This is to certify that the

thesis entitled
Design of Repairable and Fully Diagnosable Folded Programmable Logic Arrays for Yield Enhancement

presented by
Jyhyeuan Ding

has been accepted towards fulfillment
of the requirements for

Master of Science degree in Electrical Engineering

Major professor

Date March 10, 1990

DESIGN OF REPAIRABLE AND FULLY DIAGNOSABLE FOLDED PROGRAMMABLE LOGIC ARRAYS FOR YIELD ENHANCEMENT

BY

Jyhyeuan Ding

A THESIS

Submitted to

Michigan State University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1990

ABSTRACT

DESIGN OF REPAIRABLE AND FULLY DIAGNOSABLE FOLDED PROGRAMMABLE LOGIC ARRAYS FOR YIELD ENHANCEMENT

By

Jyhyeuan Ding

The entire manufacturing process of integrated circuits (ICs) has three major yield steps that affect the total production run of any IC product. These major steps are: *wafer processing yield, probe yield,* and *final test yield.* The most critical step, however, is probe yield, which dramatically affects the number of functional devices. Probe yield, defined as the number of good die from a processed wafer, is affected by both die size and defect density. If the die size or the defect density increases, then total IC yield percentages rapidly decline.

IC yield has always been a crucial factor in successful commercial manufacturing. The technologies of ICs evolved from LSI, VLSI, to ULSI in the past two decades. However, as the complexity of digital devices increases and geometry shrinks, the probability of having faulty components also increases, thereby lowering the chip yield. One practical solution to low probe yield problem is the use of fault-tolerant design.

In order to ensure large PLA (Programmable Logic Array) chips to be manufactured with reasonable yield level, a design of repairable and fault-diagnosable PLA has been proposed recently. The design achieves a full diagnosability for single and multiple stuck-at, bridging, and crosspoint faults, and has led to a significant improvement in chip yield.

However, the design requires an excessive area overhead for fault diagnosis and repair. These increases in die size would reduce the number of die that can be placed on a wafer. In this thesis, an alternative fault-tolerant design of PLAs with folding techniques is presented to reduce the die size, while still achieving the same diagnosability during the manufacturing process. In addition, the design also achieves a full testability after the chip is packaged.

This thesis presents the fault-tolerant designs of PLAs with various folding techniques provided by PLEASURE. The chip areas required for various folding techniques are compared. Results demonstrate that fault-tolerant design of PLA with simple column folding techniques generally provides a "better" solution where chip yield improvement is concerned.

The computer-aided design (CAD) tools play a very important role in VLSI design. They can reduce the turnaround time and make design changes more quickly. In this thesis, an automatic layout generator, ALGFPLA, has been developed and implemented on SUN 3/160 under UNIX operating system for generating physical layout of fault-tolerant folded PLAs.

ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation to his major advisor, Dr. Chin-Long Wey, for the guidance and encouragement given in the course of this graduate study. He also wishes to thank Dr. Tsin-Yuan Chang for his great help. Gratitude is also extended to his committee members, Dr. P. David Fisher and Dr. Donnie K. Reinhard. Finally, the author wishes to express his deep appreciation to his parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The entire manufacturing process of integrated circuits (ICs) has three major yield steps that affect the total number of functional IC products that are realized [1]. The major steps are: *wafer processing yield, probe yield,* and *final test yield. Wafer processing yield* is defined as the percentage of good wafers that survive the manufacturing process. This yield is usually above 90%. *Probe yield* is defined as the percentage of good chips per wafer. This yield ranges from 30% to 60% depending upon the wafer die size. Finally, *final test yield* is the percentage of devices that pass a final test program which occurs after the die has been packaged. Usually, this percentage is over 95%.

The most dominant of these three is *probe yield,* which dramatically affects the number of functional devices. The die size and defect density are the two dominant factors that affect the *probe yield.* The yield falls very rapidly as either the die size or the defect density increases.

The advent of Very Large Scale Integrated (VLSI) circuit technology illustrates the continuing trend toward increasing gate counts on a logic chip. As the complexity of digital devices increases and the geometry shrinks, the probability of having faulty components also increases, and thus the probe yield decreases. The low probe yield problem can be alleviated either by improving manufacture process [2], or by introducing

fault-tolerant design [3]. The former, a technique-dependent solution, is very costly and quite difficult to implement within a short time. The latter is a trend of the future in manufacturing [4 - 8].

1.1 Problem Statement and Objectives

Due to the complexity and cost of designing chips nowadays, a structured form of logic implementation is desirable [9]. During the last few years, Programmable Logic Arrays (PLAs) [10, 11] have become increasingly common for implementing Boolean logic functions in VLSI circuit chips [12]. In order to ensure large PLA chips to be manufactured with reasonable yield level, a design of fault-diagnosable and repairable PLAs has been recently proposed [4 - 8], in which a partially defective chip can be repaired without reconfiguring the external routing. The fault-tolerant design achieves a full diagnosability and repairability of single and multiple stuck-at, bridging, and crosspoint faults, and has led to a significant improvement in chip yield.

Although the structural regularity of PLAs offers design simplicity in producing VLSI circuits, PLAs generally require larger chip area and longer delay time than random logic implementation. With the recent development of sophisticated multi-level logic design tools, standard cells and gate arrays have been popularly implemented with multi-level logic circuits to reduce both chip area and propagation delay time. However, as far as the testable design and the fault tolerance for yield enhancement are concerned, the features are very difficult to apply to the gate array and standard cell implementations due to their irregularity. Therefore, in order to enhance chip yield and to make the circuit easily testable, the fault-tolerant design of the regular structure PLAs is definitely feasible in VLSI/ULSI implementation.

However, the fault-tolerant PLA design requires an excessive area overhead for the purposes of fault diagnosis and repair. The increase in die size would reduce the number

of die that can be placed on a wafer. Folding techniques [13-15] have been commonly implemented to significantly reduce the chip area. In practice, however, folded PLAs are very difficult to test and the failure rate is higher than the conventional PLA due to the complexity caused by folding. Thus, lower chip yield has been found as a problem of implementing with folded PLAs. This motivates to propose a fault-tolerant design of folded PLAs. Folding techniques are adopted to reduce the chip area, thereby increasing the number of die that can be placed on a wafer. On the other hand, the fault-tolerant design is to fully diagnose the faults in the chip, thus enhance the yield.

Generating an efficient mask layout which implements a complex circuit function often causes a bottleneck for later design stages [16]. Therefore, developing the automatic layout generation programs is quite necessary and these programs play today an increasingly important role in VLSI circuits design. The use of program generated layout of regular structures increases their generality and their reusability. In this study, an automatic layout generator is presented.

1.2 Thesis Organization

This thesis presents an innovative fault-tolerant design for folded PLAs and an automatic layout generator. Chapter 2 reviews the design of fault-tolerant PLAs and introduces some folding techniques.

Chapter 3 proposes the fault-tolerant design of a folded PLA with a simple column folding technique. This design achieves a full diagnosability and repairability for single and multiple stuck-at faults, bridging faults, and crosspoint faults during the manufacturing process.

In order to automatically generate the layout mask for a fault-tolerant design of folded PLAs, an automatic layout generator, ALGFPLA, is presented in Chapter 4. The layout generator has been implemented on SUN 3/160 under UNIX operating system.

In Chapter 5, fault-tolerant designs of PLA with various folding techniques are presented. The simulation results for various designs are compared.

Finally, Chapter 6 summarizes the work of this thesis and describes the feature research in the area of fault-tolerant PLA designs.

Chapter 2

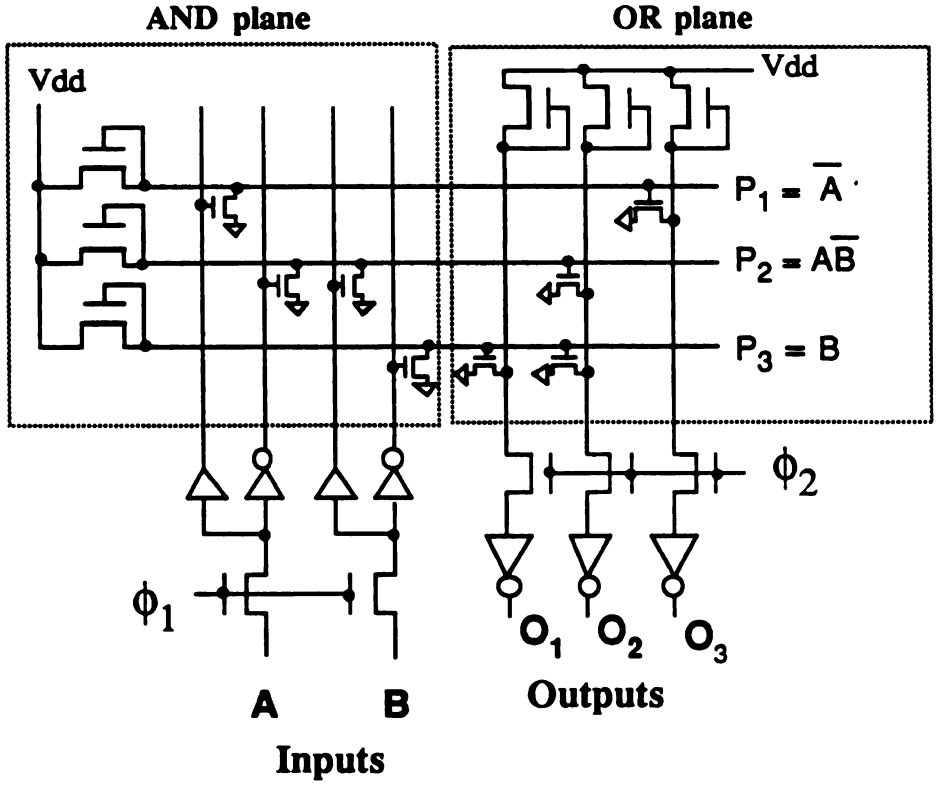
Background

2.1 Fault-Tolerant PLA Design

A Programmable Logic Array is a two-level AND-OR logic network that implements the combinational circuits. By adding the storage elements such as latches and flip-flops, PLAs can also realize sequential circuits. A typical PLA consists of two planes: AND plane and OR plane. Figure 2.1 illustrates a PLA implemented by a NOR-NOR structure in NMOS technology. Three fault models are generally considered for such a PLA structure: *Crosspoint fault*, *Stuck-at fault*, and *Bridging fault* [17 - 22].

2.1.1 Fault Models

A *crosspoint fault* is caused by the unintentional presence or absence of a transistor in the AND plane or OR plane. Four cases can be identified: *Growth fault* (or G-fault), *Shrinkage fault* (or S-fault), *Disappearance fault* (or D-fault), and *Appearance fault* (or A-fault). The first two occur in the AND plane, while the last two are in the OR plane. *A Growth fault* is caused by missing crosspoint in the AND plane. This results in the disappearance of an input variable from a product term, i.e., in the Karnaugh map, the number of minterms for this product term is increased. A *Shrinkage fault* results from an



(a)

Logic function of a PLA

$O_1 = P_3 = B$

$O_2 = P_2 + P_3 = A\overline{B} + B$

$O_3 = P_1 = \overline{A}$

(b)

Cubic notation

0	-	0	0	1
1	0	0	1	0
-	1	1	1	0
A	B	O_1	O_2	O_3

(c)

Figure 2.1 Programmable Logic Array:
(a) NMOS Implementation;
(b) Logic Functions; and
(c) Personality in Cubic Notation.

extra crosspoint in the AND plane. With the appearance of an input variable in a product term, the number of minterms is decreased. Similarly, A *Disappearance fault* has a missing crosspoint in the OR plane, and an *Appearance fault* has an extra crosspoint in the OR plane.

A *stuck-at fault*, the simplest type of fault, is a line permanently set to logic states 1 or 0. This fault is caused by the faulty line being opened or shorted to the power or ground line (GND). While a stuck-at-0 (s-a-0) fault at the input bit line causes the variable of this bit line to disappear from the product terms, a stuck-at-1 (s-a-1) faulty input bit line results in s-a-0 faults at those product lines which connect to this faulty line. Similarly, an s-a-1 faulty product line causes the output lines to have s-a-0 faults if the output lines connect to this product line. An s-a-0 product line will cause the product term of this product line to disappear from the outputs. Finally, the result of a stuck-at fault at an output line is quite obvious - the output will be stuck at its present level.

The last fault model is the *bridging fault*, which is a short between two adjacent or crossing lines. This fault forces the same logic value to appear on the bridged lines. In the NMOS technology, a wired-AND is assumed. Only when both of the bridged lines are at logic 1 will the values appear on these bridged lines be logic 1's; otherwise, they will be logic 0's.

2.1.2 Fault-Tolerant Design

The fault-tolerant PLA (FTPLA) design [6 - 8] includes both the fault-diagnosable and repairable design. While the fault diagnosability is accomplished by employing the additional shift registers, spare lines achieve the fault repairability. More specifically, Figure 2.2 illustrates the schematic diagram for a fault-diagnosable PLA (FDPLA). Two shift registers are employed: Input line's Shift Register (ISR) and Product line's Shift Register (PSR). The shift registers are operated with the control circuitries shown in

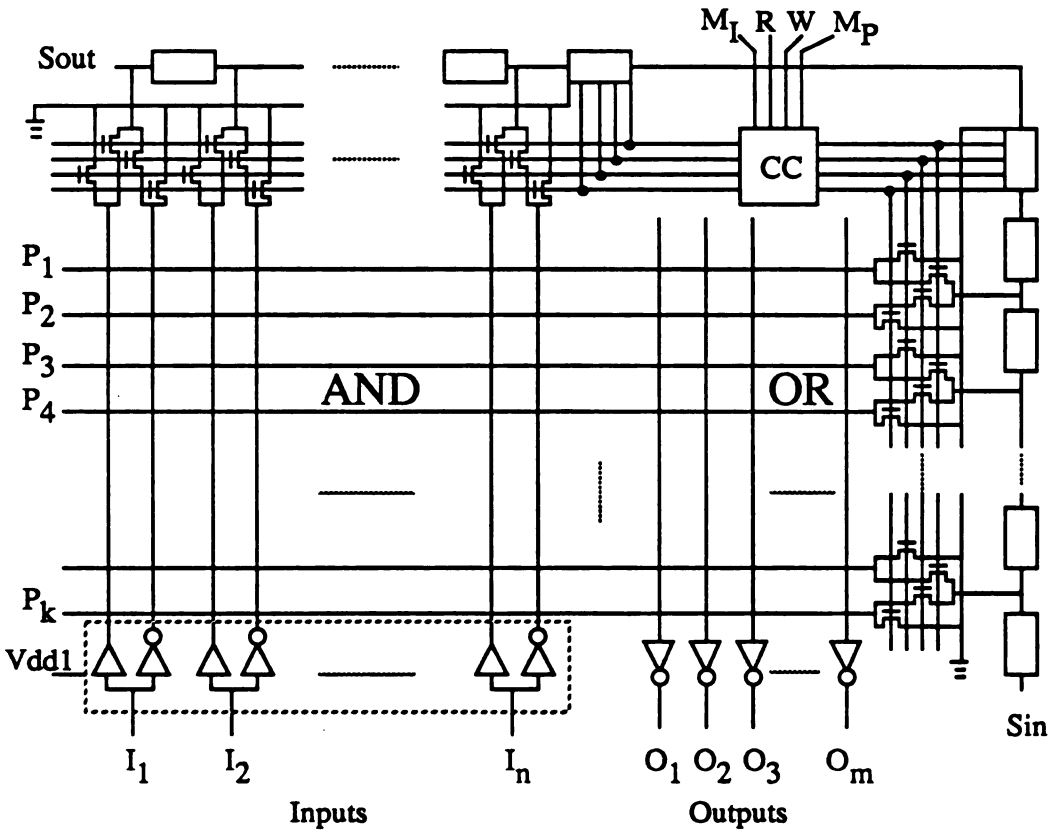


Figure 2.2 Schematic Diagram for Fault-Diagnosable PLA.

Figure 2.3.

Basically, the PSR (ISR) is used to enable only one product line (input bit line) at a time. As shown in Table 2.1, the pattern (M_P, R, W) = (0, 0, 1) writes a 1 to the (2i)-th product line (or P_{2i}) and all 0's to the remaining product lines, i.e., P_{2i} is the only enabled line. In addition, the PSR (ISR) also allows to read the content of each product line (input bit line), thereby significantly enhancing the diagnosability.

Figure 2.4 shows a schematic diagram of a fault-repairable PLA (FRPLA). The original PLA is augmented by Spare Input Selector Circuit (SISC), Spare Output Selector

Circuit (SOSC), and spare lines. Basically, when a faulty line is detected and located, we reconfigure the selector circuits to switch this faulty line to a corresponding spare line. In order to repair faults described in Section 2.1.1, a set of repair rules is summarized in Table 2.2.

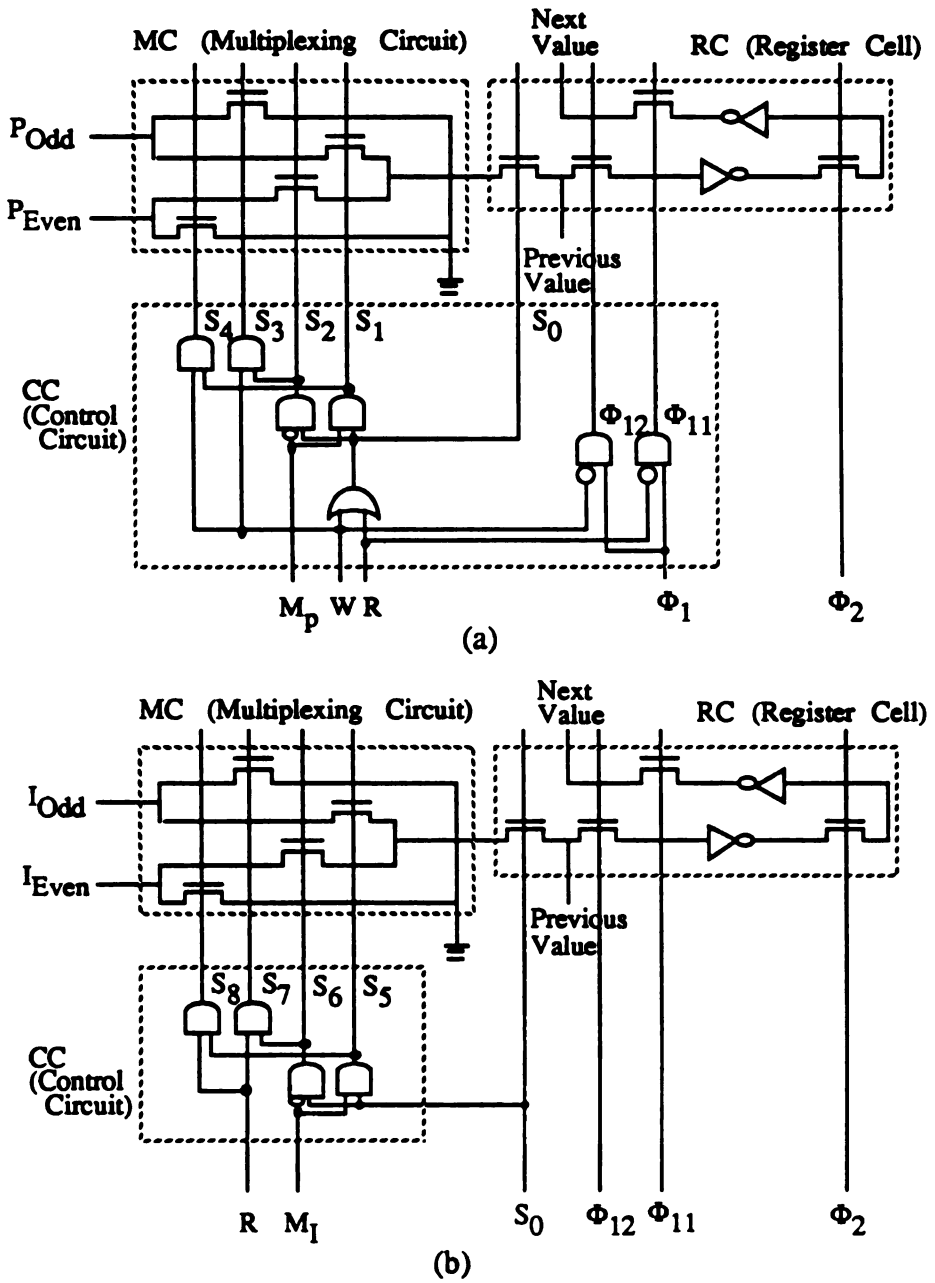


Figure 2.3 Schematic Diagrams of Shift Registers and Their Control Circuits: (a) PSR; and (b) ISR.

Table 2.1 Shift Registers Operations for: (a) PSR; and (b) ISR

(a)

W	R	M_P	S_1	S_2	S_3	S_4	Operations
0	0	x	0	0	0	0	Isolate PSR from the PLA
0	1	0	0	1	0	0	Read EVEN
0	1	1	1	0	0	0	Read ODD
1	0	0	0	1	1	0	Write data of RC to EVEN and set ODD to 0
1	0	1	1	0	0	1	Write data of RC to ODD and set EVEN to 0
1	1	x	x	x	x	x	Invalid Case

Remark: ODD --- odd-numbered product line.
EVEN--- even-numbered product line.

(b)

W	R	M_I	S_5	S_6	S_7	S_8	Operations
0	0	x	0	0	0	0	Isolate ISR from the PLA
1	0	0	0	1	0	0	Read COMP
1	0	1	1	0	0	0	Read TRUE
0	1	0	0	1	1	0	Write data of RC to COMP and set TRUE to 0
0	1	1	1	0	0	1	Write data of RC to TRUE and set COMP to 0
1	1	x	x	x	x	x	Invalid Case

Remark: COMP--- complemented bit line.
TRUE --- true bit line.

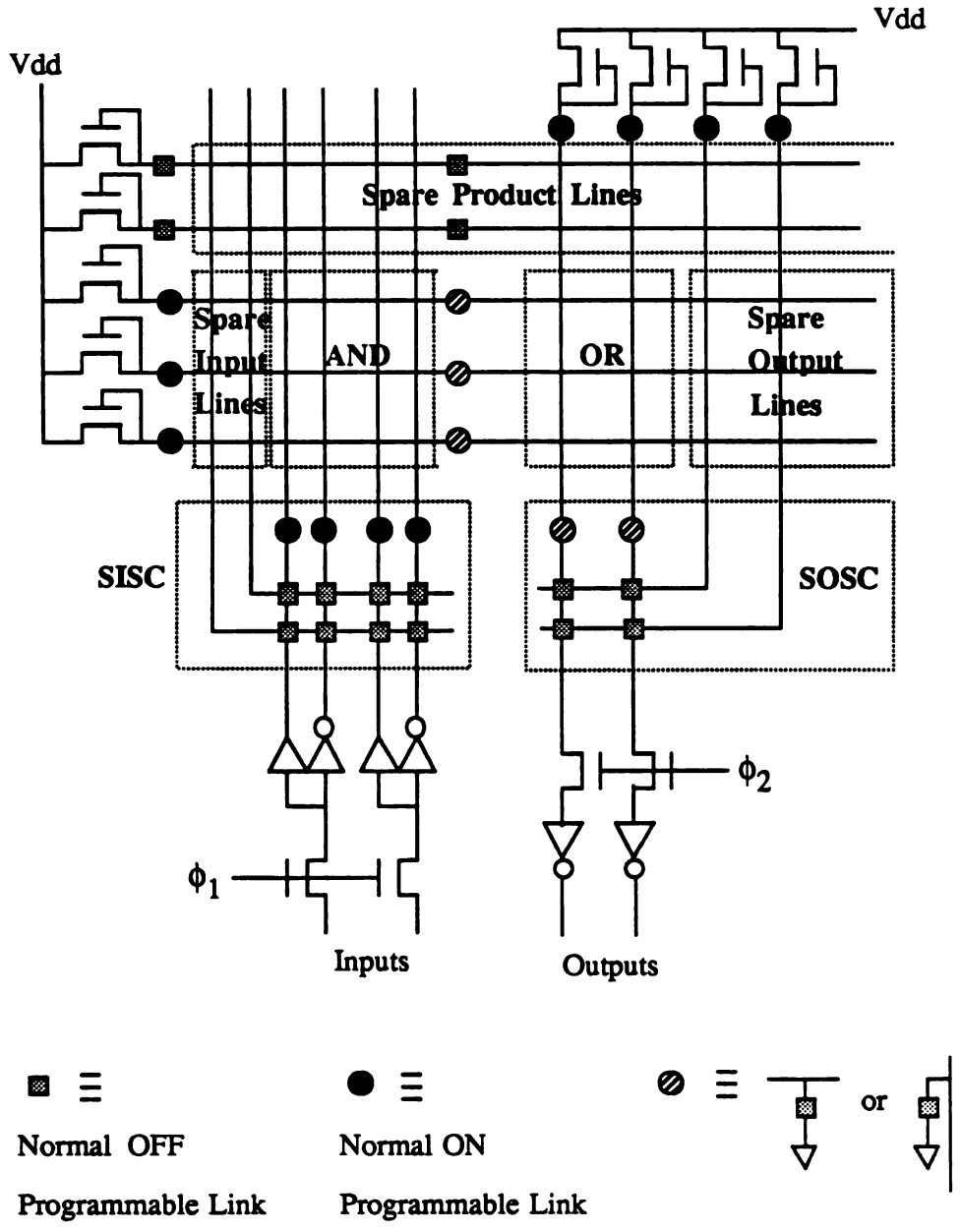


Figure 2.4 Schematic Diagram of a Fault-Repairable PLA.

Table 2.2 Repair Rules

Fault Type		Spare Line	Notes
1. Stuck-at Fault			
	Input bit line	Input bit line	remark 1
	Product line	Product line	remark 2
	Output line	Output line	remark 3
2. Crosspoint Fault			
	Growth	Input bit line	remark 1
		Product line	remark 2
	Shrinkage	Input bit line	remark 1
		Product line	don't care
	Disappearance	Product line	don't care
		Output line	remark 3
	Appearance	Product line	remark 2
		Output line	remark 3
3. Bridging Fault			
(1) Adjacent			
	Input bit lines	Input bit lines	remark 1
	Product lines	Product lines	remark 2
	Output lines	Output lines	remark 3
(2) Crossing			
	Input and product lines	Input bit line	remark 1
		and product line	remark 2
	Product and output lines	Product line	remark 2
		and output line	remark 3

Remarks: 1. Faulty bit line is disconnected from SISC, and is connected to GND.
2. Faulty product line is disconnected from its pull-up transistor, and connected to GND.
3. Faulty output line is disconnected from its pull-up transistor and SOSC, and connected to GND.

2.1.3 Fault Diagnosis and Repair Process

The fault diagnosis and repair process [8] consists of four major steps: (1) detect faults in augmented circuitry, (2) identify and repair faults in the AND plane, (3) identify and repair faults in the OR plane, and (4) repair crosspoint faults.

Faults in the AND plane include: Stuck-at faults at input bit lines, Bridging faults at adjacent input bit lines, Bridging faults between input bit lines and product lines, Stuck-at faults at product line, and Crosspoint faults (G- and S-faults). Faults in the OR plane include: Stuck-at faults at output lines, Bridging faults between output lines and product lines, Bridging faults between adjacent product lines, Bridging faults between adjacent output lines, and Crosspoint faults (A- and D-faults). In this implementation, the stuck-at and bridging faults must be repaired immediately when they are identified. Otherwise, the stuck-at faults may mask some other faults so that the precise identification of fault types cannot be made.

In this process, the augmented circuitry is tested first. Since the augmented circuitry is non-redundant, any faults are considered as fatal. As such, the repair process must be terminated. After the augmented circuitry has been proved to be fault-free, the following steps of fault diagnosis and repair process can then be implemented. Basically, the faults in the AND plane are identified as follows. Stuck-at and bridging faults at bit lines are identified by observing the contents of bit lines from the ISR cells. This is followed by observing the contents of the product lines from the PSR to locate the stuck-at and bridging faults at the product lines. In addition, the contents of the product lines can also be used to identify the crosspoint faults (G- and S- faults). Similarly, by applying patterns from the PSR and observing the output lines, one can locate the stuck-at and bridging faults at the output lines, as well as the crosspoint faults (D- and A- faults). Based on a fault map consisting of all crosspoint faults, a spare allocation algorithm can efficiently utilize the spare lines to repair crosspoint faults.

Previous research has demonstrated that the fault diagnosis and repair process can achieve a full diagnosability of single and multiple stuck-at faults, bridging faults, and crosspoint faults [8].

2.1.4 Design Evaluation

The fault-tolerant PLA design includes both fault diagnosability and repairability. The original PLA is augmented by adding extra chip area for fault diagnosis and repair use. Figure 2.5 shows the physical layout of a fault-tolerant PLA "mish", and Figure 2.6 illustrates the floor plan. The layout includes the original PLA, the spare lines, the shift registers, and control circuits. Based on the floor plan, Table 2.3 compares the chip areas required for the original PLA, (1, 2, 1)-FRPLA, and FDPLA for various PLAs, where (1, 2, 1)-FRPLA means that the FRPLA has one spare input bit line, two spare product lines, and one spare output line.

Table 2.3 depicts that the fault-tolerant design of the (100, 400, 100)-PLA with (1, 2, 1) spare assignment requires an additional 11.54% area overhead. Research has demonstrated that the yield of this design can be enhanced nearly five times higher than the nonredundant design [8].

Table 2.3 Area Overhead of FTPLAs

			Original	Augmented PLA				
			PLA	(1, 2, 1)-FRPLA		FDPLA		FTPLA
n	p	m	Area	Area	%	Area	%	%
50	190	67	2210460	134868	6.10	324800	14.69	20.80
60	200	60	2495564	142564	5.71	358400	14.36	20.07
100	200	100	4104524	189924	4.63	448000	10.91	15.54
100	400	100	8022924	253924	3.16	672000	8.38	11.54

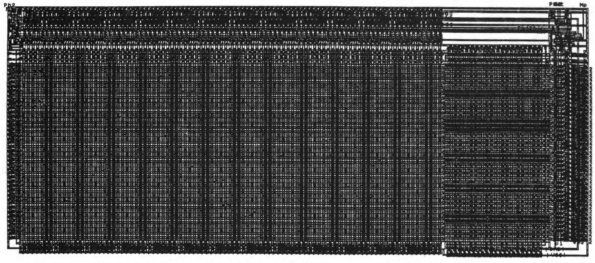


Figure 2.5 Physical Layout of FTPLA "mish".

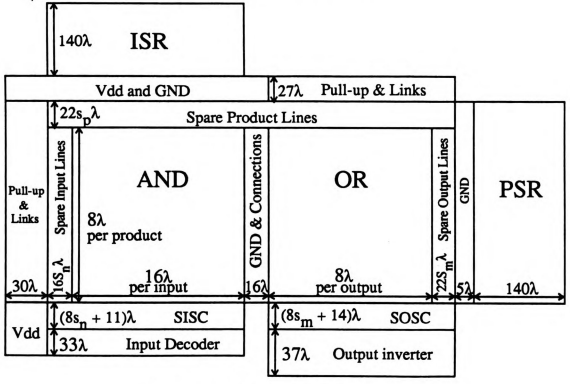


Figure 2.6 A Floor Plan of Fault-Tolerant PLA.

2.2 Folding Techniques

The conventional PLA has a major drawback of having a lower transistor rate which results in a significant waste of silicon area. One way to reduce the wasted area is to compact the array by using folding techniques. Folding is a technique that attempts to reduce the area of a PLA by exploiting its sparsity. The objective is to determine permutations of the rows (and/or columns) which permit a maximal set of column pairs (row pairs) to be implemented in the same column (row) of the physical logic array. A number of folding techniques have been proposed [13 - 15, 23 - 26]. Folding techniques overcome array sparseness by cutting and rearranging input, output, and product lines.

Figure 2.7 shows the schematic diagram of a Simple Column Folded Programmable Logical Array (SCFPLA). One input enters the top of SCFPLA, referred to as TOP input, and one enters from the bottom of SCFPLA, referred to as BOTTOM input, in the same physical column. A *bit column* is defined as the column having one or two input bit lines. If a *bit column* has two input bit lines, then they are separated by a "cut". We define the TOP output, BOTTOM output, and *output column* in the same fashion as discussed above. In this folding technique, product line folding is not allowed.

A Simple Row Folded Programmable Logical Array (SRFPLA) is a structure in which two logical rows may share one physical row. According to the array structures, two cases can be identified: AND-OR-AND structure (SRFPLA-A), and OR-AND-OR structure (SRFPLA-O), as shown in Figure 2.8.

Simple folding is just a special case of multiple folding. For a Multiple Column Folded PLA (MCFPLA), as shown in Figure 2.9, an input (or output) can enter a PLA either from the top, the bottom, or the side of the PLA. With the Multiple Row Folded PLA (MRFPLA), the array structure can be repeated in two different ways. Figure 2.9 illustrates the AND-OR and OR-AND structures for MRFPLAs.

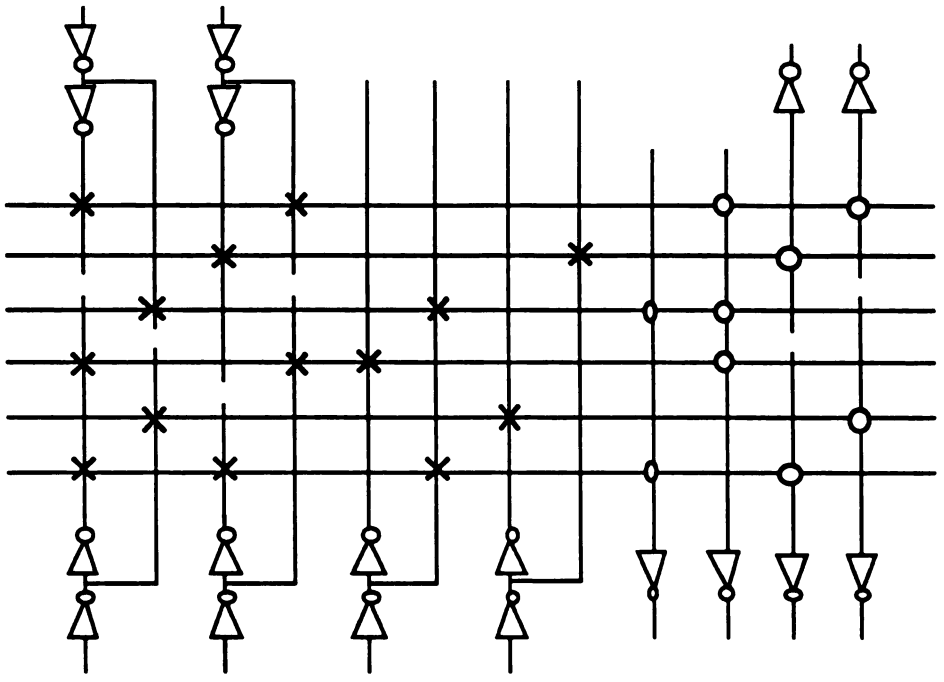


Figure 2.7 Schematic Diagram of a SCFPLA.

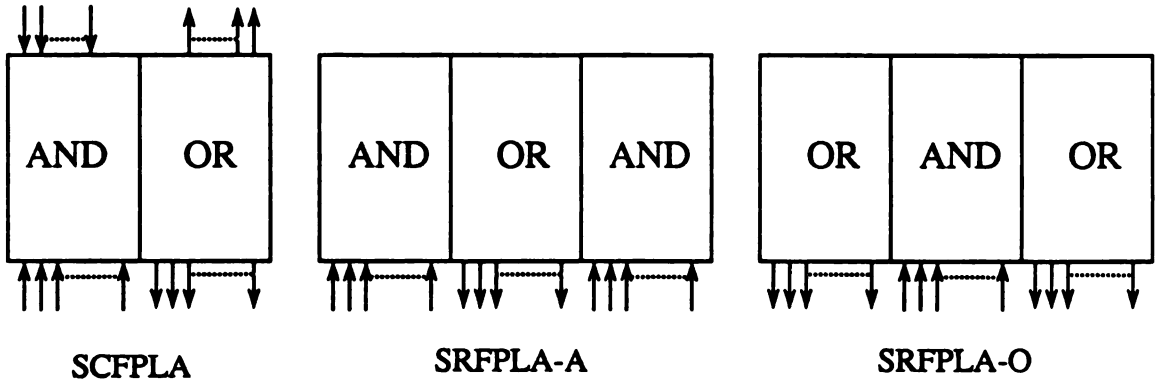


Figure 2.8 Block Diagrams of Simple Folded PLAs.

PLEASURE is an interactive program for simple/multiple constrained/unconstrained row and/or column folding of PLAs [27]. The PLA description is given as input to the program in the form of two-level sum-of-products logic implicants. The output of the program is a symbolic table representing the folded array with the positions of the active devices corresponding to the cubes of the logic function, the location of the cuts and the

contacts between columns and connection rows. The symbolic table is suitable to be processed by a layout generator which generates the mask layout of the array according to a given technology. Note that the symbolic table of PLEASURE, as listed in Table 2.4, is technology independent. Figure 2.10 shows an example of PLEASURE for simple column folding technique.

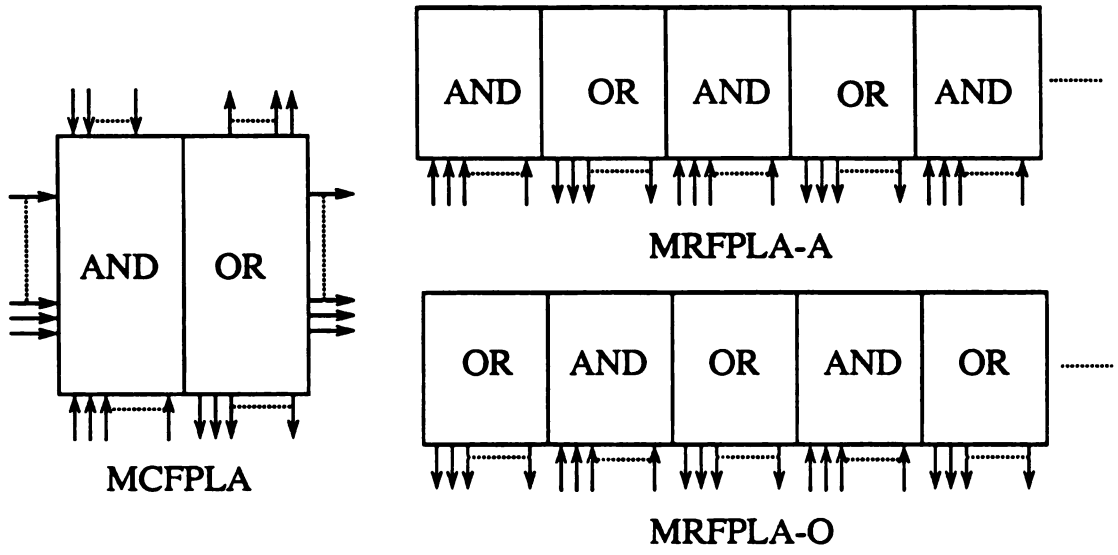


Figure 2.9 Block Diagram of Multiple Folded PLA.

Table 2.4 Symbols of PLEASURE:
(a) AND Plane; and (b) OR Plane

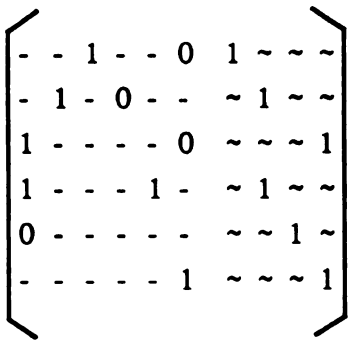
(a) AND Plane

	Normal	Split below
Contact to true bit line	1	!
Contact to complemented bit line	0	o
No contact	-	_

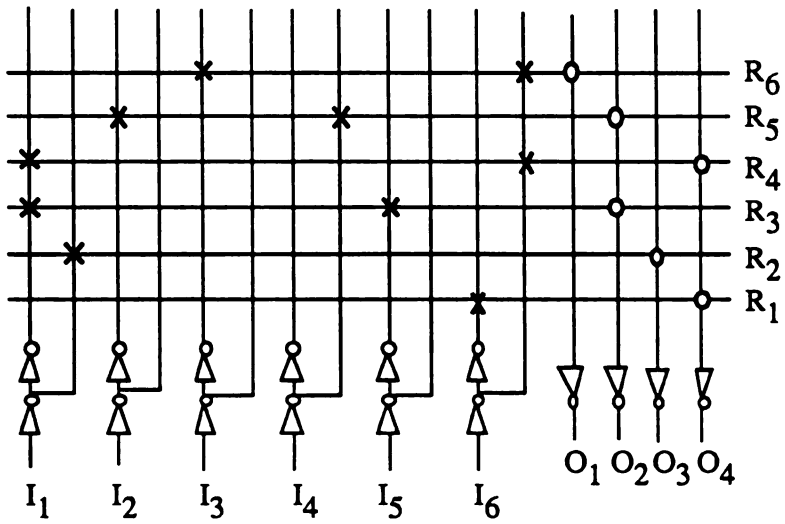
(b) OR Plane

	Normal	Split below
Contact to output line	I	i
No contact	~	=

Original:

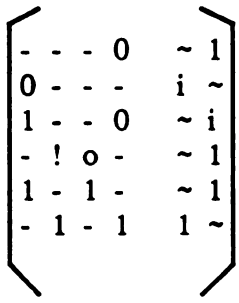


(a)

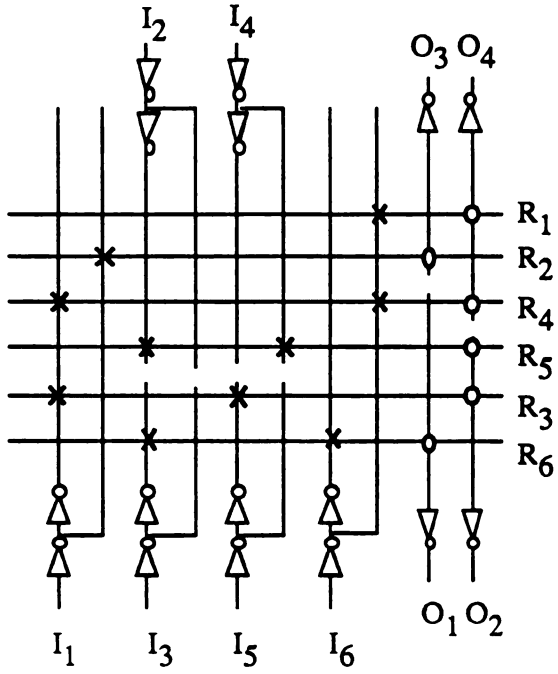


(b)

PLEASURE: SCFPLA



(c)



(d)

Figure 2.10 An Example of PLEASURE for Simple Column Folding:
(a) Personality Matrix , (b) Schematic Diagram of PLA; and
(c) Personality Matrix , (d) Schematic Diagram of SCFPLA.

Chapter 3

Fault-Tolerant SCFPLA Design

This chapter describes a fault-tolerant design of PLA with Simple Column Folding technique (SCFPLA). The proposed fault-tolerant design includes the fault-diagnosable design and repairable design.

3.1 Fault-Diagnosable SCFPLA Design

Fault diagnosability of a PLA, as shown in Figure 2.2, is accomplished by adding the shift registers ISR and PSR to the original PLA. The features of ISR and PSR significantly enhance the controllability and observability of the PLA. For a SCFPLA, since the bit columns in the folded AND plane are shared by the TOP inputs and the BOTTOM inputs, it is virtually difficult to insert the shift register ISR. Therefore, an alternate fault-diagnosable structure is proposed.

Figure 3.1 illustrates a schematic diagram of the proposed fault-diagnosable structure for the SCFPLA. Both AND and OR planes are composed of folded and unfolded parts. Similar to the fault-diagnosable design of Figure 2.2, the PSR is connected to the product lines, and the ISR is used in the unfolded part of the AND plane. In order to achieve the full diagnosability, an ISR-like structure is presented for the folded part of the AND plane.

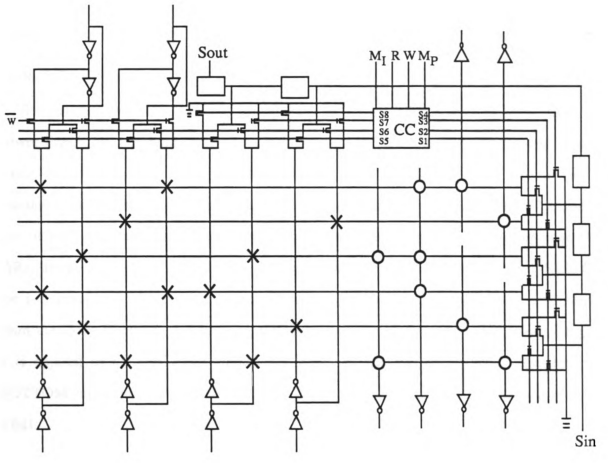


Figure 3.1 Schematic Diagram of a Fault-Diagnosable SCFPLA.

3.1.1 PSR and ISR

Figure 3.2 illustrates the PSR and ISR cells and the corresponding control circuitry. The operations of the PSR and ISR in Figure 3.2 are the same as those in Figure 2.3.

However, the redundant circuit in Figure 2.3 is removed. More specifically, the signal S_0 and the associated pass transistors, in Figure 2.3, are used to isolate the ISR and PSR from the PLA. When $S_0 = 0$, or the pass transistor is OFF, Table 2.1 shows that the signals S_1 through S_8 are also at logic 0's. This implies that the pass transistor control by S_0 is redundant.

The extra power line Vdd1, in the FDPLA of Figure 2.2 can also be eliminated. The power line Vdd1 was used to disable the input decoder in order to avoid the data conflict when we apply patterns from the ISR to the bit lines. In fact, the data conflict can be avoided by matching the patterns. Specifically, when enabling only the even-numbered bit line of the i-th input, i.e., loading 1 to this enabled bit line and 0's to other bit lines from ISRs, unlike Vdd1 is set 0 to disable the input decoders in [8], we may apply a logic 0 to the i-th input and 1's to other inputs. This application will not conflict the data loaded from the ISRs. Similarly, for enabling the odd-numbered bit line of the i-th input, a logical 1 is applied to the i-th input. Figure 3.3 illustrates the detail operation. Therefore, the BOTTOM input decoders in the unfolded AND plane do not need this extra power line Vdd1.

3.1.2 An ISR-like Structure for TOP Input Decoder

The TOP inputs of the SCFPLA, as shown in Figure 2.7, will be modified as an ISR-like structure. The modifications include the restructuring of the original SCFPLA and the use of multiplexing circuitry. The former allows enabling only one bit column of the folded part at a time, while the latter allows reading the contents of the bit columns.

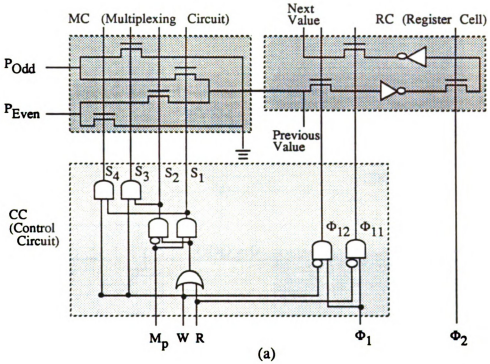


Figure 3.2 Control Circuits and Modified Cells of Shift Registers: (a) PSR; and (b) ISR.

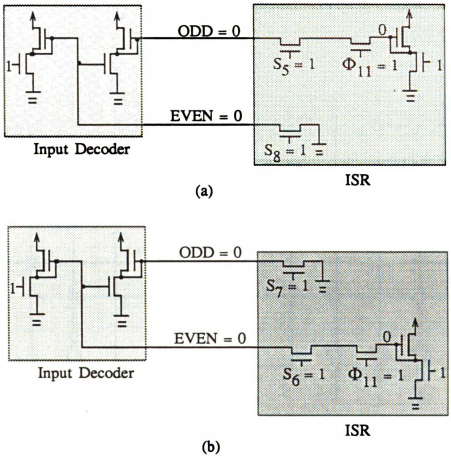


Figure 3.3 Circuit Diagram for "write" Mode ISR:
(a) $M_I = 1$; and (b) $M_I = 0$.

Figure 3.4 illustrates the schematic diagram of a restructured SCFPLA, or RSCFPLA. The RSCFPLA and the conventional SCFPLA, as shown in Figure 2.7, have the same OR plane and unfolded AND plane structure, but their folded AND plane structures are slightly different. In SCFPLA, the "cut" of the folded bit column is performed during the manufacturing process. In contrast, the RSCFPLA preserves the completeness of the folded bit columns, i.e., the "cut" process will be performed after the fault diagnosis and repair process.

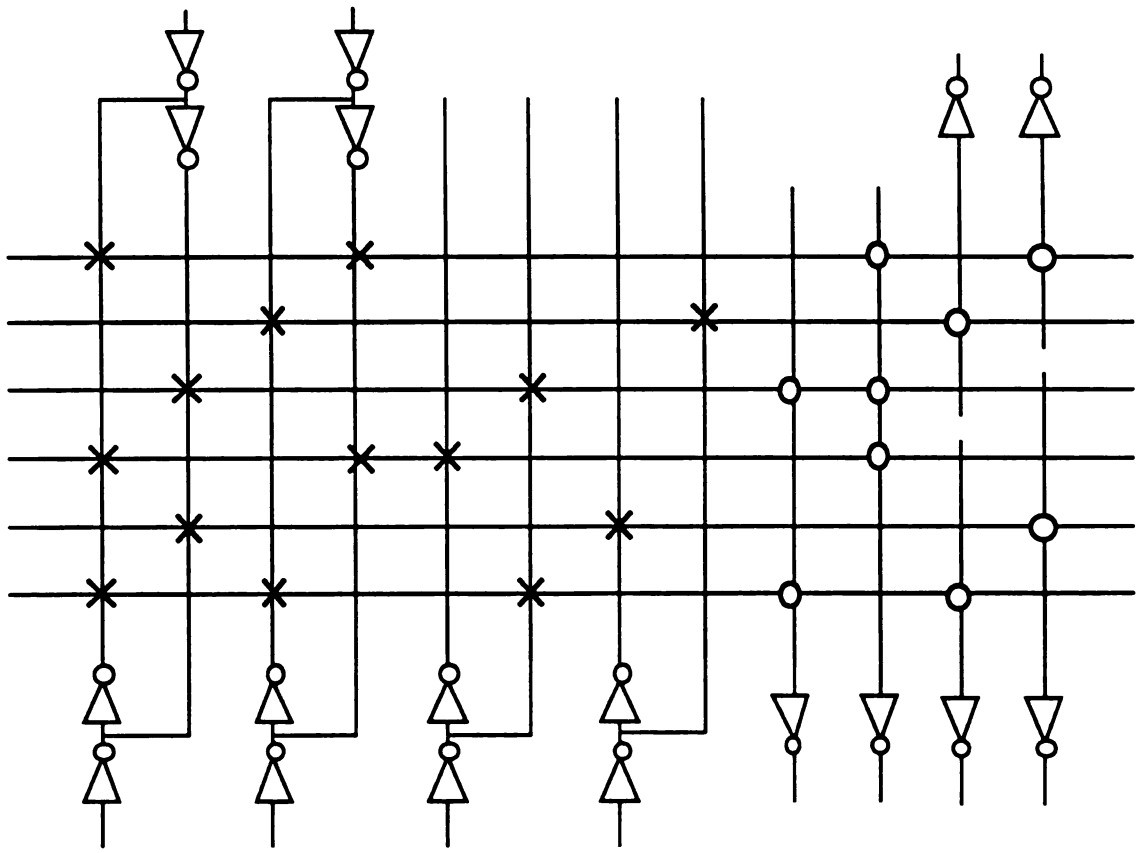


Figure 3.4 Schematic Diagram of a Restructured SCFPLA.

In order to enable only one bit column at a time, the bit column structure is also modified. Figure 3.5 (a) shows that, in a conventional SCFPLA, the true (complemented) bit line of a BOTTOM input decoder shares a bit column with the true (complemented) bit line of a TOP input decoder. In RSCFPLA, we switch over the true bit line and the complemented bit line of the TOP input decoder, as illustrated in Figure 3.5 (b). This modification gives the combined TOP and BOTTOM input decoders the capability to enable only one bit column at a time. Table 3.1 shows the combinations of TOP and BOTTOM inputs and the data being assigned to the bit columns for both the conventional SCFPLA and the RSCFPLA.

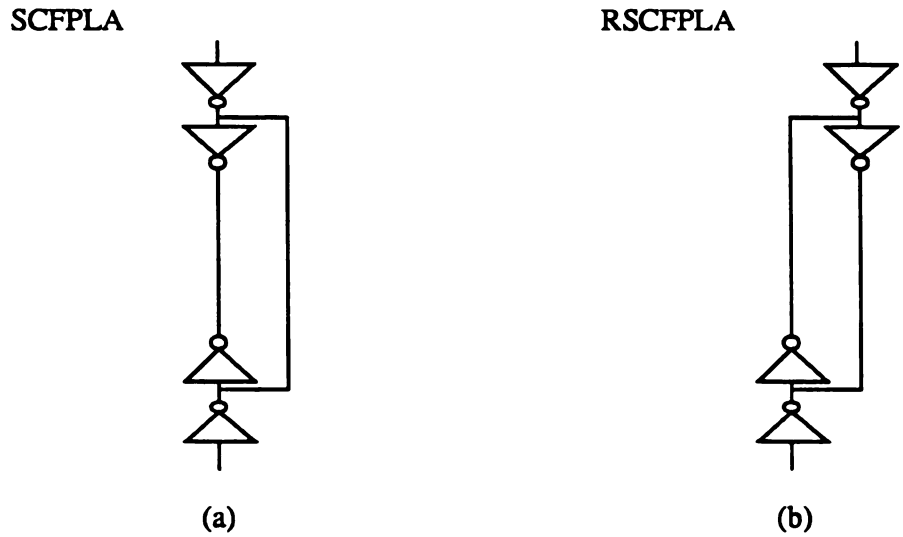


Figure 3.5 Bit Column Structures: (a) SCFPLA; and (b) RSCFPLA.

Table 3.1. Input Signal Combinations

BOTTOM INPUT	TOP INPUT	SCFPLA ODD	SCFPLA EVEN	RSCFPLA ODD	RSCFPLA EVEN
0	0	0	1	*	*
0	1	1	0	0	1
1	0	1	0	1	0
1	1	1	0	0	0

*: previous value

In a RSCFPLA, when all TOP and BOTTOM inputs hold 1's (except the i-th BOTTOM input which holds 0), only the (2i)-th bit column will have logic value 1 and all the others will have logic value 0's. This means that only the (2i)-th bit column is

enabled and all others are disabled. Conversely, when only the i-th TOP input holds 0, this enables only the (2i-1)-th bit column. In other words, the modification allows enabling only one bit column of the folded AND plane at a time.

It should be mentioned that BOTTOM input decoders of folded AND plane do not need the extra power line Vdd1. Since this proposed design allows applying signals from both TOP and BOTTOM inputs in the folded AND plane, it is not necessary to disable the BOTTOM inputs of the folded AND plane using Vdd1. As the BOTTOM inputs of the unfolded AND plane do not need Vdd1 either, the proposed fault-diagnosable design in Figure 3.1 does not require the extra power line Vdd1. Therefore, the number of extra signals needed in this revised design is less than in [8].

To read the content of each bit column in the folded AND plane, a multiplexing circuitry, as shown in Figure 3.6, is employed for each TOP input. According to ISR of Figure 3.2, the signal $\overline{W}$ is used to control the operation of "read" and "write", i.e., $\overline{W}$ = 0 reads the content of bit column, and $\overline{W}$ = 1 writes data to the bit column. The signals S_5 and S_6 are used to select the true or complemented bit columns to perform "read" function. During normal operation, or writing values to bit columns during the diagnosis and repair process, $\overline{W}$ is set to be 1 and $S_5 = S_6 = 0$, i.e., the circuit acts as a regular input decoder, as shown by the solid lines in Figure 3.6 (a). On the other hand, reading values from the bit columns during the diagnosis and repair process, $\overline{W}$ is set to be 0 and $(S_5, S_6) = (0,1)$ (or (1,0)) to read the content of the even (odd) bit column, as indicated by the dotted lines in Figure 3.6 (a). Figure 3.6 (b) shows the physical layout of the ISR-like structure, where each input takes 16 λwide and 90 λ long.

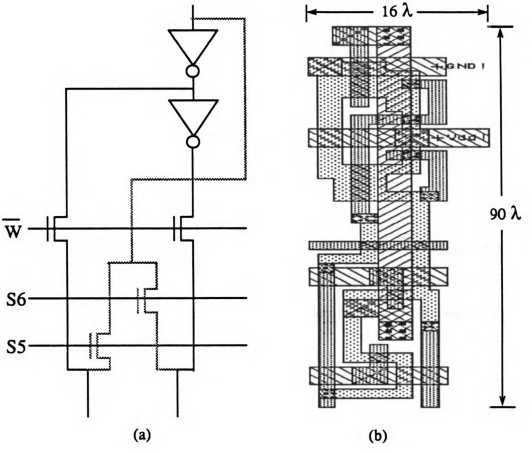


Figure 3.6 ISR-like TOP Input Decoder:
(a) Schematic Diagram; and (b) Physical Layout.

3.2 Fault-Repairable SCFPLA Design

To avoid complex routing and to repair the faulty PLA, spare input/output/product lines are added to the SCFPLA to replace the faulty lines, and two control circuits SISC and SOSC are also used for line reconfiguration.

3.2.1 Spare Input Column and SISC

Figure 2.4 shows that the SISC is added to the input portion of the conventional PLA between the input decoder and the AND plane. In this design, as shown in Figure 3.7, the SISC for the BOTTOM inputs is referred to as BSISC, while the SISC for the TOP inputs is referred to as TSISC.

When an input bit column that contains two bit lines is faulty, as indicated by the dotted line in Figure 3.7, both BSISC and TSISC are programmed, i.e., the Normal-OFF links are now ON and the Normal-ON links are now OFF, to switch these inputs to a spare input column. On the other hand, when an input bit column that contains only one input bit line is faulty, the same procedure is performed only for BSISC. After the SISCs have been reconfigured, we connect the faulty input columns to GND, to avoid the faulty input columns affecting the functions of the product lines, and disconnect the spare input columns from the GND.

3.2.2 Spare Output Column and SOSC

Similarly, Figure 3.8 shows that the TSOSC and BSOSC are added to the OR plane for the TOP and BOTTOM outputs, respectively. A Normal-OFF link between each output bit column and GND and a Normal-ON link between each spare output column and GND are also added. Notice that Normal-ON link between each output bit column and its pull-up transistor and a Normal-OFF link between each spare output column and its pull-up transistor in [8] are not necessary in this proposed design. This is because we put the pull-up transistor between the TSOSC/BSOSC and output inverter. Once we switch the TSOSC/BSOSC, the pull-up transistor has also been switched to the spare output column from the faulty output column. Therefore, the number of pull-up transistors in the OR plane of this design is less than in Figure 2.4 [8].

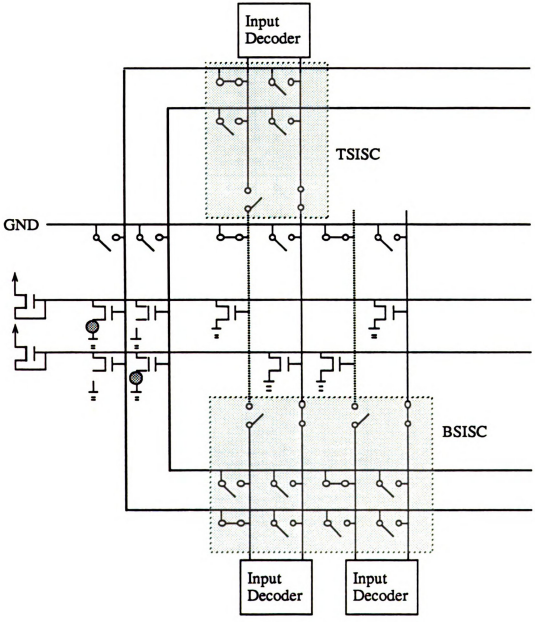


Figure 3.7 Schematic Diagram for SISC and Spare Input Column.

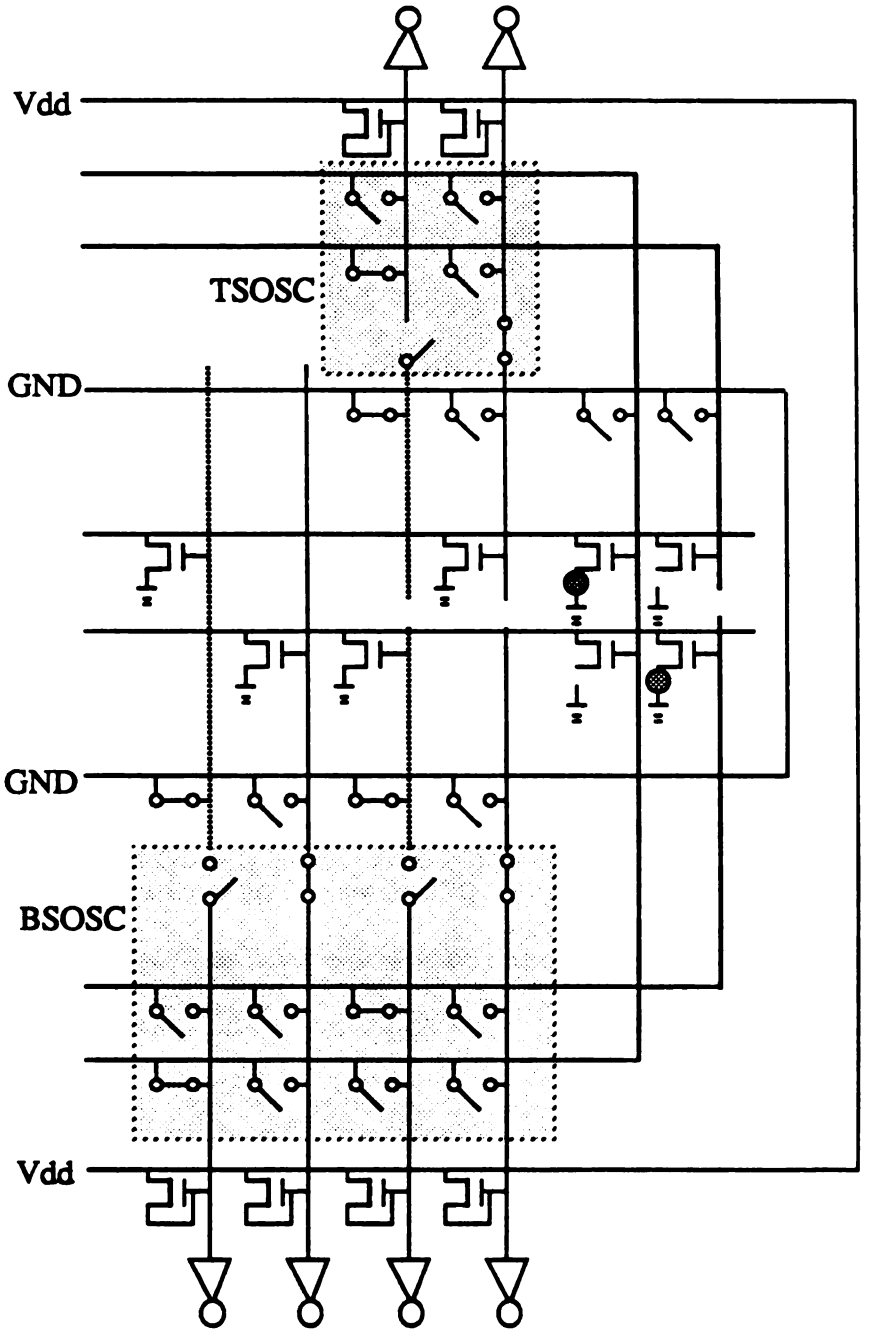


Figure 3.8 Schematic Diagram for SOSC and Spare Output Column.

3.2.3 Spare Product Lines

Figure 3.9 illustrates that both TOP spare product lines and BOTTOM spare product lines are added into the PLA. The TOP (or BOTTOM) spare line repairs the faulty product line which contains only TOP (or BOTTOM) inputs and outputs. Similar to the FRPLA design, this design also requires the following programmable links: Normal-ON link between each product line and its pull-up transistor, Normal-OFF link between each product line and GND, Normal-OFF link between each spare product line and its pull-up transistor, and Normal-ON link between each spare product line and GND.

When a faulty product line is detected and located, it is first disconnected from its pull-up transistor by programming the Normal-ON link to OFF and then it is connected to GND by programming the Normal-OFF link to ON. In this operation, two cases are identified:

1. If this faulty product line connects to only bottom side (or top side), a BOTTOM (or TOP) spare product line is programmed and this spare line is connected to its pull-up transistor and disconnected from GND by programming its links, or
2. If this faulty product line connects to both bottom side and top side, it is repaired by one TOP spare product line and one BOTTOM spare product line.

3.2.4 Repair Rules

Since the defects that are likely to occur in the SCFPLA are similar to those in the conventional PLAs, the repair rules for the SCFPLA are the same as the ones in Table 2.2. The unique fault that occurs in an SCFPLA is a bridging fault caused by the "cut" process. Specifically, in an SCFPLA, a "cut" is applied to a bit (or output) column that is shared by two bit (or output) lines. An improper "cut" process (i.e., the line is cut

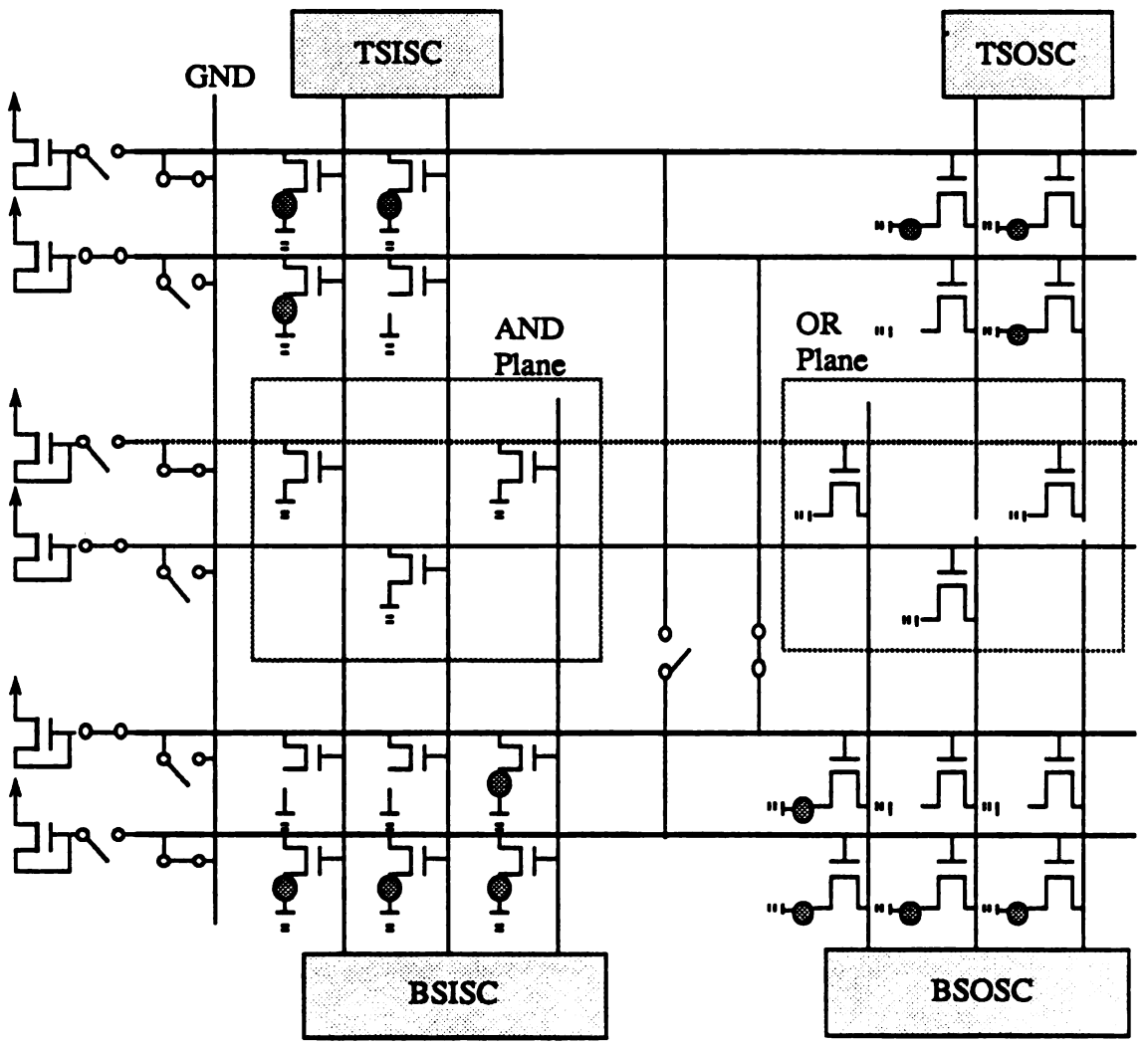


Figure 3.9 Schematic Diagram for Spare Product Lines.

incompletely) would introduce a bridging fault between two lines which is difficult to test in the conventional folded PLAs. This fault can be detected and located in the proposed fault-tolerant design. The repair of this fault is described in the fault-diagnosis and repair process.

3.3 Fault Diagnosis and Repair Process

According to the fault models and the repair rules described in the previous sections, fault diagnosis and repair process, as shown in Appendix 1, is proposed to locate and repair all single and multiple crosspoint, stuck-at, and bridging faults.

3.4 Design Evaluation

Figure 3.10 shows a floor plan of the fault-tolerant SCFPLA. The area of the fault-tolerant SCFPLA is estimated by the following formula:

$$\begin{aligned} \text{Area} = {} & (16n_b + 16s_n + 30)(22s_p + 8p + 8s_n + 43) + (8m_b + 22s_m + 21)(8s_m + \\ & 22s_p + 8p + 81) + 8m_t \times 71 + 140(8p + 16n_b - 16n_t) + 16n_t \times 90 + \\ & 16n_t(8s_n + 10) + 8m_t(8s_m + 10), \end{aligned} \tag{3.1}$$

where n_t : number of TOP input lines, s_n : number of spare input lines,
n_b : number of BOTTOM input lines,
m_t : number of TOP output lines, s_m: number of spare output lines,
m_b: number of BOTTOM output lines.
p : number of product lines, s_p: number of spare product lines.

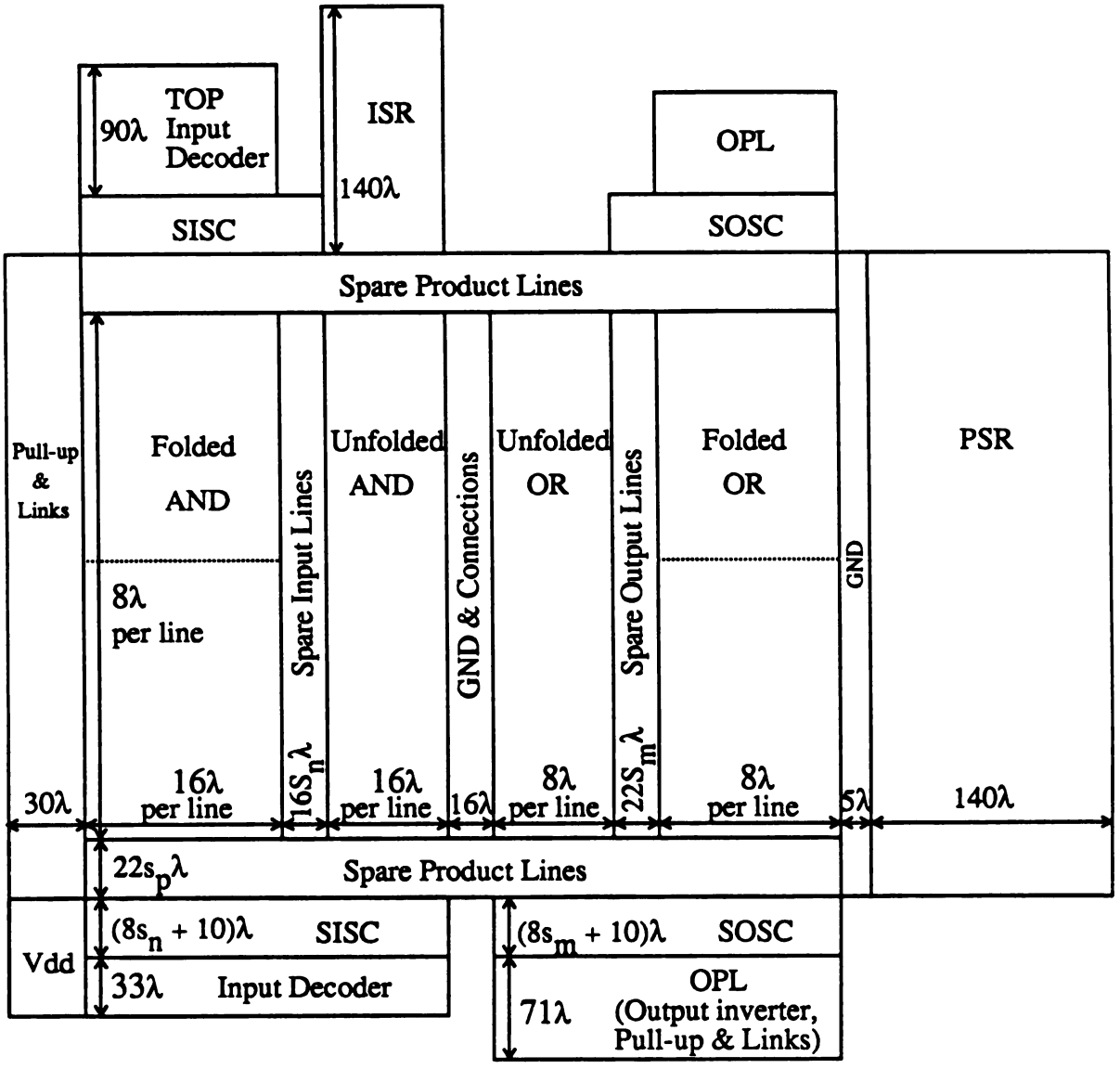


Figure 3.10 Floor Plan of the Fault-Tolerant SCFPLA.

In order to demonstrate the effectiveness of the proposed design in area reduction, the areas of various benchmark PLAs [28] are estimated. The first four columns of Table 3.2 lists the PLA names, the number of inputs (n), outputs (m), and product terms (p). The chip areas for (1, 2, 1)-FTPLAs are estimated in accordance with the floor plan shown in Figure 2.6. Columns 6 and 7 show the number of BOTTOM inputs and outputs computed using the PLEASURE program. Based on the floor plan shown in Figure 3.10, the areas of (1, 2, 1)-SCFPLA are calculated from Equation (3.1). Column 8 compares the area ratio of the proposed design versus the FTPLA design. The results show that the proposed design reduces the chip area as much as 47%. This implies that the number of die that can be placed in a wafer is doubled, thereby increasing the overall chip yield.

According to the physical layouts for both the conventional PLA design and the proposed design for the PLA "mish", as shown in Figure 3.11, the proposed design not only provides the fault tolerance, but also consumes less chip area.

3.5 Fully Testable SCFPLA Design

The key to the fully testable PLA design is the use of additional hardware to enable only one product line at a time to make the PLA fully testable [21, 29]. Although the internal structure of SCFPLA is a little different from the conventional PLA, the Input/Output (I/O) relation still remains the same. As a result, from the I/O's function, we may not even notice the slight modification of the structure. Since the fault-tolerant SCFPLA design contains PSRs to enable or disable each product line, this makes the proposed design fully testable by the pin overhead from signals W, M_p, and Sin. In other words, the proposed design achieves a full diagnosability during the manufacturing process. On the other hand, after the chip is packaged, the proposed design is turned into a fully testable design.

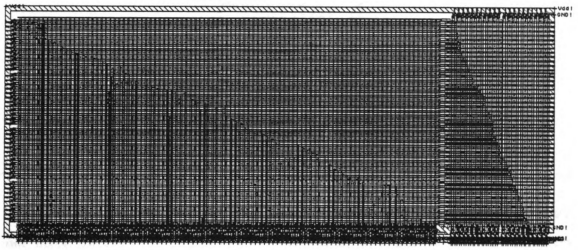
Table 3.2 Simulation Results (SCFPLA)

name	n	m	p	(1,2,1)-FTPLA Area	(1,2,1)-SCFPLA n_b	m_b	%
5xp1	7	10	65	270024	7	6	91.8
9sym	9	1	84	440832	X		
add6	*						
adr4	8	5	75	288312	8	3	94.8
alu1	12	8	19	143784	7	4	67.9
alu2	10	8	68	308080	10	5	93.7
alu3	10	8	66	300832	10	4	92.1
apla	10	12	25	162808	10	7	90.7
bc0	21	11	179	1046152	17	10	87.5
bca	16	46	180	1354736	16	31	86.3
bcb	16	39	156	1113792	16	28	89.1
bcc	16	45	137	1051496	16	30	86.3
bcd	16	38	117	856600	16	22	84.5
chkn	29	7	140	979440	24	4	84.4
ck	4	7	9	60488	3	5	83.4
co14	14	1	14	206832	X		
cps	24	102	162	2070304	21	64	76.2
dc1	4	7	9	60488	4	5	92.3
dc2	8	7	39	181048	7	4	87.6
dist	8	5	120	431232	8	4	97.1
dk17	10	11	18	133456	10	6	90.3
dk27	8	9	10	88592	8	5	90.4
dk48	15	17	21	193688	15	9	89.1
exep	28	62	109	1216408	24	31	74.3
f51m	8	8	76	309200	8	4	91.8
gary	15	11	107	562600	14	8	91.6
in0	15	11	107	562600	14	8	91.6
in1	15	17	104	595408	14	17	95.8
in2	19	10	135	763640	16	8	87.9

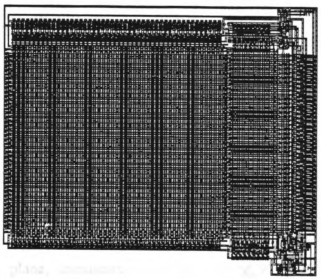
X : SCFPLA cannot be obtained by PLEASURE.
* : Number of care or column exceeds the PLEASURE limit.

Table 3.2 Continued

name	n	m	p	(1,2,1)-FTPLA Area	(1,2,1)-SCFPLA n_b	m_b	%
in3	34	29	74	779440	24	19	73.8
in4	32	20	212	1694800	23	10	74.2
in5	24	14	62	486400	16	11	74.9
in6	33	23	54	580544	18	14	62.5
in7	26	10	54	444256	16	7	69.4
jbp	36	57	122	1452752	26	31	70.6
misg	56	23	69	993384	28	12	55.2
mish	94	34	82	1757440	47	17	52.7
mlp4	8	8	127	480968	8	6	95.5
opa	17	61	79	771272	15	40	80.6
radd	8	5	75	288312	8	3	94.8
rckl	32	7	32	357216	32	6	95.0
rd53	5	3	31	117992	5	2	95.2
rd73	7	3	127	414664	7	2	96.9
risc	8	31	28	212672	7	16	79.3
root	8	5	57	231144	8	4	96.3
sqn	7	3	38	218288	X		
sqr6	6	11	50	213136	6	6	90.0
ti	43	67	213	2737784	30	38	69.5
tial	*						
vg2	25	8	110	734608	25	4	94.3
wim	4	7	9	60488	3	6	84.9
x1dn	27	6	110	755024	26	3	92.9
x2dn	82	47	104	2000128	45	24	57.6
x6dn	38	5	81	744648	24	5	70.0
x7dn	*						
x9dn	27	7	120	820464	26	4	93.1
z4	7	4	59	220920	7	2	94.4



(a)



(b)

Figure 3.11 Layouts of the PLA "mish":
(a) Conventional; and (b) Fault-Tolerant SCFPLA.

Chapter 4

Automatic Layout Generator

This chapter describes an automatic layout generator, ALGFPLA, that generates the layout mask for fault-tolerant SCFPLAs. The layout generator is built on the MPACK library [30] using MAGIC graphics editor. ALGFPLA has been implemented on SUN 3/160 under UNIX operating system.

4.1 Development

ALGFPLA consists of two major steps: (1) create a template; and (2) compile PLEASURE's symbolic output into layout masks. ALGFPLA requires a template to generate its fault-tolerant SCFPLAs. To make an ALGFPLA template, a sample fault-tolerant SCFPLA layout is designed. This sample includes at least one example of each possible combination of template tiles. It also contains all possible features, such as: the TOP and BOTTOM inputs and outputs, transistors connected to true and complemented signals in the AND plane, transistors connected to output signals in the OR plane, the spare input/output/product lines and control circuits for repairable design, and the shift registers and control circuits for fault-diagnosable design. With this template, ALGFPLA can then generate a large fault-tolerant SCFPLA. Figure 4.1 shows a template of a fault-tolerant SCFPLA in NMOS technology with two metal layers.

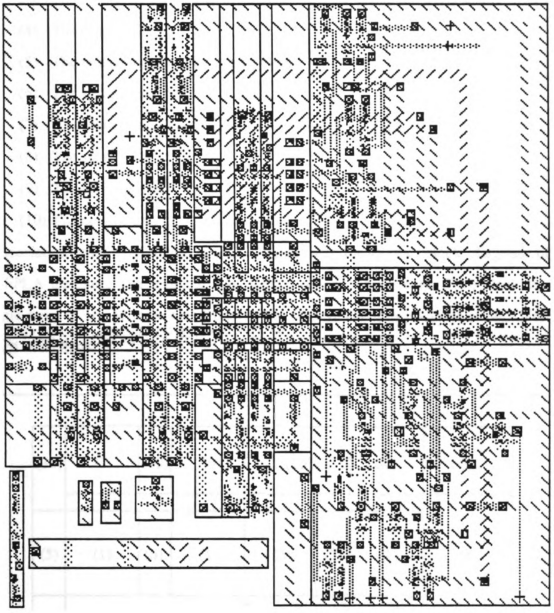


Figure 4.1 An ALGFPLA Template for Fault-Tolerant SCFPLAs.

The ALGFPLA template of Figure 4.1 is composed of the following ten blocks.

(1) The core of the AND plane.

(2) The core of the OR plane.

(3) The input.

(4) The output.

(5) The pull-up transistor of product line.

(6) The spare input line.

(7) The spare output line.

(8) The spare product line.

(9) The shift register.

(10) The connection.

In association with the above blocks, Figure 4.2 shows the block diagram of the ALGFPLA template. The template may contain one or more blocks. Each block signifies a single array of one or more tiles from the tile library. The tiles involved in each block are defined as follows.

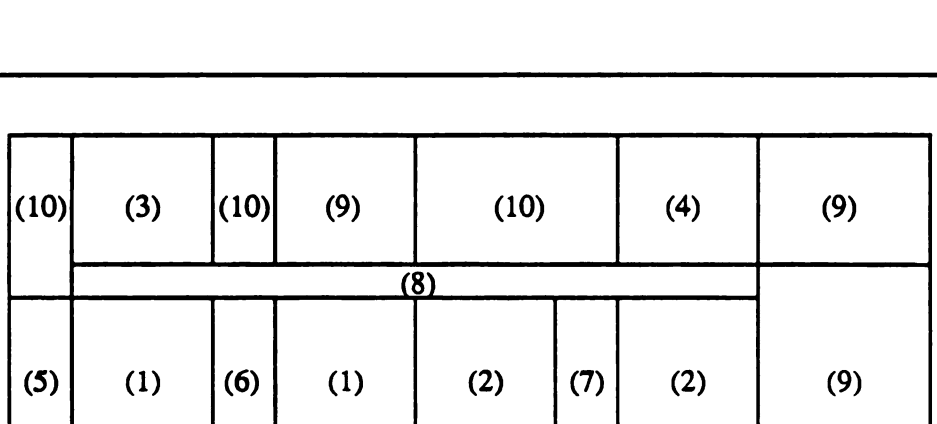


Figure 4.2 Block Diagram for the Template.

(1) The core of the AND plane (or AND-core):

Basically, the AND-core consists of input lines, product lines, and crosspoints. A crosspoint may present in either the true or the complemented bit line. It is also possible that no crosspoint presents in either line. Since the pull-down transistor, if present, is connected to ground, it is necessary to provide a contact for the pull-down transistor. Moreover, a contact may be shared by adjacent cross points to save the area.

In AND-core, three tiles *and_l*, *and_r*, and *and_null* are created for the crosspoint possibility and two tiles *and_noc*, *and_con* are for the contact possibility, as shown in Figure 4.3, where the polysilicon line and metal line represent the input bit line and product line, respectively, while the diffusion line is the GND line.

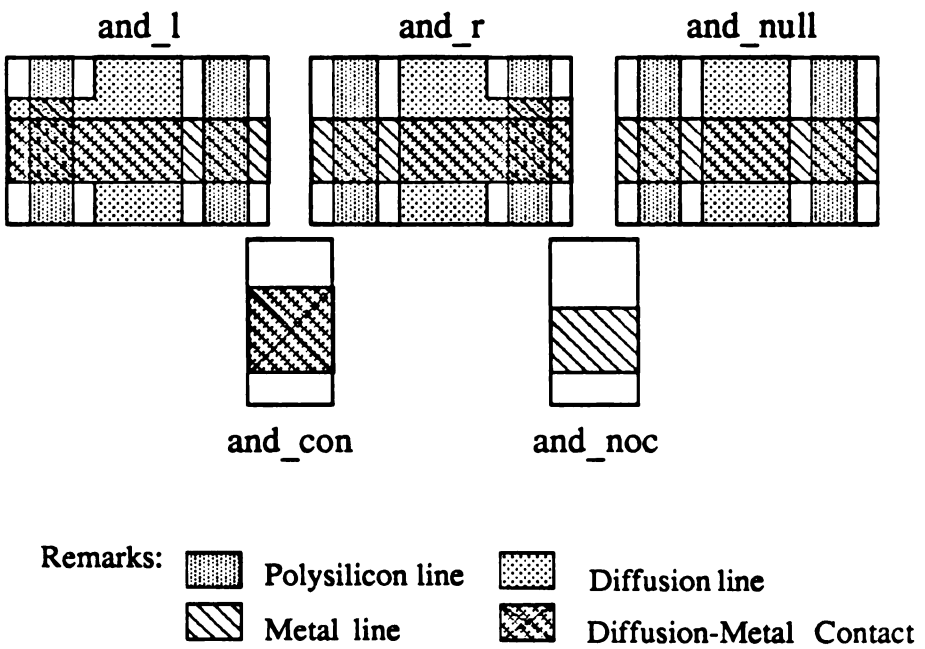


Figure 4.3 Tiles in the AND-Core.

In PLEASURE's data format, '1' or '!' physically means that a pull-down transistor exists between a product line and a true bit line. Thus, it is realized by three tiles (*and_con, and_l,* and *and_noc*), as shown in Figure 4.4. Similarly, a '0' or 'o' is realized by (*and_con, and_r,* and *and_con*), while the '-' or '_' is realized by (*and_noc, and_null,* and *and_noc*). Figure 4.5 illustrates an example of ALGFPLA compiling the AND array of the PLEASURE's symbolic format into layout mask. Notice that the adjacent tiles are overlapped as shown to save area.

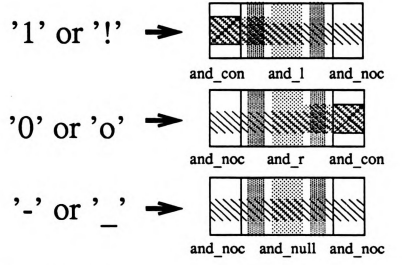


Figure 4.4 Relation Between the AND Array and the Cells.

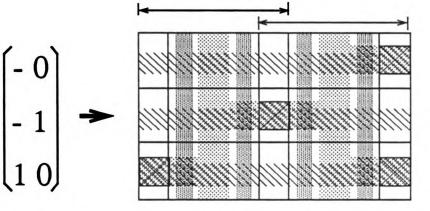


Figure 4.5 An Example of the Core of AND Plane.

(2) The core of the OR plane (or OR-core):

The OR-core consists of output lines, product lines, and crosspoints. Since the even- and odd-numbered product lines share the same ground line, the product lines pair is included in a single tile. This results in four different possibilities for the pull-down transistors: a transistor on the odd-numbered product line only, a transistor on the even-numbered product line only, transistors on both even- and odd-numbered product lines, and no transistor at all. Therefore, four tiles (*or_ud, or_d, or_u, or_null*) for these four possibilities, and two tiles (*or_noc, or_con*) for the connection between tiles are created, as shown in Figure 4.6, where the polysilicon and metal lines represent the product and output lines, respectively, while the diffusion line is a GND line.

Figure 4.7 illustrates the four possible combinations of tiles for the OR array. For example, in PLEASURE's data format, a 'I' means that a pull-down transistor exists between the output and the product lines. Therefore, two consecutive 'I's in an output column are implemented by the tiles *(or_con, or_ud,* and *or_con*). Figure 4.8 illustrates an example of ALGFPLA compiling the OR array in PLEASURE's data format into layout mask. Notice that the consecutive connection tiles in a same column are overlapped.

In the proposed SCFPLA, the "cut" for the folded part of the OR array is required in the layout mask. Similar to Figure 4.6, Figure 4.9 presents the five tiles (*or_fud, or_fu, or_fd, or_fnull, or_fnoc*) that are used to define the portion where the "cut" is needed. Figure 4.10 shows the four possible combinations of the PLEASURE's data format and the layout mask implementing these five basic tiles. Figure 4.11 gives an example of ALGFPLA compiling the personality matrix into layout mask.

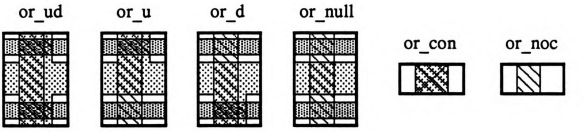


Figure 4.6 Basic Tiles in the OR-Core.

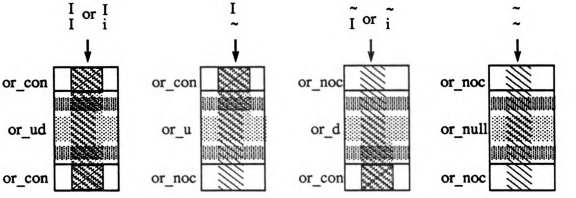


Figure 4.7 Relation Between the OR Array and the Basic Cells.

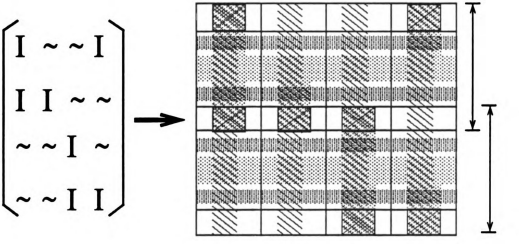


Figure 4.8 An Example of the Core of OR Plane.

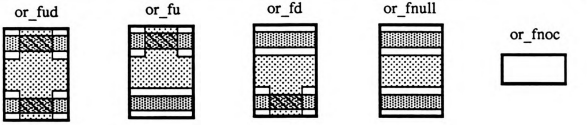


Figure 4.9 Special Tiles in the OR-Core.

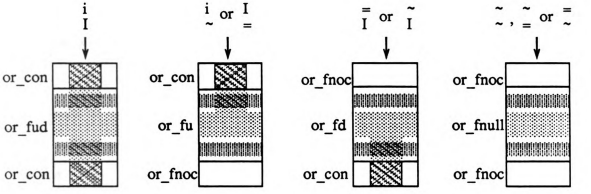


Figure 4.10 Relation Between the OR Array and the Special Cells.

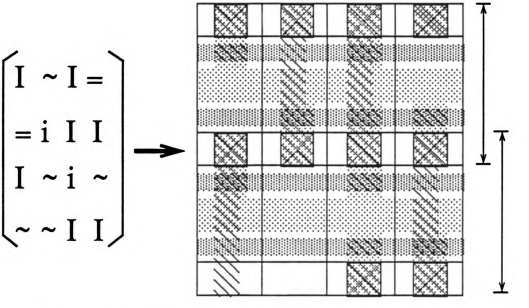


Figure 4.11 An Example for the Special Tiles in the Core of OR Plane.

(3) - (5) The input and output lines, and pull-up transistor of product line:

Figure 4.12 illustrates the tiles involved in these blocks. As shown in Figure 4.12 (a), two tiles (*input1, input2*) are used for the BOTTOM and TOP inputs, respectively. For the output lines, both tiles (*output1_s, output2_s*) are respectively represented for the TOP and BOTTOM output lines. Each output line takes 8 λ in width. In our implementation, in order to fit two adjacent output lines into a 16 λ width, both lines are compacted. Thus, the tiles (*input1_d, input2_d*) represent two adjacent output lines on both top and bottom sides as shown in Figure 4.12 (b).

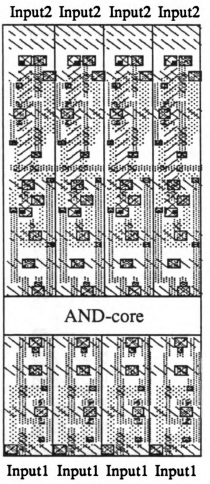
Since a product line only takes 8 λ in height, the pull-up transistor must also be designed 8 λ in height. In order to obey the design rule, two types of pull-up transistor tile (*pdt_pulup1, pdt_pullup2*) are needed, as shown in Figure 4.12 (c).

(6) The spare input line:

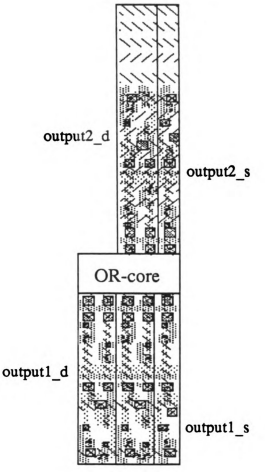
A spare input line can be partitioned into several blocks. Each block is composed of one or more tiles. Specifically, in Figure 4.13, the tile *spare_input1* is used for the connection corner of SISC and spare input from BOTTOM input, while the *spare_input4* is for the TOP input. The tile *spare_input3* is for the cross section of a spare input line and a spare product line. Finally, the tile *spare_input2* is for the regular product lines.

(7) The spare output line:

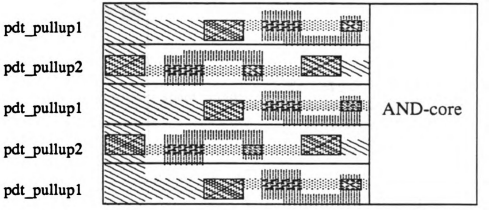
Figure 4.14 illustrates the tiles involved in this block. The tile *spare_output3* is the connection corner of SOSC and spare output line. The tile *spare_output6* is the cross section of a spare output line and a spare product line. As discussed in Figure 4.6, the odd-numbered and even-numbered product lines form a pair. A product line pair on the spare output line is constructed using two tiles (*spare_output1* and *spare_output2*). As the layout is painted



(a)



(b)



(c)

Figure 4.12 Examples: (a) Inputs; (b) Outputs; and
(c) Pull-up transistors for product lines.

from the bottom to the top side, if the number of product lines is odd, the tiles (*spare_output1* and *spare_output4*) are used for the single product line left out from the pairs. The tile *spare_output5* is for the connection corner of SOSC and the spare output line. Finally, if the number of product lines is even, the tile *spare_output7* represents the corner of SOSC and the spare output line.

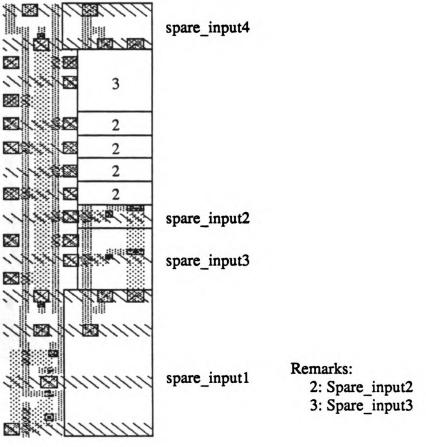


Figure 4.13 An Example of a Spare Input Line.

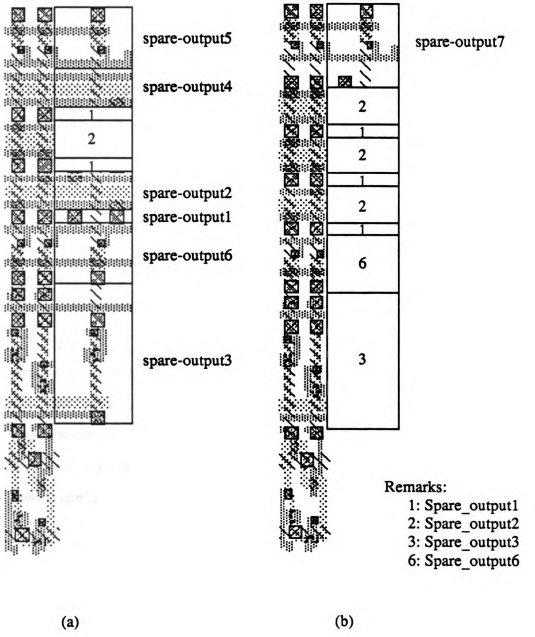


Figure 4.14 Examples of Spare Output Lines:
(a) Odd Number of Product Lines; and
(b) Even Number of Product Lines.

(8) The spare product line:

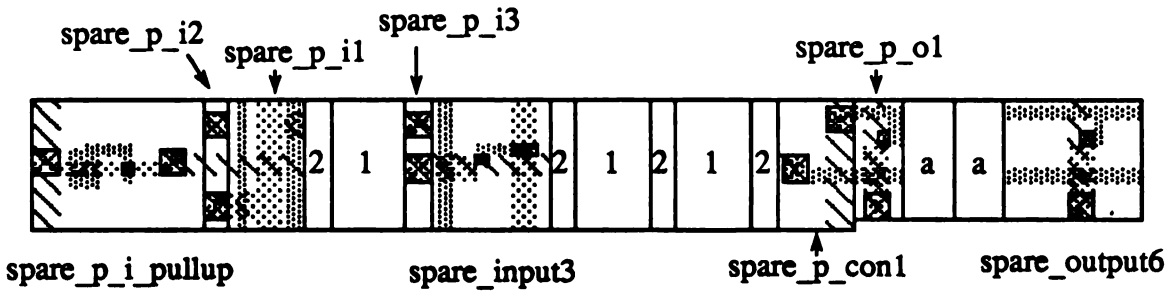
The bottom-side spare product line, as shown in Figure 4.15 (a), can be partitioned into following blocks (the tiles required for each block is given in parentheses): a pull-up transistor (*spare_p_i_pullup*), cross section of input bit lines (*spare_p_i2*, *spare_p_i1*), spare input lines (*spare_input3*), interconnection between two planes (*spare_p_con1*), cross section of output lines (*spare_p_o1*), and spare output lines (*spare_output5*).

As shown in Figure 4.15 (b) and (c), the top-side spare product line has the same partitioning as the bottom-side except the tiles in the OR array, such as *(spare_p_con2*, *spare_p_o2*, *spare_output5*) in the situation of odd number of product lines and *(spare_p_con3*, *spare_p_o3*, *spare_output7*) in the situation of even number of product lines.

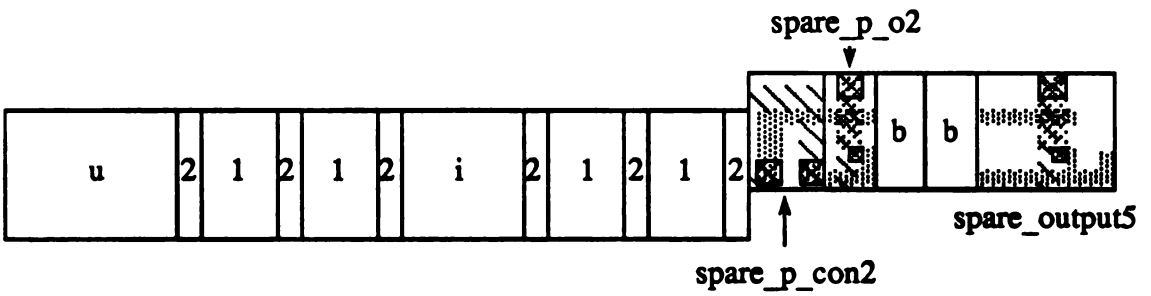
(9) & (10) The shift register and connection:

Figure 4.16 shows that the shift registers PSR and ISR and the control circuits are partitioned into four basic tiles: (*psr*, *isr*, *psr_control*, *isr_control*).

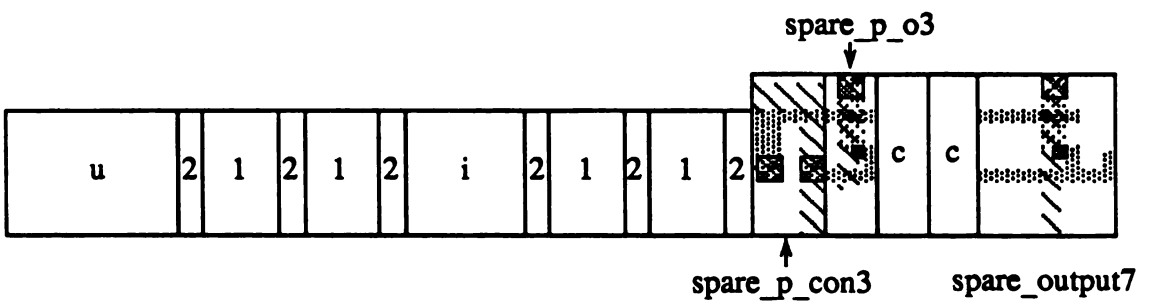
The remaining tiles are required for tile interconnection. In this implementation, 15 tiles are created. They are *(or_gnd_con*, *or_gnd_noc*, *and_or_con_d*, *con1*, *con2*, *con3*, *con4*, *con5*, *con6*, *con7*, *con8*, *con9*, *con10*, *con11*, *con12*).



(a) Bottom Side Spare Product Line



(b) Top Side Spare Product Line (Odd Number of Product Lines)



(c) Top Side Spare Product Line (Even Number of Product Lines)

Remarks:

1: Spare_p_i1	a: Spare_p_o1
2: Spare_p_i2	b: Spare_p_o2
3: Spare_p_i3	c: Spare_p_o3
u: Spare_p_i_pullup	i: Spare_input3

Figure 4.15 Examples of Spare Product Lines: (a) Bottom Side;
(b) Top Side (Odd Number of Product Lines); and
(c) Top Side (Even Number of Product Lines).

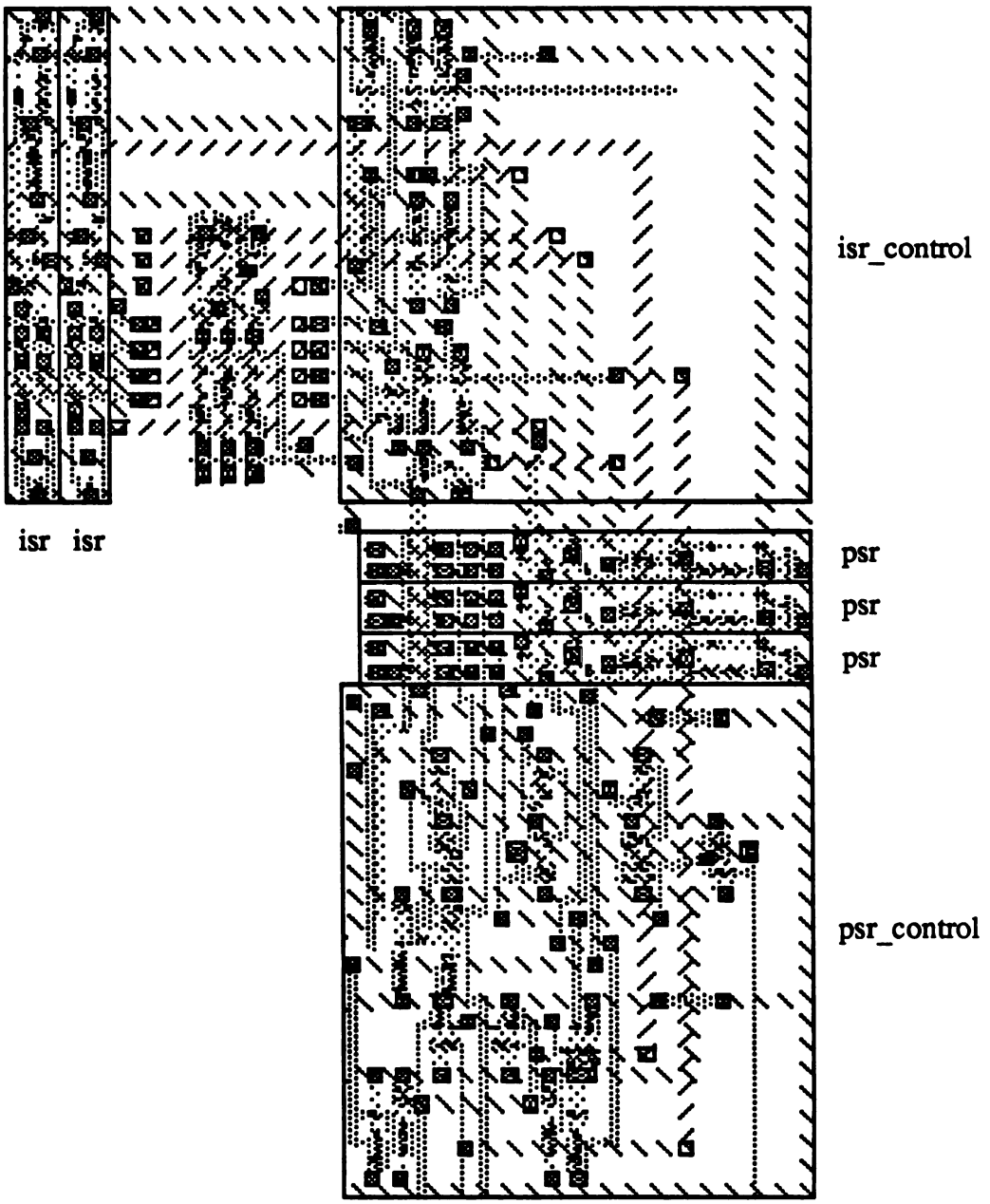


Figure 4.16 An Example of Shift Registers and Their Control Circuits.

4.2 Algorithm and Examples

Algorithm I illustrates the major steps in ALGFPLA, and the program coded in "C" is given in Appendix 2. The program assembles tiles into the desired module. Typically, the program reads a file (in PLEASURE's data format) and then calls the tile placement routine in the MPACK library.

Algorithm I:

Step 1: Initialization.

1.1: Process command line arguments.

1.2: Create a new tile.

1.3: Load in the template tiles.

Step 2: Compile input data.

2.1: Read the input data.

2.2: Separate folded columns from unfolded columns.

2.3: Restructure input bit columns.

Step 3: Layout generation

3.1: Paint and place tiles.

3.2: Print the generated layout mask.

The program must first include the file *mpack.h* which defines the interface to the MPACK library. Next, the *TPinitialize* procedure is called to process command line arguments, open an input file, and load in a template. The routine *TPcreate_tile(name)* is to create a new, empty tile and give the name "*name*". This is followed by loading in the template tiles by the routine *TPname_to_tile(name)* that assigns a unique ID for the tile. For example,

tinput1 = TPname_to_tile("input1");

The pointer *tinput1* points to the tile "*input1*" of Figure 4.12 (a).

Next, the program reads the input data and computes where to place the next tile. In our implementation, the input data in PLEASURE format is sorted to separate the folded part from unfolded part. In addition, the bit columns are restructured as discussed in Figure 3.5. After the input data are compiled, the painting and placement routine, *TPpaint_tile*, is carried out. For example,

x = TPpaint_tile (a, b, c);

means that the tile "*a*" is painted into the tile "*b*" such that its lower left corner is placed at the position "*c*" in the tile "*b*". In this implementation, the PLA is painted by the following sequence: the lower part, middle part, and upper part. When all tiles are placed the program calls the routine *TPwrite_tile* to create the output file.

Two examples are given below to demonstrate the ALGFPLA. The first example is to illustrate the use of routines in MPACK while the second is to show our layout generator.

Example 1:

A code constructing a set of tiles in a layout of Figure 4.17 is given as follows.

```
/* program which generates the layout in Figure 4.17 */
position.x = 20; position.y = 30;
rect = TPpaint_tile (tpdt_pullup1, output, align(position, tLL(tpdt_pullup1)));
rect = TPpaint_tile (tand_con, output, align(rLR(rect), tLL(tand_con)));
rect = TPpaint_tile (tand_r, output, align(rLR(rect), tLL(tand_r)));
rect = TPpaint_tile (tand_con, output, align(rLR(rect), tLL(tand_con)));
rect = TPpaint_tile (tand_l, output, align(rLR(rect), tLL(tand_l)));
/*-------------------------------------------------------------------------------------------*/
```

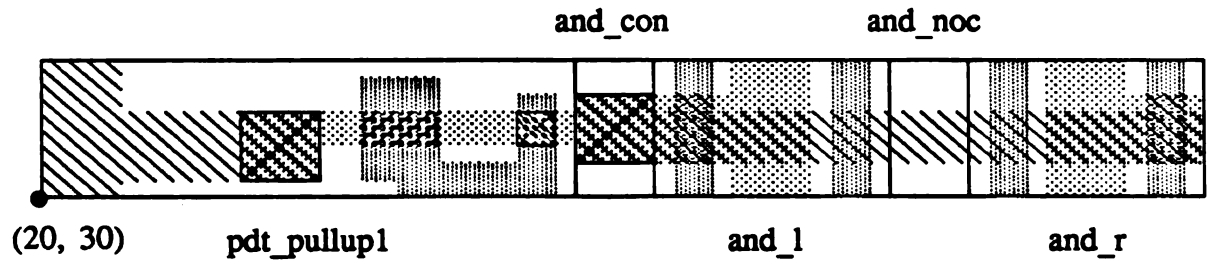


Figure 4.17 Example 1.

The code illustrates that the lower left corner of the layout is located at (20, 30). The tiles *pdt_pullup1*, *and_con*, *and_r*, *and_con*, and *and_l* are placed side by side. Note that *rLR()* specifies the location of the lower right corner of the tile, and *align()* computes the location for placement of the tile.

Example 2:

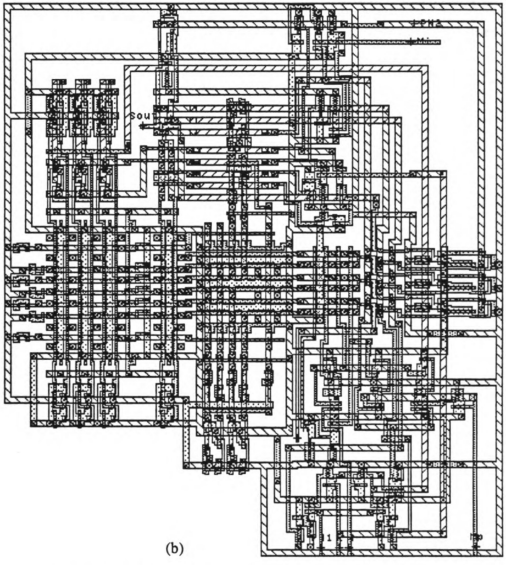
Consider an input data (in PLEASURE format), as shown in Figure 4.18 (a), the program, as listed in Appendix 2, compiles the data and generates the layout mask shown in Figure 4.18 (b).

Input:

```
1 - 0 1  I ~ ~ I
! 1 0 o  i I i ~
0 ! - -  ~ ~ I I
1 - 1 1  ~ I ~ I
0 1 0 0  I ~ I ~
```

(a)

Output:



(b)

Figure 4.18 Example 2: (a) Personality Matrix; and
(b) Layout generated by ALGFPLA.

4.3 Summary and Discussion

An automatic layout generator MRPLA was developed for repairable PLAs [8]. Basically, the layout generator was built on the MQUILT [30] routine. The major limitations of MRPLA are: (1) the use of single character symbols limit the number of possible named tiles, and (2) the requirement that tiles have the same height. These limitations result in increasing complexity for creating the tiles, particularly for complicated layouts like folded PLAs. In this study, the proposed layout generator, built on MPACK, allows us to define as many tiles as needed, place adjacent tiles with different height, and paint the tiles to any location.

Chapter 5

Fault-Tolerant Designs of PLAs with Other Folding Techniques

This chapter describes the fault-tolerant designs of PLAs with other folding techniques: SRFPLA-A, SRFPLA-O, and MCFPLA. Each fault-tolerant design includes the fault-diagnosable and repairable design. A comparison of fault-tolerant designs with various folding techniques is also provided.

5.1 Fault-Tolerant Design of SRFPLA-A

In a SRFPLA-A, as shown in Figure 2.8, the PLA is constructed with AND-OR-AND structure. Its inputs come into both AND planes and outputs come out in the middle OR plane. This reduces the height of the PLA, thereby reducing the PLA area. The repairable SRFPLA-A design can be accomplished by adding SISCs, SOSCs, and spare lines, as shown in Figure 5.1 where all dimensions are given in units of λ. Based on this floor plan, the area of the repairable SRFPLA-A design is:

$$\text{Area} = (16s_n + 22s_m + 16n + 8m + 92)(8p + 22s_p + 27) + (8s_n + 44)(16s_n + 16n + 60) + (8s_m + 51)(8p + 22s_m),$$

where
- n : number of input lines,
- m : number of output lines,
- p : number of product lines,
- s_n : number of spare input lines,
- s_m: number of spare output lines,
- s_p: number of spare product lines.

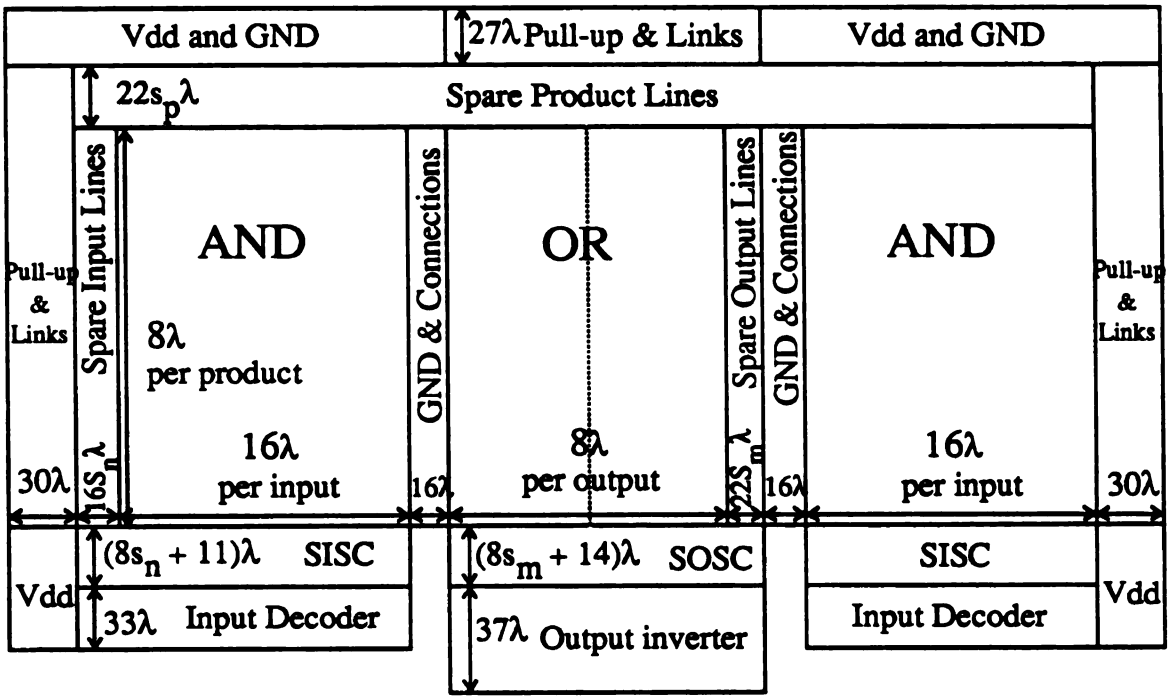


Figure 5.1 Floor Plan of the Fault-Repairable SRFPLA-A.

The fault diagnosability of a SRFPLA-A is achieved by adding the PSRs and ISRs to it. However, the separation of each row should be preserved until the fault diagnosis and repair process is completed. Since each shift register takes 16×140 λ in area, the total area for the fault-tolerant SRFPLA-A is:

Area = (area of fault-repairable SRFPLA-A) + $(2n + p) \times 8 \times 140$.

Table 5.1 lists the simulation results of the fault-tolerant SRFPLA-A design for various PLAs. Column 5 calculates the chip area required for the (1, 2, 1)-FTPLA design. With the applications of PLEASURE, the number of product lines computed for SRFPLA-A is given in column 6. The last column shows the ratio of the required chip area for the proposed fault-tolerant SRFPLA-A design over that for the FTPLA design.

Table 5.1 Simulation Results (SRFPLA-A)

name	n	m	p	(1,2,1)-FTPLA area	(1,2,1)-SRFPLA-A p	%
alu1	12	8	19	143784	13	96.5
cps	24	102	162	2070304	161	103.9
in4	32	20	212	1694800	211	109.7
in5	24	14	62	486400	61	109.9
in6	33	23	54	580544	50	101.8
in7	26	10	54	444256	52	108.3
jbp	36	57	122	1452752	114	99.7
misg	56	23	69	993384	36	68.2
mish	94	34	82	1757440	42	65.2
x2dn	82	47	104	2000128	64	71.4

5.2 Fault-Tolerant Design of SRFPLA-O

Figure 5.2 shows the floor plan of a repairable design for the SRFPLA-O. Together with the shift registers, for fault-diagnosable design, the area of the fault-tolerant SRFPLA-O design is:

$$\text{Area} = (16s_n + 22s_m + 16n + 8m + 92)(8p + 22s_p + 27) + (8s_n + 44)(16s_n + 16n) + (8s_m + 51)(22s_m + 8m + 60) + (2n + p) \times 8 \times 140,$$

where n : number of input lines, s_n : number of spare input lines,
m : number of output lines, s_m: number of spare output lines,
p : number of product lines, s_p: number of spare product lines.

Table 5.2 lists the simulation results for various PLAs.

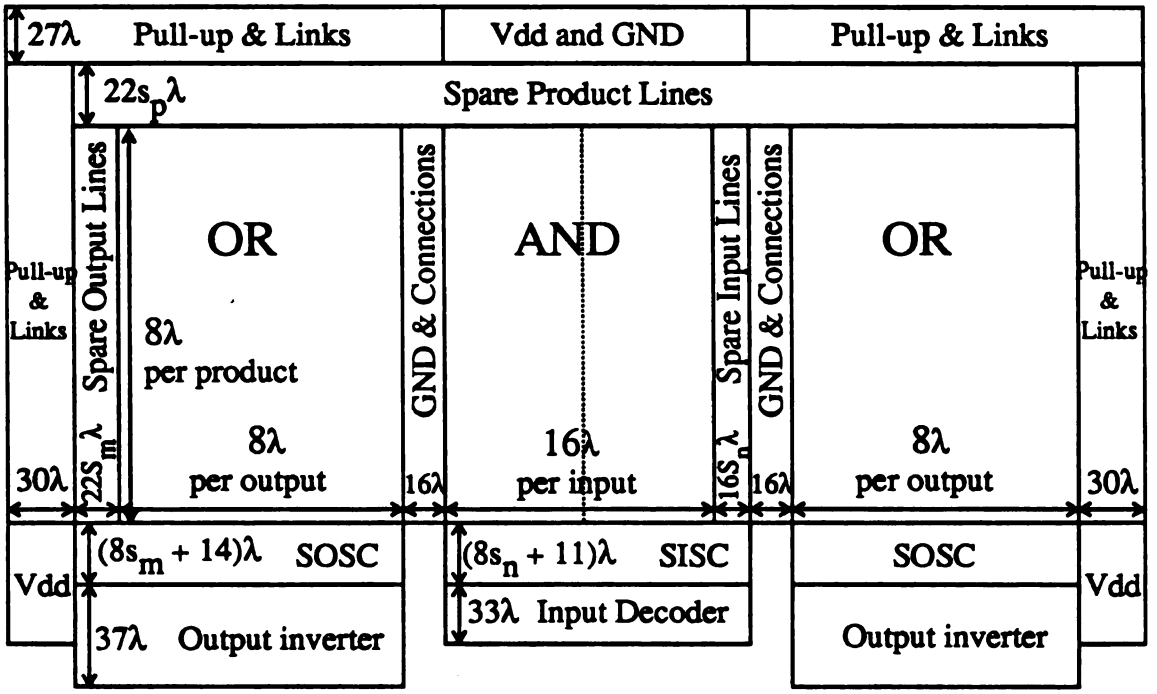


Figure 5.2 Floor Plan of the Fault-Repairable SRFPLA-O.

Table 5.2 Simulation Results (SRFPLA-O)

name	n	m	p	(1,2,1)-FTPLA area	(1,2,1)-SRFPLA-O p	(1,2,1)-SRFPLA-O %
alu1	12	8	19	143784	11	89.3
cps	24	102	162	2070304	151	96.9
in4	32	20	212	1694800	211	104.4
in5	24	14	62	486400	60	104.1
in6	33	23	54	580544	47	95.7
in7	26	10	54	444256	49	99.8
jbp	36	57	122	1452752	106	92.1
misg	56	23	69	993384	38	69.8
mish	94	34	82	1757440	48	70.6

5.3 Fault-Tolerant Design of MCFPLA

Figure 5.3 shows a schematic diagram of a PLA with multiple folding technique computed using PLEASURE and its personality matrix. The MCFPLA design allows the inputs to come into the AND plane from top, bottom, and left-hand side. Since an input line takes 16 λ in width, it is necessary to reserve two product lines (each has 8 λ in height) for a sided input line. Figure 5.4 illustrates a floor plan of a repairable and fault-diagnosable MCFPLA design. The area can be estimated by the following formula.

$$\begin{aligned}\text{Area} = {} & (16n_b + 16s_n + 30) \times (8s_m(8s_n + 22s_p + 8p + 8n_s + 43) + (8m_b + 22s_m \\ & +21) \times (8s_m + 22s_p + 8p + 8n_s + 81) + 16n_t(8s_n + 10) + 1440n_t + \\ & 8m_t \times (8s_m + 10) + 568m_t + 528n_s + (2n_b - 2n_t + p) \times 8 \times 140,\end{aligned}$$

where n_b : number of BOTTOM input lines, n_s : number of side input lines,
n_t : number of TOP input lines, s_n : number of spare input lines,
m_b : number of BOTTOM output lines, s_m: number of spare output lines,
m_t : number of TOP output lines,
p : number of product lines, s_p: number of spare product lines.

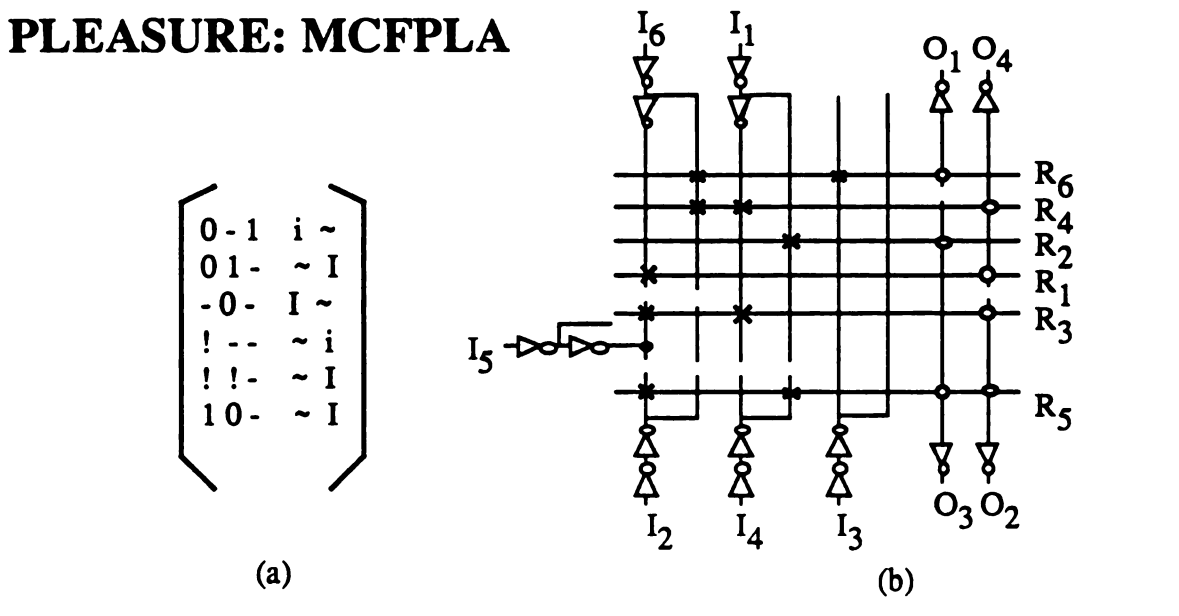


Figure 5.3 An Example of MCFPLA.

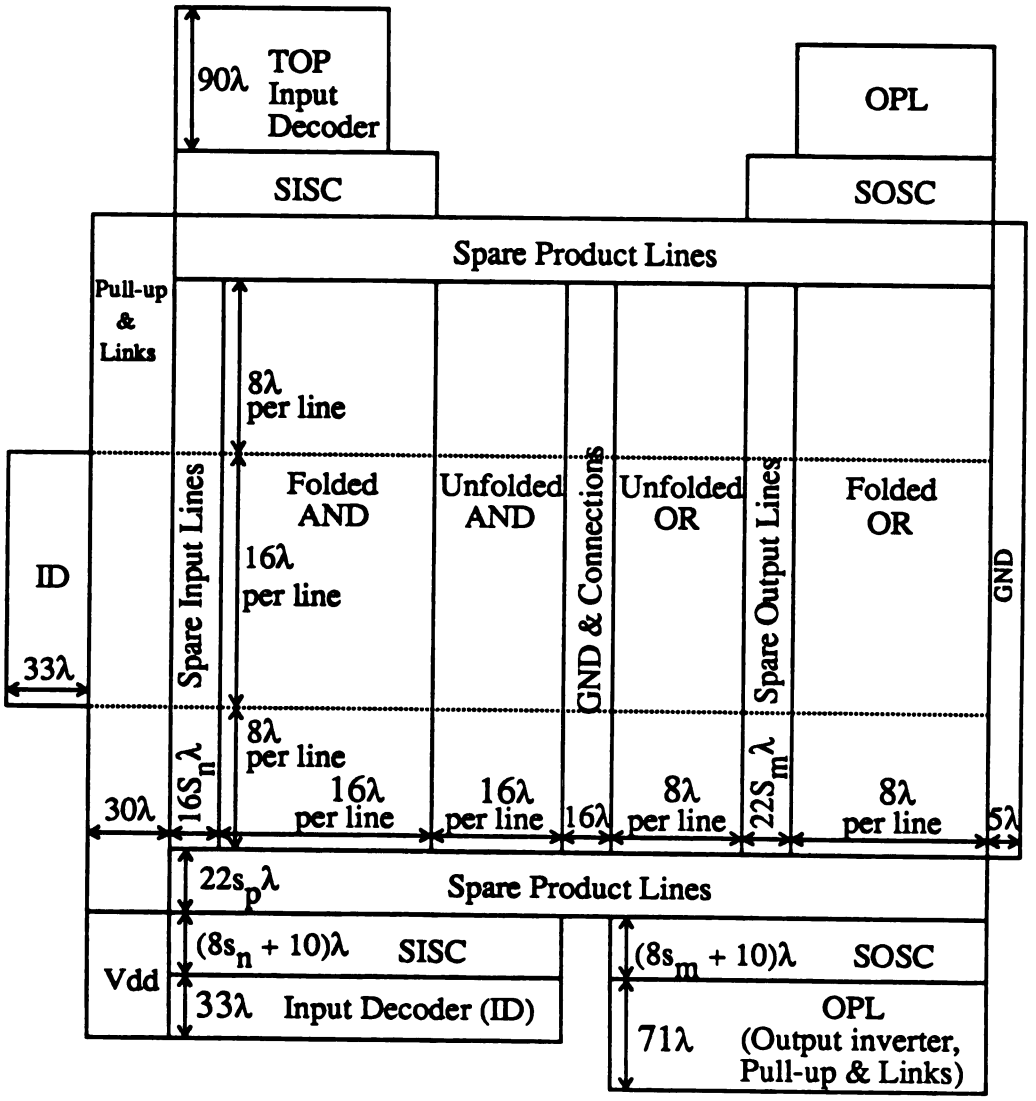


Figure 5.4 Floor Plan of the Fault-Repairable MCFPLA.

Table 5.3 summarizes the simulation results of the fault-tolerant design for various PLAs. The results show that the areas can be significantly reduced.

Table 5.3 Simulation Results (MCFPLA)

name	n	m	p	(1,2,1)-FTPLA area	(1,2,1)-MCFPLA			
					n_s	n_b	m_b	%
alu1	12	8	19	143784	2	7	5	73.1
cps	24	102	162	2070304	0	21	56	72.1
exep	28	62	109	1216408	0	24	31	74.3
in3	34	29	74	779440	8	19	19	70.6
in4	32	20	212	1694800	3	24	11	77.9
in5	24	14	62	486400	5	14	10	73.3
in6	33	23	54	580544	7	19	14	70.6
in7	26	10	54	444256	8	13	8	69.8
jbp	36	57	122	1452752	7	21	30	66.3
misg	56	23	69	993384	40	10	13	47.5
mish	94	34	82	1757440	67	21	27	57.5
ti	43	67	213	2737784	14	20	46	65.8
x2dn	82	47	104	2000128	51	23	24	54.1

5.4 Comparison

In order to compare the performance of various fault-tolerant folded PLA designs, Table 5.4 summarizes the results in Table 3.2, 5.1, 5.2, and 5.3.

Results show that the fault-tolerant designs of PLAs with column folding techniques are better than those PLAs with row folding techniques. Although the MCFPLA design is slightly better than the SCFPLA design in some cases, it should be mentioned that the fabrication process using the multiple folding technique is much more complicated than the simple folding technique. Therefore, this study suggests that fault-tolerant simple folded PLA design will provide "better" results as far as chip yield is concerned.

Table 5.4 Results Comparison

name	SRFPLA-A	SRFPLA-O	SCFPLA	MCFPLA
alu1	96.5	89.3	67.9	73.1
cps	103.9	96.9	76.2	72.1
exep	X	X	74.3	74.3
in3	X	X	73.8	70.6
in4	109.7	104.4	74.2	77.9
in5	109.9	104.1	74.9	73.3
in6	101.8	95.7	62.5	70.6
in7	108.3	99.8	69.4	69.8
jbp	99.7	92.1	70.6	66.3
misg	68.2	69.8	55.2	47.5
mish	65.2	70.6	52.7	57.5
ti	X	X	69.5	65.8
x2dn	71.4	X	57.6	54.1
x6dn	X	X	70.0	X

Remark: X indicates that the folded PLA cannot be obtained by PLEASURE.

Chapter 6

Conclusions

This chapter summarizes the major contribution of this study and outlines directions for future research.

6.1 Summary of Major Contribution

In order to ensure that large PLA chips are manufactured at a reasonable yield level, a fault-tolerant design, using folding techniques that allow full diagnosis and fault repair, is presented for yield enhancement. The major contribution in this study is that, taking the advantages of both folding techniques and fault-tolerant design technique, the proposed design not only achieves a full diagnosability of single and multiple stuck-at, bridging, and crosspoint faults, but also requires less chip area than the fault-tolerant design in [8] and thus increases the number of die that can be placed on a wafer. Moreover, the proposed design is fully testable after the chip is packaged.

The other contribution is the development of an automatic layout generator for fault-tolerant folded PLAs. Chapter 4 describes the procedure of developing an automatic layout generator. The layout generator ALGFPLA compiles the PLEASURE's data format into layout masks. The ALGFPLA has been implemented on SUN 3/160 under the UNIX operating system.

6.2 Directions for Future Research

According to the technological implementation and design style, numerous folding techniques have been proposed and implemented. The tradeoffs are area reduction and regularity, i.e., the area reduction is often paid for by the increase in irregularity. In practice, the irregularity structure generally results in increasing the defect density in the fabrication process. As a result, it is not always guaranteed that the more the area reduction, the higher the overall chip yield. A "good" folding algorithm may be good for reducing the array size, but it may be not suitable for the fault-tolerant design. The significance of the fault-tolerant design of folded PLAs should be in reducing the die size while still achieving the full diagnosability and repairability.

In this thesis, the PLEASURE program was used to demonstrate the effectiveness of the proposed fault-tolerant design. Our simulation results shown in Chapter 5 have found that the chip area reduction is not significant for other folding techniques. In practice, however, the deficiency is caused by folding algorithms that originally designate for area reduction, not for fault-tolerant design. Therefore, the development of a "good" algorithm that is suitable for fault-tolerant design is a very interesting subject for future research.

Spare line allocation and optimal redundancy for achieving the maximum yield are also very important in the fault-tolerant design, particularly for folded PLAs. A higher probability of repair can be achieved if a larger number of spares is added. However, since the added redundancy and the associated circuitry are also susceptible to defects, too much redundancy may have a "diminishing" effect on the chip. The optimal redundancy is highly dependent on both the failure rate of the fabricated chips and the folding algorithms. Consequently, achieving maximum yields using optimal redundancy is also an important research subject.

According to the technological implementation and design style, numerous folding techniques have been proposed and implemented. However, there is still no single folding

algorithm that is universally efficient for all PLAs. As a result, designers must face the problem of selecting an appropriate technique to match their goals and constraints. Therefore, it is necessary to develop a selection system that automatically produces the physical layout of the "optimal" fault-tolerant/fully testable PLA structure which meets the design requirements. We envision a system that allows the designers to: specify a set of logic functions to be realized by a PLA, optionally specify the desired yield level, select an appropriate structure that may be with or without folding depending upon the yield level desired, and to provide the physical layout of the resultant structure. A knowledge-based system can be employed to realize this vision. The knowledge base should contain the candidate folding algorithms and the corresponding layout generators.

APPENDICES

APPENDIX 1

Fault Diagnosis and repair Process

According to the fault models and the repair rules described in this thesis, this fault diagnosis and repair process can recover single and multiple stuck-at, bridging, and crosspoint faults. Keep in mind that both stuck-at and bridging faults must be repaired immediately after they have been identified.

Part 1: Detect Faults in the Augmented Circuits

The augmented circuits considered are ISR, PSR, and control circuits. Since the augmented circuits are non-redundant, they are fault detectable but not fault repairable.

To test the function of the shift registers, first, they are isolated from PLA by setting both signals R and W to logic 0. Then, by applying a sequential test pattern (0101...0101) from Sin to these registers, the outputs can be observed from Sout. Since the function of control circuit could be observed from some extra register cells, the control circuit is also fully testable. Therefore, after the augmented circuits have been tested to be fault-free, the following steps of fault diagnosis and repair process can be implemented.

Part 2: Locate and Repair Faults in the AND Plane

The faults in the AND plane include:

Type 1: Stuck-at fault at input bit column.

Type 2: Bridging fault between two adjacent input bit columns.

Type 3: Bridging fault between input bit column and product line.

Type 4: Stuck-at fault at product line.

Type 5: Bridging fault between two adjacent product lines.

Type 6: G-fault and S-fault.

Follow the steps below, the faults of Type 1 through Type 5 can be located and repaired, while the fault of Type 6 can be located.

Step 1: Set TOP input decoder and ISR to "read" mode, apply the input pattern (1, 1, ..., 1) and (0, 0, ..., 0) to the BOTTOM inputs. At the same time, set the PSR to "write" mode and write 0's to all the product lines. Thus the data on the input bit columns, read from TOP input decoder and ISR, are expected to be: ALL1 = (1, 0, 1, 0, ..., 1, 0) for the input pattern (1, 1, ..., 1), and ALL0 = (0, 1, 0, 1, ..., 0, 1) for the input pattern (0, 0, ..., 0).

The stuck-at faults at the input bit columns can be located by examining the zero bits in ALL, where ALL = ALL1 $\oplus$ ALL0. Once the stuck-at fault at the input bit column (Type 1) is located, it should be repaired immediately, i.e., switch it to a spare input line and connect this faulty input bit column to GND.

Property 1: Type 2 and Type 3 faults are equivalent to stuck-at faults.

Proof:

Case I. One of the bridged lines has stuck-at fault.

Since bridged lines should have the same logic, this bridging fault will force both bridged lines to have the same stuck-at fault.

Case II. None of the bridged lines contains stuck-at fault.

For Type 2, since the data expected on the adjacent input bit columns are always different, with the assumption of wired-AND logic, the bridged columns will be diagnosed as having stuck-at-0 fault.

For Type 3, since 0's have been written to the product lines, the bridged

input bit column will be diagnosed as having stuck-at-0 fault.

Therefore, after Step 1, all Type1 and Type 2 faults can be located and repaired while only part of Type 3 faults have been taken care of.

Step 2: Set TOP input decoder and ISR to "write" mode and assign 0's to all the input bit columns. Set PSR to "read" mode and read the value of each product line, which is expected to be 1. Therefore, a zero means a stuck-at-0 fault.

Notice that in the case of bridging fault between input bit column and product line, because in Step 1 the bridged input bit column had been connected to GND after it was diagnosed as having s-a-0 fault, the bridged product line will be diagnosed as having s-a-0 fault. (This completes the diagnosis and repair of Type 3 fault.)

Step 3: Set TOP input decoder and ISR to "write" mode and a walking 1 is passing through the input bit columns, i.e., only one bit column is enabled with value 1 and all other bit columns are disabled. Read the states of product lines from PSR.

Finally, we can construct matrix $N = [\ n_{ij}\ ]$ where n_{ij} is the inverse of the value on the i-th product line when the j-th bit column is enabled, i.e., $n_{ij} = 1\ (0)$ means there is (no) crosspoint between the j-th input bit column and the i-th product line.

Property 2: If the i-th row of matrix N contains all 0s, then the i-th product line is diagnosed as having s-a-1 fault. (This finishes the diagnosis and repair of Type 4 fault)

Proof:

In this case, either multiple crosspoint faults or s-a-1 fault could happen to this product line. For simplification of fault location process, treat it as s-a-1 fault.

Property 3: If the i-th and (i+1)-th rows of matrix N are identical, the i-th and (i+1)-th

product lines are diagnosed as having bridging fault (Type 5).

Proof:

In this case, either multiple crosspoint faults or bridging fault could happen to these two product lines. Again, for simplification of fault location process, treat this case as bridging fault between two adjacent product lines.

Now, let matrix A = [a_{ij}] be the personality matrix of AND array. After the faults of Type 1 through Type 5 have been repaired, the G-fault and S-fault (Type 6) can be located by examining the non zero bits in matrix C where C = [c_{ij}] = N $\oplus$ A = [$n_{ij} \oplus a_{ij}$]. These faults will be repaired until all the crosspoint faults of AND plane and OR plane have been located to optimally use the redundancy.

Part 3: Locate and Repair Faults in the OR Plane

The faults in the OR plane include:

Type 7 : Stuck-at fault at output line.

Type 8 : Bridging fault between output line and product line.

Type 9 : Bridging fault between two adjacent output lines.

Type 10: Bridging fault between two output lines which share one bit column.

Type 11: A-fault and D-fault.

Follow the steps below, the faults of Type 7 through Type 10 can be located and repaired, while Type 11 fault can be located.

Step 4: Set PSR to "write" mode and assign 0's to the product lines. Read the values on the output lines from both TOP and BOTTOM output inverters.

Notice that the output values should be 0's, therefore, an output value 1 indicates that the corresponding output line has stuck-at-1 fault (Type 7).

Property 4: The bridging fault between output and product lines is equivalent to stuck-at fault (Type 8).

Proof:

In the case when none of them contains stuck-at fault, the logic 0 in the bridged product line will force the bridged output line to have s-a-1 fault. On the other hand, since the faulty output line is repaired by disconnecting it from SOSC and connecting it to GND, the bridged product line is thus forced to have s-a-0 fault.

Step 5: Set PSR to "write" mode and a walking 1 is passing through the product lines, that is, enable only the i-th product line by assigning 1 to this product line and disable all other product lines by assigning 0's to these product lines. When PSR is set to "write" mode, ISR and TOP input decoder are set to "read" mode. To assign this walking 1 to the i-th product line successfully and avoid pulled-down by the input, apply a specific input pattern to the BOTTOM input decoders. This pattern can be found in the matrix N. When the walking 1 is passing through the product lines, read the states of the output lines from both BOTTOM and TOP output inverters.

A similar way as Step 3, two matrices $B = [\ b_{ij}\ ]$, $T = [\ t_{ij}\ ]$ can be constructed, for BOTTOM and TOP output inverters respectively where b_{ij} (t_{ij}) is the value at the j-th BOTTOM (TOP) output inverter when the i-th product line is enabled. Note that the j-th column of matrices B and T represent the same bit column in the folded OR plane.

Property 5: IF the j-th column of matrix B (T) contains all 0's, then the j-th bottom (top) side output line is diagnosed as having s-s-0 fault. (This completes the diagnosis and repair of Type 7 fault.)

Proof:

The j-th column of matrix B (T) containing all 0's implies that either the j-th bottom (top) output line has s-a-0 fault, or all crosspoints at this line are missed. For simplicity, diagnose it as s-a-0 fault.

Property 6: If the i-th rows of matrices B and T contain all 0's, then the i-th product line is diagnosed as having s-a-0 fault. (This completes the diagnosis and repair of Type 8 fault.)

Proof:

Again, either multiple crosspoint faults or s-a-0 fault could have happened. For simplicity, treat it as s-a-0 fault.

Property 7: If the j-th column and the (j+1)-th column of matrix B (T) are identical, then the j-th and the (j+1)-th bottom (top) side output lines are diagnosed as having bridging fault (Type 9).

Proof:

In this case, either multiple crosspoint faults or bridging fault could have happened. For simplicity, treat it as bridging fault.

Property 8: If the j-th columns of matrices B and T (when the j-th column of the matrix T is available) are identical, then the j-th BOTTOM and the j-th TOP output lines are diagnosed as having bridging fault (Type 10).

Proof:

Case I : The cut is complete.

Since the cut is complete, the upper part of the j-th column in matrix B and the lower part of the j-th column in matrix T should all be 0's. Now, if the j-th columns of matrices B and T are identical, then these two columns should contain all 0's. This case is impossible after Property 5.

Case II: The cut is incomplete.

Since the cut is incomplete, this is exactly Type 10 fault.

After the faults of Type 7 through Type 10 have been located and repaired, the remaining faults in the OR plane are Type 11 faults, i.e., A-faults and D-faults. Now, construct matrix $D = [\ d_{ij}\ ]$ where $d_{ij} = b_{ij} \oplus t_{ij}$ when t_{ij} is available, or $d_{ij} = b_{ij}$ when there is no t_{ij}. Let matrix $R = [\ r_{ij}\ ]$ be the personality matrix for OR array. A-fault and D-fault could be located by examining the non zero bits in matrix E where $e = [\ e_{ij}\ ] = D \oplus R = [\ d_{ij} \oplus r_{ij}\ ]$.

Part 4: Repair Crosspoint Faults

According to the repair rules, both G-fault and S-fault can be repaired either by spare input lines, and/or spare product lines. Similarly, both A-fault and D-fault can be repaired either by spare output lines, and/or spare product lines. By concatenating matrices C and E, a fault map is formed. The spare allocation algorithm developed in [31] can be used to efficiently repair these crosspoint faults.

Part 5: Locate and Repair Incomplete Cuts in AND Plane

After the faults of Type 1 through Type 11 have been located and repaired, in other words, there is no fault in the SCFPLA, the cut of the input bit columns is performed. However, this may also fail. Therefore, bridging fault between two input bit lines which share one bit column may happen. This fault is caused by improperly laser cutting. Although with the advent of today's laser programming techniques, the chance of having such fault is very slim, it should also been considered to achieve full diagnosability. In

fact, this fault can be easily detected by altering the input signals at the BOTTOM input decoders from 0 to 1 and observing the received signals from TOP input decoders (in "read" mode). The signal change should not affect the observed signal if the "cut" is performed properly; otherwise, this bridging fault is detected and the fault can be repaired by applying another cut. As a result, the proposed fault-tolerant SCFPLA design is a fully diagnosable and repairable design.

APPENDIX 2

ALGFPLA Program

```
#include "/usr/local/vlsi-ucb86/lib/mpack.h"
#include <stdio.h>
#include <ctype.h>
#defineTEMPLATE_DIR "~ding"
#define tempfile "ftscfpla-out"
#define even(x) (!(x & 1))
#define odd(x)  (x & 1)
#define maxdim 220
/*================================================================*/
/* subroutines declaration                                        */
/*================================================================*/
void init();
void load_in_template();
void read_in_data();
void sort_fold_unfold();
void restructure();
void shrink();
void paint_lower();
void paint_middle();
void paint_upper();
void save_result();
/*================================================================*/
/* tile names declaration                                         */
/*================================================================*/
TILE fpla;
TILE tand_null,tand_l,tand_r,tand_con,tand_noc;
TILE tor_ud,tor_null,tor_con,tor_noc,tor_d,tor_u,tor_fud,tor_fu,
     tor_fd,tor_fnull,tor_fnoc;
TILE tinput1,tinput2;
TILE toutput1_d,toutput1_s,toutput2_d,toutput2_s;
TILE tpdt_pullup1,tpdt_pullup2;
TILE tspare_input1,tspare_input2,tspare_input3,tspare_input4;
TILE tspare_output1,tspare_output2,tspare_output3,tspare_output4,
     tspare_output5,tspare_output6,tspare_output7;
TILE tspare_p_i1,tspare_p_i2,tspare_p_i3,tspare_p_i_pullup,
     tspare_p_con1,tspare_p_con2,tspare_p_con3,
     tspare_p_o1,tspare_p_o2,tspare_p_o3;
TILE tpsr,tisr,tisr_control,tpsr_control;
TILE tcon1,tcon2,tcon3,tcon4,tcon5,tcon6,tcon7,tcon8,tcon9r,tcon10,
     tcon11,tcon12,tor_gnd_con,tor_gnd_noc,tand_or_con_d;

char and[maxdim][maxdim],or[maxdim][maxdim];
int  np,ni,no,nif,nof;
RECTANGLE oldr,row_start,or_last_row;

main(argc,argv)
int argc;
char **argv;
{
```

```
        init(argc,argv);
        load_in_template();
        read_in_data();
        sort_fold_unfold();
        restructure();
        shrink();
        paint_lower();
        paint_middle();
        paint_upper();
        save_result();
}
/*================================================================*/
/* initial step                                                   */
/*================================================================*/
void init(argc,argv)
int argc;
char **argv;
{
        int i,j;
        TPinitialize(argc, argv, TEMPLATE_DIR);
        oldr=origin_rect;row_start=origin_rect;
        for (i=0;i<maxdim;i++)
        {       for (j=0;j<maxdim;j++)
                {       and[i][j]='-';
                        or[i][j] ='~';
                }
        }
}
/*================================================================*/
/* create a new tile and load in the template tiles               */
/*================================================================*/
void load_in_template()
{
        fpla = TPcreate_tile(tempfile);
        /*---------------------------------------*/
        /* 1. the core of the AND plane (AND_core)  */
                tand_null = TPname_to_tile("and_null");
                tand_l    = TPname_to_tile("and_l");
                tand_r    = TPname_to_tile("and_r");
                tand_con  = TPname_to_tile("and_con");
                tand_noc  = TPname_to_tile("and_noc");
        /*---------------------------------------*/
        /* 2. the core of the OR plane (OR_core     */
                tor_ud    = TPname_to_tile("or_ud");
                tor_null  = TPname_to_tile("or_null");
                tor_con   = TPname_to_tile("or_con");
                tor_noc   = TPname_to_tile("or_noc");
                tor_d     = TPname_to_tile("or_d");
                tor_u     = TPname_to_tile("or_u");
                tor_fud   = TPname_to_tile("or_fud");
                tor_fu    = TPname_to_tile("or_fu");
                tor_fd    = TPname_to_tile("or_fd");
                tor_fnull = TPname_to_tile("or_fnull");
                tor_fnoc  = TPname_to_tile("or_fnoc");
        /*---------------------------------------*/
        /* 3. the input                             */
                tinput1 = TPname_to_tile("input1");
                tinput2 = TPname_to_tile("input2");
        /*---------------------------------------*/
        /* 4. the output                            */
                toutput1_d = TPname_to_tile("output1_d");
                toutput1_s = TPname_to_tile("output1_s");
                toutput2_d = TPname_to_tile("output2_d");
                toutput2_s = TPname_to_tile("output2_s");
```

```
        /*-----------------------------------------*/
        /* 5. the pullup transitor for product line */
              tpdt_pullup1 = TPname_to_tile("pdt_pullup1");
              tpdt_pullup2 = TPname_to_tile("pdt_pullup2");
        /*-----------------------------------------*/
        /* 6. the spare input line                 */
              tspare_input1 = TPname_to_tile("spare_input1");
              tspare_input2 = TPname_to_tile("spare_input2");
              tspare_input3 = TPname_to_tile("spare_input3");
              tspare_input4 = TPname_to_tile("spare_input4");
        /*-----------------------------------------*/
        /* 7. the spare output line                */
              tspare_output1 = TPname_to_tile("spare_output1");
              tspare_output2 = TPname_to_tile("spare_output2");
              tspare_output3 = TPname_to_tile("spare_output3");
              tspare_output4 = TPname_to_tile("spare_output4");
              tspare_output5 = TPname_to_tile("spare_output5");
              tspare_output6 = TPname_to_tile("spare_output6");
              tspare_output7 = TPname_to_tile("spare_output7");
        /*-----------------------------------------*/
        /* 8. the spare product line               */
              tspare_p_i_pullup = TPname_to_tile("spare_p_i_pullup");
              tspare_p_i1       = TPname_to_tile("spare_p_i1");
              tspare_p_i2       = TPname_to_tile("spare_p_i2");
              tspare_p_i3       = TPname_to_tile("spare_p_i3");
              tspare_p_o1       = TPname_to_tile("spare_p_o1");
              tspare_p_o2       = TPname_to_tile("spare_p_o2");
              tspare_p_o3       = TPname_to_tile("spare_p_o3");
              tspare_p_con1     = TPname_to_tile("spare_p_con1");
              tspare_p_con2     = TPname_to_tile("spare_p_con2");
              tspare_p_con3     = TPname_to_tile("spare_p_con3");
        /*-----------------------------------------*/
        /* 9. the shift register                   */
              tisr         = TPname_to_tile("isr");
              tpsr         = TPname_to_tile("psr");
              tisr_control = TPname_to_tile("isr_control");
              tpsr_control = TPname_to_tile("psr_control");

        /*-----------------------------------------*/
        /* 10. the connection                      */
              tcon1         = TPname_to_tile("con1");
              tcon2         = TPname_to_tile("con2");
              tcon3         = TPname_to_tile("con3");
              tcon4         = TPname_to_tile("con4");
              tcon5         = TPname_to_tile("con5");
              tcon6         = TPname_to_tile("con6");
              tcon7         = TPname_to_tile("con7");
              tcon8         = TPname_to_tile("con8");
              tcon9r        = TPname_to_tile("con9r");
              tcon10        = TPname_to_tile("con10");
              tcon11        = TPname_to_tile("con11");
              tcon12        = TPname_to_tile("con12");
              tor_gnd_con   = TPname_to_tile("or_gnd_con");
              tor_gnd_noc   = TPname_to_tile("or_gnd_noc");
              tand_or_con_d = TPname_to_tile("and_or_con_d");
}
/*=========================================================*/
/* read the input data                                     */
/*=========================================================*/
void read_in_data()
{
      int x1,x2,y;
      char ch;
      y=1; ch=getchar();
```

```
        while (ch!=EOF)
        {       x1=0;
                while ((ch=='1')||(ch=='!')||(ch=='0')||(ch=='o')
                        ||(ch=='-')||(ch=='_'))
                {       and[x1++][y]=ch;
                        ch = getchar();
                }
                while (!((ch=='I')||(ch=='~')||(ch=='i')||(ch=='=')))
                        ch = getchar();
                x2=0;
                while ((ch=='I')||(ch=='~')||(ch=='i')||(ch=='='))
                {       or[x2++][y]=ch;
                        ch = getchar();
                }
                while (!((ch=='\n')||(ch==EOF))) ch = getchar();
                ch = getchar();
                y++;
        }
        np=y-1; ni=x1; no=x2;
}
/*============================================================*/
/* rearrange pla:|       and-plane      |        or-plane      | */
/               |  folded   unfolded  |   unfolded   folded | */
/*============================================================*/
void sort_fold_unfold()
{
        int i,valid,j,k;
        char tmp;
        i=0;valid=ni-1;
        while (i<=valid)
        {       j=1;
                while ((and[i][j]!='!')&&(and[i][j]!='o')&&(and[i][j]!='_')
                        &&(j<=np)) j++;
                if (j>np)
                {       for(k=1;k<=np;k++)
                        {   tmp=and[i][k];
                            and[i][k]=and[valid][k];
                            and[valid][k]=tmp;
                        }
                        valid--;
                }
                else i++;
        }
        nif=valid+1;
        i=0;valid=no-1;
        while (i<=valid)
        {       j=1;
                while ((or[i][j]!='i')&&(or[i][j]!='=')&&(j<=np)) j++;
                if (j>np)
                        i++;
                else
                {       for(k=1;k<=np;k++)
                        {       tmp=or[i][k];
                                or[i][k]=or[valid][k];
                                or[valid][k]=tmp;
                          }
                  valid--;
                }
        }
        nof=no-valid-1;
}
/*============================================================*/
/* restructure input bit columns                              */
/*============================================================*/
```

```
void restructure()
{
      int i,j;
      for (i=0;i<nif;i++)
      {     j=1;
            while ((and[i][j]!='!')&&(and[i][j]!='o')&&(and[i][j]!='_')
                  &&(j<=np)) j++;
            while (j>0)
            {     switch(and[i][j])
                  {     case '!':
                              and[i][j]='o';
                              break;
                        case 'o':
                              and[i][j]='!';
                              break;
                        case '1':
                              and[i][j]='0';
                              break;
                        case '0':
                              and[i][j]='1';
                              break;
                  }
                  j--;
            }
      }
}
/*================================================================*/
/* change the symbols below the cut and above the highest contact */
/* in the OR plane                                                */
/*================================================================*/
void shrink()
{
      int i,j,valid;
      for (i=no-nof;i<no;i++)
      {     j=1;
            while ((or[i][j]!='i')&&(or[i][j]!='=')&&(j<=np)) j++;
            valid=j;
            j++;
            while ((j<=np)&&(or[i][j]=='~'))
            {     or[i][j]='+';
                  j++;
            }
            if (or[i][valid]=='=')
            {     while ((valid>0)&&(or[i][valid]!='I'))
                  {     or[i][valid]='+';
                        valid--;
                  }
            }
      }
}
/*================================================================*/
/* paint the lower part of scfpla                                 */
/*================================================================*/
void paint_lower()
{
      int i;
      row_start.y_bot=row_start.y_bot+87;
      oldr=row_start;
      oldr=TPpaint_tile(tcon1,fpla,align(rLR(oldr),tLL(tcon1)));
      for(i=0;i<nif;i++) oldr=TPpaint_tile(tinput1,fpla,align(
                              rLR(oldr),tLL(tinput1)));
      oldr=TPpaint_tile(tspare_input1,fpla,align(
           rLR(oldr),tLL(tspare_input1)));
      for(i=nif;i<ni;i++)
```

```
        oldr=TPpaint_tile(tinput1,fpla,align(rLR(oldr),tLL(tinput1)));
    oldr.y_bot=oldr.y_bot-32;
    oldr=TPpaint_tile(tcon2,fpla,align(rLR(oldr),tLL(tcon2)));
    for(i=0;i<=no/2-1;i++)
        oldr=TPpaint_tile(toutput1_d,fpla,align(
            rLR(oldr),tLL(toutput1_d)));
    if (odd(no))
        oldr=TPpaint_tile(toutput1_s,fpla,align(
            rLR(oldr),tLL(toutput1_s)));
    oldr.y_bot  =0;
    oldr=TPpaint_tile(tcon3,fpla,align(rLR(oldr),tLL(tcon3)));
    oldr=TPpaint_tile(tspare_output3,fpla,align(
        rUL(oldr),tLL(tspare_output3)));
    oldr.y_bot  =0;
    oldr=TPpaint_tile(tpsr_control,fpla,align(
        rLR(oldr),tLL(tpsr_control)));
    row_start.y_bot=row_start.y_bot+51;
    row_start.x_right=0;

    oldr=row_start;
    oldr=TPpaint_tile(tspare_p_i_pullup,fpla,align(
        rLR(oldr),tLL(tspare_p_i_pullup)));
    for(i=0;i<nif;i++)
    {   oldr=TPpaint_tile(tspare_p_i2,fpla,align(
            rLR(oldr),tLL(tspare_p_i2)));
        oldr=TPpaint_tile(tspare_p_i1,fpla,align(
            rLR(oldr),tLL(tspare_p_i1)));
    }
    oldr=TPpaint_tile(tspare_p_i3,fpla,align(
        rLR(oldr),tLL(tspare_p_i3)));
    oldr=TPpaint_tile(tspare_input3,fpla,align(
        rLR(oldr),tLL(tspare_input3)));
    for(i=nif;i<ni;i++)
    {   oldr=TPpaint_tile(tspare_p_i2,fpla,align(
            rLR(oldr),tLL(tspare_p_i2)));
        oldr=TPpaint_tile(tspare_p_i1,fpla,align(
            rLR(oldr),tLL(tspare_p_i1)));
    }
    oldr=TPpaint_tile(tspare_p_i2,fpla,align(
        rLR(oldr),tLL(tspare_p_i2)));
    oldr=TPpaint_tile(tspare_p_con1,fpla,align(
        rLR(oldr),tLL(tspare_p_con1)));
    oldr.y_bot  =oldr.y_bot+2;
    for(i=0;i<no;i++)
        oldr=TPpaint_tile(tspare_p_o1,fpla,align(
            rLR(oldr),tLL(tspare_p_o1)));
    oldr=TPpaint_tile(tspare_output6,fpla,align(
        rLR(oldr),tLL(tspare_output6)));
    row_start.y_bot=row_start.y_bot+21;
}
/*============================================================*/
/* paint the middle part of pla                               */
/*============================================================*/
void paint_middle()
{
    int i,j;
    for (i=np;i>0;i--)
    {   oldr=row_start;
        if (odd(np-i+1))
            oldr=TPpaint_tile(tpdt_pullup1,fpla,align(
                rLR(oldr),tLL(tpdt_pullup1)));
        else    oldr=TPpaint_tile(tpdt_pullup2,fpla,align(
                rLR(oldr),tLL(tpdt_pullup2)));
        oldr=TPpaint_tile(tand_con,fpla,align(
```

```
     rLR(oldr),tLL(tand_con)));
for (j=0;j<nif;j++){
      switch (and[j][i])
      {     case ' ': case '-':
                oldr=TPpaint_tile(tand_null,fpla,align(
                     rLR(oldr),tLL(tand_null)));
                break;
            case '!': case '1':
                oldr=TPpaint_tile(tand_r,fpla,align(
                     rLR(oldr),tLL(tand_r)));
                break;
            case 'o': case '0':
                oldr=TPpaint_tile(tand_l,fpla,align(
                     rLR(oldr),tLL(tand_l)));
                break;
      }
      if ((and[j][i]=='1')||(and[j][i]=='!')
         ||(and[j+1][i]=='0')||(and[j+1][i]=='o'))
              oldr=TPpaint_tile(tand_con,fpla,align(
                   rLR(oldr),tLL(tand_con)));
      else    oldr=TPpaint_tile(tand_noc,fpla,align(
                   rLR(oldr),tLL(tand_noc)));
}
oldr=TPpaint_tile(tspare_input2,fpla,align(
     rLR(oldr),tLL(tspare_input2)));
oldr=TPpaint_tile(tand_con,fpla,align(
     rLR(oldr),tLL(tand_con)));
for (j=nif;j<ni;j++)
{     switch (and[j][i])
      {     case ' ': case '-':
                oldr=TPpaint_tile(tand_null,fpla,align(
                     rLR(oldr),tLL(tand_null)));
                break;
            case '!': case '1':
                oldr=TPpaint_tile(tand_r,fpla,align(
                     rLR(oldr),tLL(tand_r)));
                break;
            case 'o': case '0':
                oldr=TPpaint_tile(tand_l,fpla,align(
                     rLR(oldr),tLL(tand_l)));
                break;
      }
      if (j==ni-1)
            oldr=TPpaint_tile(tand_con,fpla,align(
                 rLR(oldr),tLL(tand_con)));
      else if ((and[j][i]=='1')||(and[j][i]=='!')
              ||(and[j+1][i]=='0')||(and[j+1][i]=='o'))
              oldr=TPpaint_tile(tand_con,fpla,align(
                   rLR(oldr),tLL(tand_con)));
      else    oldr=TPpaint_tile(tand_noc,fpla,align(
                   rLR(oldr),tLL(tand_noc)));
}


if (odd(np-i+1))
      oldr=TPpaint_tile(tand_or_con_d,fpla,align(
           rLR(oldr),tLL(tand_or_con_d)));
else
{     oldr.x_right=oldr.x_right+12;
      oldr.y_bot=oldr.y_bot-4;
}
if (i==1)
      or_last_row=oldr;
```

```
for (j=0;j<no;j++)
{     if (odd(np-i+1))
      {     if ((or[j][i]=='I')||(or[j][i]=='i')
               ||(or[j][i+1]=='I')||(or[j][i+1]=='i'))
                  oldr=TPpaint_tile(tor_con,fpla,align(
                       rLR(oldr),tLL(tor_con)));
            else if ((or[j][i]=='+')&&(or[j][i+1]=='+'))
                  oldr=TPpaint_tile(tor_fnoc,fpla,align(
                       rLR(oldr),tLL(tor_fnoc)));
            else oldr=TPpaint_tile(tor_noc,fpla,align(
                       rLR(oldr),tLL(tor_noc)));
      }
      else
      {     if (((or[j][i]=='I')&&(or[j][i+1]=='I'))
               ||((or[j][i]=='I')&&(or[j][i+1]=='i')))
                  oldr=TPpaint_tile(tor_ud,fpla,align(
                       rLR(oldr),tLL(tor_ud)));
            else if ((or[j][i]=='I')&&(or[j][i+1]=='~'))
                  oldr=TPpaint_tile(tor_u,fpla,align(
                       rLR(oldr),tLL(tor_u)));
            else if (((or[j][i]=='~')&&(or[j][i+1]=='I'))
                     ||((or[j][i]=='~')&&(or[j][i+1]=='i')))
                  oldr=TPpaint_tile(tor_d,fpla,align(
                       rLR(oldr),tLL(tor_d)));
            else if (((or[j][i]=='I')&&(or[j][i+1]=='+'))
                     ||((or[j][i]=='i')&&(or[j][i+1]=='+')))
                  oldr=TPpaint_tile(tor_fu,fpla,align(
                       rLR(oldr),tLL(tor_fu)));
            else if ((or[j][i]=='i')&&(or[j][i+1]=='I'))
                  oldr=TPpaint_tile(tor_fud,fpla,align(
                       rLR(oldr),tLL(tor_fud)));
            else if ((or[j][i]=='+')&&(or[j][i+1]=='I'))
                  oldr=TPpaint_tile(tor_fd,fpla,align(
                       rLR(oldr),tLL(tor_fd)));
            else if ((or[j][i]=='~')&&(or[j][i+1]=='~'))
                  oldr=TPpaint_tile(tor_null,fpla,align(
                       rLR(oldr),tLL(tor_null)));
            else oldr=TPpaint_tile(tor_fnull,fpla,align(
                       rLR(oldr),tLL(tor_fnull)));
      }
}
if (odd(np-i+1))
{     oldr=TPpaint_tile(tspare_output1,fpla,align(
            rLR(oldr),tLL(tspare_output1)));
      oldr=TPpaint_tile(tor_gnd_noc,fpla,align(
            rLR(oldr),tLL(tor_gnd_noc)));
      oldr.y_top=oldr.y_top-1;
      oldr=TPpaint_tile(tpsr,fpla,align(rUR(oldr),tLL(tpsr)));
}
else
{     oldr=TPpaint_tile(tspare_output2,fpla,align(
            rLR(oldr),tLL(tspare_output2)));
      oldr=TPpaint_tile(tor_gnd_con,fpla,align(
            rLR(oldr),tLL(tor_gnd_con)));
}

if ((i==1)&&(odd(np)))
{     or_last_row.y_bot=or_last_row.y_bot+4;
      oldr=or_last_row;
      for (j=0;j<no;j++)
      {     if ((or[j][i]=='I')||(or[j][i]=='i'))
                  oldr=TPpaint_tile(tor_d,fpla,align(
                       rLR(oldr),tLL(tor_d)));
            else oldr=TPpaint_tile(tor_null,fpla,align(
```

```
                                    rLR(oldr),tLL(tor_null)));
                    }
                      oldr=TPpaint_tile(tspare_output4,fpla,align(
                           rLR(oldr),tLL(tspare_output2)));
                      oldr=TPpaint_tile(tor_gnd_con,fpla,align(
                           rLR(oldr),tLL(tor_gnd_con)));
              }
              else if ((i==1)&&(even(np)))
              {     or_last_row.y_bot=or_last_row.y_bot+12;
                    oldr=or_last_row;
                    for (j=0;j<no;j++)
                    {     if ((or[j][i]=='I')||(or[j][i]=='i'))
                               oldr=TPpaint_tile(tor_con,fpla,align(
                                    rLR(oldr),tLL(tor_con)));
                          else oldr=TPpaint_tile(tor_noc,fpla,align(
                                    rLR(oldr),tLL(tor_noc)));
                    }
                      oldr=TPpaint_tile(tspare_output1,fpla,align(
                           rLR(oldr),tLL(tspare_output1)));
                      oldr=TPpaint_tile(tor_gnd_noc,fpla,align(
                           rLR(oldr),tLL(tor_gnd_noc)));
              }
              row_start.y_bot=row_start.y_bot+8;
        }
}
/*============================================================*/
/* paint the upper part of pla                                */
/*============================================================*/
void paint_upper()
{
        int i;
        oldr=row_start;
        oldr=TPpaint_tile(tspare_p_i_pullup,fpla,align(
             rLR(oldr),tLL(tspare_p_i_pullup)));
        for(i=0;i<nif;i++)
        {     oldr=TPpaint_tile(tspare_p_i2,fpla,align(
                    rLR(oldr),tLL(tspare_p_i2)));
              oldr=TPpaint_tile(tspare_p_i1,fpla,align(
                    rLR(oldr),tLL(tspare_p_i1)));
        }
        oldr=TPpaint_tile(tspare_p_i3,fpla,align(
             rLR(oldr),tLL(tspare_p_i3)));
        oldr=TPpaint_tile(tspare_input3,fpla,align(
             rLR(oldr),tLL(tspare_input3)));
        for(i=nif;i<ni;i++)
        {     oldr=TPpaint_tile(tspare_p_i2,fpla,align(
                    rLR(oldr),tLL(tspare_p_i2)));
              oldr=TPpaint_tile(tspare_p_i1,fpla,align(
                    rLR(oldr),tLL(tspare_p_i1)));
        }
        oldr=TPpaint_tile(tspare_p_i2,fpla,align(
             rLR(oldr),tLL(tspare_p_i2)));
        if (odd(np))
        {     oldr.y_bot  =oldr.y_bot+8;
              oldr=TPpaint_tile(tspare_p_con2,fpla,align(
                    rLR(oldr),tLL(tspare_p_con2)));
        }
        else
              oldr=TPpaint_tile(tspare_p_con3,fpla,align(
                    rLR(oldr),tLL(tspare_p_con3)));
        for(i=0;i<no;i++)
        {     if (odd(np))
                    oldr=TPpaint_tile(tspare_p_o2,fpla,align(
                         rLR(oldr),tLL(tspare_p_o2)));
```

```
            else  oldr=TPpaint_tile(tspare_p_o3,fpla,align(
                        rLR(oldr),tLL(tspare_p_o3)));
      }
      if (odd(np))
            oldr=TPpaint_tile(tspare_output5,fpla,align(
                  rLR(oldr),tLL(tspare_output5)));
      else  oldr=TPpaint_tile(tspare_output7,fpla,align(
                  rLR(oldr),tLL(tspare_output7)));
      oldr=TPpaint_tile(tcon7,fpla,align(rLR(oldr),tLL(tcon7)));
      if (odd(np))
            oldr=TPpaint_tile(tcon12,fpla,align(rUL(oldr),tLL(tcon12)));
      else
            oldr=TPpaint_tile(tcon10,fpla,align(rUL(oldr),tLL(tcon10)));
      row_start.y_bot=row_start.y_bot+21;
      oldr=row_start;
      oldr=TPpaint_tile(tcon5,fpla,alignrLR(oldr),tLL(tcon5)));
      for(i=0;i<nif;i++)
            oldr=TPpaint_tile(tinput2,fpla,align(rLR(oldr),tLL(tinput2)));
      oldr=TPpaint_tile(tspare_input4,fpla,align(
           rLR(oldr),tLL(tspare_input4)));
      oldr=TPpaint_tile(tcon6,fpla,align(rUL(oldr),tLL(tcon6)));
      oldr.y_bot  =oldr.y_bot-16;
      for(i=0;i<ni-nif;i++)
            oldr=TPpaint_tile(tisr,fpla,align(rLR(oldr),tLL(tisr)));
      oldr=TPpaint_tile(tcon4,fpla,align(rLR(oldr),tLL(tcon4)));
      oldr.y_bot  =oldr.y_bot+6;
      oldr=TPpaint_tile(tcon8,fpla,align(rLL(oldr),tLL(tcon8)));
      for(i=nof;i<no;i++)
            oldr=TPpaint_tile(tcon9r,fpla,align(rLR(oldr),tLL(tcon9r)));
      for(i=0;i<=nof/2-1;i++)
            oldr=TPpaint_tile(toutput2_d,fpla,align(
                  rLR(oldr),tLL(toutput2_d)));
      if (odd(nof))
            oldr=TPpaint_tile(toutput2_s,fpla,align(
                  rLR(oldr),tLL(toutput2_s)));
      oldr=TPpaint_tile(tcon11,fpla,align(rLR(oldr),tLL(tcon11)));
      oldr.y_bot  =oldr.y_bot-6;
      oldr=TPpaint_tile(tisr_control,fpla,align(
           rLR(oldr),tLL(tisr_control)));
}
/*=============================================================*/
/* save the result                                             */
/*=============================================================*/
void save_result()
{
      TPwrite_tile(fpla,"");
}
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Sear, D., "Is Bigger Always Better?" *ASIC Technology and News*, pp. 14-16, May 1989.

[2] Strojwas, A. J., "Design for Manufacturability and Yield," Proceeding of 26th ACM/IEEE Design Automation Conference, Anaheim, CA., pp. 710-713, June 1989.

[3] Moore, W. R., " A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield," IEEE Proceedings, Vol. 74, No. 5, pp. 684-698, May 1986.

[4] Wey, C. L., Vai, M. K., and F. Lombardi, "On the Design of a Redundant Programmable Logic Array (RPLA)," IEEE Journal of Solid-State Circuits, Vol. SC-22, No. 1, pp. 114-117, February 1987.

[5] Wey, C. L., "On Yield Considerations for the Design of Redundant Programmable Logic Arrays," IEEE Trans. on Computer-Aided Design, Vol. CAD-7, No. 4, pp. 528-535, April 1988.

[6] Wey, C. L., "Fault Location in Repairable Programmable Logic Arrays," Proceeding of IEEE International Test Conference, Washington DC, pp. 679-685, August 1989.

[7] Chang, T. Y., and C. L. Wey, "Design of Fault Diagnosable and Repairable PLA's," IEEE Journal of Solid-State Circuits, Vol. SC-24, No. 5, pp. 1451-1454, October 1989.

[8] Chang, T. Y., Ph.D. Dissertation, Department of Electrical Engineering, Michigan State University, October 1989.

[9] Patil, Suhas S., and Terry A. Welch, "A Programmable Logic Approach for VLSI," IEEE Trans. on Computers, Vol. c-28, No. 9, pp. 594-601, September 1979.

[10] Law, H.-F. S., and M. Shoji, "PLA Design for BELLMAC-32A microprocessor," Proceeding of International Conference Circuits and Computers, pp. 161-164, 1982.

[11] Fleisher, H., and L. I. Maissel, "An Introduction to Array Logic," IBM Journal Research and Development, Vol. 19, pp. 98-109, March 1975.

[12] Mead, C. A., and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. 1980.

[13] Hachtel, G. D., Newton, A. T., and A. L. Sangiovanni-Vincentelli, "An Algorithm for Optimal PLA Folding," IEEE Trans. on Computer-Aided Design, Vol. CAD-1, No. 2, pp. 63-76, April 1982.

[14] Egan, J. R., and C. L. Liu, "Optimal Bipartite Folding of PLA," Proceeding of 19th ACM/IEEE Design Automation Conference, pp. 141-146, June 1982.

[15] Luby, M., Vazirani, U., Vazirani, V., and A. Sangiovanni-Vincentelli, "Some Theoretical Results on the Optimal PLA Folding Problem," Proceeding of IEEE International Conference on Circuits and Computers, pp. 165-170, September

1982.

[16] Obrebska, M., Chuquillanqui, S., and H. Derantonian, "PLA and Custom Design," in *Design Methodologies*, edited by S. Goto, Elservier Science Publishers (North-Holland), pp. 83-122, 1986.

[17] Abraham, J. A., and W. K. Fuchs, "Fault and Error Models for VLSI," IEEE Proceedings, Vol. 74, No. 5, pp. 639-654, May 1986.

[18] Ostapko, D. L., and S. J. Hong, "Fault Analysis and Test Generation for Programmable Logic Arrays," IEEE Trans. on Computers, Vol. C-28, No. 9, pp. 617-626, September 1979.

[19] Pradhan, D. K., and K. Son, "The Effects of Untestable Faults in PLAs and a Design for Testability," Proceeding of IEEE International Test Conference, Philadelphia, PA, pp. 359-367, November 1980.

[20] Smith, J. E., "Detection of Faults in Programmable Logic Arrays," IEEE Trans. on Computers, Vol. C-28, No. 11, pp. 845-853, November 1979.

[21] Somenzi, F., and S. Gai, "Fault Detection in Programmable Logic Arrays," IEEE Proceedings, Vol. 74, No. 5, pp. 655-668, May 1986.

[22] Treuer, R., H. Fujiwara, and V. K. Agarwal, "Implementing a Built-In Self-Test PLA Design," IEEE Design and Test of Computers, pp. 37-48, April 1985.

[23] Hu, T. C., and Y. S. Kuo, "Optimum Reduction of Programmable Logic Arrays," Proceeding of 20th ACM/IEEE Design Automation Conference, pp. 553-558, June 1983.

[24] De Micheli, G., "Computer-Aided Synthesis of PLA-Based Systems," Memo. No. UCB/ERL M84/31, Electronics Research Lab., University of California,

Berkeley, April 1984.

[25] Makarenko, D., and J. Tartar, "An Efficient Algorithm for the Optimal Folding of PLAs," Proceeding of IEEE International Conference on Computer Design: VLSI in Computers, pp. 57-60, October 1985.

[26] Makkarenko, D., and J. Tartar, "A Statistical Analysis of PLA Folding," IEEE Trans. on Computer-Aided Design, Vol. CAD-5, No. 1, pp. 39-51, January 1986.

[27] De Micheli, G., and A. L. Sangiovanni-Vincentelli, "PLEASURE: A Computer Program for Simple/Multiple Constrained/Unconstrained Folding of Programmable Logic Arrays," Proceeding of 20th ACM/IEEE Design Automation Conference, pp. 530-537, June 1983.

[28] Brayton, R., Hachtel, G. D., McMullen, C. T., and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publisher, Hingham, MA., 1985.

[29] Khakbaz, J., "A Testable PLA Design with Low Overhead and High Fault Coverage," IEEE Trans. on Computers, Vol. C-33, No. 8, pp. 743-745, August, 1984.

[30] Mayo, R. N., and J. K. Ousterhout, "Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool," Proceeding of 20th ACM/IEEE Design Automation Conference, Miami Beach, FL., pp. 270-276, June 1983.

[31] Kuo, S. Y., and W. K. Fuchs, "Fault Diagnosis and Spare Allocation for Yield Enhancement in a Large Reconfigurable PLA," Proceeding of IEEE International Test Conference, Washington DC, pp. 944-951, September 1987.