UNIVERSITY OF CINCINNATI

Date: January 14, 2004

I, Vinod Narayanan,
hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Electrical Engineering

It is entitled:

"A Built-In Self Testing Method For Embedded Multi-Port Memory Arrays"

This work and its defense approved by:

Chair: Wen-Ben Jone
Fred Beyette
Harold Carter

A BUILT-IN SELF-TESTING METHOD FOR EMBEDDED MULTIPORT MEMORY ARRAYS

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering
January 2003

by

Vinod Narayanan

B.E.(E.E.), Madurai Kamraj University, India, June 2000

Thesis Advisor and Committee Chair: Dr. Wen Ben Jone

Dedication

To my dearest parents

Abstract

With recent advances in semiconductor technologies, the design and use of memories for realizing complex system-on-a-chip (SoC) is very widespread. The growing need for storage in computer, communication, and network appliances has motivated new advancements in faster and more efficient ways to test memories. Semiconductor memories are considered to represent 30-35% of the semiconductor market currently. The design flexibility and performance offered by component generators have made the life of the circuit designer easier, but have posted new challenges to test. Thus, as time progresses and the demand for data storage goes high, testing semiconductor memories efficiently becomes more and more important.

Efficient testing schemes for single-port memories have been readily available. Multiport memories are widely used in multi-processor systems, telecommunication ASICs etc. Research papers which define multi-port memory fault models and give march tests for the same are currently available. However, little work has been done to use the power of serial interfacing for testing multi-port memories. In this thesis, we discuss some basics about the architecture of two-port memories and fault models for the same. We will then use the serial testing mechanism to propose new algorithms which can prove effective to reduce the hardware cost considerably in a chip with many multi-port memories. Once we understand how serial interfacing helps testing two-port memories, one possible extension is to use serial interfacing for p-port memories (p>2). The proposed method based on the serial interfacing technique has the advantages of high fault coverage, low hardware overhead and tolerable test application time. Precise fault modeling and efficient test design are both important to keep the test cost and test time in acceptable limits.

Acknowledgements

I wish to express my sincere thanks to my advisor, Dr. Wen-Ben Jone, for the guidance and constructive criticism that he provided throughout my work. He was always ready to provide his guidance and help in solving problems, even during weekends. This work would never have taken this form without his encouragement and expertise. Thank you Dr. Jone, for all that you have done for us.

I would like to thank the members of my thesis committee, Dr. Carla Purdy and Dr. Harold Carter, for spending their invaluable time reviewing this work. Special thanks are also due to my close friends for the wonderful time I have had in their company. I would like to thank the Almighty for all He has done for me, and pray that He always guides me in the right direction. Last, but not the least, I would like to express my gratitude to my parents who have helped mould me into what I am today.

Contents

1 Introduction **1**

2 Background **4**
2.1 Introduction 4
2.2 Serial Interfacing Technique and SMarch Algorithm 5
2.3 Redundant Operations and RSMarch Algorithm 6
2.4 Selective Bit-Pattern Generation and Bi-Directional Serial Interface 9
2.5 Multiple-Port Memory Testing 10

3 Two-Port Fault Models **13**
3.1 Single-Cell Two-Port Faults 14
3.2 Double-Cell Two-Port Faults 14
3.2.1 2PF2aa 14
3.2.2 2PF2vv 15
3.2.3 2PF2av 16

4 Serial Interfacing Design for Multiport Memory Testing **17**
4.1 Conditions to Detect 2PF1 17
4.1.1 Conditions to Detect wDRDF & wDRDF 17
4.1.2 Conditions to Detect wRDF & wRDF 18
4.1.3 Conditions to Detect wRDF & wTF 18
4.1.4 Conditions to Detect All 2PF1 Faults 18
4.2 Parallel March Algorithm 2PF1 19
4.3 Serial March Algorithm 2PF1 20

5 Serial March Algorithm for Coupling Faults **26**
5.1 Conditions to Detect 2PF2aa 27
5.2 Parallel March Algorithm 2PF2aa 28
5.3 Conditions to Detect 2PF2vv 30
5.4 Parallel March Algorithm 2PF2vv 31
5.5 Unified Parallel March Algorithm 2PF2aa-vv 32
5.6 Serial March Algorithm 2PF2aa-vv 32
5.7 Detection of 2PF1 faults by SMarch 2PF2aa-vv Algorithm 33
5.8 Conditions to Detect 2PF2av 33
5.9 Parallel March Algorithm 2PF2av 35
5.10 Serial March Algorithm 2PF2av 35

6 Testing for Same-Word Faults **41**
6.1 Detection of Same-Word 2PF2aa Faults 42
6.1.1 Bi-Directional Serial March and Address-Sensitive Fault Detection 45
6.1.2 Fault Coverage Analysis for 2PF2aa Same-Word Faults 48
6.2 Detection of Same-Word 2PF2vv Faults 51
6.2.1 Fault Coverage Analysis for 2PF2vv Same-Word Faults 51
6.3 Consistency of the Final SMarch 2PF2aa-vv Algorithm 52

7 Redundant Operations and Fault Coverage **55**
7.1 BIST Architecture for Parallel Testing of Arrays with Multiple Ports 55
7.2 The RSMarch 2PF2aa-vv Algorithm 56
7.2.1 Detection of 2PF1 Faults 58
7.2.2 Detection of 2PFaa Faults 58
7.2.3 Detection of 2PFvv Faults 59
7.2.4 RSMarch 2PF2aa-vv Algorithm detects same-word faults 60
7.3 The RSMarch 2PF2av Algorithm 60
7.3.1 Detection of 2PF2av Faults 61

8 Conclusions and Future Work **62**

List of Figures

2.1 Serial interfacing technique . . . 6
2.2 SMarch algorithm . . . 6
2.3 RSMarch algorithm . . . 7
2.4 Testing of multiple buffers using redundant operations . . . 8
2.5 Selective bit-pattern generation . . . 10
2.6 Complimentary bit-pattern generation . . . 11
2.7 Bi-directional serial interface architecture . . . 12

3.1 Taxonomy of 2PFs . . . 14

4.1 2PF1 serial interfacing memory state after $(w0 : n, rx : n)^3$ . . . 21
4.2 2PF1 serial interfacing memory state (w1 : r0) - 1st time . . . 23
4.3 2PF1 serial interfacing memory state (r0 : w1) - 1st time . . . 23
4.4 2PF1 serial interfacing memory state (w1 : r0) - 2nd time . . . 24
4.5 2PF1 serial interfacing memory state (r0 : w1) - 2nd time . . . 24
4.6 2PF1 serial interfacing memory state (w1 : r0) - 3rd time . . . 25
4.7 2PF1 serial interfacing memory state (r0 : w1) - 3rd time . . . 25

5.1 Classification of 2PF2 . . . 27
5.2 2PF2aa aggressor-victim exhibit. . . . 28
5.3 2PF2aa parallel interfacing memory state after w0:n. . . . 29
5.4 2PF2aa parallel interfacing memory state after w1:r0. . . . 29
5.5 2PF2aa parallel interfacing memory state after w0:r1. . . . 30
5.6 2PF2aa illustration (i) - address(victim) < address(aggressor) . . . 30
5.7 2PF2aa illustration (ii) - address(victim) < address(aggressor) . . . 31
5.8 2PF2vv aggressor-victim exhibit. . . . 31
5.9 2PF2av serial interfacing architecture . . . 36
5.10 2PF2av march order. . . . 37
5.11 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 1st time . . . 38
5.12 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 2nd time . . . 39
5.13 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 3rd time . . . 39

6.1 2PF2aa-vv serial interfacing memory state after $(w0 : rx, rx, w1)^3$. . . . 43
6.2 2PF2aa-vv serial interfacing memory state after $(r0 : r0)$. . . . 44
6.3 2PF2aa-vv serial interfacing memory state (r0 : w1) - 1st time (i) . . . 45
6.4 2PF2aa-vv serial interfacing memory state (w1 : r0) - 1st time (i) . . . 46
6.5 2PF2aa-vv serial interfacing memory state (r0 : w1) - 2nd time (i) . . . 47
6.6 2PF2aa-vv serial interfacing memory state after $(r0 : r0)$ with address(agressor)> address(victim). . . . 48

6.7 2PF2aa-vv serial interfacing memory state (r0 : w1) - 1st time (ii) 49
6.8 2PF2aa-vv serial interfacing memory state (w1 : r0) - 2nd time (ii) 50
6.9 2PF2aa-vv Serial Interfacing memory state (r0 : w1) (iii) 53
6.10 2PF2aa-vv serial interfacing memory state (r1 : w0) (iii) 53

7.1 Bist architecture for multiple RAMs 56

List of Tables

1.1 Percentage of logic forecast in SoC design 1

3.1 List of 2PFs; x=0,1 and d=don't care 15

4.1 List of 2PF1s 17
4.2 Parallel march algorithm 2PF1 satisfies condition 2PF1 19

5.1 List of 2PF2s 26
5.2 List of 2PF2aas 27
5.3 List of 2PF2vvs 30
5.4 List of 2PF2aa-vvs 33
5.5 2PF1 detection by SMarch 2PF2aa-vv 34
5.6 List of 2PF2avs - fault detection summary1 40

6.1 List of 2PF2 same-word faults. 42
6.2 Fault detection summary(i). 51
6.3 Fault detection summary(ii). 52

Chapter 1

Introduction

With the improvement of VLSI technologies, many components can now be fabricated into a single chip. This is also referred to as system-on-chip design or SoC design. A system-on-chip (SoC) integrated circuit generally contains processors, memories, and peripheral interface devices on a single chip. For example, totally 32 millions of transistors are fabricated on the PNX8525 chip that includes two programmable processor cores and 237 embedded memory arrays, some are large while some are small [32]. This induces various testing problems due to the inaccessibility of components, especially in memory modules, as all transistors are so densely packed that they are very vulnerable to fabrication defects. While SoC designs have the advantages of higher performance, lower power consumption, and smaller area when compared with system-on-board designs, test development is now identified as a major bottleneck. An important part of a SoC is memory arrays which are used in the form of small arrays or buffers (embedded memory arrays) between subsystems with different data consumption rates. In the future, the percentage of memory logic in SoC's is estimated to be as high as 94%. From Table 1.1, it is estimated that the percentage of memory logic in SoC will increase from 52% currently to as high as 94% by the year 2014 [9]. Further, we do not like to throw the chip away if there exist faulty memory cells. Hence, the need of testing the memories on chip is critical so that further repair can be performed.

Yr	Node	% Area New Logic	% Area Reused Logic	% Area Memory
1999	180nm	64	16	20
2002	130nm	32	16	52
2005	100nm	16	13	71
2008	70nm	8	9	83
2011	50nm	4	6	90
2014	35nm	2	4	**94**

Table 1.1: Percentage of logic forecast in SoC design

Memory is an integral part of parallel computers of today. Efficient parallel computers are hard to manufacture because of difficult technical problems. The most important is: how to arrange efficient communication between processors? Theoretically, this problem can be best solved by using shared main memory between all processors [21][13][5]. The most obvious way to implement shared memories is to use multiport RAM as building blocks of the true shared memory. Multiport RAM is a memory that has multiple ports to access memory cells simultaneously and independently of each other. In parallel computers, one processor is usually connected to one port.

From the processor point of view, there exists a uniform, shared memory connected to it. Other processors do not affect the operation of the processor.

Multiport memories are also used in a wide variety of applications, including wireless, wireline and storage area networking segments. In wireless infrastructure applications, multiport memories are used to store and manipulate packet data between FPGA, ASIC, and/or DSP devices operating in different clock domains on the baseband processing card. Wide area and storage networks employing high-performance communication protocols (such as OC-48, Gigabit Ethernet, or Fibre Channel) also use high-density multiported memories to buffer data packets, between the backplane interconnect and the data port, to avoid loss of data and maintain efficient flow control. The size of buffers is driven by the difference between input and output clocking speeds, the buffering time required by the system, and the bus width. These requirements dictate the need for a high-density multi-port memory in communication systems as system speeds continue to increase.

Maxwell Technologies' 7025E dual-port RAM, high-speed CMOS microcircuit features a greater than 100 krad (Si) total dose tolerance, depending upon space mission. The 7025E RAM is designed to be used as a stand-alone 128k-bit dual-port RAM or as a combined MASTER/SLAVE dual-port RAM for 32-bit or more word systems. This design results in full-speed, error-free operations without the need for additional discrete logic. The 7025E RAM provides two independent ports with separate control, address, and I/O pins that permit independent, asynchronous access for reads or writes to any location in memory.

Analog Devices Super Harvard Architecture Computer (SHARC)(ADSP-2106x) 32-bit floating-point DSP targets communications, speech, sound, graphics, and imaging applications. The ADSP-2106x CPU actually executes using only on-chip memory for a range of application codes. SHARC contains a 512-kbyte on-chip memory organized into two banks of dual-port RAM that can be combined into 16-, 32-, or 40-bit data and 48-bit instructions. This RAM holds large chunks of critical code and delivers sustained single-cycle memory accesses.

Cyprus the CY7C0853V dual-port RAM has been unveiled for wireless base station and storage-area network designs. This device provides 9-Mb of synchronous, pipelined memory capable of buffering large packets of data between two independent clock domains. The memory device delivers a 256K x 36-b configuration and provides up to 9.6 Gbps of bandwidth. As a true dual-port memory, the FLEx72 DP RAM provides simultaneous access to any location in the memory - either port can write and read data into and out of any memory location at the same time. This feature, combined with the ability to run both ports in two independent clock domains, enables the buffering of large packets of data between two processing elements in a system. It also eliminates bus contention issues by enabling system architects to create a distributed bus architecture. By providing the market's first product of its kind, Cypress has set the industry pinout for 18 Mbit x72 dual-port memories. Dual-port RAM is also finding a home in packet storage in SAN (Fiberchannel) hardware and layer-3 switches to accommodate larger delays found in highly-distributed networks.

Motorola DSP561xx family's 16-bit fixed-point DSP has a on-chip program RAM and a dual-ported data RAM; each has its own address and data bus. The dual-ported data RAM allows the address generator to deliver two addresses per pipeline cycle, yielding two data reads or one read and one write.

Thus, multiport RAMs are very important in parallel computers, SoCs, telecommunication, and various application-specific digital circuits [8]. In this thesis, we propose a mechanism for the chip to detect multiport faults by itself. This means we include built-in self-test (BIST) circuitry

on the chip which is able to detect the faulty cells in multiple-port embedded memory arrays. Based on the advanced fault models proposed by [6], the idea of serial interfacing technique which has been widely used in embedded memory array testing [4][19] is extended to test multiple-port defects. As in [12], the serial interface test method has the benefits of low hardware overhead, high fault coverage, and tolerable test application time. Due to the use of serial scan, the routing area for multiple-port test data can be highly reduced. Totally, only two data lines (one for each port) are required to deliver test patterns to each multiple-port memory array. Each array requires only two wires (one for each port) to route the test responses to the (common, global) multiple input signature register (MISR). Without the use of serial scan testing, the wires used for test pattern and response delivering will cause a severe routing problem, especially the memory arrays dealt with are muitiported. Another important feature of the proposed method that only one BIST controller is required. This also reduces a significant amount of hardware overhead for testing. High fault coverage can be achieved since the most advanced fault modeling is used in this research. Test patterns are designed to target faults which are related to multiple ports. Note that it is impossible to detect multiport faults using a single-port fault model. Finally, test application time is tolerable by the proposed serial scan test method, since all arrays are tested in parallel (as long as the power consumption is allowed). Though the simulataneous testing of all arrays will incur the problem of redundant (i.e, extra) test patterns, our results show that they do not cause any problem in fault coverage. Thus, tolerance of redundant test patterns benefits both hardware overhead (to simplify the BIST controller design), and test application time (virtually all arrays can be tested in parallel).

This thesis is organized as follows:

Chapter 2 describes the necessary background, the serial interfacing architecture, and parallel multiport testing.

Chapter 3 mentions the two-port fault models. A basic classification of two-port faults is performed on the basis of faults involved in a single or double cell.

Chapter 4 provides a detailed design of using the serial interfacing technique to test single-cell two-port faults in embedded memories. The conditions to detect the faults and associated algorithms (both parallel and serial) are discussed. The serial algorithm proposed is named SMarch 2PF1.

Chapter 5 discusses the application of the serial interfacing technique to detect coupling faults in multiport memories. Two algorithms namely SMarch 2PF2aa-vv and SMarch 2PF2av are proposed to detect different types of coupling faults.

Chapter 6 deals with multiport faults that can occur in the same word. Note that Chapters 4 and 5 only deal with multiport faults in different words. Thus, in Chapter 6, we extend and modify the SMarch 2PF2aa-vv to detect faults in the same word also.

Chapter 7 is dedicated to the scenario that the algorithms proposed so far are not affected by redundant operations. In SOCs with numerous small buffers, a common BIST circuit that uses these algorithms can be designed. Thus, it is vital to prove that excessive march operations to some smaller sized buffers (word length $n' <$ max word length n or bit length $c' <$ max bit length c) do not affect the validity of testing all faults in all buffers.

Chapter 8 concludes the thesis with suggested future work in meeting the challenge of efficiently detecting, diagnosing, and repairing multiple-port embedded memories distributed spatially in different areas of the SoC.

Chapter 2

Background

In this chapter, we review an efficient method to test spatially distributed small memory arrays. We introduce the concepts of serial interfacing technique as applied for the SMarch and RSMarch algorithms. We then describe the problem of selective bit-pattern generation associated with serial interfacing technique and a solution in the form of a bi-directional serial interfacing technique. The major portion of this chapter references the work in [10] as background material for this thesis.

2.1 Introduction

With the improvement in VLSI technologies, many components can be fabricated into a single chip. This induces various testing problems due to the inaccessibility of components. In today's era, we find more and more importance being given to systems-on-chip (SoC). We have already considered the importance of testing and diagnosing faults in memory buffers, present in these SoC's in Chapter 1. Further, the possibility of having a large number of small memory arrays spatially distributed on a chip/board is more common than having large memory arrays concentrated in a particular area of the chip. Hence, this thesis concentrates on fault testing of small memory arrays.

As we are dealing with small memory arrays, let us consider some of the problems in testing these buffers as follows:

Small Size of Memory Arrays The small size of memory arrays does not justify the addition of a built-in self testing (BIST), diagnosis (BISD), and repairing (BISR) controller to each buffer. This too will induce much hardware overhead. Note that this solution can be ideal in case of large memory arrays; but for small memory arrays, it just isn't feasible.

Distributed Nature of Memory Arrays The trend nowadays is to have many small memory arrays spatially distributed across different regions of the chip. Hence, because of the previous point, if we settle for a single controller, then the routing overhead, in terms of wires for sending the test patterns to test these spatially distributed memory arrays and wires for getting the output responses from each of these arrays back to the single controller will be very large. Hence, the distributed nature of these buffers poses a routing overhead constraint.

Embedded Memory Arrays Most buffers are deeply embedded inside the chip and are thus hard to test using external testers [10].

In some applications, the buffer widths are very large. If the test patterns are routed simultaneously from the BIST controller to memory modules, then the routing area overhead might not be tolerable. Thus, the *serial interfacing technique* with a test algorithm called SMarch was proposed as an access interface between the BIST controller and memory buffers [4]. The routing area overhead can be dramatically reduced. However, the deficiency of the serial interfacing technique is that testing might need to be performed sequentially (module by module) in some cases. To further alleviate the difficulties, a test method called RSMarch has been proposed in [12] with redundant test operations. The method has the advantages of high fault coverage, low hardware overhead, and low test application time. However, it is still uneconomical if a chip must be abandoned when only a small number of defects occur. Thus, it is beneficial if the defects in memory buffers can be diagnosed and repaired. The concepts of the serial interfacing technique, the SMarch algorithm and the RSMarch algorithm are covered in latter sections.

2.2 Serial Interfacing Technique and SMarch Algorithm

The serial interfacing technique shown in Figure 2.1 was proposed to test embedded memory arrays using a BIST technique in which only two serial scan data signals are required for each memory module to access the memory contents [4]. This saves considerable routing area. The serial interfacing technique is extremely useful for memory testing when many small memory buffers are spatially distributed on the entire chip. The basic idea of the serial interfacing technique is to synthesize the I/O port of each buffer as a scan chain wherein the test patterns are provided and memory contents are read. By fixing the addressing lines, successive read/write operations on a given address and serial scan in of the test patterns can result in scanning out the contents of the memory word. Based on this test architecture, the SMarch algorithm is used to generate test patterns and to evaluate test responses for each memory array [4]. The SMarch algorithm is represented in Figure 2.2. SMarch is a march-like test similar to algorithm C and has six march elements. The notation 'c' represents the number of bits for each word, and $\Uparrow$ ($\Downarrow$) represents test marching from low-order (high-order) words to high-order (low-order) words. For example , march element 3 of Figure 2.2 contains two groups of input-output patterns, $(\mathrm{r1w0})^c$ and $(\mathrm{r0w0})^c$, for each memory address. By setting *Si* to be '0' during the entire march element 3, the first 'c' operations would continuously read '1' from *So* and write a '0' to *Si* as shown in Figure 2.2. This immediately fills the addressed word with '0'. Then, the next 'c' operations read '0' from *So* and write '0' to *Si*. SMarch changes the contents of memory one bit at a time by a pair of read and write operations [11]. Also, note that the SMarch algorithm observes one bit at a time. It is not difficult to verify that the test patterns supported by SMarch are able to detect stuck-at, transition, coupling, and sequential faults which occur in the memory array [4]. Since the addressing faults have been successfully mapped to memory cell array faults, they are also detected by the same test patterns [26].

In order to support the SMarch algorithm, additional multiplexers are inserted to the I/O port of the memory cell array for the purpose of serial scan as shown in Figure 2.1. The added multiplexers and the memory I/O port thus from a serial scan chain, allows the BIST circuit to provide one bit of scan-in data (Si) and to observe one bit of scan-out data (So) for each memory read/write operation during testing. More details of the serial interfacing technique can be found in [4]. The beauty of the serial interfacing technique lies in its small hardware overhead. It is amazing to note that only two lines are required in test data application and observation (instead of '2c' where 'c' is the word width) for each memory buffer. This reduces the area required for routing, especially if we have a worst case of a large number of memory buffers spatially distributed

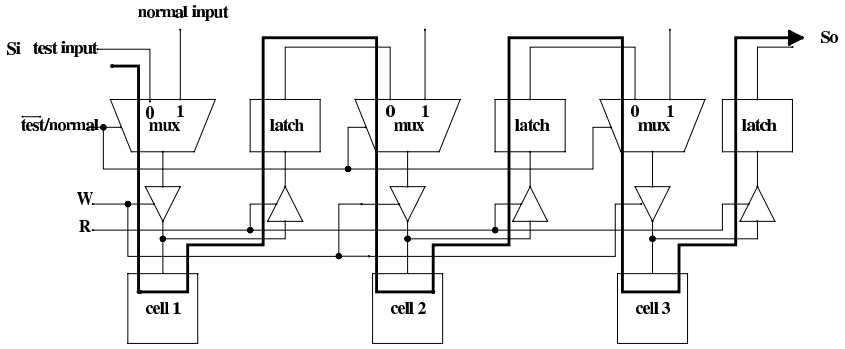

Figure 2.1: Serial interfacing technique

over the entire chip. Further, the input data line can be shared by all buffers. Therefore, the number of interconnections between the BIST controller and each memory cell array/buffer is totally independent of the array size. This special feature of the serial interfacing technique results in the greatest benefit for a circuit containing many buffers (especially when they are widely spread on the chip) with different sizes. This will be considered in the next section. Although the test time also increases due to the (serial) shift operation, this does not cause any trouble for our application since most memory cell arrays do not contain a tremendous number of bits.

$$
\begin{aligned}
&\text{M1} \Uparrow (rxw0)^c(r0w0)^c \\
&\text{M2} \Uparrow (r0w1)^c(r1w1)^c \\
&\text{M3} \Uparrow (r1w0)^c(r0w0)^c \\
&\text{M4} \Uparrow (r0w1)^c(r1w1)^c \\
&\text{M5} \Uparrow (r1w0)^c(r0w0)^c \\
&\text{M6} \Uparrow (r0w1)^c(r1w1)^c
\end{aligned}
$$

Figure 2.2: SMarch algorithm

The beauty of the serial interfacing technique therefore lies in its small hardware overhead, as specially, only two lines are required for test data application and test output observation for each memory buffer.

Next, we consider as to how we can use the serial interfacing technique for parallel testing of memory buffers. We introduce the concept of RSMarch algorithm which will be used for parallel testing of different sized memory buffers.

2.3 Redundant Operations and RSMarch Algorithm

In the previous section, we have seen several advantages of using the serial interfacing technique for testing spatially distributed memory arrays. Considerable routing area can be saved by using this technique. However, testing needs to be performed sequentially when using this technique with the SMarch algorithm. This increases the test time considerable, especially if there are a large number of such buffers. In this section, we consider the concept of introducing redundant operations in SMarch algorithm in order to enable parallel testing of different sized memory arrays.

To test a set of spatially distributed memory modules with various sizes, an efficient parallel BIST method called RSMarch was proposed in [12]. All memory modules receive a single test data line, controlled by a single BIST controller, and are tested simultaneously. This BIST method, taking advantages of the serial interfacing technique [4], has the benefits of high fault coverage, low hardware overhead, and low test application time. It uses the RSMarch algorithm in order to test the memory buffers in parallel. The RSMarch test algorithm, shown in Figure 2.3 includes six march elements (M1- M6), and each march element is divided into two portions: the *horizontal* operations and the *vertical* operations.

$$
\begin{aligned}
\mathbf{M1}: & \Uparrow_0^{n'-1}(\mathbf{rxw0})^{c'}(\mathbf{r0w0})^{c-c'}(\mathbf{r0w0})^{c'}(\mathbf{r0w0})^{c-c'} \\
& (\Uparrow_0^{n'-1}(\mathbf{r0w0})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Uparrow_0^{(n \bmod n')-1}(\mathbf{r0w0})^{2c} \\
\mathbf{M2}: & \Uparrow_0^{n'-1}(\mathbf{r0w1})^{c'}(\mathbf{r1w1})^{c-c'}(\mathbf{r1w1})^{c'}(\mathbf{r1w1})^{c-c'} \\
& (\Uparrow_0^{n'-1}(\mathbf{r1w1})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Uparrow_0^{(n \bmod n')-1}(\mathbf{r1w1})^{2c} \\
\mathbf{M3}: & \Uparrow_0^{n'-1}(\mathbf{r1w0})^{c'}(\mathbf{r0w0})^{c-c'}(\mathbf{r0w0})^{c'}(\mathbf{r0w0})^{c-c'} \\
& (\Uparrow_0^{n'-1}(\mathbf{r0w0})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Uparrow_0^{(n \bmod n')-1}(\mathbf{r0w0})^{2c} \\
\mathbf{M4}: & \Downarrow_0^{n'-1}(\mathbf{r0w1})^{c'}(\mathbf{r1w1})^{c-c'}(\mathbf{r1w1})^{c'}(\mathbf{r1w1})^{c-c'} \\
& (\Downarrow_0^{n'-1}(\mathbf{r1w1})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Downarrow_0^{(n \bmod n')-1}(\mathbf{r1w1})^{2c} \\
\mathbf{M5}: & \Downarrow_0^{n'-1}(\mathbf{r1w0})^{c'}(\mathbf{r0w0})^{c-c'}(\mathbf{r0w0})^{c'}(\mathbf{r0w0})^{c-c'} \\
& (\Downarrow_0^{n'-1}(\mathbf{r0w0})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Downarrow_0^{(n \bmod n')-1}(\mathbf{r0w0})^{2c} \\
\mathbf{M6}: & \Downarrow_0^{n'-1}(\mathbf{r0w1})^{c'}(\mathbf{r1w1})^{c-c'}(\mathbf{r1w1})^{c'}(\mathbf{r1w1})^{c-c'} \\
& (\Downarrow_0^{n'-1}(\mathbf{r1w1})^{2c})^{\lfloor n/n' \rfloor - 1} \\
& \Downarrow_0^{(n \bmod n')-1}(\mathbf{r1w1})^{2c}
\end{aligned}
$$

Figure 2.3: RSMarch algorithm

Let us compare the RSMarch algorithm with the SMarch algorithm introduced in the previous section. The RSMarch has additional redundant operations as compared to the SMarch algorithm. Consider march element M2 as an example. As shown in Figure 2.3, the first row of M2 contains a series of (r0w1) and (r1w1) test operations used to test the entire memory array. The

notation of (rawb) means that data read from the scan chain (So) has logic value 'a', while data written to the scan chain (Si) has logic value 'b', where 'a' and 'b' can be '0' or '1'. The "rx" in march element M1 denotes that the read data is a "don't care" value. It can be found that two *horizontally redundant* operations, each with $(\mathrm{r1w1})^{c-c'}$, will exist if the buffer width c' is smaller than c. Here, c' (n') is the word width (word number or capacity) of the memory buffer under consideration, while c (n) is the largest word width (number) of all memory buffers. Let us consider an example [10] to illustrate how the redundant operations enable parallel testing of different sized memory buffers.

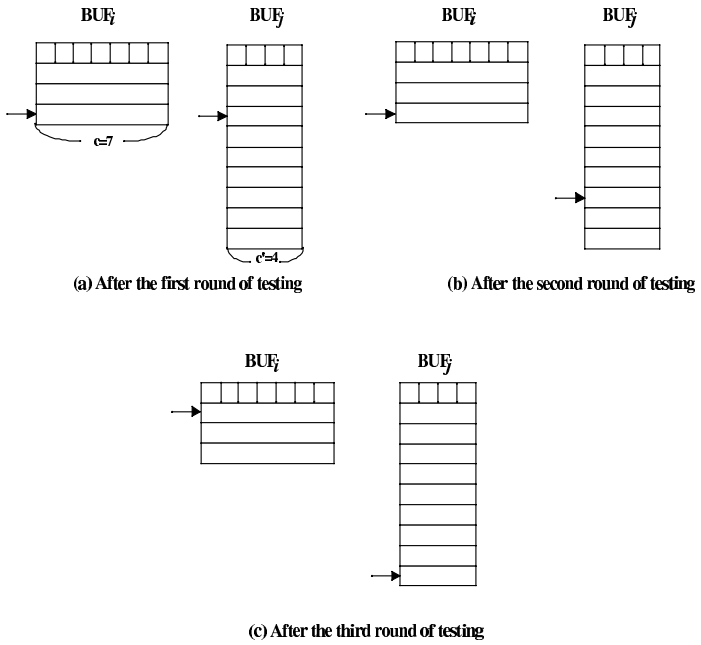

Figure 2.4: Testing of multiple buffers using redundant operations

For the example in Figure 2.4, we have the horizontal operations of $(\mathrm{r0w1})^7(\mathrm{r1w1})^{7-7}$ $(\mathrm{r1w1})^7(\mathrm{r1w1})^{7-7}$for BUF_i. Note that the BUF_i is the memory buffer with the maximum buffer width (7). Similarly, we have the horizontal operations of $(\mathrm{r0w1})^4$ $(\mathrm{r1w1})^{7-4}(\mathrm{r1w1})^4$ $(\mathrm{r1w1})^{7-4}$ for BUF_j, where $(\mathrm{r1w1})^{7-4}$ is a horizontally redundant operation due to the smaller word width in BUF_j.

The second row of march element M2 represents the marching operations by which the entire array is excessively scanned. These are the vertical redundant operations of the memory buffers. For example in Figure 2.4, the BUF_i has word number n' equal 4 and the deepest buffer BUF_j has (maximum) word number equal 10 (i.e., *n=10*), then the entire array of BUF_i will be superfluously scanned one more time. This depends on the relationship between n and n'. Finally, the third row of each march element as shown in Figure 2.3 gives the marching operations which

excessively scan part of the memory array when the march element finally terminates. Both the second and the third row of M2 introduce what we term as *vertically redundant* operations. For n=10 and n'=4, the first two words of BUF_i will be scanned three times when the entire march element is finished. Thus, we have the vertically redundant operations of $(\Uparrow_0^3(\text{r1w1})^{2*7})^1$ for all words of BUF_i and $(\text{r1w1})^{2*7}$ for the first two words of BUF_i (i.e., $\Uparrow_0^1(\text{r1w1})^{2*7}$). However, the vertically redundant operations will not be existent for BUF_j. Again, the third test session of M2 may or may not exist depending on the relationship between n' and n. [10].

The redundant operations have been introduced so that parallel testing of memory buffers of different sizes could be done. In this case all the memory buffers receive the same test patterns. The memory buffers are implemented using SRAM. An appropriate memory cell array fault model is discussed in Chapter 3. Next, we consider how only certain bit-patters are generated if we use the serial interfacing technique and hence introduce the concept of the bi-directional serial interfacing technique.

2.4 Selective Bit-Pattern Generation and Bi-Directional Serial Interface

Due to the serial scan circuit structure, in the serial interfacing technique, a special limitation called *selective bit-pattern generation* can greatly affect the testing of certain faults. Let us discuss the bit-patterns generated when data is written into the scan-in port S/i and read from the scan-out port S/o as shown in Figure 2.5. Two cases have been depicted:

1. Write all cells with logic 1 in a memory that initially contains all 0's (left side of Figure 2.5)
2. Write all cells with logic 0 in a memory that initially contains all 1's (right side of Figure 2.5)

These bit-patterns can help detect certain faults, however not all faults. Hence, we need a complementary set of bit-patterns which can sensitize and detect the remaining faults. The bi-directional serial interfacing technique helps achieve our goal in this respect, since we can generate a new set of bit-patterns shown in Figure 2.6. The bi-directional serial interface architecture is shown in Figure 2.7.

In this new structure, additional hardware must be added to support the new test architecture. From Figure 2.7, we observe that the solid lines represent the right shift path while the dashed lines denote the left-shift path. Each of both paths is enabled by the shift_left/right signal. Therefore, the data flow direction is controlled by selecting an appropriate input for the multiplexers. It is important to note that the hardware overhead introduced due to this new architecture is not much. Based on this bi-directional serial interface structure, we are able to read and write data in a right-shift or left-shift way. When the multiplexers receive the shift_right (left) command, then the serial interface performs a right (left) shift operation. Note where the test patterns are applied and from which end the test responses of the memory word are read. When the serial interface is performing the right (left) shift operation, S/i feeds test patterns to the left-hand (right-hand) side of the scan chain/memory word, while S/o receives test responses from the right-hand (left-hand) side of the scan chain/memory word.

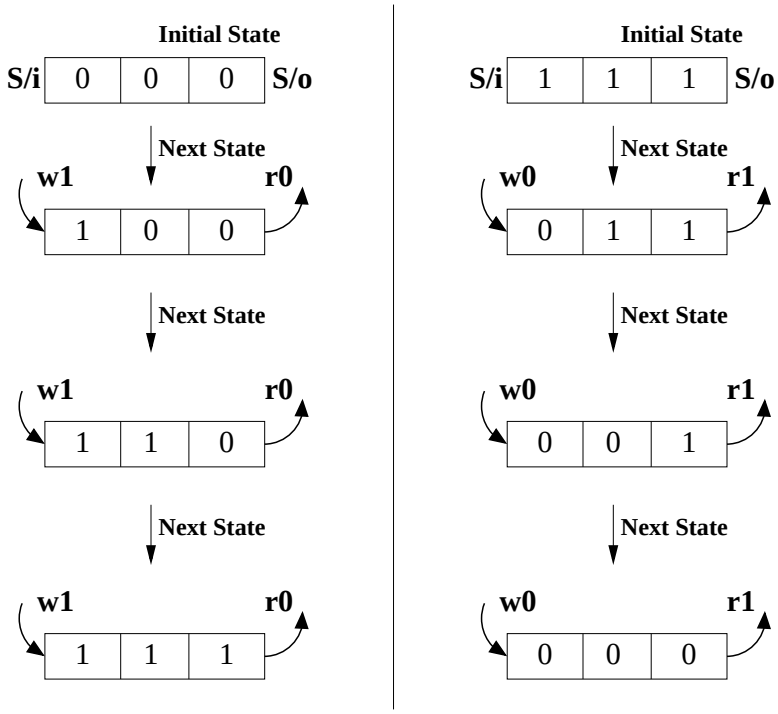

Figure 2.5: Selective bit-pattern generation

2.5 Multiple-Port Memory Testing

Multi-port memories are widely used in multi-processor systems and special applications such as telecommunications, etc. However, in spite of their increasing popularity, limited work on their testability has been published. An ad-hoc test technique with no specific fault model was described in [20]. Serial test algorithms for embedded multi-port memories were reported in [4]. However, the used fault models are very simplistic and restricted to shorts between ports. A new fault model, the so-called complex coupling fault, and its test was developed by [1] and [2]. This fault model is based on the traditional idempotent coupling fault (CFid). The individual CFids of which the complex coupling fault is composed, are too weak to sensitize a fault; however, their fault effects may be combined when the CFids are activated through different ports. This makes this fault model unique for multi-port memories. However, this is just one way in which fault effects may be combined; many fault models exist, such that many combined fault effects may result. In [28] and [27], it has been shown theoretically that the conventional tests for single-port memories are insufficient for multi-port memories. Moreover, theoretical fault models together with their tests were developed. However, the introduced fault models are not based on any experimental/industrial analysis. In addition, the proposed tests have a time complexity which is exponentially proportional with the number of ports in the multi-port memory, that makes them not practical. In [29], port interferences in 2P memories were experimentally analyzed, based on an industrial design and SPICE simulation; however, the analysis was restricted only to the interference between two ports. A similar but theoretical work has been reported in [34].

In [31], a complete analysis of all spot-defects in a p-port SRAM design has been performed based on simulation. A transformation of electric faults caused by the defects into realistic functional fault models (FFM) has been presented. It was understood that p-port faults cannot be

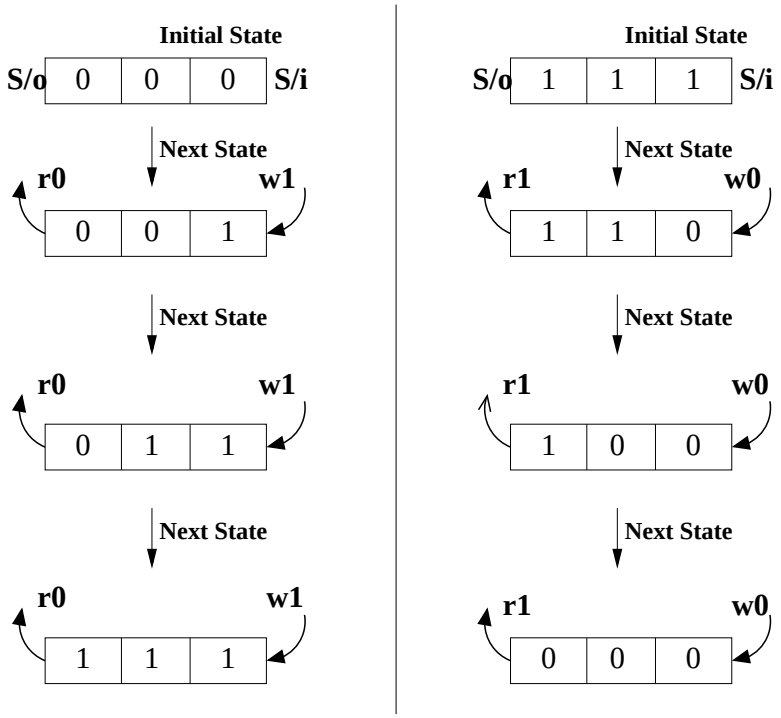

Figure 2.6: Complimentary bit-pattern generation

sensitized using single-port operations and the authors came to a conclusion that special tests for multi-port faults (i.e 2PF, 3PF etc.) are required. The time complexity of test algorithms was explained in this paper, and it was shown that the worst-case time complexity to test a multi-port memory is θ(n,p) where n is the max size of the memory and p is the number of ports. This was found very much feasible by industrial standards. In [30], a *parallel testing methodology* was used to test 2P faults involving single and double cells. Two reduced parallel algorithms namely March S2PF- and March D2PF- shown below were proposed. March S2PF- can detect all 2PF1, 2PF2aa and 2PF2vv faults, while March D2PF- can detect 2PF2av faults.

March S2PF-

{

$\Uparrow (w0 : n)$;

$\Uparrow (r0 : r0, r0 : \_, w1 : r0)$;

$\Uparrow (r1 : r1, r1 : \_, w0 : r1)$;

$\Downarrow (r0 : r0, r0 : \_, w1 : r0)$;

$\Downarrow (r1 : r1, r1 : \_, w0 : r1)$;

$\Downarrow (r0 : \_)$;

}

March D2PF-

{

$\Uparrow (w0 : n)$;

$\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r0_{r+1,c}, r1_{r,c} : w1r+1, c, w0_{r,c} : r1_{r+1,c}, r0_{r,c} : w0r+1, c))$;

$\Uparrow_{c=0}^{C-1} (\Uparrow_{r=0}^{R-1} (w1_{r,c} : r0_{r,c+1}, r1_{r,c} : w1r, c+1, w0_{r,c} : r1_{r,c+1}, r0_{r,c} : w0r, c+1))$;

}

The testing methodology used is

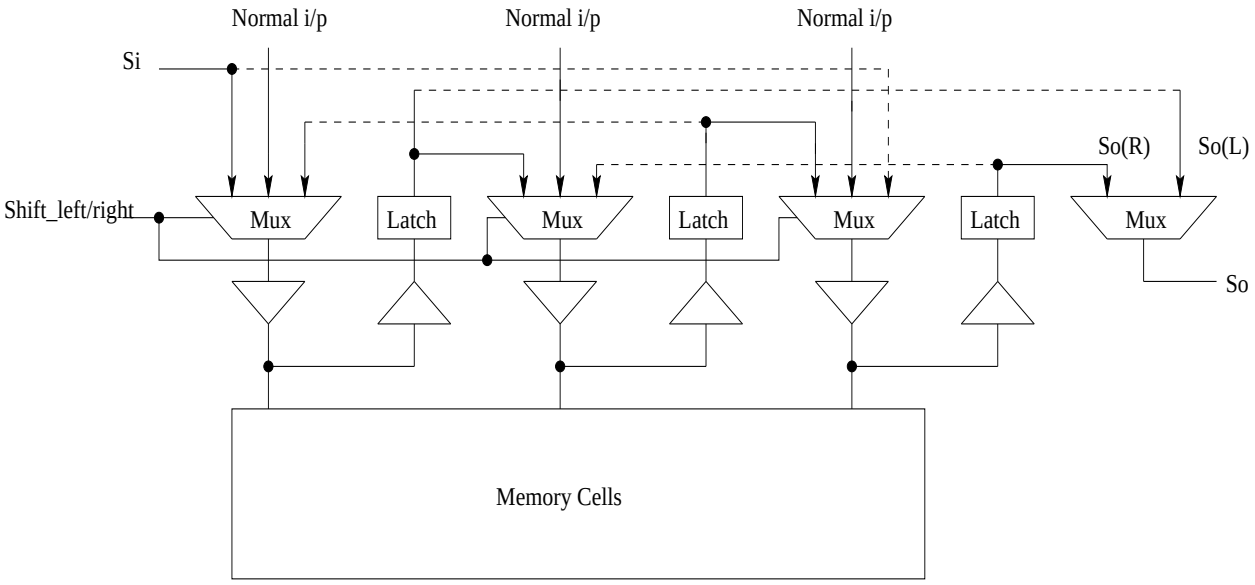

Figure 2.7: Bi-directional serial interface architecture

1. To apply test patterns to detect single port faults on Port A and Port B.

2. For dies that pass step 1, apply March S2PF- and March D2PF- through both ports.

The parallel algorithm March S2PF- proposed in [30] can only detect faults which are sensitized by a simultaneous write operation on Port A and read operation on Port B. However, we can have a fault which is sensitized by a simultaneous read operation on Port A and a write operation on Port B. This has not been accounted for. We fix the problem in this thesis by proposing a new parallel algorithm which accounts for all four cases which sensitize faults namely - r0:w1, r1:w0, w0:r1, w1:r0.

The parallel testing method is efficient for few long buffers which are not physically spaced too far away from one another. However, when we consider the case of a vast number of distributed small buffers, this technique introduces tremendous overhead in area and power. Hence, we propose a serial testing scheme to test distributed small buffers. The fault models considered are discussed in Chapter 3 in detail. March algorithms to detect the stated fault models are discussed in Chapters 4 and 5. Note that faults in the same word are also considered in Chapter 6, which have not been covered in other research efforts in the past.

Chapter 3

Two-Port Fault Models

In this chapter, we will define fault models for two-port memories[28] to give readers enough background information to understand many different faults. We follow the notation used in [31] and [30].

Strong fault - A memory fault that can be fully sensitized by an operation, e.g., SP read or write operation fails, two simultaneous read operations fail etc. This means that the state of the v-cell is incorrectly changed, cannot be changed, or that sense amplifiers return incorrect results.

Weak fault - A memory fault which is partially sensitized by an operation, e.g., due to a defect that creates a small disturbance of the voltage of a cell. A fault can be fully sensitized (i.e. becomes strong) when two or more weak faults are sensitized simultaneously, since their fault effects can be additive. This may occur when a 2P operation is applied.

In the presence of a weak fault, all SP(read and write) operations pass correctly and 2P operations may pass correctly. Latter is the case where the weak fault effects of the weak faults are not strong enough to fully sensitize the fault.

Notation for strong and weak faults

1. *<fault1> & <fault2>*: This denotes a 2PF consisting of two weak faults, and "&" denotes the fact that both faults occuring simultaneously form the 2PF.
2. F denotes a strong fault F, while wF denotes a weak fault. For example RDF denotes a strong read destructive fault, while wRDF denotes a weak read destructive fault.

A two-port faults cannot be sensitized using SP operations, and it requires the use of two ports simultaneously. A 2PF is a combination of two weak faults. Fault effects of two or more weak faults may be additive, and hence can be fully sensitized when the weak faults are activated simultaneously.Two-port faults can be divided into faults involving a single cell and faults involving two cells as follows.:

- 2PF1 - combination of two single-cell weak faults.
- 2PF2 - combination of weak single-cell faults involving two cells.
  - 2PF2aa
  - 2PF2vv
  - 2PF2av

A taxonomy of realistic 2PFs is given in Fig 3.1, and a detailed list of all 2PFs is given in Table 3.1.

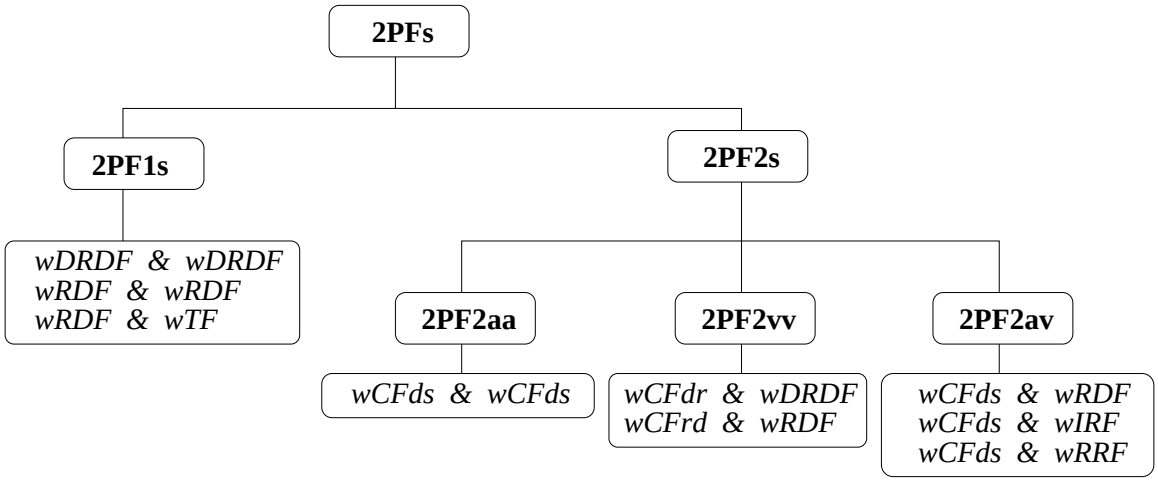

Figure 3.1: Taxonomy of 2PFs

3.1 Single-Cell Two-Port Faults

This section describes two-port faults where only one cell is involved. The 2PF1s are based on combination of two single-cell weak faults. Also, the two a-cells are the same as the v-cell. In order to sensitize a 2PF1, the same cell has to be acted upon simultaneously via the two ports. To denote a 2PF1 fault, the following notation can be used.

$< S1 : S2/F/R >$ - It denotes a two-port fault involving a single victim cell. S1 and S2 describe the sensitizing operations or states of the cell, while ":" denotes that S1 and S2 are applied simultaneously through the two ports. F describes the faulty value of the v-cell. The sensitizing operations are applied to the same cell as where the fault appears. R is the read result of S1 (and S2) if it is a read operation.

1. *wDRDF & wDRDF:* Applying simultaneous read to a single cell causes the cell to flip, while the sense amplifier returns the correct value. There are two 2FP's (fault primitives): $< r0 : r0/ \uparrow /0 >$ and $< r1 : r1/ \downarrow /1 >$
2. *wRDF & wRDF:* Applying simultaneous read operations to a single cell causes the cell to flip, and the sense amplifier returns an incorrect value. There are two 2FP's: $< r0 : r0/ \uparrow /1 >$ and $< r1 : r1/ \downarrow /0 >$
3. *wRDF & wTF:* A cell fails to undergo a write transition if a read operation is applied to the same cell simultaneously. There are two 2FP's: $< r0 : w \uparrow /0/\_ >$ and $< r1 : w \downarrow /1/\_ >$

3.2 Double-Cell Two-Port Faults

This section describes two-port faults where each one involves two cells. Depending on to which cells the two simultaneous operations are applied (to the a-cell or the v-cell), the 2PF2s are divided into three types which are explained below.

3.2.1 2PF2aa

This fault is sensitized in victim cell Cv by applying two simultaneous operations to the same aggressor cell Ca. In this case, the 2PF is combination of two weak faults involving two cells,

FFM	Fault Primitives
wDRDF & wDRDF	$< r0 : r0/\uparrow/0 >, < r1 : r1/\downarrow/1 >$
wRDF & wRDF	$< r0 : r0/\uparrow/1 >, < r1 : r1/\downarrow/0 >$
wRDF & wTF	$< r0 : w\uparrow/0/\_ >, < r1 : w\downarrow/1/\_ >$
wCFds & wCFds	$< w0 : rd; 0/\uparrow/\_ >, < w0 : rd; 1/\downarrow/\_ >$ $< w1 : rd; 0/\uparrow/\_ >, < w1 : rd; 1/\downarrow/\_ >$ $< rx : rx; 0/\uparrow/\_ >, < rx : rx; 1/\downarrow/\_ >$
wCFdr & wDRDF	$< 0; r0 : r0/\uparrow/0 >, < 0; r1 : r1/\downarrow/1 >$ $< 1; r0 : r0/\uparrow/0 >, < 1; r1 : r1/\downarrow/1 >$
wCFrd & wRDF	$< 0; r0 : r0/\uparrow/1 >, < 0; r1 : r1/\downarrow/0 >$ $< 1; r0 : r0/\uparrow/1 >, < 1; r1 : r1/\downarrow/0 >$
wCFds & wRDF	$< w0 : r0/\uparrow/1 >, < w0 : r1/\downarrow/0 >$ $< w1 : r0/\uparrow/1 >, < w1 : r1/\downarrow/0 >$
wCFds & wIRF	$< w0 : r0/0/1 >, < w0 : r1/1/0 >$ $< w1 : r0/0/1 >, < w1 : r1/1/0 >$
wCFdS & wRRF	$< w0 : r0/0/? >, < w0 : r1/1/? >$ $< w1 : r0/0/? >, < w1 : r1/1/? >$

Table 3.1: List of 2PFs; x=0,1 and d=don't care

and both weak faults have the same a-cell and v-cell cell. We denote a 2PF2aa using the following notation.

$< Sa : Sa; Sv/F/R >$ - This denotes a fault primitive whereby both sensitizing operations (Sa:Sa) are applied simultaneously to the a-cell. Sv denotes the state of the v-cell. F denotes the value of the faulty cell Cv. R will be replaced by a "_", since Sv cannot be a read operation. The 2PF2aa consists of one functional fault model (FFM): *wCFds & wCFds*. Applying two simultaneous operations to the same a-cell will sensitize a fault in the v-cell ; i.e. v-cell flips

$< w0 : rd; 0/\uparrow/\_ >$, $< w0 : rd; 1/\downarrow/\_ >$
$< w1 : rd; 0/\uparrow/\_ >$, $< w1 : rd; 1/\downarrow/\_ >$
$< r0 : r0; 0/\uparrow/\_ >$, $< r0 : r0; 1/\downarrow/\_ >$
$< r1 : r1; 0/\uparrow/\_ >$, $< r1 : r1; 1/\downarrow/\_ >$

3.2.2 2PF2vv

This fault is sensitized in cell Cv by applying two simultaneous operations to the same cell Cv, while the a-cell has to be in a certain state. This fault is a combination of two weak faults involving two cells whereby the operation has to be performed to the v-cell, while the a-cell is in a certain state. We denote a 2PF2vv using the following notation.

$< Sa; Sv : Sv/F/R >$ - It denotes a fault primitive whereby both sensitizing operations (Sv:Sv) are applied simultaneously to the v-cell. Also, Sa describes the state of the a-cell. 2PF2vv consists of two FFMs: *wCFrd & wRDF* with 4 FP's and *wCFdr & wDRDF* with 4 FP's. If the a-cell is in a certain state and two simultaneous reads are performed to the v-cell, then the v-cell flips with either correct or incorrect output.

1. *wCFrd & wRDF*

   $< 0; r0 : r0/\uparrow/1 > < 0; r1 : r1/\downarrow/0 >$

$< 1; r0 : r0/ \uparrow /1 > < 1; r1 : r1/ \downarrow /0 >$

2. *wCFdr & wDRDF*

   $< 0; r0 : r0/ \uparrow /0 > < 0; r1 : r1/ \downarrow /1 >$

   $< 1; r0 : r0/ \uparrow /0 > < 1; r1 : r1/ \downarrow /1 >$

3.2.3 2PF2av

This fault is sensitized by applying two simultaneous operations: one to cell Ca and the other to cell Cv, and can be represented by the following notation.

$< Sa; Sv/F/R >$ - It denotes a fault primitive whereby the sensitizing operation Sa is applied to the a-cell simultaneously with the sensitizing operation Sv applied to the v-cell. This type of fault consists of three FFMs: *wCFds & wRDF* , *wCFds & wIRF* and *wCFds & wRRF* each with 4 FP's.

1. *wCFds & wRDF*: A read operation applied to cell Cv flips the cell and the sense amplifier returns an incorrect value, if a write operation is applied to cell Ca simultaneously.

   $< w1 : r1/ \downarrow /0 > < w1 : r0/ \uparrow /1 >$

   $< w0 : r1/ \downarrow /0 > < w0 : r0/ \uparrow /1 >$

2. *wCFds & wIRF*: A read operation applied to cell Cv returns an incorrect value, if a write operation is applied to cell Ca simultaneously. The state of Cv does not change.

   $< w1 : r1/1/0 > < w1 : r0/0/1 >$

   $< w0 : r1/1/0 > < w0 : r0/0/1 >$

3. *wCFds & wRRF*: A read operation applied to cell Cv returns a random value if a write operation is applied to cell Ca simultaneously. The state of Cv does not change.

   $< w1 : r1/1/? > < w1 : r0/0/? >$

   $< w0 : r1/1/? > < w0 : r0/0/? >$

Chapter 4

Serial Interfacing Design for Multiport Memory Testing

In this chapter we propose the idea of serial interfacing to sensitize and detect one particular type of two-port faults - 2PF1. March algorithms to detect the other types of two-port faults can be found in the next section. So, let us investigate the march algorithm to detect 2PF involving one cell assuming a parallel architecture first, then transform the same algorithm to a serial scheme. FFMs involved in 2PFs are discussed in Table 3.1. In this chapter, we consider only the FFMs for 2PF1 faults as shown in Table 4.1.

FFM	**Fault Primitives**
wDRDF & wDRDF	$< r0 : r0/\uparrow/0 >, < r1 : r1/\downarrow/1 >$
wRDF & wRDF	$< r0 : r0/\uparrow/1 >, < r1 : r1/\downarrow/0 >$
wRDF & wTF	$< r0 : w\uparrow/0/\_ >, < r1 : w\downarrow/1/\_ >$

Table 4.1: List of 2PF1s

In the following section, we will discuss some basic conditions which need to be satisfied in a march algorithm to detect 2PF1s. We will investigate the conditions for all three types of 2PF1 faults, and derive a condition for 2PF1 faults as a whole. Then, we will present a parallel march algorithm which satisfies the condition to detect 2PF1 faults. Finally, we shall derive a serial scheme for the parallel march 2PF1 algorithm.

4.1 Conditions to Detect 2PF1

4.1.1 Conditions to Detect wDRDF & wDRDF

The *wDRDF & wDRDF* FFM consists of two fault primitives (FPs): $< r0 : r0/\uparrow/0 >$ and $< r1 : r1/\downarrow/1 >$. It has to be considered only for 2P memories which allow two simultaneous read operations from the same location (i.e, address). Any *wDRDF & wDRDF* fault can be detected by a march test which contains the two march elements of Case A and the two march elements of Case B below.

1. Case A : detection of $< r0 : r0/\uparrow/0 >$

   $\Updownarrow (... , r0 : r0); \Updownarrow (r0 : \_ , ...)$

2. Case B : detection of $< r1 : r1/\downarrow/0 >$

   $\Updownarrow (... , r1 : r1); \Updownarrow (r1 : \_ , ...)$

The first pair of simultaneous read operations through both ports in each first march element sensitizes the fault, and the fault effect will be detected by the single read operation of each second march element. Note that "$\_$" denotes any allowed operation.

4.1.2 Conditions to Detect wRDF & wRDF

The *wDRDF & wDRDF* FFM consists of two FPs: $< r0 : r0/\uparrow/1 >$ and $< r1 : r1/\downarrow/0 >$. It has to be considered only for 2P memories which allow two simultaneous read operations of the same location. Any *wRDF & wRDF* fault will be detected by a march test which contains the two march elements of Case A and the two march elements of Case B below.

1. Case A : detection of $< r0 : r0/\uparrow/1 >$

   $\Updownarrow (... , r0 : r0, \ ...)$

2. Case B : detection of $< r1 : r1/\downarrow/0 >$

   $\Updownarrow (... , r1 : r1, \ ...)$

The pair of simultaneous read operations through the two ports in each march element sensitize and detect the fault. It is easy to understand that the condition for *wRDF & wRDF* is a subset of the condition for *wDRDF & wDRDF* . Hence any test detecting *wDRDF & wDRDF* will also detect *wRDF & wRDF* .

4.1.3 Conditions to Detect wRDF & wTF

The *wRDF & wTF* FFM consists of two FPs: $< r0 : w\uparrow/0/\_ >$ and $< r1 : w\downarrow/1/\_ >$. It has to be considered only for 2P memories which allow two simultaneous read and write operations at the same location, whereby the read data is discarded. Any *wRDF & wTF* fault will be detected by a march test which contains the two march elements of Case A and the two march elements of Case B.

1. Case A : detection of $< w\uparrow: r0/0/\_ >$

   $\Updownarrow (... , w1 : r0)$; $\Updownarrow (r1 : \_ , ...)$

2. Case B : detection $< w\downarrow: r1/1/\_ >$

   $\Updownarrow (... , w0 : r1)$; $\Updownarrow (r0 : \_ , ...)$

The pair of simultaneous operations through both ports in each march element sensitize the fault, which will be detected by the second single read operation.

4.1.4 Conditions to Detect All 2PF1 Faults

Any 2PF1 fault will be detected by a march test which contains both pairs of march elements of Case A (i.e. A(i) and A(ii)) or both pairs of march elements of Case B (i.e. B(i) and B(ii)).

1. Case A:

   (i) To detect $< r0 : r0/ \uparrow /0 >$, $< r0 : r0/ \uparrow /1 >$ and $< w \downarrow : r1/1/\_ >$

   $\Updownarrow (... , w0 : r1)$; $\Updownarrow (... , r0 : r0)$; $\Updownarrow (r0 : \_ , ...)$

   (ii) To detect $< r1 : r1/ \downarrow /1 >$, $< r1 : r1/ \downarrow /0 >$ and $< w \uparrow : r0/0/\_ >$

   $\Updownarrow (... , w1 : r0)$; $\Updownarrow (... , r1 : r1)$; $\Updownarrow (r1 : \_ , ...)$

2. Case B:

   (i) To detect $< r0 : r0/ \uparrow /0 >$, $< r0 : r0/ \uparrow /1 >$ and $< w \uparrow : r0/0/\_ >$

   $\Updownarrow (... , r0 : r0)$; $\Updownarrow (... , r0 : \_ , ...)$

   $\Updownarrow (... , w1 : r0)$; $\Updownarrow (... , r1 : \_ , ...)$

   (ii) To detect $< r1 : r1/ \downarrow /1 >$, $< r1 : r1/ \downarrow /0 >$ and $< w \downarrow : r1/1/\_ >$

   $\Updownarrow (... , r1 : r1)$; $\Updownarrow (... , r1 : \_ , ...)$

   $\Updownarrow (... , w0 : r1)$; $\Updownarrow (... , r0 : \_ , ...)$

This *condition 2PF1* applies to a 2P memory supporting simultaneous read and write of the same location. If this is not the case, then the FFM *wRDF & wTF* is not realistic, and hence *condition 2PF1* can be simplified to *wRDF & wRDF* .

4.2 Parallel March Algorithm 2PF1

Let us use the conditions described in the previous section to frame a parallel marching algorithm that can detect all 2PF1 faults. The parallel march algorithm to detect all two port faults involving a single cell is:

$\{ \Uparrow \frac{M0}{(w0:n)}; \Uparrow \frac{M1}{(w1:r0,r1:r1,r1:w0,r0:w1)}; \Uparrow \frac{M2}{(w0:r1,r0:r0,r0:w1,r1:n)}; \}$

There are three march operations M0, M1 and M2. M1 has four elements namely M1-1, M1-2, M1-3, M1-4. Similarly, M2 has four elements namely M2-1, M2-1, M2-3, M2-4. We can observe on close inspection that this algorithm satisfies the conditions to detect all 2PF1 faults. Refer Table. 4.2.

Condition Case	March Operation
A(i)	M2-1, M2-2, M2-3
A(ii)	M1-1, M1-2, M1-3
B(i)	M2-2, M2-3, M1-1, M1-2
B(ii)	M1-2, M1-3, M2-1, M2-2

Table 4.2: Parallel march algorithm 2PF1 satisfies condition 2PF1

M0:

It has only one operation and initializes all memory cells to 0.

M1:

1. (w1 : r0) sensitizes the fault $< w \uparrow : r0/0/\_ >$. Here, by w $\uparrow$, the cell value must have changed to logic 1. Instead, if it remains a 0, the fault can be detected by the next read operation in M1.

2. (r1 : r1) sensitizes the fault $< r1 : r1/\downarrow/1 >$, and detects the fault $< r1 : r1/\downarrow/0 >$. Further, it detects $< w\uparrow : r0/0/\_ >$ sensitized by the (w1:r0) operation.

3. (r1 : w0) detects $< r1 : r1/\downarrow/1 >$ and sensitizes $< r1 : w\downarrow/1/\_ >$.

4. (r0 : w1) detects $< r1 : w\downarrow/1/\_ >$ and prepares for operation (w0 : r1) in M2 by writing the memory with all 1's.

M2:

1. (w0 : r1) sensitizes the fault $< w\downarrow : r1/1/\_ >$. Here, by w↓, the cell value must have changed to a 0. Instead, if it remains a logic 1, the fault can be detected by the next read operation in M2.

2. (r0 : r0) sensitizes the fault $< r0 : r0/\uparrow/0 >$ and detects the fault $< r0 : r0/\uparrow/1 >$. Further, it detects $< w\downarrow : r1/1/\_ >$ sensitized by the (w0:r1) operation.

3. (r0 : w1) detects $< r0 : r0/\uparrow/0 >$ and sensitizes $< r0 : w\uparrow/0/\_ >$.

4. (r1 : n) detects $< r0 : w\uparrow/0/\_ >$.

Thus, we see that with this march algorithm, all 2PF1 faults can be detected. Next, let us implement this serially.

4.3 Serial March Algorithm 2PF1

We take a three-cell memory array to describe the serial implementation technique used [Fig. 4.1]. C1, C2 and C3 are memory cells. LA-1(LB-1), LA-2(LB-2) and LA-3(LB-3) are latches corresponding to port A(B).

M0:

This operation just writes 0's to all cells. The corresponding serial implementation is $(w0 : n, rx : n)^3$. This writes logic 0 to all cells serially. Note that $(w0 : n, rx : n)^3$ means doing a (w0: n, rx : n) three times where "n" denotes a no-operation, and port A is used to initialize all cells to logic 0. After $(w0 : n, rx : n)^3$, the current memory state would be as in Fig. 4.1. Note that port A is depicted in this figure in bold lines and port B in normal lines. Dotted lines are used for control read/write signals.

M1:

Next, we want to perform (w1:r0) simultaneously through both ports, w1 through port A and r0 through port B. This can be implemented serially by $(w1 : r0, r0 : w1)^3$. Operation $(w1 : r0)$ means that we perform w1 (r0) at input (output) port A (B) of the serial interface in Fig. 4.1. Operation $(r0 : w1)$ can be discussed similarly.

w1:r0,r0:w1 - 1st time
Here, the 1st cell C1 is written with 1, and 0 is read from C1 at the same time by $(w1 : r0)$ as shown in Fig. 4.2. Note that simultaneous write and read operations are performed to cells C2 and C3 as well, though value 0 is written into both cells. In Fig. 4.3, logic 1 is written to latch LA-1 by $(r0 : w1)$, and the logic value will be propagated to C2 in the $2nd$ time operation discussed below.

w1:r0,r0:w1 - 2nd time

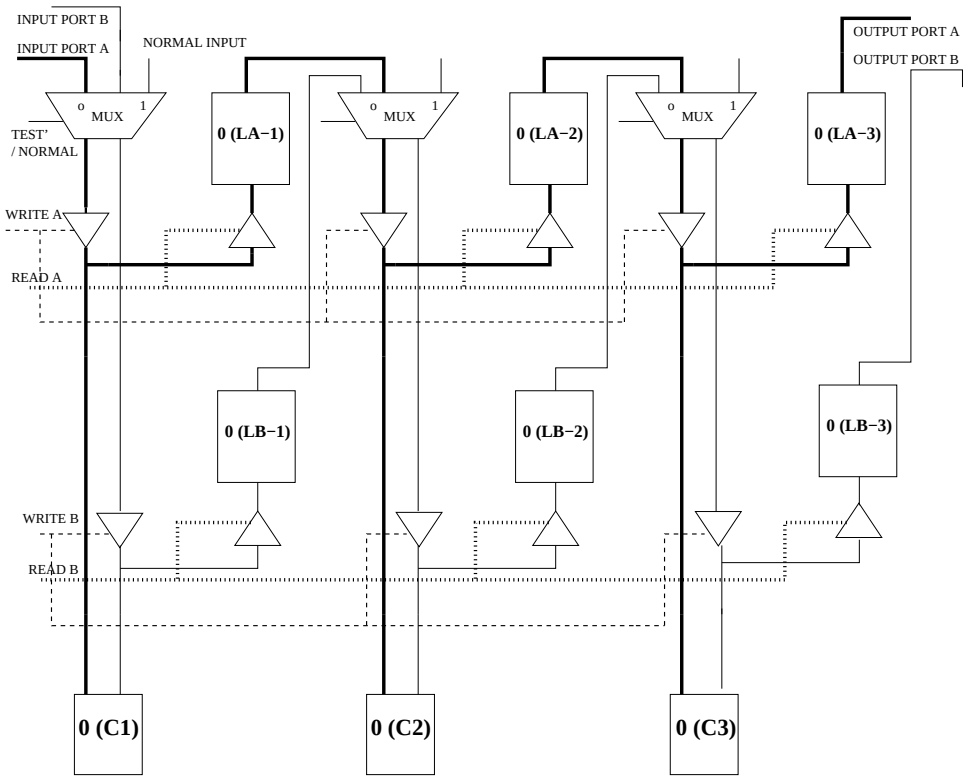

Figure 4.1: 2PF1 serial interfacing memory state after $(w0 : n, rx : n)^3$

Now, logic 1 stored in LA-1 is propagated to C2 and then LA-2 as shown in Fig. 4.4 and Fig. 4.5. The simultaneous write and read operations are performed to cells C1 and C3 as well with logic values 1 and 0 respectively.

w1:r0,r0:w1 - 3rd time
Finally, logic 1 stored in LA-2 is propagated to C3 and the LA-3 as shown in Fig. 4.6 and Fig. 4.7.

Thus, we have successfully performed w1:r0 serially. Now, the memory contains all 1's. Note that if the 1st cell has a $< w1 : r0/0/\_ >$ fault, then 2nd and 3rd cells will not be tested, because no logic 1 will be propagated to C2 and C3. But, the fault effect at C1 will be propagated to the serial scan output and observed. The discussion can be extended to defective C2 or C3 cells. *Thus,* $< w1 : r0/0/\_ >$ *fault is detected until now.* This forms the basic idea of using serial interfacing for two-port memories. We emphasize that the (w1:r0) sub-operation of the ith (w1:r0, r0:w1) operation is used to sensitize the $< w1 : r0/0/\_ >$ fault for cell Ci. The second sub-operation (r0:w1) is just used to propagate fault effects as will be discussed in Theorem 4.1. The role of each sub-operation for different fault types might change as will be discussed in Theorems 4.1 to 4.3.

Consequently, the serial equivalent of parallel operation (w0 : r0) would be $(w0 : r0, r0 : w0)^n$, while the serial equivalent of (w1 : r0) would be $(w1 : r0, r0 : w1)^n$, where n is the width of the array. The serial equivalents of other parallel march operations can be discussed similarly. Proceeding in the same way using the parallel 2PF1 march algorithm as the base, we get an equivalent serial march algorithm which uses only one test input and one test output to thoroughly sensitize and detect all 2PF1 faults. However, we *do not* have to create a one-to-one correspondence between parallel and serial testing algorithms. A little tweaking of the serial algorithm would help us reduce more operations and decrease redundancy. We arrived at a concise serial march algorithm

to detect all 2PF1 faults as follows.

$$\{ \Uparrow \frac{M0}{(w0{:}n)} \; ; \Uparrow \frac{M1{-}1}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}2}{(r1{:}r1)}, \frac{M1{-}3}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}4}{(r0{:}r0)}, \frac{M1{-}5}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}6}{(w0{:}r1,r1{:}w0)^3} \}$$

There are two march operations M0 and M1. The 6 elements of march operation M1 are labelled from M1-1 to M1-6. M1-1(i) denotes the element M1-1 being performed the ith time in the algorithm. For example, M1-3(2) denotes (r1 : w0 , w0 : r1) being performed the second time.

Theorem 4.1. *All wRDF & wTF faults can be detected by the serial 2PF1 algorithm.*
Proof : Without loss of generality, we use a 3-bit memory array with two ports as shown in Fig. 4.1. Assume bit C1 has fault $< w1 : r0/ \downarrow /\_ >$. By applying the first operation (i.e., w1:r0) of M1-1(1), C1 contains a faulty logic value 0. The faulty value will be latched into LA-1 by the second operation (i.e., r0:w1) of M1-1(1).

By applying the first operation of M1-1(2), the fault effect stored in LA-1 will be written into C2, and then propogated to LA-2 by the second operation of M1-1(2). Finally, the fault effect stored in LA-2 will be written to C3 by the first operation of M1-1(3), and then propogated to LA-3 by the second operation of M1-1(3). Thus, the fault effect can be observed by the scan output. **Q.E.D**

The detection of faults in other bits can be discussed similarly. Further, the detection of other wRDF *&* wTF faults can be proved in the same manner.

Theorem 4.2. *All wDRDF & wDRDF faults can be detected by the serial 2PF1 algorithm.*
Proof : Assume bit C1 has fault $< r0 : r0/ \uparrow /0 >$. By applying operation M1-4 (i.e., r0:r0), C1 contains a faulty logic value 1 but the value latched into LA-1 and LB-1 is still logic value 0 at this time. This is the wDRDF *&* wDRDF fault. The faulty value, however, will be latched into LA-1 by the first operation (i.e., r0:w1) of M1-5(1).

By applying the second operation of M1-5(1), the fault effect stored in LA-1 will be written into C2, and then propogated to LA-2 by the first operation of M1-5(2). Finally, the fault effect stored in LA-2 will be written to C3 by the second operation of M1-5(2), and then propogated to LA-3 by the first operation of M1-5(3). Thus, the fault effect can be observed by the scan output. **Q.E.D**

The detection of faults in other bits can be discussed similarly. Further, the detection of other wDRDF *&* wDRDF faults can be proved in the same manner.

Theorem 4.3. *All wRDF & wRDF faults can be detected by the serial 2PF1 algorithm.*
Proof : This proof can be performed similar to the proof of Theorem 4.2. **Q.E.D**

Thus, we conclude from Theorems 4.1, 4.2 and 4.3 that the serial march algorithm 2PF1 can sensitize and detect all two port faults involving a single cell.

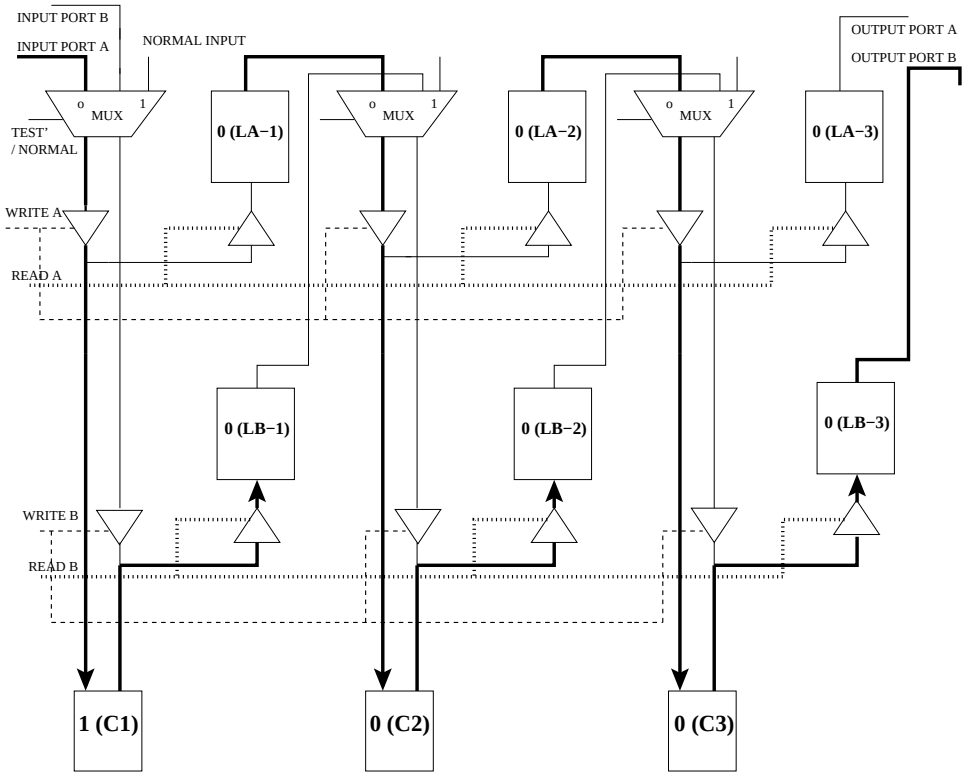

Figure 4.2: 2PF1 serial interfacing memory state (w1 : r0) - 1st time

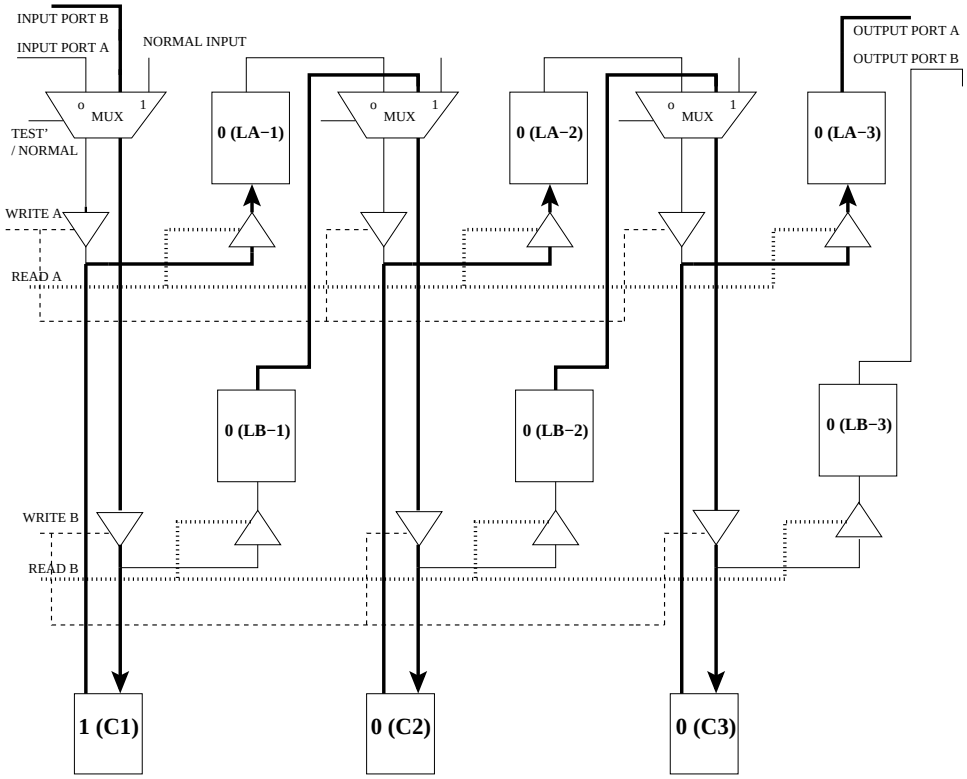

Figure 4.3: 2PF1 serial interfacing memory state (r0 : w1) - 1st time

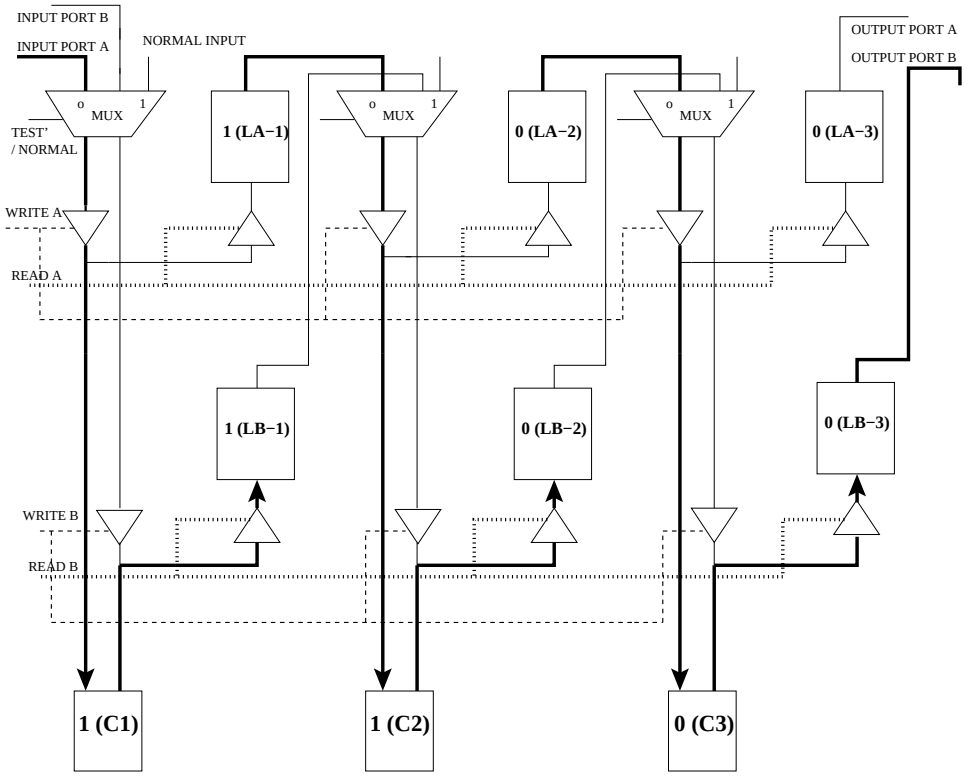

Figure 4.4: 2PF1 serial interfacing memory state (w1 : r0) - 2nd time

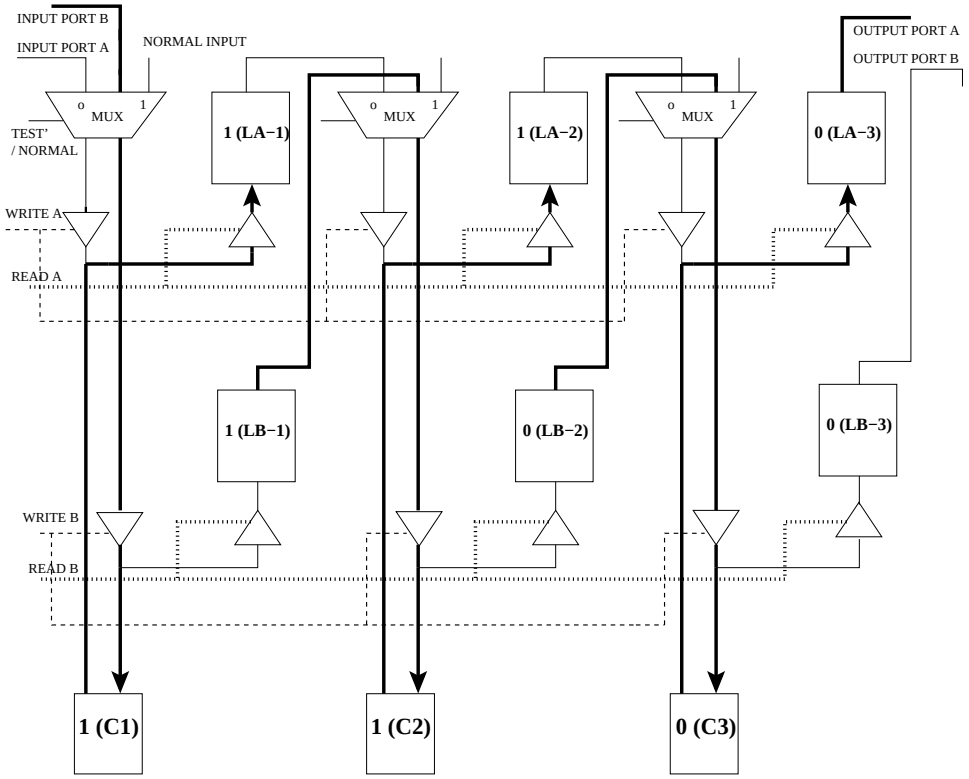

Figure 4.5: 2PF1 serial interfacing memory state (r0 : w1) - 2nd time

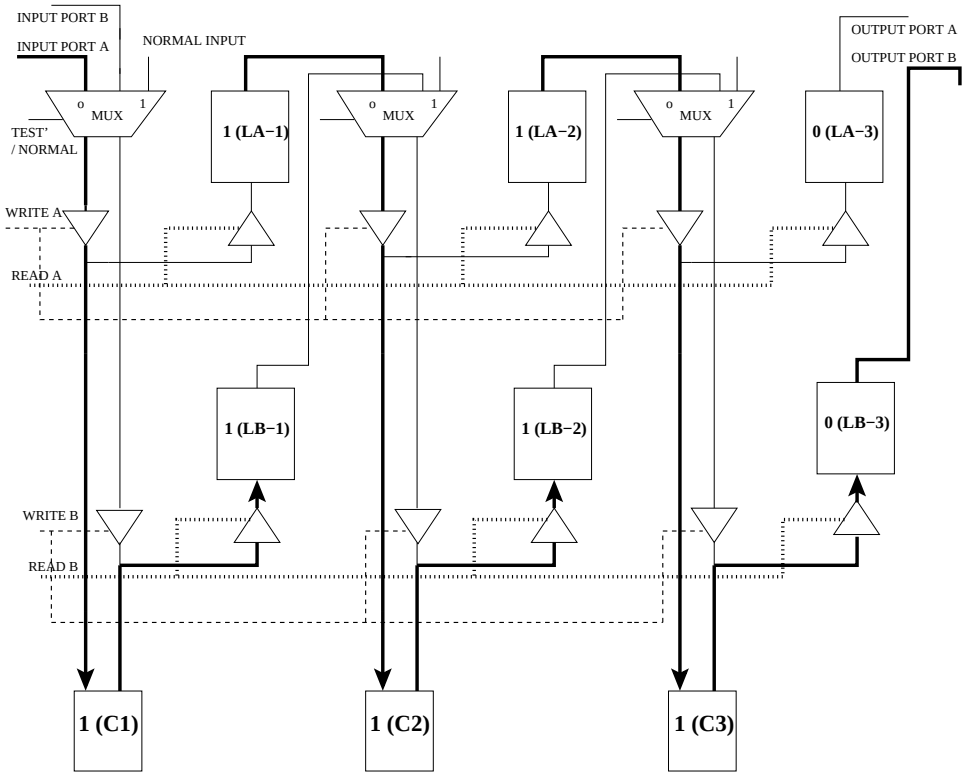

Figure 4.6: 2PF1 serial interfacing memory state (w1 : r0) - 3rd time

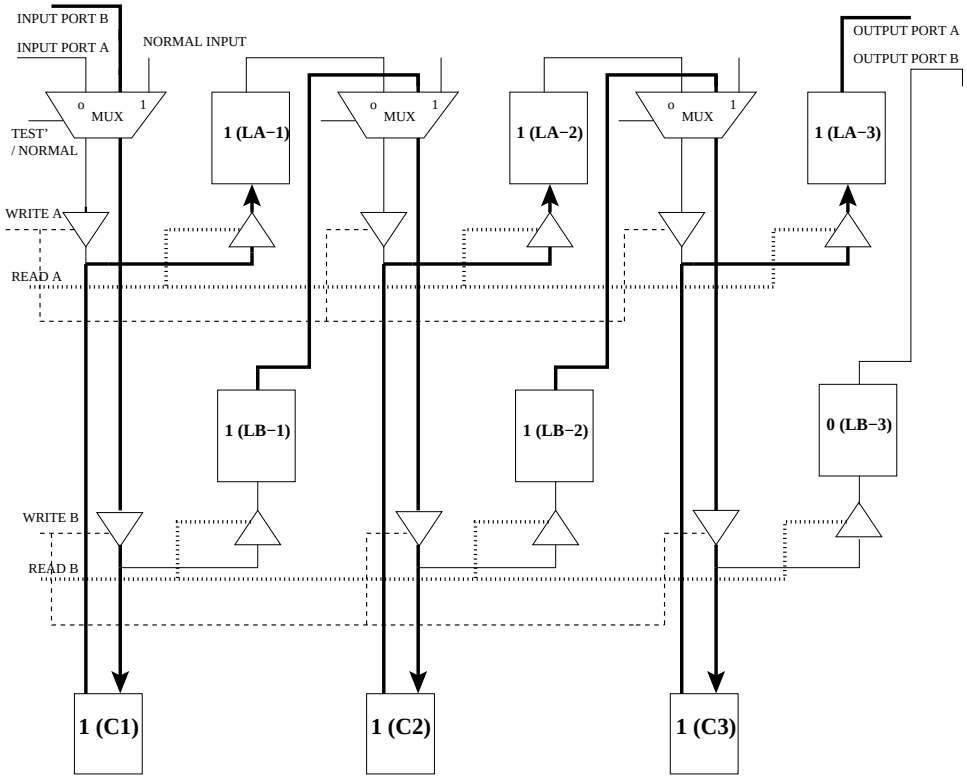

Figure 4.7: 2PF1 serial interfacing memory state (r0 : w1) - 3rd time

Chapter 5

Serial March Algorithm for Coupling Faults

In this chapter, we propose the idea of serial interfacing to sensitize and detect 2PF2 faults. We investigate the march algorithm to detect each 2PF fault involving two cells assuming a parallel architecture first, then transforming the same algorithm into a serial scheme as in Chapter 4.

FFM	**Fault Primitives**
wCFds & wCFds	$< w0 : rd; 0/\uparrow/\_ >, < w0 : rd; 1/\downarrow/\_ >$ $< w1 : rd; 0/\uparrow/\_ >, < w1 : rd; 1/\downarrow/\_ >$ $< rx : rx; 0/\uparrow/\_ >, < rx : rx; 1/\downarrow/\_ >$
wCFdr & wDRDF	$< 0; r0 : r0/\uparrow/0 >, < 0; r1 : r1/\downarrow/1 >$ $< 1; r0 : r0/\uparrow/0 >, < 1; r1 : r1/\downarrow/1 >$
wCFrd & wRDF	$< 0; r0 : r0/\uparrow/1 >, < 0; r1 : r1/\downarrow/0 >$ $< 1; r0 : r0/\uparrow/1 >, < 1; r1 : r1/\downarrow/0 >$
wCFds & wRDF	$< w0 : r0/\uparrow/1 >, < w0 : r1/\downarrow/0 >$ $< w1 : r0/\uparrow/1 >, < w1 : r1/\downarrow/0 >$
wCFds & wIRF	$< w0 : r0/0/1 >, < w0 : r1/1/0 >$ $< w1 : r0/0/1 >, < w1 : r1/1/0 >$
wCFdS & wRRF	$< w0 : r0/0/? >, < w0 : r1/1/? >$ $< w1 : r0/0/? >, < w1 : r1/1/? >$

Table 5.1: List of 2PF2s

The FFMs involved in 2PFs have been discussed in Table 3.1. Let us consider the FFMs for 2PF2s only as shown in Table 5.1. In the following section, we will discuss some basic conditions which need to be satisfied in marching algorithms to detect 2PF2s. We will investigate the conditions for all three types of 2PF2 faults, then derive the test condition for each of them. Then, we will present the parallel march algorithm to detect each type of 2PF2. Finally, we shall derive a serial scheme for the parallel march 2PF2 algorithm. Depending on which cells (to the a-cell and/or to the v-cell) the two simultaneous operations are applied, the 2PF2 class is divided into three types: 2PF2aa, 2PF2vv and 2PF2av as shown in Fig 5.1.

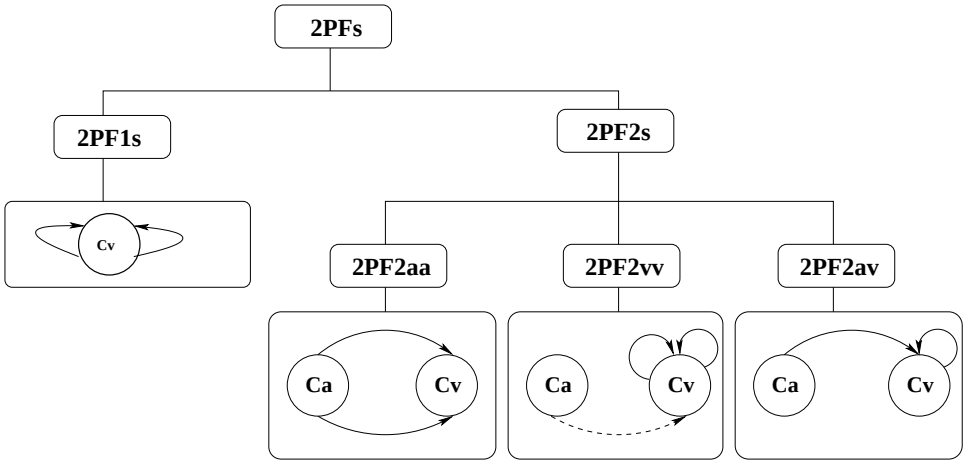

Figure 5.1: Classification of 2PF2

5.1 Conditions to Detect 2PF2aa

The 2PF2aa consists of one FFM, i.e., *wCFds & wCFds* with eight FPs as shown in Table 5.2 where x ϵ 0, 1 and d is don't care. In order to detect 2PF2aa, we have

1. to apply sensitizing operations to a cell Ca; a belongs to $0, 1, 2, ..., n-2, n-1$ and
2. to detect the fault in cell Cv; v not equal to a.

FFM	Fault Primitives
wCFds & wCFds	$< w0 : rd; 0/ \uparrow /\_ >, < w0 : rd; 1/ \downarrow /\_ >$ $< w1 : rd; 0/ \uparrow /\_ >, < w1 : rd; 1/ \downarrow /\_ >$ $< rx : rx; 0/ \uparrow /\_ >, < rx : rx; 1/ \downarrow /\_ >$

Table 5.2: List of 2PF2aas

The order in which Ca is selected is not important. Therefore, address order $\updownarrow_{a=0}^{n-1}$ can be specified. Any 2PF2aa is detectable by a march test which contains both march elements of Case A and both march elements of Case B.

- Case A:
  $\updownarrow_{a=0}^{n-1}$ (r0:r0 , ... , w1:rd , ... , r1:r1 , ... , w0:rd);
  $\updownarrow_{a=0}^{n-1}$ (r0:_ , ...)
- Case B:
  $\updownarrow_{a=0}^{n-1}$ (r1:r1 , ... , w0:rd , ... , r0:r0 , ... , w1:rd);
  $\updownarrow_{a=0}^{n-1}$ (r1:_ , ...)

The operations in the first march element of Case A (order is not important) will sensitize all 2PF2aa faults, when the fault effect is to drive the victim cell from logic 0 to logic 1, because the march element contains all sensitizing operations. If the march addressing order is increasing

and the v-cell has a higher (lower) address than the a-cell, then the faults in case A will be detected by the (r0:r0) ((r0:-)) operation of the first (second) march element in cass A. Similar explanation holds true for Case B, when the fault effect is to drive the victim cell from logic 1 to logic 0.

In the above condition, simultaneous read and write of the same location is assumed to be supported. If this is not the case, then the 2PF2aa will consist only of FPs sensitized by simultaneous read operations; as a consequence, the operations "wx:rd" in the condition should be replaced with "wx:n" where n denotes no operation and x ϵ 0,1.

5.2 Parallel March Algorithm 2PF2aa

Based on the conditions discussed above, the march algorithm for 2PF2aa can be presented as below.

{
$\Uparrow (w0 : n)$;
$\Uparrow (w1 : r0, r1 : r1, w0 : r1, r0 : r0, r0 : w1, r1 : w0)$;
$\Uparrow (r0 : n, w1 : n)$;
$\Uparrow (w0 : r1, r0 : r0, w1 : r0, r1 : r1, r1 : w0, r0 : w1)$;
$\Uparrow (r1 : n)$
}

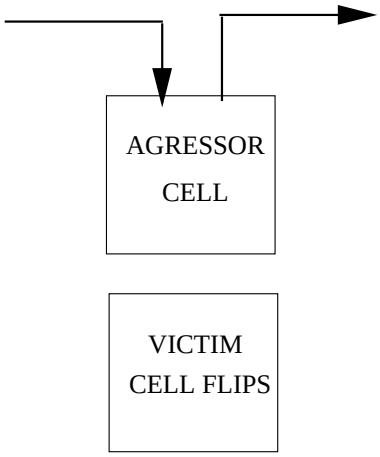

Figure 5.2: 2PF2aa aggressor-victim exhibit.

The 1st march element M0 writes 0 into all memory locations as shown in Fig. 5.3. When we proceed to M1, the 1st to 6th operations sensitize all wCFds & wCFds faults, if the background data is logic 1. If the march address order is increasing, the fault will be detected by the 1st operation of M1, when Address(v) > Address(a). But, the fault will be detected by the 1st operation of M2, when Address(a) > Address(v). We try to apply *(w1:r0)* for the 1st address as an example (Fig. 5.4). We see that faulty victim cells with address order higher than the aggressor cells flip from 0 to 1 as shown in Fig. 5.4. Thus, $< w1 : r0; 0/ \uparrow /\_ >$ is sensitized. We then sensitize the $< r1 : r1; 0/ \uparrow /\_ >$ fault by *(r1: r1)* . Next, we sensitize the $< w0 : r1; 0/ \uparrow /\_ >$ fault by *(w0: r1)* , and we find that another fault exists as indicated by one more "0" to "1" flip as shown in Fig. 5.5.

Similarly, $< r0 : r0; 0/ \uparrow /\_ >$, $< r0 : w1; 0/ \uparrow /\_ >$ and $< r1 : w0; 0/ \uparrow /\_ >$ faults are also sensitized by the 4th, 5th and 6th operations in M1. The corresponding victim cells will change

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 5.3: 2PF2aa parallel interfacing memory state after w0:n.

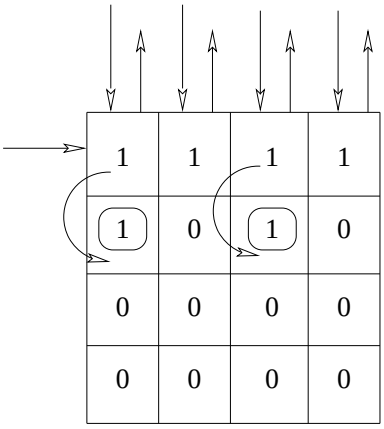

Figure 5.4: 2PF2aa parallel interfacing memory state after w1:r0.

to logic 1 if they are faulty. We now go to the next higher address and continue M1. When we proceed , faults $< w1 : r0; 0/\uparrow /\_ >$, $< r1 : r1; 0/\uparrow /\_ >$, $< w0 : r1; 0/\uparrow /\_ >$, $< r0 : r0; 0/\uparrow /\_ >$, $< r0 : w1; 0/\uparrow /\_ >$, and $< r1 : w0; 0/\uparrow /\_ >$ are all detected. Now, this takes care of the fact when the Address(v cell) > Address(a cell).

If Address(v cell) < Address(a cell), after all operations are over in M1 till the last address, we must read again to make sure that there are no more victims on top because of aggressors at the bottom. Hence the operation (r0:n) must be performed in M2 to detect the fault effect left by M1. Thus, we see that $< w1 : r0; 0/\uparrow /\_ >$, $< r1 : r1; 0/\uparrow /\_ >$, $< w0 : r1; 0/\uparrow /\_ >$, $< r0 : r0; 0/\uparrow /\_ >$, $< r0 : w1; 0/\uparrow /\_ >$, and $< r1 : w0; 0/\uparrow /\_ >$ are all sensitized and detected no matter whether the A(v) is > or < than A(a).

Finally, we rewrite the memory with 1's, and similar argument holds true for M3 and M4 for victim cell transitioning from 1 to 0. That is to sensitize and detect $< w0 : r1; 0/\uparrow /\_ >$, $< r0 : r0; 1/\downarrow /\_ >$, $< w1 : r0; 1/\downarrow /\_ >$, $< r1 : r1; 1/\downarrow /\_ >$, $< r1 : w0; 1/\downarrow /\_ >$, and $< r0 : w1; 1/\downarrow /\_ >$. Thus all 2pf2aa faults are sensitized and detected no matter whether the A(v) is > or < than A(a).

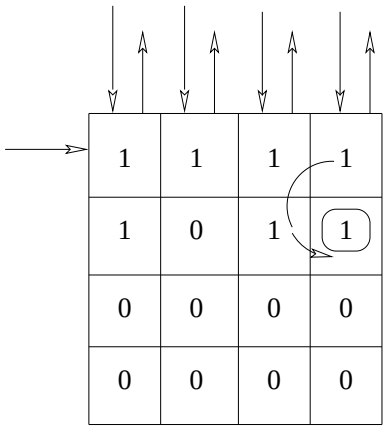

Figure 5.5: 2PF2aa parallel interfacing memory state after w0:r1.

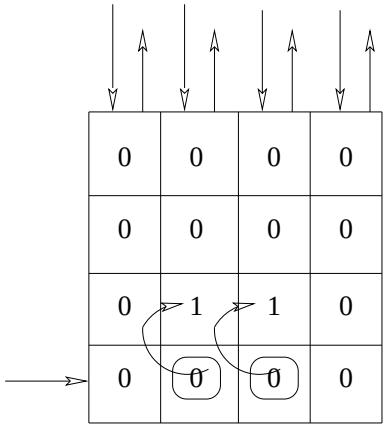

Figure 5.6: 2PF2aa illustration (i) - address(victim) < address(aggressor)

5.3 Conditions to Detect 2PF2vv

The 2PF2vv consists of two FFMs, i.e., *wCFdr & wDRDF* and *wCFrd & wRDF* with four FPs each as shown in Table 5.3.

FFM	**Fault Primitives**
wCFdr & wDRDF	$<0;r0:r0/\uparrow/0>,<0;r1:r1/\downarrow/1>$ $<1;r0:r0/\uparrow/0>,<1;r1:r1/\downarrow/1>$
wCFrd & wRDF	$<0;r0:r0/\uparrow/1>,<0;r1:r1/\downarrow/0>$ $<1;r0:r0/\uparrow/1>,<1;r1:r1/\downarrow/0>$

Table 5.3: List of 2PF2vvs

Any one of *wCFdr & wDRDF* and *wCFrd & wRDF* is detectable by a march test, if the test exercises all pairs of cells (Ca,Cv) whereby a ϵ 0,1,...,v-1,v+1,...,n-2,n-1 and each pair undergoes the four states 00,01,10 and 11. In addition, in each state, two simultaneous read operations followed by at least a single read operation have to be applied to the v-cell.

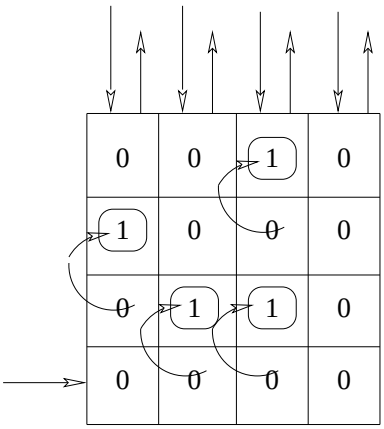

Figure 5.7: 2PF2aa illustration (ii) - address(victim) < address(aggressor)

5.4 Parallel March Algorithm 2PF2vv

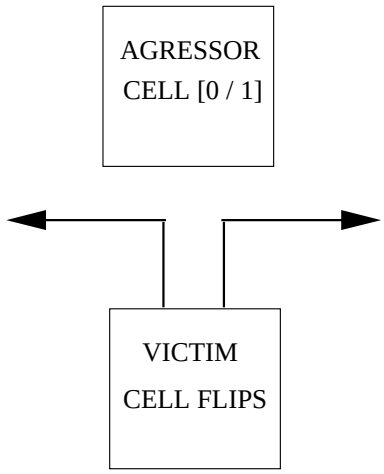

Figure 5.8: 2PF2vv aggressor-victim exhibit.

The Parallel march algorithm for 2PF2vv is:
{
$\Uparrow (w0 : n)$; $\Uparrow (r0 : r0, r0 : w1, r1 : r1, r1 : w0)$;
$\Uparrow (w1 : n)$; $\Uparrow (r1 : r1, r1 : w0, r0 : r0, r0 : w1)$;
}

Here, M0 initializes all memory cells to 0. M1 sensitizes and detects faults when the a-cell is in state - 0. For example, (r0 : r0) sensitizes and detects $< 0; r0 : r0/ \uparrow /1 >$; but only sensitizes $< 0; r0 : r0/ \uparrow /0 >$. Then, each cell is read again and logic 1 is written (simultaneously) to all cells of the word. This detects $< 0; r0 : r0/ \uparrow /0 >$. Operation (r1 : r1) sensitizes and detects $< 0; r1 : r1/ \downarrow /0 >$; but only sensitizes $< 0; r1 : r1/ \downarrow /1 >$. Then, each cell is read again and 0 is written (simultaneously) to all cells of the word. This detects $< 0; r1 : r1/ \downarrow /1 >$ and thereby keeps the state of the aggressor 0 for the next sequence.

Similarly, M3 can detect $< 1; r1 : r1/ \downarrow /0 >$, $< 1; r1 : r1/ \downarrow /1 >$, $< 1; r0 : r0/ \uparrow /1 >$, and $< 1; r0 : r0/ \uparrow /0 >$. Thus, the parallel algorithm for 2PF2aa and the parallel algorithm for

2PF2vv have been discussed so far.

5.5 Unified Parallel March Algorithm 2PF2aa-vv

Now let us create a unified parallel algorithm for 2PF2aa and 2PF2vv named as 2PF2aa-vv which is given by

$\{$
$\Uparrow (w0 : n)$;
$\Uparrow (r0 : r0, r0 : w1, r1 : r1, r1 : w0, w1 : r0, w0 : r1)$;
$\Uparrow (r0 : n, w1 : n)$;
$\Uparrow (r1 : r1, r1 : w0, r0 : r0, r0 : w1, w0 : r1, w1 : r0)$;
$\Uparrow (r1 : n)$
$\}$

It can be observed that the 2PF2aa-vv algorithm is exactly the same as the 2PF2aa algorithm except the fact that the order of elements in M1 and M3 are varied. On closer look, we can definitely see that this order will not have an impact on fault coverage and that all 2PF2aa faults will be successfully detected. Further, the 2PF2vv algorithm is a subset of the 2PF2aa-vv algorithm, and we can easily prove that all faults detected by 2PF2vv can also be detected by 2PF2aa-vv.

5.6 Serial March Algorithm 2PF2aa-vv

The serial equivalent implementation to sensitize and detect all 2PF2vv and 2PF2aa faults is:

$\{ \Uparrow \frac{M0}{((w0{:}rx)^3{:}n)}$;
$\Uparrow \frac{M1{-}1}{(r0{:}r0)}, \frac{M1{-}2}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}3}{(r1{:}r1)}, \frac{M1{-}4}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}5}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}6}{(w0{:}r1,r1{:}w0)^3}$;
$\Uparrow \frac{M2}{((r0{:}w1)^3{:}n)}$;
$\Uparrow \frac{M3{-}1}{(r1{:}r1)}, \frac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \frac{M3{-}3}{(r0{:}r0)}, \frac{M3{-}4}{(r0{:}w1,w1{:}r0)^3}, \frac{M3{-}5}{(w0{:}r1,r1{:}w0)^3}, \frac{M3{-}6}{(w1{:}r0,r0{:}w1)^3}$;
$\Uparrow \frac{M4}{((r1{:}w0)^3{:}n)}$ $\}$

Theorem 5.1. *All 2PF2aa faults can be detected by the serial 2PF2aa-vv algorithm.*
Proof : Without loss of generality, we use a 3-bit memory array with two ports as shown in Fig. 4.1. Assume bit C1 *of address 2* has the fault $< r0 : w1/0/ \uparrow /\_ >$ and bit C1 *of address 1* is the agressor cell which causes the fault. Let us apply the serial march operation M1 *to address 1* to sensitize the fault in address 2. By applying the first operation (i.e., r0:w1) of M1-2(1), C1 *of address 2* contains a faulty logic value 1. Thus, the fault has been sensitized. The rest of the M1 operations are performed in sequence, in address 1. These operations do not carry much importance to sensitize and detect this particular case we discuss here.

Now, when the march operation M1 is performed on address 2, the set of operations M1-1 and M1-2 will assist in detecting the faulty logic value 1. The detection of faults in other bits can be discovered similarly. We have looked at the case when Address(agressor)<Address(victim). If Address(agressor)>Address(victim), then the fault will be sensitized by march operation M1-2 as in the previous case, but detected by operation M2. **Q.E.D**

Finally, the detection of other 2PF2aa faults can be proved in the same manner.

Theorem 5.2. *All 2PF2vv faults can be detected by the serial 2PF2aa-vv algorithm.*

Proof : Assume bit C1 *of address 2* has fault $< 0; r0 : r0/ \uparrow /0 >$. After operation M0, all agressor cells (if any) take the logic value 0. By applying operation M1-1 (i.e., r0:r0) on address 2, C1 *of address 2* contains a faulty logic value 1. The faulty value will be latched into LA-1 by the first operation (i.e., r0:w1) of M1-2(1).

By applying the second operation of M1-2(1), the fault effect stored in LA-1 will be written into C2, and then propogated to LA-2 by the first operation of M1-2(2). Finally, the fault effect stored in LA-2 will be written to C3 by the second operation of M1-2(2), and then propogated to LA-3 by the first operation of M1-2(3). Thus, the fault effect can be observed by the scan output. **Q.E.D**

The detection of faults in other bits can be discovered similarly. Further, the detection of other 2PF2vv faults can be proved in the same manner.

Thus, we conclude from Theorems 5.1 and 5.2 that the serial march algorithm 2PF2aa-vv can sensitize and detect all 2PF2aa and all 2PF2vv faults. We use Table 5.4 to summarize our findings.

FFM	**Fault Primitives**	**Sensitized by**
wCFds & wCFds	$< w0 : r1; 0/ \uparrow /\_ >$	M1-6
	$< w0 : r1; 1/ \downarrow /\_ >$	M3-5
	$< w1 : r0; 0/ \uparrow /\_ >$	M1-5
	$< w1 : r0; 1/ \downarrow /\_ >$	M3-6
	$< r0 : r0; 0/ \uparrow /\_ >$	M1-1
	$< r0 : r0; 1/ \downarrow /\_ >$	M3-3
	$< r1 : r1; 0/ \uparrow /\_ >$	M1-3
	$< r1 : r1; 1/ \downarrow /\_ >$	M3-1
wCFdr & wDRDF	$< 0; r0 : r0/ \uparrow /0 >$	M1-1
	$< 0; r1 : r1/ \downarrow /1 >$	M1-3
	$< 1; r0 : r0/ \uparrow /0 >$	M3-3
	$< 1; r1 : r1/ \downarrow /1 >$	M3-1
wCFrd & wRDF	$< 0; r0 : r0/ \uparrow /0 >$	M1-1
	$< 0; r1 : r1/ \downarrow /1 >$	M1-3
	$< 1; r0 : r0/ \uparrow /0 >$	M3-3
	$< 1; r1 : r1/ \downarrow /1 >$	M3-1

Table 5.4: List of 2PF2aa-vvs

5.7 Detection of 2PF1 faults by SMarch 2PF2aa-vv Algorithm

We use Table 5.5 to show that SMarch 2PF2aa-vv can also sensitize and detect all 2PF1 faults.

5.8 Conditions to Detect 2PF2av

In order to detect the presence of such faults in cell Cv, we have to

1) Select all pairs (Ca,Cv) whereby a ϵ {0, 1, ...,v-1,v+1,...n-2,n-1},

2) Apply sensitizing operations to the two cells, and

FFM	Fault Primitives	Sensitized and Detected by
wDRDF & wDRDF	$< r0 : r0/\uparrow/0 >$	M1-1, M1-2
	$< r1 : r1/\downarrow/1 >$	M3-1, M3-2
wRDF & wRDF	$< r0 : r0/\uparrow/1 >$	M1-1, M1-2
	$< r1 : r1/\downarrow/0 >$	M3-1, M3-2
wRDF & wTF	$< r0 : w\uparrow/0/\_ >$	M1-2
	$< r1 : w\downarrow/1/\_ >$	M3-2

Table 5.5: 2PF1 detection by SMarch 2PF2aa-vv

3) read the cell Cv.

The order in which Cv is selected is not important if the march operations are well designed. The only requirement is that v has to take on all values from the set 0, 1, 2, ..., n-2, n-1. The order in which Ca is selected is not important either. The only requirement is that cell a has to take on all values from the set 0, 1, ..., v-1, v+1, ..., n-2, n-1. In the above, it is assumed that the a-cell and v-cell can be any cell of the memory array. However, this is not the case in real designs. In small buffers, we only consider the a and v cells to be in different words and to be more specific, in adjacent words. For example, the a-cell for address i, can only have v-cells in address i-1 or i+1. This reduction in a-cell and v-cell locations has a significant impact on the condition to detect the 2PF2av faults and therefore on the test. It reduces time complexity from $O(n^2)$ to O(n).

Any wCFds &wRDF and wCFds&wIRF is detectable by a march test, if the test contains all pairs of march elements of Case A, of Case B, of Case C, and of Case D. These four pairs of march elements can be combined into one, two, three, four, five, six, or seven march elements. In addition, a march test satisfying this condition can also probabilistically detect wCFds&wRRF. That means that the detection of this fault cannot be guaranteed due to the fact that the read operation produces a random value. Note that the letter "i" denotes the ith address while "n" denotes the total number of addresses in all further discussions.

1) Case A (to detect $< w1 : r1/\downarrow/0 >$ **and** $< w1 : r1/1/0 >$
$(\Uparrow_{i=0}^{n-1} (\ldots, w1_i : r1_{i+1}, \ldots))$;
$(\Uparrow_{i=0}^{n-1} (\ldots, r1_i : w1_{i+1}, \ldots))$;
2) Case B (to detect $< w1 : r0/\uparrow/1 >$ **and** $< w1 : r0/0/1 >$
$(\Uparrow_{i=0}^{n-1} (\ldots, w1_i : r0_{i+1}, \ldots))$;
$(\Uparrow_{i=0}^{n-1} (\ldots, r0_i : w1_{i+1}, \ldots))$;
3) Case C (to detect $< w0 : r1/\downarrow/0 >$ **and** $< w0 : r1/1/0 >$
$(\Uparrow_{i=0}^{n-1} (\ldots, w0_i : r1_{i+1}, \ldots))$;
$(\Uparrow_{i=0}^{n-1} (\ldots, r1_i : w0_{i+1}, \ldots))$;
4) Case D (to detect $< w0 : r0/\uparrow/1 >$ **and** $< w0 : r0/0/1 >$
$(\Uparrow_{i=0}^{n-1} (\ldots, w0_i : r0_{i+1}, \ldots))$;
$(\Uparrow_{i=0}^{n-1} (\ldots, r0_i : w0_{i+1}, \ldots))$;

The operation $(w1_i : r1_{i+1})$ in Case A (denoted as A.1) will sensitize and detect $< w1 : r1/\downarrow/0 >$, $< w1 : r1/1/0 >$ and may detect $< w1 : r1/1/? >$ in which the v-cell and a-cell are in adjacent words, and the address of a-cell is smaller than that of the v-cell. The operation $(r1_i : w1_{i+1})$ in Case A (denoted as A.2) will sensitize and detect the same faults in which the v-cell and a-cell are in adjacent words, and the address of a-cell is larger than that of the v-cell.

Both these operations are "necessary" for complete fault coverage. The same rule holds true for cases B, C, and D.

5.9 Parallel March Algorithm 2PF2av

The parallel march algorithm for 2PF2av shown below contains two march operations where M2 satisfies all conditioned mentioned above. The test length is $9n$ where n is the number of memory words.

$\Updownarrow (w0 : n);$
$\Uparrow_{i=0}^{n-1}$
(
$w1_i : r0_{i+1}$, $r1_i : w1_{i+1}$, $w0_i : r1_{i+1}$, $r0_i : w0_{i+1}$,
$w0_i : r0_{i+1}$, $r0_i : w1_{i+1}$, $w1_i : r1_{i+1}$, $r1_i : w0_{i+1}$
);

5.10 Serial March Algorithm 2PF2av

The architecture we use to test 2PF2av faults will slightly differ from our earlier architecture of testing 2PF1, 2PF2aa and 2PF2vv faults. In the earlier architecture, both ports were used to operate at the same address; but in this new architecture, each port will operate at a different address. The reason for this is because we are dealing with aggressor-victim faults at different addresses. Therefore, the serial implementation is:

$\Uparrow_{i=0}^{n-2} (w0, r0)_{addi}^{3} : (w0, r0)_{addi+1}^{3};$
$\Uparrow_{i=0}^{n-2}$
(
$(w0, r0)_{addi}^{3} : (r0, w0)_{addi+1}^{3},$
$(r0, w1)_{addi}^{3} : (w1, r0)_{addi+1}^{3},$
$(w1, r1)_{addi}^{3} : (r1, w1)_{addi+1}^{3},$
$(r1, w0)_{addi}^{3} : (w0, r1)_{addi+1}^{3}$
)

Let us now look into the details of using a single port for a single address. The architecture is shown in Fig. 5.9 for two ports being operated at two addresses i and i+1. Port A is currently operating on cells 1, 2 and 3 of address 1, while port B is operating on cells 4, 5 and 6 of address 2.

The previous logic can also be depicted in a simplified manner as in Fig. 5.10. Steps 1, 2 and 3 occur in consecutive fashion.

Step 1: Port A → Address 1 , Port B → Address 2
Step 2: Port A → Address 2 , Port B → Address 3
Step 3: Port A → Address 3 , Port B → Address 4

It can be easily verified that M0 is used to initialize all memory cells to logic 0. Now, we move to the next march operation which is

$\Uparrow_{i=0}^{n-2}$
(
$(w0, r0)_{addi}^{3} : (r0, w0)_{addi+1}^{3},$
$(r0, w1)_{addi}^{3} : (w1, r0)_{addi+1}^{3},$

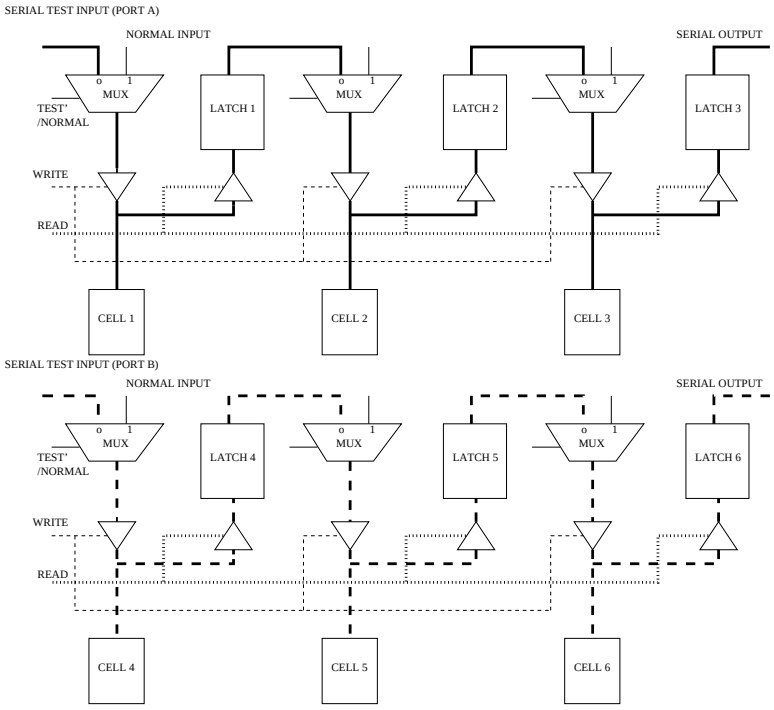

Figure 5.9: 2PF2av serial interfacing architecture

$(w1, r1)^3_{addi} : (r1, w1)^3_{addi+1}$,
$(r1, w0)^3_{addi} : (w0, r1)^3_{addi+1}$
)

First, let us look at the operation $(w0, r0)^3_{addi} : (r0, w0)^3_{addi+1}$.

On the i^{th} address and $(i+1)^{th}$ address, we perform a (w0,r0) and a (r0,w0) respectively for the first time. Here, port A (B) is assigned to address i (i+1) and the w0 (r0) operation is performed first to the left-hand (right-hand) side of the serial interface, and then r0 (w0) is performed to the right-hand (left-hand) side of the serial interface. Thus, it can be found that operations w0 and r0 are performed to ports A and B simultaneously. Then, operations r0 and w0 are performed to ports A and B simultaneously. This causes logic 0 to be written into the 1st cell and 1st latch of address i, and the 1st cell of address i+1 to be read and written correspondingly as shown in Fig. 5.11. We emphasize that the read/write operations are performed to all cells of both addresses. The same operations are repeated to both words two more times as shown in Figures 5.12 and 5.13. It can be easily proved that this march sub-element will detect $< w0 : r0/ \uparrow /1 >$ and $< w0 : r0/0/1 >$ defects. In fact, operation (w0:r0) to ports A and B respectively is used to sensitize this kind of faults when the victim cell has larger address than the aggresor cell. Similarly, operation (r0:w0) to ports A and B respectively is used to sensitize this kind of faults when the victim cell has smaller address than the aggresor cell. We illustrate similar findings in Table 5.6.

Theorem 5.3. *All wCFds & wRDF 2PF2av faults can be detected by the SMarch2PF2av algorithm.*
Proof : Without loss of generality, we use a 6-bit memory array with two ports as shown in Fig. 5.9. Assume bit C4 *of address 2* has the fault $< w0 : r0/ \uparrow /1 >$ and bit C1 *of address 1* is the agressor cell which causes the fault. Let us apply the march operation M1 *to addresses 1 and 2* to sensitize the fault in address 2. By applying the first half of M1-1(1) (i.e., $w0_{addi} : r0_{addi+1}$), C4 *of*

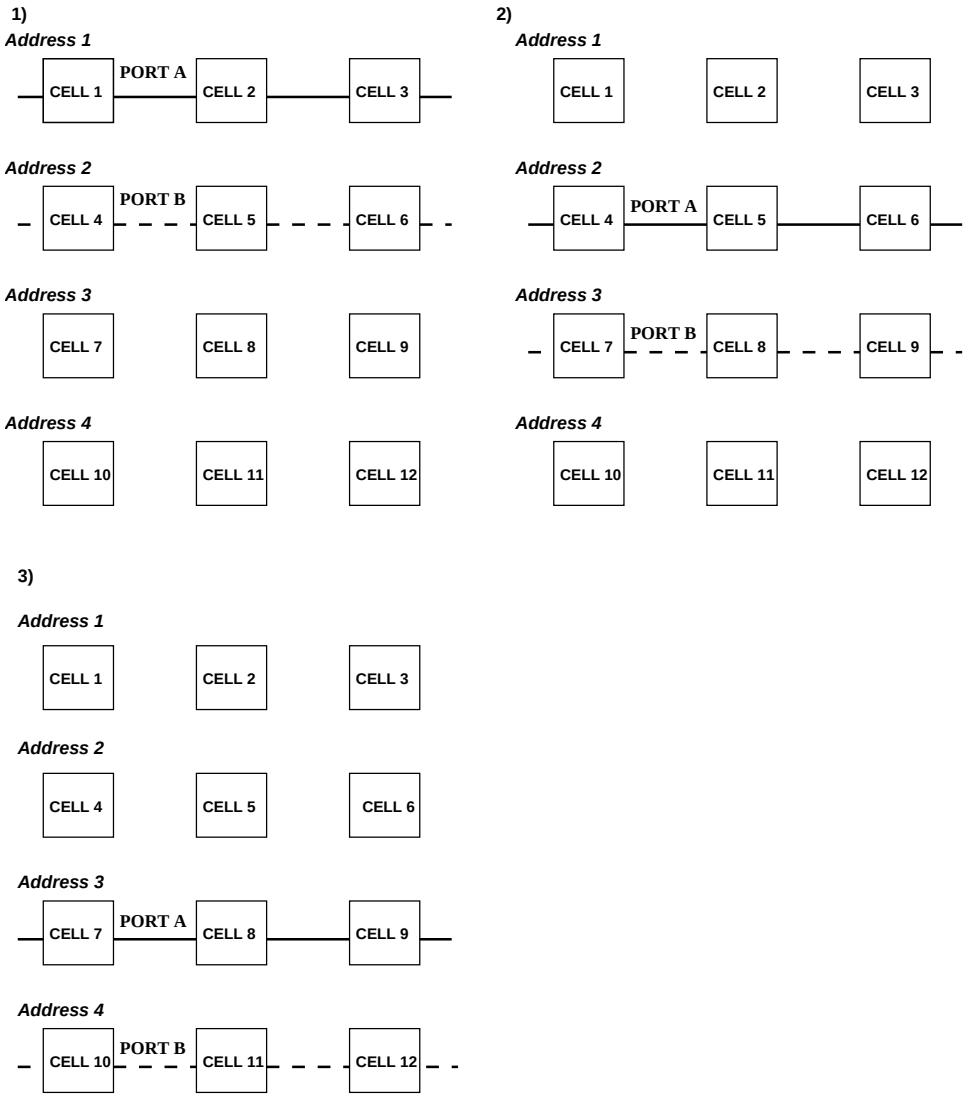

Figure 5.10: 2PF2av march order.

address 2 contains a faulty logic value 1 which is also written to latch L4. Thus, the fault has been sensitized.

Now, when the second half of the first operation (i.e., $r0_{addi} : w0_{addi+1}$) M1-1(1) is performed, the faulty logic value 1 in L4 is written to cell C5. Similarly, the complete operation of M1-1(2) succeeds in writing the faulty logic value 1 to cell C6. By performing the first half of operation M1-1(3), the faulty logic value 1 is available at latch L6 and ready to be read by the serial output. The detection of faults in other bits can be discussed similarly. We have looked at the case when Address(agressor)<Address(victim). If the Address(agressor)>Address(victim), then the fault will be sensitized by the second half of M1-1(1) (i.e., $r0_{addi} : w0_{addi+1}$) and detected by the second half of M1-1(3). **Q.E.D**

Finally, the detection of other wCFds *&* wRDF 2PF2av faults can be proved in the same manner.

Theorem 5.4. *All wCFds & wIRF 2PF2av faults can be detected by the SMarch2PF2av algorithm.*

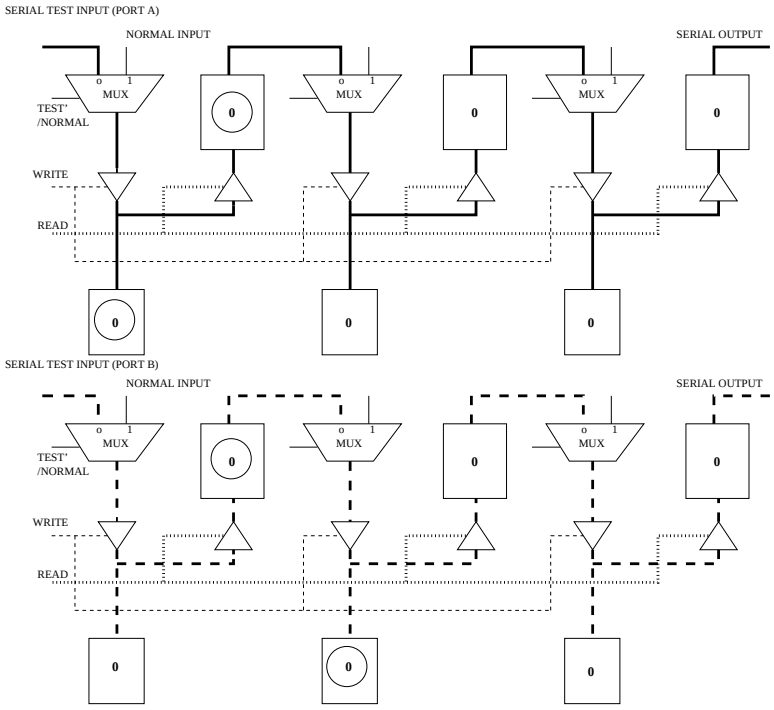

Figure 5.11: 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 1st time

Proof : The only difference between the wCFds *&* wRDF FP and the wCFds *&* wIRF FP is that, in the latter case, the victim cell is not flipped. In both cases, the output of the victim cell is the *incorrect value*. Hence, the sequence of sensitization and detection of the faults for the wCFds *&* wIRF FP, is the same as the sequence for the wCFds *&* wRDF FP explained in Theorem 5.3. **Q.E.D**

Thus, we conclude from Theorems 5.3 and 5.4 that the serial march algorithm SMarch2PF2av can sensitize and detect all deterministic 2PF2av faults.

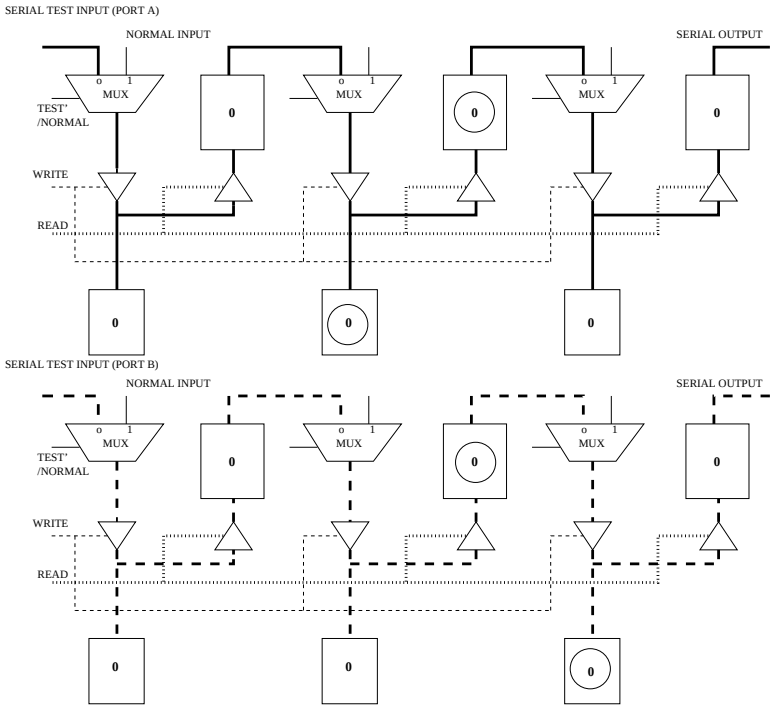

Figure 5.12: 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 2nd time

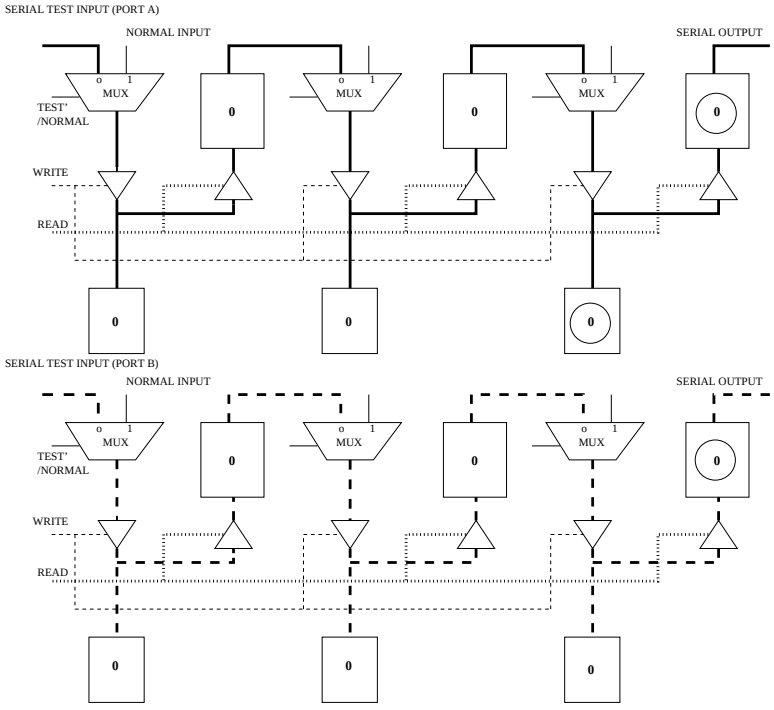

Figure 5.13: 2PF2av serial interfacing memory state $(w0, r0)_i : (r0, w0)_{i+1}$ - 3rd time

FFM	**Fault Primitives**	**Sensitized and Detected by**
wCFds *&* wRDF	$< w0 : r0/\uparrow/1 >, < r0 : w0/\uparrow/1 >$	M1-1
	$< w1 : r0/\uparrow/1 >, < r0 : w1/\uparrow/1 >$	M1-2
	$< w1 : r1/\downarrow/0 >, < r1 : w1/\downarrow/0 >$	M1-3
	$< w0 : r1/\downarrow/0 >, < r1 : w0/\downarrow/0 >$	M1-4
wCFds *&* wIRF	$< w0 : r0/0/1 >, < r0 : w0/0/1 >$	M1-1
	$< w1 : r0/0/1 >, < r0 : w1/0/1 >$	M1-2
	$< w1 : r1/1/0 >, < r1 : w1/1/0 >$	M1-3
	$< w0 : r1/1/0 >, < r1 : w0/1/0 >$	M1-4
wCFdS *&* wRRF	$< w0 : r0/0/? >, < r0 : w0/0/? >$	M1-1 (probablistically)
	$< w1 : r0/0/? >, < r0 : w1/0/? >$	M1-2 (probablistically)
	$< w1 : r1/1/? >, < r1 : w1/1/? >$	M1-3 (probablistically)
	$< w0 : r1/1/? >, < r1 : w0/1/? >$	M1-4 (probablistically)

Table 5.6: List of 2PF2avs - fault detection summary1

Chapter 6

Testing for Same-Word Faults

So far, we have discussed different types of two port fault models and ways to detect them, if they exist in different words (addresses). In this chapter, we shall prove that the march algorithm we proposed earlier also detect faults at the same word. Note that we only need to consider 2PF2aa and 2PF2vv faults, since 2PF2av faults cannot occur at the same word. We use the **SMarch 2PF2aa-vv** algorithm with slight modification to detect the same word faults.

Consider the serial march algorithm 2pf2aa-vv.

$$
\begin{array}{l}
\{ \\
\Uparrow \frac{M0}{((w0,rx)^3:n)} \ ; \\
\Uparrow \frac{M1{-}1}{(r0{:}r0)}, \frac{M1{-}2}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}3}{(r1{:}r1)}, \frac{M1{-}4}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}5}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}6}{(w0{:}r1,r1{:}w0)^3} \ ; \\
\Uparrow \frac{M2}{((r0,w1)^3:n)} \ ; \\
\Uparrow \frac{M3{-}1}{(r1{:}r1)}, \frac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \frac{M3{-}3}{(r0{:}r0)}, \frac{M3{-}4}{(r0{:}w1,w1{:}r0)^3}, \frac{M3{-}5}{(w0{:}r1,r1{:}w0)^3}, \frac{M3{-}6}{(w1{:}r0,r0{:}w1)^3} \ ; \\
\Uparrow \frac{M4}{((r1,w0)^3:n)} \\
\}
\end{array}
$$

We alter the algorithm to

$$
\begin{array}{l}
\{ \\
\Uparrow_L^R \frac{M0}{((w0,rx)^3:n)} \ ; \\
\Uparrow_L^R \frac{M1{-}1}{(r0{:}r0)}, \frac{M1{-}2}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}3}{(r1{:}w1,w1{:}r1)}, \frac{M1{-}4}{(r1{:}r1)}, \frac{M1{-}5}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}6}{(r0{:}w0,w0{:}r0)}, \\
\quad \frac{M1{-}7}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}8}{(w1{:}r1,r1{:}w1)}, \frac{M1{-}9}{(w0{:}r1,r1{:}w0)^3}, \frac{M1{-}10}{(w0{:}r0,r0{:}w0)} \ ; \\
\Uparrow_L^R \frac{M2}{((r0,w1)^3:n)} \ ; \\
\Uparrow_L^R \frac{M3{-}1}{(r1{:}r1)}, \frac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \frac{M3{-}3}{(r0{:}w0,w0{:}r0)}, \frac{M3{-}4}{(r0{:}r0)}, \frac{M3{-}5}{(r0{:}w1,w1{:}r0)^3}, \frac{M3{-}6}{(r1{:}w1,w1{:}r1)}, \\
\quad \frac{M3{-}7}{(w0{:}r1,r1{:}w0)^3}, \frac{M3{-}8}{(w0{:}r0,r0{:}w0)}, \frac{M3{-}9}{(w1{:}r0,r0{:}w1)^3}, \frac{M3{-}10}{(w1{:}r1,r1{:}w1)} \ ; \\
\Uparrow_L^R \frac{M4}{((r1,w0)^3:n)} \\
\}
\end{array}
$$

The reason why we add extra march operations (for example) will be explained later. Note that, by doing this, we **do not** endanger the detection of the 2PF2aa or 2PF2vv faults which were discussed in the previous chapter. We shall introduce faults to explain how our algorithm can successfully detect these FPs. In the following discussion, for the easiness of discussion, we show the detection of same-word faults by assuming a memory with 3 bits in each word. The general case can be easily extended.

6.1 Detection of Same-Word 2PF2aa Faults

Before we proceed, let us look at each case of 2PF2aa faults at the same word in the form of a table shown in Table. 6.1. We use the following two faults to illustrate how 2PF2aa-vv can sensitize and detect 2PF2aa faults in the same word.

1. $< r0 : r0; 0/ \uparrow /\_ >$,
2. $< r0 : w1; 0/ \uparrow /\_ >$.

Sensitizing operation on "a" cell	**Fault Effect on "v" cell**
r0:r0	$0 \uparrow 1$ or $1 \downarrow 0$
r1:r1	$0 \uparrow 1$ or $1 \downarrow 0$
r0:w1	$0 \uparrow 1$ or $1 \downarrow 0$
r1:w0	$0 \uparrow 1$ or $1 \downarrow 0$
w1:r0	$0 \uparrow 1$ or $1 \downarrow 0$
w0:r1	$0 \uparrow 1$ or $1 \downarrow 0$

Table 6.1: List of 2PF2 same-word faults.

Consider the memory array after all cells in the memory have logic 0 by applying M0 (Fig. 6.1). Next, we apply (r0:r0) to both ports simultaneously. On performing (r0:r0), we have a memory state as shown in Fig. 6.2 where "A" represents the agressor cell and "V" represents the victim cell. When we apply a (r0:r0) operation on cell C1 which is the agressor, cell C2 flips from 0 to 1. Now, this sensitizes the same-word fault $< r0 : r0; 0/ \uparrow /\_ >$.

After (r0:r0), according to our algorithm we apply $(r0 : w1, w1 : r0)^3, (r1 : w1, w1 : r1)$ serially. Note that in all the figures, each fault effect is marked with a circle. Let us notice the circle transgress across the memory from one side to another, where we will be reading it. On performing $(r0 : w1, w1 : r0)$ the first time, we get the memory states as shown in Fig. 6.3 and Fig. 6.4. Watch the circle (victim cell) being shifted by one memory position from left to right at the end of $(r0 : w1, w1 : r0)$ - one time. Now, let us apply $(r0 : w1, w1 : r0)$ the second time. On performing (r0:w1), we get the memory state as shown in Fig. 6.5. Thus, we see that the same-word fault $< r0 : r0; 0/ \uparrow /\_ >$ has been sensitized and detected. The operation $(r1 : w1, w1 : r1)$ is performed now for reasons which will be explained later. We have discussed the case when the address of the agressor is smaller than the address of victim. What happens if the aggressor address is greater than the victim address? We have a memory set-up as shown in Fig. 6.6. It is not difficult to understand that, even with Address(victim) < Address(agressor), the fault can also be detected in the similar fashion. Thus we conclude that the same-word fault $< r0 : r0; 0/ \uparrow /\_ >$ can be sensitized and detected regardless of the relative addresses of the victim and aggressor cells. This is also summarized in Table 6.2 (the 1st group of double-read faults).

Assume that a $< r0 : w1; 0/ \uparrow /\_ >$ fault exists in our memory array. After applying the (r0:r0) operation, we get the memory array and latches to be filled with zeros. On performing $(r0 : w1)$ the first time, we get the memory state as shown in Fig. 6.7. Note that the 1st cell C1 is the agressor and the 2nd cell C2 is the victim. When a (r0:w1) operation is applied to C1, it overwrites the cell C2 (which should contain a 0 at this stage) with a 1. Here, we assume that the victim cell is coupling-dominant, i.e., the coupling effect caused by the agressor on the victim is

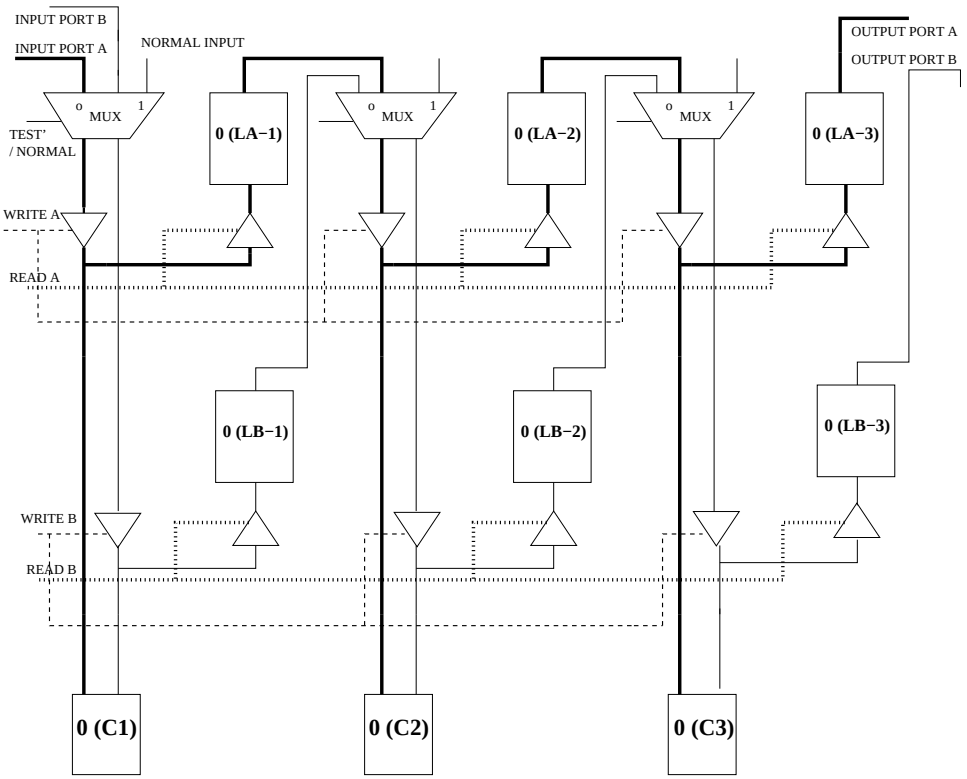

Figure 6.1: 2PF2aa-vv serial interfacing memory state after $(w0 : rx, rx, w1)^3$.

stronger than the write operation which is happening on the victim, at the same point of time. If the victim cell is write-dominant, however, then such a coupling fault will not occur. We continue the march operations as in our algorithm and reach (w1:r0) - second time.

After performing (w1:r0) the second time, this fault is detected as shown in Fig. 6.8. Thus, we observe that the same-word fault $< r0 : w1; 0/ \uparrow /\_ >$ can be sensitized and detected when Address(agressor) $<$ Address(victim). We can sensitize and detect other such same-word faults when Address(agressor) $<$ Address(victim) in the similar fashion. This covers some of the faults in the same word as shown in Table 6.2 (the 1st group of read/write faults).

Next, we shift our attention to the other undetected faults. Let us look at our serial marching algorithm once again.

$$
\begin{aligned}
&\{\\
&\Uparrow_L^R \ \frac{M0}{((w0,rx)^3:n)} \ ;\\
&\Uparrow_L^R \ \frac{M1{-}1}{(r0{:}r0)}, \frac{M1{-}2}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}3}{(r1{:}w1,w1{:}r1)}, \frac{M1{-}4}{(r1{:}r1)}, \frac{M1{-}5}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}6}{(r0{:}w0,w0{:}r0)},\\
&\quad \frac{M1{-}7}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}8}{(w1{:}r1,r1{:}w1)}, \frac{M1{-}9}{(w0{:}r1,r1{:}w0)^3}, \frac{M1{-}10}{(w0{:}r0,r0{:}w0)} \ ;\\
&\Uparrow_L^R \ \frac{M2}{((r0,w1)^3:n)} \ ;\\
&\Uparrow_L^R \ \frac{M3{-}1}{(r1{:}r1)}, \frac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \frac{M3{-}3}{(r0{:}w0,w0{:}r0)}, \frac{M3{-}4}{(r0{:}r0)}, \frac{M3{-}5}{(r0{:}w1,w1{:}r0)^3}, \frac{M3{-}6}{(r1{:}w1,w1{:}r1)},\\
&\quad \frac{M3{-}7}{(w0{:}r1,r1{:}w0)^3}, \frac{M3{-}8}{(w0{:}r0,r0{:}w0)}, \frac{M3{-}9}{(w1{:}r0,r0{:}w1)^3}, \frac{M3{-}10}{(w1{:}r1,r1{:}w1)} \ ;\\
&\Uparrow_L^R \ \frac{M4}{((r1,w0)^3:n)}\\
&\}
\end{aligned}
$$

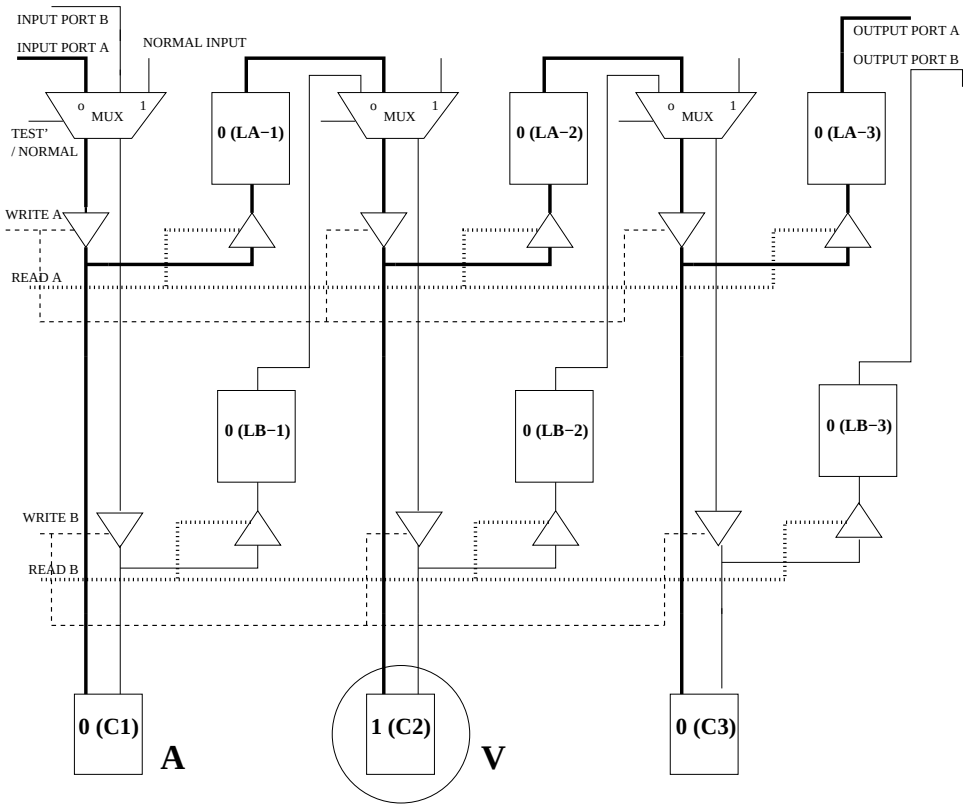

Figure 6.2: 2PF2aa-vv serial interfacing memory state after $(r0 : r0)$.

As we see clearly, after operation M0, the memory state is logic 0 in all cells. Assuming there is no $< r0 : r0; 0/ \uparrow /\_ >$ or $< r0 : r0; 1/ \downarrow /\_ >$ faults, we still have an all-zero memory state after applying the (r0:r0) operation in M1. The next operation we perform is $(r0 : w1, w1 : r0)^3, (r1 : w1, w1 : r1)$. This element starts writing logic 1 serially into the memory step by step as shown in Fig. 6.9. Consider step 3 where we have an agressor cell which happens to be the second cell, while the victim which happens to be the first cell. Thus, a (r0:w1) operation on the 2nd cell is able to check if a flip occurs from 1 down to 0 in the first cell. This precisely tests the $< r0 : w1; 1/ \downarrow /\_ >$ fault. Note that the agressor cell has a bitwise address location greater than the victim cell. In a similar fashion, step 4 tests the occurence of the same fault with C2 as the victim and C3 as the agressor, and so on. Remember that M1-2(1) produces (1 0 0) in the memory array. Operation (r0:w1) of M1-2(2) sensitizes the fault in C1 (victim) caused by C2 (aggressor). Operation (r0:w1) of M1-2(3) shifts the fault to C2, and (r1:w1) of M1-3 shifts the fault to C3. Operation (w1:r1) of M1-3 finally detects the fault. Therefore, we need an additional operation M1-3 for succesful detection.

Thus, the same word fault $< r0 : w1; 1/ \downarrow /\_ >$ can be sensitized and detected when Address(agressor) $\succ$ than Address(victim). We can prove in a similar fashion that faults $< r1 : w0; 0/ \uparrow /\_ >$, $< w1 : r0; 1/ \downarrow /\_ >$ and $< w0 : r1; 0/ \uparrow /\_ >$ can be sensitized and detected using elements 5, 6, 7, 8, 9 and 10 of M1 as shown in Table 6.2 (the 2nd group of read/write faults). Refer Fig. 6.10 to understand more.

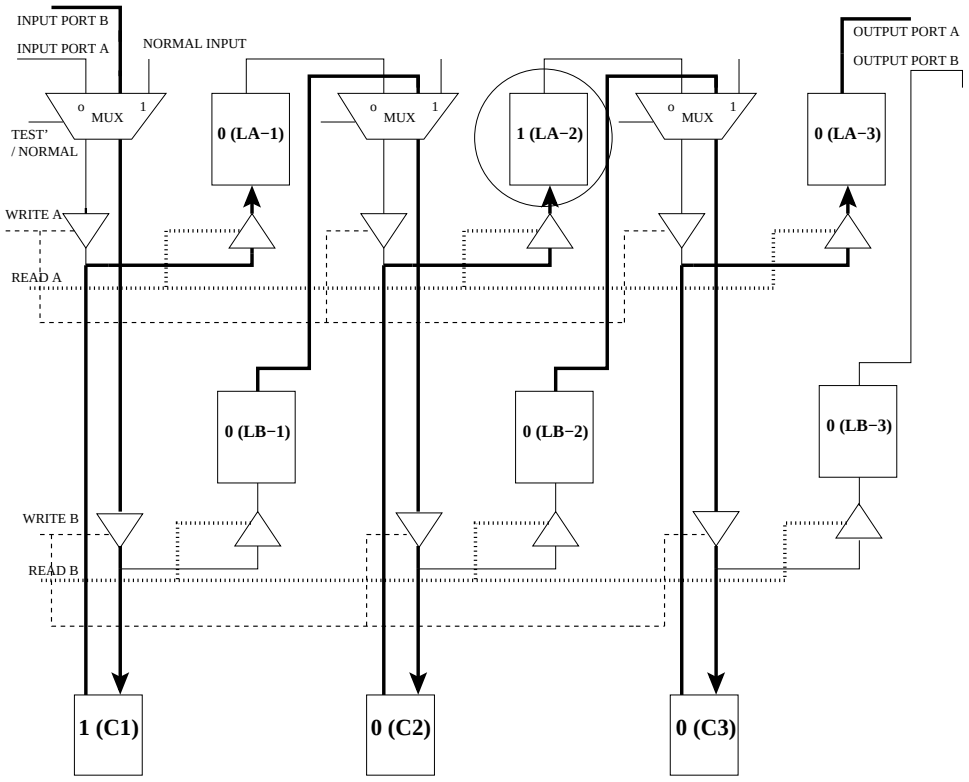

Figure 6.3: 2PF2aa-vv serial interfacing memory state (r0 : w1) - 1st time (i)

6.1.1 Bi-Directional Serial March and Address-Sensitive Fault Detection

As shown in Table 6.2, we still need to detect the following read/write faults.
$< r0 : w1; 0/ \uparrow /\_ >$
$< r1 : w0; 1/ \downarrow /\_ >$
$< w1 : r0; 0/ \uparrow /\_ >$
$< w0 : r1; 1/ \downarrow /\_ >$
when A>V and
$< r0 : w1; 1/ \downarrow /\_ >$
$< r1 : w0; 0/ \uparrow /\_ >$
$< w1 : r0; 1/ \downarrow /\_ >$
$< w0 : r1; 0/ \uparrow /\_ >$
when A<V

We introduced the concept of bi-directional serial interfacing in Chapter 2. Here, we use its power to detect the remaining faults. We know that march element M1 is symmetric to march element M3. If we make these two operations march in opposite directions, then our problem will be solved and the remaining read/write faults can be detected. This is easy to understand because our data background would be

step 1) 0 0 0
step 2) 1 0 0
step 3) 1 1 0
step 4) 1 1 1

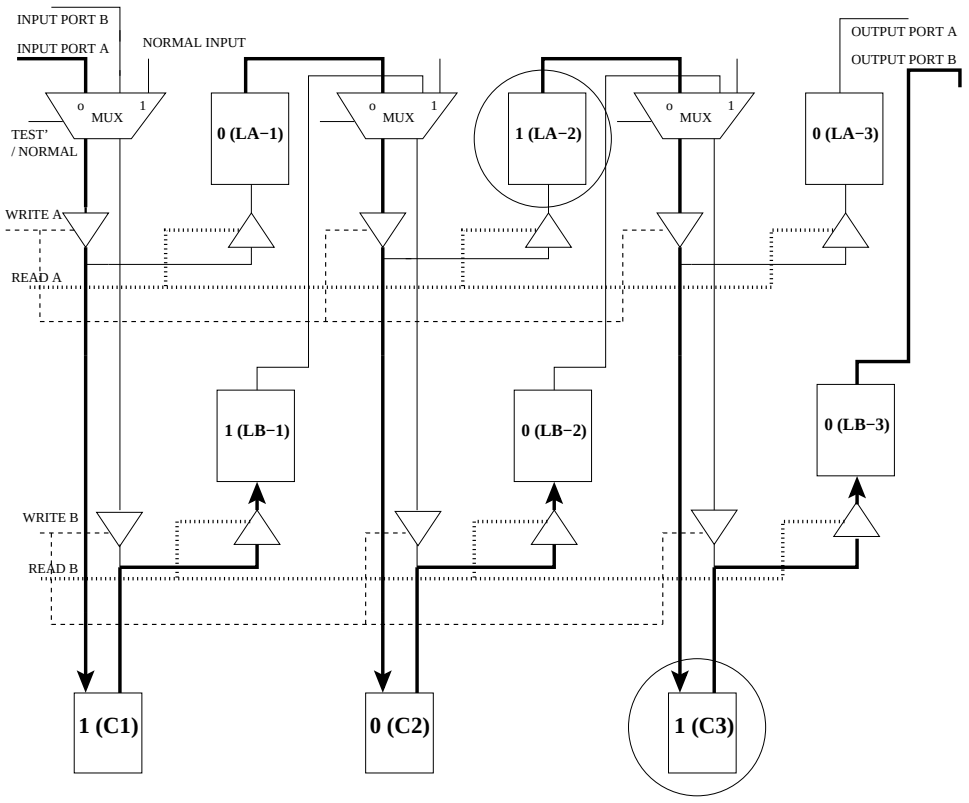

Figure 6.4: 2PF2aa-vv serial interfacing memory state (w1 : r0) - 1st time (i)

if the memory is populated from 0's to 1's by a serial input at the left side. However, with the serial input at the right side, it is populated in the following fashion.

step 1) 0 0 0
step 2) 0 0 1
step 3) 0 1 1
step 4) 1 1 1

The remaining read/write faults are detected by M3 as shown in Table 6.3. Thus, our serial march algorithm which can detect all same-word faults except $< r0 : r0; 1/ \downarrow /\_ >$ and $< r1 : r1; 0/ \uparrow /\_ >$ is

$$
\begin{aligned}
&\{\\
&\Uparrow_L^R \ \frac{M0}{((w0,rx)^3:n)} \ ;\\
&\Uparrow_L^R \ \frac{M1{-}1}{(r0{:}r0)}, \frac{M1{-}2}{(r0{:}w1,w1{:}r0)^3}, \frac{M1{-}3}{(r1{:}w1,w1{:}r1)}, \frac{M1{-}4}{(r1{:}r1)}, \frac{M1{-}5}{(r1{:}w0,w0{:}r1)^3}, \frac{M1{-}6}{(r0{:}w0,w0{:}r0)},\\
&\qquad \frac{M1{-}7}{(w1{:}r0,r0{:}w1)^3}, \frac{M1{-}8}{(w1{:}r1,r1{:}w1)}, \frac{M1{-}9}{(w0{:}r1,r1{:}w0)^3}, \frac{M1{-}10}{(w0{:}r0,r0{:}w0)} \ ;\\
&\Uparrow_L^R \ \frac{M2}{((r0,w1)^3:n)} \ ;\\
&\Uparrow_R^L \ \frac{M3{-}1}{(r1{:}r1)}, \frac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \frac{M3{-}3}{(r0{:}w0,w0{:}r0)}, \frac{M3{-}4}{(r0{:}r0)}, \frac{M3{-}5}{(r0{:}w1,w1{:}r0)^3}, \frac{M3{-}6}{(r1{:}w1,w1{:}r1)},\\
&\qquad \frac{M3{-}7}{(w0{:}r1,r1{:}w0)^3}, \frac{M3{-}8}{(w0{:}r0,r0{:}w0)}, \frac{M3{-}9}{(w1{:}r0,r0{:}w1)^3}, \frac{M3{-}10}{(w1{:}r1,r1{:}w1)} \ ;\\
&\Uparrow_L^R \ \frac{M4}{((r1,w0)^3:n)}\\
&\}
\end{aligned}
$$

Finally, we modify our serial march algorithm to detect the last two faults $< r0 : r0; 1/ \downarrow /\_ >$ and $< r1 : r1; 0/ \uparrow /\_ >$. We change the M2 and M4 operations to achieve the following

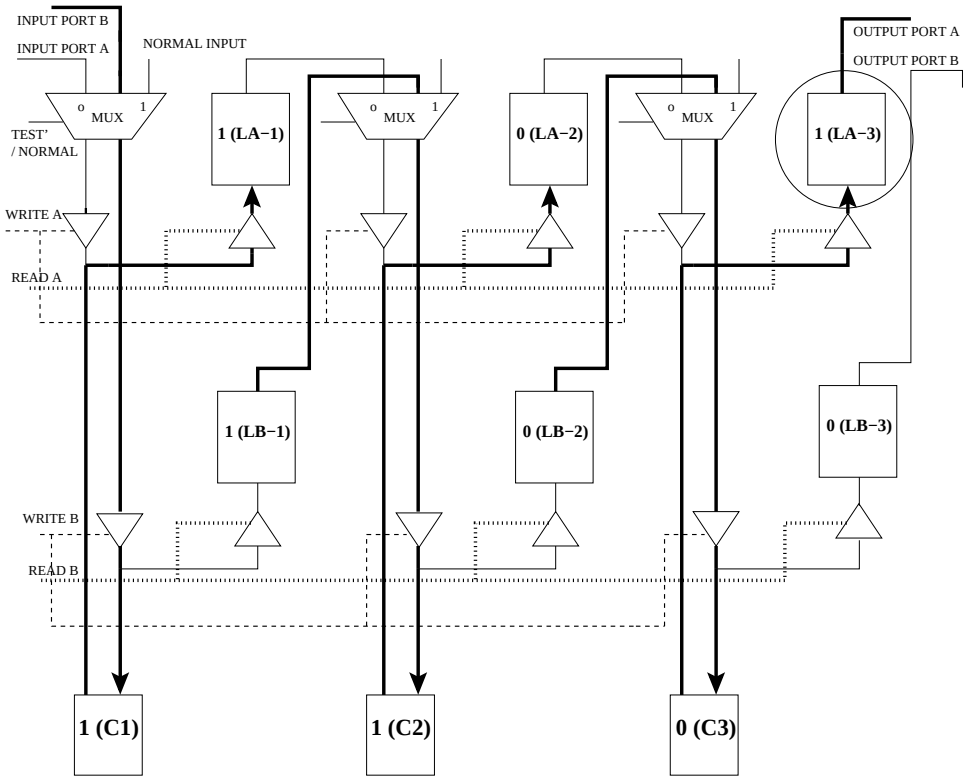

Figure 6.5: 2PF2aa-vv serial interfacing memory state (r0 : w1) - 2nd time (i)

algorithm.

$$
\begin{array}{l}
\{ \\
\Uparrow_{L}^{R}\ \frac{M0}{((w0,rx)^3:n)}\ ; \\
\Uparrow_{L}^{R}\ \frac{M1-1}{(r0:r0)}, \frac{M1-2}{(r0:w1,w1:r0)^3}, \frac{M1-3}{(r1:w1,w1:r1)}, \frac{M1-4}{(r1:r1)}, \frac{M1-5}{(r1:w0,w0:r1)^3}, \frac{M1-6}{(r0:w0,w0:r0)}, \\
\quad \frac{M1-7}{(w1:r0,r0:w1)^3}, \frac{M1-8}{(w1:r1,r1:w1)}, \frac{M1-9}{(w0:r1,r1:w0)^3}, \frac{M1-10}{(w0:r0,r0:w0)}\ ; \\
\Uparrow_{L}^{R}\ \frac{M2}{((r0,w1),(r0:r0))^3}\ ; \\
\Uparrow_{R}^{L}\ \frac{M3-1}{(r1:r1)}, \frac{M3-2}{(r1:w0,w0:r1)^3}, \frac{M3-3}{(r0:w0,w0:r0)}, \frac{M3-4}{(r0:r0)}, \frac{M3-5}{(r0:w1,w1:r0)^3}, \frac{M3-6}{(r1:w1,w1:r1)}, \\
\quad \frac{M3-7}{(w0:r1,r1:w0)^3}, \frac{M3-8}{(w0:r0,r0:w0)}, \frac{M3-9}{(w1:r0,r0:w1)^3}, \frac{M3-10}{(w1:r1,r1:w1)}\ ; \\
\Uparrow_{L}^{R}\ \frac{M4}{((r1,w0),(r1:r1))^3} \\
\}
\end{array}
$$

M2 first uses the (r0,w1) operation on port A to start injecting 1's in the memory array. Thus at the end of (r0,w1) of M2(1), we have the memory state as 1 0 0. Assume a $< r0 : r0; 1/ \downarrow /\_ >$ fault in cell C1. Now, performing a (r0:r0) operation to ports A and B sensitizes this fault when Address(A)>Address(V) where C1 is the victim and C2 is the agressor. Thus, C1 flips from 1 to 0. On performing (r0,w1) of M2(2) through port A, we have the memory state as 1 0 (faulty) 0. Note that we expected a 1 1 0 but obtained a 1 0 0. Thus, the fault has been transmitted from C1 to C2. Proceeding on the same lines, the $< r0 : r0; 1/ \downarrow /\_ >$ is detected when Address(A)>Address(V). Faults in other bits can be detected in a similar fashion. On close inspection, we can also understand that M2 can also detect all $< r1 : r1; 0/ \uparrow /\_ >$ faults where Address(A)<Address(V). M4 is exactly complementary to M2 and serves to detect $< r0 : r0; 1/ \downarrow /\_ >$ when Address(A)<Address(V), and $< r1 : r1; 0/ \uparrow /\_ >$ when Address(A)>Address(V). We illustrate the summary of faults detected

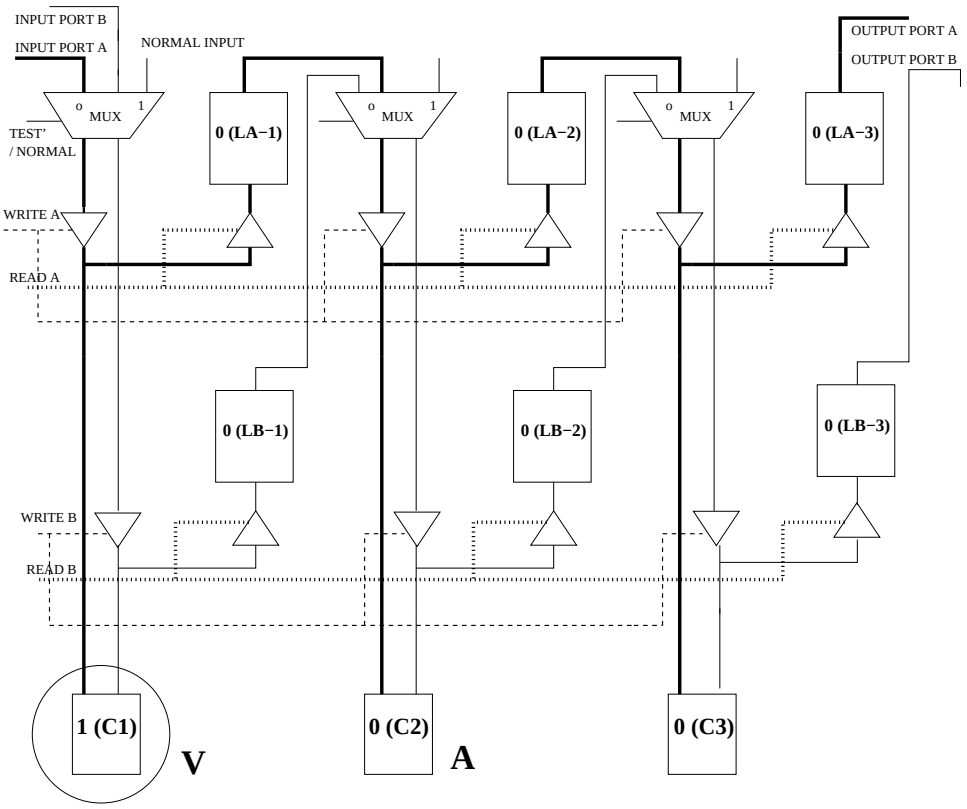

Figure 6.6: 2PF2aa-vv serial interfacing memory state after $(r0 : r0)$ with address(agressor)> address(victim).

and the operations which detect the fault in Table 6.3. Consequently, the modified SMarch2PF2aa-vv can detect all same-word 2PF2aa faults.

6.1.2 Fault Coverage Analysis for 2PF2aa Same-Word Faults

Theorem 6.1. *All same-word group I read/write faults (Table 6.3) can be detected by the SMarch2PF2aa-vv algorithm.*

Proof : Without loss of generality, we use a 3-bit memory array with two ports as shown in Fig. 4.1. Assume bit C2 has the fault $< r0 : w1; 0/ \uparrow /\_ >$ and bit C1 is the agressor cell which is causing the fault. Let us apply march element M1 to detect the fault. By applying the first operation (i.e., r0:r0) of M1-1, C1, LA-1 and LB-1 contain logic 0. The next operation is (r0:w1) of M1-2(1) which sensitizes the fault and causes C2 to flip from 0 to the faulty logic value 1, which is in turn transmitted from C2 to LB-2 by (w1:r0) of M1-2(1) Now, (r0:w1) of M1-2(2) is performed which writes the faulty value 1 from LB-2 to C3, which is transmitted to LB-3 by (w1:r0) of M1-2(2). Thus, the faulty logic value 1 is available at the serial output to be read. **Q.E.D**

Further, the detection of other same word faults namely

$< r1 : w0; 1/ \downarrow /\_ >$
$< w1 : r0; 0/ \uparrow /\_ >$
$< w0 : r1; 1/ \downarrow /\_ >$

can be proved in the same manner. Here, we proved the detection of such faults when Address(A) < Address(V). We can also prove that the detection of such faults holds true when Address(A) >

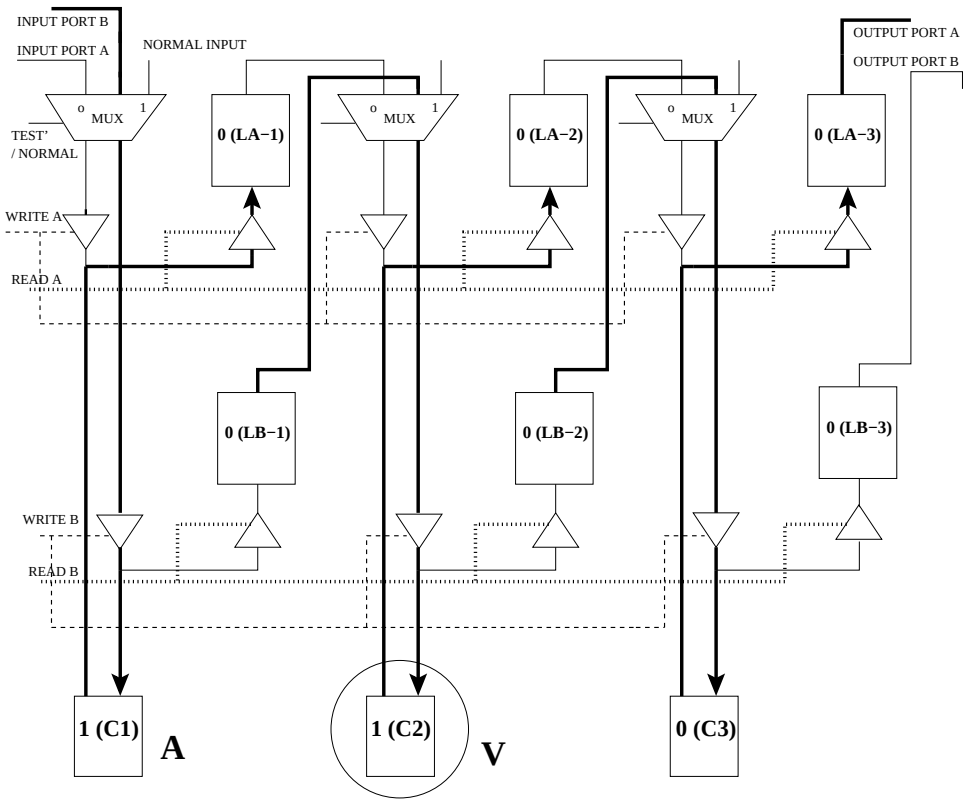

Figure 6.7: 2PF2aa-vv serial interfacing memory state (r0 : w1) - 1st time (ii)

Address(V) because we use bi-directional serial marching and march operation M3 can detect such faults in a similar way.

Theorem 6.2. *All same-word group II read/write faults (Table 6.3) can be detected by the SMarch2PF2aa-vv algorithm.*

Proof : Assume bit C1 has fault $< r0 : w1; 1/ \downarrow /\_ >$ and bit C2 is the agressor cell which causes the fault. By applying the first operation (i.e., r0:r0) of M1-1, C1, LA-1 and LB-1 contain a 0. The next operation is (r0:w1) of M1-2(1) which writes logic 1 into C1 and (w1:r0) of M1-2(1) transmits the 1 to LB-1. The next operation is (r0:w1) of M1-2(2) which basically reads a 0 from C2 and writes a 1 on C2. This sensitises the $< r0 : w1; 1/ \downarrow /\_ >$ fault present in C1 and flips the state of C1 from logic 1 to logic 0. The fault effect is in-turn transmitted by (w1:r0) of M1-2(2) from C1 to LB-1. On performing (r0:w1) of M1-2(3), the faulty 0 is written to cell C2 and transmitted to LB-2 by (w1:r0) of M1-2(3). Finally, (r1:w1) of M1-3 moves the faulty 0 to cell C3. The fault effect is in-turn transmitted to LB-3 by (w1:r1) of M1-3 where it can be read through the serial-out and detected. **Q.E.D**

Further, the detection of other same word faults namely

$< r1 : w0; 0/ \uparrow /\_ >$
$< w1 : r0; 1/ \downarrow /\_ >$
$< w0 : r1; 0/ \uparrow /\_ >$

can be proved in the same manner. Here, we proved the detection of such faults when Address(A) > Address(V). We can also prove that the detection of such faults holds true when Address(A) < Address(V). This is because we use bi-directional serial marching, and march element M3 can detect such faults in a similar way.

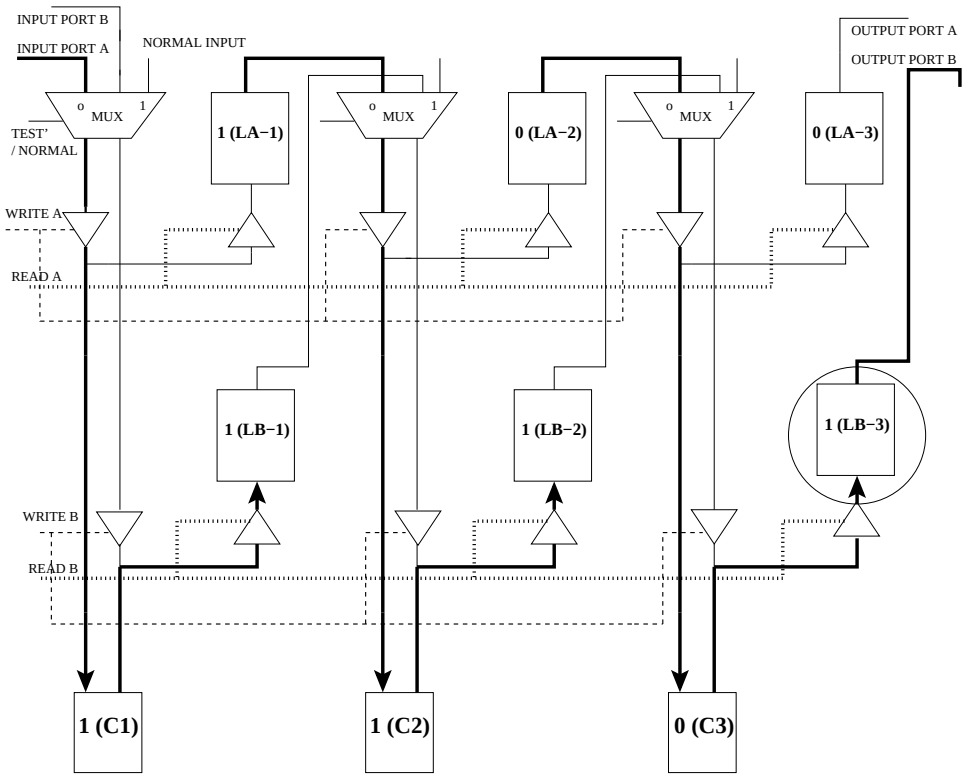

Figure 6.8: 2PF2aa-vv serial interfacing memory state (w1 : r0) - 2nd time (ii)

It is straightforward to trace that $< r0 : r0; 0/ \uparrow /\_ >$ can be sensitized by M1-1 and detected by M1-2. Similarly, $< r1 : r1; 1/ \downarrow /\_ >$ can be sensitized by M1-4 and detected by M1-5. Therefore, theorems for *group I double-read faults (Table 6.3)* are not discussed.

Theorem 6.3. *All same-word group II double-read faults (Table 6.3) can be detected by the SMarch2PF2aa-vv algorithm.*

Proof : Assume bit C1 has fault $< r0 : r0; 1/ \downarrow /\_ >$ and, bit C2 is the agressor cell which causes this fault. Let us apply march element M2 which is $[(r0, w1), (r0 : r0)]^3$ to sensitize the fault. This operation basically means that we apply a (r0,w1) operation through port A constantly, and at every step, we perform a double-read (ro:r0) through both ports. That is

step 1: (1 0 0) and perform (r0:r0)
step 2: (1 1 0) and perform (r0:r0)
step 3: (1 1 1) and perform (r0:r0)

Operation (r0,w1) through port A of M2(1) achieves writing a 1 to cell C1. Operation (r0:r0) of M2(1) activates the aggressor C2 to sensitize the fault $< r0 : r0; 1/ \downarrow /\_ >$ present in C1, and flips the state of C1 to a faulty logic value 0. Operation (r0,w1) of M2(2) achieves writing the faulty 0 to cell C2. Operation (r0:r0) of M2(2) is not of much importance at this point, since the fault has been activated. Operation (r0,w1) of M2(3) achieves writing the faulty 0 to cell C3. (r0:r0) of M2(3) reads the faulty 0 through the serial out, so the $< r0 : r0; 1/ \downarrow /\_ >$ fault in cell C1 is detected where Address(A)>Address(V). **Q.E.D**

Further, the detection of another same-word faults namely $< r1 : r1; 0/ \uparrow /\_ >$ where Address(A)<Address(V) can be proved in the same manner using M2. We can also prove that the detection of $< r0 : r0; 1/ \downarrow /\_ >$ [Address(A)<Address(V)] and $< r1 : r1; 0/ \uparrow /\_ >$

Fault Primitives	(A)<(V)	Operation	(A)>(V)	Operation
$< r0 : r0; 0/ \uparrow /\_ >$	X	M1-1, M1-2	X	M1-1, M1-2
$< r1 : r1; 1/ \downarrow /\_ >$	X	M1-4, M1-5	X	M1-4, M1-5
$< r0 : r0; 1/ \downarrow /\_ >$				
$< r1 : r1; 0/ \uparrow /\_ >$				
$< r0 : w1; 0/ \uparrow /\_ >$	X	M1-2		
$< r1 : w0; 1/ \downarrow /\_ >$	X	M1-5		
$< w1 : r0; 0/ \uparrow /\_ >$	X	M1-7		
$< w0 : r1; 1/ \downarrow /\_ >$	X	M1-9		
$< r0 : w1; 1/ \downarrow /\_ >$			X	M1-2, M1-3
$< r1 : w0; 0/ \uparrow /\_ >$			X	M1-5, M1-6
$< w1 : r0; 1/ \downarrow /\_ >$			X	M1-7, M1-8
$< w0 : r1; 0/ \uparrow /\_ >$			X	M1-9, M1-10

Table 6.2: Fault detection summary(i).

[Address(A)>Address(V)] holds true because march operation M4 can detect such faults in a similar way.

Thus, from Theorems 6.1, 6.2 and 6.3, we can conclude that all 2PF2aa same word faults can be detected by the modified SMarch2PF2aa-vv algorithm.

6.2 Detection of Same-Word 2PF2vv Faults

The 2PF2vv faults consist of two FFMs: *wCFrd & wRDF* with 4 FP's and *wCFdr & wDRDF* with 4 FP's. If the a-cell is in a certain state and two simultaneous reads are performed to the v-cell, then the v-cell flips with either correct or incorrect output.

1. *wCFrd & wRDF* (Group I)

   $< 0; r0 : r0/ \uparrow /1 > < 0; r1 : r1/ \downarrow /0 >$

   $< 1; r0 : r0/ \uparrow /1 > < 1; r1 : r1/ \downarrow /0 >$

2. *wCFdr & wDRDF* (Group II)

   $< 0; r0 : r0/ \uparrow /0 > < 0; r1 : r1/ \downarrow /1 >$

   $< 1; r0 : r0/ \uparrow /0 > < 1; r1 : r1/ \downarrow /1 >$

We shall give one theorem to prove the detection of *wCFdr & wDRDF*2PF2vv faults by SMarch 2PF2aa-vv. The detection of *wCFrd & wRDF is easier compared to the detection of* wCFdr & wDRDF, and can be easily proved in a similar manner.

6.2.1 Fault Coverage Analysis for 2PF2vv Same-Word Faults

Theorem 6.4. *All same-word group II double-read faults can be detected by the SMarch2PF2aa-vv algorithm.*

Fault Group	Fault Primitives	(A)<(V)	Operation	(A)>(V)	Operation
Group I R/R	$< r0 : r0; 0/ \uparrow /\_ >$	X	M1-1, M1-2	X	M1-1, M1-2
	$< r1 : r1; 1/ \downarrow /\_ >$	X	M1-4, M1-5	X	M1-4, M1-5
Group II R/R	$< r0 : r0; 1/ \downarrow /\_ >$	X	M4	X	M2
	$< r1 : r1; 0/ \uparrow /\_ >$	X	M2	X	M4
Group I R/W	$< r0 : w1; 0/ \uparrow /\_ >$	X	M1-2	X	M3-5
	$< r1 : w0; 1/ \downarrow /\_ >$	X	M1-5	X	M3-2
	$< w1 : r0; 0/ \uparrow /\_ >$	X	M1-7	X	M3-9
	$< w0 : r1; 1/ \downarrow /\_ >$	X	M1-9	X	M3-7
Group II R/W	$< r0 : w1; 1/ \downarrow /\_ >$	X	M3-5, M3-6	X	M1-2, M1-3
	$< r1 : w0; 0/ \uparrow /\_ >$	X	M3-2, M3-3	X	M1-5, M1-6
	$< w1 : r0; 1/ \downarrow /\_ >$	X	M3-9, M3-10	X	M1-7, M1-8
	$< w0 : r1; 0/ \uparrow /\_ >$	X	M3-7, M3-8	X	M1-9, M1-10

Table 6.3: Fault detection summary(ii).

Proof : Assume bit C1 has the fault $< 0; r0 : r0/ \uparrow /0 >$ and bit C2 is the agressor cell which causes the fault. Let us apply march sub-element M1-1 which is $(r0 : r0)$ to sensitize the fault. This operation flips C1 from logic state 0 to logic state 1, but the latches LA-1 and LB-1 still read logic 0. This is the deceptive read destructive fault. Thus, the faulty logic value resides in cell C1 now. The next operation (r0:w1) of M1-2(1) transmits the fault effect to LA-1, which is in-turn written to cell C2 by (w1:r0) of M1-2(1). Operation (r0:w1) of M1-2(2) reads the faulty logic 1 from cell C2 to LA-2, while (w1:r0) of M1-2(2) writes the fault effect to C3. Operation (r0:w1) of M1-2(3) reads the faulty 1 from cell C3 to LA-3, where it is available to be read at the serial output. We have proved the detection of $< 0; r0 : r0/ \uparrow /0 >$ when Address(A) > Address(V). In a similar way, we can use the same march operation M1-1 to sensitize the fault, and M1-2 to detect the fault even when Address(A) < Address(V). **Q.E.D**

Further, the detection of the other similar same-word fault namely $< 1; r1 : r1/ \downarrow /1 >$ can be proved in the same manner using operation M1-4 to sensitize and M1-5 to detect the fault.

Theorem 6.5. *All same-word group I double-read faults can be detected by the SMarch2PF2aa-vv algorithm.*
Proof : This theorem can be proved as in Theorem 6.4, and is thus omitted. **Q.E.D**

Thus, from theorems 6.4 and 6.5, we can conclude that all 2PF2vv same word faults can be detected by the modified SMarch2PF2aa-vv algorithm.

6.3 Consistency of the Final SMarch 2PF2aa-vv Algorithm

The final SMarch 2PF2aa-vv which can detect all same-word faults is

{
$\Uparrow_L^R \frac{M0}{((w0,rx)^3:n)}$;
$\Uparrow_L^R \frac{M1-1}{(r0:r0)}, \frac{M1-2}{(r0:w1,w1:r0)^3}, \frac{M1-3}{(r1:w1,w1:r1)}, \frac{M1-4}{(r1:r1)}, \frac{M1-5}{(r1:w0,w0:r1)^3}, \frac{M1-6}{(r0:w0,w0:r0)}$,

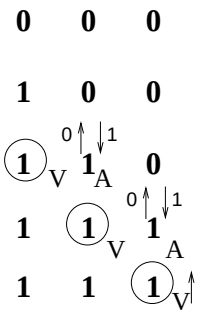

Figure 6.9: 2PF2aa-vv Serial Interfacing memory state (r0 : w1) (iii)

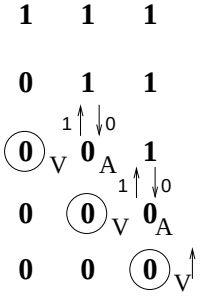

Figure 6.10: 2PF2aa-vv serial interfacing memory state (r1 : w0) (iii)

$$
\begin{aligned}
&\tfrac{M1{-}7}{(w1{:}r0,r0{:}w1)^3}, \tfrac{M1{-}8}{(w1{:}r1,r1{:}w1)}, \tfrac{M1{-}9}{(w0{:}r1,r1{:}w0)^3}, \tfrac{M1{-}10}{(w0{:}r0,r0{:}w0)}\ ;\\
\Uparrow_L^R\ &\tfrac{M2}{((r0,w1),(r0{:}r0))^3}\ ;\\
\Uparrow_R^L\ &\tfrac{M3{-}1}{(r1{:}r1)}, \tfrac{M3{-}2}{(r1{:}w0,w0{:}r1)^3}, \tfrac{M3{-}3}{(r0{:}w0,w0{:}r0)}, \tfrac{M3{-}4}{(r0{:}r0)}, \tfrac{M3{-}5}{(r0{:}w1,w1{:}r0)^3}, \tfrac{M3{-}6}{(r1{:}w1,w1{:}r1)},\\
&\tfrac{M3{-}7}{(w0{:}r1,r1{:}w0)^3}, \tfrac{M3{-}8}{(w0{:}r0,r0{:}w0)}, \tfrac{M3{-}9}{(w1{:}r0,r0{:}w1)^3}, \tfrac{M3{-}10}{(w1{:}r1,r1{:}w1)}\ ;\\
\Uparrow_L^R\ &\tfrac{M4}{((r1,w0),(r1{:}r1))^3}\\
\}&
\end{aligned}
$$

Due to the vast changes imposed on SMarch2PF2aa-vv from Chapter 5, it is possible that the final algorithm may not detect the *2PF2aa and 2PF2vv faults in different words*. To validate our algorithm, we use two theorems to prove that the modified SMarch 2PF2aa-vv can still sensitize and detect all 2PF2aa and 2PF2vv faults in different words.

Theorem 6.6. *All 2PF2aa faults in different words can be detected by the final serial 2PF2aa-vv algorithm.*

Proof : Without loss of generality, we use a 3-bit memory array with two ports as shown in Fig. 4.1. Assume bit C2 of address 2 has the fault $< r0 : w1/0/ \uparrow /\_ >$, and bit C1 of address 1 is the agressor cell which causes the fault. Let us apply the serial march operation M1 *to address 1* to sensitize the fault in address 2. By applying the first operation (i.e., r0:w1) of M1-2(1), C2 of address 2 contains a faulty logic value 1. Thus, the fault has been sensitized. The rest of the M1 operations are performed in sequence, in address 1. These operations do not carry much importance to sensitize and detect this particular case we discuss here.

Now, when march element M1 is performed on address 2, the set of operations M1-1 and M1-2 will assist in detecting the faulty logic value 1. The detection of faults in other bits can be discovered similarly. We have proved the case when Address(agressor)<Address(victim). If Address(agressor)>Address(victim), then the fault will be sensitized by march operation M1-2 as in the previous case, but detected by operation M2. **Q.E.D**

Finally, the detection of other 2PF2aa faults can be proved in the same manner.

Theorem 6.7. *All 2PF2vv faults in different words can be detected by the final serial 2PF2aa-vv algorithm.*
Proof : Assume bit C2 of address 2 has fault $< 0; r0 : r0/ \uparrow /0 >$. By applying operation M1-1 (i.e., r0:r0) on address 2, C2 of address 2 contains a faulty logic value 1. The faulty value will be latched into LA-1 by the first operation (i.e., r0:w1) of M1-2(1).

By applying the second operation of M1-2(1), the fault effect stored in LA-1 will be written into C2, and then propogated to LA-2 by the first operation of M1-2(2). Finally, the fault effect stored in LA-2 will be written to C3 by the second operation of M1-2(2), and then propogated to LA-3 by the first operation of M1-2(3). Thus, the fault effect can be observed by the scan output. The detection of faults in other bits can be discovered similarly. **Q.E.D**

Further, the detection of other 2PF2vv faults can be proved in the same manner.

Thus, we conclude from Theorems 6.6 and 6.7 that the serial march algorithm 2PF2aa-vv can still sensitize and detect all 2PF2aa and all 2PF2vv faults in different words. Finally, from Theorems 6.1-6.7, we conclude that all 2PF2aa, 2PF2vv faults (same-word and different words) can be detected by the SMarch2PF2aa-vv algorithm.

Chapter 7

Redundant Operations and Fault Coverage

7.1 BIST Architecture for Parallel Testing of Arrays with Multiple Ports

Most parallel memory test methods deal with single large memory array testing by virtually or physically partitioning a large array into several blocks that are then tested simultaneously [16][22][18][15]. Only few research works concentrate on testing separate memory arrays in parallel[24][4]. In [4] *daisy chain connection* is used to share the BIST control circuit among different memory modules. But test time is too high and all RAMs under test need to have the same number of words (not practical). In [24] each memory module receives separate control signals from the BIST controller which increases routing and design overhead. In this thesis, we develop an efficient BIST architecture that can test multiple dual-port memory modules with different sizes concurrently. Two algorithms namely *RSMarch 2PF2aa-vv and RSMarch 2PF2av* are derived correspondingly from *SMarch 2PF2aa-vv and SMarch 2PF2av* , which were discussed in the previous chapters. It is proposed that these algorithms tolerate some redundant (thus useless) march operations to obtain successful sharing of data and control signals (from the BIST controller to the buffers of different sizes). The advantages are

1. Low hardware overhead because of serial interfacing.
2. Small test time due to parallel testing of multiple memory modules.
3. High fault coverage since redundant operations do not mask fault detections.

Fig. 7.1 represents a BIST methodology to test three buffers simultaneously. The maximum word width (number) of all buffers is used to determine the word width (number) of the BIST controller. Thus, the large-size buffers dominate the entire test process, and all buffers receive the same number of input test patterns as well as generate the same number of output responses. That is, all buffers receive the same control and data signals from the BIST controller. Note that since we test dual-port memories, we have two serial interfaces for each buffer. Based on this concept, the small-size buffers might receive extra test patterns and generate redundant output responses. For example, if the first (second) buffer has word width and word number equal 5 and 3 (4 and 6), then the BIST controller will determine the word width as 5 and the word number as 6. Thus, the second buffer will have one bit test that is not really useful for each word testing (*horizontal*

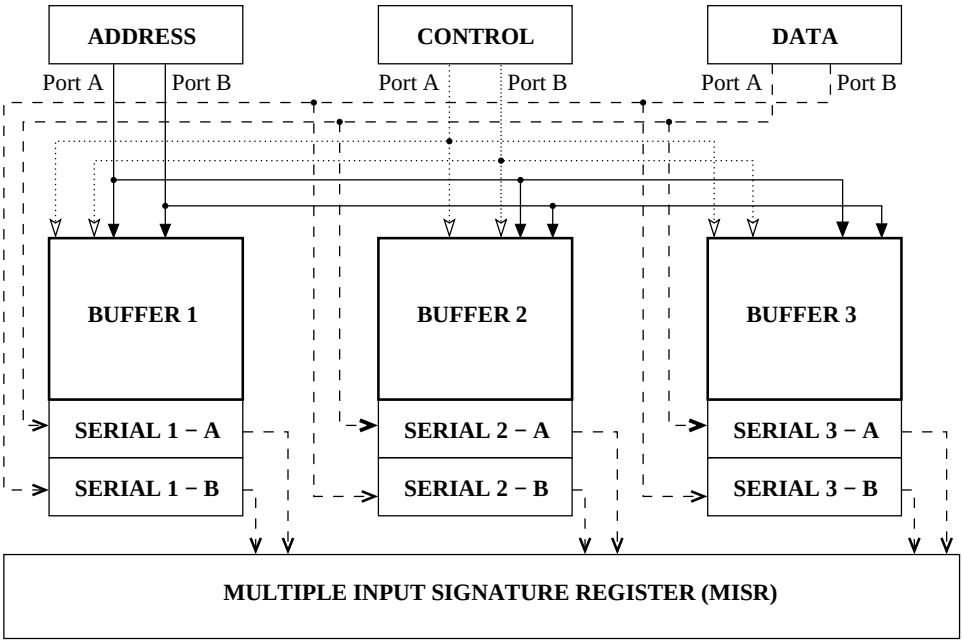

Figure 7.1: Bist architecture for multiple RAMs

redundancy), while all words of the first buffer will be tested twice for each march element (*vertical redundancy*). If these two types of over-testing do not result in any side-effect, then the hardware overhead can be minimized, the fault coverage can be maintained, and the goal of parallel testing can be successfully accomplished.

7.2 The RSMarch 2PF2aa-vv Algorithm

As discussed in the previous chapter, the SMarch2PF2aa-vv algorithm which can detect all 2PF1, 2PF2aa and 2PF2vv faults is

$$
\begin{aligned}
&\{\\
&\Uparrow^R_L \ \tfrac{M0}{((w0,rx)^3:n)}\ ;\\
&\Uparrow^R_L \ \tfrac{M1-1}{(r0:r0)}, \tfrac{M1-2}{(r0:w1,w1:r0)^3}, \tfrac{M1-3}{(r1:w1,w1:r1)}, \tfrac{M1-4}{(r1:r1)}, \tfrac{M1-5}{(r1:w0,w0:r1)^3}, \tfrac{M1-6}{(r0:w0,w0:r0)},\\
&\quad \tfrac{M1-7}{(w1:r0,r0:w1)^3}, \tfrac{M1-8}{(w1:r1,r1:w1)}, \tfrac{M1-9}{(w0:r1,r1:w0)^3}, \tfrac{M1-10}{(w0:r0,r0:w0)}\ ;\\
&\Uparrow^R_L \ \tfrac{M2}{((r0,w1),(r0:r0))^3}\ ;\\
&\Uparrow^L_R \ \tfrac{M3-1}{(r1:r1)}, \tfrac{M3-2}{(r1:w0,w0:r1)^3}, \tfrac{M3-3}{(r0:w0,w0:r0)}, \tfrac{M3-4}{(r0:r0)}, \tfrac{M3-5}{(r0:w1,w1:r0)^3}, \tfrac{M3-6}{(r1:w1,w1:r1)},\\
&\quad \tfrac{M3-7}{(w0:r1,r1:w0)^3}, \tfrac{M3-8}{(w0:r0,r0:w0)}, \tfrac{M3-9}{(w1:r0,r0:w1)^3}, \tfrac{M3-10}{(w1:r1,r1:w1)}\ ;\\
&\Uparrow^R_L \ \tfrac{M4}{((r1,w0),(r1:r1))^3}\\
&\}
\end{aligned}
$$

Let $\alpha =$

$$
\begin{aligned}
&((r0,r0):n)((r0:w1,w1:r0)^{c'})((r1:w1,w1:r1)^{c-c'})(r1:w1,w1:r1)\\
&((r1,r1):n)((r1:w0,w0:r1)^{c'})((r0:w0,w0:r0)^{c-c'})(r0:w0,w0:r0)\\
&((w1:r0,r0:w1)^{c'})((w1:r1,r1:w1)^{c-c'})(w1:r1,r1:w1)\\
&((w0:r1,r1:w0)^{c'})((w0:r0,r0:w0)^{c-c'})(w0:r0,r0:w0)
\end{aligned}
$$

and $\beta =$

$((r1, r1) : n)((r1 : w0, w0 : r1)^{c'})((r0 : w0, w0 : r0)^{c-c'})(r0 : w0, w0 : r0)$
$((r0, r0) : n)((r0 : w1, w1 : r0)^{c'})((r1 : w1, w1 : r1)^{c-c'})(r1 : w1, w1 : r1)$
$((w0 : r1, r1 : w0)^{c'})((w0 : r0, r0 : w0)^{c-c'})(w0 : r0, r0 : w0)$
$((w1 : r0, r0 : w1)^{c'})((w1 : r1, r1 : w1)^{c-c'})(w1 : r1, r1 : w1)$

The *RSMarch 2PF2aa-vv* algorithm, derived from the SMarch 2PF2aa-vv, which takes redundant operations into account is

M0:
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((w0, rx)^{c'} : n)((w0, r0)^{c-c'} : n)\};$
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((w0, r0)^{c} : n)\}^{[n/n']-1};$
$\{^{R}_{L} \Uparrow_{0}^{n \bmod n'-1} ((w0, r0)^{c} : n)\}$
M1:
$\{^{R}_{L} \Uparrow_{0}^{n'-1} \alpha\};$
$\{^{R}_{L} \Uparrow_{0}^{n'-1} \alpha\}^{[n/n']-1};$
$\{^{R}_{L} \Uparrow_{0}^{n \bmod n'-1} \alpha\};$
M2:
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((r0, w1), (r0 : r0))^{c'} ((r1, w1), (r1 : r1))^{c-c'}\};$
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((r1, w1), (r1 : r1))^{c}\}^{[n/n']-1};$
$\{^{R}_{L} \Uparrow_{0}^{n \bmod n'-1} ((r1, w1), (r1 : r1))^{c}\};$
M3:
$\{^{L}_{R} \Uparrow_{0}^{n'-1} \beta\};$
$\{^{L}_{R} \Uparrow_{0}^{n'-1} \beta\}^{[n/n']-1};$
$\{^{L}_{R} \Uparrow_{0}^{n \bmod n'-1} \beta\};$
M4:
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((r1, w0), (r1 : r1))^{c'} ((r0, w0), (r0 : r0))^{c-c'}\};$
$\{^{R}_{L} \Uparrow_{0}^{n'-1} ((r0, w0), (r0 : r0))^{c}\}^{[n/n']-1};$
$\{^{R}_{L} \Uparrow_{0}^{n \bmod n'-1} ((r0, w0), (r0 : r0))^{c}\};$

The structure of RSMarch 2PF2aa-vv is slightly different from SMarch 2PF2aa-vv in that redundant operations induced by smaller buffers are existent. In RSMarch 2PF2aa-vv, the first row of each march element gives the marching operations for the first-run scan of the entire memory array. It can be found that horizontally redundant operations (for example in M1(α): $(r1 : w1, w1 : r1)^{c-c'}$) might exist if buffer width c' is smaller than c. The second row of each march element gives the marching operations by which the entire memory array will be excessively scanned. For example, if BUF_i has word number n' equal 4 and there exists a buffer with maximum word number equal 10 (i.e., n=10), then the entire buffer array of BUF_i will be superfluously scanned one more time. Note that this test session might or might not be existent depending on the relationship between n and n'. Finally, the third row of each march element gives the marching operations which excessively scan part of the memory array when the march element finally terminates. Following the same example (i.e., n=10, n'=4), the first two words of BUF_i will be scanned three times when the entire march element is finished. Again, this test session may or may not exist depending on the relationship between n' and n. Next, we will show that RSMarch 2PF2aa-vv is able to accomplish the same fault coverage as SMarch 2PF2aa-vv, even in the presence of horizontal and vertical redundant operations.

7.2.1 Detection of 2PF1 Faults

Theorem 7.1. *All 2PF1 faults can be detected by the RSMarch2PF2aa-vv algorithm, if horizontal redundant operations are existent.*

Proof : Without loss of generality, we use $< w1 : r0/ \downarrow /\_ >$ to describe the scenario for fault detection with horizontal redundant operations, and the general case can be easily implied. Assume the word under test contains c' bits while the maximum word width of all arrays under test is c. If the word contains a $< w1 : r0/ \downarrow /\_ >$ fault, it will be detected by $(w1 : r0, r0 : w1)^{c'}$ operations of M1 and observed by the MISR. After non-redundant operations $(w1 : r0, r0 : w1)^{c'}$ are applied to the word, the content of the word is filled with logic 1. Thus, redundant operations $((w1 : r1, r1 : w1)^{c-c'})$ will not change the state of the word. In fact, only more logic 1 is fed into the MISR. Thus, the detection of $< w1 : r0/ \downarrow /\_ >$ fault is not affected at all. Further, the content of the word under test is not affected by the redundant operations either, and this gives the same condition of the word for the following march operations. **Q.E.D**

Theorem 7.2. *All 2PF1 faults can be detected by the RSMarch2PF2aa-vv algorithm, if vertical redundant operations are existent.*

Proof : Again, without loss of generality, we use $< w1 : r0/ \downarrow /\_ >$ to describe the scenario for fault detection with vertical redundant operations, while the general case can be easily extended. Assume the array under test contains n' words where n' is smaller than n. For each word containing the $< w1 : r0/ \downarrow /\_ >$ fault, the fault will definitely be detected $[n/\ n']$-1 extra times, and the fault effects will be fed into the MISR. Depending on n and n', the last vertical operation may or may not detect the fault again. If the address of the word containing $< w1 : r0/ \downarrow /\_ >$ fault is $\leq (n\ mod\ n') - 1$, then the fault will be additionally detected one more time. Thus, the fault effects are fed into the MISR more than once, and this will not affect the detection of the fault. After the vertical redundant operations terminate, the entire array contains logic 0 and this indicates that the vertically redundant operations will not change the content of the memory. This secures that the test conditions of the following march elements(if any) will not be changed. **Q.E.D**

7.2.2 Detection of 2PFaa Faults

Theorem 7.3. *All 2PF2aa faults can be detected by the RSMarch2PF2aa-vv algorithm, if horizontal redundant operations are existent.*

Proof : Without loss of generality, we use $< w1 : r0; 0/ \uparrow /\_ >$ to describe the scenario for fault detection with horizontal redundant operations, and the general case can be easily implied. Consider 2 words x and y in the memory array under test, where x holds the agressor cell and y holds the victim cell. Assume the word-width of the array under test (i.e width of x, y etc) is c' bits while the maximum word width of all arrays under test is c. Take a close look at α (of march operation M1) which is applied to each address location in increasing order.

$\alpha =$
$((r0, r0) : n)((r0 : w1, w1 : r0)^{c'})((r1 : w1, w1 : r1)^{c-c'})(r1 : w1, w1 : r1)$
$((r1, r1) : n)((r1 : w0, w0 : r1)^{c'})((r0 : w0, w0 : r0)^{c-c'})(r0 : w0, w0 : r0)$
$((w1 : r0, r0 : w1)^{c'})((w1 : r1, r1 : w1)^{c-c'})(w1 : r1, r1 : w1)$
$((w0 : r1, r1 : w0)^{c'})((w0 : r0, r0 : w0)^{c-c'})(w0 : r0, r0 : w0)$

Consider non-redundant march operations, i.e., $((w1 : r0, r0 : w1)^{c'})$, to be applied to x. This flips the faulty cell in y to logic 1 and sensitizes the fault. Now, consecutive operations, i.e., $((w1 : r1, r1 : w1)^{c-c'})(w1 : r1, r1 : w1)((w0 : r1, r1 : w0)^{c'})((w0 : r0, r0 : w0)^{c-c'})(w0 : r0, r0 : w0)$ are applied to x. However, this has no impact on the faulty cell which is present at y (no linked faults). Moreover, after the non-redundant operations $((w1 : r0, r0 : w1)^{c'})$ are applied to x, the

content of the word is filled with logic 1. Thus, the redundant operations $((w1 : r1, r1 : w1)^{c-c'})$ will not change the state of the word. In fact, only more logic 1 is fed into the MISR.

Now, the address increments and the faulty cell in y is detected by the non-redundant operations, i.e., $((r0, r0) : n)((r0 : w1, w1 : r0)^{c'})$, and observed by the MISR . Thus, the detection of $< w1 : r0/0/ \uparrow /\_ >$ fault is not affected by redundant operations at all. Further, the content of the word under test is not affected by the redundant operations either, and this gives the same condition of the word for the following march operations. **Q.E.D**

Theorem 7.4. *All 2PF2aa faults can be detected by the RSMarch2PF2aa-vv algorithm, if vertical redundant operations are existent.*
Proof : Without loss of generality, we use $< w1 : r0; 0/ \uparrow /\_ >$ to describe the scenario for fault detection with vertical redundant operations, and the general case can be easily implied. We had proved in Theorem 7.3 that the non-redundant operation $\{\Uparrow_0^{n'-1} \ \alpha \ \}$ can sensitize and detect a $< w1 : r0/0/ \uparrow /\_ >$ fault *one time* in each word containing the fault.

Assume the array under test contains n' words where n' is smaller than n. For each word containing the $< w1 : r0/ \downarrow /\_ >$ fault, the fault will definitely be detected $[n/n']$-1 extra times due to the redundant operations $\{\Uparrow_0^{n'-1} \ \alpha \ \}^{[n/n']-1}$, and the fault effects will be fed into the MISR. Depending on n and n', the last vertical operation $\{\Uparrow_0^{n \ mod \ n'-1} \ \alpha \ \}$ may or may not detect the fault again. If the address of the word containing $< w1 : r0/ \downarrow /\_ >$ fault is $\leq (n \ mod \ n') - 1$, then the fault will be additionally detected one more time. Thus, the fault effects are fed into the MISR more than once, and this will not affect the detection of the fault. After the vertical redundant operations terminate, the entire array contains logic 0 and this indicates that the vertically redundant operations will not change the content of the memory. This secures that the test conditions of the following march elements(if any) will not be changed. **Q.E.D**

7.2.3 Detection of 2PFvv Faults

Theorem 7.5. *All 2PF2vv faults can be detected by the RSMarch2PF2aa-vv algorithm, if horizontal redundant operations are existent.*
Proof : Without loss of generality, we use $< 0; r0 : r0/ \uparrow /1 >$ to describe the scenario for fault detection with horizontal redundant operations, and the general case can be easily implied. Assume the word under test contains c' bits, while the maximum word width of all arrays under test is c. Take a close look at α (of march operation M1) which is applied to each address location in increasing order. If the word contains a $< 0; r0 : r0/ \uparrow /1 >$ fault, it will be sensitized by non-redundant operation $((r0, r0) : n)$ and detected by non-redundant operation $((r0 : w1, w1 : r0)^{c'})$, and observed by the MISR . After non-redundant operations $((r0 : w1, w1 : r0)^{c'})$ are applied to the word, the content of the word is filled with logic 1. Thus, redundant operations $((r1 : w1, w1 : r1)^{c-c'})$ will not change the state of the word. In fact, only more logic 1 is fed into the MISR. Thus, the detection of $< 0; r0 : r0/ \uparrow /1 >$ fault is not affected by redundant operations at all. Further, the content of the word under test is not affected by the redundant operations either, and this gives the same condition of the word for the following march operations. **Q.E.D**

Theorem 7.6. *All 2PF2vv faults can be detected by the RSMarch2PF2aa-vv algorithm, if vertical redundant operations are existent.*
Proof : The proof for this theorem is similar to the proof of Theorem 7.4, and hence is not discussed in detail. **Q.E.D**

7.2.4 RSMarch 2PF2aa-vv Algorithm detects same-word faults

Theorem 7.7. *All same-word faults can be detected by the RSMarch2PF2aa-vv algorithm, if horizontal redundant operations are existent.*

Proof : Without loss of generality, we use $< r0 : r0; 0/ \uparrow /\_ >$ to describe the scenario for fault detection with horizontal redundant operations, and the general case can be easily implied. Assume the word under test contains c' bits while the maximum word width of all arrays under test is c. Take a close look at α (of march operation M1) which is applied to each address location in increasing order. Assume word x contains an agressor at cell a and a victim at cell v. If x contains a $< r0 : r0; 0/ \uparrow /\_ >$ fault, it will be sensitized by the non-redundant operation $((r0, r0) : n)$ applied to cell a thereby causing cell v to flip. This will be detected by non-redundant operations $((r0 : w1, w1 : r0)^{c'})$, and observed by the MISR. After the non-redundant operations $((r0 : w1, w1 : r0)^{c'})$ are applied to the word, the content of the word is filled with logic 1. Thus, redundant operations $((r1 : w1, w1 : r1)^{c-c'})$ will not change the state of the word. In fact, only more logic 1 is fed into the MISR. Thus, the detection of $< r0 : r0; 0/ \uparrow /\_ >$ fault is not affected at all. Further, the content of the word under test is not affected by the redundant operations either, and this gives the same condition of the word for the following march operations. **Q.E.D**

Theorem 7.8. *All same-word faults can be detected by the RSMarch2PF2aa-vv algorithm, if vertical redundant operations are existent.*

Proof : The proof for this theorem is similar to the proof of Theorem 7.4 and hence is not discussed in detail. **Q.E.D**

7.3 The RSMarch 2PF2av Algorithm

We know that the SMarch 2PF2av algorithm which can detect all 2PF2av faults is

$\Uparrow_{i=0}^{n-2} (w0, rx)^3_{addi} : (w0, rx)^3_{addi+1}$;
$\Uparrow_{i=0}^{n-2}$
(
$(w0, r0)^3_{addi} : (r0, w0)^3_{addi+1}$,
$(r0, w1)^3_{addi} : (w1, r0)^3_{addi+1}$,
$(w1, r1)^3_{addi} : (r1, w1)^3_{addi+1}$,
$(r1, w0)^3_{addi} : (w0, r1)^3_{addi+1}$
)

Let $\gamma =$

$((w0, r0)^{c'}{}_{addi} : (r0, w0)^{c'}{}_{addi+1})((w0, r0)^{c-c'}{}_{addi} : (r0, w0)^{c-c'}{}_{addi+1})$
$((r0, w1)^{c'}{}_{addi} : (w1, r0)^{c'}{}_{addi+1})((r1, w1)^{c-c'}{}_{addi} : (w1, r1)^{c-c'}{}_{addi+1})$
$((w1, r1)^{c'}{}_{addi} : (r1, w1)^{c'}{}_{addi+1})((w1, r1)^{c-c'}{}_{addi} : (r1, w1)^{c-c'}{}_{addi+1})$
$((r1, w0)^{c'}{}_{addi} : (w0, r1)^{c'}{}_{addi+1})((r0, w0)^{c-c'}{}_{addi} : (w0, r0)^{c-c'}{}_{addi+1})$

The *RSMarch 2PF2av* , derived from the SMarch 2PF2av algorithm, which takes redundant operations into account is

{
M0:
$\{\Uparrow_{i=0}^{n'-2} ((w0, rx)^{c'}{}_{addi} : (w0, rx)^{c'}{}_{addi+1})((w0, r0)^{c-c'}{}_{addi} : (w0, r0)^{c-c'}{}_{addi+1})\}$;
$\{\Uparrow_{i=0}^{n'-2} ((w0, r0)^{c}{}_{addi} : (w0, r0)^{c}{}_{addi+1})\}^{[n/n']-1}$;
$\{\Uparrow_{i=0}^{n \ mod \ n'-2} ((w0, r0)^{c}{}_{addi} : (w0, r0)^{c}{}_{addi+1})\}$

M1:
$\{\Uparrow_{i=0}^{n'-2}\ \gamma\ \}$;
$\{\Uparrow_{i=0}^{n'-2}\ \gamma\ \}^{[n/n']-1}$;
$\{\Uparrow_{i=0}^{n\ mod\ n'-2}\ \gamma\ \}$
}

7.3.1 Detection of 2PF2av Faults

Theorem 7.9. *All 2PF2av faults can be detected by the RSMarch 2PF2av algorithm, if horizontal redundant operations are existent.*

Proof : Without loss of generality, we use $< w1 : r0/ \uparrow /1 >$ to describe the scenario for fault detection with horizontal redundant operations, and the general case can be easily implied. Assume the victim word under test(y) contains c' bits while the maximum word width of all arrays under test is c. If the word contains a $< w1 : r0/ \uparrow /1 >$ fault (i.e. write 1 to word x and read 0 for y causing the victim cell in y to flip), it will be detected by $((r0, w1)^{c'}{}_y : (w1, r0)^{c'}{}_x)$ non-redundant operations and observed by the MISR. After non-redundant operations $((r0, w1)^{c'}{}_y : (w1, r0)^{c'}{}_x)$ are applied to the word, the content of the word is filled with logic 1. Thus, redundant operations $((r1, w1)^{c-c'}{}_y : (w1, r1)^{c-c'}{}_x)$ will not change the state of the word. In fact, only more logic 1 is fed into the MISR. Thus, the detection of $< w1 : r0/ \uparrow /1 >$ fault is not affected at all. Further, the content of the word under test is not affected by the redundant operations either, and this gives the same condition of the word for the following march operations. **Q.E.D**

Theorem 7.10. *All 2PF2av faults can be detected by the RSMarch 2PF2av algorithm, if vertical redundant operations are existent.*

Proof : The proof for this theorem is similar to the proof of Theorem 7.4, and hence is not discussed in detail. **Q.E.D**

Chapter 8

Conclusions and Future Work

In this thesis, we have proposed a new BIST method to detect faults in dual-port memories using the serial interfacing technique. We have given a complete analysis of two-port faults covered, and we have proposed serial marching algorithms to detect these faults. But, why do we need to test dual-port faults in memories? Inductive fault analysis (IFA) was used at Intel to determine that 94% of the faults that occur in memory dies are 1PFs and only 6% are 2PFs[6]. However, 6% is a really a huge percentage considering the drive towards shipping products with a near-zero defect level. This is the first time that serial interfacing is applied to detect advanced fault models of dual-port embedded memory arrays for SoC circuits. The proposed test architecture and BIST technique has the advantages of high fault coverage, low hardware overhead and tolerable test application time for memory arrays with short word-count. For memory arrays with large word-count, *split memory testing* was proposed in [12]. The basic idea is to split a large memory array into small sections and test each section using serial interfacing simultaneously.

Since we use the serial interfacing technique, dataflow is scheduled in a very organized manner as it moves through the memory and a test pattern is applied every clock cycle. One immediate conclusion that can be reached is that, based on the proposed BIST technique, built-in self diagnosis (BISD)[25] can be achieved. Instead of using a global MISR for test reponse analysis, a comparator can be used for each individual dual-port array to monitor the faulty bit number. The comparator is nothing but a simple finite state machine. Since the memory address is implemented by a global counter during testing, the word address and bit number of a faulty cell can be easily determined. We can also foresee that redundant operation will not cause any problem for the BISD technique based on serial interfacing. Thus, the BISD circuits can be designed in a very elegant test architecture.

After we perform BISD on the array, we can remove faulty cells and replace them with good ones. Thus, another area of research which stems from this thesis is in the area of built-in self repair (BISR)[14][3][23][17]. We have embedded memory arrays distributed in different places around the chip, rather than in a single centralized location. These small arrays need to have some percentage of memory cells set aside as spare/redundant cells which can be possible replacements for cells identified faulty by the dual-port march algorithms. The percentage of redundancies required is mainly based on array size and fault distribution. The replacement mechanism (cell replacement, column replacement, or row replacement) is required to be determined considering the area overhead to be of primary importance, especially because we are dealing with embedded memory arrays. Ideally, after such a procedure is determined, we have the BIST, BISD and BISR circuitry to be integrated in the form of a single controller performing all the test, diagnosis and repairs for memory arrays on a SoC circuit. Also, one more feasible and important extension needs

to be discussed. In this thesis, we have focussed on BIST for two-port memories because the usage of two-port memories in the industry exceeds that of multiport memories with number of ports(p) > 2. But, what if some specific applications need the use of memories with $p > 2$. How do we test them by serial interfacing efficiently? Some march tests which use parallel interfacing technique for p-port memories have been proposed in [7][33], but no research has been done on serial interfacing. This must be investigated and solved as more and more applications do require embedded memories with a large number of ports to enhance the circuit performance.

Bibliography

[1] V.C. Alves and M. Nicolaidis. Detecting complex coupling fault in multi-port rams. In *IMAG Research Report No RR978*, 1991.

[2] V.C. Alves, M. Nicolaidis, and H. Bederr. Testing complex couplings in multi-port memories. In *IEEE Transactions on VLSI Systems*, pages 59–71, 1995.

[3] T. Chen and G. Sunada. A self-testing and self-repairing structure for ultra-large capacity memories. In *Proceedings of International Test Conference*, pages 623–631, 1992.

[4] B. Nadeau Dostie, A. Silburt, and V. K. Agarawal. Serial interfacing technique for embedded memory testing. In *IEEE Design and Test of Computers*, pages 52–63, April 1990.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of 10th ACM STOC*, pages 114–118, 1978.

[6] S. Hamdioui, M. Rodgers, A.J. van de Goor, and D Eastwick. March tests for realistic faults in two-port memories. In *Memory Technology, Design and Testing*, pages 73 – 78, 2000.

[7] S. Hamdioui, M. Rodgers, A.J. van de Goor, and D Eastwick. Detecting unique faults in multi-port srams. In *Test Symposium 2001. - Proceedings. 10th Asian*, pages 37 – 42, 2001.

[8] Technical Article http://www.cypress.com/. Understanding specialty memories: Dual-port rams. In *Cypress Semiconductor*, 2000.

[9] ITRS-01. International technology roadmap for semiconductors, 2001.

[10] W. B. Jone and D. C. Huang. A parallel built-in self-diagnostic method for embedded memory arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(4):449–465, April 2002.

[11] W.B. Jone, D.C. Huang, and S.R. Das. An efficient bist method for non-traditional faults of embedded memory arrays. In *Proceedings of the 19th IEEE Instrumentation and Measurement Technology Conference*, volume 1, pages 601–606, 2002.

[12] W.B. Jone, D.C. Huang, S.C. Wu, and K.J. Lee. An efficient bist method for small buffers. In *VLSI Test Symposium*, pages 246 – 251, 1999.

[13] R. M. Karp and R. E Miller. Parallel program schemata. In *Journal of Computer and System Sciences*, pages 147–195, 1969.

[14] I. Kim, Y. Zorian, G. Komoriya, H. Pham, F. Higgins, and J. Lewandowski. Built-in self-repair for embedded high density sram. In *Proceedings of International Test Conference*, pages 1112–1119, 1998.

[15] J. C. Lee, Y. S. Kang, and S. Kang. A parallel test algorithm for pattern sensitive faults in semiconductor random access memories. In *Proc. Int'l Symp. on Circuit and Systems*, pages 2721–2724, 1997.

[16] P. Mazumder and J. K. Patel. Parallel testing for pattern-sensitive faults in semiconductor random-access memories. In *IEEE Trans. on Computers C-38*, pages 394–407, 1989.

[17] P. Mazumder and J. S. Yih. A novel built-in self-repair approach to vlsi memory yield enhancement. In *Proceedings of International Test Conference*, pages 833–841, 1990.

[18] Y. Morooka, S. Mori, M.Miyamoto, and M. Yamada. An address maskable parallel testing for ultra high density drams. In *Proc. Int'l Test Conf*, pages 556–563, 1991.

[19] B. Nadeau-Dostie, A. Silburt, and V.K Agarwal. A serial interfacing technique for built-in and external testing of embedded memories. In *Custom Integrated Circuits Conference*, pages 22.2/1 – 22.2/5, 1989.

[20] M.J. Raposa. Dual port static ram testing. In *In Proc. IEEE International Test Conference*, pages 362–368, 1998.

[21] J. T Schwarz. Large parallel computers. In *Journal of the ACM*, pages 25–32, 1966.

[22] T. Shridhar. A new parallel test approach for large memories. In *IEEE Design and Test of Computers*, pages 15–22, 1986.

[23] A. Tanabe. A 30ns 64-mb dram with built-in self-test and self-repair function. In *IEEE J. Solid-State Circuits*, volume 27, pages 1525–1531, November 1992.

[24] L. Ternullo, R. Dean Adams, J. Connor, and G. S. Koch. Deterministic self-test of a high-speed embedded memory and logic processor subsystem. In *Proc. Int'l Test Conf.*, pages 33–44, 1995.

[25] R. Treuer and V. K. Agrawal. Built-in self-diagnosis for repairable embedded ram's. In *IEEE Design and Test of Computers*, pages 24–33, June 1993.

[26] A.J. van de Goor. *Testing Semiconductor Memories:Theory and Practice*. John Wiley and Sons, 1991.

[27] A.J. van de Goor and S. Hamdioui. Consequences of port restrictions on testing two-port memories. In *Proc. Int'l Test Conf.*, pages 63–72, 1998.

[28] A.J. van de Goor and S. Hamdioui. Fault models and tests for two-port memories. In *Proc. 16th IEEE VLSI Test Symp*, pages 401–410, 1998.

[29] A.J. van de Goor and S. Hamdioui. Port interference faults in two-port memories. In *Proc. Int'l Test Conf*, pages 1001–1010, 1999.

[30] A.J. van de Goor and S. Hamdioui. Efficient tests for realistic faults in dual-port srams. In *Proc. VLSI Test Symp.*, pages 395 – 400, 2002.

[31] A.J. van de Goor and S. Hamdioui. Thorough testing of any multiport memory with linear tests. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21 of *2*, pages 217 – 231, 2002.

[32] B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and debug strategy of the pnx8525 nexperia digital video platform system chip. In *Proceedings of International Test Conference*, pages 121–130, 2001.

[33] Chi-Feng Wu, Chih-Tsun Huang, Kuo-Liang Cheng, and Cheng-Wen Wu. Fault simulation and test algorithm generation for random access memories. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 480–490, 2002.

[34] J. Zhao, S.Irrinki, M. Puri, and F. Lombardi. Detection of inter-port faults in multi-port static rams. In *Proc. VLSI Test Symp.*, pages 297–302, 2000.