf.* 6 reported in Table 4.1).

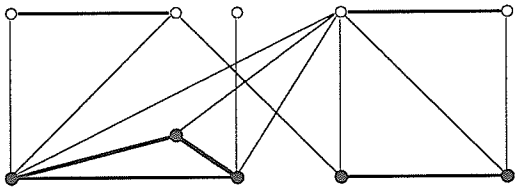


Figure 4.8 Partitioned conflict graph for "fsm2b".

The problem of partitioning V into V_{01} and V_{10} in an auspicious manner still needs to be addressed. Ideally, when solving the two subgraph decomposition problems separately, the partition should minimize expression 4.5:

$$max(\,|\mathbf{EDG}(G_{01})|,\,|\mathbf{EDG}(G_{10})|\,) \tag{4.5}$$

One potential partitioning criterion is to minimize the number of edges in E_{01} and E_{10} (*i.e.* maximize $|\widehat{E}|$). This is similar to the SIMPLE MAX CUT problem which asks if V can be partitioned such that there are at least K edges with one endpoint in V_{01} and one in V_{10}. This problem is known to be NP-complete [Gare79, p. 210]. Meeting this criterion, however, will not necessarily minimize expression 4.5. For example, the partition shown in Figure 4.9 has fewer edges in E_{01} and E_{10} than that of Figure 4.8, but one more check bit is required.

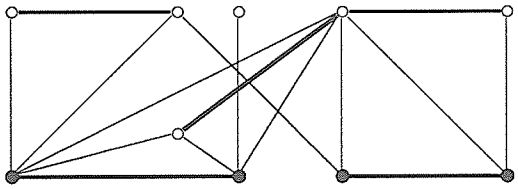


Figure 4.9 Another partitioning of conflict graph for "fsm2b".

It is not clear how one should proceed to solve the partitioning problem. Several small conflict graphs (including that of Figure 4.8) have been partitioned manually in hope of gaining insight into an algorithmic procedure. The manual attempts at partitioning tended to proceed as described by Algorithm 4.3.

Algorithm PG(G):

- $V_{01} \leftarrow V_{10} \leftarrow \{\}$ (1)
- $\widehat{V} \leftarrow V$ (2)
- **while** $\widehat{V} \neq \varnothing$ (3)
  - $T \leftarrow$ {a vertex with maximum degree from $\widehat{V}$} (4)
  - $\alpha \leftarrow 01$ (5)
  - **while** $T \neq \varnothing$ (6)
    - $V_\alpha \leftarrow V_\alpha \cup T$ (7)
    - $\widehat{V} \leftarrow \widehat{V} - T$ (8)
    - $T \leftarrow \{V_j \in \widehat{V} :\ V_j$ is adjacent to V_i for $V_i \in T : \}$ (9)
    - $\alpha \leftarrow \overline{\alpha}$ (10)
- **for all** $V_i \in V$ (11)
  - change the partition V_i is assigned to (12)
  - **if** the change results in a better solution (13)
    - keep it (14)
  - **else** (15)
    - change V_i back to its original partition (16)
- **return** V_{01}, V_{10} (17)

Algorithm 4.3 Partition conflict graph.

The first part of Algorithm 4.3 (the nested while loops) straightforwardly constructs an initial partition in a greedy fashion with respect to the size of $\widehat{E}$. Alternating between V_{01} and V_{10}, any unassigned vertices adjacent to $V_{01(10)}$ are assigned to $V_{10(01)}$. In the second part (the for loop), a pass over all vertices sees if the reassignment of any vertex improves the partition.

Table 4.2 presents a comparison of the conflict graph decomposition of Algorithm 4.2 with that obtained after partitioning. Bear in mind that the former requires one additional check bit to form the 1-out-of-2 subcode, while the latter requires 2 check bits for the same. These few results suggest that conflict graph partitioning is a worthwhile endeavour to pursue, reducing check bits by almost one-half in some instances.

PLA	$\lvert S\rvert_{\text{Alg 4.2}}$	$\lvert S\rvert_{\text{partitioned}}$
fsm1	6	3
fsm2a	5	3
fsm2b	6	2
fsm2c	8	5
router	7	4
traffic	6	1

Table 4.2 Decomposition of some partitioned conflict graphs.

4.2 NONCONCURRENCY

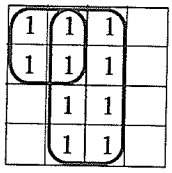
The first step in making a PLA nonconcurrent is to represent it as a functional array, *i.e.* partition the input space into N disjoint cube arrays†, C_{Z^i}, such that $F(x)=Z^i$ for all minterms $x \in C_{Z^i}$, where $Z^i \in \mathcal{Z}$, and $\mathcal{Z}=\{Z^1, Z^2, \ldots, Z^N\}$ denotes the N distinct output patterns produced by the PLA. The problem naturally divides into N independent subproblems, one for each C_{Z^i}. The C_{Z^i} arrays can be treated as specifications for single output combinational functions. The next step is to make each C_{Z^i} 1-concurrent.

The result of these steps, strictly speaking, does not represent a nonconcurrent PLA,

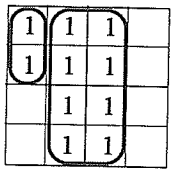
† If a truth table is available, then the input space may be more easily partitioned into sets of minterms.

since false minterms (corresponding to output pattern **0**) are excluded, and therefore a complete cover is not achieved. However, for on-line checking strategies, an all 0 encoding is not allowed.

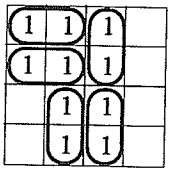
If the original PLA specification includes don't care outputs, then these outputs must be assigned values before the set $\mathcal{Z}$ can be determined. The particular assignment chosen affects the achievement of both nonconcurrency and testability goals, but finding an optimal assignment is an ill-defined, and therefore difficult, problem. For this dissertation, such PLA's are first minimized, thus rendering completely specified outputs.



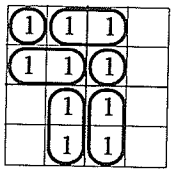
a) example C_{Z^i}



b) minimal solution



c) possible [Wang79] solution



d) worse [Wang79] solution

Figure 4.10 Possible poor solutions resulting from technique of [Wang79].

The only published method for generating a nonconcurrent PLA specification is that of Wang and Avizienis [Wang79] (*cf.* section 2.3.4.1). Their method involves the introduction of extra outputs in conjunction with splitting cubes to reduce concurrency. These extra outputs are not necessary to achieve nonconcurrency. Their method to eliminate concurrency is as follows. Suppose cubes $C^j, C^k \in C_{Z^i}$ and $C^j \cap C^k = I$, $I \neq \emptyset$. Arbitrarily select minterm m_s from I. Arbitrarily select minterm m^* from $adj(m_s) \cap C_{Z^i}$, where $adj(c)$ represents the *adjacency set* of c, which is the set of minterms adjacent to cube (or minterm) c. Minterms m_s and m^* combine to form a 1-

cube which becomes part of the solution. This procedure is repeated on $C_{Z^i} \# m_s \# m^*$ until all concurrencies are eliminated. They suggest no strategy which could be employed to select m_s and m^* so as to achieve a smaller solution size. Furthermore the simple example in Figure 4.10c demonstrates that their procedure can perform rather poorly for a large class of situations, even if the "best" choices for m_s and m^* happen to be made. Figure 4.10d shows what might happen if poor choices for m^* occur, also.

4.2.1 Limited Concurrency

A set of cubes (product terms) is said to have *limited concurrency* if all pairs of cubes with unequal output parts are disjoint (*i.e.* the cubes form a functional array). Concurrency is limited to only product terms with identical output parts. A minimal PLA with limited concurrency may be obtained by using any minimization procedure (*eg.* ESPRESSO) to minimize each of the N subproblems based on C_{Z^i} as described above.

For some on-line checking strategies, the nonconcurrency requirement may be acceptably relaxed to limited concurrency. For the simple parity strategy, where some product terms themselves are treated as pseudo-outputs to ensure odd weight codewords, limited concurrency is applicable only to those Z^i with odd weight. The effect of multiple product terms from C_{Z^i} being selected simultaneously, is similar to an error of type ξ between identically encoded terms, as described in section 4.1.2

4.2.2 Making PLA's 1-Concurrent

Algorithm 4.4 may be used to translate a cubical PLA specification, L, into an equivalent functional array, $\hat{L}$, while simultaneously ensuring that $\hat{L}$ is 1-concurrent. Note that the algorithm is concerned with both the input and output parts of a product term. No two cubes (*i.e.* their input parts) are allowed to intersect, and all minterms belonging to a cube must assert the same output pattern (which is exactly the output part). This algorithm does not attempt to find a minimal solution. Subsequent algorithms take

this functional array and attempt to obtain minimal solutions for the N subproblems defined at the beginning of section 4.2.

A product term in L is denoted $L^i=(C^i, D^i)$, where C^i represents the cube matched by the AND-plane and binary vector D^i represents the outputs asserted by the product term line†. The main premise of the design of the algorithm is that array L contains the product terms yet to be processed, and array $\widehat{L}$ contains the 1-concurrent terms determined thus far. The algorithm is recursive, and is initially invoked NC(L, $\varnothing$). The result is a non-minimal 1-concurrent array; adjacent cubes with identical output parts need to be identified so that they can be merged together to form larger cubes.

Algorithm NC(L, $\widehat{L}$):

- **for all** $L^i \in L$ (1)
  - $T \leftarrow \{\widehat{L}^k \in \widehat{L} : C^i \cap \widehat{C}^k = \varnothing\}$ (2)
  - $\widehat{L} \leftarrow \widehat{L} - T$ (3)
  - **for all** $\widehat{L}^k \in \widehat{L}$ (4)
    - $\widehat{L} \leftarrow \widehat{L} - \widehat{L}^k$ (5)
    - $I \leftarrow C^i \cap \widehat{C}^k$ (6)
    - **if** $D^i \supset \widehat{D}^k$ **or** ($C^i \not\subset \widehat{C}^k$ **and** $D^i = \widehat{D}^k$) **then** (7)
      - $T \leftarrow T \cup \{(\widehat{C}^k \# I, \widehat{D}^k)\}$ (8)
    - **else** (9)
      - **if** $D^i \not\subset \widehat{D}^k$ **then** (10)
        - $T \leftarrow T \cup \{(\widehat{C}^k \# I, \widehat{D}^k)\} \cup (I, D^i \vee \widehat{D}^k)$ (11)
      - **else** $T \leftarrow T \cup \widehat{L}^k$ (12)
      - $t \leftarrow \{(C^i \# I, D^i)\}$ (13)
      - **if** $\widehat{L} = \varnothing$ **or** $t = \varnothing$ **then** (14)
        - $T \leftarrow \widehat{L} \cup T \cup t$ (15)
      - **else** $T \leftarrow T \cup$ NC(t, $\widehat{L}$) (16)
      - $\widehat{L} \leftarrow L^i \leftarrow \varnothing$ (17)
  - $\widehat{L} \leftarrow T \cup L^i$ (18)
- **return** $\widehat{L}$ (19)

Algorithm 4.4 Make PLA 1-concurrent.

† Note that covering relation, $\supset$, is defined differently for cubes and binary vectors (*cf.* section 1.2).

The basic idea of Algorithm 4.4 is to take one term at a time from L and incorporate it into $\widehat{L}$. Term L^i from L is processed against each $\widehat{L}^k$ in $\widehat{L}$. One $\widehat{L}^k$ at a time is removed from $\widehat{L}$, and that part of $\widehat{L}^k$ which is disjoint from L^i is accumulated into T. Once all of $\widehat{L}$ has been considered, T contains the cubes reflecting the complete incorporation of L^i into $\widehat{L}$, *i.e.* T becomes $\widehat{L}$ for then next L^i. During this process all or part of L^i may become absorbed by some $\widehat{L}^k$.

For a given L^i and $\widehat{L}^k$, Figure 4.11 enumerates the intersection situations which may arise. Line 2 of Algorithm 4.4 handles the simple case where the input parts of the two cubes are disjoint; $\widehat{L}^k$ is simply added to T. The remaining cases are those where the input parts of the two cubes have a non-null intersection.

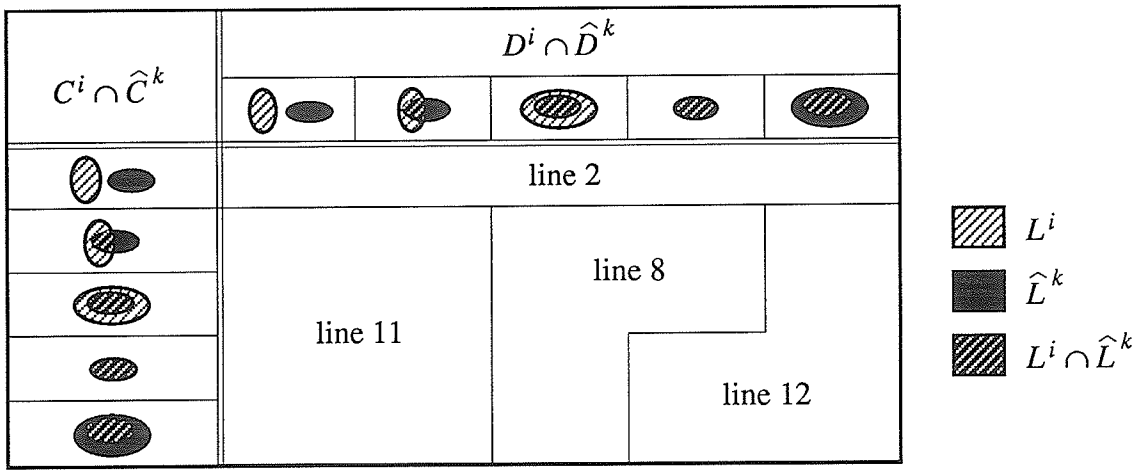


Figure 4.11 Cube intersection situations for Algorithm 4.4.

Line 8 handles the situations where D^i covers $\widehat{D}^k$. In this case L^i absorbs the intersection, I, of the two cubes, and $\widehat{L}^k$ reduced by I is added to T. This action is also performed for the situations where $D^i = \widehat{D}^k$ and C^i is not covered by $\widehat{C}^k$, because some part of L^i would remain anyway. L^i remains intact for the next iteration of the algorithm. This latter case is illustrated in Figure 4.12.

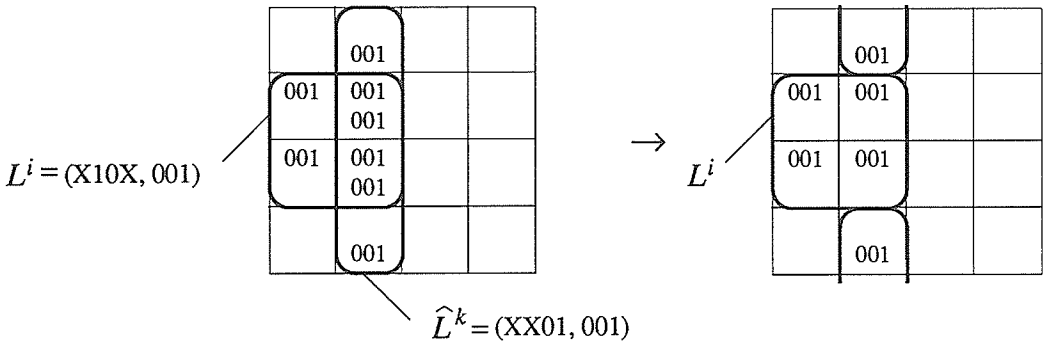


Figure 4.12 Example application of line 8 of Algorithm 4.4.

Line 12 handles the situations where $D^i = \widehat{D}^k$ and C^i is covered by $\widehat{C}^k$. In this case L^i is completely absorbed by $\widehat{L}^k$ (in line 13) which is added to T. The case where $\widehat{D}^k$ covers D^i is also handled by line 12; $\widehat{L}^k$ is retained completely again, but this time when L^i is reduced by I it is not necessarily completely absorbed into $\widehat{L}^k$. This latter case is shown in Figure 4.13.

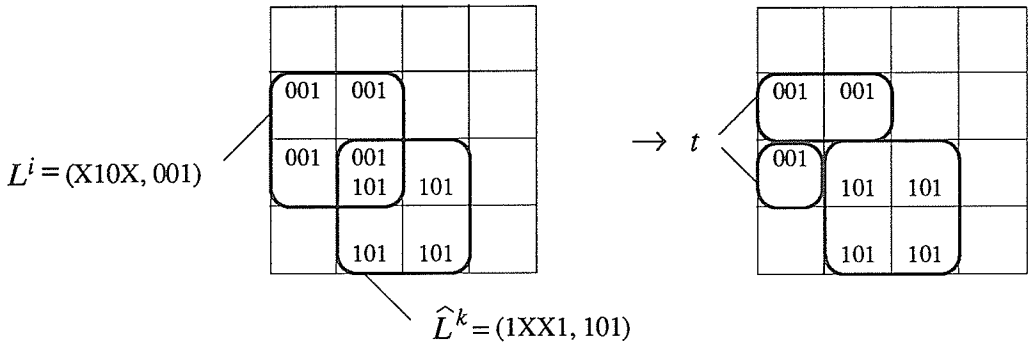


Figure 4.13 Example application of line 12 of Algorithm 4.4.

The case where D^i and $\widehat{D}^k$ are unordered is dealt with in line 11. $\widehat{L}^k$ is split into two parts, that which intersects with L^i and everything else. Both parts are added to T and L^i is reduced by I. See Figure 4.14 for an example of this situation.

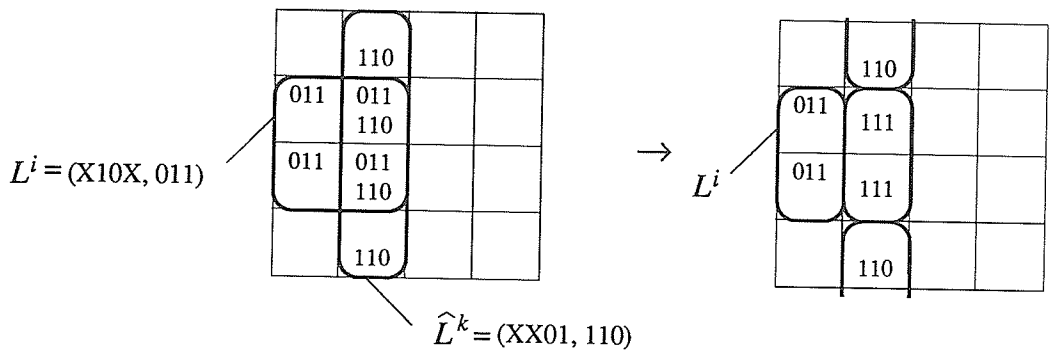


Figure 4.14 Example application of line 11 of Algorithm 4.4.

Whenever L^i is reduced by I in line 13 (for those situations that are indicated in Figure 4.11 as being dealt with by line 11 or 12), the result of the sharp operation, t, may in fact be an array of cubes instead of just a single cube (*eg.* see Figure 4.13). If this is the case, then the algorithm is recursively applied on array t and whatever remains yet to be processed in $\hat{L}$, unless either t or $\hat{L}$ is null, then the recursion is bypassed. For simplicity, the case where t is actually a single term is dealt with recursively, even though L^i could be assigned the value of t and the algorithm allowed to continue.

4.2.2.1 Simple Merge Procedure

Merging to obtain a minimal 1-concurrent realization may be performed. One simple-minded approach sorts cubes by size then, starting at the smallest size, finds adjacent cubes and combines them to obtain a next size cube. A more sophisticated merge algorithm would find all cubes which are adjacent to a given cube, and try to decide which potential merger would be most beneficial. Note that Algorithm 4.4 could be replaced by a simple sort/scan of a truth table. This scan would identify minterms which produce a common output pattern. Merging to form larger cubes would still be done but starting from minterms instead of disjoint cubes.

The method of [Wang79] (section 4.2) would too benefit from merging adjacent

cubes. However, even with merging, their algorithm may produce poor results. For the two cases shown in Figure 4.10c and 4.10d, merging could reduce the solution size from 5 to 3, and from 6 to 5, respectively; the minimal solution requires 2 terms.

4.2.2.2 Single Output 1-Concurrent Minimization

Algorithm 4.5 performs single output boolean function minimization under the constraint that the solution be 1-concurrent. It starts from a minterm specification of the function. It uses a heuristic based on counting adjacent minterms. For large functions this may tend to be as expensive as working with truth tables. The algorithm is given a list of minterms corresponding to the *ON*-set of a single output boolean function.

Algorithm Minimize-1-Concurrent(m)

- $C \leftarrow \{\}$ (1)
- $A \leftarrow$ adjacency counts for m (2)
- **repeat until** $m = \varnothing$ (3)
  - $\hat{c} \leftarrow$ **SelectPrimeCube**(m, A) (4)
  - $m \leftarrow m - \hat{c}$ (5)
  - update A to reflect removal of $\hat{c}$ from m (6)
  - $C \leftarrow C \cup \hat{c}$ (7)
- **return** C (8)

Algorithm 4.5 Minimization constrained to 1-concurrency.

Heuristics based on the following criteria guide the cube selection function, **SelectPrimeCube**, used in line 4 of Algorithm 4.5.

Criterion 1: The largest cubes available should be included because they cover many minterms and thus fewer terms may be required. Once a cube is adopted as part of the solution, some of the other original prime cubes may no longer be candidates. During conventional minimization, covered minterms may be treated as don't care terms, but here once a minterm is covered, it must be considered to be in the *OFF*-set. This is why

the term "large" was used above instead of "prime". If prime cubes are recomputed each time a term is added to the solution, then it would be acceptable to say the "largest of the prime cubes" above. The term prime will be used in the following with the assumption that primality is relative to the current stage of minimization.

Criterion 2: If a collection of disjoint essential primes covers most of some other prime, then that other prime should not be included in the solution even if it is large.

Criterion 3: Suppose the largest prime is a k-cube. Each minterm belonging to that cube will be adjacent to at least k other minterms. Conversely, if a k-cube exists, then it is likely that a minterm with large adjacency count will belong to it. If a minterm has high adjacency count but is not part of the k-cube, then it must be the focal point of several smaller primes. It is desirable to avoid generation of all prime cubes each step of the algorithm. There may be a very large number of them, and most of them will be ignored in this iteration, and many will be invalid subsequently. The ideal situation is to determine a list of the largest primes only. Adjacency counts may be useful in locating these. One approach might be to generate all primes which contain minterms with the highest adjacency count, A_{max}.

Criterion 4: In the case where two or more k-cubes satisfy the above criteria, some basis to choose one over the others is required. If some of the minterms belonging to a k-cube have adjacency counts equalling k, then that k-cube is the only prime covering those minterms (*i.e.* it is essential when doing conventional minimization). If the k-cube is not selected to be part of the solution, then the minterms which had counts of k can only be covered by subcubes of the original k-cube. Thus the k-cube with the most minterms with counts equal to k should be selected.

Criterion 5: If all counts associated with a k-cube exceed k, then it seems as though all minterms belonging to the cube are adjacent to some minterms outside the cube, and hence they may all be covered as a natural consequence of covering the adjacent

minterms. Thus such a k-cube should probably not be selected.

Criterion 6: Considering the situation of Criterion 4, two or more k-cubes may have identical numbers of minterms with counts equalling k. In this case, there are several possible criteria which can be used to break the tie. One is to select the k-cube with the fewest minterms having counts equal to A_{max}. Another is to sum the counts for each candidate cube and choose the one with the smallest total. A third is to sort the counts for each candidate cube into ascending order and select the cube with sorted count sequence which is lexicographically less than the others. The common goal of all these methods is to leave as much "adjacency" as possible after the selected cube is removed so that there may be a rich selection of large primes in the next iteration. Alternatively it is desired to avoid creating isolated minterms with no adjacencies. For each minterm with count $k+r$ in the selected k-cube, there are r unselected minterms which will have their adjacency counts reduced by 1. A k-cube with smaller total will thus have a smaller impact on the remaining minterms. The argument for using lexicographic order of counts is that minterms with small counts can be covered in only a few different ways. Thus the more of these included in the selected k-cube, the better.

Criterion 7: Related to Criterion 6, if two or more k-cubes have identical count distributions, then additional aspects must be considered. All candidate cubes have the same size set of unselected adjacent minterms (adjacency set). Consider the sum of adjacency counts for each set, and select the k-cube whose adjacency set has the highest total. The rationale is that the adjacency set with the highest total will have the most adjacency left after the selected k-cube is removed.

Algorithm 4.6, as sketched below, represents one design for the **SelectPrimeCube** function which is based on some of the above criteria. Let $A_i = |adj(m_i)|$ denote the adjacency count for minterm m_i. Let $\Sigma(c)$ denote the sum of adjacency counts for minterms in cube (or cube array) c. Let $primes(m_i)$ denote the set of prime cubes which contain minterm m_i.

Algorithm SelectPrimeCube(m, A)

- $A_{max} \leftarrow max(A)$ (1)
- $\mathcal{P} \leftarrow \{ primes(m_i) : A_i = A_{max} \}$ (2)
- $k \leftarrow \max_{P_j \in \mathcal{P}} size(P_j)$ (3)
- $\mathcal{P} \leftarrow \{ P_j \in \mathcal{P} : size(P_j) = k \}$ (4)
- **if** $|\mathcal{P}| > 1$ **then** (5)
  - $\mathcal{P} \leftarrow \{ P_j \in \mathcal{P} : |\{m_i \in P_j : A_i = k\}|$ is maximal $\}$ (6)
  - **if** $|\mathcal{P}| > 1$ **then** (7)
    - $\mathcal{P} \leftarrow \{ P_j \in \mathcal{P} : \sum(P_j)$ is minimal $\}$ (8)
    - **if** $|\mathcal{P}| > 1$ **then** (9)
      - $\mathcal{P} \leftarrow \{ P_j \in \mathcal{P} : \sum(adj(P_j))$ is maximal $\}$ (10)
- **return** any $P_j \in \mathcal{P}$ (11)

Algorithm 4.6 Heuristic to select a cube for Algorithm 4.5.

An algorithm which works with cubes instead of minterms would be preferable.

4.2.2.3 Experimental Results

Table 4.3 presents some results comparing the number of product terms (and concurrency) obtained by four minimization strategies. The column "Normal Minimization" represents the minimized PLA obtained by using version 2.1 ESPRESSO [Bray84] in the usual way. The columns labelled "1-Concurrent" present the results of two different methods of making the PLA's 1-concurrent. Both begin by using Algorithm 4.4 to make the PLA 1-concurrent. The first, "Simple Merge", indicates the solution size after merging adjacent cubes with identical output patterns (as described in section 4.2.2.1), and the second lists the results obtained by using Algorithm 4.5 on the N subproblems. For the column "Limited Concurrency", as defined in section 4.2.1, p represents a lower bound on the number of product terms to achieve 1-concurrency, k

represents the concurrency of such a realization, and p_{OFF} indicates the minimal number of terms required to cover the *OFF*-set.

Cube intersection is a compatibility relation, and k-concurrency may be determined as the size of the largest maximal compatible for the relation.

				Normal Minimization		1-Concurrent		Limited Concurrency		
						Simple				
PLA	m	n	N	p	k	Merge	Alg. 4.5	p	k	p_{OFF}
alu1	12	8	80	19	12	1410	1032	953	12	3
alu2	10	8	67	68	36	380	361	361	8	0
apla	10	12	37	25	4	73	71	71	4	29
bis1	4	3	5	5	2	7	7	7	2	1
bis2	4	3	4	5	2	6	6	6	2	2
bis5	4	4	8	10	3	10	10	10	1	1
bis6	4	3	5	6	3	7	7	7	2	3
bw	5	28	22	22	2	22	22	22	1	2
dc1	4	7	10	9	3	10	10	10	1	2
dc2	8	7	91	39	6	129	129	129	4	9
dist	8	5	21	120	15	166	164	164	2	1
dk17	10	11	24	18	3	53	49	49	7	25
dk27	9	9	38	10	5	41	41	41	3	6
eg	3	5	6	6	3	6	6	6	1	0
f2	4	4	10	8	2	10	10	10	1	5
fsm1	10	6	13	15	4	18	17	17	2	10
fsm2a	6	6	14	18	5	24	24	24	2	2
fsm2b	5	7	11	16	2	16	16	16	1	6
fsm2c	8	7	24	31	5	33	33	33	2	9
gary	15	11	69	107	9	134	133	133	5	14
misex1	8	7	10	12	3	13	13	13	2	5
misex2	25	18	34	28	5	591	472	471	216	1
rd53	5	3	5	31	6	31	31	31	1	1
router	11	12	15	22	4	23	23	23	2	3
sao2	10	4	9	58	10	73	64	58	9	2
traffic	5	7	8	9	3	10	10	10	2	0
vg2	25	8	23	110	18	2156	1921	1584	48	12
wim	4	7	11	9	4	13	13	13	2	0
z4	7	4	15	59	26	127	127	127	1	1

Table 4.3 Number of product terms (and k-concurrency) obtained by various minimization strategies.

It appears that certain PLA's (such as "alu1", "alu2", "misex2" and "vg2") are inherently strongly concurrent, and have no feasible 1-concurrent realization.

4.3 SUMMARY

This chapter showed how to augment a nonconcurrent PLA with extra outputs so that a simple parity check can detect all errors due to Class I, II and III faults, except for fault type I.8. The output patterns must belong to an odd weight unordered code where all pairs of codewords satisfy the even weight intersection criterion. Extending output patterns to construct such a code is facilitated by an edgewise exclusive-OR sum of cliques decomposition of the conflict graph representing pairs of patterns which originally have odd weight intersections. A heuristic algorithm for performing such a decomposition was described, and experimental results giving the number of extra output bits required to encode a number of example PLA's was reported. Generally, a PLA with n outputs requires about n check bits.

A variation of this testing strategy was suggested. It takes advantage of checking circuitry needed to detect errors due to type I.8 faults. The two rightmost PLA outputs are designed to only produce patterns from the 1-out-of-2 code. Errors of type ξ which happen to be detectable by the 1-out-of-2 code need not be detectable by parity check. Considerably fewer check bits are required as a result of a partitioned conflict graph.

The case where an unencoded PLA produces all $N=2^n$ n-bit output patterns was analyzed. It was found that exactly n check bits are required.

The concept of limited concurrency was introduced. But, if product term lines are used as pseudo-outputs to ensure each codeword has odd weight, then it is only applicable to product terms whose actual output parts have odd weight.

An algorithm to transform the cubical specification of a normal PLA into a 1-concurrent array, was presented. Another algorithm which performs 1-concurrent minimization of single output functions was described. Experimental results were shown which suggest that there exist PLA's whose 1-concurrent (or nonconcurrent) realization inherently requires a very large increase in number of product terms as compared to conventional minimization.

Chapter 5

OR-*k* TESTING

Another method for detecting errors of type ξ in nonconcurrent PLA's is presented in this chapter. Single bit errors and those due to a fault of type I.8 are covered by a 1-out-of-2 encoded parity scheme as described in section 4.1.1.1 (*cf.* Figure 5.7a, also). Portions of this chapter have appeared as [Marc89b] and [Marc90a].

5.1 OR-*k* TESTABILITY

5.1.1 Definition

Consider an m-input, n-output nonconcurrent PLA realizing some combinational functions using p product terms. Suppose the PLA normally produces N distinct output patterns $\mathcal{Z}=\{Z^1, Z^2, \ldots, Z^N\}$, where the bits of pattern Z^i are denoted $Z^i_1, Z^i_2, \ldots, Z^i_n$. The notation $\xi^{i,j}$ will be used to represent both an error of type ξ involving Z^i and Z^j, and the output pattern produced by the error. Let $\Xi = \{\xi^{i,j} = Z^i \vee Z^j, i \neq j\}$, denote the set of output patterns produced by all errors of type ξ. Define $\widetilde{E} = \Xi - \mathcal{Z}$, to be the subset of erroneous output patterns not produced by an error-free PLA.

Consider a selection of k of the PLA's n outputs, $z_\Psi=(z_{\psi_1}, z_{\psi_2}, \ldots, z_{\psi_k})$, where $\Psi=(\psi_1, \psi_2, \ldots, \psi_k)$ is an ordered subset of $\{1, 2, \ldots, n\}$. If there are no Z^i for which all these outputs are 1's, then whenever we observe them to be all 1's, we know that an error is present.

Definition 5.1 The product term, $z_{\psi_1} z_{\psi_2} \ldots z_{\psi_k}$, is a *viable OR-k test term*, (or simply $\Psi=(\psi_1, \psi_2, \ldots, \psi_k)$ is a *viable test*) if $Z^i_\Psi \neq 11..1$ is true for all $i=1..N$.

To detect an error $\xi^{i,j} \in \widetilde{E}$ where $\xi_{\Psi}^{i,j} = 11..1$, one need only observe that the k output positions identified by Ψ are all 1's. A viable test is conceptually realized as a k-input AND gate. Several viable tests may be required to detect all expected errors, and they may be OR'ed together to produce a single error indicator. A checker thus consists of a two level AND/OR array with each k-input AND implementing a viable test. This online checking strategy is called *OR-k testing* because certain k-bit subsets of the n output bits are used such that when all k bits are 1 this implies that an error has been detected which was caused by the bitwise OR'ing of two or more product terms, *i.e.* an error of type ξ. If the set of naturally occurring viable tests is insufficient to cover all errors, which will certainly be the case if any $\xi^{i,j} \in \mathcal{Z}$, then it is necessary to augment the PLA with extra check bit outputs which ensure the testability of the PLA. Determination of these check bits will be considered in the next subsection and again in section 5.2.

5.1.2 Uniform *k*

This section presents a method for determining check bits for an OR-k strategy where all viable tests are uniformly the same size. Specifically k=3, or an OR-3 strategy will be developed, however the extension to general k is straightforward.

Formally define $\xi^{i,j}$ and $\eta^{i,j}$ by equations 5.1a and 5.1b:

$$\xi^{i,j} = Z^i \vee Z^j \tag{5.1a}$$

$$\eta^{i,j} = Z^i \wedge Z^j \tag{5.1b}$$

Definition 5.2 Errors $\xi^{i,j}$ and $\xi^{u,v}$ are *equivalent* if $\xi^{i,j} = \xi^{u,v}$ and $\eta^{i,j} = \eta^{u,v}$.

Error $\xi^{i,j}$ is said to dominate error $\xi^{u,v}$, $\xi^{i,j} \geq \xi^{u,v}$, if all tests for $\xi^{u,v}$ also test $\xi^{i,j}$. This relation is formally stated in Definition 5.3 in terms of the usual covering relations on binary vectors. For example, $\xi^{i,j} \geq \xi^{u,v}$, given:

$$Z^i = 110\ 0110 \qquad \xi^{i,j} = 110\ 1111$$
$$Z^j = 100\ 1101 \qquad \eta^{i,j} = 100\ 0100$$
$$Z^u = 000\ 1101 \qquad \xi^{u,v} = 000\ 1111$$
$$Z^v = 000\ 0111 \qquad \eta^{u,v} = 000\ 0101$$

The tests for $\xi^{i,j}$ are (1,4,6), (2,4,6), (4,5,6) and (4,6,7), while those for $\xi^{u,v}$ are (4,5,6) and (4,6,7), (assuming 110 0111 and 110 1101 are also in $\mathcal{Z}$).

Definition 5.3 Error $\xi^{i,j}$ *dominates* $\xi^{u,v}$ if $\xi^{i,j} \supseteq \xi^{u,v}$ and $\eta^{i,j} \wedge \xi^{u,v} \subseteq \eta^{u,v}$.

Algorithm 5.1 augments a PLA with extra check bit outputs to ensure OR-3 testability. A set of viable tests, T, is also determined (*cf.* section 5.1.3 for an example).

Algorithm OR-3($\mathcal{Z}$)

- $E \leftarrow \{(\xi^{i,j},\eta^{i,j})\colon\ i{=}1..N, j{=}1..N, i{\neq}j\}$ (1)
- $V \leftarrow \{(\alpha,\beta,\gamma)\colon\ \alpha,\beta,\gamma{\in}\{1..n\}, \alpha{<}\beta{<}\gamma, Z^i_{\alpha,\beta,\gamma}{\neq}111\ \forall\ i\}$ (2)
- $T \leftarrow \{\}$ (3)
- **repeat until** $E = \varnothing$ (4)
  - $\tilde{E} \leftarrow \{\xi^{i,j}\colon\ (\xi^{i,j},\eta^{i,j}){\in}E, \xi^{i,j}{\notin}\mathcal{Z}\}$ (5)
  - $I \leftarrow$ **TableI**($\tilde{E}$, V) (6)
  - $T \leftarrow T \cup$ **MinCover**(I) (7)
  - $E \leftarrow E - \{(\xi^{i,j},\eta^{i,j}){\in}E\colon\ \xi^{i,j}{\notin}$ **ZeroColumn**(I)$\}$ (8)
  - **if** $E \neq \varnothing$ **then** (9)
    - $II \leftarrow$ **TableII**(**RemoveDominating**(E)) (10)
    - $\alpha,\beta \leftarrow$ **SelectRow**(II) (11)
    - $E \leftarrow E -$ {errors covered by row α,β} (12)
    - $\gamma \leftarrow n \leftarrow n{+}1$ (13)
    - update $\mathcal{Z}$ and E to reflect $z_\gamma{=}\overline{z_\alpha \cdot z_\beta}$ (14)
  - **if** $E \neq \varnothing$ **then** (15)
    - $V \leftarrow \{(\alpha,\beta,\gamma)\colon\ \alpha,\beta{\in}\{1..n{-}1\}, \alpha{<}\beta, Z^i_{\alpha,\beta,\gamma}{\neq}111\ \forall\ i\}$ (16)
- **return** $\mathcal{Z}$, T (17)

Algorithm 5.1 Ensure OR-3 testability.

Note that E as defined in line 1 contains exactly one representative of each class of equivalent errors. The set $\tilde{E}$, which represents those erroneous output patterns not in $\mathcal{Z}$, and the set of viable tests, V, are used to construct **TableI**. **TableI** is a binary matrix with rows representing viable tests and columns for each output pattern in $\tilde{E}$. A **TableI** entry is 1 if the corresponding viable test actually detects the associated erroneous output pattern. The function **MinCover**, used in line 7, identifies a minimal set of viable tests which detect all detectable errors in **TableI**. Undetectable errors are those with a zero column in **TableI** and these are identified by function **ZeroColumn**. Line 8 removes covered errors from the list of errors still to consider, E.

If undetectable errors remain, due either to the previous condition or because some error patterns are in $\mathcal{Z}$, then check bits must be defined. Any pair of existing output bit positions, α and β, can be used to determine the value for the new output, z_γ, such that α,β,γ is a viable test. The only restrictions are that if $Z^i_{\alpha,\beta}=11$ then Z^i_γ must be 0, and that Z^j_γ must be 1 for some j. The rule $z_\gamma=\overline{z_\alpha \cdot z_\beta}$ satisfies these constraints. **TableII** (line 10) specifies the coverage of each new test based on α and β versus the non-dominating errors remaining. Test α,β,γ detects error $\xi^{i,j}$ if $\xi^{i,j}_{\alpha,\beta}=11$ and $\eta^{i,j}_{\alpha,\beta}\neq 11$. A minimal cover of **TableII** could be used to define all the check bits and associated tests, but by defining one check bit each iteration, the algorithm may capitalize on any other viable tests involving z_γ which may arise. A heuristic is used in line 11 to select an α,β pair which tests the most remaining errors.

Unfortunately, dominating errors which are not tested by the addition of z_γ, have to be reconsidered in subsequent iterations because the addition of a check bit output can invalidate the dominance relation of untested errors. Consider two errors, $\xi^{i,j}$ and $\xi^{u,v}$, where $\xi^{i,j}\geq\xi^{u,v}$ before z_γ is defined, and neither is tested by α,β,γ. Suppose $Z^i_{\alpha,\beta,\gamma}=Z^j_{\alpha,\beta,\gamma}=110$ while $Z^u_{\alpha,\beta}=Z^v_{\alpha,\beta}\neq 11$ and $Z^u_\gamma=Z^v_\gamma=1$; $\xi^{i,j}$ no longer dominates $\xi^{u,v}$ because $\xi^{i,j}_\gamma=0$ while $\xi^{u,v}_\gamma=1$.

The set of test terms, T, which is obtained by Algorithm 5.1 may contain redundant tests, because errors detected by tests determined in one iteration may also be covered by tests found in subsequent iterations. A second invocation of Algorithm 5.1 on the augmented set of outputs, $\mathcal{Z}'$, obtained by the first invocation of Algorithm 5.1, could be used to find a solution requiring fewer tests (which is minimal for $\mathcal{Z}'$, but not necessarily for $\mathcal{Z}$). Lines 9 to 16 of the algorithm do not apply as $\mathcal{Z}'$ is known to be unordered and OR-3 testable.

5.1.3 An Example

In this section, Algorithm 5.1 will be demonstrated on PLA "eg" (*cf.* Appendix A). For this PLA, $\mathcal{Z}$={5, 7, 11,14, 20, 29}, or in binary:

$$Z^1 = 00101 \qquad Z^4 = 01110$$
$$Z^2 = 00111 \qquad Z^5 = 10100$$
$$Z^3 = 01011 \qquad Z^6 = 11101$$

There are $\binom{6}{2}$ or 15 error events to consider, no two which are equivalent. The set, E={ $E^{i,j}$= ($\xi^{i,j}$,$\eta^{i,j}$)}, obtained by line 1 is†:

$E^{1,2}$= (00111, 00101) $E^{1,3}$= (01111, 00001) $E^{4,5}$= (11110, 00100)
$E^{3,4}$= (01111, 01010) $E^{1,5}$= (10101, 00100) $E^{4,6}$= (11111, 01100)
$E^{2,4}$= (01111, 00110) $E^{2,5}$= (10111, 00100) $E^{3,6}$= (11111, 01001)
$E^{1,4}$= (01111, 00100) $E^{5,6}$= (11101, 10100) $E^{2,6}$= (11111, 00101)
$E^{2,3}$= (01111, 00011) $E^{1,6}$= (11101, 00101) $E^{3,5}$= (11111, 00000)

Only 3 of the $\binom{5}{3}$ or 10 possible tests are viable (set V on line 2 of Algorithm 5.1). The set $\widetilde{E}$ (line 5) represents 12 of the 15 error events, *i.e.* all $\xi^{i,j}$ except for three errors: $\xi^{1,2}$, $\xi^{1,6}$, $\xi^{5,6} \in \mathcal{Z}$. These 12 error events produce 5 distinct erroneous output patterns. This yields **TableI** (line 6) as shown in Figure 5.1. (Note that (1,4) is in fact a viable OR-2 test.)

---

† This list is presented in an order which is useful when determining the domination relation.

		$\tilde{E}$				
		01111	10101	10111	11110	11111
V	(1,2,4)	0	0	0	1	1
	(1,3,4)	0	0	1	1	1
	(1,4,5)	0	0	1	0	1

Figure 5.1 TableI — iteration 1.

Test (1,3,4) covers all nonzero columns of **TableI** (line 7), testing 3 output patterns which correspond to 6 of the 12 error events represented. This leaves 9 error events to consider, 6 uncovered in **TableI**, and 3 with output patterns contained in $\mathcal{Z}$. Based on these 9 error events, the following dominance relations hold:

$$\xi^{1,3} \geq \xi^{1,2} \qquad \xi^{1,3} \geq \xi^{2,3}$$
$$\xi^{1,4} \geq \xi^{1,2} \qquad \xi^{1,4} \geq \xi^{2,4}$$

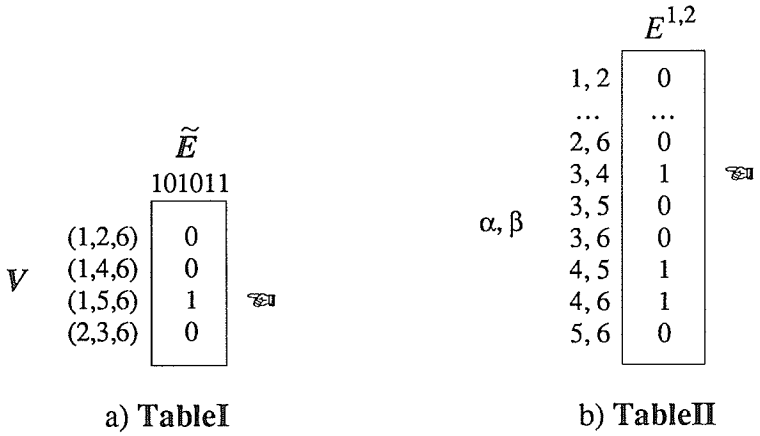
Therefore $E^{1,3}$ and $E^{1,4}$ are excluded from **TableII** (line 10 of Algorithm 5.1) shown in Figure 5.2. All pairs of existing output positions, α,β, may be used to define a new output, at position $\gamma=n+1$, in such a way that (α,β,γ) is a viable test. Figure 5.2 also indicates (on the right) how many error events (+ dominating errors) each potential viable test detects.

		$E^{i,j}$							
		1, 2	3, 4	2, 4	2, 3	1, 5	5, 6	1, 6	
α, β	1, 2						1	1	2
	1, 3					1		1	2
	1, 4								0
	1, 5					1	1	1	3
	2, 3		1	1	1		1	1	5+2
	2, 4			1	1				2+2
	2, 5		1	1	1		1	1	5+2
	3, 4	1	1		1				3+2
	3, 5		1	1	1	1	1		5+2
	4, 5	1	1	1					3+2

Figure 5.2 TableII — iteration 1.

The 2,3 row is selected (in line 11) to define check bit output $z_6=\overline{z_2 \cdot z_3}$ which results in test (2,3,6) covering 5 errors from **TableII** plus both of the dominating errors. At this point a cover of **TableII** could be determined, using say 2,3 3,4 and 1,5, thus defining 3 check bit outputs and 3 tests for a final solution of 3 extra outputs and 4 tests. However, by selecting just one and iterating, a smaller solution is found.

Two error events remain in the enlarged PLA, $\xi^{1,5}$= 101011 and $\xi^{1,2}= Z^2$. A viable test now exists for the former, and an additional check bit, $z_7=\overline{z_3 \cdot z_4}$, is trivially defined for the latter (see Figure 5.3).



a) **TableI** b) **TableII**

Figure 5.3 Iteration 2.

Therefore the PLA is made OR-3 testable by the addition of two check bit outputs. At this point, Algorithm 5.1 returns with 4 tests: (1,3,4), (2,3,6), (1,5,6) and (3,4,7), and the augmented output patterns:

$$Z^1 = 00101\ 11 \qquad Z^4 = 01110\ 00$$
$$Z^2 = 00111\ 10 \qquad Z^5 = 10100\ 11$$
$$Z^3 = 01011\ 11 \qquad Z^6 = 11101\ 01$$

Reapplying Algorithm 5.1 on these patterns, it is discovered that the 15 errors produce 9 distinct patterns, and only 3 of the 9 viable tests are required to cover all of them. In fact, test (1,3,4) of the solution found above is redundant. This example PLA can therefore be made OR-3 testable through the addition of 2 check bits and 3 test terms.

5.1.4 Experimental Results

This section presents the results of applying Algorithm 5.1 on a number of small PLA's which were put into a 1-concurrent form using the techniques of section 4.2.2. For these experiments, the **RemoveDominating** function (line 10 of Algorithm 5.1) returns E unchanged, as in practice it was found that computation of the dominance relation cost more than processing the dominating error events. In all cases, Algorithm 5.1 was applied a second time to determine a minimal set of OR-3 test terms for the augmented PLA.

Table 5.1 shows the number of check bit outputs, n_C, and test terms in T required to make each PLA OR-3 testable. A simple chip area cost estimator is used to determine an indication of the area overhead incurred by the OR-3 checking strategy. Equation 5.2 gives the *cost estimation function* used. The cost of a naked PLA†, or of a 1-concurrent PLA with no check bit outputs, is determined by using the appropriate value of p and setting n_C=T=0. The *overhead* of an augmented PLA relative to some reference PLA is defined by equation 5.3.

$$Cost = (2m + n + n_C)p + (n + n_C)T \tag{5.2}$$

$$Overhead = (Cost_{augmented} \div Cost_{reference} - 1) \times 100\% \tag{5.3}$$

Note that Table 5.1 is intended to reflect the cost of OR-3 testability and as such omits the additional overhead due to two-rail parity check bits and false product terms required to ensure a complete covering.

† A normal minimized PLA, which is not augmented for testability, is referred to as a *naked* PLA.

PLA	m	n	N	OR-3 Testability n_C	OR-3 Testability T	Overhead: Algorithm 5.1 *vs.* Naked	Overhead: Algorithm 5.1 *vs.* 1-Concurrent	Overhead: 1-Concurrency Alone
alu1	12	8	80	10	10	7059%	32%	5332%
alu2	10	8	67	10	8	628	37	431
apla	10	12	37	9	16	306	43	184
bis1	4	3	5	2	2	84	31	40
bis2	4	3	4	2	2	60	33	20
bis5	4	4	8	3	3	43	43	0
bis6	4	3	5	2	2	53	31	17
bw	5	28	22	1	10	37	37	0
dc1	4	7	10	3	3	56	40	11
dc2	8	7	91	12	7	418	57	231
dist	8	5	21	9	8	100	46	37
dk17	10	11	24	11	25	367	72	172
dk27	9	9	38	10	20	603	71	310
eg	3	5	6	2	3	50	50	0
f2	4	4	10	5	4	115	72	25
fsm1	10	6	13	5	6	52	34	13
fsm2a	6	6	14	5	11	108	56	33
fsm2b	5	7	11	6	11	88	88	0
fsm2c	8	7	24	7	8	55	45	6
gary	15	11	69	16	14	81	46	24
misex1	8	7	10	3	3	33	23	8
misex2	25	18	34	8	72	1882	18	1586
rd53	5	3	5	2	3	19	19	0
router	11	12	15	5	11	45	39	5
sao2	10	4	9	5	6	37	24	10
traffic	5	7	8	2	5	54	38	11
vg2	25	8	23	7	7	1859	12	1646
wim	4	7	11	4	4	116	49	44
z4	7	4	15	6	6	193	36	115

Table 5.1 OR-3 testability results.

The attainment of 1-concurrency (and in general nonconcurrency) contributes significantly to total overhead. It is the dominant factor for those PLA's for which the OR-3 scheme is blatantly inappropriate, "alu1", "misex2" and "vg2". For these three PLA's, the overhead of OR-3 relative to 1-concurrent is just 32, 18 and 12 percent respectively.

In general, it appears that high overhead makes OR-3 testing unfeasible.

5.2 OR-k TESTABILITY — FOCUS ON UNORDEREDNESS

From Corollary 3.2 it is known that the set of output patterns must be unordered for all errors of type ξ to be detectable. A possible variation of Algorithm 5.1 would be to first augment the PLA with check bit outputs to ensure unorderedness. However, once $\mathcal{Z}$ is unordered, it can be shown that the PLA is OR-k testable for varying k (*cf.* section 5.2.2); additional check bits are unnecessary.

5.2.1 Unordered Codes

Consider the problem of extending a set of binary vectors, $\mathcal{Z}$, to make it unordered. A minimal number of check bits are appended to each $Z^i \in \mathcal{Z}$. The elements of $\mathcal{Z}$ form a partial order which can be represented by a Hasse diagram ([Prat67, p.57]). For example, the Hasse diagram shown in Figure 5.4 represents $\mathcal{Z}$={3, 8, 9, 10, 12, 18, 30}.

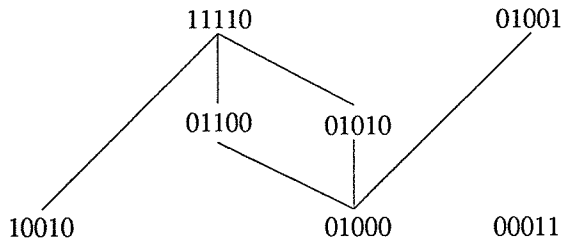


Figure 5.4 Sample Hasse diagram.

A *chain* may be informally defined as any downwards path (*eg.* 11110—01100—01000 and 01010—01000 are two chains, but the latter is not maximal length). Let Z^i represent any element from $\mathcal{Z}$, and Z^j denote any other element which follows Z^i along some chain. Each element on a chain covers all following elements. Alternatively, for any Z^i there exists a position α such that Z^i_α=1 and Z^j_α=0 for all Z^j which are farther down a chain. Also Z^j_β=0 for all positions β where Z^i_β=0. To make a chain unordered, it is necessary to define a new bit position γ such that Z^i_γ=0 and Z^j_γ=1 for each Z^j following Z^i. For

any two binary numbers, $a<b$, there exists a position γ such that $0=a_\gamma < b_\gamma=1$. Therefore, appending any increasing binary number sequence to successive elements along a chain will make it unordered. This is the sole modification required to make $\mathcal{Z}$ unordered. Since any increasing sequence of values will work, so will (0, 1, 2, ...), and thus the minimal number of bits required is $h = \lceil \log_2 L \rceil$, where L is the number of elements in the longest chain. For the partial order shown in Figure 5.4, $L=3$, so 2 bits are required to make it unordered.

Three systematic methods for assigning check bit values to each element of $\mathcal{Z}$ are considered here (for 1 and 2, maximal length chain is implied):

1) assign 2^h-1 to the bottom element of each chain, proceed in sequence upwards,
2) assign 0 to the top element of each chain, proceed in sequence downwards,
3) for each element, in any order, determine what subrange of $[0..2^h-1]$ is admissible and select one value at random.

For method 2 (method 1 is similar), the check bit sequence 0, 1, 2, ... is assigned to elements along each chain except for the case where two or more initial subchains of different length join together and have a common ending subchain. In this situation, the ending subchain continues in sequence from the longest of the initial subchains. For the example of Figure 5.4, three chains join at 01000, and since the longest is of length 2 (and has check bits 0 and 1 assigned to it), node 01000 is assigned check bit value 2.

Smith [Smit84] described a related procedure for making $\mathcal{Z}$ unordered. First $\mathcal{Z}$ is sorted topologically, then the sorted list is divided into partitions of contiguous elements such that each partition is unordered, and each partition is assigned a check bit value. Many topological orderings, and partitions of each, are possible. General design strategies, such as the three unordering methods described above, are awkward to formulate within Smith's frame of reference.

5.2.1.1 Continuing Example

The Hasse diagram shown in Figure 5.5 is that for the ongoing example PLA of section 5.1.3. Clearly only one check bit is required to make this set unordered; the assignment according to method 2 (top down) is also indicated in the figure. When Algorithm 5.1 is applied to this set of extended vectors it is determined to be OR-3 testable using 4 test terms and no other additional output check bits are required. In general, OR-3 testability may require additional check bit outputs. This solution with 1 check bit output and 4 test terms, in most circumstances, is lower cost than the previous best solution of 2 extra outputs and 3 test terms, as found in section 5.1.3.

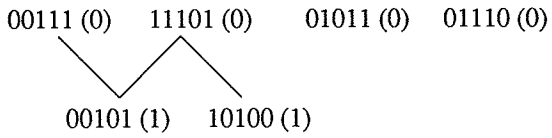


Figure 5.5 Hasse diagram and method 2 unordering assignment (in parentheses) for example PLA of section 5.1.3.

5.2.2 Non-Uniform

In this section, the uniform OR-k checking strategy of section 5.1.2 is extended to include test terms of varying size rather than requiring all test terms to act on the same number of PLA outputs. Theorem 5.1 guarantees that if $\mathcal{Z}$ is unordered then the PLA is OR-k testable for some $k \geq 2$.

Theorem 5.1 Given $\mathcal{Z}$ is unordered, for any error $\xi^{i,j}$ there exists a viable test of size $k \geq 2$ which detects it.

Proof: From equation 5.1a $\xi^{i,j} = Z^i \vee Z^j$, where $Z^i, Z^j \in \mathcal{Z}$. Let Ψ represent the positions of the 1's in $\xi^{i,j}$. No element of $\mathcal{Z}$ has 1's in all these positions because if one did then it would cover both Z^i and Z^j. Therefore Ψ is a viable test for $\xi^{i,j}$. □

A minimal set of test terms may be synthesized in a relatively straightforward manner by using normal minimization methods to minimize the checker function, $G(Z)$, defined by equation 5.4. If the PLA output, Z, is an element of $\mathcal{Z}$, then the checker should indicate no error (0). If the output is an element of Ξ, then the checker should indicate an error (1). All other output patterns may be treated as don't care minterms of $G(Z)$. Note that if $\mathcal{Z}$ is unordered, then Ξ, the set of erroneous output patterns, and $\mathcal{Z}$ are disjoint.

$$G(Z) = \begin{cases} 0, & Z \in \mathcal{Z} \\ 1, & Z \in \Xi \\ X, & \text{otherwise} \end{cases} \tag{5.4}$$

A positive (unate) realization of $G(Z)$ must exist as a consequence of Theorem 5.1 and the definition of a viable test. However, a minimizer (*eg.* ESPRESSO) may produce a non-positive result, possibly using fewer terms than a minimal positive solution. To force a minimizer to produce a positive minimization for $G(Z)$, redefine it as equation 5.5. Note that no value for Z satisfies: $Z^i \supseteq Z \supseteq \xi^j$.

$$G(Z) = \begin{cases} 0, & \exists\, Z^i \in \mathcal{Z} :\ Z^i \supseteq Z \\ 1, & \exists\, \xi^j \in \Xi :\ \xi^j \not\supseteq Z \\ X, & \text{otherwise} \end{cases} \tag{5.5}$$

In terms of cubes, equation 5.5 is rather simple. For each binary vector in $\mathcal{Z}$ create a cube replacing all 1's with X's and with output 0 (*eg.* 0110 → 0XX0/0); for each binary vector in Ξ create a cube replacing all 0's with X's and with output 1 (*eg.* 0111 → X111/1). To see that equation 5.5 forces a positive minimization, suppose the contrary, *i.e.* that a cube C with c_i=0 is part of a minimal cover. For C to be an implicant of $G(Z)$ as defined by equation 5.5, it can only cover minterms from cubes derived from equation 5.5 (as described above) whose i^{th} coordinate is X (don't care minterms from equation 5.4

with z_i=0 which are adjacent to an *OFF*-set minterm, are included in the *OFF*-set by equation 5.5), hence C is not prime because $C \circ c_i \rightarrow X$ is also an implicant of $G(Z)$. Therefore no such cube C is part of a prime implicant cover of $G(Z)$, and a positive minimization is assured.

5.2.2.1 Continuing Example

The checker function (derived using equation 5.4) for the ongoing example of section 5.1.3 is illustrated in Figure 5.6. Note that the solution requires 1 check bit output as before, but now 3 test terms suffice instead of the 4 determined in section 5.2.1.1.

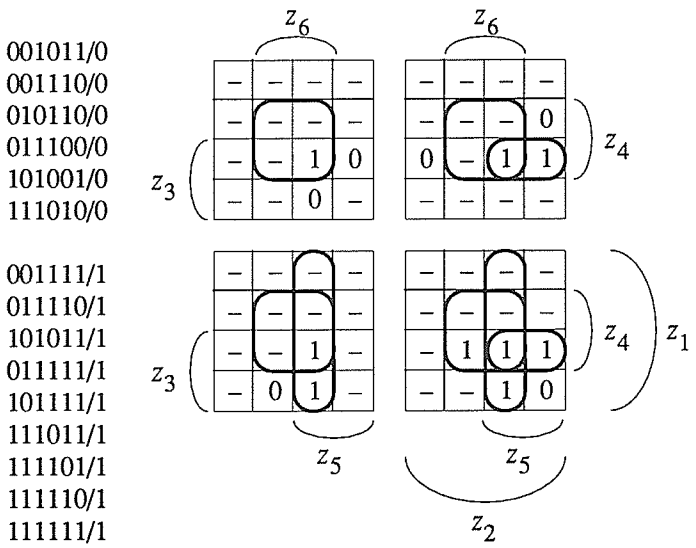


Figure 5.6 Derivation of OR-*k* test terms for example PLA.

5.2.3 More Experimental Results

Table 5.2 presents results for OR-*k* testability for the same PLA's listed in Table 5.1. All three unordering methods described in section 5.2.1 were tried for each PLA, positive checkers were generated according to equation 5.5, and the best results are reported. Results for OR-*k* testing using modified Berger encoding (*cf.* section 2.3.4.3) to make

$\mathcal{Z}$ unordered are also shown. Overhead figures for OR-3 testability from section 5.1.4 are repeated here for comparison. Note that for the OR-k scheme $n_C = \lceil \log_2 \mathsf{L} \rceil$.

Non-positive checkers have also been generated, and generally are the same size as the positive ones. For PLA's "dk17", "dist", "dc2" and "gary", the positive checker is 1, 2, 3 and 7 terms larger, respectively; for "dk27" and "router" it is 1 term smaller.

	OR-k Scheme							Modified Berger OR-k			OR-3 Scheme (from Table 5.1)		
PLA	*N*	L	U	n_C	*T*	Overhead *vs.* Naked	Overhead *vs.* 1-Concurrent	n_C	*T*	Overhead *vs.* Naked	n_C	*T*	Overhead *vs.* Naked
alu1	80	8	†	3	76	5978%	12%	3	76	5978%	10	10	7059%
alu2	67	8	3	3	39	510	15	3	43	513	10	8	628
apla	37	6	1	3	24	256	25	3	22	252	9	16	306
bis1	5	3	1,3	2	2	84	31	2	3	93	2	2	84
bis2	4	2	†	1	1	38	15	1	1	38	2	2	60
bis5	8	4	3	2	3	32	32	2	4	37	3	3	43
bis6	5	3	‡	2	3	61	38	2	3	61	2	2	53
bw	22	2	1	1	10	37	37	3	9	41	1	10	37
dc1	10	4	1,2	2	5	59	43	3	5	70	3	3	56
dc2	91	6	2	3	86	370	42	3	89	373	12	7	418
dist	21	4	2,3	2	17	54	13	2	18	55	9	8	100
dk17	24	4	3	2	30	260	32	3	30	274	11	25	367
dk27	38	5	2	3	31	493	45	3	34	507	10	20	603
eg	6	2	‡	1	3	36	36	2	3	50	2	3	50
f2	10	2	†	1	10	88	50	1	10	88	5	4	115
fsm1	13	4	‡	2	8	38	22	2	8	38	5	6	52
fsm2a	14	4	2	2	13	80	35	2	15	85	5	11	108
fsm2b	11	3	3	2	13	55	55	2	15	61	6	11	88
fsm2c	24	5	1	3	12	37	29	3	15	41	7	8	55
gary	69	8	2	3	60	53	23	3	62	53	16	14	81
misex1	10	4	3	2	4	31	21	3	6	44	3	3	33
misex2	34	4	3	2	66	1705	7	2	85	1725	8	72	1882
rd53	5	2	†	1	3	11	11	1	3	11	2	3	19
router	15	3	1	2	9	28	22	3	12	38	5	11	45
sao2	9	3	‡	2	7	23	11	2	7	23	5	6	37
traffic	8	3	3	2	5	54	38	3	7	76	2	5	54
vg2	23	7	1	3	11	1739	5	3	15	1739	7	7	1859
wim	11	5	3	3	3	96	35	3	6	118	4	4	116
z4	15	4	†	2	11	145	14	2	11	145	6	6	193

U – best unordered encoding method: 1 – bottom up, 2 – top down, 3 – random
† – minimal unordering is unique
‡ – all three encoding methods result in the same number of test terms

Table 5.2 OR-k testability results.

Generally OR-k performs better than OR-3. Solution size is sensitive to the method used to make $\mathcal{Z}$ unordered. For example, "misex2" using method 1 requires 85 test terms compared to the 66 terms using method 3. Some PLA's have a unique minimal unordering, while others may be made unordered in many ways; for example, "bw" has 262,144 possible unorderings. For some PLA's, a modified Berger encoding requires more check bits than a minimal unordering. Overhead figures for modified Berger unordering are similar to or worse than the methods of section 5.2.1.

Nonconcurrency still remains as the dominant source of overhead. For Table 5.2, an OR-k testable PLA is on average 27 percent larger than a 1-concurrent PLA.

It is interesting to note that for one PLA, "f2", the unique minimal unordering requires a single check bit which happens to be the parity of the original output bits.

5.2.4 Variations on OR-k Testing

Several considerations arise with regard to the OR-k strategy.

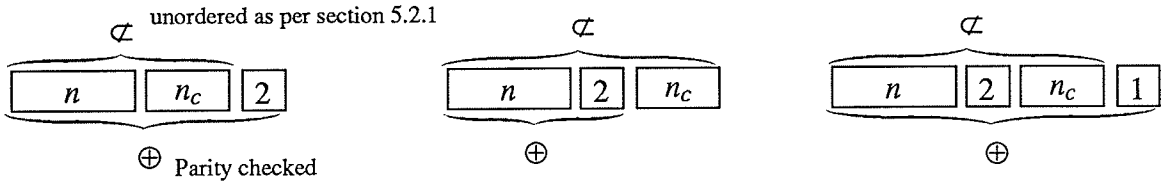
5.2.4.1 Multiple Faults

The strategy is designed to detect errors of type ξ, that is the erroneous selection of two product terms of an otherwise 1-concurrent array. However any error resulting from the selection of any number of multiple product terms is also detectable. From Theorem 3.2, no single fault can cause such an error, thus the strategy is capable of detecting a large and significant class of multiple faults. Note that Chen , Fuchs and Abraham [Chen85] ascribe this sort of multiple-ξ error model to nonconcurrent PLA's.

5.2.4.2 Variations

A variation similar to that of section 4.1.6 may be applied to the OR-k strategy. Recall that a 1-out-of-2 code and a parity check are required to detect errors other than those of type ξ. Specifically, the two-rail parity check bits, which were proposed in

section 4.1.1.1 and adopted at the beginning of this chapter, may be used to advantage. If the 1-out-of-2 encoded parity check bits are defined first, then this will tend to make the set of output patterns less ordered. However, single bit errors in the additional check bits defined as per section 5.2.1 would not be protected by parity. This arrangement is depicted as Variation 1 in Figure 5.7b; the basic codeword structure for OR-k testing, assumed in the previous sections, is shown as Figure 5.7a.



a) Basic OR-k scheme from section 5.2.3.

b) Variation 1: two-rail parity defined before unordering.

c) Variation 2: arbitrary two-rail code + parity.

Figure 5.7 Variations of codeword partitioning.

Alternatively, if an arbitrary pair of check bits is encoded in a 1-out-of-2 code, and a third check bit is introduced for the parity check, then the unordering procedure may be simplified. The codeword structure for this arrangement is shown as Variation 2 in Figure 5.7c. The elements of a partial order ($\mathcal{Z}$), represented by a Hasse diagram such as Figure 5.4, could then be partitioned into two sets ($\mathcal{Z}_{01}$ and $\mathcal{Z}_{10}$ according to the 1-out-of-2 codeword assigned) and each subproblem solved independently. However, at best, the elements along the longest chain would be split between the two partitions and the resulting longest chain would be half as long. This would only reduce n_C by one. Since the two-rail parity check bits are being supplanted by an arbitrary 1-out-of-2 encoding plus a parity check bit, the net effect is no change in total number of check bits. It is unclear how this partitioning affects the number of test terms required.

Two-rail parity check bits are more likely to be beneficial in reducing the number of test terms, than in reducing n_C. The set $\mathcal{Z}$ may be considered partitioned into $\mathcal{Z}_{01}$ and

$\mathcal{Z}_{10}$ on the basis of these check bits (even if the parity check bits are defined after unordering). Firstly, these two check bit outputs alone form a viable OR-2 test, which is checked by the checker which verifies the 1-out-of-2 codeword. All errors $\xi^{i,j}$ where $Z^i \in \mathcal{Z}_{01}$ and $Z^j \in \mathcal{Z}_{10}$ are detectable by this test. Any other viable test including either or both of these two outputs can not detect any other error. Consequently, the covered errors need not be considered further, and the two-rail parity outputs need not participate in the generation of the remaining test terms. Secondly, errors of type ξ involving pairs of product terms from the same partition may in fact result in a parity error. Such errors need not be considered further either.†

Variation 3 continues to use the basic arrangement of Figure 5.7a, but uses a checker design which takes into account errors detectable by the two-rail parity code. To synthesize such a checker, simply assign such detectable erroneous output patterns‡ the value "X" instead of "1" in equation 5.4 or 5.5. For equation 5.5, only the minterm associated with the already covered error (rather than the cube it induces) need be considered a don't care, and that minterm may still be defined as "1" because of some other error event.

A fourth unordering method, in addition to those defined in section 5.2.1, may now be appropriate, specifically one which attempts to evenly split $\mathcal{Z}$ into $\mathcal{Z}_{01}$ and $\mathcal{Z}_{10}$ on the basis of codeword parity. This would tend to maximize the number of errors of type ξ covered by two-rail parity. The details of such a method are not considered here.

The checker obtained in section 5.2.2.1, for the ongoing example of section 5.1.3, is modified in Figure 5.8 to illustrate the benefit of considering the error coverage of two-rail parity (Variation 3). Two of the erroneous output patterns are detected by the 1-out-of-2 code (denoted "C" on the map), and four others by the parity check (denoted "P" on the map). The remaining three patterns are detectable using a single test term. (If we

† Multiple fault coverage as defined in section 5.2.4.1 may be diminished.

‡ $\mathcal{Z}$ and Ξ include the two-rail parity bits at this point, but they are dropped when minimizing $G(Z)$.

only take into account those errors detectable by the 1-out-of-2 code alone, then 3 test terms would still be required.) Therefore the PLA is made OR-*k* testable using 1 check bit output and 1 test term in addition to the 1-out-of-2 encoded parity check bits.

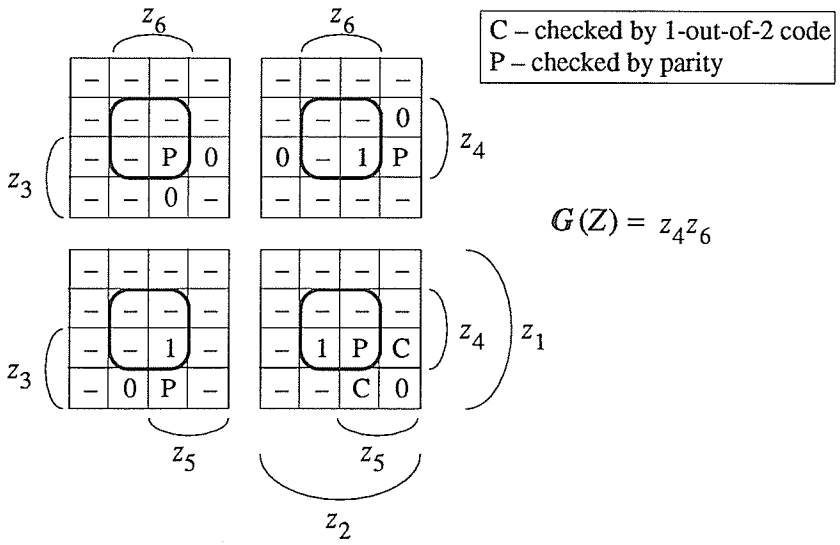


Figure 5.8 Variation 3 (OR-*k*) checker design for "eg" PLA.

Table 5.3 shows results for checkers designed according to Variation 3 for the same PLA's as are in Table 5.2. Note that of the three unordering methods tried, that which was "best" for a basic OR-*k* checker in Table 5.2, is not necessarily "best" for Variation 3. The average overhead, relative to a 1-concurrent PLA, is 15 percent for Variation 3 as compared to 27 percent noted previously for the basic checker design. The average number of terms for a checker is 5.6 for Variation 3 compared to 19.5 for basic OR-*k*.

PLA	OR-k — Variation 3					Basic OR-k (from Table 5.2)			
	U	n_C	T	Overhead *vs.* Naked	Overhead *vs.* 1-Concurrent	U	T	Overhead *vs.* Naked	Overhead *vs.* 1-Concurrent
alu1	†	3	26	5888 %	10 %	†	76	5978 %	12 %
alu2	1	3	7	492	11	3	39	510	15
apla	1	3	9	228	15	1	24	256	25
bis1	2	2	0	65	18	1,3	2	84	31
bis2	†	1	0	31	9	†	1	38	15
bis5	‡	2	1	22	22	3	3	32	32
bis6	2	2	0	38	18	‡	3	61	38
bw	1	1	5	20	20	1	10	37	37
dc1	1,3	2	1	33	19	1,2	5	59	43
dc2	2	3	24	301	21	2	86	370	42
dist	1	2	5	51	11	2,3	17	54	13
dk17	1,2	2	10	213	15	3	30	260	32
dk27	1,3	3	10	400	22	2	31	493	45
eg	‡	1	1	18	18	‡	3	36	36
f2	†	1	3	51	21	†	10	88	50
fsm1	‡	2	3	28	13	‡	8	38	22
fsm2a	2,3	2	3	56	17	2	13	80	35
fsm2b	3	2	2	18	18	3	13	55	55
fsm2c	1	3	4	26	18	1	12	37	29
gary	2	3	22	40	13	2	60	53	23
misex1	1	2	1	21	12	3	4	31	21
misex2	2	2	12	1648	4	3	66	1705	7
rd53	†	1	1	9	9	†	3	11	11
router	2	2	2	14	9	1	9	28	22
sao2	2	2	1	20	9	‡	7	23	11
traffic	‡	2	1	30	17	3	5	54	38
vg2	2	3	3	1737	5	1	11	1739	5
wim	1,3	3	1	81	25	3	3	96	35
z4	†	2	3	141	12	†	11	145	14

U – best unordered encoding method: 1 – bottom up, 2 – top down, 3 – random
† – minimal unordering is unique
‡ – all three encoding methods result in the same number of test terms

Table 5.3 OR-*k* testability results — Variation 3.

5.2.4.3 Limited Concurrency and the TSC Goal

The OR-*k* testing strategy meets the TSC goal of producing a noncodeword as its first incorrect output due to a sequence of modeled faults, for a large class of fault sequences. The scheme is designed to detect errors produced by any Class I fault and consequently Class II and III faults are also covered. What remains to be shown is that no sequence of undetectable faults can result in a modeled fault producing an undetectable error.

For checking schemes restricted to observing circuit outputs (*i.e.* no internal circuit nodes are observed), limited concurrency is at least as effective as nonconcurrency, because an undetectable fault which transforms a 1-concurrent PLA into one with limited concurrency can always be found. It will be shown shortly (Theorem 5.2) that limited concurrency is in fact more effective than 1-concurrency. Consider a 1-concurrent realization where cubes 0X00 and XXX1 are associated with the same codeword (illustrated in Figure 4.10b). A fault which transforms the former cube to 0X0X (resulting in the map shown in Figure 4.10a) is clearly undetectable but the PLA is now 2-concurrent. Therefore, the 1-concurrency design criterion may be relaxed to that of limited concurrency which is less expensive (in terms of both design effort and solution size).

Theorem 5.2 An OR-k PLA realized with limited concurrency is SFS with respect to all sequences of Class I, II and III faults.

Proof: Assume the PLA is realized such that a minimal number of product terms are used for each output pattern. Consider the product terms, C_{Z^i}, associated with a given output pattern, Z^i; treat C_{Z^i} as a single output function. No sequence of undetectable faults can transform the array such that one of the terms realizing C_{Z^i} becomes covered by the rest. This is a result of having a minimal (irredundant) realization of C_{Z^i}; a sequence of faults producing a realization with one fewer term (the covered term being redundant) is impossible. Therefore each product term in C_{Z^i} must contain some distinguished minterm which is uniquely covered by that term. Any single fault affecting the output part of a particular product term is detectable when the distinguished minterm is applied at the inputs. Any sequence of faults affecting the AND-plane will either cause a detectable error (either of type ξ or producing an all 0 output), or be an undetectable transformation of the array which preserves the distinguished minterm property. □

When faults transform a 1-concurrent PLA to one with limited concurrency, it is unlikely that the result will be minimal. Thus some product term may become completely covered by the rest. The output pattern associated with that term may now be altered by a sequence of undetectable faults each causing a single output to change 1→0. If another fault causes that term to no longer be covered by the rest, then an undetectable error may arise. This last fault in some sense "undoes" the effect of some of the previous faults which caused the product term to become redundant in the first place. One may characterize the OR-k scheme as being designed to detect single bit 1→0 errors and multiple 0→1 errors.

Table 5.4 presents overhead figures for the same checking arrangement as Table 5.3, namely Variation 3 OR-k testing relative to the naked PLA, except that the PLA's are realized with limited concurrency instead of 1-concurrency. As shown in Table 4.3, both realizations require identical numbers of product terms for most of the example PLA's. Therefore, overhead figures shown in Tables 5.3 and 5.4 are nearly identical.

A second set of overhead calculations is also shown in Table 5.4. These are based on a more realistic chip area cost estimator, equation 5.6, which is derived from actual PLA layout parameters taken from [Bosw85a], and explained further in Appendix C.

$$\begin{aligned} Cost = \; & 64\,[(2m + n + n_C)p + (n + n_C)T] + \\ & 484p + 128m + (n + n_C)(176a + 308) + 548T + 308a \end{aligned} \tag{5.6}$$

The cost of a normal minimized PLA, or one with limited concurrency only, is determined by using the appropriate value of p and setting n_C=T=a=0; a=1 for a PLA which is OR-k testable.

OR-k — Variation 3 — Limited Concurrency Overhead *vs.* Naked PLA		
PLA	equation (5.2)	equation (5.6)
alu1	5433 %	4958 %
alu2	492	472
apla	228	216
bis1	65	69
bis2	31	39
bis5	22	29
bis6	38	45
bw	20	26
dc1	33	41
dc2	301	292
dist	51	49
dk17	213	203
dk27	400	357
eg	18	33
f2	51	62
fsm1	28	34
fsm2a	56	57
fsm2b	18	24
fsm2c	26	28
gary	40	42
misex1	21	27
misex2	1644	1546
rd53	9	10
router	14	19
sao2	9	9
traffic	30	38
vg2	1415	1390
wim	81	79
z4	141	134

Table 5.4 OR-*k* overhead results — Variation 3 with limited concurrency.

The simple cost estimator (of 5.2) seems to correlate well with the more realistic one.

5.2.5 Checker Design

Three distinct sections comprise the checker for this on-line checking strategy. A parity generator computes the parity of all but the parity check bits. The two-rail output of this circuit is compared to the 1-out-of-2 encoded parity check bits. Both sources are verified as being 1-out-of-2 codewords. Finally the OR-*k* test terms are computed and if

any term is a "1", then an error is considered detected. Parity generators and two-rail comparators/checkers are well-known. This section will consider the self-checking properties of the OR-k section of the checker.

The checker as conceptually described in section 5.1.1, and defined as $G(Z)$ in section 5.2.2, consists of a T-input OR gate OR'ing the outputs of T AND gates of various fan-in. The AND gate inputs are driven by the PLA outputs. For an error-free PLA, the checker (OR-gate) outputs a "0"; it outputs a "1" upon detection of an error of type ξ. A positive realization of $G(Z)$ is desirable for two reasons. Firstly, since no complemented literals are used, no inverters have to be implemented. Secondly, Smith proved (Theorem 2 [Smit77]) that a two-output checker which is TSC with respect to unidirectional faults must be inverter-free, and an inverter-free circuit must realize a positive function.

As described above, a checker output stuck-at-0 fault would mask all errors. Two approaches are considered. The first is to redefine $G(Z)$ as a code translator. For example $G'(Z)$ defined by equation 5.7 produces a 1-out-of-2 encoded error indicator.

$$G'(Z) = \begin{cases} 01 \text{ or } 10, & Z \in \mathcal{Z} \\ 00 \text{ or } 11, & Z \in \Xi \\ XX, & \text{otherwise} \end{cases} \tag{5.7}$$

Equation 5.7 provides considerable opportunities for choice in defining $G'(Z)$, the only restriction is that each output pattern, 01 and 10, is defined to appear for some PLA codeword outputs (so that a stuck-at fault on either of G''s outputs will not go undetected). This degree of flexibility and the fact that two functions are actually being defined, makes it difficult to determine low overhead designs. A checker with a positive realization always exists as shown by equation 5.8 (α=1,2). Define $G' = (g_1, g_2)$, $\mathcal{Z}_1=\{Z\in\mathcal{Z}:\ G'(Z) = 01\}$, and $\mathcal{Z}_2=\{Z\in\mathcal{Z}:\ G'(Z) = 10\}$.

$$g_\alpha(Z) = G(Z) + \sum_{Z^i \in Z_\alpha} \prod_{Z^i_j=1} z_j \tag{5.8}$$

The second approach is to enhance the checkability of the simple design. One technique is described here. Suppose $Z^i \in \mathbb{Z}$, $Z^i=\mathbf{1}$†. Arrange that all OR-k check bits associated with Z^i be 1's except for one, $Z^i_\alpha=0$. Ensure that $Z^j_\alpha=1$ for all $j \neq i$. Otherwise disregard Z^i while determining OR-k check bits and test terms. When the product term associated with Z^i is selected, all test terms should be asserted. Consider the checker circuit outlined in Figure 5.9. Although not TSC, this design is a considerable improvement over the simple checker for a small increase in overhead. Note that in addition to finding the OR of all test terms, the AND of all test terms is also computed.

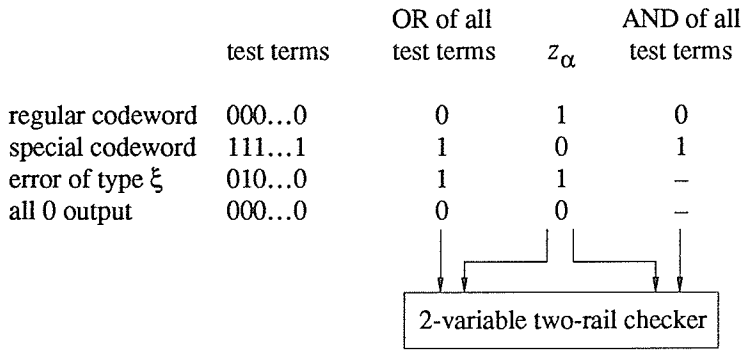


Figure 5.9 OR-k test term checker.

The self-checking properties of this checker may be determined for a single stuck-at fault model. A two test term checker with these AND and OR gates is illustrated in Figure 5.10.

† If this is not the case naturally, then the system may be periodically placed into test mode where a **1** output is selected by 'forbidden' input vectors, as suggested by Kling and Banerjee [Klin86].

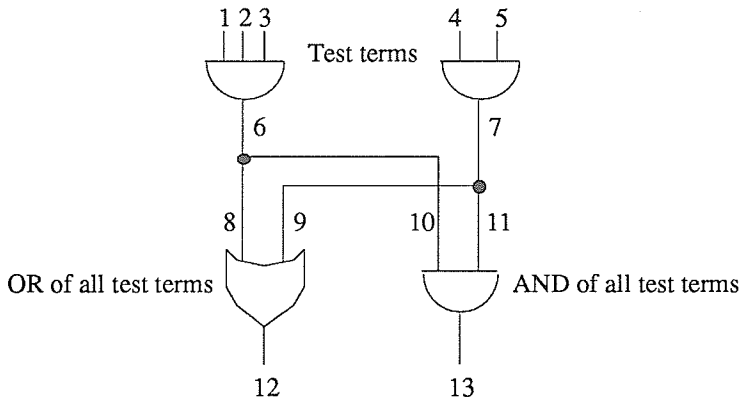


Figure 5.10 OR-*k* checker example.

Consider the faults of line l_i stuck-at-α, denoted l_i/α, for i=1..13 and α=0,1. Stuck-at faults on l_{12} and l_{13} are tested by the test term outputs for regular and special codewords as shown in Figure 5.9. The faults $l_6/1$ and $l_7/1$ dominate $l_{12}/1$, while $l_8/1$ and $l_9/1$ are equivalent to $l_{12}/1$. The faults $l_1/0$, $l_2/0$, $l_3/0$, $l_4/0$, $l_5/0$, $l_6/0$, and $l_7/0$ dominate $l_{13}/0$, while $l_{10}/0$ and $l_{11}/0$ are equivalent to $l_{13}/0$. The faults $l_1/1$, $l_2/1$, $l_3/1$, $l_4/1$ and $l_5/1$ are secure in the sense that the test term they drive will be asserted appropriately (or for some codeword outputs, but this simply signals the checker fault) even in the presence of these faults. Only the faults $l_8/0$, $l_9/0$, $l_{10}/1$, and $l_{11}/1$ are undetectable and may mask an error indication. This may be alleviated somewhat by duplicating the gates which compute the OR and AND of all test terms. Alternatively, the checker may be periodically tested off-line, perhaps using some BIST technique.

5.3 SUMMARY

This chapter presented a family of on-line checking strategies called OR-*k* testing. These strategies are designed to detect all errors from single faults (Class I, II and III) in nonconcurrent PLA's. In general, an OR-*k* testable PLA is designed to produce codewords composed of three codes:

a) the rightmost two outputs are encoded in a 1-out-of-2 code to detect errors of type I.8 and an all 0 output,

b) a parity check bit is used to detect single bit errors, and

c) codewords are unordered to detect errors of type ξ.

A two-rail parity code was advocated to fulfill a) and b) simultaneously.

The concept of a viable OR-k test term was introduced. A viable test is a selection of outputs which are normally never all 1's, such that when they are observed to be all 1's, then it may be concluded that it is due to an error of type ξ. The OR-k checking strategy thus is designed to recognize noncodewords produced by errors of type ξ, rather than trying to verify that each output pattern is a codeword. All multiple faults where each causes an error of type ξ, are detectable.

The first form of OR-k testing considered was one where all test terms are the same size, specifically an algorithm was given to extend $\mathcal{Z}$ with check bits and determine test terms of size three such that all errors of type ξ are detectable, *i.e.* an OR-3 strategy.

It was noted that $\mathcal{Z}$ must be unordered if all errors of type ξ are to be detectable. Then it was shown that if $\mathcal{Z}$ is unordered, then there exists a viable test of size $k \geq 2$ for each error of type ξ without the need for any additional check bits, although extra check bits may be required to ensure OR-3 testability. The approach of using test terms of varying size was called the basic OR-k strategy. Methods for augmenting $\mathcal{Z}$ with a minimal number of check bits to make it unordered, and for deriving OR-k test terms to detect all errors of type ξ, were presented.

Variations of OR-k testing, which take advantage of the two-rail parity check bits, were suggested. Most benefit accrues from Variation 3 which simply specifies that all type-ξ errors which are detectable by two-rail parity need not be testable by an OR-k test term.

Limited concurrency was shown to be more secure (SFS) than 1-concurrency, as well as being less expensive in terms of both area overhead and design effort.

Experimental results were presented for a number of sample PLA's and several testing strategies, including: OR-3, basic OR-k, Variation 3 with limited concurrency, and the latter again but using a chip area cost estimator function based on actual layout figures. The most significant component contributing to area overhead is that due to concurrency (*i.e.* either 1-concurrency or limited concurrency). Table 5.5 summarizes the average number of check bits (excluding two-rail parity), average number of test terms, and average overhead (relative to a 1-concurrent PLA, or one with limited concurrency, as appropriate) for the forms of OR-k testing studied.

Strategy	Concurrency	Cost Function	n_C	T	Overhead
OR-3	1-concurrent	eqn. (5.2)	5.9	10.0	42.1 %
Basic OR-k	1-concurrent	eqn. (5.2)	2.1	19.5	27.4
Variation 3	limited concurrency	eqn. (5.2)	2.1	5.6	14.9
Variation 3	limited concurrency	eqn. (5.6)	2.1	5.6	19.4

Table 5.5 Averages of results from Tables 5.1 to 5.4.

The number of test terms required to make a PLA OR-k testable was found to be sensitive to the actual check bits defined to make $\mathcal{Z}$ unordered. For some PLA's, the "best" unordering for basic OR-k is not necessarily the "best" for Variation 3.

The design of a practical, if not TSC, OR-k checker was presented along with an analysis of its self testing capabilities. The checker is a two-level circuit, and thus incurs minimal delay overhead.

Chapter 6

COMPARISON WITH CONCURRENT STRATEGIES

Chapters four and five dealt exclusively with nonconcurrent PLA's, or with PLA's having limited concurrency. The circuit area overhead attributed to the achievement of such concurrency is significant, and one might use this as an argument against the use of these techniques. In this chapter, we examine the overhead associated with on-line checking strategies which use conventionally minimized PLA's, and compare this with the overhead associated with OR-*k* testing.

6.1 COMPARISONS

Several online checking strategies will be compared to those introduced previously in chapters four and five. The overhead of an online checked PLA, versus the area required by a naked PLA, may be generally attributed to one of three sources: increased number of product term lines, extra outputs, and checker circuitry. The following three subsections address these in turn.

6.1.1 Increased Number of Product Terms

Any PLA checking strategy for which normal PLA outputs are embedded in some error detecting code, generally requires more product term lines to realize the enlarged set of output functions (which now include check bit outputs defined so that each PLA output is a codeword). Table 6.1 illustrates this increase in product terms for a set of sample PLA's augmented by each of several checking strategies. Some of the table entries are empty because the augmented PLA's were impractical to construct using straightforward computer programs.

PLA	Naked	Limited	Parity	Mod 3 Residue		Berger		
				Method 1	Method 2	Normal	Modified	[Fuch84]
alu1	19	956	380	618	660	956	956	2074
alu2	68	361	183	256	198	296	296	563
apla	25	100	46	80	45	87	87	759
bis1	5	8	6	8	6	8	8	14
bis2	5	8	7	8	6	8	8	14
bis5	10	11	11	11	6	11	11	15
bis6	6	10	7	10	5	10	10	14
bw	22	24	24	24	8	24	24	32
dc1	9	12	11	11	5	12	12	16
dc2	39	138	76	112	85	114	121	101
dist	120	165	148	160	87	167	167	153
dk17	18	74	35	62	29	67	64	447
dk27	10	47	29	38	31	47	46	177
eg	6	6	6	6	4	6	6	8
f2	8	15	11	15	9	15	15	15
fsm1	15	27	25	26	11	27	27	729
fsm2a	18	26	22	26	17	26	26	46
fsm2b	16	22	21	22	12	22	22	30
fsm2c	31	42	38	41	21	43	42	195
gary	107	147	130	141	91	145	145	15534
misex1	12	18	18	18	8	18	18	169
misex2	28	472	181	313	203	408	407	
rd53	31	32	32	32	26	32	32	32
router	22	26	25	26	18	26	26	1724
sao2	58	60	60	60	44	60	60	942
traffic	9	10	9	9	7	10	10	32
vg2	110	1596	619	649	397	1596	1596	
wim	9	13	12	13	8	13	13	13
z4	59	128	92	123	113	105	105	70

Table 6.1 Product terms for PLA's augmented under various checking strategies.

The column of Table 6.1 labelled "Limited" represents the number of product terms in a complete covering realized with limited concurrency. The results shown in Table 4.3 suggest that nonconcurrency may be achieved with nearly the same number of product terms as is limited concurrency. Method 2 (mod 3) shows the size of the separate residue PLA of [Saye85] (*cf.* section 3.2.2). Modified Berger encoding is that of [Mak82] (*cf.* section 2.3.4.3). The strategy of [Fuch84] utilizes the Berger check bits of the inputs, as well as the outputs, to derive output check bits (*cf.* section 2.3.4.4). All augmented PLA's were minimized by ESPRESSO.

All checking strategies incur an increase in the number of product terms required to realize an augmented PLA. Even the strategies which provide only partial error coverage, parity and mod 3 residue codes, have expensive realizations for some PLA's. Limited concurrency compares favourably with the strategies based on Berger codes. The explosion in the number of product terms for the scheme of [Fuch84], suggested in section 2.3.4.4, may be seen to occur for several instances.

6.1.2 Extra Outputs

Check bit outputs generally must be created in addition to the PLA's normal n outputs in order to ensure that each output pattern is a codeword from an error detecting code. For most checking strategies considered, there are simple expressions for the number of check bit outputs (or a bound on the same). Table 6.2 summarizes these expressions for the strategies being compared here. Recall that L is the length of the longest chain of the Hasse diagram of the partial order defined by $\mathcal{Z}$; $\mathsf{L} \leq n+1$. Check bits required to construct a two-rail or two-rail parity subcode for the proposed checking strategies, Simple Parity and OR-k, respectively, are included in these expressions.

Checking Strategy	Number of Check Bit Outputs
parity	1
mod 3	2
Berger	$\lceil \log_2 n+1 \rceil$
modified Berger	$\leq \lceil \log_2 n+1 \rceil$ †
[Fuch84]	$\leq \lceil \log_2 \max(n, m)+1 \rceil + 1$†
Simple parity (4.1.1)	$\leq n + 1$
OR-k (5.2.2)	$\lceil \log_2 \mathsf{L} \rceil + 2$

† depends on maximum/minimum weight of patterns before encoding

Table 6.2 Extra outputs required for various checking strategies.

Table 6.3 shows the number of extra outputs required by the above strategies for a set of sample PLA's. Also shown are bounds on the minimal number of check bits as determined by the method of section 3.3. These are determined for both the normal minimized PLA's, and for ones with limited concurrency.

PLA	*n*	**Minimal**		**Simple Parity**	**OR-k**	**Berger**		
		Naked	LC			Normal	Modified	[Fuch84]
alu1	8		4	11	5	4	4	5
alu2	8		4	13	5	4	3	5
apla	12	3	3	13	5	4	4	5
bis1	3	2	3	4	4	2	2	4
bis2	3	2	2	2	3	2	2	4
bis5	4	3	3	7	4	3	3	4
bis6	3	2	3	5	4	2	2	4
bw	28	2	2	21	3	5	5	6
dc1	7	2	3	8	4	3	3	4
dc2	7			8	5	3	3	5
dist	5	3	3	6	4	3	3	5
dk17	11	3	3	11	4	4	3	5
dk27	9	2	3	11	5	4	3	5
eg	5	2	2	4	3	3	2	4
f2	4	2	2	5	3	3	2	4
fsm1	6	3	3	7	4	3	3	5
fsm2a	6	3	3	6	4	3	3	4
fsm2b	7	2	3	7	4	3	2	4
fsm2c	7	3	3	9	5	3	3	5
gary	11			15	5	4	4	5
misex1	7	3	3	8	4	3	3	5
misex2	18		3	15	4	5	3	6
rd53	3	2	2	4	3	2	2	4
router	12	3	3	8	4	4	3	5
sao2	4	2	3	5	4	3	2	5
traffic	7	2	2	7	4	3	3	4
vg2	8		4	9	5	4	3	6
wim	7	3	3	10	5	3	3	4
z4	4	2	3	5	4	3	3	4

Table 6.3 Extra outputs for PLA's augmented under various checking strategies.

With the exception of "Simple Parity"†, all strategies shown in Table 6.3 require similar numbers of check bits, and are nearly minimal. Conflict graph partitioning

† Pseudo-outputs are not included in the check bit counts for Simple Parity shown in Table 6.3.

proposed in section 4.1.6 will tend to bring "Simple Parity" more in line with the others. Extra outputs are therefore less significant of a distinguishing factor than the increase in the number of product terms.

6.1.3 Checkers

Code checkers clearly introduce a non-negligible area overhead. What is not so clear, is the magnitude of this overhead. Several different checker designs for each code have been proposed (*cf.* section 2.2.2), although some of these designs are constrained to specific code subclasses. Additionally, clever check circuit design, layout or sharing (with other CUC's on the same chip) make it difficult to compare results reported in the literature. For this discussion, simple estimates of checker size based on logic gates will be used. Checker delay will also be considered in terms of gate levels. An exclusive-OR gate will be conservatively modeled as two logic gates with a delay of 2, also. For the remainder of this section, assume that a codeword consists of n information bits† and n_C check bits, and that $k = \left\lceil \frac{n}{2} \right\rceil$, $h = \lceil \log_2 k \rceil$ and $h' = \lceil \log_2 n + 1 \rceil - 1$‡. All codes considered are systematic (separable).

A parity checker may be implemented with n–1 two-input EXOR gates arranged in a tree structure of depth $\lceil \log_2 n \rceil$. Therefore, such a checker requires $4k$–2 gates and has a delay of $2h$+2, while one with two-rail outputs requires $4k$–4 gates and has a delay of $2h$. The layout of a tree-shaped circuit tends to require considerable additional area for routing wires, and its triangular shape may create pockets of unused chip area which are difficult to utilize for other circuitry. An alternate realization is a linear cascade of EXOR gates, however the delay jumps to $4k$–2 (*i.e.* through all n–1 EXOR gates).

A checker for the mod 3 residue code consists of a residue generator and a 2-bit comparator. Even though the comparator design must account for two representations for

† For convenience, assume n is even as this simplifies some of the expressions which follow.

‡ $h' = h$ unless $n=2^{h+1}$, then $h' = h+1$.

a zero residue, *i.e.* 00 and 11, its area will be considered negligible for our purposes. Since the mod 3 code belongs to the family of "low cost residue codes", there exists an economical method to compute the residue of an n-bit binary number. This consists of a tree of residue adder modules, where each module is a 2-bit adder with end-around carry [Wake78]. A straighforward realization of such a module in terms of full and half adders is shown in Figure 6.1. Using textbook gate level designs for these adders [Diet78], a module may be shown to consist of five EXOR gates, four AND gates and one OR gate†, *i.e.* 15 logic gates as defined earlier, with a total delay of 7. The entire generator consists of k–1 residue adder modules in a tree h modules deep. Therefore, a mod 3 residue generator may be constructed from 15(k–1) logic gates, and will have a delay of 7h. This is nearly four times that of a parity generator.

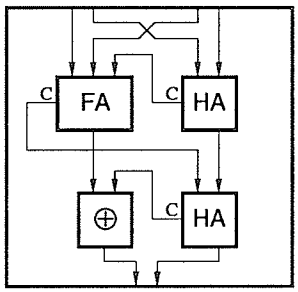


Figure 6.1 Mod 3 residue adder module.

The well known structure of a Berger code checker shown in Figure 6.2 is called a *normal checker*. The 0's counter (or 1's counter) is typically just an adder tree. Such an adder tree requires $\sum_{j=0}^{h-1} 2^j(h-j)$ full adder modules and has a delay of $2h$–1 modules [Maro78]‡. Using the same assumptions as for the mod 3 residue generator, a textbook implementation of a full adder consists of 7 logic gates and has a delay of 4. The delay of the 0's counter alone is nearly 4 times that of a parity checker.

---

† A 13 gate two-level sum-of-products realization of this module also exists.

‡ These expressions are exact only when $n=2^{h+1}-1$.

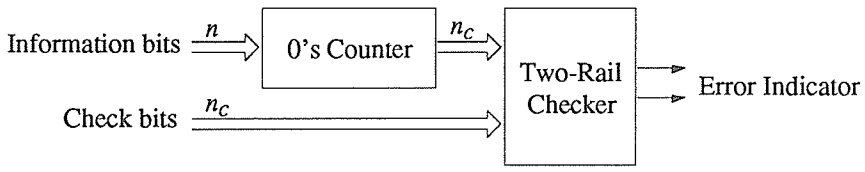


Figure 6.2 Berger code Normal Checker.

A t-variable two-rail comparator (or checker) may be implemented with a modular tree structure similar to the mod 3 residue generator. The basic comparator module is a 2-level circuit made up of four AND gates and two OR gates [Wake78]. Therefore, a two-rail code checker may be constructed from $6(t-1)$ logic gates, and will have a delay of $2\lceil \log_2 t \rceil$. For the normal Berger code checker, an n_C-variable two-rail comparator is required; $n_C = h'+1$.

Recently, alternative structures for Berger code checkers have been proposed which have improved delay characteristics compared to adder based designs ([Pies87, Lo88]). These are based on the observation that a Berger code consists of the union of several particular m-out-of-n codes. Lo and Thanawastien [Lo88] presented a comparison of two designs for various values of n, which is summarized in Table 6.4. Marouf and Friedman's [Maro78] adder based design is also shown.

		[Maro78]		[Pies87]†		[Lo88]†	
n	h'	Gates	Delay	Gates	Delay	Gates	Delay
6	2	36	12	61	8	67	10
7	2	40	14	87	8	76	10
8	3	55	17	93	10	101	12
14	3	91	20	261	11	208	12
15	3	95	20	315	11	227	12
16	4	113	25	321	13	260	15
31	4	206	28	1279	13	720	15
32	5	227	31	1285	15	824	17

† from [Lo88]

Table 6.4 Comparison of Berger code checkers.

The designs of [Pies87] and [Lo88] exhibit improved delay characteristics (approximately $3h'+1$ and $3h'+3$, respectively), but at the expense of area overhead. This is not surprising, since m-out-of-n checkers are not unlike table lookup.

Checkers for modified Berger codes and the encoding of [Fuch84] require a parallel adder or subtracter to adjust the 0's count before comparison with the check bits.

Figure 6.3 graphically illustrates the number of gates required by each of the checking strategies of interest, for the set of sample PLA's in Tables 6.1 and 6.2. The results shown for OR-k are based on Variation 3 (*cf.* Table 5.3).

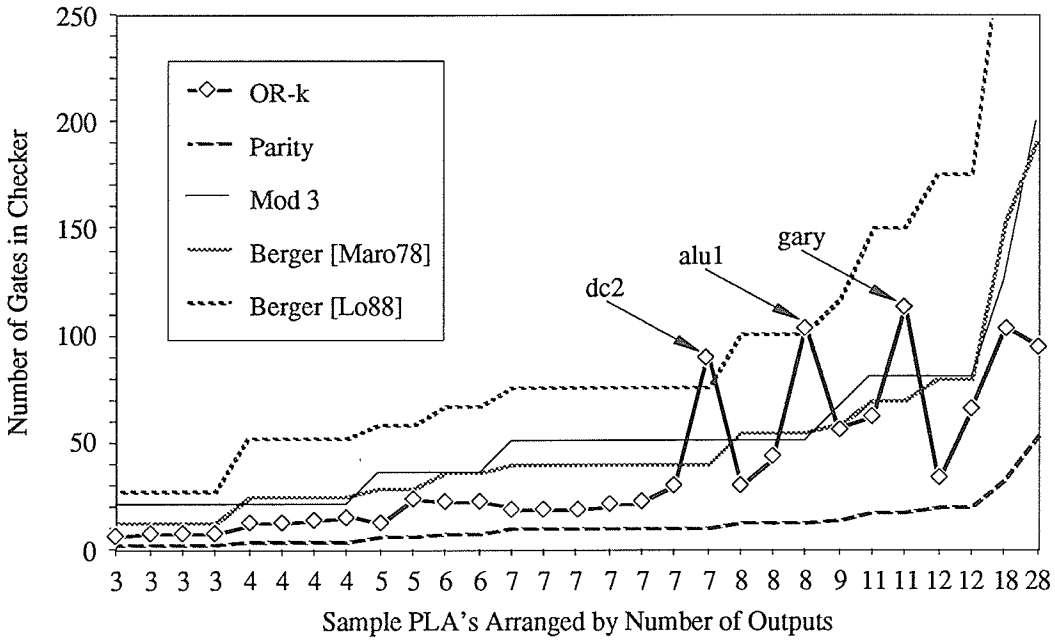


Figure 6.3 Comparison of checker size for various strategies.

The checker proposed for OR-k testing in section 5.2.5 effectively requires $T+2$ gates and has a delay of 2, to produce a 2-variable two-rail error indicator. The value T is function dependent. In addition to this, the two-rail parity of all $n+n_C$ outputs is generated and this is compared with the two-rail parity prediction outputs. Since the OR-k portion of the checker is implemented as an array, we propose that equation 6.1 is more

representative of gate overhead than is T+2. The gate overhead for the parity generator, as described earlier, is combined with this to obtain the total gate cost for an OR-k checker.

$$\text{OR-}k \textit{ checker size} = \left\lceil \frac{T(n+n_C+2)}{4} \right\rceil \tag{6.1}$$

Checker size for all strategies in Figure 6.3, except for OR-k testing, depends on n. The size of an OR-k checker depends on T, which is function dependent. There are some PLA's with quite large OR-k checkers, for example, "alu1", "dc2" and "gary".

In regard to checker delay, the parity checker is the dominant factor in the delay of an OR-k checker, and a parity checker is the fastest of the checkers considered. An OR-k checker is of comparable size to an adder based Berger checker (or a mod 3 checker, which has very similar overhead characteristics to Marouf and Friedman's Berger checker) but is nearly four times faster. An OR-k checker is slightly faster than Berger checkers by [Pies87] and [Lo88], but it is generally significantly smaller.

Some other checking schemes which are based on nonconcurrent PLA's, use 1-out-of-n codes. A 1-out-of-n code checker design which requires $3k$+2 gates and has a delay of 3 has been developed by Izawa, and is described by Tohma in [Tohm86]. However, many of the gates in this design have large fan-in, thus this expression for gates is not comparable to the overhead expressions for the other checkers, which are roughly based on two-input gates.

6.2 COST OF UNORDEREDNESS

Table 6.1 and section 6.1.1 suggest that the realization of a PLA with outputs encoded in a Berger code is inherently nearly as expensive as limited concurrency (or nonconcurrency). In this section, reasons for this will be put forth in general for PLA's with unordered output patterns. A unidirectional error detecting code must be unordered.

Suppose L is the cubical array specifying a minimized realization of a multi-output function, $F(X)$, whose set of output patterns, $\mathcal{Z}$, is unordered. A minimal realization is a direct result of product term sharing, *i.e.* finding and using product terms which contribute to more than one output generally results in a smaller PLA overall than if each output (single-output function) were minimized separately, even though some outputs may require more terms in the shared solution than when solved separately. Suppose that $F(x)=Z^j$, for some particular input vector x.

Consider the set of product terms selected by x: $L'=\{L^{i1}, L^{i2}, \ldots, L^{ik}\}$. Clearly, the set of cubes $C'=\{C^{i1}, C^{i2}, \ldots, C^{ik}\}$ must intersect, and produce the output pattern $Z^j=D^{i1} \vee D^{i2} \vee \ldots \vee D^{ik}$.

Theorem 6.1 If there exists an input vector which selects a subset L'' of L' and no other product terms, then the output D'' (which is the OR of the output parts of all product terms in L'') must equal Z^j.

Proof: Clearly $D'' \subseteq Z^j$. If $D'' \subset Z^j$, then $\mathcal{Z}$ would be ordered. □

Theorem 6.2 If there exists a subset L'' which intersects with some other set of product terms, L^α, then either a) $D'' \vee D^\alpha = Z^j$, or b) $D^\alpha \not\subset Z^j$ and $Z^j \not\subset D'' \vee D^\alpha$.

Proof: Clearly $D'' \subseteq Z^j$. Let $t=D'' \vee D^\alpha$. If $D''=Z^j$, then t must equal Z^j, to keep $\mathcal{Z}$ unordered (*i.e.* $D^\alpha \subseteq Z^j$). Consider $D'' \subset Z^j$. If $D^\alpha \subseteq Z^j$ then t must equal Z^j, otherwise we would have $t \subset Z^j$ and $\mathcal{Z}$ would be ordered. The case $D^\alpha \supset Z^j$ is not permitted because $t \supset Z^j$ would make $\mathcal{Z}$ ordered. If $D^\alpha \not\subset Z^j$, then $t \subset Z^j$ is impossible, and $Z^j \not\subset t$ must be true (condition b) otherwise $\mathcal{Z}$ would be ordered. □

As a consequence of the conditions imposed by the above theorems, product term sharing seems to be severely restricted. Consider the schematic example of this concept shown in Figure 6.4. The input vector x (marked with an asterisk) selects the three

product terms shown. Any other product terms which cover minterms associated with the output value 0011 (other than the one marked with an asterisk) must have output parts which are covered by 0011. Furthermore, minterms associated with output patterns 0001 and 0010 must be covered by other product terms which either cause such minterms to have value 0011 or to be unordered with respect to 0011.

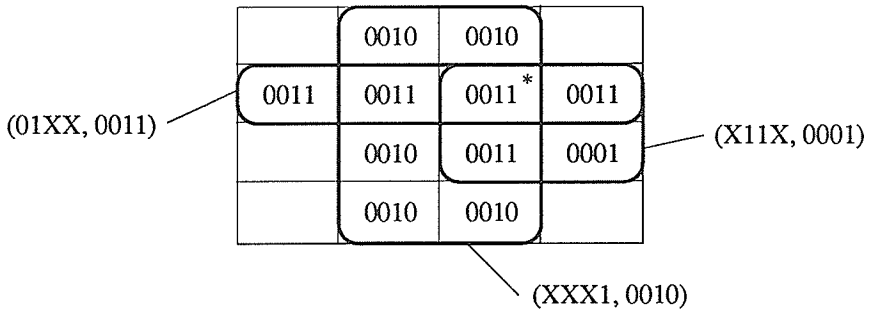
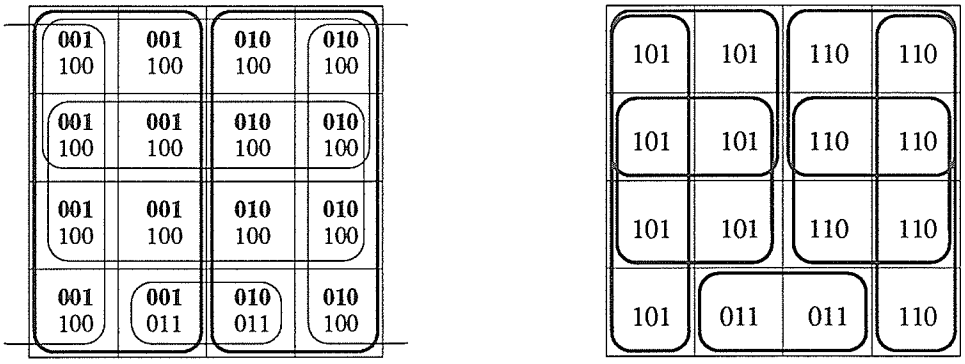


Figure 6.4 Example of product term sharing restrictions — partial covering.

Limited concurrency is surely an upper bound on the number of product terms required to realize a PLA with unordered outputs (the anomalies in Table 6.1, "dist" and "fsm2c", are likely due to the minimizer not finding a true minimal realization). The fact that for normal Berger encoding, 22 of the 29 sample PLA's met this bound can not be attributed to inadequacies of the minimizer. The remaining 7 PLA's have, on average, 4.59 times more product terms than corresponding naked realizations, while the limited concurrency forms for these PLA's require on average 1.15 times more product terms than the normal Berger realizations.

Figure 6.5a illustrates a small example function (unordered) which may be realized using fewer product terms than limited concurrency (Figure 6.5b). We conjecture that the conditions of Theorems 6.1 and 6.2 are sufficiently restrictive that it is unlikely that an arbitrary PLA satisfies them in such a way as to result in significantly fewer product terms than limited concurrency.



a) 6 term minimization b) 7 term limited concurrency

Figure 6.5 Example unordered function.

6.3 SUMMARY

The testing strategies proposed in chapters four and five were compared to other PLA checking schemes which do not require nonconcurrency. Schemes were evaluated in terms of the overhead contributions of: increased number of product terms, extra outputs and checker size. Checker delay was also considered. The actual area required for an encoded PLA and its checker may be affected by optimizations performed during mask layout, and is not considered in this dissertation.

The OR-*k* strategy fares well compared to schemes based on Berger codes. For some PLA's, limited concurrency of the OR-*k* strategy requires more product terms than a PLA encoded in a Berger code. The OR-*k* scheme uses fewer extra outputs in some instances, but in general, both schemes have numbers of extra outputs which are nearly minimal. Two types of Berger code checker were considered, one which is slow but small, while the other is fast but huge. An OR-*k* checker is faster than the latter and generally smaller than the former.

Empirical evidence and theoretical justification were advanced in support of the claim that any PLA which realizes an unordered set of output patterns will be as, or nearly as, expensive in terms of product terms as is limited concurrency (or nonconcurrency).

Chapter 7

CONCLUSION

This dissertation presents the results of an investigation into the on-line checking of programmable logic arrays. Computer tools which automate the logical minimization and physical layout of PLA structures, provide designers with the leverage necessary to achieve high productivity and quick turnaround for design revision in a fast paced market place. For this reason, array design techniques continue to be attractive to designers. As with integrated circuits in general, design for testability of PLA's is an important issue.

Several PLA design strategies geared towards off-line testing or BIST have appeared in the literature. Such structures make testing easier or faster at the expense of increased chip area and slower operating speed. Area overhead reported for the various techniques ranges from 15 to 100 percent, but an average of 30 or 40 percent is widely accepted as being typical. Besides the inconvenience of stopping normal system operation to perform testing, off-line testing strategies remain vulnerable to transient failures.

On-line checking techniques continuously monitor circuit behaviour for erroneous operation. Area overhead incurred is generally assumed to be greater than that for BIST. Results of the current investigation suggest that overhead for on-line checked PLA's is inherently function dependent regardless of the strategy employed. It is argued that existing on-line checking strategies suffer from one or more of the following deficiencies: inadequate fault model, poor error coverage, excessive area and/or delay overhead. Additionally, it may be argued that schemes based on the unidirectional error model† represent somewhat of an overkill, as a large proportion of errors that such schemes are capable of detecting are in fact unlikely to occur according to a comprehensive fault

† PLA's realized using the decoder design of Figure 2.2b produce only unidirectional errors.

model. Initial motivation for this dissertation stemmed from mod 3 residue and Berger encoded PLA's, and the feeling that the former might have poor error coverage and that checkers for the latter might be too expensive.

There exist several quantitative measures to gauge the effectiveness of checking strategies, such as mod 3, not designed to cover all errors produced by faults from a given fault model. A measure based on the concept of an error event (*EEDA*) was found to provide a useful new perspective. The proportion of faults causing only undetectable errors, ϕ_3^1, also appears to be a significant property which is totally obscured by the "averaging" type of measures proposed by others.

Error enumeration for a set of sample PLA's found that a significant percentage of the possible errors would go undetected by a mod 3 scheme, notably, up to 9 percent of modelled faults produced undetectable errors only. Furthermore, error event detection for the same PLA varied by up to 18 percent simply as a result of different assignments of numeric weights to the PLA outputs. Coverage in terms of canonic error patterns was found to be even more pessimistic. On the basis of several effectiveness measures, mod 3 was judged to be only slightly better than a parity check scheme. The measure indicating the greatest distinction between mod 3 and parity is *EEDA*. Determination of a quantitative procedure for ranking two checking schemes (or two different realizations of the same circuit under a given scheme) on the basis of a collection of effectiveness measures is a problem for further research.

Analytic investigation of the mod 3 scheme confirmed, unsurprisingly, that $2/3$ of all possible error events are detectable. Furthermore, $2/3$ of unidirectional errors also appear to be detectable, but since the class of unidirectional errors includes the subclass of single bit errors (which are 100 percent detectable by mod 3 codes), coverage of multi-bit unidirectional errors is actually less than $2/3$. Considering that the area overheads of mod 3 and Berger (which detects all unidirectional errors) encoded PLA's are comparable, one may conclude that the mod 3 code is not appropriate for checking PLA's.

Ideally, one strives to design an on-line checking scheme which is in some way optimal for the circuit it checks. Therefore, careful characterization of a PLA's erroneous behaviour under a comprehensive fault model was undertaken. The important concept of covered faults was introduced, which, along with the well known concept of fault equivalency, allows one to classify modelled faults so that only a reduced subset of the faults need be enumerated to obtain an error characterization. Specifically, if a checking scheme is capable of detecting all errors due to single stuck-at faults and line break faults (on bit and product lines), then all errors due to any single fault from the fault model are detectable, including crosspoint and adjacent line bridging faults, which others have shown to correspond to observed defects.

One result stemming from the availability of an error characterization for specific PLA's, is a procedure for determining the minimal number of check bits required to ensure detection of all error events which could arise due to modelled faults. Although the procedure is computationally expensive and impractical for large circuits, it is nevertheless useful to apply it on smaller PLA's to see how close various checking strategies come to the optimal.

Another consequence of performing the analysis required to derive an error characterization, is that for nonconcurrent PLA's the erroneous behaviour may be expressed as a succinct error model. Specifically, modelled faults cause three kinds of error: a single-bit error, an error where one or more (perhaps all) rightmost outputs are zero, and an error of type ξ (the bitwise OR of two codewords).

Two new on-line checking strategies based on the nonconcurrent error model are proposed. Both have a common base consisting of parity checked outputs (to detect single-bit errors), and arranging that the rightmost two outputs are encoded in a 1-out-of-2 code (to detect rightmost outputs being held at zero). These strategies are called Simple Parity and OR-*k*.

The Simple Parity scheme demonstrates that a parity checker (which is a part of the common base) can also detect errors of type ξ if check bits are defined appropriately. One advantage of this approach is that the check bit definition procedure is based solely on the range of the function which the PLA implements. The problem transforms to one of graph decomposition in terms of a novel edge-wise exclusive-OR operator. A heuristic algorithm to perform this decomposition was presented, and it appears to produce minimal or near-minimal solutions. Potential future enhancements to this algorithm may reduce its computational complexity.

However, it may be necessary to, as much as, double the number of PLA outputs and include several of the product term lines (as pseudo-outputs) in the parity check to achieve the capability of detecting errors of type ξ. Doubling the number of outputs is still less expensive than duplicating the entire PLA. Compared to other nonconcurrent PLA checking schemes which use a 1-out-of-p or parity check on all p product term lines, Simple Parity represents an improvement in checker size. A graph partitioning technique is suggested to take advantage of the error detection afforded by the 1-out-of-2 code, also in the common base. Results in terms of reduced number of check bits appear promising, but the as yet manual (intuitive) partitioning procedure needs to be formalized into an effective automated algorithm. All on-line checking strategies considered in this dissertation are based on a single fault model, but it was observed that Simple Parity provides poor coverage of multiple faults. The main contribution of this checking scheme is that it can be done.

Rather than attempting to verify that an output pattern belongs to a particular unordered code, the OR-k strategy is designed to recognize errors of type ξ. In fact, all multiple faults causing such errors are detectable. The other errors in the model are covered by a two-rail parity check. Any nonconcurrent PLA with unordered outputs is OR-k testable. A systematic methodology for defining check bits to extend an arbitrary set of binary vectors (a PLA's original set of outputs) so that it is unordered was

presented. For several sample PLA's, this procedure obtained slightly fewer check bits than required by modified Berger encoding, however both OR-k and Berger encoded strategies result in nearly optimal number of check bits, as discussed previously. For most PLA's, a minimal number of check bits may be assigned values in many ways. Three methods of performing such assignments were performed experimentally on all sample PLA's. Each of these methods was superior for some of the cases. More work is required to attempt to design an assignment process which minimizes checker overhead.

A function dependent OR-k checker is straightforwardly constructed from the set of unordered output patterns. The method is most effective if the number of such patterns is small; this appears to be the case for many industrial PLA's. An OR-k checker is a positive function, thus allowing an array construction with the PLA outputs forming checker inputs. Because it is a positive function, no negated literals are required and hence no input decoders. A variation which frees the checker from having to detect errors which happen to be detected by the two-rail parity check, results in a significant reduction in checker size. Experimental evidence indicates that OR-k checker overhead is generally less (considerably less in the case of recent "fast" Berger checkers) than that of checkers for concurrent PLA's augmented to produce Berger or mod 3 encoded outputs. Checker delay is essentially that of the base parity check, which is nearly four times faster than Berger or mod 3 checkers.

Minimizing a multiple-output PLA under the 1-concurrent constraint (*i.e.* to achieve nonconcurrency) appears to be a computationally expensive process. The first step is to partition the input space into groups of minterms or cubes, each group associated with a distinct output pattern. This step is relatively easy, and an algorithm is presented (Algorithm 4.4). The next step is to treat each group as a single output function and find a minimal 1-concurrent realization for it. A reasonably effective heuristic driven algorithm to do this is proposed (Algorithm 4.5), however, since it operates in terms of

minterms, it is applicable only to PLA's which have a small number of input variables. Another algorithm which operates in terms of cubes is a potential topic for future work.

Limited concurrency is much easier to achieve. Simply use a conventional procedure to minimize each group obtained via Algorithm 4.4. Limited concurrency turns out to be an important concept in many ways. Error event enumeration is a simple function of the set of output patterns. Besides being less expensive to compute, limited concurrency may require fewer product terms than 1-concurrency; in fact the number of product terms required for limited concurrency is a lower bound on 1-concurrency. Limited concurrency is also an upper bound on the number of product terms required for a concurrent realization of a PLA with unordered outputs. For many PLA's considered, both 1-concurrency and Berger encoding identically meet their respective bounds. PLA's augmented to produce unordered outputs, and minimized conventionally, were found to require nearly as many product terms as nonconcurrent PLA's. Some PLA's have limited concurrency representations which are inherently much larger than that found by conventional minimization. Such PLA's are expensive to check regardless of whether unidirectional error detecting codes or nonconcurrent techniques are used. When a large number of product terms are required, techniques which use a checker on the product term lines should be avoided.

The chip area cost estimators of Chapter 5 and gate counts of Chapter 6 provide only approximate overhead figures. More accurate results require the construction of an actual mask layout. This would take into account routing and floorplanning of tree-like checker structures. This is a consideration for a later time.

The main contribution of this dissertation is the OR-k testing strategy. An OR-k testable PLA was shown to be strongly fault secure if realized using limited concurrency. The OR-k testing scheme results in PLA overheads comparable to that of Berger encoded PLA's, however the OR-k checker is smaller and faster.

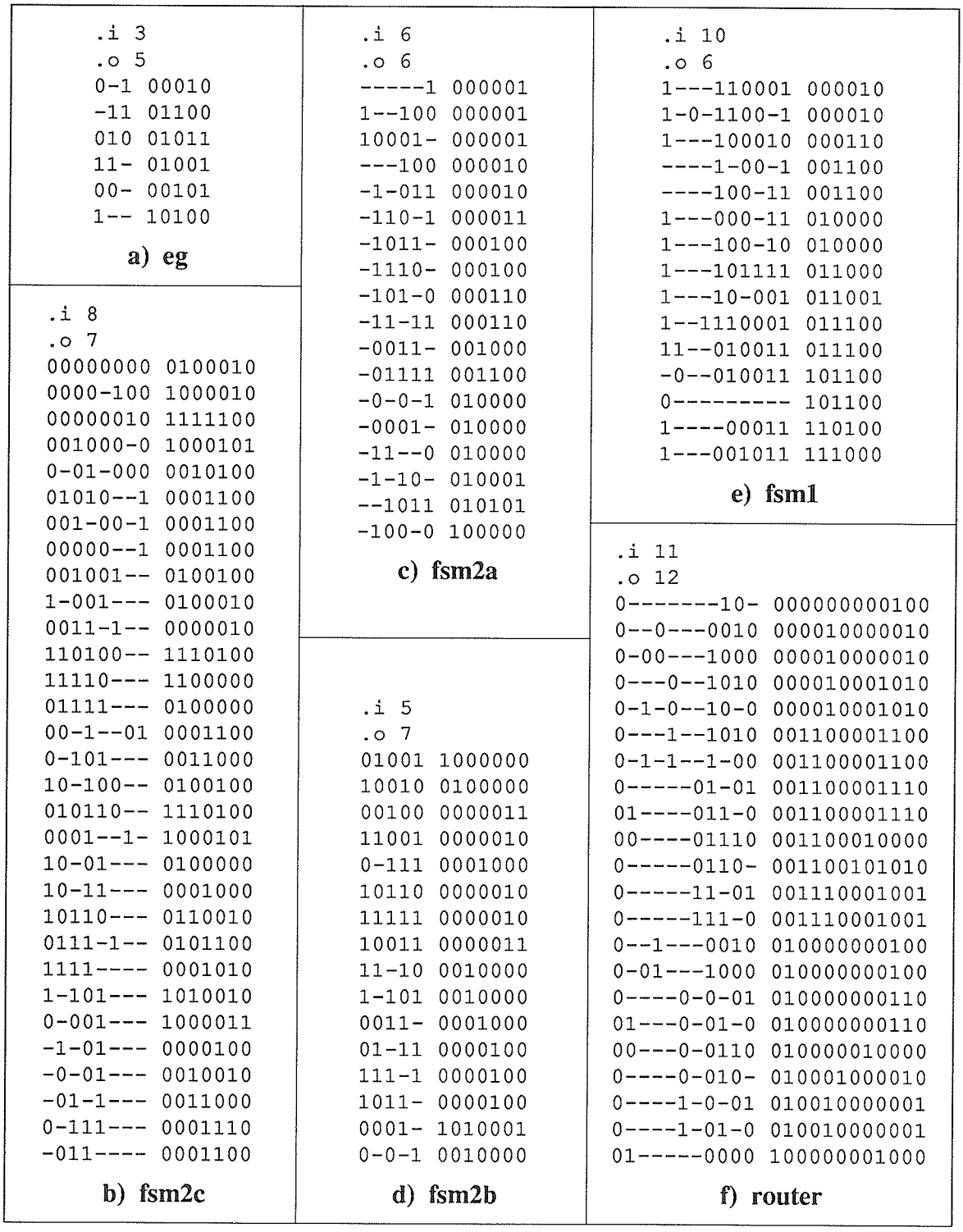
Appendix A

SAMPLE PLA'S

Table A.1 lists the sample PLA's used in this dissertation (see Figure A.1 also).

PLA	*m*	*n*	*N*	C/D	Source	Comment
all4	4	4	16			hypothetical PLA's which produce all 2^n possible output patterns (see Table 4.1)
all5	5	5	32			
all6	6	6	64			
all7	7	7	128			
all8	8	8	256			
alu1	12	8	80	D	E	E – ESPRESSO test suite [Bray84]
alu2	10	8	67	D	E	D – data path logic
apla	10	12	37	C	E	C – control path logic
bis1	4	3	5		M	M – McBOOLE distribution tape [Dage86]
bis2	4	3	4		M	bis2 is the PLA in Figure 1.2
bis5	4	4	8		M	
bis6	4	3	5		M	
bw	5	28	22		F	F – FSM benchmarks [MCNC87]
dc1	4	7	10	C	E	
dc2	8	7	91	C	E	
dist	8	5	21	D	E	
dk17	10	11	24	C	E	
dk27	9	9	38	C	E	
eg	3	5	6		Fig. A.1a	toy example introduced in section 5.1.3
f2	4	4	10		F	
f51m	8	8	255	D	E	
fsm1	10	6	13	C	Fig. A.1e	set of PLA's designed by the author that are used in the construction of two finite state machines which form part of a Video Display Controller
fsm2a	6	6	14	C	Fig. A.1c	
fsm2b	5	7	11	C	Fig. A.1d	
fsm2c	8	7	24	C	Fig. A.1b	
gary	15	11	69	C	E	
misex1	8	7	10		F	
misex2	25	18	34		F	
rd53	5	3	5	D	E	
router	11	12	15	C	Fig. A.1f	part of a multiprocessor interconnection switch
sao2	10	4	9		F	
traffic	5	7	8	C		traffic light controller [Mead80]
vg2	25	8	23	C	E	
wim	4	7	11	C	E	
z4	7	4	15	C	E	
5xp1	7	10	128	D	E	

Table A.1 Sample PLA's.



```
.i 3
.o 5
0-1 00010
-11 01100
010 01011
11- 01001
00- 00101
1-- 10100
```

a) eg

```
.i 8
.o 7
00000000 0100010
0000-100 1000010
00000010 1111100
001000-0 1000101
0-01-000 0010100
01010--1 0001100
001-00-1 0001100
00000--1 0001100
001001-- 0100100
1-001--- 0100010
0011-1-- 0000010
110100-- 1110100
11110--- 1100000
01111--- 0100000
00-1--01 0001100
0-101--- 0011000
10-100-- 0100100
010110-- 1110100
0001--1- 1000101
10-01--- 0100000
10-11--- 0001000
10110--- 0110010
0111-1-- 0101100
1111---- 0001010
1-101--- 1010010
0-001--- 1000011
-1-01--- 0000100
-0-01--- 0010010
-01-1--- 0011000
0-111--- 0001110
-011---- 0001100
```

b) fsm2c

```
.i 6
.o 6
-----1 000001
1--100 000001
10001- 000001
---100 000010
-1-011 000010
-110-1 000011
-1011- 000100
-1110- 000100
-101-0 000110
-11-11 000110
-0011- 001000
-01111 001100
-0-0-1 010000
-0001- 010000
-11--0 010000
-1-10- 010001
--1011 010101
-100-0 100000
```

c) fsm2a

```
.i 5
.o 7
01001 1000000
10010 0100000
00100 0000011
11001 0000010
0-111 0001000
10110 0000010
11111 0000010
10011 0000011
11-10 0010000
1-101 0010000
0011- 0001000
01-11 0000100
111-1 0000100
1011- 0000100
0001- 1010001
0-0-1 0010000
```

d) fsm2b

```
.i 10
.o 6
1---110001 000010
1-0-1100-1 000010
1---100010 000110
----1-00-1 001100
----100-11 001100
1---000-11 010000
1---100-10 010000
1---101111 011000
1---10-001 011001
1--1110001 011100
11--010011 011100
-0--010011 101100
0--------- 101100
1----00011 110100
1---001011 111000
```

e) fsm1

```
.i 11
.o 12
0-------10- 000000000100
0--0---0010 000010000010
0-00---1000 000010000010
0---0--1010 000010001010
0-1-0--10-0 000010001010
0---1--1010 001100001100
0-1-1--1-00 001100001100
0-----01-01 001100001110
01----011-0 001100001110
00----01110 001100010000
0-----0110- 001100101010
0-----11-01 001110001001
0-----111-0 001110001001
0--1---0010 010000000100
0-01---1000 010000000100
0----0-0-01 010000000110
01---0-01-0 010000000110
00---0-0110 010000010000
0----0-010- 010001000010
0----1-0-01 010010000001
0----1-01-0 010010000001
01-----0000 100000001000
```

f) router

Figure A.1 ESPRESSO specifications for PLA's introduced in this dissertation.

Appendix B

MOD 3 CANONIC ERROR ANALYSIS

This appendix gives proofs for two Lemmata, and provides detailed derivations of equations first presented in section 3.2.2.2. The notation from section 3.2.2.2 is used.

Lemma B.1 Each *BWA* detects $2/3$ of all directed error patterns.

Proof: An error pattern, E, is detected if $e=|\sum e_i w_i|_3$ is non-zero. Let $e_{1(2)}$ denote the mod 3 sum of all e_i with associated weight w_i=1(2). Hence, $e=|e_1+2e_2|_3$. For e to be 0, we must have $e_1=e_2$ mod 3. Since $e_{1(2)}$ represents the mod 3 sum of an arbitrary combination of $n/2$ digits from {0, 1, 2}, it will have value 0, 1 or 2 equally often over all possible combinations of digits†. Since the error pattern digits associated with e_1 and e_2 are independent of each other, $e_1=e_2=0$ for $1/9$ of all n-digit error patterns; likewise for $e_1=e_2=1$ and $e_1=e_2=2$. Thus $e_1=e_2$ mod 3, for $1/3$ of the error patterns, and it follows that each *BWA* detects $2/3$ of the error patterns. □

Lemma B.2 No two noncomplementary *BWA*'s detect the same set of error patterns.

Proof: For any two *BWA*'s to be distinct, at least one pair of weights must be exchanged, and at least one weight must be the same. Without loss of generality, assume that BWA_1=121... and BWA_2=211..., where the unspecified weights may be arranged arbitrarily and distinctly for BWA_1 and BWA_2. The 3-bit error pattern 2110...0 (1210...0) is detected only by BWA_1 (BWA_2). □

† This is easily proved using induction.

Derivation of (3.13)

$$SCsize = \frac{\binom{n}{n_{max}}\binom{n-n_{max}}{n_{min}}}{r!} \tag{3.13}$$

Let a structural class be abstractly represented as a pattern made up of symbols from $\{\alpha, \beta, \gamma\}$ which are mapped onto $\{a, b, c\}$ to form canonic error patterns, where n_α, n_β, and n_γ denote the number of times each symbol appears in the abstract representation and $n_\alpha \geq n_\beta \geq n_\gamma$. Assume that $n_\alpha = n_{max}$ and $n_\gamma = n_{min}$. Clearly there are $\binom{n}{n_{max}}\binom{n-n_{max}}{n_{min}}$ distinct ways of arranging the abstract symbols to form patterns with n_α α's, n_β β's and n_γ γ's. Each of these abstract patterns corresponds to exactly one canonic error pattern (*eg.* $\beta\alpha\alpha\beta\gamma\alpha\alpha\gamma \rightarrow abbacbbc$), but some of the abstract patterns produce the same *CEP* (*eg.* $\gamma\alpha\alpha\gamma\beta\alpha\alpha\beta \rightarrow abbacbbc$). The situations which produce the same *CEP* are as follows, where r is the maximum number of identical non-zero values contained in the triple $(n_\alpha, n_\beta, n_\gamma)$:

n_α	n_β	n_γ	r	description
n	0	0	1	$\alpha\alpha\alpha\ldots\alpha \rightarrow aaa\ldots a$
$n-k$	k	0	1	$k \neq n/2$; no two abstract patterns have same *CEP*
k	k	0	2	$k = n/2$; exchanging all $\alpha \leftrightarrow \beta$ produces same *CEP*
$n-k-j$	k	j	1	$k \neq j$, $k \neq n-j-k$; no two abstract patterns have same *CEP*
$n-2k$	k	k	2	$k \neq n/3$; exchanging all $\beta \leftrightarrow \gamma$ produces same *CEP*
k	k	$n-2k$	2	$k \neq n/3$; exchanging all $\alpha \leftrightarrow \beta$ produces same *CEP*
k	k	k	3	$k = n/3$; exchanging any pair of $\{\alpha, \beta, \gamma\}$ produces same *CEP*

Therefore, $\binom{n}{n_{max}}\binom{n-n_{max}}{n_{min}}$ divided by $r!$ produces the desired result.

Derivation of (3.14) and (3.15)

$$\begin{aligned} S = \{\ (x,y): \quad & x+2y \neq 2A+B \bmod 3, \\ & 0 \le x \le A, \\ & 0 \le y \le B, \\ & x+y \le n/2, \\ & (A+B)-(x+y) \le n/2\ \} \end{aligned} \tag{3.14}$$

$$CCCsize\langle A,B\rangle = \frac{1}{2}\sum_{i=1}^{|S|} \binom{A}{x_i}\binom{B}{y_i}\binom{n-A-B}{n/2-x_i-y_i} \tag{3.15}$$

A structural class represents all error patterns with n_α α's, n_β β's and n_γ γ's, and symbols $\{\alpha, \beta, \gamma\}$ may be mapped onto $\{0, 1, 2\}$ in all possible ways. Without loss of generality, consider the error pattern $E = 000\ldots0111\ldots1222\ldots2$ from the structural class. For any BWA (with equal numbers of 1's and 2's) we have Figure B.1.

$$\begin{array}{ll} E & 00000\,00\ldots0\ 11111\,11\ldots1\ 22222\,22\ldots2 \\ BWA & \underbrace{11\ldots1\,22\ldots2}_{n_{01}\ \ n_{02}}\ \underbrace{11\ldots1\,22\ldots2}_{n_{11}=x\ \ n_{12}}\ \underbrace{11\ldots1\,22\ldots2}_{n_{21}=y\ \ n_{22}} \\ & \underbrace{\qquad\qquad}_{n_0}\qquad\underbrace{\qquad\qquad}_{n_1=A}\qquad\underbrace{\qquad\qquad}_{n_2=B} \end{array}$$

Figure B.1 Relationship between error pattern E and a BWA.

The following identities also hold:

$$n_{01} + n_{11} + n_{21} = n/2 \tag{B.1}$$

$$n_{02} + n_{12} + n_{22} = n/2 \tag{B.2}$$

Alternatively, these identities may be expressed as inequalities:

$$n_{11} + n_{21} \le n/2 \tag{B.3}$$

$$n_{12} + n_{22} \le n/2 \tag{B.4}$$

Consider the variable substitutions: x for n_{11}, y for n_{21}, A for n_1, and B for n_2, and note that $n_{22} = B-y$, and $n_{12} = A-x$. Equation B.3 becomes $x+y \leq n/2$, and equation B.4 becomes $(A+B) - (x+y) \leq n/2$. Clearly, $0 \leq x \leq A$, and $0 \leq y \leq B$ must also hold. We can redefine $e=|\sum e_i w_i|_3$ in terms of these variables, *i.e.* equation B.5.

$$\begin{aligned} e &= | n_{11} + 2n_{12} + 2n_{21} + n_{22} |_3 \\ &= | x + 2(A-x) + 2y + B-y |_3 \\ &= | 2A + B - x + y |_3 \\ &= | (2A + B) - (x + 2y) |_3 \end{aligned} \tag{B.5}$$

To obtain $e \neq 0$, the inequality: $x+2y \neq 2A+B \bmod 3$, must be satisfied. For structural class $\langle A,B \rangle$, the set of solutions for x, y (given by equation 3.14) specifies the *BWA* configurations for which $e \neq 0$, *i.e.* the *BWA*'s which detect canonic error patterns belonging to the structural class.

For a particular solution, (x_i, y_i), the bit weights within partitions n_0, n_1 and n_2 may be arranged arbitrarily, thus there are $\binom{A}{x_i}\binom{B}{y_i}\binom{n-A-B}{n/2-x_i-y_i}$ arrangements. This number includes complementary *BWA*'s, so only half are distinct. Equation 3.15 thus specifies the total number of *BWA*'s which detect errors belonging to structural class $\langle A,B \rangle$.

Derivation of (3.16)

$$CEP detected\langle A,B \rangle = \frac{1}{r!}\sum_{i=1}^{|S|}\binom{n/2}{x_i}\binom{n/2-x_i}{y_i}\binom{n/2}{A-x_i}\binom{n/2-(A-x_i)}{B-y_i} \tag{3.16}$$

Determination of the number of *CEP*'s detected by a *BWA* involves a similar counting argument to that for equation 3.15. Figure B.1 (rearranged as Figure B.2) and equation 3.14 still apply, but now we count the number of ways that the 0's, 1's and 2's can be arranged within w_1 and w_2.

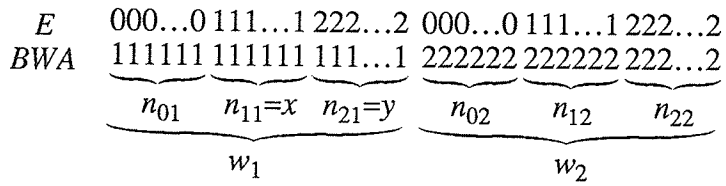


Figure B.2 Another view of the relationship between error pattern E and a BWA.

Note that $w_1 = w_2 = n/2$, $n_{12} = A-x$, and $n_{22} = B-y$. The number of ways that the error bits associated with w_1 can be arranged, for a given (x_i, y_i), is $\binom{n/2}{x_i}\binom{n/2-x_i}{y_i}$, and that for w_2 is $\binom{n/2}{A-x_i}\binom{n/2-(A-x_i)}{B-y_i}$. Each of these arrangements corresponds to exactly one canonic error pattern, but some produce the same CEP. The factor $r!$ accounts for this in a manner similar to that described for equation 3.13. Equation 3.16 specifies the total number of errors, from structural class $\langle A,B \rangle$, which are detected by each BWA.

Derivation of (3.17)

$$SCDA = \frac{CEP detected}{SCsize} = \frac{CCCsize}{\#BWA} \tag{3.17}$$

Equation 3.17 follows from a straightforward, but tedious, manipulation of equations 3.13, 3.15, 3.16 and the expression $\frac{1}{2}\binom{n}{n/2}$ representing the number of BWA's.

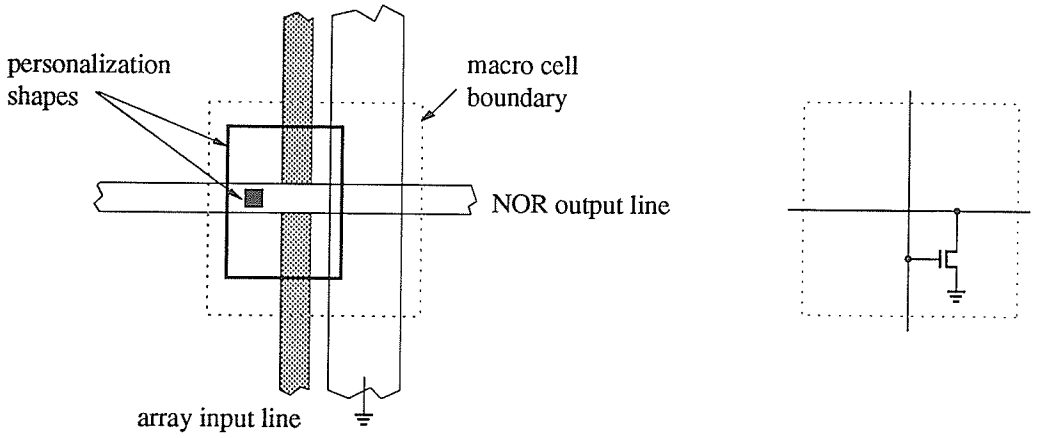
Appendix C

PLA AREA COST CALCULATION

This appendix provides detailed derivation of equation 5.6, first presented in section 5.2.4.3, which is a PLA area estimator based on actual PLA layout parameters from [Bosw85a].

$$\begin{aligned} Cost = \; & 64\,[(2m + n + n_c)p + (n + n_c)T] + \\ & 484p + 128m + (n + n_c)(176a + 308) + 548T + 308a \end{aligned} \tag{5.6}$$

A PLA is usually constructed as a regular arrangement of "macro cells". Only a small number of different cell types are necessary. Most of the cells making up a PLA are crosspoint cells. A typical layout for such a cell is shown in Figure C.1a ([Dill88]). Two variations of a crosspoint cell are used. If a crosspoint transistor is desired, then the "personalization shapes" are included, otherwise they are omitted. The physical layout shown in Figure C.1a corresponds to the transistor diagram of Figure C.1b.



a) Crosspoint cell layout

b) Transistor diagram

Figure C.1 Crosspoint macro cell.

The set of macro cells actually used to realize a PLA, and their corresponding dimensions, are technology dependent. Boswell [Bosw85a] presented the details for one particular PLA implementation. His PLA model is shown in Figure C.2, and the corresponding macro cell parameters are listed in Table C.1. Note the absence of input decoders and output inverters.

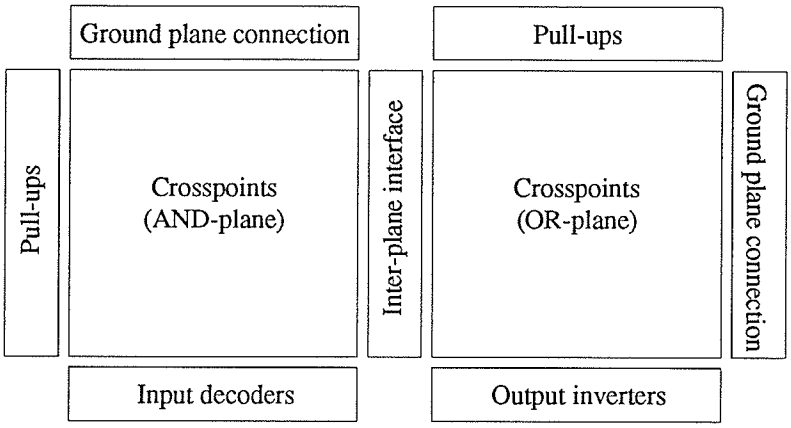


Figure C.2 PLA layout according to macro cell type.

Macro Cell Type	Cell Size	Number of Cells
Crosspoint	64	$2mp+np$
Pull-up	308	$p+n$
Ground plane connection	64	$2m+p$
Inter-plane interface	112	p
Input decoder	—	m
Output inverter	—	n

Table C.1 PLA macro cell dimensions from [Bosw85a].

Figure C.2 and Table C.1 correspond to a naked PLA. The area of cells represented by Table C.1 may be seen to be expressed by equation 5.6 with n_c=T=a=0.

An OR-k checker basically must compute T test terms (product terms) from $n+n_C$ PLA outputs and find the OR of all test terms. This is essentially another array with $n+n_C$ inputs, T product term lines, and one output. Since the checker realizes a positive function, input decoders are not required. The same set of macro cells that are used in the main PLA may be used to realize an OR-k checker, as shown in Figure C.3. The macro cells required to realize such a checker are summarized in Table C.2.

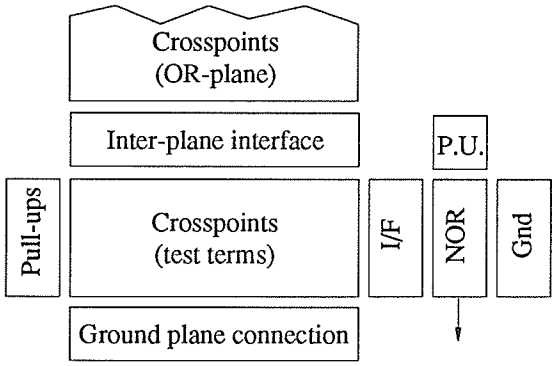


Figure C.3 OR-k checker layout according to macro cell type.

Macro Cell Type	Cell Size	Number of Cells
Crosspoint	64	$(n+n_C+1)T$
Pull-up	308	$T+1$
Ground plane connection	64	$n+n_C+T$
Inter-plane interface	112	$n+n_C+T$

Table C.2 OR-k checker macro cell requirements.

Equation 5.6 may be seen to account for all macro cells listed in Tables C.1 and C.2. This equation may be used to compute the cost of either a naked PLA or an augmented one (a=1). Terms in equation 5.6 which involve the variable "a" correspond to cells which exist only in the augmented PLA (and which do not vanish by setting n_C=T=0).

REFERENCES

1. [Abra86] J.A. Abraham, "Fault and Error Models for VLSI," *Proceedings of the IEEE*, **74(5)** (1986), 639-654.

2. [Agar80] V.K. Agarwal, "Multiple Fault Detection in Programmable Logic Arrays," *IEEE Transactions on Computers*, **C-29(6)** (1980), 518-522.

3. [Agar86] V.K. Agarwal, "Easily Testable PLA Design," in *VLSI Testing*, T.W. Williams *ed.*, North Holland, Amsterdam, The Netherlands, 1986, 65-93.

4. [Ande73] D.A. Anderson & G. Metze, "Design of Totally Self-checking Circuits for *m*-out-of-*n* Codes," *IEEE Transactions on Computers*, **C-22(3)** (1973), 263-269.

5. [Ashj77] M.J. Ashjaee & S.M. Reddy, "On Totally Self-Checking Check Circuits for Separable Codes," *IEEE Transactions on Computers*, **C–26** (1977), 737-744.

6. [Bane82] P. Banerjee & J.A. Abraham, "Fault Characterization of VLSI MOS Circuits," *IEEE International Conference on Circuits and Computers*, **ICCC-82** (1982), 564-568.

7. [Bane84] P. Banerjee & J.A. Abraham, "Characterization and Testing of Physical Failures in MOS Logic Circuits," *IEEE Design & Test of Computers*, (1984), 76-86.

8. [Berg61] J.M. Berger, "A Note on Error Detecting Codes for Asymmetric Channels," *Information and Control*, **4** (1961), 68-73.

9. [Bisw85] N.N. Biswas & J. Jacop, "A Testable PLA Design with Minimal Hardware and Test Set," *International Test Conference 1985 Proceedings*, (1985), 583-588.

10. [Blau88] M. Blaum, "Systematic Unidirectional Burst Detecting Codes," *IEEE Transactions on Computers*, **C-37(4)** (1988), 453-457.

11. [Bord82] J.M. Borden, "Optimal Asymmetric Error Detecting Codes," *Information and Control*, **53(1)** (1982), 66-73.

12. [Bose81] B. Bose, "On Systematic *SEC-MUED* Code," *Digest of Papers 11th International Symposium on Fault-Tolerant Computing*, **FTCS-11** (1981), 265-267.

13. [Bose82a] P. Bose & J.A. Abraham, "Test Generation for Programmable Logic Arrays," *Proceedings 19th ACM IEEE Design Automation Conference*, (1982), 574-580.

14. [Bose82b] B. Bose & D.K. Pradhan, "Optimal Unidirectional Error Correcting/Detecting Codes," *IEEE Transactions on Computers*, **C-31(6)** (1982), 564-568.

15. [Bose82c] B. Bose & T.R.N. Rao, "Theory of Unidirectional Error Correcting/Detecting Codes," *IEEE Transactions on Computers*, **C-31(6)** (1982), 521-530.

16. [Bose84a] B. Bose & D.J. Lin, "PLA Implementation of *k*-out-of-*n* Code TSC Checker," *IEEE Transactions on Computers*, **C-33(6)** (1984), 583-588.

17. [Bose84b] B. Bose & D.J. Lin, "Systematic Unidirectional Error Detecting Codes," *Digest of Papers 14th International Symposium on Fault-Tolerant Computing*, **FTCS-14** (1984), 94-99.

18. [Bose85] B. Bose & D.J. Lin, "Systematic Unidirectional Error-Detecting Codes," *IEEE Transactions on Computers*, **C-34(11)** (1985), 1026-1032.

19. [Bose86a] B. Bose, "Burst Unidirectional Error-Detecting Codes," *IEEE Transactions on Computers*, **C-35(4)** (1986), 350-353.

20. [Bose86b] B. Bose & J. Metzner, "Coding Theory for Fault-Tolerant Systems," in *FAULT-TOLERANT COMPUTING — Theory and Techniques*, D.K. Pradhan *ed.*, Prentice-Hall, Englewood Cliffs, NJ., 1986, 265-335.

21. [Bose87] B. Bose, "On Unordered Codes," *Digest of Papers 17th International Symposium on Fault-Tolerant Computing*, **FTCS-17** (1987), 102-107.

22. [Bosw85a] C. Boswell, "A Comparative Study of Testable Design of Programmable Logic Arrays," M.E. Thesis, University of Newcastle, N.S.W. Australia, 1985.

23. [Bosw85b] C. Boswell, K.K. Saluja & K. Kinoshita, "Design of Programmable Logic Arrays for Parallel Testing," *Journal of Computer System Science and Engineering*, **1(1)** (1985), 5-16.

24. [Bozo84] S. Bozorgui-Nesbat & E.J. McCluskey, "Lower Overhead Design for Testability of Programmable Logic arrays," *International Test Conference 1984 Proceedings*, (1984), 856-865.

25. [Bozo86] S. Bozorgui-Nesbat & E.J. McCluskey, "Lower Overhead Design for Testability of Programmable Logic Arrays," *IEEE Transactions on Computers*, **C-35(4)** (1986), 379-383.

26. [Bray84] R.K. Brayton, G.D. Hachtel, C.T. McMullen & A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

27. [Breu85] M. Breuer & X. Zhu, "A Knowledge-Based System for Selecting a Test Methodology for a PLA," *Proceedings 22nd ACM IEEE Design Automation Conference*, (1985), 259-265.

28. [Cart68] W.C. Carter & P.R. Schneider, "Design of Dynamically Checked Computer," *Proceedings of the 4th IFIPS Congress*, **vol. 2** (1968), 878-883.

29. [Cha78] C.W. Cha, "A Testing Strategy for PLAs," *Proceeding of the 15th Design Automation Conference*, (1978), 326-334.

30. [Chen85] C.Y. Chen, W.K. Fuchs & J.A. Abraham, "Efficient Concurrent Error Detection in PLAs and ROMs," *Proceedings 1985 IEEE International Conference on Computer Design*, (1985), 525-529.

31. [Daeh81] W. Daehn & J. Mucha, "A Hardware Approach to Self-Testing of Large Programmable Logic Arrays," *IEEE Transactions on Circuits and Systems*, **CAS-28(11)** (1981), 1033-1037.

32. [Dage86] M.R. Dagenais, V.K. Agarwal & N.C. Rumin, "McBOOLE: A New Procedure for Exact Logic Minimization," *IEEE Transactions on Computer-Aided Design*, **CAD-5(1)** (1986), 229-238.

33. [Diet78] D.L. Dietmeyer, *Logic Design of Digital Systems*, Allyn and Bacon, Inc., Boston, Mass., 1978.

34. [Dill88] T.E. Dillinger, *VLSI Engineering*, Prentice-Hall, Englewood Cliffs, NJ., 1988.

35. [Dong82] H. Dong, "Modified Berger Codes for Detection of Unidirectional Errors," *Digest of Papers 12th International Symposium on Fault-Tolerant Computing*, **FTCS-12** (1982), 317-320.

36. [Eich80] E.B. Eichelberger & E. Lindbloom, "A Heuristic Test Pattern Generator for Programmable Logic Arrays," *IBM Journal of Research and Development*, **24** (1980), 15-22.

37. [Eich83] E.B. Eichelberger & E. Lindbloom, "Random Pattern Coverage and Diagnosis for LSSD Logic Self-Test," *IBM Journal of Research and Development*, **27** (1983), 265-272.

38. [Fitz89] K. Fitzgerald, "Design for Testability Arrives," *The Institute, A news supplement to IEEE Spectrum,* November, 1989, pp. 1 & 6.

39. [Flei75] H. Fleisher & L.I. Maissel, "An Introduction to Array Logic," *IBM Journal of Research and Development*, **19** (1975), 98-109.

40. [Frei62] C.V. Freiman, "Optimal Error Detecting Codes for Completely Asymmetric Binary Channels," *Information and Control*, **5** (1962), 64-71.

41. [Fuch84] W.K. Fuchs & J.A. Abraham, "A Unified Approach to PLA's with Concurrent Error Detection in Highly Structured Logic Arrays," *Digest of Papers 14th International Symposium on Fault-Tolerant Computing*, **FTCS-14** (1984), 4-9.

42. [Fuch87] W.K. Fuchs, C.-Y. Chen & J.A. Abraham, "Concurrent Error Detection in Highly Structured Logic Arrays," *IEEE Journal of Solid-State Circuits*, **SC-22(4)** (1987), 583-594.

43. [Fuji80] H. Fujiwara, K. Kinoshita & H. Ozaki, "Universal test sets for programmable logic arrays," *Digest of Papers 10th International Symposium on Fault-Tolerant Computing*, **FTCS-10** (1980), 137-142.

44. [Fuji81] H. Fujiwara & K. Kinoshita, "A Design of Programmable Logic Arrays with Universal Tests," *IEEE Transactions on Circuits and Systems*, **CAS-28(11)** (1981), 1027-1032.

45. [Fuji83] E. Fujiwara, "A Self-Testing Group-Parity Prediction Checker and Its Use for Built-In Testing," *Digest of Papers 13th International Symposium on Fault-Tolerant Computing*, **FTCS-13** (1983), 146-153.

46. [Fuji84a] E. Fujiwara, N. Mutoh & K. Matsuoka, "A Self-Testing Group-Parity Prediction Checker and Its Use for Built-In Testing," *IEEE Transactions on Computers*, **C-33(6)** (1984), 578-583.

47. [Fuji84b] H. Fujiwara, "A New PLA Design for Universal Testability," *IEEE Transactions on Computers*, **C-33(8)** (1984), 745-750.

48. [Fuji85] E. Fujiwara & K. Matsuoka, "A Totally Self-Checking Generalized Prediction Checker and Its Use for Built-In Testing," *Digest of Papers 15th International Symposium on Fault-Tolerant Computing*, **FTCS–15** (1985), 384-389.

49. [Fuji87] E. Fujiwara & K. Matsuoka, "A Self-Checking Generalized Prediction Checker and Its Use for Built-In Testing," *IEEE Transactions on Computers*, **C-36(1)** (1987), 86-93.

50. [Fuji88a] H. Fujiwara, "Design of PLA's with Random Pattern Testability," *IEEE Transactions on Computer-Aided Design*, **CAD-7** (1988), 5-10.

51. [Fuji88b] H. Fujiwara, O. Fujisawa & K. Hikone, "Enhancing Random-Pattern Coverage of Programmable Logic Arrays via Masking Technique," *International Test Conference 1988 Proceedings*, (1988), 642-648.

52. [Gait85a] N. Gaitanis, "Totally Self-Checking Checkers for Low-Cost Arithmetic Codes," *IEEE Transactions on Computers*, **C-34(7)** (1985), 596-601.

53. [Gait85b] N. Gaitanis, "A Totally Self-Checking Error Indicator," *IEEE Transactions on Computers*, **C-34(8)** (1985), 758-761.

54. [Gali80] J. Galiay, Y. Crouzet & M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers*, **C-29(6)** (1980), 527-531.

55. [Gare79] M.R. Garey & D.S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W.H. Freeman, San Fransisco, 1979.

56. [Gels86] P.P. Gelsinger, "Built in Self-Test of the 80386," *Proceedings 1986 IEEE International Conference on Computer Design*, (1986), 169-173.

57. [Gola84] P. Golan, "Design of Totally Self-Checking Checker for 1-out-of-3 Code," *IEEE Transactions on Computers*, **C-33(3)** (1984), 285.

58. [Gold77] L.H. Goldstein, "A Probabilistic Analysis of Multiple Faults in LSI Circuits," R-77-304, IEEE Computer Society Repository,

59. [Gras82] G. Grassl & H. Pfleiderer, "A Self-Testing PLA," *Proceedings 1982 IEEE International Solid-State Circuits Conference*, (1982), 60-61.

60. [Gras83] G. Grassl & H.J. Pfleider, "A Function-Independent Self-Test for Large Programmable Logic Arrays," *INTEGRATION, the VLSI Journal*, **1** (1983), 71-80.

61. [Ha85] D.S. Ha & S.M. Reddy, "On the Design of Testable Domino PLA's," *International Test Conference 1985 Proceedings*, (1985), 567-573.

62. [Ha86] D.S. Ha & S.M. Reddy, "On the Design of Random Pattern Testable PLAs," *International Test Conference 1986 Proceedings*, (1986), 688-695.

63. [Ha88] D.S. Ha & S.M. Reddy, "On the Design of Pseudoexhaustive Testable PLA's," *IEEE Transactions on Computers*, **C-37(4)** (1988), 468-472.

64. [Hach82] G.D. Hachtel, A.R. Newton & A.L. Sangiovanni-Vincentelli, "An Algorithm for Optimal PLA Folding," *IEEE Transactions on Computer-Aided Design*, **CAD-1(2)** (1982), 63-76.

65. [Hass83] S.Z. Hassan & E.J. McCluskey, "Testing PLA's Using Multiple Parallel Signature Analyzers," *Digest of Papers 13th International Symposium on Fault-Tolerant Computing*, **FTCS-13** (1983), 422-425.

66. [Hong74] S.J. Hong, R.G. Cain & D.L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM Journal of Research and Development*, **18** (1974), 443-458.

67. [Hong80] S.J. Hong & D.L. Ostapko, "FITPLA: A Programmable Logic Array for Function-Independent Testing," *Digest of Papers 10th International Symposium on Fault-Tolerant Computing*, **FTCS-10** (1980), 131-136.

68. [Hort89] P.D. Hortensius, R.D. McLeod & H.C. Card, "Parallel Random Number Generation for VLSI Using Cellular Automata," *IEEE Transactions on Computers*, **C-38(10)** (1989), 1466-1473.

69. [Hua84] K.A. Hua, J.-Y. Jou & J.A. Abraham, "Built-In Tests for VLSI Finite-State Machines," *Digest of Papers 14th International Symposium on Fault-Tolerant Computing*, **FTCS-14** (1984), 292-297.

70. [Hugh84] J.L.A. Hughes, E.J. McCluskey & D.J. Lu, "Design of Totally Self-Checking Comparators with an Arbitrary Number of Inputs," *IEEE Transactions on Computers*, **C-33(6)** (1984), 546-550.

71. [Jha85] N.K. Jha & J.A. Abraham, "Techniques for Efficient MOS Implementation of Totally Self-Checking Checkers," *Digest of Papers 15th International Symposium on Fault-Tolerant Computing*, **FTCS-15** (1985), 430-435.

72. [Jha87a] N.K. Jha & M.B. Vora, "A Systematic Code for Detecting *t*-Unidirectional Errors," *Digest of Papers 17th International Symposium on Fault-Tolerant Computing*, **FTCS-17** (1987), 96-101.

73. [Jha89a] N.K. Jha, "Separable Codes for Detecting Unidirectional Errors," *IEEE Transactions on Computer-Aided Design*, **CAD-8** (1989), 571-574.

74. [Jha89b] N.K. Jha, "A Totally Self-Checking Checker for Borden's Code," *IEEE Transactions on Computer-Aided Design*, **CAD-8(7)** (1989), 731-736.

75. [Khak82a] J. Khakbaz & E.J. McCluskey, "Concurrent Error Detection and Testing for Large PLA's," *IEEE Transactions on Electron Devices*, **ED-29(4)** (1982), 756-764.

76. [Khak82b] J. Khakbaz, "Totally Self-Checking Checker for 1-out-of-*n* Code Using Two-Rail Codes," *IEEE Transactions on Computers*, **C-31** (1982), 677-681.

77. [Khak84a] J. Khakbaz, "A Testable PLA Design with Low Overhead and High Fault Coverage," *IEEE Transactions on Computers*, **C-33(8)** (1984), 743-745.

78. [Khod79] B. Khodadad-Mostashiry, "Parity Prediction in Combinational Circuits," *Digest of Papers 9th International Symposium on Fault-Tolerant Computing*, **FTCS-9** (1979), 185-188.

79. [Klin86] R.M. Kling & P. Banerjee, "A Novel Circuit Design Providing Concurrent Error Detection in PLA's," *Proceedings 1986 IEEE International Conference on Computer Design*, (1986), 588-591.

80. [Ko77] D.C. Ko & M.A. Breuer, "The Design of Self-Checking Multi-Output Combinational Circuits," *Proceedings of the National Computer Conference*, (1977), 711-721.

81. [Ko78] D.C. Ko & M.A. Breuer, "Self-Checking of Multi-Output Combinational Circuits Using Extended-Parity Techniques," *Journal of Design Automation and Fault-Tolerant Computing*, **2** (1978), 29-62.

82. [Köen79] B. Köenemann, J. Mucha & G. Zwiehoff, "Built-In Logic Block Observation Techniques," *International Test Conference 1979 Proceedings*, (1979), 37-41.

83. [Kuba84] J.R. Kuban & J.E. Salick, "Testing Approaches in the 68020," *VLSI Design*, **5(11)** (1984), 22-30.

84. [Lala86] P.K. Lala, "On Built-In Testing of VLSI Chips," *International Test Conference 1986 Proceedings*, (1986), 719-720.

85. [Law82] H.-F.S. Law & M. Shoji, "PLA Design for the BELLMAC-32A Microprocessor," *IEEE International Conference on Circuits and Computers*, **ICCC-82** (1982), 161-164.

86. [Lin88] D.J. Lin & B. Bose, "Theory and Design of t-Error Correcting and $d(d>t)$-Unidirectional Error Detecting (t-*EC d*-*UED*) Codes," *IEEE Transactions on Computers*, **C-37(4)** (1988), 433-439.

87. [Liu87] C.Y. Liu, K.K. Saluja & S.J. Upadhyaya, "BIST-PLA: A Built-In Self-Test Design of Large Programmable Logic Arrays," *Proceedings 24th ACM IEEE Design Automation Conference*, (1987), 385-391.

88. [Liu88] D. Liu & E.J. McCluskey, "Design of Large Embedded CMOS PLAs for BIST," *IEEE Transactions on Computer-Aided Design*, **CAD-7** (1988), 50-59.

89. [Lo88] J.-C. Lo & S. Thanawastien, "The Design of Fast Totally Self-Checking Berger Code Checkers Based on Berger Code Partitioning," *Digest of Papers 18th International Symposium on Fault-Tolerant Computing*, **FTCS-18** (1988), 226-231.

90. [Lu84] D.J. Lu & E.J. McCluskey, "Quantitative Evaluation of Self-Checking Circuits," *IEEE Transactions on Computer-Aided Design*, **CAD-3(2)** (1984), 150-155.

91. [Mak82] G.P. Mak, J.A. Abraham & E.S. Davidson, "The Design of PLAs with Concurrent Error Detection," *Digest of Papers 12th International Symposium on Fault-Tolerant Computing*, **FTCS-12** (1982), 303-310.

92. [Maly86] W. Maly, "Fault Models for the NMOS Programmable Logic Array," *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, (1986), 467-470.

93. [Maly87] W. Maly, "Realistic Fault Modeling for VLSI Testing," *Proceedings 24th ACM IEEE Design Automation Conference*, (1987), 173-180.

94. [Marc88] D.M. Marcynuk, "Investigations into Concurrent Error Detection of PLAs," Tech. Report 88-115, Dept. Computer Science, University of Manitoba, Winnipeg, November, 1988.

95. [Marc89a] D.M. Marcynuk, "Concurrent Checking of Nonconcurrent PLAs," *Congressus Numerantium*, **68** (1989), 99-108.

96. [Marc89b] D.M. Marcynuk, "Custom Concurrent Checking of PLAs," *Record of the Fourth Technical Workshop: New Directions for IC Testing*, (1989), 123-133.

97. [Marc90a] D.M. Marcynuk, "A Technique for Concurrent Checking of PLA's," *Congressus Numerantium*, **75** (1990), 235-246.

98. [Marc90b] D.M. Marcynuk, "On-line Checking of Programmable Logic Arrays — Source Code Listings," Tech. Report 90-116, Dept. Computer Science, University of Manitoba, Winnipeg, July, 1990.

99. [Maro77] M.A. Marouf & A.D. Friedman, "Efficient Design of Self-Checking Checkers for *m*-out-of-*n* Codes," *Digest of Papers 7th International Symposium on Fault-Tolerant Computing*, **FTCS-7** (1977), 143-149.

100. [Maro78] M.A. Marouf & A.D. Friedman, "Design of Self-Checking Checkers for Berger Codes," *Digest of Papers 8th International Symposium on Fault-Tolerant Computing*, **FTCS-8** (1978), 179-184.

101. [McCl82] E.J. McCluskey, "Verification Testing," *Proceedings 19th ACM IEEE Design Automation Conference*, (1982), 495-500.

102. [McCl84] E.J. McCluskey, "Verification Testing — A Pseudo Exhaustive Test Technique," *IEEE Transactions on Computers*, **C-33(6)** (1984), 541-546.

103. [MCNC87] "FSM Benchmarks," *Microelectronics Center of North Carolina*, 1987.

104. [Mead80] C.A. Mead & L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

105. [Min84] Y. Min, "A PLA Design for Ease Of Test Generation," *Digest of Papers 14th International Symposium on Fault-Tolerant Computing*, **FTCS-14** (1984), 436-442.

106. [Min88] Y. Min & J. Li, "Strongly Fault Secure PLA's and Totally Self-Checking Checkers," *IEEE Transactions on Computers*, **C-37(7)** (1988), 863-867.

107. [Mueh77] E.I. Muehldorf & T.W. Williams, "Optimized Stuck Fault Test Pattern Generation for PLA Macros," *Digest of Papers, 1977 Semiconductor Test Symposium*, (1977), 89-101.

108. [Nany87] T. Nanya & T. Kawamura, "On Error Indication for Totally Self-Checking Systems," *IEEE Transactions on Computers*, **C-36(11)** (1987), 1389-1392.

109. [Nany88a] T. Nanya, S. Mourad & E.J. McCluskey, "Multiple Stuck-at Fault Testability of Self-Testing Checkers," *Digest of Papers 18th International Symposium on Fault-Tolerant Computing*, **FTCS-18** (1988), 381-386.

110. [Nick80] V.V. Nickel, "VLSI — The Inadequacy of the Stuck at Fault Model," *International Test Conference 1980 Proceedings*, (1980), 378-381.

111. [Nico84] M. Nicolaidis, I. Jansch & B. Courtois, "Strongly Code Disjoint Checkers," *Digest of Papers 14th International Symposium on Fault-Tolerant Computing*, **FTCS-14** (1984), 16-21.

112. [Nico88] M. Nicolaidis & B. Courtois, "Strongly Code Disjoint Checkers," *IEEE Transactions on Computers*, **C-37(6)** (1988), 751-756.

113. [Nico89] M. Nicolaidis, "Self-Exercising Checkers for Unified Built-In Self-Test (*UBIST*)," *IEEE Transactions on Computer-Aided Design*, **CAD-8(3)** (1989), 203-218.

114. [Niko86] D. Nikolos, N. Gaitanis & G. Philokyprou, "Systematic t-Error Correcting/All Unidirectional Error Detecting Codes," *IEEE Transactions on Computers*, **C-35(5)** (1986), 394-402.

115. [Osta79] D.L. Ostapko & S.J. Hong, "Fault Analysis and Test Generation for Programmable Logic Arrays (PLA's)," *IEEE Transactions on Computers*, **C-28(9)** (1979), 617-627.

116. [Pies87] S.J. Piestrak, "Design of Fast Self-Testing Checkers for a Class of Berger Codes," *IEEE Transactions on Computers*, **C-36(5)** (1987), 629-634.

117. [Prad80] D.K. Pradhan & K. Son, "The Effect of Untestable Faults in PLAs and a Design for Testability," *International Test Conference 1980 Proceedings*, (1980), 359-367.

118. [Prat67] R.E. Prather, *INTRODUCTION TO SWITCHING THEORY: A Mathematical Approach*, Allyn and Bacon, Inc., Boston, 1967.

119. [Rajs84] J. Rajski & J. Tyszer, "Easily Testable PLA Design," *Tenth EUROMICRO Symposium on Microprocessing and Microprogramming*, (1984), 139-146.

120. [Rajs85] J. Rajski & J. Tyszer, "Combinatorial Approach to Multiple Contact Fault Coverage in Programmable Logic Arrays," *IEEE Transactions on Computers*, **C-34(6)** (1985), 549-553.

121. [Rajs86] J. Rajski & J. Tyszer, "The Influence of Masking Phenomenon on Coverage Capability of Single Stuck Fault Test Sets in PLAs," *IEEE Transactions on Computers*, **C-35(1)** (1986), 81-85.

122. [Rama82] K.S. Ramanatha & N.N. Biswas, "A Design for Complete Testability for Programmable Logic Arrays," *International Test Conference 1982 Proceedings*, (1982), 67-74.

123. [Rama83] K.S. Ramantha & N.N. Biswas, "A Design for Testability of Undetectable Crosspoint Faults in Programmable Logic Arrays," *IEEE Transactions on Computers*, **C-32(6)** (1983), 551-557.

124. [Redd84] S.M. Reddy, V.D. Agrawal & S.K. Jain, "A Gate Level Model for CMOS Combinational Circuits with Application to Fault Detection," *Proceedings 21st ACM IEEE Design Automation Conference*, (1984), 504-509.

125. [Redd85] S.M. Reddy & D.S. Ha, "On the Design of Testable PLA's," *Proceedings 1985 Conference on Information Sciences and Systems*, (1985), 1-9.

126. [Redd87] S.M. Reddy & D.S. Ha, "A New Approach to the Design of Testable PLA's," *IEEE Transactions on Computers*, **C-36(2)** (1987), 201-211.

127. [Regh86] H.K. Reghbati, "Fault Detection in PLAs," *IEEE Design & Test of Computers*, **3(6)** (1986), 43-50.

128. [Robi88] M. Robinson & J. Rajski, "An Algorithmic Branch and Bound Method for PLA Test Pattern Generation," *International Test Conference 1988 Proceedings*, (1988), 784-795.

129. [Roth66] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, **10** (1966), 278-291.

130. [Sali85] J. Salick, M.R. Mercer & B. Underwood, "Built-In Self-Test Input Generator for Programmable Logic Arrays," *International Test Conference 1985 Proceedings*, (1985), 115-125.

131. [Salu81] K.K. Saluja, K. Kinoshita, & H. Fujiwara, "A Multiple Fault Testable Design of Programmable Logic Arrays," *Digest of Papers 11th International Symposium on Fault-Tolerant Computing*, **FTCS-11** (1981), 44-46.

132. [Salu83] K.K. Saluja, K. Kinoshita & H. Fujiwara, "An Easily Testable Design of Programmable Logic Arrays for Multiple Faults," *IEEE Transactions on Computers*, **C-32(11)** (1983), 1038-1044.

133. [Salu85a] K.K. Saluja, H. Fujiwara & K. Kinoshita, "A Testable Design of Programmable Logic Arrays with Universal Control and Minimal Overhead," *International Test Conference 1985 Proceedings*, (1985), 574-582.

134. [Salu85b] K.K. Saluja, K. Kinoshita & C. Boswell, "A Design of Parallel Testable Programmable Logic Arrays," *Proceedings of the International Symposium on Circuits and Systems*, **ISCAS-85** (1985), 1325-1328.

135. [Salu86] K.K. Saluja & S.J. Upadhyaya, "A Built-In Self Testing PLA Design with High Fault Coverage," *Proceedings 1986 IEEE International Conference on Computer Design*, (1986), 596-599.

136. [Salu87a] K.K. Saluja, R. Sharma & C.R. Kline, "Concurrent Comparative Testing Using BIST Resources," *Proceedings of the IEEE International Conference on Computer-Aided Design*, (1987), 336-339.

137. [Salu87b] K.K. Saluja, H. Fujiwara & K. Kinoshita, "A Testable Design of Programmable Logic Arrays with Universal Control and Minimal Overhead," *Computers & Mathematics with Applications*, **13(5/6)** (1987), 503-517.

138. [Saye85] I.L. Sayers & D.J. Kinniment, "Low-Cost Residue Codes and Their Application to Self-Checking VLSI Systems," *IEE Proceedings-E*, **132(4)** (1985), 197-202.

139. [Sche72] D.R. Schertz & G. Metze, "A New Representation for Faults in Combinational Digital Circuits," *IEEE Transactions on Computers*, **C-21(8)** (1972), 858-866.

140. [Sedm79] R.M. Sedmak, "Design for Self-Verification: An Approach for Dealing with Testability Problems in VLSI-Based Designs," *International Test Conference 1979 Proceedings*, (1979), 112-120.

141. [Sedm80] R.M. Sedmak, "Implementation Techniques for Self-Verification," *International Test Conference 1980 Proceedings*, (1980), 267-278.

142. [Serr85] M. Serra & J.C. Muzio, "Testing Programmable Logic Arrays by Sum of Syndrome," Fault Detection Research Group Report Number 10-1985, University of Victoria,

143. [Serr87] M. Serra & J.C. Muzio, "Testing Programmable Logic Arrays by Sum of Syndrome," *IEEE Transactions on Computers*, **C-36(9)** (1987), 1097-1101.

144. [Serr88] M. Serra, "Some Experiments on the Overhead for Concurrent Checking," *Working Papers of the Third Technical Workshop: New Directions for IC Testing*, (1988), 207-212.

145. [Shar88] R. Sharma & K.K. Saluja, "An Implementation and Analysis of a Concurrent Built-In Self-Test Technique," *Digest of Papers 18th International Symposium on Fault-Tolerant Computing*, **FTCS-18** (1988), 164-169.

146. [Smit77] J.E. Smith, "The Design of Totally Self-Checking Check Circuits for a Class of Unordered Codes," *Journal of Design Automation and Fault-Tolerant Computing*, **2** (1977), 321-342.

147. [Smit78] J.E. Smith & G. Metze, "Strongly Fault Secure Logic Networks," *IEEE Transactions on Computers*, **C-27(6)** (1978), 491-499.

148. [Smit79] J.E. Smith, "Detection of Faults in Programmable Logic Arrays," *IEEE Transactions on Computers*, **C-28(11)** (1979), 845-853.

149. [Smit84] J.E. Smith, "On Separable Unordered Codes," *IEEE Transactions on Computers*, **C-33(8)** (1984), 741-743.

150. [Some84] F. Somenzi, S. Gai, M. Mezzalama & P. Prinetto, "PART: Programmable Array Testing Based on a Partitioning Algorithm," *IEEE Transactions on Computer-Aided Design*, **CAD-3(2)** (1984), 142-149.

151. [Some86] F. Somenzi & S. Gai, "Fault Detection in Programmable Logic Arrays," *Proceedings of the IEEE*, **74** (1986), 655-668.

152. [Son80] K. Son & D.K. Pradhan, "Design of Programmable Logic Arrays for Testability," *International Test Conference 1980 Proceedings*, (1980), 163-166.

153. [Son81] K. Son & D.K. Pradhan, "Completely Self-Checking Checkers in PLAs," *International Test Conference 1981 Proceedings*, (1981), 231-237.

154. [Tami84] Y. Tamir & C.H. Séquin, "Design and Application of Self-Testing Comparators Implemented with MOS PLA's," *IEEE Transactions on Computers*, **C-33(6)** (1984), 493-506.

155. [Tao86] D.L. Tao, P.K. Lala & C.R. Hartmann, "A Concurrent Testing Strategy for PLAs," *International Test Conference 1986 Proceedings*, (1986), 705-709.

156. [Tao87] D.L. Tao, P.K. Lala & C.R. Hartmann, "Three-level Totally Self-Checking Checker for 1-out-of-*n* code," *Digest of Papers 17th International Symposium on Fault-Tolerant Computing*, **FTCS-17** (1987), 108-113.

157. [Tao88] D.L. Tao, "Applications of Coding Techniques in the Design of Self Checking PLAs," Ph.D. dissertation, Syracuse University, 1988.

158. [Tohm86] Y. Tohma, "Coding Theory for Fault-Tolerant Systems," in *FAULT-TOLERANT COMPUTING — Theory and Techniques*, D.K. Pradhan *ed.*, Prentice-Hall, Englewood Cliffs, NJ., 1986, 336-415.

159. [Treu85a] R. Treuer, H. Fujiwara & V.K. Agarwal, "Implementing a Built-In Self-Test PLA Design," *IEEE Design & Test of Computers*, **2**(2) (1985), 37-48.

160. [Treu85b] R. Treuer, H. Fujiwara & V.K. Agarwal, "A Low Overhead High Coverage, Built-In Self-Test PLA Design," *Digest of Papers 15th International Symposium on Fault-Tolerant Computing*, **FTCS-15** (1985), 112-117.

161. [Treu87] R. Treuer, V.K. Agarwal & H. Fujiwara, "A New Built-In Self-Test Design for PLA's with High Fault Coverage and Low Overhead," *IEEE Transactions on Computers*, **C-36**(3) (1987), 369-373.

162. [Upad88] S.J. Upadhyaya & K.K. Saluja, "A New Approach to Design for BIST PLAs for High Fault Coverage," *IEEE Transactions on Computer-Aided Design*, **CAD-7** (1988), 60-67.

163. [Wads78] R.L. Wadsack, "Fault Modelling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, **57** (1978), 1449-1474.

164. [Wake78] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, North-Holland, New York, NY, 1978.

165. [Wang79] S.L. Wang & A. Avizienis, "The Design of Totally Self-Checking Circuits Using Programmable Logic Arrays," *Digest of Papers 9th International Symposium on Fault-Tolerant Computing*, **FTCS-9** (1979), 173-180.

166. [Wei86] R.S. Wei & A. Sangiovanni-Vincentelli, "PLATYPUS: A PLA Test Generation Tool," *IEEE Transactions on Computer-Aided Design*, **CAD-5** (1986), 633-644.

167. [Yaji81] S. Yajima & T. Aramaki, "Autonomously Testable Programmable Logic Arrays," *Digest of Papers 11th International Symposium on Fault-Tolerant Computing*, **FTCS-11** (1981), 41-43.

168. [Zhu88] X.-A. Zhu & M.A. Breuer, "Analysis of Testable PLA Designs," *IEEE Design & Test of Computers*, **5**(4) (1988), 14-28.

DEFINITION INDEX