POLITECNICO DI TORINO
Repository ISTITUZIONALE

Infrastructures and Algorithms for Testable and Dependable Systems-on-a-Chip

Original
Infrastructures and Algorithms for Testable and Dependable Systems-on-a-Chip / Di Carlo, S.. - ELETTRONICO. - (2003), pp. 1-106.

Availability:
This version is available at: 11583/2709266 since: 2018-07-17T08:31:44Z

Publisher:
Politecnico di Torino

Published
DOI:10.6092/polito/porto/2709266

Terms of use:
openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

04 August 2020

POLITECNICO DI TORINO
DOCTORAL SCHOOL
PhD Course in Computer and Control Engineering – XV cycle

PhD Dissertation

Infrastructures and Algorithms for Testable and Dependable Systems-on-a-Chip

Stefano Di Carlo

Tutor	**Coordinator of the PhD**
Prof. Paolo Prinetto	prof. Pietro Laface

2003

This work is subject to the Creative Commons Licence

Contents

List of Tables VI

List of Figures VII

1 Infrastructures and Algorithms for Testable and Dependable SoC 1
- 1.1 High Performance SoC Test Diagnosis and Failure Analysis 2
- 1.2 Memory Dominance 3

I High Performance SoCs Test, Diagnosis and Failure Analysis 5

2 Introduction 9

3 State-of-the-Art 11

4 Architecture Overview 13

5 HD^{2}BIST hierarchy configuration and scheduling 19

6 RT-level Implementation 23
- 6.1 Test Block Architecture 23
- 6.2 Test Processor Architecture 24
- 6.3 IEEE 1149.1 interface 25
- 6.4 Scan Chain Router 25
- 6.5 Test Control Bus 26
- 6.6 Token Format 26

7 Using HD^{2}BIST in SoC Testing 29
- 7.1 Testing BISTed and BIST-Ready Cores 29
- 7.2 Testing Scan Cores 30
- 7.3 Testing glue logic 30

8 Case study 33
- 8.1 DacTOPplus Architecture 33
- 8.2 DacTOP Test Structure 33

8.3 DacTOPplus Test Structure . . . 36
8.3.1 Running a test program . . . 37

9 Conclusions 41

II Memory Dominance 43

10 Introduction 47

11 An introduction to memory Test 49
11.1 The fault model . . . 49
11.1.1 Memory Cell Faults . . . 49
11.1.2 Address decoder fault . . . 50
11.1.3 Read/Write logic faults . . . 50
11.2 Memory test Algorithms . . . 50

12 Memory BIST 51
12.1 The BIST Processor . . . 52
12.2 Wrapper Structure . . . 54
12.2.1 Dispatcher . . . 55
12.2.2 Port Wrapper . . . 55
12.2.3 Multiplexing . . . 56
12.3 Diagnosis . . . 57
12.4 Further optimizations . . . 57
12.4.1 Sharing Wrappers among SRAMs clusters . . . 57
12.4.2 Using a Topological approach for complex coupling fault testing . . . 58
12.5 Case study . . . 58
12.5.1 Case Study Architecture . . . 59
12.5.2 Case Study BISTArchitecture . . . 60
12.5.3 Case Study BIST Scheduling . . . 61
12.5.4 Experimental results . . . 62

13 Automatic Test Generation 67
13.1 State of the Art . . . 67
13.2 Memory Model . . . 68
13.3 March Test Generation Algorithm . . . 71
13.3.1 GTS Reordering . . . 74
13.3.2 GTS minimization . . . 74
13.3.3 March Test Generation . . . 75
13.4 BFEs equivalence . . . 75
13.4.1 Experimental Results . . . 76

14 Memory Test Algorithms Verification 77
14.1 The Fault Simulator Architecture 77
14.2 The Electrical and Physical Model 79
14.3 Input Test Sequences 79
14.4 Simulator Engine 80
14.4.1 Test Analysis Module 81

15 Self Repair 83
15.1 State of the art 84
15.2 The conceptual architecture 85
15.2.1 Memory Built In Self Repairing 86
15.2.2 Memory Built In Self Testing 87
15.3 Actual Implementation 89
15.4 BISTAR validation 91
15.5 Experimental results 91
15.5.1 Area overhead 91
15.5.2 BISR Logic Fault Coverage 93

16 Conclusions 95

A Publication list 97
A.1 International journals 97
A.1.1 Year 2003 97
A.2 International conferences 97
A.2.1 Year 2002 97
A.2.2 Year 2001 98
A.2.3 Year 2000 98

Bibliography 101

List of Tables

8.1 DacTOPplus dimensions in Synopsys® equivalent gates 34
8.2 DacTOP area with wrapped modules 36
8.3 HD^{2}BISTed DacTOP area occupation 36
8.4 Area result of DacTOPplus 38
12.1 March Algorithm Test Primitives 53
12.2 8 bits Background patterns $\mathrm{BP_j}$ for CFsts 54
12.3 March Algorithm Representation 63
12.4 Meanings of the Output Synchronization signal 64
12.5 Memory Wrapper overhead 65
12.6 Total area overhead 65
13.1 Reordering Rewrite Rules 74
13.2 Reordering Rewrite Rules 75
13.3 Experimental Results 76
15.1 Percentage of Area overhead introduced by the BISTAR Architecture . . . 92
15.2 BISR Logic Fault Coverage 93

List of Figures

4.1 HD^{2}BIST Ring . . . 14
4.2 TCB (token bus) and TDB (scan bus) . . . 15
4.3 Test Block and core wrapper . . . 15
4.4 Test Processor architecture . . . 16
4.5 Connecting two rings . . . 17
5.1 TDB connections . . . 20
6.1 Test Block General Architecture . . . 24
7.1 Using of TDB to test different types of cores . . . 30
7.2 Pseudo TDB lines for testing the glue logic . . . 31
8.1 DacTOPplus schema . . . 34
8.2 DacTOP HD^{2}BIST schema . . . 35
8.3 DacTOPplus with HD^{2}BIST . . . 37
8.4 Test program PROG[1] of the Top Level TP . . . 38
8.5 Test program PROG[2] of the TLTP . . . 39
8.6 Test program PROG[1] of the DacTOP macro . . . 39
8.7 Test program PROG[4] of the Top Level TP . . . 40
8.8 Test program PROG[3] of the DacTOP macro . . . 40
12.1 Basic memory-BIST Architecture . . . 52
12.2 Wrapper architecture . . . 55
12.3 Test Instruction execution diagram . . . 56
12.4 Wrapper structure . . . 57
12.5 Results_Scan_Chain . . . 58
12.6 VC12AD memories organization . . . 59
12.7 VC12AD floorplan . . . 60
12.8 VC12AD BIST Architecture . . . 61
12.9 Wrappers area . . . 62
13.1 M_0 FSM representing a fault free RAM . . . 69
13.2 M_1: $\langle\uparrow,0\rangle$ Idempotent Coupling Fault Representation . . . 70
13.3 BFE model for $\langle\uparrow,0\rangle$ Coupling Fault . . . 70
13.4 TPG for $\{\langle\uparrow,1\rangle\langle\uparrow,0\rangle\}$ . . . 72
14.1 Simulator architecture . . . 78
14.2 Example of test algorithm including Neighborhood cells and a cycle statement . . . 80
14.3 The simulator engine layered structure . . . 80

15.1 BISTAR conceptual architecture 85
15.2 Memory layout 87
15.3 Minimization of the address critical path 90

“The Silicon Infrastructure Opportunity”

“Economists have long argued that the key factor for the boom in interstate automobile travel was not the mass production of the car as in Ford’s Model T in the 1920s, but rather the highway infrastructure built near the end of World War II and into the postwar growth era. Market analysts have argued that the key factor for the adoption of semiconductor Intellectual Property (IP) was the realization that no System-on-a-Chip (SoC) semiconductor design team could develop every block of IP from scratch in a viable time-to-market window.

Despite all the announcements over SoCs in the trade press during the past few years, the widespread deployment of SoCs has still not occurred (with the exception of a few general-purpose, graphics, and game processor designs). However, SoCs will come into their own at the 0.13-micron technology node, a confluence of integrated complexity (Moore’s law) and process fungibles (having memory and analog devices on a base-line digital process). Although both trends are necessary, they will not be sufficient to drive widespread SoC adoption.

Unlike prior process generations, the nanometer era will need more than just hardware-based tooling to realize production volumes. The mass production of 130-nm SoCs will require the commercial availability of Infrastructures (software-assisted technologies that span the ramp from design tape-out to manufacturing). Examples of software-assisted tooling include mask synthesis, embedded test, process-yield simulation, automatic test generation and built-in reliability. By 2005, SoC component revenues will grow to $35 billion (nearly 20% of overall chip revenues, not including microprocessor units). The semiconductor IP component market (licensing and royalties) will be at $2.6 billion. The silicon infrastructure market will catapult to $3.4 billion. These bold predictions impinge on the ability of the emerging silicon infrastructure vendors to extract fair value for value added, that is a business model predicated on Time-To-Volume (TTV) royalty versus selling software seats as in the traditional electronic-design-automation model. The silicon infrastructure market has already seen some early success stories on this front. Hindsight will decisively prove that the most significant driver for the adoption of SoCs was the emergence of the silicon infrastructure market. Only history will tell which forward-looking innovators will have made the most of this opportunity, and which will have been left behind in the dust of antiquated business models.”

[Erach Desai, DESAIsive Technology Research
IEEE Design & Test of Computer Systems
Guest Editor’s Introduction
May-June2002]

Chapter 1

Infrastructures and Algorithms for Testable and Dependable SoC

Every new node of semiconductor technologies provides further miniaturization and higher performances, increasing the number of advanced functions that electronic products can offer. Silicon area is now so cheap that industries can integrate in a single chip usually referred to as System-on-Chip (SoC), all the components and functions that historically were placed on a hardware board. Although adding such advanced functionality can benefit users, the manufacturing process is becoming finer and denser, making chips more susceptible to defects. Today's very deep-submicron semiconductor technologies (0.13 micron and below) have reached susceptibility levels that put conventional semiconductor manufacturing at an impasse. Being able to rapidly develop, manufacture, test, diagnose and verify such complex new chips and products is crucial for the continued success of our economy at-large. This trend is expected to continue at least for the next ten years making possible the design and production of 100 million transistor chips [83] [86].

To speed up the research, the National Technology Roadmap for Semiconductors identified in 1997 a number of major hurdles to be overcome. Some of these hurdles are related to test and dependability [37].

Test is one of the most critical tasks in the semiconductor production process where Integrated Circuits (ICs) are tested several times starting from the wafer probing to the end of production test. Test is not only necessary to assure fault free devices but it also plays a key role in analyzing defects in the manufacturing process [84]. This last point has high relevance since increasing time-to-market pressure on semiconductor fabrication often forces foundries to start volume production on a given semiconductor technology node before reaching the defect densities, and hence yield levels, traditionally obtained at that stage. The feedback derived from test is the only way to analyze and isolate many of the defects in today's processes and to increase process's yield.

With the increasing need of high quality electronic products, at each new physical assembly level, such as board and system assembly, test is used for debugging, diagnosing and repairing the sub-assemblies in their new environment. Similarly, the increasing reliability, availability and serviceability requirements, lead the users of high-end products performing periodic tests in the field throughout the full life cycle.

To allow advancements in each one of the above scaling trends, fundamental changes are expected to emerge in different Integrated Circuits (ICs) realization disciplines such as IC design, packaging and silicon process. These changes have a direct impact on test methods, tools and equipment. Conventional test equipment and methodologies will be inadequate to assure high quality levels. On chip specialized block dedicated to test, usually referred to as Infrastructure IP (*Intellectual Property*), need to be developed and included in the new complex designs to assure that new chips will be adequately tested, diagnosed, measured, debugged and even sometimes repaired [85].

In the following of this introduction, some of the scaling trends in designing new complex SoCs will be analyzed one at a time, observing their implications on test and identifying the key hurdles/challenges to be addressed. The goal of the remaining of the thesis is the presentation of possible solutions. It is not sufficient to address just one of the challenges; all must be met at the same time to fulfill the market requirements.

1.1 High Performance SoC Test Diagnosis and Failure Analysis

The design of high performance SoCs requires increased performances of test equipments. While semiconductor off-chips speeds have improved at 20% per year, tester speed has improved at rate of 12% per year. Tester timing errors are approaching the cycle time of the fastest devices. Furthermore the so called bandwidth-gap between external pin and internal parts of a complex SoC denies the possibility of testing at operating speed modern devices. It has been demonstrated that, most of the new types of defects can only be detected if the device is tested at operating speed. The solution today is the use of Built-In-Self-Test (BIST), which means the integration of test capability directly on-chip [1]. Many solutions have been proposed in literature but the complexity of new designs makes them difficult to implement. Usually, each component or function in a SoC is now available as a pre-designed functional block, or *embedded core,* whose internal structure is usually hidden to the core integrator. Moreover, cores may embed different test architectures, like Full-Scan [1], Partial-Scan [1], or BIST, and may be *reused* in different designs and integrated with other cores coming from different vendors. In addition, when a certain combination of cores is often used together, the system integrator or the core provider may decide to create a new core from that combination. Hence, today's SoCs may become tomorrow's cores in more complex SoCs. This new design philosophy based on a hierarchical reuse of cores, leads to a radical change in the test engineering process, requiring the adoption of IP test infrastructure able to fully support *core reuse*, *hierarchical design*, and *multiple test strategies integration.*

In addition, failure analysis and diagnosis in faulty chips is becoming a key problem. The traditional failure analysis comprises fault localization, silicon reprocessing and physical characterization and inspection steps. The migration towards smaller geometries severely challenges this analysis process. The only solution is gathering failure data by using embedded diagnosis infrastructure IP, such as signature analyzers, dedicated test vehicles or on-chip test processors, and then analyzing the obtained data by off-chip fault localization methodologies and tools.

1.2 Memory Dominance

There is a clear trend to integrate large quantities of memories on a chip. Memories are designed with aggressive design rules and tend to be more prone to manufacturing defects. Memories in complex SoC are deeply embedded in the circuits and are not easily accessible from the outside to apply test program. New IP Infrastructures to embed the test programs and to allow diagnosis and debug operation are required.

Furthermore the increasing scale of integration and the introduction of new manufacturing processes introduce new classes of defects not present in the previous technologies. To efficiently deal with them, designers need to automate the process of memory test program generation. This goal involves the realization of algorithms able to address the problem of: (i) describing new fault models (ii) generating tests able to cover these new fault models, (iii) verifying the effectiveness of the generated test programs.

Finally, embedded memories were traditionally testable but not reparable. Many applications today need to keep on working in case of fault. This is addressed in today's memories by introducing redundancy, i.e. spare elements. However, having redundancy only does not solve the problem, it is necessary to find optimal solutions to detect the defects and to allocate the redundant elements. This involves the realization of dedicate infrastructure IPs to realize Self-Repairable Memories.

Part I

High Performance SoCs Test, Diagnosis and Failure Analysis

Related publications

Portions of the material described in this part of the thesis has been subject of following scientific publications: [6],[19], [18].

Chapter 2

Introduction

As mentioned in chapter 1 the new SoC design philosophy based on a hierarchical reuse of already made cores, requires the adoption of IP test infrastructures able to fully support core reuse, hierarchical design, and multiple test strategies integration.

The problem of testing already-made components bought from different vendors is a well-known problem in the field of discrete components. Specifically, the test of boards is based on the direct access to the component's pins using bed-of-nails whose probes contact the pads and the wire of the board under test. In SoC design, direct accessibility to interconnections and cores' boundaries is not possible, but test patterns have still to be carried from their source, either a BIST-controller or an external Automatic Test Equipment (ATE) [1], to the core and vice versa. The problem has to be solved by designing an access-architecture, usually referred to as *Test Access Mechanism* (TAM), to activate the test functions, possibly delivering test patterns, and gathering the results of the test of any core in the overall SoC hierarchy. In general, a TAM has to guarantee three main properties:

- *Core accessibility*: the test of the core has to be controllable by a limited set of SoC boundary signals;
- *Reusability*: the access mechanism should be easily reconfigurable to allow the reuse of cores with different test architectures;
- *Lowest overhead*: area, routing, and performance overheads must be kept at a minimum.

To worsen the situation, test execution in general, and BIST in particular, typically result in a circuit activation-rate and power consumption higher than the one of the normal operation mode, thus limiting the possibility of testing concurrently all the cores embedded in a SoC. A SoC test strategy has therefore to be organized in several sessions, each carefully planned to fulfill the required power dissipation constraints.

It is now clear how the SoC integration task would be simpler if core designs were more test-friendly, and SoC designers would have more flexibility in choosing the best overall test methodologies for their chips. For this purpose, the IEEE P1500 standard for embedded-core test is under development. The P1500 standard strives to provide a "plug-and-play" methodology of integrating core testability into a SoC. Its goal is to ensure test-friendliness and interoperability of cores coming from diverse sources. P1500 concentrates on a standardized, but configurable and scalable core interface or *wrapper* that allows easy *access* to the internal test methods of the core. The *Test Access Mechanism* (TAM), which has the task of managing the execution of the test of the overall chip, is out of the current scope of P1500 and therefore must still be designed by the test engineer.

This part of the thesis proposes a possible solution to this problem. An innovative TAM named *Hierarchical-Distributed-Data BIST* (HD^{2}BIST), able to address several of the most critical issues in SoC testing, will be presented. HD^{2}BIST allows a smooth integration and management of cores with different test strategies (e.g., Full Scan [1], Partial Scan [1], BIST-ready [1], or BISTed cores [1]) and built-in test access protocols, either user-defined or automatically inserted using commercial BIST and Design-for-Testability insertion tools. It is fully compatible with a hierarchical design methodology, allowing accessing any core of the system independently from its hierarchical depth.

Being already addressed by the IEEE P1500 task force, HD^{2}BIST does not focus on the problem of *core isolation*. Nevertheless, although HD^{2}BIST does not require the cores of the system to be P1500 compliant, it assumes at least a simple wrapper to be placed around each core in order to guarantee its *isolation* during the test execution. Obviously, in the scenario of a P1500 compliant design, a significant part of the HD^{2}BIST structures would be merged with the P1500 wrapper to optimize performances and minimize the area overhead.

To reduce the power consumption and therefore to overcome one of the main drawbacks of the BIST methodology in SoCs, the HD^{2}BIST approach allows defining the test scheduling of the cores using sophisticated control flow mechanisms. A very low area overhead is guaranteed by the fact that the HD^{2}BIST architecture is fully customizable and adaptable to the test requirements of the cores integrated in the system, and thus provides a trade-off capability among routing, area, and test length.

Chapter 3

State-of-the-Art

As introduced in chapter 2 the two main issues to be targeted in SoC testing are *core isolation* and *core accessibility.* [87] gives an overview on the current solutions to create testable and diagnosable embedded core-based SoCs, and presents a generic conceptual architecture consisting of three structural elements. It introduces the basic concepts of test pattern *source*, test pattern *sink*, and the *test access mechanism* (TAM).

Several TAM architectures have been proposed in literature. *Macro Test* in [56] separates tests into protocols and test patterns. It introduces a dedicated test access mechanism but also takes advantage of existing on chip functionalities. [39] and [38] use the functional transparent mode of the cores to propagate the test data through the system. The approach requires low-test area but quite high-test application time since the test data need to be propagated trough the cores. In [47] the core under test is directly and parallel accessed from the IC pins, inserting additional wires connected to the core terminal and *multiplexed* into the existing IC pins. As a main drawback, the approach is not scalable and introduces high area costs and long test time. [75] and [76] access the core under test from the IC pins using multiple *test buses* of different width shared by multiple cores. Each test bus allows testing one single core at the time. In [41] a new architecture template has been proposed for system level connection based on switching networks.

Alternative solutions reuse the approaches developed for board level interconnects testing, through the JTAG IEEE 1149.1 Boundary Scan Architecture [61]. Some aspects however limit the effectiveness of the approach: the Boundary Scan is optimized for testing interconnections between components in a board and it does not allow at-speed testing. In [65] the authors present a variation of the IEEE 1149.1 standard based on a partial boundary scan ring around the core, whereas in [77] they suggest to provide each core of an addressable Test Access Port (TAP) in order to directly address the core to be tested, and to introduce a special hierarchical *TAP* to manage group of cores as a single one. To test interconnections among cores, the method proposed in [24] requires the insertion of test collar cells on the virtual core input and output pins, to create different connections between the cells and the system data bus. In [55], the cores are wrapped by an ad-hoc interface (*TestShell*), and connected by a proper test bus (*TestRail),* that delivers the test data patterns and control signals.

To allow flexibility in test scheduling, [5] and [42] proposed a centralized controller to activate the BIST sessions one at a time, whereas [81] suggests distributing the test management on different *BIST Resource Controllers* (*BRCs*).

The area overhead reduction was addressed in [62]: a local test controller is associated to each core, and a dedicated bus inserted to connect the controllers, deliver data patterns, and link the controllers to the Unit Under Tests (UUT). More UUTs can share the same controller, properly configuring the local test controller. A modular, generic and re-configurable TAM architecture has been proposed in [58], based on a packet switching communication network. Finally, the concept of hierarchical distribution of the test management is introduced in [22][21] and [23] with the *Hierarchical* and *Distributed BIST* architecture, able to manage different hierarchical levels of BISTed blocks.

Concurrently with the development of proper testing architectures, some initiatives started to standardize the cores embedding process. The *Virtual Socket Interface Alliance* (VSIA) and the *P1500 IEEE Standard Working Group* are the most relevant ones. The VSI Alliance intends to define, develop, ratify, test and promote open specifications relating to data formats, test methodologies and interfaces, to help the reuse of intellectual property blocks from different sources in the design and development of SoCs [67]. The P1500 IEEE Standard Working Group aims at defining a uniform but flexible hardware interface between an embedded core and its environment, capable of delivering predefined test patterns to and from the embedded cores [2][82][49].

Despite the novelty of the approaches, none of the proposed solutions efficiently fulfills all the test requirement of new generation SoCs.

Chapter 4

Architecture Overview

This chapter presents the general architecture of the HD^{2}BIST test access mechanism. For the sake of clarity, HD^{2}BIST is here introduced as a flat architecture, but the reader should keep in mind that it is a hierarchical structure, fully adaptable to the hierarchical architecture of complex SoCs.

The main actor in the HD^{2}BIST is a bus-based communication link, named *Test BUS (TBUS)*, which implements an effective solution to the problem of *core access* and *reuse.* From a conceptual point of view, the data exchanged on the TBUS can be clustered into two categories:

- *Control Data*, used to configure and control the HD^{2}BIST test structures.
- *Test Data*, used to carry the test vectors needed to test the cores embedded into the system. Test vectors can be generated on-chip by BIST controllers, or be applied from the outside to the SoC Input/Output signals using Automatic Test Equipments (ATE).

To reflect this logical classification, the TBUS is split into two sub-buses named *Test Control Bus (TCB)* and *Test Data Bus (TDB)*, respectively. Both test busses have a *ring structure* (Figure 4.1), to guarantee a simple and technology independent approach, and to offer a high degree of flexibility and dependability.

The difference of information exchanged on the two busses imposes the use of different *Communication Protocols.* The amount of information exchanged over the TCB is usually very low and it can be easily coded as a predefined set of commands, thereinafter referred to as *Test Primitives.* A *Token Based Protocol* has been chosen as the most effective solution for the TCB. On the contrary, data exchanged on the TDB is mainly composed of a large amount of test patterns and test responses. This characteristic makes a *Scan Chain Based Protocol* [1] the most suitable solution for data transmitted on the TDB (Figure 4.2).

As detailed later, the scan chain approach allows a very flexible *sharing mechanism* of the TDB to concurrently test different cores, or to reuse the same lines of the TDB to test different cores in different times, thus minimizing the routing overhead.

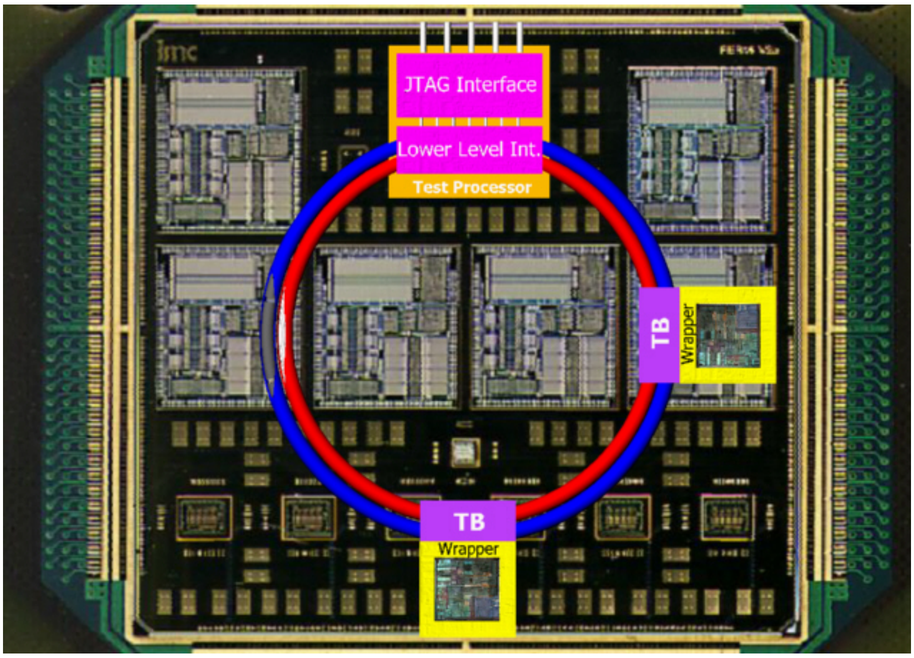


Figure 4.1. HD^{2}BIST Ring

The TBUS structure implemented by HD^{2}BIST provides a high degree of reliability. Thanks to the Scan Chain protocol, the TDB itself can be easily tested using a standard scan test approach [1]. The dependability of the TCB is guaranteed by implementing it as a bi-directional link. The same information transmitted on a *forward link* is then sent back on a *backward link* to verify its correctness. In case of a transmission error, a diagnosis procedure can be executed to locate the fault.

The TBUS proves to be an efficient TAM only if an appropriate *bus interface* with the cores under test and a *bus manager* are properly defined and implemented. In the HD^{2}BIST these rules are played by two special blocks named *Test Block* (TB) and *Test Processor* (TP), respectively. Each TB/TP connected to the TBUS is identified by a unique address defined at design time.

The Test Block (Figure 4.3) can be easily customized to support the specific test solution implemented in the core. In particular, its internal structure is optimized for (1) *full and partial scan cores* [1], using the TDB to apply test vectors and to gathered test results, (2) *BISTed cores* [1], using the TCB to send BIST commands and to read BIST results, and (3) *BIST-ready cores* [1], using the TDB to exchange test vectors between the core under test and its BIST controller.

The Test Processor controls, through the TBUS, the HD^{2}BIST structures inserted in the chip, and schedules the execution of the test of each core in the chip itself. It is interfaced to the TBUS through a *Lower Level Interface* (Figure 4.4) implemented either as a Finite State Machine (FSM) or as a micro-programmed machine, and able to execute a sequence of *Test Primitives* implementing a set of *Test Programs* defined by the core

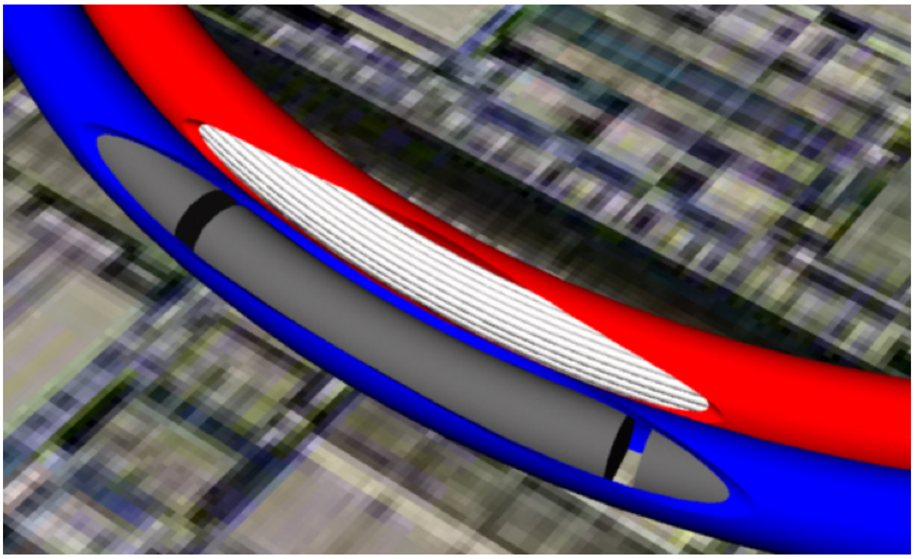


Figure 4.2. TCB (token bus) and TDB (scan bus)

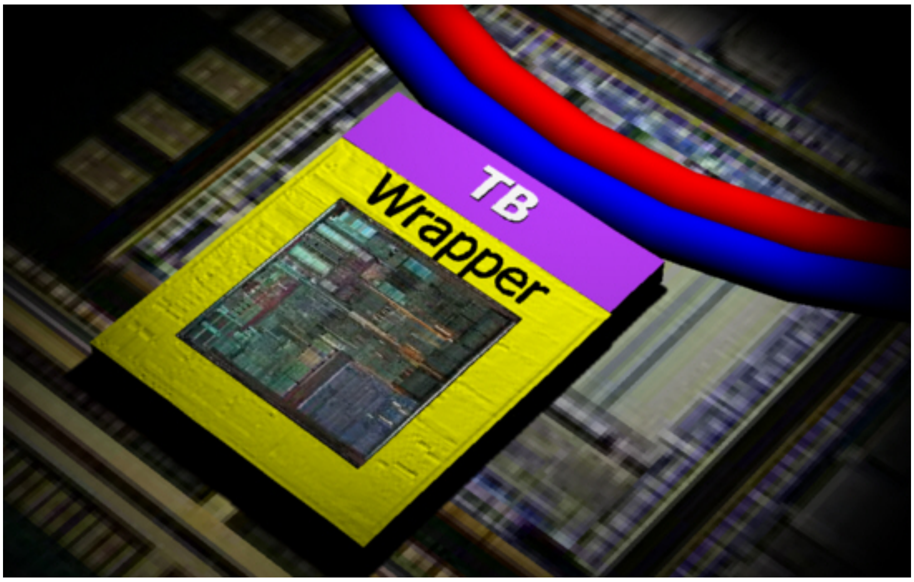


Figure 4.3. Test Block and core wrapper

integrator. Each test primitive corresponds to one (or more) token(s) exchanged over the TCB, and it can be either generated by the TP itself, or applied from the outside of the SoC through a *IEEE 1149.1 JTAG* interface [61]. The JTAG defines a standard Test Access Port (TAP) for board components. The use of a JTAG interface allows having a standardized protocol to access the TP and the HD^{2}BIST structure in general and to control the execution of the Test Programs from an external ATE.

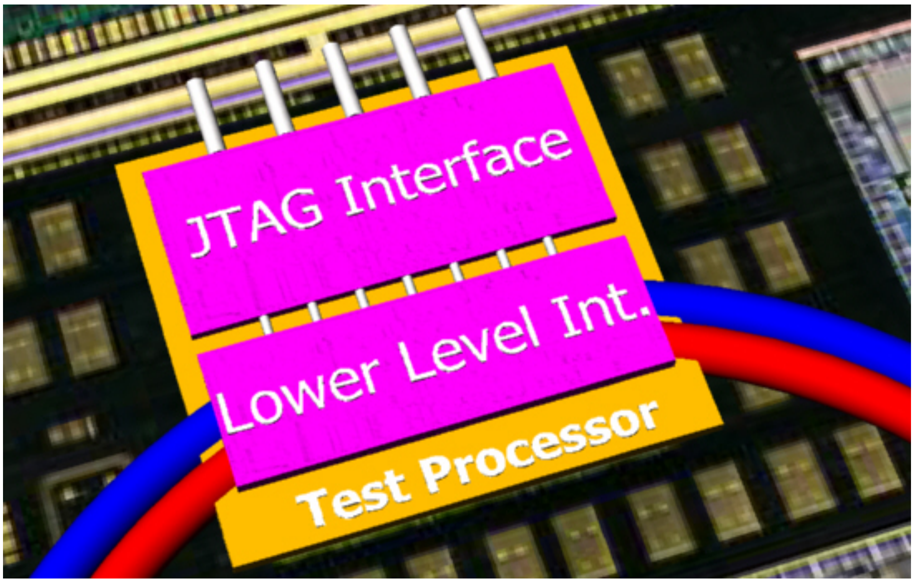


Figure 4.4. Test Processor architecture

As explained in the Section 1, the design of a complex SoC usually follows a hierarchical approach. To be compatible with this design philosophy, HD^{2}BIST allows connecting different TBUS to achieve a hierarchical structure. The element that allows the connection of different busses is the TP. In this case, instead of being controlled by a JTAG interface, the TP is interfaced to the second TBUS through a TB-like interface named *Upper Level Interface* (Figure 4.5). Each TP manages the test of the cores connected to its TBUS, only. The Upper Level Interface allows viewing each TP as a complex TB and therefore managing it as a single Unit Under Test: a TP placed at the i^{th} level[1] will be considered by the TP of level $(i-1)^{th}$ as a standard TB, and therefore dealt with as any other core under test. Nevertheless, its task is to translate the test primitives coming from the $(i-1)^{th}$ level in the execution of the appropriate Test Program for the blocks belonging to its TBUS. Using this approach, TPs support a distributed approach in the execution of the system test. The general idea is that each TP is able to resolve the commands coming from the upper bus in all the operations needed on the lower bus (Figure 4.5).

As mentioned before each TB and TP is identified by a unique address defined at design time. To extend this concept when the SoC is reused as an embedded core in a more complex system, each TBUS is considered as a distinct address domain. In this way, two cores belonging to different TBUS can share the same address without any conflict. Using this approach, when a SoC is reused as an embedded core, the only part that has to be modified is the Upper Level Interface of the most external TP, which, in the case of a stand-alone SoC, is a JTAG interface, whereas in the case of a SoC used as a core it is

---

[1] i=0 is associated to the highest level, i.e. the chip level.

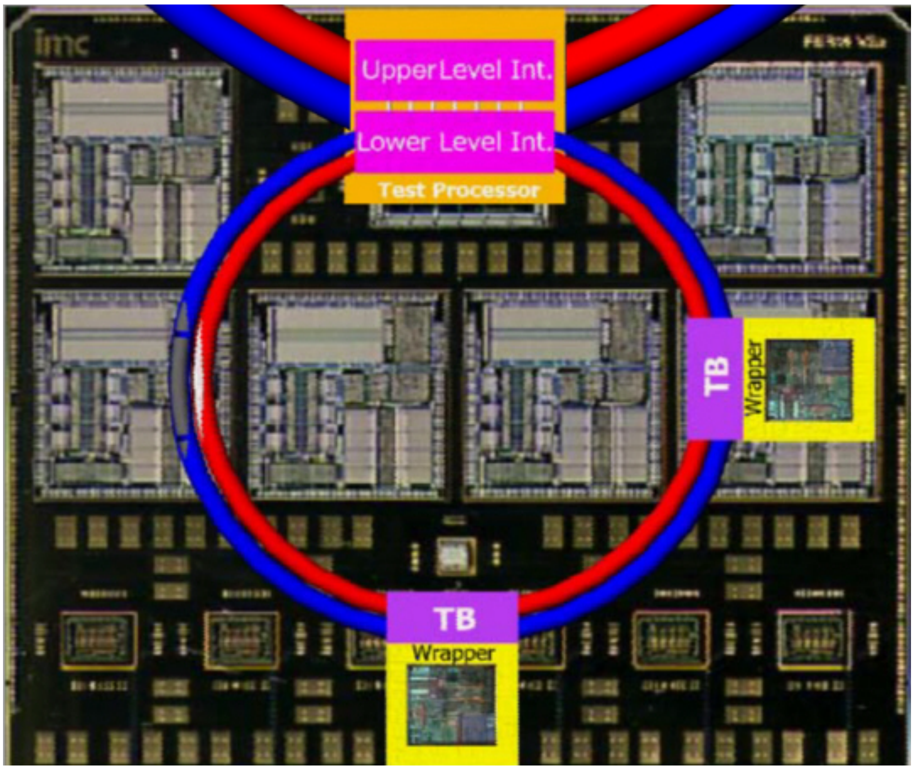

Upper Level Int.
Lower Level Int.
Test Processor
TB
Wrapper
TB
Wrapper
imc

Figure 4.5. Connecting two rings

a TB-like interface connecting the TP to the upper level TBUS.

Chapter 5

HD^{2}BIST hierarchy configuration and scheduling

The hardware architecture described in chapter 4 can be easily configured in different operative modes to create a connection in the SoC hierarchy between a core and its test patterns source/sink, to test or to perform diagnosis on faulty components and to accurately schedule the test execution of the overall SoC.

Both the *Test Structure Configuration* and the *Test Scheduling* are possible thanks to a set of *Test Primitives* implementing *Test Programs*. This mechanism defines a software level that, to some extent, allows separating the task of designing the HD^{2}BIST hardware from the task of programming and configuring it.

A Test Programs is a set of Test Primitives issued as tokens to the blocks (TBs and TPs) connected to the TCB. Conceptually, there are *atomic* and *macro* primitives. There is no semantic difference between them, but the result of their execution depends on the target block. *Atomic primitives* are commands received by a TB and used to configure the wrapper of a core or to change the status of some signals at the core boundaries; *macro primitives* are sent to those TPs that connect different hierarchical levels in the TBUS tree. In this case, the execution of the test primitive results in the activation of another test program used to manage the test of the lower hierarchical levels.

Using this software level, it is therefore possible to describe the test of a SoC as a collection of Test Programs. The order in which the test programs are executed does not necessarily have to be chosen at design time. In fact, HD^{2}BIST provides two ways of delivering a Test Primitive to the blocks under test. In *external mode,* the test instructions are sent from the outside of the system through the top-level TP, possibly using an ATE connected to the TP JTAG interface. In *internal mode*, the tokens are generated on-chip by each TP and the JTAG interface is used only to select the desired test program. The two modes are not mutually exclusive and can be integrated to add flexibility to the overall test strategy. In particular, the internal mode is mostly used to activate BIST procedures and to read their results. The external mode is instead exploited to create a direct data path form the outside to the core under test to perform diagnosis or to apply test patterns using an external ATE.

In order to exploit the external mode, the *SETENV* and *UNSETENV* macro primitives are used to configure the TCB and TDB lines to directly access, from the outside, any core of the system, regardless its hierarchical depth. SETENV is sent to the TP blocks to create a *bypass connection* between two adjacent hierarchical levels. After a SETENV primitive, all the test instructions sent to level i$^{\text{th}}$ are directly forwarded to the (i+1)$^{\text{th}}$ level. Resorting to a sequence of SETENV primitives it is therefore possible to reach, from the outside, any level of the hierarchy. This operation is usually referred to as *Environment Setup*. The UNSETENV primitive is used to restore the normal functionality of the TBUS.

In general, each Test Program requires a different *configuration* of the HD^{2}BIST. In this context, a configuration is defined as a connection scheme between the cores under test and the TDB lines used to transmit test vectors; each connection scheme is referred to as *Configuration Mode.* The set of Configuration Modes is fully customizable by the test engineer, the only constraints being that each TB has to implement at least a *Bypass Mode*, where each TDB line is not used and all the data coming into a TB is forwarded to the next block on the bus (Figure 5.1).

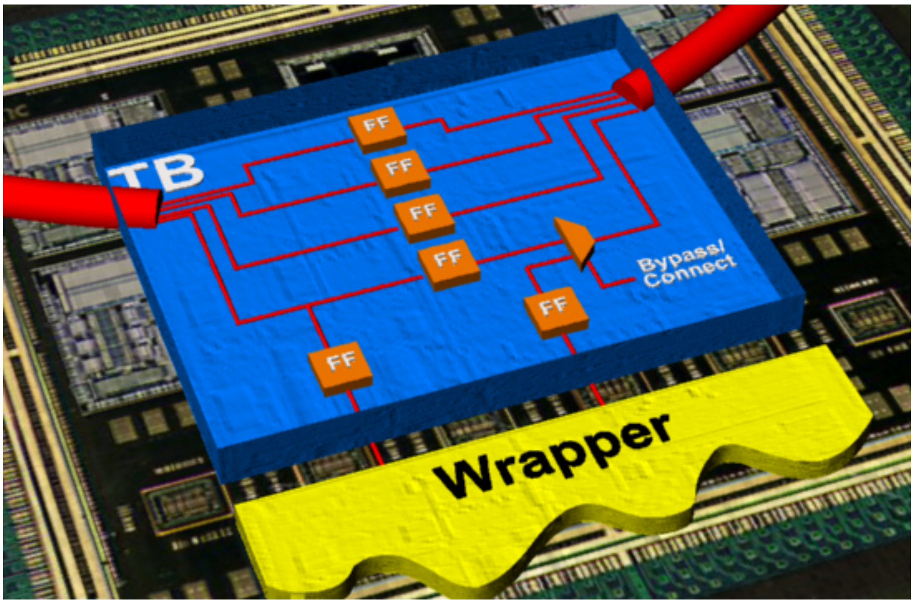


Figure 5.1. TDB connections

From the user point of view, the configuration mode can be set using the Test Primitive *CONF*. The possibility of setting different Configuration Modes allows sharing the TDB lines between different blocks in a single test session ("*Width*" sharing) or reusing the same data line to test different cores in different test sessions ("*Time*" sharing) thus obtaining the maximum efficiency from the available test resources.

The last issue solved by the bus-based approach is the Test Scheduling problem. Despite some scheduling mechanisms are already addressed solving the Test Structure Configuration problem where the "Width" and "Time" sharing are intrinsic scheduling facilities, four Test Primitives have been defined to manage the test session of BISTed and BIST-ready blocks.

The definition of a scheduling algorithm for BISTed and BIST-ready cores can be translated into an appropriate sequence of configurations, activations and collections of test results of the BIST routines. The following test primitives can efficiently code these commands:

- *START*: starts the execution of a BIST routine;
- *COLLECT*: collects the results of a BIST routine.

The two primitives are useful to define simple sequential tests, but do not allow complex scheduling algorithms where decisions should be taken depending on some test results. To overcome this problem two additional test primitives have been defined:

- *WAIT*: suspends the execution of a Test Program until the BIST procedure or the execution of the Test Program of one or more blocks is completed. This instruction allows the test engineer to address possible power consumption issues raised by the concurrent test of multiple blocks in the system.
- *JUMP*: depending on the result of the BIST (or Test Program) of one or more blocks, the Test Program execution jumps to a certain label. This command improves the flexibility in the test scheduling, allowing the test engineer to take decisions on-the-fly as, for example, to skip testing additional parts of an already revealed faulty component.

Chapter 6

RT-level Implementation

After introducing the main features of the HD^{2}BIST architecture, this paragraph details how the test structures actually may appear at the RT level. The proposed implementation can be seen as a sort of guideline for the test engineer, who can obviously customize it to his target application, implementing the needed features, only.

6.1 Test Block Architecture

As explained in chapter 4, the TBUS is split in two busses: the TCB and TDB. Consequently, each TB includes a *Control Interface Block* and a *Data Interface Block* (Figure 6.1).

The Control Interface Block is in charge of managing the information coming from the TCB. Inside a Control Interface Block it is possible to identify a *Token Transmission Unit*, which sends and receives tokens on the TCB using the transmission protocol described in the sequel of this paragraph, and a *Token Execution Unit.* The execution of a token is implemented as one or more read/write operations on a register file. The primitives presented in chapter 5 are translated into elementary register operations.

The register file is organized in three different sections:

1. *Test Control Registers:* each bit of a Test Control Register is used to directly drive one of the control pins of the core like a start BIST signal;
2. *Test Status Registers:* each bit of a Test Status Register is connected to a core output and is used to read the core status during the test;
3. *Test Mode Registers:* they are usually used to set up configuration modes (see chapter 5). The content of the Test Mode Registers therefore codes the connection schema between the TDB lines and the core pins.

The number of registers and their size are customizable by the user depending on the target application.

The Data Interface Block is in charge of correctly routing the information transmitted on the TDB. Inside a Data Interface Block, each TDB line can assume two different states:

1. *Bypass*: the data received on the line is directly forwarded to the next block of the TBUS;

1. *Connect*: the data received on the line is forwarded to a core input and on the same line a core output is forwarded to the next block.

Each Configuration Mode corresponds to a different TDB routing configuration. The state of each line is obtained by properly decoding the contents of one of the Test Mode Registers.

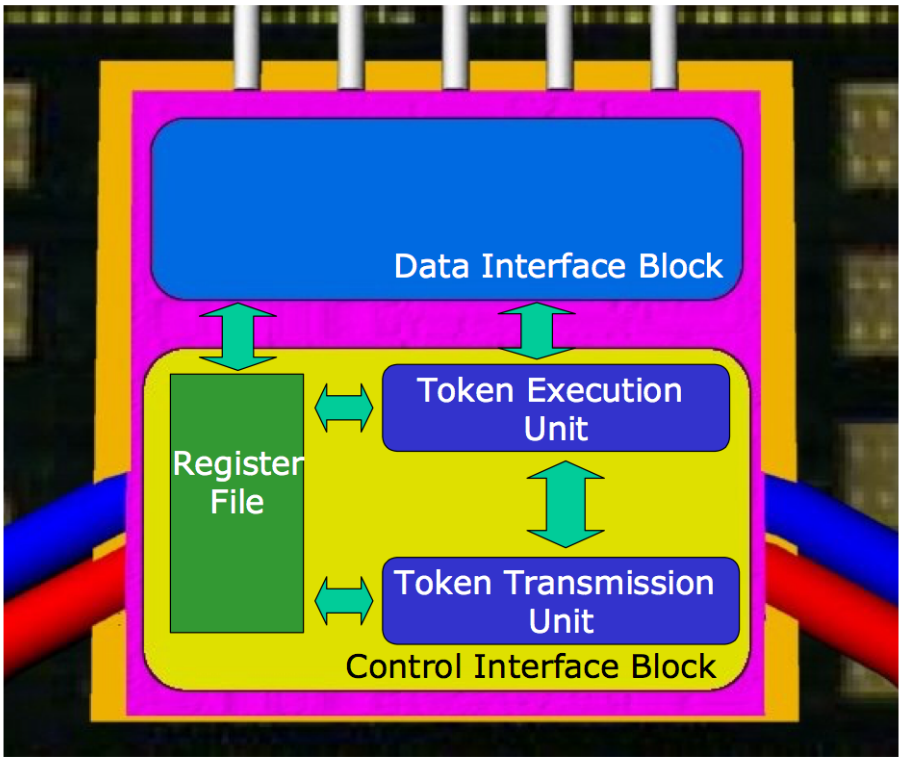


Figure 6.1. Test Block General Architecture

6.2 Test Processor Architecture

The main role of the TP is to execute Test Programs, each implemented as a sequence of tokens. Tokens and Test Patterns are sent on the TBUS through the Lower Level Interface (see chapter 4) using the same Control and Data Interface Blocks described in section 6.1. Each TP can be implemented as either an FSM statically executing the test programs, or a μ-programmed machine executing the test programs stored in a local memory.

The interface changes accordingly to the hierarchical level the TP belongs to. In particular we can distinguish between the top level and all the other hierarchical levels. At the top level, the TP is interfaced to the SoC boundaries through an IEEE 1149.1 compliant Test Access Port (TAP), which makes possible to access all the SoC test structures via an external ATE using a standard protocol. In all the other cases, when the TP is used to connect two different hierarchical levels, the interface to the upper TBUS is implemented as in a normal TB, and a Routing Unit is used to directly connect the lines of the two TDBs (if this is required by any of the Configuration Modes). The Test Control, Data, and Configuration Registers are used to configure, control and read the test results form the Lower Level Interface.

6.3 IEEE 1149.1 interface

Providing the HD^{2}BIST with a JTAG IEEE 1149.1 Test Access Port (TAP) interface is necessary to control the top level TP using a standard protocol and to simplify the use of an external ATE to test full or partial scan cores. Mainly and IEEE 1149.1 compliant TAP defines two signal named TDI and TDO used to access in a serial way a set of user defined JTAG registers.

As mentioned in chapter 5, from the external user point of view, an HD^{2}BIST SoC can be used in two different functional modes:

- *Internal Mode*: tokens and test data in general are generated on-chip. The IEEE 1149.1 interface is only used to start the execution of the Test Programs and to read out the test results. The test program can be selected by loading an appropriate value in a JTAG register. At the end of the test, the results are loaded into a JTAG Register in the form of a status word that can be scanned out through the TDO pin.
- *External Mode*: in this configuration mode the TCB is controlled using the TDI and TDO signals. The tokens to be transmitted on the TCB are loaded into a JTAG Register. In this situation the TDB is controlled using a set of dedicated input lines usually referred to as scan-in and scan-out pins. Directly controlling the TBUS, the Top Level TP is conceptually substituted by the external ATE. This configuration mode is particularly useful for diagnosis and debugging phases.

The two functional modes are not mutually exclusive. They can be combined together to improve flexibility.

6.4 Scan Chain Router

When the number of data lines needed to test a core is not compatible with the number of lines of the TDB, a *Scan Chain Router* can be placed between each TB and the corresponding core. This block is able to merge a set of core lines into a single one, thus reducing the number of TDB lines needed during the test. This block is particularly useful to test full-scan cores [1]. Introducing a Scan Chain Router, the test engineer can trade off between the routing, area overhead (number of TDB lines) and test time (long scan chain). The Scan Chain Router configuration can be set using a dedicated Test Mode

Register. The same approach can be adopted inside a TP connecting two hierarchical levels with different TDB width.

6.5 Test Control Bus

While the data is transmitted on the TDB using a simple scan protocol, tokens exchanged on the TCB require a more complex transmission protocol.

The Token Transmission Protocol is designed to make the transmission more tolerant to hardware faults. The protocol is based on the implementation of the TCB as a bi-directional link named *Forward Bus* and *Backward Bus,* respectively. The Forward Bus is used to actually transmit the token, whereas the Backward Bus is used to send the received data back to the sender in order to check its correctness and detect a wide range of transmission errors. In case of inconsistency, the block enters an *Error State* and blocks all future transmission. Using an appropriate diagnostic routine, the TP connected to the faulty bus can locate the fault, save this information into an appropriate Test Status Register, and make it accessible from the upper level of the hierarchy. In addition, an optional CRC can be inserted in each token to improve the token transmission reliability. On both the forward and the backward bus, tokens are transmitted in a serial way using a start/stop bit protocol. Each sender can transmit only one token at a time and, after the transmission, it has to wait for the token to come back before removing it form the bus and sending a new one.

6.6 Token Format

The HD^{2}BIST does not define a fixed token format, the only requirement being the implementation of a set of token fields needed to support the HD^{2}BIST basic functionalities. The required fields are listed below:

- *OP_CODE* (Operative Code): it specifies the operation to be performed. The number of possible operations is customizable. At least a Read and a Write operation must be defined.

- *DEST_ADDR* (Destination Address): it specifies the address of the destination block. It can specify also a broadcast or a group address.

- *SOURCE_ADDR* (Source Address): it specifies the address of the sender.

- *REG_ADDR* (Register Address): it identifies the register addressed by the op-code (see Section 5.1).

- *MASK*: It is a bit mask. In the case of a WRITE operation, it specifies the value to be written in the register; for a READ operation it carries the result of the read operation. The length of this field depends on the maximum size of the registers present in the TBs.

- *CRC* (Circular Redundancy Code): It is an additional optional field for inserting a CRC code (often a simple parity code) to improve the token transmission reliability.

Chapter 7

Using HD^{2}BIST in SoC Testing

The HD^{2}BIST architecture allows testing SoCs, which integrate IP cores characterized by different test strategies. For each IP the HD^{2}BIST architecture has to provide a way to transport test patterns to the core, and test responses from the core itself to the pattern sink. The solution offered by HD^{2}BIST relies on the configuration features of the TBUS. As explained before, the *SETENV* test primitive allows configuring the TDB lines in order to create a direct path between any pair of blocks connected to a TBUS, even not necessarily belonging to the same hierarchical level. The path configuration is executed using the token-based protocol of the TCB.

In general, testing a complex SoC usually requires facing four different test scenarios: BISTed or BIST-Ready cores, Scan-cores, and glue logic. The following sections explain how each of them can be addressed using the HD^{2}BIST architecture.

7.1 Testing BISTed and BIST-Ready Cores

The situation of a BISTed core, i.e., a core embedding all the logic needed to execute its test and provide the test results can be addressed using the TCB only. The TP configures the core wrapper to isolate the core under test, and write the proper values in the TB registers to directly control the test pins of the core able to activate the BIST procedure.

In the simple case of a system embedding BISTed cores only, the HD^{2}BIST may be limited to a TCB, a TP and a TB for each core. Nevertheless, at the end of the test session, BISTed cores often release not only a Boolean result, but in case of failure they allow downloading a set of diagnosis data. In this case the TDB can be used to route this information outside the SoC.

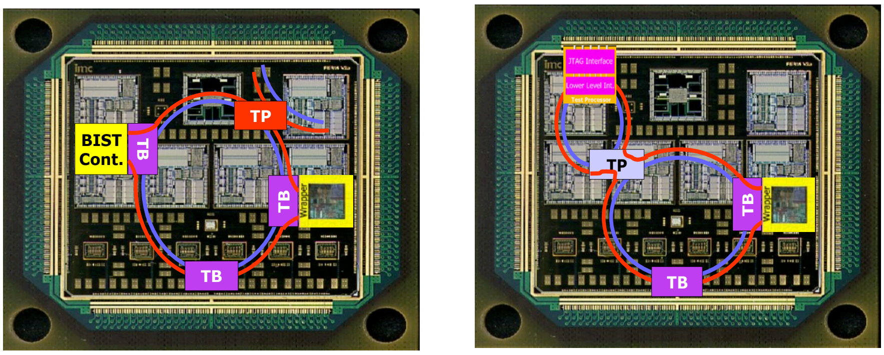
In a different way, a BIST-Ready core is not entirely self-testable; it requires an external BIST Controller to generate and apply the test patterns and to verify the correctness of the core responses. Typical examples of BIST-Ready cores are RAM memories, whose BIST logic is usually implemented in a separate BIST Controller executing a March test [69]. In this scenario, HD^{2}BIST has to enable a direct path from the BIST Controller outputs to the memory inputs and from the memory outputs to the BIST Controller inputs (Figure 7.1-a). This task is achieved by defining a Test Program able to configure a direct connection over the TDB between the BIST Controller and the core I/Os. At this point, the test can be executed at-speed without any modification in the test protocol, the only test time overhead being the initial and final latency due to the bypass TBs on the path.

When the SoC integrates more than one copy of the same core, it is possible to exploit the HD^{2}BIST to reuse the same BIST controller to test all the identical cores. With two different Test Programs it is possible to configure the connection between the BIST Controller and, one after the other, the different instances of the core.

7.2 Testing Scan Cores

Full scan cores need test patterns applied from external test equipment. They represent the most difficult situation, especially when they are deeply inserted in the hierarchy of the SoC. This is the scenario in which the hierarchical structure of the HD^{2}BIST is efficient to create a direct connection between the SoC boundaries and the cores I/O (Figure 7.1-b). The Top Level TP is set in External Mode and acts as a TDB line router, allowing the connection between TDB lines belonging to different hierarchical levels. After a configuration phase, it is possible to test the core applying the patterns directly from the SoC boundaries.

The original test patterns for the core do not need to be significantly modified, and the only test time overhead introduced is caused by the latency required to traverse the HD^{2}BIST hierarchy.



(a)

(b)

Figure 7.1. Using of TDB to test different types of cores

7.3 Testing glue logic

It is not unusual that the chip integrator adds glue logic around the cores embedded in the system. This logic is usually not localized and cannot be wrapped or considered as a core. Glue logic is usually tested using a full-scan approach . If it is possible to split the glue logic into different domains, each referring to a certain TBUS hierarchical level, the glue logic scan chains can be directly connected to a TP and considered as pseudo TDB lines. In this way it is possible to test the glue logic with the same approach used to test full-scan cores (Figure 7.2).

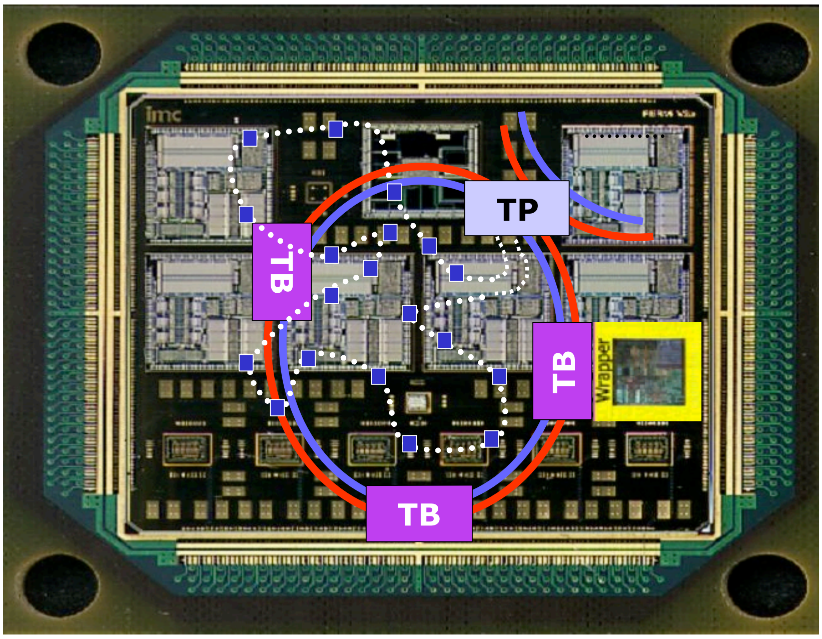


Figure 7.2. Pseudo TDB lines for testing the glue logic

Chapter 8

Case study

To demonstrate the effectiveness of the proposed TAM, the HD^{2}BIST architecture has been implemented on an industrial case study. The example has been chosen to underline the flexibility of the proposed approach in terms of *reusability* of the test structures in a hierarchical architecture, *test patterns delivering* for full-scan cores, *scheduling* of BIST-ready blocks, and *integration* of the HD^{2}BIST architecture with BIST structures automatically generated using commercial tools.

The following sections analyze the case study, providing experimental results in terms of area and test time overhead.

8.1 DacTOPplus Architecture

The case study, named *DacTOPplus,* is a LSI Logic® circuit used in transmission devices. DacTOPplus is composed of four identical macro-cores (*DacTOP*) and two BISTed RAMs (8192x8) (Figure 8.1).

Each DacTOP macro is composed of four sub-modules: one transmission macro-cell NDS_TX, one receiving macro-cell NDS_RX and two identical NDS macro-cells.

The NDS_RX, NDS_TX macro-cells are full-scan modules with seven scan chains, whereas the two NDS modules have been considered as glue-logic, and all their flip-flops are connected through a single scan chain. The circuit is realized using the G11 LSI Logic® library. Table 8.1 reports the area occupied by the test case in Synopsys® Equivalent Gates [69].

In the sequel, we will focus first on the test structure implemented in the DacTOP macros, and then on the complete test case, including the four DacTOP macros as well as the two BISTed memory modules.

8.2 DacTOP Test Structure

The test structure implemented in each DacTOP macro is composed of a single HD^{2}BIST chain controlled by a TP. The NDS_TX and the NDS_RX macros, packaged by a P1500-like wrapper, are controlled by two TBs, whereas the NDS modules are treated as glue

Table 8.1. DacTOPplus dimensions in Synopsys® equivalent gates

CORE	Equivalent Gates
NDS	99,801
NDS_RX	102,688
NDS_TX	102,802
DacTOP	430,356
BISTed RAM	163,694
DacTOPplus	2,048,814

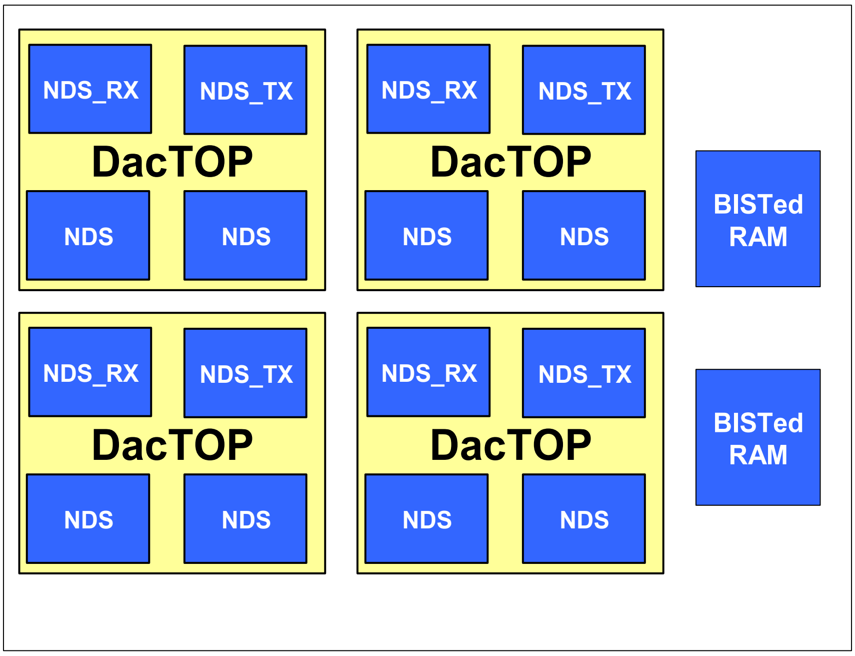

NDS_RX
NDS_TX
DacTOP
NDS
NDS
NDS_RX
NDS_TX
DacTOP
NDS
NDS
BISTed RAM
NDS_RX
NDS_TX
DacTOP
NDS
NDS
NDS_RX
NDS_TX
DacTOP
NDS
NDS
BISTed RAM

Figure 8.1. DacTOPplus schema

logic, and therefore directly controlled (or tested) by the TP. The HD^{2}BIST structure inserted in each DacTOP macro is therefore composed of (Figure 8.2):

- One Test Bus split into:
  - One *Test Control Bus* 1-bit wide.
  - One *Test Data Bus*. Since each module has 7 scan chains and the TDB has to transmit the Scan Enable and the Reset signals, the TDB should be composed of nine lines.
- Two *Test Blocks (TBs)*. Each *Test Block* is able to implement three connection modes:

  - *Bypass mode*, where the TDB is in bypass mode;
  - *Connect mode*, where the TDB is connected to the scan chains and the scan patterns can be delivered to the module;
  - *Glue mode*, where the core wrapper is set to isolate the core and used trough the TDB to apply test patterns to the glue logic placed at the core boundary.

- One *Test Processor (TP)* implementing three test programs PROG[1-3], used to connect the NDS_RX, NDS_TX, and the two NDS macros respectively, to the *Test Data Bus*. Each program sets a different target block in Connect mode and the others in Bypass mode. The Test Processor implements three connection modes:
  - *Bypass* mode where the upper TDB controlled by the TP is in bypass mode (see chapter 5);
  - *Connect* mode where the upper level TDB is connected with the lower one;
  - *Glue* mode, where the TP creates a direct path from the outside to the scan chain connecting the glue logic;

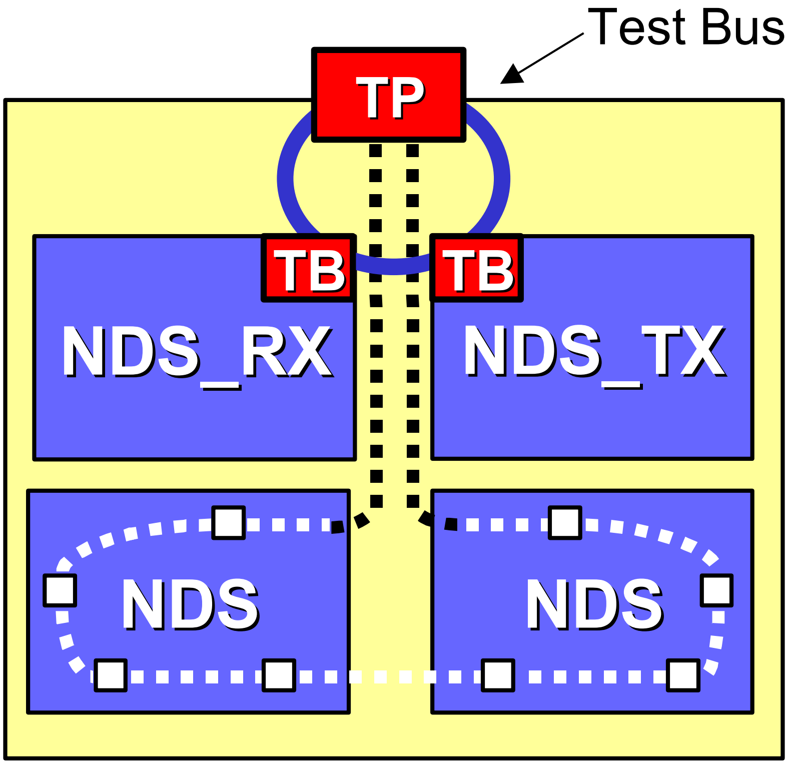


Figure 8.2. DacTOP HD^{2}BIST schema

Tables 8.2 and 8.3 report the area obtained synthesizing the DacTOP test case and the HD^{2}BIST architecture using the G11 LSI Logic® library.

Table 8.2. DacTOP area with wrapped modules

CORE	**Equivalent Gates**
Glue Logic	199,602
Wrapped NDS_RX	112,049
Wrapped NDS_TX	110,976
DacTOP with wrappers	447,891

Table 8.3. HD^{2}BISTed DacTOP area occupation

CORE	**Equivalent Gates**
TB of NDS_RX	3,695
TB of NDS_TX	3,701
TP	6,145
HD^{2}BISTed DacTOP	461,434

The area overhead of the HD^{2}BIST structure w.r.t. the original DacTOP area is the 7.03%, whereas the overhead w.r.t. the DacTOP area including the wrappers is 2.97%. We included the wrapped version of the DacTOPplus since we consider the wrappers a test requirement independent from the HD^{2}BIST structure.

8.3 DacTOPplus Test Structure

The test structure inserted in the DacTOPplus test case is composed of one HD^{2}BIST chain at the top level, and one HD^{2}BIST chain for each DacTOP module. No modifications are necessary to reuse the test architecture implemented in each DacTOP macro at the top-level. The top level chain is built of the following blocks (Figure 8.3):

- One *Test Bus* split into:
  - One *Test Control Bus* one line wide;
  - One *Test Data Bus* 9 lines wide (each *DacTOP* module needs nine lines, whereas the BISTed RAMs do not need any data line);
- One *Test Block* for each RAM;
- Four *Test Processors*, one for each DacTOP macro, as described in the previous section;

- One *Top Level Test Processor* with a IEEE 1149.1 interface. In the Test Processor we implemented 13 different test programs, which can be executed in any desired order:
    - PROG[1]: it starts the BIST of the two RAMs, waits for the BIST to end, and reads the test results;
    - PROG[2-4]: they start respectively PROG[1-3] of the first DacTOP and they wait for their end. They then connect the Scan In of the TAP interface with the first DacTOP in order to scan out the test results;
    - PROG[5-7]: the same as PROG[2-4] but with the second DacTOP module;
    - PROG[8-10]: the same as PROG[2-4] but with the third DacTOP module;
    - PROG[11-13]: the same as PROG[2-4] but with the fourth DacTOP module;

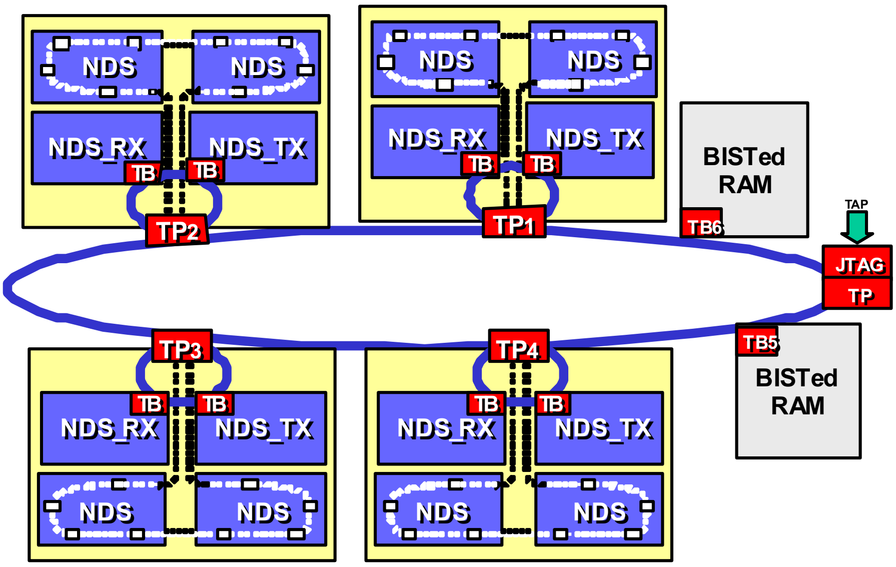

NDS
NDS
NDS_RX
NDS_TX
TB
TB
TP2
NDS
NDS
NDS_RX
NDS_TX
TB
TB
TP1
BISTed RAM
TB6
TAP
JTAG
TP
TP3
TP4
TB5
BISTed RAM
TB
TB
NDS_RX
NDS_TX
NDS
NDS
TB
TB
NDS_RX
NDS_TX
NDS
NDS

Figure 8.3. DacTOPplus with HD^{2}BIST

Table 8.4 reports the area obtained synthesizing the DacTOPplus using the G11 LSI Logic® library.

The area overhead of the HD^{2}BIST structure w.r.t. the original DacTOPplus area is 6.61%, whereas the overhead w.r.t. the DacTOPplus area including the wrappers is 3.06%.

8.3.1 Running a test program

To show how it is possible to actually exploit the HD^{2}BIST architecture to run the system test, we detail three different test programs that target the BISTed RAMs, one DacTOP

Table 8.4. Area result of DacTOPplus

CORE	**Equivalent Gates**
HD^{2}BISTed DacTOP	461,434
TB of RAM	2,956
TP_TAP	5,958
HD^{2}BISTed DacTOPplus	2,184,997

NDS_RX module, and the glue logic. Each program can be run loading the corresponding code into a dedicated JTAG register.

Figure 8.4 presents PROG[1] of the top-level DacTOP, which activates the test of the two BISTed RAMs. The program activates the BIST procedures of the two BISTed RAMs and then starts polling the two Test Blocks waiting for the end of the test. Test programs allows a very flexible implementation of any test scheduling: PROG[1] executes the BIST of the two memories in parallel, but, by simply exchanging instruction #3 and #4, it is possible to test the two memories sequentially.

```
Program PROG[1]
{
Conf ALL,BYPASS /* Configure all the TP/TB in BYPASS
mode since the TDB is not used during BIST
phase */
Start TB5 /* Start the first RAM BIST by sending a
start primitive to TB1*/
Start TB6 /* Start the second RAM BIST */
Wait ALL /* Wait for the end of all the BIST.
This primitive is Implemented using a
polling mechanism */
Collect ALL /* Read the BIST results contained in the RAM
TBs and store them in The Top Level TP.
In case of fault the faulty Block can
be located using the external mode to
have Direct access to the TBs and BIST
controllers. */
}
```

Figure 8.4. Test program PROG[1] of the Top Level TP

The second example is PROG[2] of the top-level TP (Figure 8.5), which creates a path on the TDB to directly connect the top-level TP to the NDS_RX module of the first DacTOP. After properly configuring the top-level chain, the TDB of the first DacTOP module is connected to the top-level TDB to allow the top-level TP to start the execution of PROG[1] in the first DacTOP (Figure 8.6). PROG[1] of the DacTOP macro, creates a path from the scan chains of NDS_RX to the Test Data Bus lines. As soon as PROG[1] is completed a path has been set from the top-level to the addressed block in the hierarchy, and the test patterns can be applied using the scan signals.

```
Program PROG[2]
{
Conf ALL,BYPASS /* Configure all the TB/TP in BYPASS mode */
Conf TP1,Connect /* The TP of the first DacTOP macro is set
in Connect mode, to connect its TDBus to
the top level TDB */
Start TP1,PROG[1]/* Start the execution of PROG[1] of TP1 */
Wait TP1 /* Wait for the end of PROG[1] of TP1.
From this moment the NDS_RX block of the
first DactTop is accessible from the top
level.*/
}
```

Figure 8.5. Test program PROG[2] of the TLTP

```
Program PROG[2]
{
Conf ALL,BYPASS /* Configure all the TB/TP in BYPASS mode */
Conf TB1_1,Connect /* The first TB of the chain is set in
Connect mode, to connect the scan chains of
NDS_RX to the TDB */
}
```

Figure 8.6. Test program PROG[1] of the DacTOP macro

In the last example we detail PROG[4] of the top-level TP (Figure 8.7), which enable the test of the first DacTOP glue logic (the two NDS macros) and interconnections. After configuring the top-level chain, the TP of the first DacTOP macro is configured to directly control the scan-chain connecting the glue-logic of the module. Moreover, PROG[3] of the DacTOP macro (Figure 8.8) is executed to set all the wrappers of the macro in ExtTest mode, so that the values scanned in their scan chains can be applied to the input of he glue logic.

```
Program PROG[4]
{
Conf ALL,BYPASS /* Configure all the TB/TP in BYPASS mode */
Conf TP1,glue /* TP1 is set in Glue mode, i.e., it is
configured to directly control the glue logic
scan chain. In particular, the first two lines
of the top-level TDB are connected to the Scan
In/Scan Out signals of the glue logic chain,
and the third line is connected to the first
line of the lower Test Data Bus to create
a scan path through the wrappers, necessary to
apply test patterns to the glue logic
boundaries */
Start TP1,PROG[3]/* Start the execution of PROG[3] of TP1 */
Wait TP1 /* Wait for the end of PROG[3] of TP1. From
this moment the patterns to test glue logic
can be applied.*/
}
```

Figure 8.7. Test program PROG[4] of the Top Level TP

```
Program PROG[3]
{
Conf ALL,Glue /* Configure all the TB in glue mode
allowing to apply test patterns trough
the core wrappers*/
}
```

Figure 8.8. Test program PROG[3] of the DacTOP macro

Chapter 9

Conclusions

In this part of the thesis the problem of test and diagnosis of complex SoC designed using embedded cores has been addressed. The proposed solution HD^2BIST, is a complete hierarchical framework to support the definition of the scheduling strategies and data patterns delivering mechanisms of the embedded cores of a complex system. The main goal of the HD^2BIST architecture is to maximize and simplify the reuse of the built-in test architectures giving the chip designer the highest flexibility in planning the overall SoC test strategy. HD^2BIST defines a TAM able to provide a direct “virtual” access to each core of the system, and can be conceptually considered on a higher level w.r.t. the P1500 standard, whose main target is to make the test interface of each core independent from the vendor. A complex case study has been presented to demonstrate the effectiveness of the approach in terms of complexity and area overhead.

Part II

Memory Dominance

Related publications

Portions of the material described in this part of the thesis has been subject of following scientific publications: [14],[27],[7],[10], [9], [20], [13].

Chapter 10

Introduction

SRAMs are nowadays widely used as embedded memories in a plenty of SoCs. As an example, Mlti-port SRAMs are basic elements of telecommunications or multiprocessor systems. They allow to speed-up the system, in particular when the memory has to serve many concurrent requests. Today's technologies allow the design and manufacturing of memory chips up to 11 ports, and Multi-port RAM generators are commonly available in many vendor libraries as LSI-Logic, Texas Instruments and ST Microelectronics.

Memory-BIST is today a pressing issue. The test engineer has to define test strategies for complex SoCs including several multi-port SRAMs of different size (number of bits, number of words), access protocol (asynchronous, synchronous), and timing. Apart from the required design time, there are several issues to be solved, including minimizing BIST area and routing overhead, selecting the proper number of BIST controllers, fulfilling power budget constraints, supporting diagnostic capabilities and defining the best test algorithm with respect to the target technology and application. Among the different types of algorithms proposed to test random access memories (RAM), March Tests have proven to be faster, simpler, regularly structured and linear in complexity [69]. March Tests are able to cover a wide range of memory faults, but due to the technology advance, new types of faults arise and should be tested. Different March Tests of variable complexity have been proposed in literature, each optimally covering a different set of memory faults. All of them have been manually generated, a task that requires a lot of time, expertise, that does not always allow to obtain an optimal solution, and that sometimes does not succeed in covering particularly complex memory faults. An automatic approach to generate test algorithms to be used in memory-BIST architectures is a pressing issue in complex SoC design and test. These new algorithms have to be always validated. Memory fault simulation is therefore necessary to compute the Fault Coverage of a test sequence every time a new defect is discovered and the corresponding fault model defined. Computing and limiting the test application power consumption is also another important issue, especially when the memory test is implemented as a BIST procedure.

An emerging trend in memory design is to have not only testable memory but also repairable memories. The emerging field of Self-Repair Computing is expected to have a major impact on deployable systems for space missions and defense applications, where high reliability, availability, and serviceability are needed; but also on new commercial applications were high level of availability is a must.

In the following of this chapter, each of the proposed issues will be analyzed and a set of solutions will be presented.

Chapter 11

An introduction to memory Test

This chapter introduces the basic concepts and background on memory testing. It is a short overview to help the reader understanding the following chapters.

11.1 The fault model

Memories are usually tested using a functional approach. The functional model of a simple SRAM is made of the following units: (i) the memory cell array, (ii) the address decoder and (iii) the read/write logic. Memory faults can be classified depending on their location.

11.1.1 Memory Cell Faults

Memory cell faults can be split in two categories: *single cell faults* and *multiple cell faults*. Single cell faults are restricted to a single cell of the memory array and are classified as follow:

- *Stuck-at faults (SAF)*: the logic value of a cell is always fixed to a certain value. It can be a stuck-at zero (SAF-0) if the fixed value is ‘0’ or stuck-at one (SAF-1) if the fixed value is ‘1’;
- *Transition faults* (TF): a cell fails to perform a transition from ‘0’ to ‘1’ or vice versa.

In multiple cell faults the logic value of one or more cells is influenced by the logic value of one or more others cells. The former is called the coupled cell(s) whereas the latter is called the coupling cell(s). Coupling faults occur due to capacitive linkage of storage cells. It should be noted that these faults are not bi-directional. Some of the commonly occurring coupling faults are:

- *Idempotent coupling faults (CFid)*: they occur when a transition in one cell forces the logic value of the coupled cell to a fixed value;
- *Inversion coupling fault (CFin)*: a transition in the coupling cell inverts the logic value of the coupled cell.

11.1.2 Address decoder fault

Faults in the decoder circuitry of a memory chip manifest as follows:

- The decoder may not access the addressed cell;
- The decoder accesses a non-addressed cell;
- Multiple cells may be accessed with the same address;
- A cell is accessed by multiple addresses.

It is always possible to map address faults into one or more memory cell array faults.

11.1.3 Read/Write logic faults

The R/W logic may have stuck at 1/0 faults that are mapped into the cell array. These faults may arise when one or more of the I/O pins of the memory array are stuck-at or stuck-open.

11.2 Memory test Algorithms

Among the different types of algorithms proposed to test random access memories (RAM), March Tests have proven to be faster, simpler, regularly structured and linear in complexity [69]. A March Test consists of a sequence of *March Elements*, each composed of a sequence of basic operations to be performed on each cell of the memory, in either ascending ($\Uparrow$) or descending order ($\Downarrow$), before proceeding to the next cell. The possible memory operations are:

- *w0:* write the logic value '0' in the target cell;
- *w1:* write the logic value '1' in the target cell;
- *r0*: read the content of the target cell and verify that it is equal to '0';
- *r1*: read the content of the target cell and verify that it is equal to '1'.

A march-element is usually represented by the sequence of operations in brackets. An example of March Test able to detect SAF composed by three march element is $\{(\Uparrow w_1)(\Downarrow r_1 w_0)(\Downarrow r_0)\}$.

Chapter 12

Memory BIST

Commercial tools are nowadays available for the automatic insertion of memory-BIST in complex SoCs [52], [57]. Nevertheless, they are not able to efficiently deal with complex situations where Multi-port SRAMs of different sizes (number of bits, number of words), access protocol (asynchronous, synchronous), and timing coexist in the same design. This paragraph proposes a new memory-BIST architecture characterized by a single *BIST Processor*, in charge of testing all (or a subset of) the SRAMs of the SoC. It is based on a μ-programmable architecture, executing elementary test primitives stored in a dedicated memory and a set of *Wrappers* placed around each memory (Figure 12.1).

Each wrapper is composed of a set of *Port-Wrappers* (one per memory port) and a *Dispatcher*. The Port-Wrapper contains the standard blocks to perform a memory-BIST (i.e., an address generator, a pattern generator, and a comparator), and an interface block designed to manage the communications between the SRAM and the BIST Processor, regardless the memory access protocol. The Dispatcher is a simple Finite State Machine used to serially collect the test primitives and to deliver them to the various Port-Wrappers. A minimal set of communication signals allows the BIST Processor to execute and synchronize the test algorithm of all the memories under test, whereas a scan chain connecting all the Port-Wrappers allows full diagnosis capabilities.

The proposed schema presents several advantages. Among the others, it allows running concurrently the BIST of a set of SRAMs with different number of ports, size, accessing protocol and timing and it allows to freely select the set of memories to be tested, using either ad-hoc-test primitives stored in the test program, or a dedicated scan chain to properly set an ad-hoc status bit in each memory.

The use of a single BIST controller and a minimum set of communication signals allows minimizing the BIST area overhead and the connectivity around each SRAMs whereas the μ-programmable architecture of the BIST Processor provides the test engineer a flexible and reusable block, that can be used to manage the BIST of any number of memories regardless the used test algorithm.

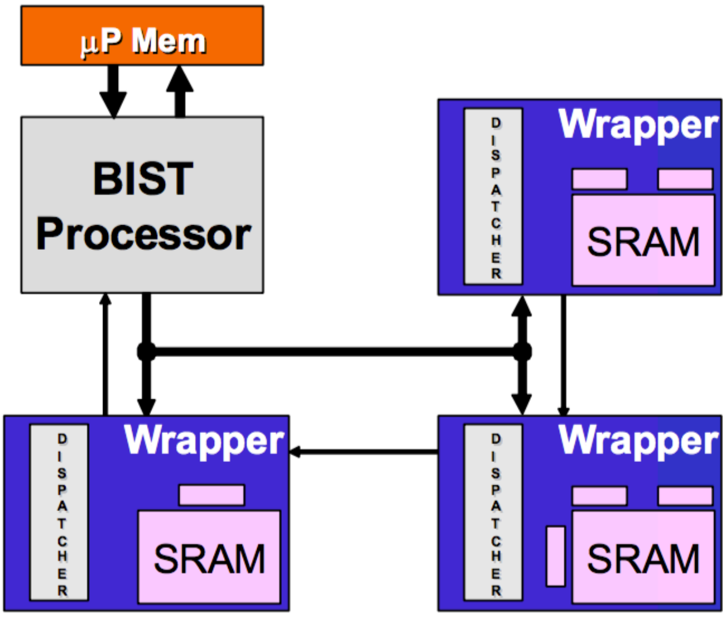


Figure 12.1. Basic memory-BIST Architecture

12.1 The BIST Processor

As introduced in the previous section, the proposed memory-BIST uses a single BIST Processor to test all the memories of the system. To increase flexibility, it is implemented as μ-programmable machine. The *test algorithm* (a March Algorithm [69][44]) is stored in a dedicated *μProgram-Memory* and it is coded through a set of *test primitives*. The μProgram-Memory can be either a ROM or an In-System Programmable module. In the former case, the test program is fixed at design time, whereas in the latter one a custom and appropriate test algorithm can be loaded into the memory at test time.

The BIST Processor reads from the *μProgram-Memory* one test primitive at a time, forwards it to all the Wrappers of the SRAMs under test, and waits until its completion in all the target memories.

When the test program is completed (i.e., all the test primitives have been applied), the BIST Processor reads the test results from each RAM. If a fault is detected, the faulty RAM is located resorting to a set of diagnostic facilities explained in the following sections.

The BIST processor and the μProgram-Memory architectures are optimized for the characteristics of March Tests used in multi-ports memory test. The main characteristic of these algorithms is the access to the ports of the memories using nested cycles:

$$\{\Uparrow_A (\Uparrow_{B=0}^{A-1} (\Uparrow_{C=B+1}^{n} \cdots))\} \tag{12.1}$$

where $\Uparrow_{B=0}^{A-1}\Uparrow_{C=B+1}^{n}$ denotes a nested addressing sequence [59].

Table 12.1 summarizes the set of *test primitives* needed to implement such a March

Algorithm.

Table 12.1. March Algorithm Test Primitives

Primitive	**Description**
W0	Write pattern
W1	Write a not pattern
R0	Read and verify a pattern
R1	Read and verify a not(pattern)
Inc	Increment the address generator and define the end of a March Element
Dec	Decrement the address generator and define the end of a March Element
IncCond	Conditionally increment the address generator
DecCond	Conditionally decrement the address generator
Sub	Increment the address generator
Add	Decrement the address generator
Load	Load a value in the address generator
Nme	New March Element
NOP	No Operation
NextBP	Next Background Pattern
Conf	Define the set of SRAM under test
End	End of test

Each test-program step is coded in the μProgram-memory as a sequence of test primitives, one for each memory port. As an example, let's consider the following March Algorithm used to test an 8-bit dual port SRAM (the convention for the operation is (portA:portB)):

$$
\begin{aligned}
&\{\\
&\underset{M_0}{\Uparrow (w_0 : w_0)};\ \underset{M_1}{Downarrow(r_0 : r_0; w_1 : w_1)};\ \underset{M_2}{\Uparrow (r_1 : r_1)};\\
&\underset{M_3}{\Updownarrow (w_{BP0} : w_{BP0}, r_{BP0} : r_{BP0}, \ldots, w_{BP7} : w_{BP7}, r_{BP7} : r_{BP7})};\\
&\underset{M_4}{\Updownarrow (w_0 : -)};\underset{M_5}{\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=0}^{v-1} (w_{1_a} : r_{0_v}, w_{0_a} : r_{0_v}, n : r_{0_v}))};\\
&\underset{M_6}{\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=v+1}^{n-1} (w_{1_a} : r_{0_v}, w_{0_a} : r_{0_v}, n : r_{0_v}))};\\
&\underset{M_7}{\Updownarrow (w_1 : -)};\underset{M_8}{\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=0}^{v-1} (w_{0_a} : r_{1_v}, w_{1_a} : r_{1_v}, n : r_{1_v}))};\\
&\underset{M_9}{\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=v+1}^{n-1} (w_{0_a} : r_{1_v}, w_{1_a} : r_{1_v}, n : r_{1_v}))};\\
&\}
\end{aligned}
\tag{12.2}
$$

The March elements M_0-M_3 realize the MATS algorithm [69] , properly expanded as proposed in [68] to cover intra-word CFsts faults. The pattern used during the test

(BP_0-BP_7), are taken from the set of Background Patterns in Table 12.2.

Table 12.2. 8 bits Background patterns BP_j for CFsts

J	Background Pattern
0	00000000
1	11111111
2	00001111
3	11110000
4	00110011
5	11001100
6	01010101
7	10101010

The March elements M_4-M_9 represent the March 2PF2,2 proposed in [44] to test wCF_i&wCF_i. The algorithm can be coded using the set of primitives shown in Table 12.3.

An important issue to be faced when running concurrently the BIST of several modules is fulfilling power budget constraints. In fact, BIST typically results in a circuit activation rate higher than the normal one [81], and an over-dissipation of power may seriously damage the devices. Moreover, the variety of SRAMs that can be found in a complex architecture may require different test algorithms. To address these two issues, the proposed approach implements a very flexible scheduling mechanism. In particular, it is possible to select the set of memories to be tested using either a special test primitive in the μProgram-Memory, as part of the test algorithm, or setting a dedicated flag into the memory Wrapper through a scan chain. Only the Wrappers of the selected memories will execute the test primitives received from the BIST Processor. In this way, several test algorithms may be stored in the μProgram-Memory and may be applied sequentially to different sets of memories.

12.2 Wrapper Structure

The Wrapper placed around each memory has to execute the test primitives broadcasted by the BIST Processor, regardless the memory access protocol. Moreover, the Wrapper is the only element in the architecture taking care of the number of ports, the size and the access protocol of the memory it is placed around (See Figure 12.2). It is composed of two types of blocks: one Dispatcher to receive the test primitives from the BIST Processor and distribute them to the ports of the memories and a Port-Wrapper for each RAM port. The Port-wrapper generates the test patterns (address and data) and verifies the correct behavior of the memory according to the command received from the Dispatcher. The result of each primitive is signaled via an output line.

Two kinds of Port Wrappers are available: one for the first port of each memory *(FPW, First Port Wrapper)* and one for the other ports *(OPW, Others Port Wrapper)*. The main difference between the two lays in the fact that each OPW receives as an input the address

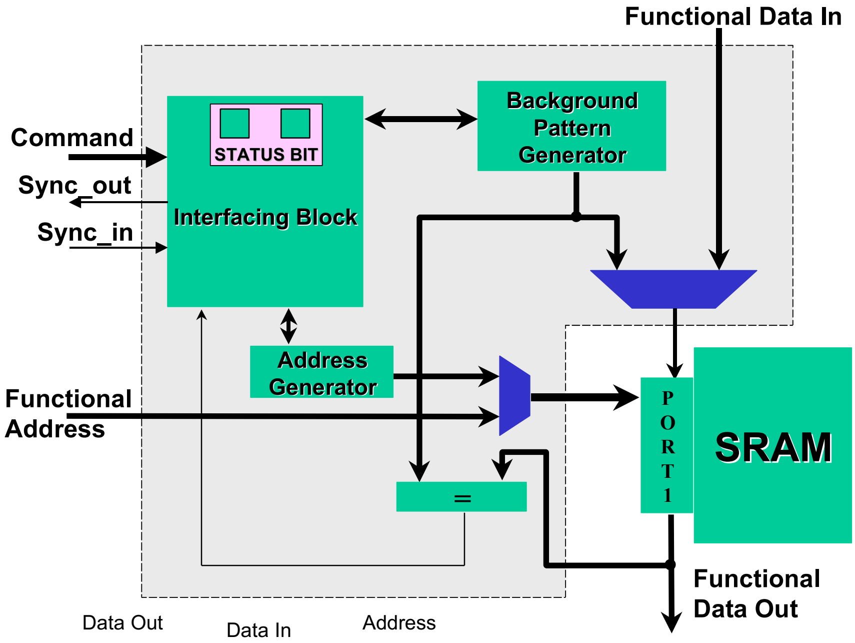

Functional Data In
Command
Sync_out
Sync_in
STATUS BIT
Interfacing Block
Background Pattern Generator
Address Generator
Functional Address
P O R T 1
SRAM
=
Functional Data Out
Data Out
Data In
Address

Figure 12.2. Wrapper architecture

value generated by the previous port wrapper.

12.2.1 Dispatcher

The dispatcher receives from the BIST Processor the test primitives for all the port wrappers. The BIST Processor sends a test command per clock cycle to all dispatchers (the first command is sent to the FPWs, the second one to the first OPWs, etc.). The dispatcher saves all the commands in a temporary register. Since each wrapper has no information about the size of the other wrappers, a run signal is sent after all the commands to start the execution.

As an example, the execution of the (W0:R0) instruction for a dual port memory is shown in Figure 12.3.

12.2.2 Port Wrapper

Figure 12.4 shows the internal structure of a FPW. The *Address Generator (AG)* is in charge of generating the address where the test pattern, provided by the *Background Pattern Generator (BPG)*, has to be written or verified. Several BPGs are available to target different fault type [68]. The correctness of the content of a memory cell is evaluated through a simple C*omparator.*

Two Status Bits are used to set the memory in *transparent* or in *test mode* (the

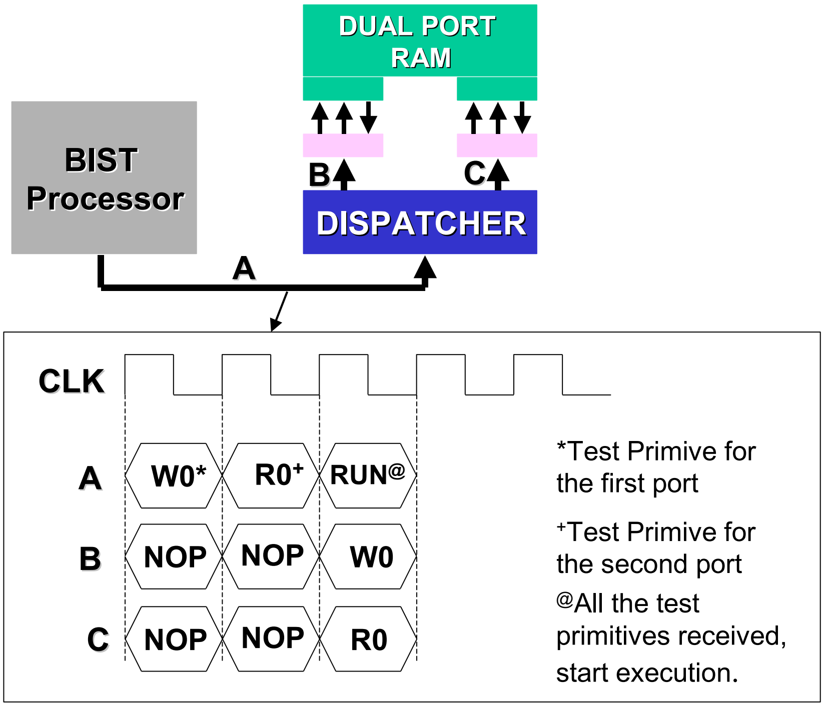

DUAL PORT RAM
BIST Processor
B
C
DISPATCHER
A
CLK
A
W0*
R0⁺
RUN@
B
NOP
NOP
W0
C
NOP
NOP
R0
*Test Primive for the first port
⁺Test Primive for the second port
@All the test primitives received, start execution.

Figure 12.3. Test Instruction execution diagram

Mode_Status_Bit) and to store the *test results* at the end of the test algorithm (the *Result_Status_ Bit*). The status bits of all Wrappers are connected by two scan chains, respectively called *NormTest_Scan_Chain* and *Results_Scan_Chain*.

Finally, each FPW includes an *Interface Block* able to receive the test primitives from the Dispatcher and a synchronization signal from the previous port wrapper. It produces the *output synchronization signals* needed by the BIST Processor to schedule the next test primitive to be executed. The *output synchronization signal* assumes different meaning depending on the received test primitive (See Table 12.4).

The structure of the OPW is similar to the FPW. In order to execute the class of March algorithm explained in section 12.1, this wrapper includes some additional blocks, since it has to generate a subset of the entire addressing space, depending on the address generated by the previous port wrapper.

12.2.3 Multiplexing

To minimize the routing overhead, the signals exchanged between the BIST Processor and the memory Wrappers (command signals, synchronization signal and scan chain signals) are multiplexed. In particular, these signals are multiplexed at the port-wrapper level. All the information is routed using 6 signals only (4 command signals and 2 synchronization signals).

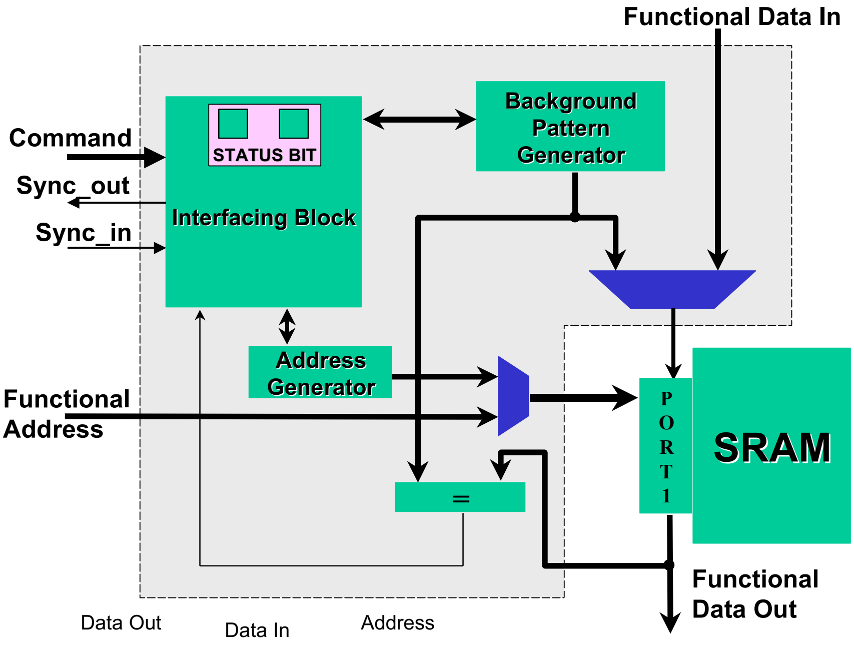

Functional Data In
Command
Sync_out
Sync_in
STATUS BIT
Interfacing Block
Background Pattern Generator
Address Generator
Functional Address
=
P O R T 1
SRAM
Functional Data Out
Data Out
Data In
Address

Figure 12.4. Wrapper structure

12.3 Diagnosis

When a fault is detected, the proposed approach allows collecting diagnostic information concerning the location of the faulty SRAM, the port where the fault has been detected, the address of the faulty cell and the detecting pattern. This information is stored into the *Result_Status_Bit*, the *Address Generator*, and the *Background Pattern Generator* of each Port-Wrapper and can be scanned-out via the *Results_Scan_Chain*. In particular, depending on the result of the test (*Result_Status_Bit*), each Port-Wrapper configures its portion of the *Results_Scan_Chain* in one of the following two ways (Figure 12.5):

- *Result_Status_Bit='1'*: the RAM is not faulty; only the Result_Status_Bit is placed on the scan chain.
- *Result_Status_Bit='0'*: the RAM is faulty; the Result_Status_Bit is chained to the content of the Address Generator and the Background Pattern Generator.

12.4 Further optimizations

12.4.1 Sharing Wrappers among SRAMs clusters

To further reduce the BIST area overhead, the designer can share a single Wrapper for a cluster of identical SRAMs (same type, width, and addressing space).

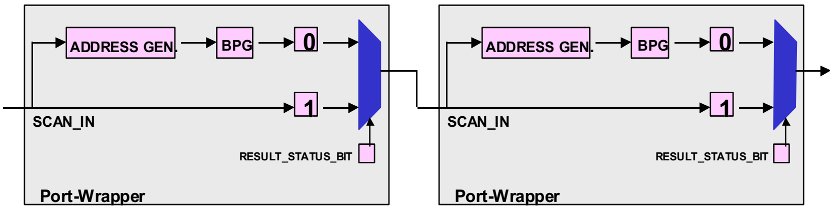


Figure 12.5. Results_Scan_Chain

This optimization is made at the Port-Wrapper level. For each Port-Wrapper only one Address Generator and one Background Pattern generator are needed. The only difference with the previously described Port-Wrapper structure is that a shared Port-Wrapper contains a pair of Status Bits and a comparator for each RAM. In this way, when a fault is detected, the Result Status Bit of the faulty memory is set, the RAM is disconnected, and the Wrapper continues the test of the remaining memories of the cluster. Obviously, in this case, the status of the Address Generator and the BPG of the faulty RAM are not preserved. To collect diagnostic information, the test must be re-executed targeting the faulty RAM, only.

12.4.2 Using a Topological approach for complex coupling fault testing

The approach proposed in this paper is useful to describe March Algorithms for multi-port RAMs with complexity of $O(n^m)$ where n is the number of cells and m the number of ports. For practical applications, these algorithms result in very long test sequences. It is possible, as proposed in [59], to optimize the address generator of each OPW to generate the address for a Topological Approach. The approach consists in detecting all coupling faults between adjacent cells only. Using this optimization the test complexity can be reduced to $O(n)$ without significant fault coverage reduction.

12.5 Case study

A case study has been used to evaluate the effectiveness of the proposed memory-BIST and to gather experimental results. The circuit, named VC12AD, is a part of a telecommunication ASIC designed by Italtel SpA. The same circuit has also been used by both Italtel SpA and Siemens ICN as a benchmark for the evaluation of commercial BIST Insertion Tools.

The target circuit has been described in VHDL and synthesized using the *G10 LSI-Logic™ library*, which provides a set of SRAMs of different sizes.

The VC12AD counts up to 860K Synopsys™ equivalent gates (excluding RAMs), plus 36 small-sized SRAMs, for a total of 14,704 bits (Figure 12.6).

The case study aims at evaluating the BIST architecture complexity when applied to a set of SRAMs with very different characteristics and the area overhead after the BIST insertion.

12.5.1 Case Study Architecture

Figure 12.6 and Figure 12.7 show a conceptual view of the VC12AD organization and its actual floor plan. The 36 SRAMs of the circuit are grouped in four distinct macro-areas whose characteristics are listed below

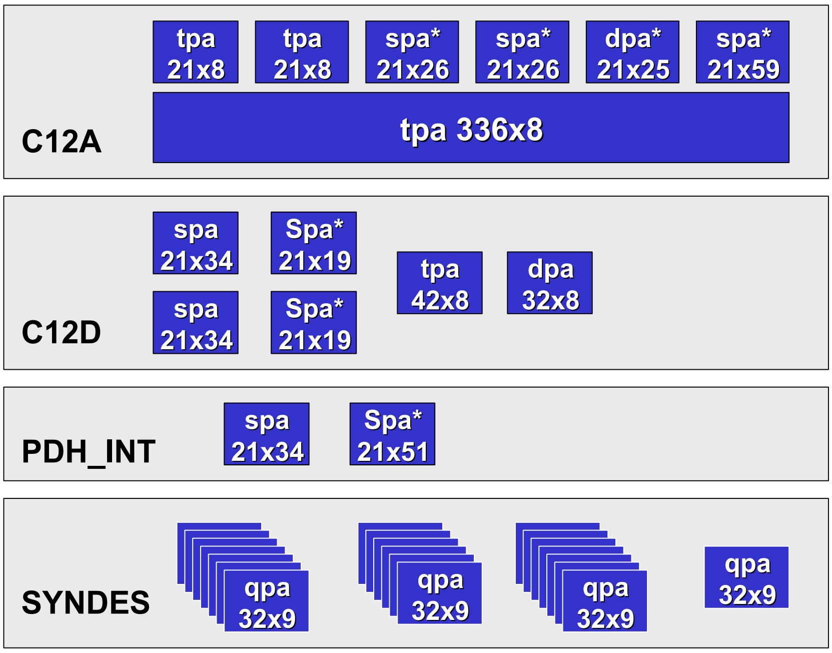

C12A
tpa 21x8
tpa 21x8
spa* 21x26
spa* 21x26
dpa* 21x25
spa* 21x59
tpa 336x8
C12D
spa 21x34
Spa* 21x19
spa 21x34
Spa* 21x19
tpa 42x8
dpa 32x8
PDH_INT
spa 21x34
Spa* 21x51
SYNDES
qpa 32x9
qpa 32x9
qpa 32x9
qpa 32x9

Figure 12.6. VC12AD memories organization

- C12A : it contains 7 RAMs[1]:

n. of instances	**Type**	**Size**(Number of words) x (bits per word)
2	tpa	21x8
2	spa*	21x26
1	dpa	21x25
1	spa	21x59
1	tpa	336x8

- C12D : it contains 6 RAMs:

---

[1] spa: single port asynchronous RAM; spa*: single port asynchronous RAM with 1 write enable for each data bit; dpa: dual port asynchronous RAM (one port dedicated to write and one dedicated to read); tpa: triple port asynchronous RAM (one port dedicated to write and two ports dedicated to read);

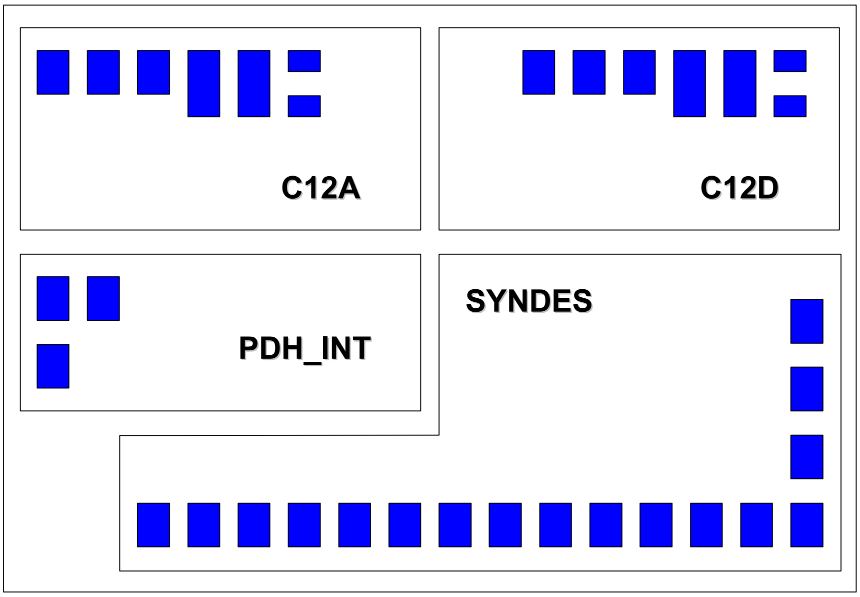


Figure 12.7. VC12AD floorplan

n. of instances	Type	Size
1	dpa	32x8
1	tpa	42x8
2	spa	21x34
2	spa*	21x19

- PDH INT : it contains 2 RAMs:

n. of instances	Type	Size
1	spa	21x34
1	spa	21x51

- SYNDES : It consists of 21 identical blocks. Each of them contains one instance of a qda32x9 (asynchronous quadruple port RAM with two ports dedicated to write and two dedicated to read).

12.5.2 Case Study BISTArchitecture

The number of wrappers is minimized resorting, whenever possible, to clusters of SRAMs. As a consequence:

- Within C12A, the 2 modules tpa21x8 and the 2 modules spa*21x26 are treated as two clusters;

- Within C12D, the 2 modules spa21x34 and the 2 modules spa* are treated as two clusters
- Within SYNDES, the memories are organized as four clusters of 7, 7, 6, and 1 element, respectively.

The design of the BIST architecture has been strongly influenced by the actual floor plan, where, for example, the 3 spa21x34 SRAMs (2 located inside C12D and 1 in PDH_INT) are too far to be included in a single *cluster.*

The overall VC12AD structure after the BIST insertion is in Figure 12.8.

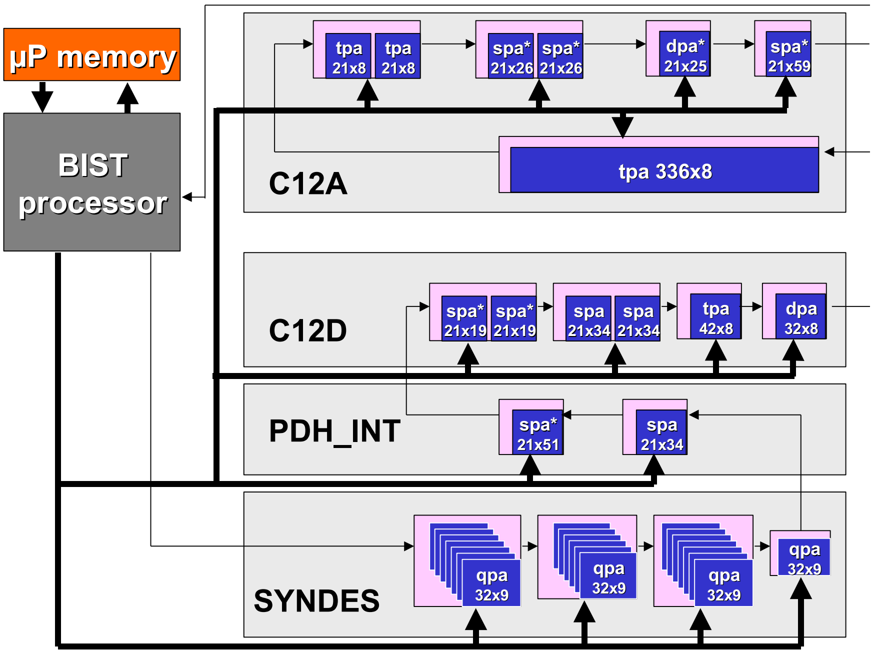

µP memory
BIST processor
C12A
tpa 21x8
tpa 21x8
spa* 21x26
spa* 21x26
dpa* 21x25
spa* 21x59
tpa 336x8
C12D
spa* 21x19
spa* 21x19
spa 21x34
spa 21x34
tpa 42x8
dpa 32x8
PDH_INT
spa* 21x51
spa 21x34
SYNDES
qpa 32x9
qpa 32x9
qpa 32x9
qpa 32x9

Figure 12.8. VC12AD BIST Architecture

12.5.3 Case Study BIST Scheduling

Due to the different characteristics of the VC12AD SRAMs (read/write ports, read-only ports, and write-only ports are present), it is not possible to adopt a unique March Algorithm for the overall circuit. The BIST has been therefore organized in four sessions, each one using an appropriate March algorithm:

- Session 1: All single port RAMs are tested concurrently;
- Session 2: All dual port RAMs are tested concurrently;
- Session 3: All triple port RAMs are tested concurrently;

- Session 4: All quadruple port RAMS are tested concurrently.

12.5.4 Experimental results

The area occupation of each memory and its Wrapper is in Table 12.5, whereas Figure 12.9 shows the contributions of the functional blocks of each Wrapper.

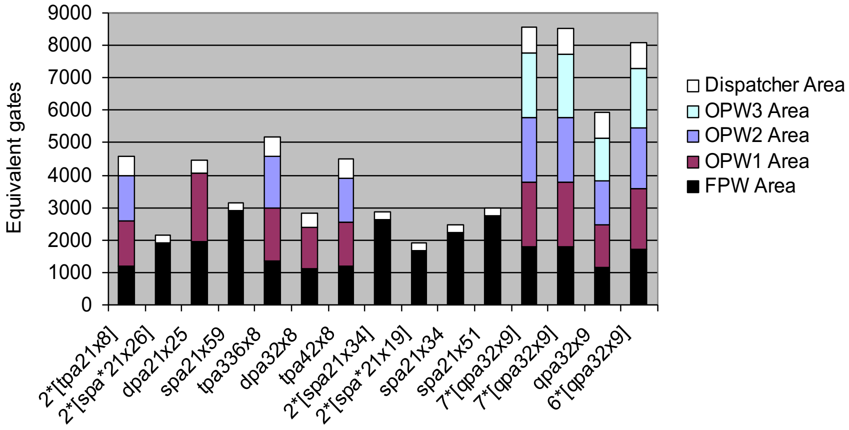


Figure 12.9. Wrappers area

The total area overhead including the Wrappers and the BIST Processor is in Table 12.5.

As shown in Table 12.5, the BIST processor and the µProgram-memory area overhead is a fix contribution and it is not influenced by the number of SRAMs in the system.

Table 12.3. March Algorithm Representation

March Element	**Primitive**	
	Port A	**Port B**
$\Uparrow (w_0 : w_0)$	NME	NME
	INC	INC
	W0	W0
$\Uparrow (r_0 : r_0, w_1 : w_1$	NME	NME
	R0	R0
	W1	W1
	DEC	DEC
$\Uparrow (r_1 : r_1)$	NME	NME
	R1	R1
	INC	INC
$\Updownarrow (w_{BP0} : w_{BP0}, r_{BP0} : r_{BP0}, \cdots, w_{BP7} : w_{BP7}, r_{BP7} : r_{BP7})$	NME	NME
	W0	W0
	R0	R0
	W1	W1
	R1	R1
	NEXTBP	NEXTBP
	INC	INC
$\Updownarrow (w : 0 : -)$	NME	NOP
	W0	NOP
	INC	NOP
$\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=0}^{v-1} (w_{1_a} : r_{0_v}, w_{0_a} : r_{0_v}, n : r_{0_v}))$	NME	NME
	W1	R0
	W0	R0
	NOP	R0
	NOP	INCCOND
	INC	NOP
$\Updownarrow_{v=0}^{n-1} (\Updownarrow_{a=v+1}^{n-1} (w_{0_a} : r_{1_v}, w_{1_a} : r_{1_v}, n : r_{1_v}))$	NME	NOP
	NOP	LOAD
	NOP	ADD
	NOP	NME
	W0	R1
	W1	R1
	NOP	R1
	NOP	INC
	INC	COND
$\ldots$		
	END	END

Table 12.4. Meanings of the Output Synchronization signal

Received Primitive	**Output synchronization signal meaning**	**Rationale**
Write / Read	End of Instruction (EOIN)	Set to ‘1’when the instruction is finished and the input synchronization signal is equal to ‘1’. In this way the BIST Processor receives the logic-AND of the output signals generated by the memories under test and the input EOIN signal of the BIST Processor switches to ‘1’ only when all the EOIN signals of the memories under test have been set to ‘1’, i.e., all the memory Wrappers has completed the execution of the instruction
Inc/Dec CondInc/CondDec	End of Address (EOAD)	Set to ‘1’ when the whole addressing space has been visited by the AG
NextBP	End of Background Pattern (EOBP)	Set to ‘1’ when all the background patterns have been used
End	Results	Set to the logic AND among the result start bit and the synchronization signal of the preceding port wrapper
During Diagnosis	ScanResult	Set to the results status bits in order to form the Results_Scan_Chain
During scheduling configuration	ScanSched	Set to the mode status bits in order to form the NormTest_Scan_Chain

Table 12.5. Memory Wrapper overhead

RAM	**Wrapper Area**
2*[tpa21x8]	4,574
2*[spa*21x26]	2,165
dpa21x25	4,470
spa21x59	3,148
tpa336x8	5,169
dpa32x8	2,829
tpa42x8	4,509
2*[spa21x34]	2,870
2*[spa*21x19]	1,925
spa21x34	2,162
spa21x51	2,989
7*[qpa32x9]	8,543
7*[qpa32x9]	8,543
6*[qpa32x9]	8,084
qpa32x9	5,924

Table 12.6. Total area overhead

Glue Logic area	862,347
Total RAM area	380,503
Total Wrapper area	68,177
BIST processor area	5,431
µProgram memory area	4,459
Total	1,320,917
Total area overhead	**6,28%**

Chapter 13

Automatic Test Generation

As mentioned in the Introduction, automatic generation of memory test algorithms is a pressing issue in testing complex SoC. This chapter presents a methodology to automatically generate March Tests. A general representation is used to model known memory faults, and to possibly add new user-defined faults. With respect to previously proposed approaches, which exhaustively generate all the possible March Tests and then select the optimal one, the proposed approach allows generating the optimal March Tests in a very low computation time without exhaustive searches. In particular, the automatic March Test generation process is performed in the following steps: (i) the target memory fault list is modeled into a set of Finite State Machines (FSMs) representing the faulty memory behaviors; (ii) a weighted graph is generated, which represent all the possible test patterns able to cover each target fault model; (iii) an optimal test sequence is generated finding an optimal path connecting all the nodes of the graph; (iv) from the defined optimal sequence, a minimal March Test is derived applying a set of linear complexity transformations.

13.1 State of the Art

The problem of the automatic generation of March Tests has been already faced and several publications can be found in literature. [74, 72, 73] present an algorithm for March Test Generation exploiting a transition tree. The transition tree is generated in such a way that each path from the root node to a leaf represents a March Test. The March Test able to address the selected fault list is searched into the tree. The main problem of this approach is that the transition tree is unbounded. In order to limit the size of the tree, an upper bound on the number of nodes in a path is used. This can cause a high number of reiterations to find a solution making the algorithm inefficient and time consuming. Furthermore, when dealing with undetectable faults, the computation time becomes infinite. In addition, this method performs an exhaustive search to find the shortest path on the transition tree. As the size of the transition tree increases, the algorithm becomes more and more inefficient.

In [80] the authors present a branch and bound method that limits the search process to the parts of the tree where a solution exists and therefore a solution is found much faster and more efficiently.

An approach able to cover additional faults is presented in [60]. It mainly targets the diagnosis of memory faults and uses a fault description that allows modeling all possible single cell and two cells faults that occur in memory arrays. This approach still uses exhaustive search and is affected by the same problems of [74, 72, 73].

13.2 Memory Model

The problem of the automatic generation of March Tests requires, first of all, the definition of a formal model able to represent the behavior of both the good and the faulty memory. In [29] and [28] the problem has been solved proposing a memory behavioral model based on Finite State Machines (FSM). An n one-bit cells memory can be represented using a deterministic Mealy Automata:

$$M = (Q, X, Y, \delta, \lambda) \tag{13.1}$$

where:

- $Q = \{(0,1,-)^n\}$ is the set of the possible *memory states* where the symbol represents the value of a non initialized memory cell;
- $X = \{r^i, w_0^i, w_1^i | 0 \leq i \leq n-1\} \cup \{T\}$. This alphabet is composed of all the possible memory operations. In particular:
  - r^i corresponds to a read operation performed on the cell i;
  - w_d^i corresponds to a write operation of the value performed on the cell i;
  - T corresponds to a wait operation for a defined period of time. This additional element is needed to deal with Data Retention Faults [74].
- $Y = \{0,1,-\}$ is the output alphabet;
- $\delta = Q \times X \mapsto Q$ is the state transition function;
- $\lambda = Q \times X \mapsto Y$ is the output function.

Using the proposed model, a fault free two cells RAM is represented by the FSM shown in Figure 13.1, conventionally named M_0 in the reminder of this section. In M_0, the letters i and j are used to identify the first and the second cell, respectively.

The proposed model is not manageable when used to represent large memories; nevertheless, the model of a two-cell memory is general enough to model memory faults. Therefore, the behavior of a faulty memory is modeled using a deterministic Mealy Automata:

$$M_i = (Q_i, X, Y_i, \delta_i, \lambda_i) \tag{13.2}$$

where:

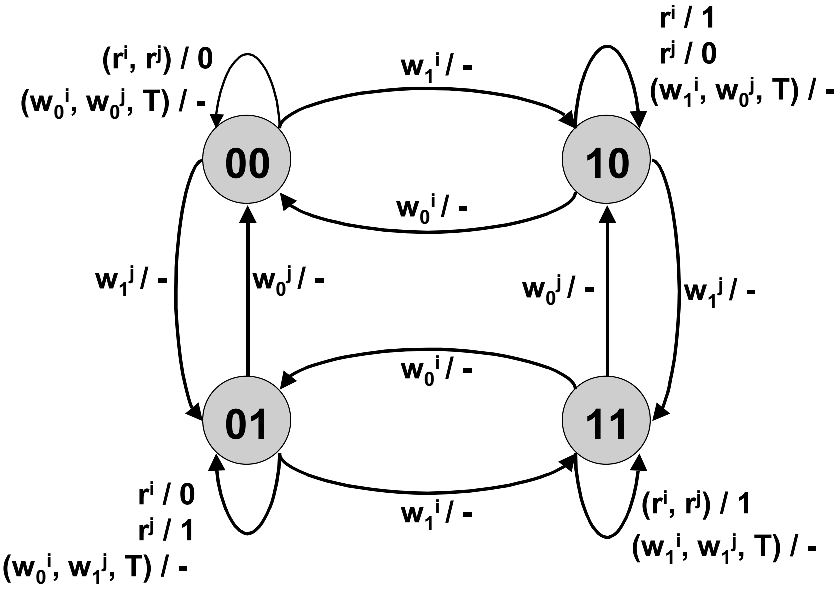


Figure 13.1. M_0 FSM representing a fault free RAM

- $Q_i \subseteq Q$ is the set of states;
- $Y_i \subseteq Y$ is the output alphabet;
- $\delta_i = Q_i \times X \mapsto Q_i$ is the state transition function
- $\lambda_i = Q_i \times X \mapsto Y_i$ is the output function.

The set of states used to represent a faulty memory is a subset of the whole set Q (See Equation 13.2) since only the cells involved in the fault should be represented. This consideration makes possible the use of the proposed model for very large memories as well. Moreover, the given representation for faulty memories is general enough to be used to model most of the known faults.

Considering as an example the Idempotent Coupling Fault $\langle \uparrow, 0 \rangle$ [70] (where the notation $\langle S, F \rangle$ denotes a fault involving two cells; S describes the condition of the first cell to sensitize the fault in the second cell denoted by F), we obtain the FSM shown in Figure 13.2. From now on, we will assume that the address of cell i is lower then the address of cell j.

As previously mentioned, since the fault involves two cells only, the cardinality of Q_i is four. The difference between the M_0 and M_1 machine is in the δ function, as pointed out by the two-bolded edges shown in Figure 13.2.

Looking at the M_1 machine it is possible to split each fault into a set of *Basic Fault Effect (BFE)* [60][80]. A BFE can be described by a M_i FSM with a δ_i function that differs

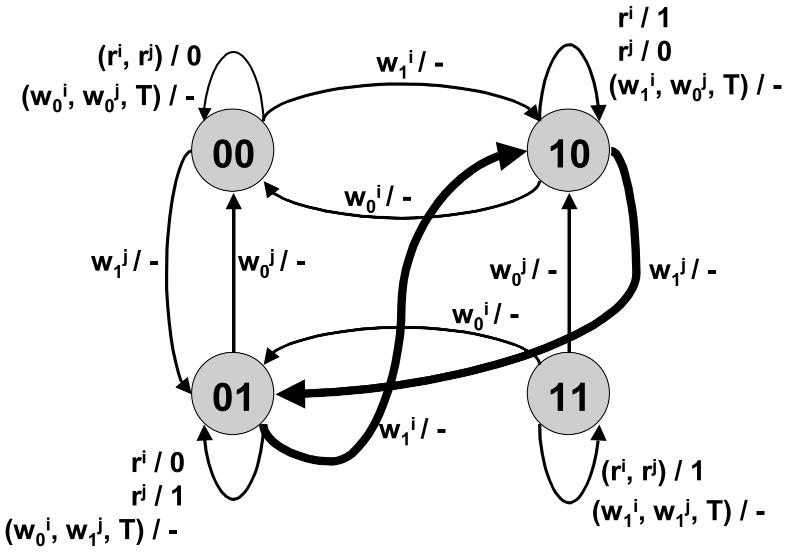

(r^i, r^j) / 0
(w_0^i, w_0^j, T) / -
w_1^i / -
r^i / 1
r^j / 0
(w_1^i, w_0^j, T) / -
00
10
w_0^i / -
w_1^j / -
w_0^j / -
w_0^j / -
w_1^j / -
w_0^i / -
01
11
w_1^i / -
r^i / 0
r^j / 1
(w_0^i, w_1^j, T) / -
(r^i, r^j) / 1
(w_1^i, w_1^j, T) / -

Figure 13.2. M$_1$: ⟨↑ ,0⟩ Idempotent Coupling Fault Representation

from δ_0 by one transition only, or with a λ_i function that differs from λ_0 by one output value only. Considering the example proposed in Figure 13.2, it is possible to identify two different BFEs modeled by the two FSM shown in Figure 13.3. For the sake of simplicity only the relevant edges are represented.

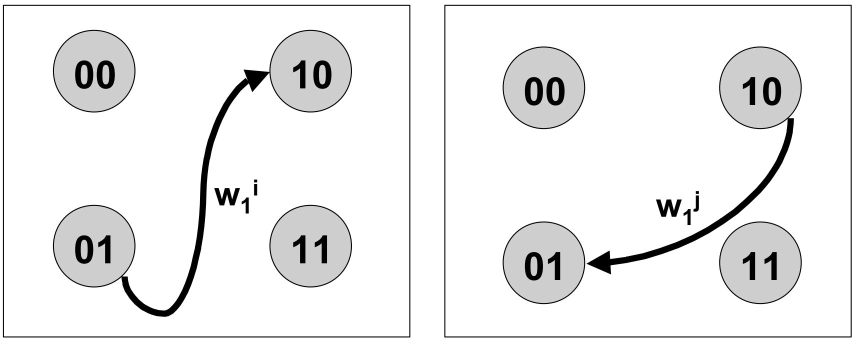

00
10
w_1^i
01
11
00
10
w_1^j
01
11

Figure 13.3. BFE model for ⟨↑ ,0⟩ Coupling Fault

Each BFE can be covered generating a Test Pattern (TP) defined as a triplet:

$$TP = (I, E, O) \tag{13.3}$$

where:

- $I = \{(0,1)^k \mid 0 \leq k \leq n-1\}$ is the initialization state;

- $E = \{e \mid e \in X\}$ is the operation needed to excite the BFE;
- $O = \{r_d^k \mid d \in (0{,}1), 0 \leq k \leq n-1\}$ is the operation needed to observe the fault effect. The notation r_d^i means “read the content of the cell i and verify that its value is equal to d”.

For the proposed example the two BFEs can be tested by the following two TPs:

- $TP_1 = (01, w_1^i, r_1^j))$
- $TP_2 = (10, w_1^j, r_1^i))$

13.3 March Test Generation Algorithm

In this section the generation algorithm will be presented. It exploits the possibility of automatically generating March Tests without exhaustive searches. The algorithm, starting from an unconstrained list of target BFEs, generates a non-redundant march test to cover all of them.

In a first phase, the algorithm analyzes the set of test patterns needed to cover each target BFE, and generates a weighted graph named *Test Pattern Graph (TPG)*. Each TPG node is associated to a TP. The graph is strongly connected, i.e., each node is connected to all the others.

The *weight* of each edge represents the number of memory operations needed to reach the initialization state of the target node (S_T), starting from the observation state of the source node (S_S). In a formal way it can be defined as [45]:

$$weigh = hamming - distance(S_S, S_T) \tag{13.4}$$

Lets consider as an example the $FaultList = \{\langle\uparrow,1\rangle\langle\uparrow,0\rangle\}$ [70]. The faults are modeled by four different BFEs, respectively tested by the following set of TPs:

- $TP_1 = (01, w_1^i, r_1^j))$
- $TP_2 = (10, w_1^j, r_1^i))$
- $TP_3 = (00, w_1^i, r_0^j))$
- $TP_4 = (00, w_1^j, r_0^i))$

The proposed set of test patterns generates the TPG shown in Figure 13.4.

Starting from the TPG the algorithm extracts a so-called *Global Test Sequence (GTS)*. A GTS is a set of memory operations able to detect the target BFEs. Different GTSs can be obtained by simply concatenating all the different TPs in multiple ways, i.e., to make different visits of the TPG. Since the TPG is strongly connected, the total number of possible GTS can be computed as follow:

$$\#GTS = V! \tag{13.5}$$

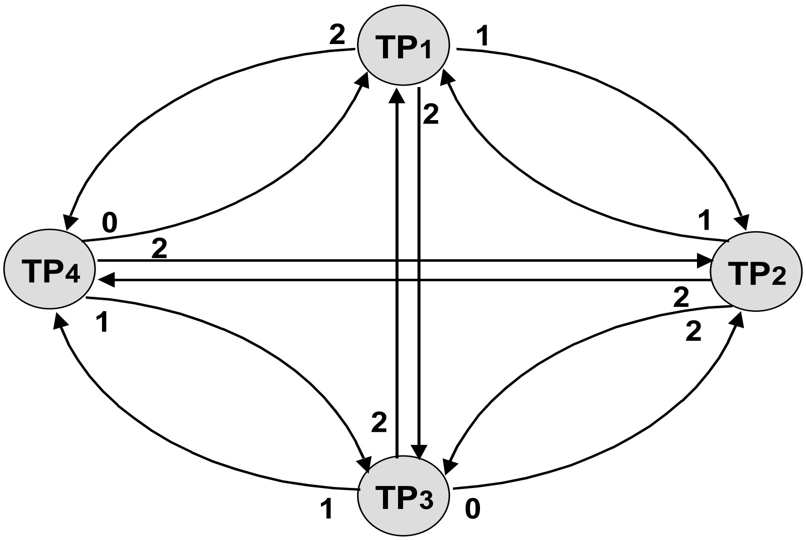


Figure 13.4. TPG for $\{\langle\uparrow,1\rangle\langle\uparrow,0\rangle\}$

where V is the number of nodes in the TPG.

In a fault-list containing a large amount of BFEs, the space of all the possible GTS becomes unmanageable. It is therefore necessary to identify a particular subset of GTSs able to generate non-redundant March Tests. A possible solution is to consider TPG visits with minimum weight only. Thanks to the function used to weight the TPG edges, these visits generate GTSs with minimum number of test operation (see Equation 13.4). Considering two nodes connected by a 0 weight edge, the test sequences obtained by their concatenation does not need the initialization part of the second TP.

The use of GTSs with minimum number of operations seems a good choice since there is a strict correlation between the GTS length and the March test complexity.

The generation of minimum length GTSs is a typical instance of the *Asymmetric Traveling Salesman Problem (ATSP)* [40]. The ATSP is probably the most well known member of the wider field of the *combinatorial optimization problems.* In a general instance of the ATSP, one is given V nodes and a matrix $d_{i,j}$ storing the distance or cost function to go from node i to node j . A “tour” consists of a list of V nodes, (*tour[i]*) where each node appears once and only once. The ATSP tries to find the tour with the minimum length, where the length is defined to be the sum of the lengths along each step of the tour:

$$length = \sum_{k=0}^{V-1} d_{tour[k],tour[k+1]} \tag{13.6}$$

and *tour[V]* is identified with *tour[0]* to make it periodic.

The main difference with respect to our problem is that the solution of the ATSP is a cycle whereas a GTS is identified by a non-cyclic path (i.e., the first and last node do not need to be the same). To solve the problem two dummy nodes used to close the cycle are introduced. Despite the ATSP is a NP-hard problem, several algorithms able to give an exact solution with very low computation time in problems with low number of nodes (50 nodes), can be found in literature [30].

The GTSs obtained by solving the ATSP problem are able to test all the addressed BFE but are not yet March Tests. A March Test is a particular Test Sequence respecting a set of conditions [69]. It is therefore necessary to apply a set of modifications to transform a GTS into an equivalent March Test.

Before applying the modifications, it is possible to perform a further optimization. The use of GTSs starting with a "00" or "11" initialization state allows obtaining March Tests of the lowest possible complexity. This optimization, which allows reaching a minimal solution considering all the minimum length GTSs, can be expressed as an additional constraint in the ATSP:

$$length = \sum_{k=0}^{V-1} d_{tour[k],tour[k+1]} \tag{13.7}$$

s.t.

$$TP_{tour[0]} = (00,11,\ldots,\ldots) \tag{13.8}$$

Looking at the example of Figure 13.4, a possible ATSP solution is the following GTS:

$$GTS = w_0^i, w_0^j, w_1^i, r_0^j, w_1^j, r_1^i, w_0^i, w_0^j, w_1^j, r_0^i, w_1^i, r_1^j \tag{13.9}$$

The process of March Test generation from a GTS passes through three different steps:

- GTS reordering
- GTS minimization
- March Test Generation

Each step corresponds to a different set of *Rewrite Rules* [36]. Since a GTS can be considered as a string where each symbol is a memory operation, the rewrite rules can be effectively represented resorting to the *Regular Expression* formalism [3]. All the possible memory operations are defined by the X alphabet defined in Equation 13.2.

For the sake of simplicity we define two subsets of instructions:

- $w = \{w_d^i, w_d^j\}$ is the set of possible memory write operations;
- $r = \{r_d^i, r_d^j\}$ is the set of possible memory read operations.

The regular expression formalism is extended introducing three new operators:

- End Symbol Operator: $\hat{s}$ marks the symbols as not further modifiable (terminal symbol);
- Red Operator: $[S]_R$ marks the symbols with the red color;

- Blue Operator: $[S]_B$ marks the symbols with the blue color.

The use of colored symbols is useful during the March Test generation phase to identify the boundaries of the different March Elements. The next subsections summarize the rewrite rules used during the three different phases.

13.3.1 GTS Reordering

The reordering phase reorders the GTS memory instructions taking into account the constraints needed to obtain a March Test [69]. In this phase each modification is defined by a *Pattern* and by a *Rewrite Rule* (see Table 13.1). The pattern is a regular expression that identifies all the strings on which the rewrite rule must be applied. The reordering process stops when all the GTS symbols are modified into terminal ones.

Table 13.1. Reordering Rewrite Rules

Pattern	**Rewrite Rule**
$(\hat{w}\vert\hat{r})^* w^i_d w^j_d (w\vert r)^*$	$w^i_d w^j_d \xrightarrow{M1} \hat{w}^i_d \hat{w}^j_d$
$(\hat{w}\vert\hat{r})^* w^i_d w^i_d (w\vert r)^*$	$w^i_d w^i_d \xrightarrow{M2} \hat{w}^i_d w^i_d$
$(\hat{w}\vert\hat{r})^* w^i_d w^j_{\overline{d}} (w\vert r)^*$	$w^i_d w^j_{\overline{d}} \xrightarrow{M3} \hat{w}^i_d w^j_{\overline{d}}$
$(\hat{w}\vert\hat{r})^* \hat{r}^i_d \underbrace{(\hat{w}^i_d\vert\hat{w}^j_d\vert\hat{w}^j_{\overline{d}})^*}_{s1} \underbrace{(\hat{w}^j_d\vert\hat{w}^j_{\overline{d}})^*}_{s2} r^i_d (w\vert r)^*$	$\hat{r}^i_d S_1 S_2 r^i_d \xrightarrow{M4} \hat{r}^i_d [\hat{r}^i_d]_R [S_1 S_2]_B$

Applying the reordering rules on the GTS in Equation 13.9 it is possible to obtain the following reordered sequence:

$$GTS_R = \hat{w}^i_0, \hat{w}^j_0, [\hat{r}^i_0]_R, [\hat{w}^i_1]_B, \hat{w}^j_1, \hat{r}^i_1, \hat{w}^i_0, \hat{w}^j_0, [\hat{r}^j_0]_R, [\hat{w}^j_1]_B, \hat{w}^i_1, \hat{r}^j_1 \qquad (13.10)$$

13.3.2 GTS minimization

The minimization phase deletes redundant subsequences in order to reduce the sequence to the minimum set of absolutely necessary operations only. The rewrite rules applied in this phase consider the GTS starting from left to right (see Table 13.2). This phase is repeated until no further minimization can be applied. In this context the \$ symbol is used to denote the end of the GTS and the color of the symbols does not affect the application of the rules.

Applying the minimization rewrite rules on the reordered GTS_R of Equation 13.10 it is possible to obtain the following minimal sequence:

$$GTS_M = \hat{w}^i_0, [\hat{r}^i_0]_R, [\hat{w}^i_1]_B, \hat{r}^i_1, \hat{w}^i_0, [\hat{r}^j_0]_R, [\hat{w}^j_1]_B, \hat{r}^j_1 \qquad (13.11)$$

Table 13.2. Reordering Rewrite Rules

Rewrite Rules
$\hat{w}^i_d\hat{w}^j_d \xrightarrow{R1} \hat{w}^i_d \quad \hat{r}^i_d\hat{r}^j_d \xrightarrow{R1} \hat{r}^i_d$
$\hat{w}^i_d\hat{w}^i_d \xrightarrow{R1} \hat{w}^i_d \quad \hat{r}^i_d\hat{r}^i_d \xrightarrow{R1} \hat{r}^i_d$
$\hat{r}^i_d\hat{w}^i_{\overline{d}}\hat{w}^i_d\hat{r}^j_d\hat{w}^j_{\overline{d}}\hat{w}^j_d \xrightarrow{R3} \hat{r}^i_d\hat{w}^i_{\overline{d}}\hat{w}^i_d\hat{r}^j_d$
$\hat{r}^i_d\hat{w}^i_{\overline{d}}\hat{w}^i_d\hat{r}^j_d\hat{w}^j_{\overline{d}}\$ \xrightarrow{R3bis} \hat{r}^i_d\hat{w}^i_{\overline{d}}\hat{w}^i_d\hat{r}^j_d\$$

13.3.3 March Test Generation

This last phase uses the minimized GTS to generate a March Test. The input sequences are analyzed from left to right and the March Elements are generated according to the following rules:

- Rule 1: subsequences identified by $(\hat{w}^i_d|\hat{r}^j_d)(\hat{w}^j_d|\hat{r}^j_d)$ regular expression close a March Element and open a new one;
- Rule 2: subsequences identified by $[\hat{r}]_R([\hat{w}]_B*)$ regular expression are joined in a single March Element despite they are executed on i or on j. The last blue marked operation closes the March Element.

The addressing order is generated using the following rules:

- Rule 3: March Elements starting with colored operation performed on i cells have addressing order $\Uparrow$;
- Rule 4: March Elements starting with colored operation performed on j cells have addressing order $\Downarrow$;
- Rule 5: March Elements starting with non-colored operations have addressing order $\Updownarrow$.

Applying the generation rules on the GTS_M of Equation 13.11 it is possible to obtain the following $8n$ non-redundant March Test:

$$M = \Uparrow w_0 \Uparrow r_0w_1 \Uparrow r_1w_0 \Downarrow r_0w_1 \Downarrow r_1 \tag{13.12}$$

13.4 BFEs equivalence

In some cases it is possible to obtain a BFE modeling a fault already covered by another BFE. A typical case is the Inversion Coupling Fault $\langle\uparrow,\updownarrow\rangle$ [69]. It can be split into two BFEs tested by the following TPs:

- $TP_1 = (00, w^i_1, r^j_0)$

- $TP_2 = (01, w_1^i, r_1^j)$

Although two TPs are generated, only one of them is necessary to cover the fault. Therefore, the ATSP problem must be modified to take into account only the necessary test patterns. This goal can be achieved grouping the TPG nodes into equivalence classes (C_i).

In case of a TPG with k equivalence classes, using the $|C_i|$ notation to indicate the cardinality of the C_i class, it is possible to generate $E = \prod_{i=0}^{k-1} |C_i|$ different TPG. On each one of the obtained graphs the ATSP problem must be solved identifying E possible GTS. The minimum length GTS is considered as the best one.

13.4.1 Experimental Results

This section reports some experimental results obtained applying the proposed algorithm to automatically generate March Tests to cover different sets of faults.

The algorithm has been implemented in about 5000 lines of C code. The ATSP has been solved using a Fortran code able to give exact solutions to the problem [30]. For each March Test, it is reported the computation time needed for the generation process, the complexity, and the complexity of an equivalent March Test found in literature. All the experiments have been performed on a *Compaq Presario 17XL370, PIII 650Mhz* based Laptop with 128 MB of RAM. The source code has been compiled with the gcc C compiler and the g77 Fortran compiler (`http://www.gnu.org`).

Table 13.3 shows the March tests obtained to cover some combinations of Stuck-At Faults (SAF), Transition Faults (TF), Address Decoder Faults (ADF), and Inversion and Idempotent Coupling Faults (CFin and CFid).

Table 13.3. Experimental Results

Fault List	**Generated March Tests and their complexity**	**CPU** Time	**Equivalent Known March Test**
SAF	$\{\Uparrow w_1 \Downarrow r_1 w_0 \Downarrow r_0\}$ (4n)	0.49	MATS (4n)
SAF,ADF	$\{\Uparrow w_1 \Uparrow r_1 w_0 \Downarrow r_0 w_1\}$ (5n)	0.53	MATS+ (5n)
SAF,TF,ADF	$\{\Uparrow w_0 \Uparrow r_0 w_1 \Downarrow r_1 w_1 \Uparrow r_0\}$ (6n)	0.61	MATS++ (6n)
SAF,TF,ADF, CFin	$\{\Uparrow w_0 \Downarrow w_1 \Downarrow r_1 w_0 \Uparrow r_0 w_1\}$ (6n)	0.69	MarchX (6n)
SAF,TF,ADF, CFin, CFid	$\{\Uparrow w_1 \Uparrow r_1 w_0 \Uparrow r_0 w_1 \Downarrow r_1 w_0\}$ $\Downarrow r_0 w_1 \Downarrow r_1\}$ (10n)	0.85	MarchC- (10n)
CFin	$\{\Uparrow w_0 \Uparrow R_0 w_1 w_0 \Downarrow r_0\}$ (5n)	0.57	Not found

Chapter 14

Memory Test Algorithms Verification

Due to the complexity of both fault models and memory architectures, manual analysis [69] of the memory fault coverage is not anymore possible. In [35], a memory simulator (Memory Animation Package Plus, MAP+) has been proposed. This tool, developed at the Delft University of Technology, has been employed as a simulation tool for the evaluation of new and known test algorithms in presence of different faults. Although very interesting especially from an academic point of view, this tool does not allow a very detailed fault simulation.

In this section the architecture of a new flexible memory fault simulator, designed to address all the most critical issues in today's memories test generation and validation will be presented. Besides the fault coverage computation, already addressed by other similar tools, [79] the proposed simulator supports the test engineer in optimizing the test algorithm and in addressing power consumption constraints. The tool is in fact able to compute the power consumption generated by the test input sequence, and to suggest a modification of the test algorithm in case its application does not fulfill a user-defined power consumption constraint.

14.1 The Fault Simulator Architecture

Figure 14.1 presents the simulator overall architecture. The *Object Oriented Memory Simulator* reads two main input files containing the memory functional and electrical models and the input test sequences, and simulates the execution of the input test sequence storing, for each memory cell, the logical and electrical temporal evolution. After the simulation, a Test Analysis module reads the target Fault Model files and computes their coverage w.r.t. the input test sequence. It generates two output files storing a detailed test report, and, whenever possible, an optimized input test sequence able to provide the same results of the original one.

The memory model is split in two parts:

- a functional model, represented as a Finite State Machine (see section 13.2);

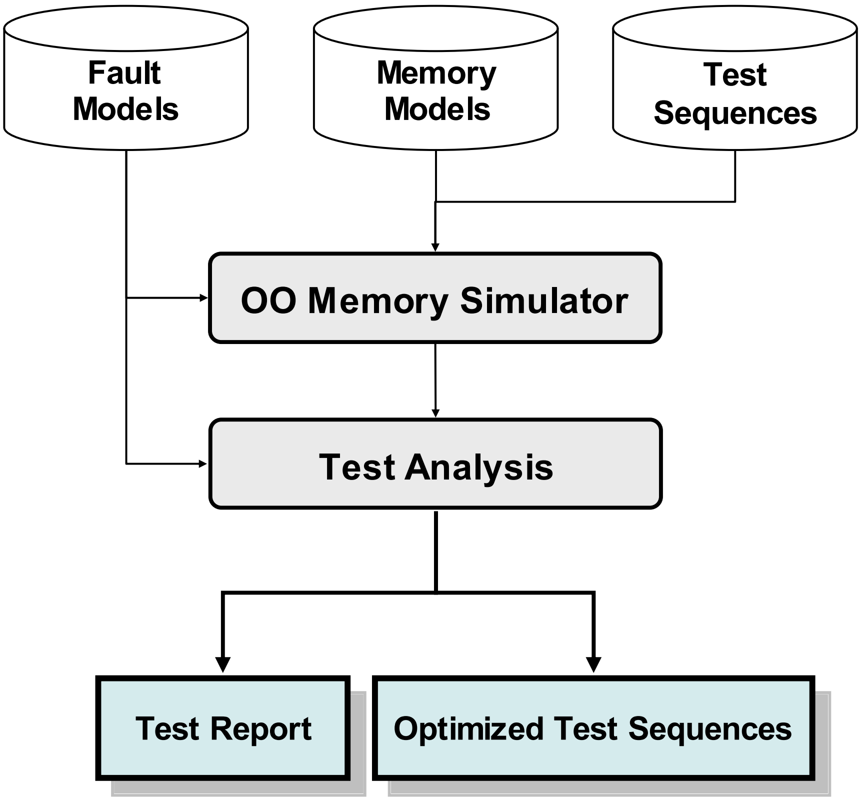


Figure 14.1. Simulator architecture

- an electrical and physical model, storing all the operating, technological, and topological characteristics of the memory (see section 14.2);

The Fault Model files, formalized as collections of Basic Fault Effects (see section 13.2), describe the faulty behavior that the input test sequence is designed to detect.

The Test Sequence files describe the sequence of operations applied to test the memory array. Using a proprietary language, it is possible to describe complex test algorithms as well as simple sequences of input patterns.

The Test Report file contains detailed information about:

- the Fault Coverage for each fault class and, when necessary, diagnostic information about the cells where the fault is not covered;
- the total power consumption caused by the application of the test sequences;
- if the computed power consumption is higher than the allowed limit, if possible, the simulator provides the suggested maximum clock frequency that allows to meet the power requirements;

Finally, the Test Analysis module outputs an optimized test sequence, where redundant elementary operations not affecting the final fault coverage are removed.

14.2 The Electrical and Physical Model

Besides the memory behavior, the user can specify a set of electrical parameters that constitute the memory electrical and physical model. In particular, it is possible to specify:

1. the typical *operating conditions* (e.g., supply voltage, operating temperature, etc.): this part is included in the simulator for future developments, to take into account not only the functional behavior of the memory but also its operating conditions;
2. the actual row/column topological organization of the memory array: this information is necessary to compute the coverage of faults involving adjacent cells;
3. the timing and electrical characteristics (e.g., access time, operating and stand-by current, etc.): these characteristics allow the simulator to compute different parameters like the average power dissipation caused by the application of a given test algorithm.

14.3 Input Test Sequences

The memory input test sequences are defined using a language able to describe complex test algorithms as well as simple sequences of input patterns.

In particular, the language constructs have been defined to help the description of:

- Simple *Read* or *Write* operations scheduled at a given time;
- *March elements*: set of instructions repeated on all the cells of the memory array [69];
- *Burst cycles*: a single instruction to be executed on consecutive memory cells;
- *Neighborhood cells*: a set of neighborhood cells on which executing a given set of operations;
- *Transparent Tests*: write operations where the written value is a function of the memory cell content;
- *Background patterns*: the set of patterns to be used to test word-oriented memories;
- *C-like* statements: *for* loops, *if-then-else* constructs, and evaluation of simple Boolean expressions that allow the definition of complex test algorithms;
- *Changes in the operating conditions*: operations that, for example, simulate a change of the operating temperature of the memory. These operations have been introduced to allow, in the future, the modeling of fault depending on the memory operating conditions..

Figure 14.2 presents an example of part of the Walking 1/0 algorithm described using the proposed language.

```
// Walking 1/0 (first part)
walking10::     any ( w 0 );
for ( i = 0; i < words_in_array; i = i + 1 )
{
               w [i] 1;
      neigh array any [i] ( r 0 );
```

Figure 14.2. Example of test algorithm including Neighborhood cells and a cycle statement

14.4 Simulator Engine

The proposed memory fault simulator has been designed using a layered object oriented approach. The models describing the memory, the faults, and the test input sequences are classes with a predefined set of methods and properties.

The simulator engine is designed using an onion skin-like approach (Figure 14.3), where each layer targets a different functionality of the simulation: access to the memory array, the electrical behavior, the temporal behavior, and the I/O behavior. This approach allows designing an efficient, modular, and very easily upgradeable tool. The only constraint of each layer is its interface; any layer internal behavior or structure can be redesigned, modified, or upgraded without redesigning the whole simulator.

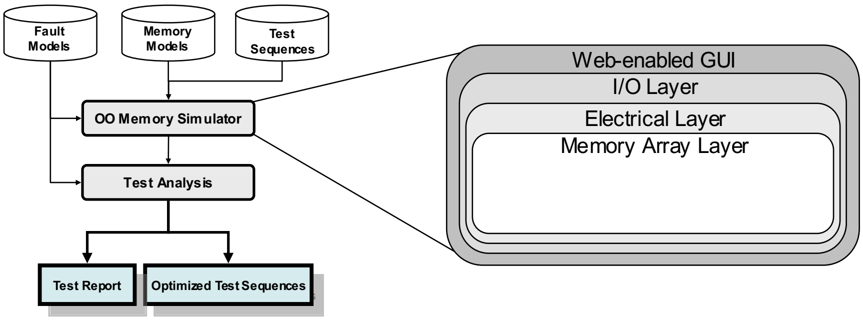

Fault Models
Memory Models
Test Sequences
OO Memory Simulator
Test Analysis
Test Report
Optimized Test Sequences
Web-enabled GUI
I/O Layer
Electrical Layer
Memory Array Layer

Figure 14.3. The simulator engine layered structure

The *Graphical User Interface* (GUI) of the simulator is completely web-enabled. This feature allows to have a user-platform-independent tool, which is always up-to-date with the most recent version, and does not require any installation process. The user does not require a high computational power on its machine, and can easily access the tool from any internet connection point. This characteristic is particularly useful in an academic environment, where students always outnumber the available workstations.

The *I/O Layer* implements the possible I/O operations on the memory. The allowed operations, with their temporal constraints, are read from the memory model. In this way, the simulator is also able to check if all the operations executed by the test algorithm are legal for the target memory chip. The I/O Layer is also in charge of reading and applying the input test sequence defined in one of the input files. In order to optimize the simulation time, the simulator computes from the Fault Model files and the input test files, the smallest number of memory cells that guarantee the ability of computing the final result for the real memory size. The *Electrical Layer* computes and logs the temporal evolution of the electrical and physical characteristics of each cell during the application of the input sequence. At each instant, the state of a cell is defined as the voltage level of the cell plus a *timeout*, after which the voltage of the cell has to be re-evaluated even if the value of the cell is not changed by an external operation. A transition to another state is triggered by an event. Possible events are: a read or write operation on a cell, the expiration of the timeout of the cell, or the variation of the memory operating conditions. The user can define the behavior of the memory cells when a transition is triggered. For example, it is possible to define the function that computes the voltage level of a cell upon the expiration of a timeout. This layer can be made transparent (and therefore disabled) if the electrical evolution of the memory is not required by the user.

Finally, the *Memory Array Layer* is used to log the logical evolution of the memory content only. This layer considers the memory as a matrix of words as defined by the memory model.

14.4.1 Test Analysis Module

For each Fault Model, the *Test Analysis Module* considers each set of Test Patterns defining a BFE, and verifies, using a pattern matching algorithm, if it has been executed during the simulation of the input test sequence. If all the Test Patterns belonging to a Fault Model have been executed on a cell, then, for that cell, the fault is covered. Besides fault coverage, the Test Analysis module also computes the total power consumption caused by the application of the input test sequences.

An interesting feature of the Test Analysis Module is its ability to suggest optimizations to the input test algorithm. In particular, it is able to:

- check the non-redundancy of each elementary operation in the input test sequence, and suggest a possible optimization;
- suggest a possible modification of the test sequence in order to fulfill the power consumption constraints.

To check the non-redundancy of each *elementary operation*, the Test Analysis Module builds a *Coverage Matrix* (CF) where each row represents the elementary operations whereas the columns the target fault models. A matrix cell is set to the value one if the corresponding elementary operation contributes to the coverage of the fault represented by the column. An input sequence is able to detect all the target faults if for each CF column exist at least one row containing a cell set to one. The test sequence is non-redundant if all the matrix rows are needed to cover the target faults. If this is not the case, the module outputs a new test sequence trimmed of all the redundant elementary operations. This is a typical instance of the *Set Covering* problem applied on the Coverage Matrix. The Set Covering finds the minimum number of rows needed to cover all the columns. This approach has been successfully applied on many known March Tests, where redundant blocks have never been found.

Finally, to address power consumption, the Test Analysis Module is able to compute either the maximum clock frequency that allows to meet the power constraints, or, if the clock is not modifiable, it inserts *delay* instructions in the input test sequence.

Chapter 15

Self Repair

The last point to address is the possibility of having not only testable RAM but also to have repair capability. The emerging field of Self-Repair Computing is expected to have a major impact on deployable systems for space missions and defense applications that need to survive and perform at optimal functionality during long duration in unknown, harsh and/or changing environments. Examples of such applications include outer solar system exploration, missions to comets and planets with severe environmental conditions, long lasting space-borne surveillance platforms, defensive counter-measures, long-term nuclear waste and other hazardous environment monitoring and control. Self-Repair Computing is also expected to greatly enrich the area of commercial applications in which high availability and serviceability is needed; such applications range from biomedical devices to automotive applications.

The process of repairing a RAM is divided into several steps. In a first phase, a test algorithm is executed on the memory array. If a fault is detected, it is necessary to locate it (diagnosis) and to allocate redundant memory space to replace the faulty cell. When these operations are built-in into the RAM architecture, the steps are named respectively BIST (Built-In Self-Test), BISD (Built-In Self-Diagnosis), BIRA (Built-In Redundancy-Allocation) and BISR (Built-In Self-Repair).

There are different possible solutions to insert redundant space into a memory array:

- *Row/column only*: The memory contains spare rows or columns. When a fault has to be repaired, the row/column containing the fault is replaced with one of the spare ones. This solution allows the manufacturer to easily repair faulty cells, but it does not allow optimal use of redundant space since repairing a single fault requires allocating a whole spare row or column.
- *Row-column*: The memory contains both spare rows and columns. Each fault can be repaired either by using a spare row or a spare column. When multiple faults are detected, this technique allows a more efficiently repair by selecting the best combination of spare rows/columns.
- *Cell-only*: Instead of repairing an entire row or column, when a fault is detected, only the address of the faulty cell is re-mapped on to a new cell, thus allowing an

optimal allocation of the redundant space.

This paragraph proposes an innovative architecture for SRAMs, characterized by BISR capabilities based on Cell-only redundant space allocation at the user level. The memory is not electrically repaired, but spare cells replace faulty ones using an on-line address re-mapping scheme. The repair process is transparent to the user, and is independent from the memory physical implementation.

In the proposed approach, the self-repair architecture is coupled with an ad-hoc defined on-line transparent BIST algorithm. The on-line BIST is therefore executed concurrently with the memory normal behavior, and is able to detect the appearance of a wide range of faults, including coupling faults usually not detectable during end-of-production (EOP) or power-up tests. These faults have in fact a higher probability to appear in very high-density RAMs only when the circuit reaches high temperatures. Since this condition can not be guaranteed except after a long period of usage, EOP or power-up tests usually do not provide a good coverage.

The mentioned on-line BIST algorithm allows also implementing a very efficient Built-In Self-Diagnosis and Redundancy-Allocation strategy.

To assess the quality of the proposed architecture, a simulation-based fault injection environment has been set up to emulate the appearance of different faults in the memory module. Experiments were performed to validate the detection capability of the BIST circuitry first, and then the functionality of the BISR logic.

15.1 State of the art

So far, most of the research activities on self repair techniques focused on FPGAs [54, 51, 78, 64], and [26]. Built-in Self-Test and Built-in Self-Repair schemes have been proposed as potential solutions to the problem of repairing memories mainly at the manufacturer level (i.e., at the end-of-production) [63, 66, 31, 34, 43, 46, 25]. The scheme proposed in [63] actually limits self-repair only to field failures. To remove manufacturing defects, it employs the traditional row-column repair approach, based on an on-chip micro-programmed BIST scheme and self-repair logic block, with a spare memory block. In [66] the authors employ an elaborate on-chip RISC processor to collect and analyze full failure bitmaps to figure out a repair solution. Besides the complexity of the RISC processor, the method also requires that a large enough fault-free block of the RAM under test must be available to store the failure bitmap. In [31], ultra-large capacity single-chip memories are considered. The proposed architecture uses a hierarchical organization to achieve optimal conditions for memory access time. [34, 43, 46, 25] analyze algorithms that optimize the repair solution for a given bit failure pattern in a redundant RAM.

All these solutions are typically adopted for stand-alone memories, where it is possible (at the end of production) to carry out a hardware repairing through anti-fuse/laser techniques. Nevertheless, in today's high integrated circuits that embed large memories, the very low if not null physical accessibility to the cores makes hardware repairing not anymore feasible or cost effective.

Recently, new techniques have been introduced to perform a soft memory repair of memories through self-reconfiguration of the addressing space [4, 48, 50]. In the first two approaches, a Built-In Self-Test and Repair architecture is inserted into the RAM. The self-test is started at the system power up and the information on the faulty cells is stored and used thereinafter to reconfigure the memory. In the self-repair circuit, two registers are inserted for each redundant row or column. The first register stores the address of a faulty row (or column) whereas the second register stores the addresses of the row (or column) that replace the faulty one. When a faulty cell is addressed from the external, the circuit reacts changing the address value with the correct one. This solution is very expensive in terms of routing overhead. In fact, for each redundant row or column, a set of signals (for addressing the memory) is routed from the BISR circuits to the memory. In [48], to simplify the spare allocation procedure a column-only repair strategy is considered.

15.2 The conceptual architecture

The conceptual idea underlying the proposed approach is to couple an on-line transparent BIST algorithm with a *functional self-repair* architecture in the same *Built-In-Self-Test-and-Repair* (BISTAR) logic.

Functional self-repair means that a faulty cell must be replaced by a spare one using an address re-mapping scheme. The BIST part of the logic executes an on-line test, based on a linear algorithm, to detect single stuck-at, transition, coupling, and address faults.

A conceptual view of the BISTAR architecture is presented in Figure 15.1. The actual implementation of the BISTAR logic aiming at minimizing critical paths is presented in section 15.3. The self-repair logic is based on a Content Addressable Memory (CAM) used to re-map the address of the faulty cells. The BISTAR controller is in charge of executing the test algorithm and controlling the repair procedures.

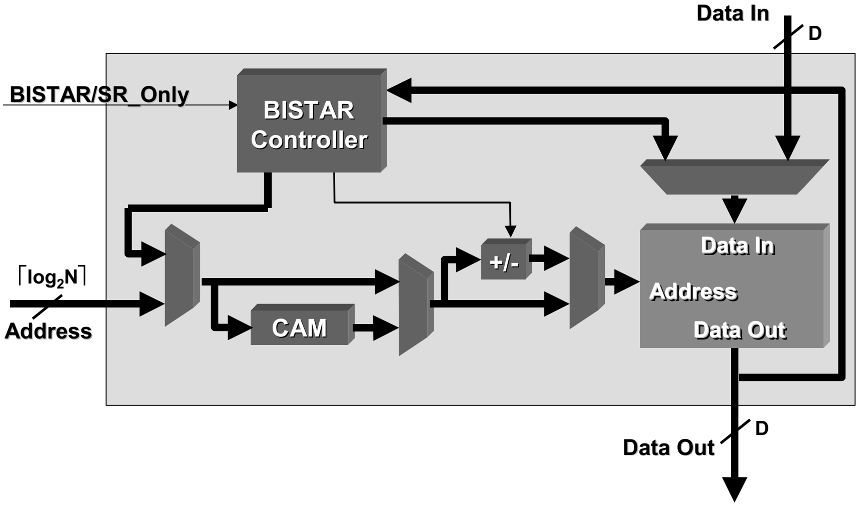


Figure 15.1. BISTAR conceptual architecture

In order to make the approach more general, and to allow the user to fulfill power budget constraints and to perform diagnosis from the outside, the core has two possible functional modes, selected via the input signal `BISTAR/SR_only`:

- `SR_only`: the BIST algorithm is disabled, but the self-repair capability is active. Despite new faults are not detected, the re-mapping addressing mechanism is still active for previously detected faulty cells.
- `BISTAR`: both self-test and self-repair capabilities are enabled. The BISTAR controller continuously executes the test algorithm and possibly repairs faulty cells.

The self-repair capability is exploited during the self-test of the memory array in order to guarantee the transparency from the user point of view. The proposed methodology mainly includes three phases executed sequentially: *isolation*, *test execution*, and *repairing* or *restoring*. Each cell $\mathrm{c_x}$ under test is *isolated* by functionally replacing it with one of the available spare cells $\mathrm{s_i}$: the content of $\mathrm{c_x}$ is copied into $\mathrm{s_i}$ and its address is stored into the CAM. During the test, any external operation on $\mathrm{c_x}$ is actually performed on $\mathrm{s_{i.}}$ The cell $\mathrm{c_x}$ is then *tested* on-line executing the algorithm described in subsection 15.2.2. To prevent the degradation of the memory performance, the test execution is temporary suspended to serve any external memory access. At the test completion, if no fault has been detected, the content of $\mathrm{c_x}$ is *restored* and its address erased from the CAM; otherwise, $\mathrm{s_i}$ is thereinafter used as a *repair* cell for $\mathrm{c_x}$.

15.2.1 Memory Built In Self Repairing

The proposed BISR strategy aims at keeping constant the memory storing capability seen by the user. Faulty cells are functionally replaced by spare ones via a dynamic on-the-fly reconfiguration of the memory addressing space.

From an external user point of view, the memory has a nominal addressing space of N cells, of m-bits each. The actual memory module, instead, has an effective storage capacitance of $N+K$ cells, K being the number of spare cells added for self-repairing.

To optimize the allocation of the redundant memory space, the approach is based on a cell-only repair strategy. Therefore, when a fault is detected in a cell, instead of repairing an entire row or column, only the faulty cell is re-mapped on a spare one.

Address re-mapping is achieved by a K-lines Content Addressable Memory (CAM), each line $\mathrm{l_i}$ corresponding to a spare cell $\mathrm{s_i}$. In particular, the line $\mathrm{l_i}$, $0 \leq i \leq K$, of the CAM stores the address of a faulty cell $\mathrm{c_j}$, $0 \leq j \leq K+N$. In this way cell $\mathrm{c_j}$ is functionally replaced by the spare cell $\mathrm{s_i}$. This solution allows reducing the area and the routing overhead. Instead of using an additional register into the CAM in which to store the address of the redundant cell (as proposed in [4] and [48]) the association between the line position and the redundant cell is hardwired. Note that the repairable memory array space includes all the $N+K$ cells of the memory, thus allowing us to repair spare cells as well.

Whenever a cell $\mathrm{c_x}$ of the memory is accessed, its address is first looked up in the CAM. Two cases can occur. On one hand, if $\mathrm{c_x}$ has been previously detected faulty (or is currently a cell under test), its address has been stored into the CAM. In such a case, when accessed, the CAM reacts with a hit and outputs the address of the replace cell s_i, and a proper mux routes it to the memory array. Any operation on the faulty cell c_x is thus performed on its replace cell s_i. On the other hand, if a spare cell does not currently replace the target cell $_x$, its address, not being stored in the CAM, is directly transferred to the memory array.

15.2.2 Memory Built In Self Testing

The proposed On-line self-test logic implements a custom *transparent* memory test algorithm, which does not therefore affect the memory content.

To achieve high dependability, the memory has to be repaired guaranteeing very low fault latency. Moreover, a significant variety of faults need to be targeted. A custom test algorithm has thus been adopted, optimized to best exploit the knowledge of the memory layout. The algorithm has linear complexity and addresses faults both occurring inside a memory cell and involving pairs of physically adjacent cells. The assumption of knowing the memory layout does not limit the applicability of the method, since foundries usually provide details about the internal memory structure, and tools are available to extract this information (e.g., FlexStream by LSI Logic [53]). Anyhow, if this is not feasible, it is always possible to implement a quadratic algorithm targeting the same faults without requiring such an internal knowledge.

As shown in Figure 15.2, the memory is tiled by a group of non-overlapping neighborhoods. As far as the cardinality of the neighborhood set is concerned, the Type-2 neighborhood choice of [69] has been done. In the sequel, c_x and n_j ($0 \leq j \leq 7$) will denote the *base cell* and the 8 *neighborhood cells*, respectively.

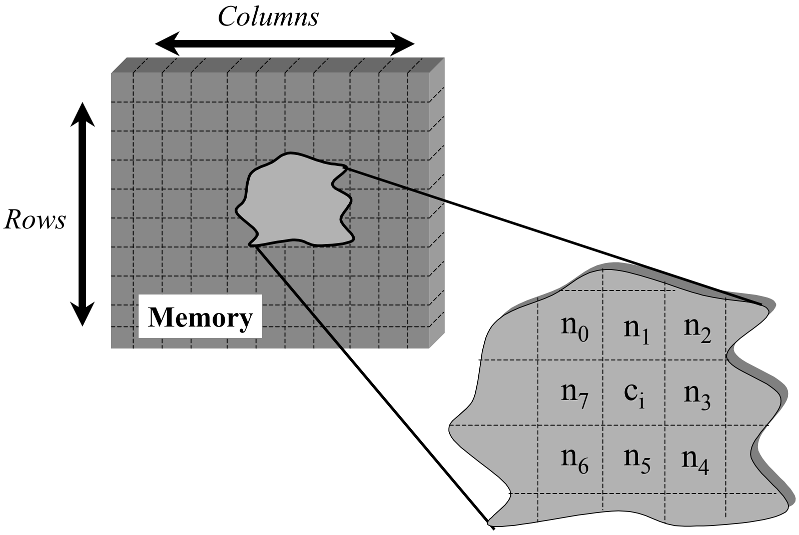


Figure 15.2. Memory layout

The implemented test algorithm targets the following faults:

- *Stuck-at* faults on the base cell;
- *Transition* faults on the base cell;
- *Intra-word coupling* faults (CFs) on the base cell for word oriented RAMs;
- *Inter-word coupling faults* between the base cell and its 8 neighborhoods.
- In particular, for both Intra- and Inter-word coupling faults, *idempotent* CFs ($\mathrm{CF_{id}}$) and *inversion* CFs ($\mathrm{CF_{in}}$) are covered [69].

The detection of *Intra-word* CFs is accomplished resorting to the Walking 1/0 intermixed complement Data Background Sequence proposed in [71].

To detect *Inter-word coupling faults*, the following test is executed on each pair of adjacent cells $\mathrm{c_x}$, $\mathrm{n_j}$:

$$\begin{aligned} &\{ \\ w_D(c_x), w_D(n_j), r_D(c_x), w_{notD}(n_j), r_d(c_x).w_d(n_j), r_D(c_x), w_{notD}, \\ (c_x), w_D n_j, r_{notD}(c_x), w_{notD}(n_j), r_notD(c_x), w_D(n_j), r_notD(c_x) \\ &\} \end{aligned} \tag{15.1}$$

where:

- r and w represent read and write operations, respectively;
- D and notD represent any background pattern and its complement, respectively.

The read/write operation on the base cell of both a value D and its complement notD covers stuck-at faults, whereas the transition fault on the base cell $\mathrm{c_x}$ are covered by the sequence:

$$\begin{aligned} \{w_D(c_x), r/w(n_1), r_D(c_x), r/w(n_1), w_{notD}(c_x), r/w(n_1), r_{notD}(c_x)\}, \\ \{w_D(c_x), r/w(n_2), r_D(c_x)\} \end{aligned} \tag{15.2}$$

To minimize the test time, *Intra-word* and *Inter-word* testing are properly interleaved. In fact, for any pair of adjacent cells $\mathrm{c_x}$, $\mathrm{n_j}$, a different background pattern is used. Whenever the number of needed patterns and the number of neighborhoods differ, the background patterns or the test of neighborhood pairs are repeated accordingly, to fill up the gap.

The adopted algorithm does not specifically target Address decoder faults. Nevertheless, the following address faults are covered:

- $\mathrm{c_x}$ is not addressable because there is no link between the address decoder and the enable signal of the cell;

- c_x is not addressable, and address(c_x) accesses n_j;
- c_x is not addressed by address(c_x), and both n_j and c_x are reached by address(n_j);
- c_x is addressed by address(c_x), but both n_j and c_x are reached by address(n_j).

According to FlexStream tool, the memory is considered organized by columns: cells that are adjacent inside a column have consecutive address into the memory. The functional address of the neighborhoods n_j can be easily computed based on the address of c_x, as follows:

$$\{address(n_j)|0 \leq j \leq 7 = \begin{cases} address(c_x) \pm \#Rows\pm, j \in \{0,2,4,6\} \\ addres(c_x) \pm 1, j \in 1,5 \\ address(c_x) \pm \#Rows, j \in \{3,7\} \end{cases} \tag{15.3}$$

To reduce the complexity of the BIST controller, the memory is conceptually considered as a toroid, thus assuming the most left-hand and the most right-hand columns being adjacent as well as the top and the bottom rows. In such a way the test length is slightly increased (coupling faults with a very low occurrence probability are tested), but the controller size is significantly reduced.

The proposed algorithm has a complexity of (8*14) $N = 112\ N$, being N the number of memory cells, and 14 the number of memory accesses performed on each pair of neighborhood cells.

During testing, the cell under test c_x and one of its neighborhood cells n_j are isolated by replacing them with two spare ones: their original content is copied into the spare cells and the CAM content updated for address re-mapping. The test algorithm is then executed on the pair c_x, n_j. If no faults are detected, the original content of n_j is restored and its re-mapping address in the CAM removed. Then, the next neighborhood cell is considered. If the test is successful for all the eight pairs c_x, n_j, then c_x is restored and the next cell of the memory is set under test. Otherwise, if c_x is found faulty, the test is repeated on the same pair of cells to distinguish between permanent and transient faults. If the repeated test fails as well, c_x is considered as a permanent faulty cell and its functional replacement by the spare cell is not removed. In this way, c_x is no longer accessible, and the memory functionally repaired.

15.3 Actual Implementation

To minimize the address critical path of Figure 15.1, the CAM structure has been implemented using a register array and a proper encoding logic, thus allowing obtaining the actual implementation of Figure 15.3. There is no a-priori constraint about the type of CAM one can use. Using a non-volatile memory would keep the addressing space reconfiguration status when the system is powered-down. The combinational logic has been therefore synthesized fixing as a constraint the minimum propagation delay. The only performance degradation introduced w.r.t. the original memory protocol is a constant increase of the set up time of the memory corresponding to the time required to propagate the address.

Three output signals (`Faulty_Data`, `Stop_Reparing`, and `Repaired`) are provided to further increase the dependability properties of the core:

- `Faulty_Data` is asserted to point out that the data read by the user are potentially corrupted. After a permanent repair, the signal is asserted whenever the user read the content of a spare cell without having written it previously. In such a case, in fact, the content of the cell can be faulty. The signal is reset by the first writing operation on the replacing cell.

- `Stop_Reparing` is asserted whenever no additional spare cells are available to execute the test. Thereinafter the `SR_only` functional mode (see section 15.1) is entered.

- `Repaired` is asserted whenever the addressed cell has been previously found faulty and replaced by a spare one. The signal can be helpful, for instance, for diagnosis purposes. To perform the diagnosis of the memory the user can:
  - force the module to enter the `SR_only` mode;
  - perform a read operation on any cell of the module;
  - for each address, check whether the `Repaired` is asserted or not. If asserted, the target cell is faulty.

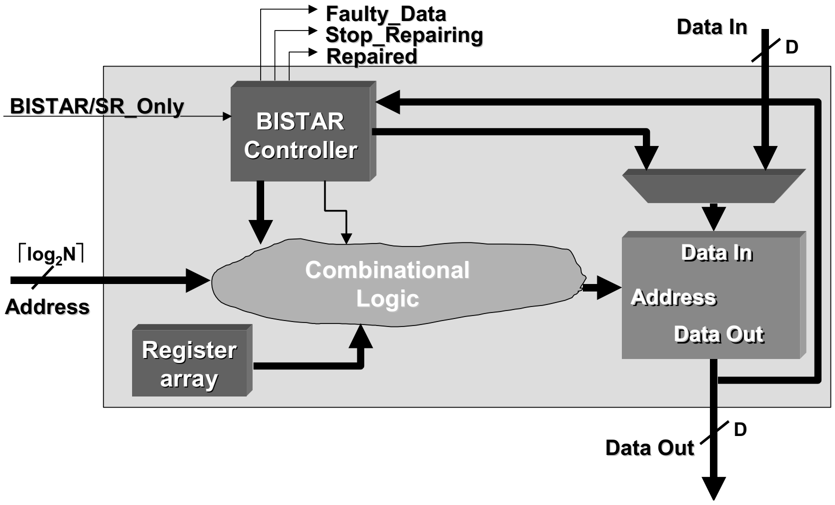

Faulty_Data
Stop_Repairing
Repaired
Data In
D
BISTAR/SR_Only
BISTAR Controller
$\lceil \log_2 N \rceil$
Address
Combinational Logic
Data In
Address
Data Out
Register array
Data Out
D

Figure 15.3. Minimization of the address critical path

15.4 BISTAR validation

To validate the proposed BISTAR architecture, it is necessary to demonstrate that faults appearing into the memory are detected by the BIST circuitry, and that the BISR logic adequately reconfigures the memory address space. The adopted methodology consists in the following three steps:

1. One or more faults are injected into both the memory cells and addressing logic;
2. An interval time is waited, until the BISTAR architecture localizes and repairs the fault;
3. The memory is exercised to verify its correct behavior after the repairing process.

To emulate a faulty memory, an ad-hoc memory wrapper has been designed able to intercept data and address flows coming *in* and *out* of the memory, and to modify them according to a predefined fault model. The proposed approach is extremely flexible, since the wrapper functionality is customizable to each specific experiment, defining, at simulation time, the type and the number of faults to be injected into the memory.

The Fault Injector is described in VHDL as a set of modular blocks, each allowing the insertion of a particular fault.

To verify the correct memory behavior after each fault injection and the consequent repair process, a MATS$^+$ March Test algorithm [69] is performed.

15.5 Experimental results

The experimental results presented in this section have been gathered on several implementations of various-sized BISTAR architectures built around a static RAM m10p111hab, included in the LSI Logic^{TM} G10 library. In particular, to deeply analyze the impact in terms of area overhead, all the possible combinations of the following cores have been synthesized by Synopsys^{TM} using the above-mentioned G10 library:

- 8, 16, and 32 bits word;
- 1k, 2k, 4k and 8k words;
- 16, 32 and 64 spare cells.

15.5.1 Area overhead

Table 15.1 collects the synthesis results concerning the area overhead, expressed as percentage of the area added to achieve BISR and BIST functionality's w.r.t. the area of the original memory module. Spare cells are considered as part of the BISR logic and therefore contribute to the area overhead.

Table 15.1. Percentage of Area overhead introduced by the BISTAR Architecture

		Spare Cells		
Number of Bits word	**Words**	**16**	**32**	**64**
8-bits word	1K	36.52%	68.45%	134.26%
	2K	22.81%	42.82%	83.98%
	4K	13.28%	24.98%	48.99%
	8K	7.47%	13.94%	27.34%
16-bits word	1K	22.47%	41.08%	79.48%
	2K	14.25%	26.14%	50.62%
	4K	8.38%	15.43%	29.90%
	8K	4.86%	8.89%	17.24%
32-bits word	1K	12.85%	22.74%	43.06%
	2K	8.39%	14.93%	28.32%
	4K	4.66%	8.32%	15.82%
	8K	2.62%	4.66%	8.87%

To provide the designer with a quick (although approximate) estimation of the area overhead, let's consider a memory of N words, K spare cells and D bits word. The percentage of area overhead introduced by the BISR and BIST circuitry is:

$$\begin{aligned} AreaOverhead\% = & \left(\frac{Area_{BISRandBIST}}{Area_{RAM}} \cdot 100\right)\% \approx \\ \approx & \left(\frac{Area_{spare}+Area_{CAM}+Area_{BISTAR}+Area_{routing}}{Area_{RAM}} \cdot 100\right)\% \end{aligned} \tag{15.4}$$

where:

$$\begin{cases} Area_{RAM} = O(N \cdot D) \\ Area_{spare} = O(K \cdot D) \\ Area_{CAM} = O(K \cdot log_2 N) \\ Area_{BISTAR} = O(log_2 N)(assuming N > D) \end{cases} \tag{15.5}$$

Taking into account Equation 15.4, the expression Equation 15.5 can be reduced to:

$$AreaOverhead\% = \left\{\left(\frac{D + c \cdot log_2 N}{D \cdot N}\right) \cdot K \cdot 100\right\}\% \tag{15.6}$$

For a given memory, the term in rounded parenthesis is a constant, N and D being fixed. As expected, Equation 15.6 states that the area overhead is proportional to the number of spare cells K. The constant parameter c in Equation 15.5 strongly depends on the target synthesis library. When dealing with the LSI Logic G10 library it has been experimentally proved to be about 22.

15.5.2 BISR Logic Fault Coverage

In order to evaluate the repair capabilities of the BISR logic the following experiment consisting in three steps has been setup:

1. One fault is injected in the BISR logic using the Sunrise tool;
2. An interval time is waited, to allow the BISR architecture to localize the fault and consequently reconfigure the memory:
3. The memory is tested using a March test in order to verify its correct behavior after the reconfiguration.

Although, the RAM core has not been explicitly designed to cover faults located in the BISR logic, the module was able to repair the 92.7% of the faults inserted in the BISR Controller and the 89.16% of the faults inserted in the remaining part of the BISR logic (Table 15.2).

Table 15.2. BISR Logic Fault Coverage

Module	Total	repaired
BISR Controller	972	92.7%
BISR Logic (CAM, etc.)	2224	89.16%

Chapter 16

Conclusions

In this part of the thesis the problem of massive use of memories in complex SoCs has been analyzed. Four major problems have been addressed: (i) memory-BIST, (ii) Automatic Test Generation, (iii) Test Verification, (iv) Automatic Repair. For each of the above mentioned problems a possible solution has been proposed giving experimental results to demonstrate the effectiveness of the solutions.

Appendix A

Publication list

A.1 International journals

A.1.1 Year 2003

- [12] A. Benso, S. Di Carlo, G. Di Natale and P. Prinetto, "Online self-repair of FIR filters" in IEEE Design & Test of Computers, vol. 20, no. 3, Page(s) 50-57, May-June 2003.

- [18] A. Benso, S. Di Carlo, P. Prinetto and Y. Zorian, "A hierarchical infrastructure for SoC test management," in IEEE Design & Test of Computers, vol. 20, no. 4, Page(s) 32-39, July-Aug. 2003.

- [13] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto and M. L. Bodoni, "Programmable built-in self-testing of embedded RAM clusters in system-on-chip architectures" in IEEE Communications Magazine, vol. 41, no. 9, Page(s) 90-97, Sept. 2003

A.2 International conferences

A.2.1 Year 2002

- [10] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "Specification and Design of a new Memory Fault Simulator", IEEE Asian Test Symposium (ATS 2002), November 2002, Page(s): 92 - 97.

- [11] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "Static Analysis of SEU Effects on Software Applications", IEEE International test Conference (ITC'02), Baltimore (Maryland), October 2002, Page(s): 500-508.

- [32] S. Chiusano, S. Di Carlo, P. Prinetto, "Automated Synthesis of SEU Tolerant Architectures from OO Description", IEEE International On Line Test Workshop (IOLTW'02), Isle of Bendor, France, July 2002, Page(s): 26-31.

- [9] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "An Optimal Algorithm for the Automatic Generation of March Tests", IEEE Design Automation and Test in Europe (DATE'02), Paris (F), March 2002, Page(s): 938-943.

A.2.2 Year 2001

- [7] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "Memory Read Faults: Taxonomy and Automatic March Test Generation", IEEE Asian Test Symposium (ATS'01), Kyoto (Japan), November 2001, Page(s): 157-163.
- [15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, "Control-Flow Checking via Regular Expression", IEEE Asian Test Symposium (ATS'01), Kyoto (Japan), November 2001, Page(s): 299-303.
- [16] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, "Software Dependability Techniques validated via Fault Injection Experiments", IEEE RADiation and its Effects on Components and Systems Conference (RADECS'01), Grenoble (F), November 2001, Page(s): 299-303.
- [17] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, "Validation of a Software Dependability Tool Via Fault Injection Experiments", IEEE International On Line Testing Workshop (IOLTW'01), Taormina (Italy), July 2001, Page(s): 3-8.
- [33] S. Chiusano, S. Di Carlo, P. Prinetto, H.J. Wunderlich, "On Applying the Set Covering Model to Reseeding", IEEE Design Automation and Test in Europe (DATE'01), Munich (G), March 2001, Page(s): 156-160.
- [8] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "SEU Effect Analysis in a Open-Source Router via a Distributed Fault Injection Environment", IEEE Design Automation and Test in Europe (DATE'01), Munich (G), March 2001, Page(s): 219-223.

A.2.3 Year 2000

- [14] A. Benso, S. Chiusano, S. Di Carlo, G. Di Natale, M. Lobetti-Bodoni, P. Prinetto, "A programmable BIST architecture for clusters of Multiple-Port SRAMs", IEEE International Test Conference (ITC'00), Atlantic City (NJ), USA, October 2000, Page(s): 557-566.
- [19] A. Benso, S. Di Carlo, S. Chiusano, P. Prinetto, F. Ricciato, M. Spadari, Y. Zorian, "HD2BIST: a Hierarchical Framework for BIST Scheduling, Data patterns delivering and diagnosis in SoCs", IEEE International Test Conference (ITC'00), Atlantic City (NJ), USA, October 2000, Page(s): 892-901.
- [27] A. Benso, S. Chiusano, S. Di Carlo, G. Di Natale, P. Prinetto, M. Lobetti-Bodoni, "An effective distributed BIST architecture for RAMs", IEEE European Test Workshop, Lisbon (P), May 2000, Page(s): 119-124.

- [6] A. Benso, S. Chiusano, S. Di Carlo, M. Lobetti-Bodoni, P. Prinetto, M. Spadari, Y. Zorian, "On Integrating a Proprietary and a Commercial Architecture for Optimal BIST Performances in an Industrial SoCs", IEEE International Conference on Computer Design (ICCD'00), Austin (TX), USA, September 2000, Page(s): 539-540.

Bibliography

[1] Miron Abramovici, Melvin A Breuer, and Arthur D Friedman. *Digital systems testing and testable design*, volume 2. Computer science press New York, 1990.

[2] Saman Adham, Nortel Debashis, Bhattacharya Ti, Dwayne Burek Logicvision, CJ Clark, Intellitech Mike, Collins Cisco, et al. Preliminary outline of the ieee p1500 scalable architecture for testing embedded cores. In *in Proc. IEEE VLSI Test Symposium (VTS), IEEE Computer*. Citeseer, 1999.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[4] Owen S Bair, Adam Kablanian, Charles Li, and Farzad Zarrinfar. Method and apparatus for configurable build-in self-repairing of asic memories design, November 19 1996. US Patent 5,577,050.

[5] Frans Beenker, Rob Dekker, Richard Stans, and Max Van der Star. Implementing macro test in silicon compiler design. *IEEE Design & Test of Computers*, 7(2):41–51, 1990.

[6] A. Benso, S. Di Carlo, S. Chiusano, P. Prinetto, F. Ricciato, M. L. Bodoni, and M. Spadari. On integrating a proprietary and a commercial architecture for optimal bist performances in socs. In *Proceedings 2000 International Conference on Computer Design*, pages 539–540, 2000.

[7] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. Memory read faults: taxonomy and automatic test generation. In *Proceedings 10th Asian Test Symposium*, pages 157–163, 2001.

[8] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. Seu effect analysis in a open-source router via a distributed fault injection environment. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 219–223, 2001.

[9] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. An optimal algorithm for the automatic generation of march tests. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 938–943, 2002.

[10] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. Specification and design of a new memory fault simulator. In *Test Symposium, 2002. (ATS '02). Proceedings of the 11th Asian*, pages 92–97, Nov 2002.

[11] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. Static analysis of seu effects on software applications. In *Proceedings. International Test Conference*, pages 500–508, 2002.

[12] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. Online self-repair of fir filters. *IEEE Design Test of Computers*, 20(3):50–57, May 2003.

[13] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and M. L. Bodoni. Programmable built-in self-testing of embedded ram clusters in system-on-chip architectures. *IEEE Communications Magazine*, 41(9):90–97, Sept 2003.

[14] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and M. Lobetti Bodoni. A programmable bist architecture for clusters of multiple-port srams. In *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, pages 557–566, 2000.

[15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. Control-flow checking via regular expressions. In *Proceedings 10th Asian Test Symposium*, pages 299–303, 2001.

[16] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. Software dependability techniques validated via fault injection experiments. In *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No.01TH8605)*, pages 269–274, Sept 2001.

[17] A. Benso, S. Di Carlo, G. Di Natale, L. Tagliaferri, and P. Prinetto. Validation of a software dependability tool via fault injection experiments. In *Proceedings Seventh International On-Line Testing Workshop*, pages 3–8, 2001.

[18] A. Benso, S. Di Carlo, P. Prinetto, and Y. Zorian. A hierarchical infrastructure for soc test management. *IEEE Design Test of Computers*, 20(4):32–39, July 2003.

[19] A. Benso, S. Chiusano, S. Di Carlo, P. Prinetto, F. Ricciato, M. Spadari, and Y. Zorian. Hd2bist: a hierarchical framework for bist scheduling, data patterns delivering and diagnosis in socs. In *Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159)*, pages 892–901, 2000.

[20] A. Benso, S. Chiusano, G. Di Natale, and P. Prinetto. An on-line bist ram architecture with self-repair capabilities. *IEEE Transactions on Reliability*, 51(1):123–128, Mar 2002.

[21] Alfredo Benso, Silvia Cataldo, Silvia Chiusano, Paolo Prinetto, and Yervant Zorian. Hd-bist: a hierarchical framework for bist scheduling and diagnosis in socs. In *Test Conference, 1999. Proceedings. International*, pages 1038–1044. IEEE, 1999.

[22] Alfredo Benso, Silvia Cataldo, Silvia Chiusano, Paolo Prinetto, and Yervant Zorian. A high-level eda environment for the automatic insertion of hd-bist structures. *Journal of Electronic Testing*, 16(3):179–184, 2000.

[23] Alfredo Benso, Silvia Chiusano, Stefano Di Carlo, Paolo Prinetto, Fabio Ricciato, Maurizio Spadari, and Yervant Zorian. Hd/sup 2/bist: a hierarchical framework for bist scheduling, data patterns delivering and diagnosis in socs. In *Test Conference, 2000. Proceedings. International*, pages 892–901. IEEE, 2000.

[24] Debashis Bhattacharya. Hierarchical test access architecture for embedded cores in an integrated circuit. In *VLSI Test Symposium, 1998. Proceedings. 16th IEEE*, pages 8–14. IEEE, 1998.

[25] Dilip K Bhavsar. An algorithm for row-column self-repair of rams and its implementation in the alpha 21264. In *Test Conference, 1999. Proceedings. International*, pages 311–318. IEEE, 1999.

[26] Ray Bittner and Peter Athanas. Wormhole run-time reconfiguration. In *Proceedings*

of the 1997 ACM fifth international symposium on Field-programmable gate arrays, pages 79–85. ACM, 1997.

[27] M. Lobetti Bodoni, A. Benso, S. Chiusano, S. Di Carlo, G. Di Natale, and P. Prinetto. An effective distributed bist architecture for rams. In *Proceedings IEEE European Test Workshop*, pages 119–124, 2000.

[28] Janusz A Brzozowski and Bruce F Cockburn. Detection of coupling faults in rams. *Journal of Electronic Testing*, 1(2):151–162, 1990.

[29] Janusz A Brzozowski and Helmut Jürgensen. A model for sequential machine testing and diagnosis. *Journal of Electronic Testing*, 3(3):219–234, 1992.

[30] G. Carpaneto, E. Dell'Amico, and I. Toth. A branch-and-bound algorithm for large scale asymmetric traveling salesman prob-lems. ftp://netlib2.cs.utk.edu/toms/index.html, Technical Report Dipartimento di Economia Politica, Facoltà di Economia e Commericio, Modena University, 1990.

[31] Tom Chen and Glen Sunada. Design of a self-testing and self-repairing structure for highly hierarchical ultra-large capacity memory chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):88–97, 1993.

[32] S. Chiusano, S. Di Carlo, and P. Prinetto. Automated synthesis of seu tolerant architectures from oo descriptions. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, pages 26–31, 2002.

[33] S. Chiusano, S. Di Carlo, P. Prinetto, and H. J. Wunderlich. On applying the set covering model to reseeding. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 156–160, 2001.

[34] J. R. Day. A fault-driven, comprehensive redundancy algorithm. *IEEE Design Test of Computers*, 2(3):35–44, June 1985.

[35] S. Demidenko, A. van de Goor, S. Henderson, and P. Knoppers. Simulation and development of short transparent tests for ram. In *Proceedings 10th Asian Test Symposium*, pages 164–169, 2001.

[36] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-J rg Kreowski. *Handbook of graph grammars and computing by graph transformation*, volume 3. world Scientific, 1999.

[37] Yield Enhancement. International technology roadmap for semiconductors. [Online:] http://public.itrs.net/Files/1997UpdateFinal/ORTC1997final.pdf, 2000 Update.

[38] Indradeep Ghosh, Sujit Dey, and Niraj K Jha. A fast and low-cost testing technique for core-based system-chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):863–877, 2000.

[39] Indradeep Ghosh, Niraj K Jha, and Sujit Dey. A low overhead design for testability and test generation technique for core-based systems. In *Test Conference, 1997. Proceedings., International*, pages 50–59. IEEE, 1997.

[40] Alan Gibbons. *Algorithmic graph theory.* Cambridge university press, 1985.

[41] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, pages 250–256. ACM, 2000.

[42] Oliver F Haberl and Thomas Kropf. Hist: A methodology for the automatic insertion of a hierarchical self test. In *Test Conference, 1992. Proceedings., International*, page 732. IEEE, 1992.

[43] Ramsey W. Haddad, Anton T. Dahbura, and Anup B. Sharma. Increased throughput for the testing and repair of rams with redundancy. *IEEE Transactions on Computers*, 40(2):154–166, 1991.

[44] S. Hamdioui and A. J. van de Goor. Consequences of port restrictions on testing two-port memories. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pages 63–72, Oct 1998.

[45] Richard W Hamming. *Coding and Theory.* Prentice-Hall, Englewood Cliffs, 1980.

[46] Nany Hasan and CL Liu. Minimum fault coverage in reconfigurable arrays. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 348–353. IEEE, 1988.

[47] Venkata Immaneni and Srinivas Raman. Direct access test scheme-design of block and core cells for embedded asics. In *Test Conference, 1990. Proceedings., International*, pages 488–492. IEEE, 1990.

[48] Adam Kablanian, Thomas P Anderson, Chuong T Le, Owen S Bair, and Saravana Soundararajan. Built-in self repair system for embedded memories, June 9 1998. US Patent 5,764,878.

[49] Rohit Kapur, Brion Keller, Bernd Koenemann, Maurice Lousberg, Paul Reuter, Tony Taylor, and Prab Varma. P1500-ctl: towards a standard core test language. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 489–490. IEEE, 1999.

[50] Ilyoung Kim, Yervant Zorian, Goh Komoriya, Hai Pham, Frank P Higgins, and Jim L Lewandowski. Built in self repair for embedded high density sram. In *Test Conference, 1998. Proceedings., International*, pages 1112–1119. IEEE, 1998.

[51] John Lach, William H Mangione-Smith, and Miodrag Potkonjak. Efficiently supporting fault-tolerance in fpgas. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 105–115. ACM, 1998.

[52] Logic Vision. Logic vision website. [Online] http://www.logicvision.com, Feb. 2000.

[53] LSILogic. Flexstream reference manual, Mar. 1999.

[54] William H Mangione-Smith and Brad L Hutchings. Configurable computing: The road ahead. In *Reconfigurable Architectures Workshop*, pages 81–96, 1997.

[55] Erik Jan Marinissen, Robert Arendsen, Gerard Bos, Hans Dingemanse, Maurice Lousberg, and Clemens Wouters. A structured and scalable mechanism for test access to embedded reusable cores. In *Test Conference, 1998. Proceedings., International*, pages 284–293. IEEE, 1998.

[56] Erik Jan Marinissen and Maurice Lousberg. Macro test: A liberal test approach for embedded reusable cores. In *Digest of Papers of IEEE International Workshop on Testing Embedded Core-Based Systems (TECS)*, pages 1–2, 1997.

[57] Mentor Graphics. Mentor graphics web site. [Online] http://www.mentrog.com/dft, Feb. 2000.

[58] Mohsen Nahvi and Andre Ivanov. A packet switching communication-based test access mechanism for system chips. In *Proceedings of the IEEE European Test Workshop (ETW'01)*, page 81. IEEE Computer Society, 2001.

[59] M. Nicolaidis, V. Castro Alves, and H. Bederr. Testing complex couplings in multiport memories. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):59–71, March 1995.

[60] Dirk Niggemeyer, Michael Redeker, and Elizabeth M Rudnick. Diagnostic testing of embedded memories based on output tracing. In *Memory Technology, Design and Testing, 2000. Records of the 2000 IEEE International Workshop on*, pages 113–118. IEEE, 2000.

[61] Institute of Electrical and Electronics Engineers. *IEEE Standard Test Access Port and Boundary-scan Architecture: Approved February 15, 1990, IEEE Standards Board; Approved June 17, 1990, American National Standards Institute*. IEEE, 1990.

[62] M. Spadari, T. Vaida, and P. Ghosh. A hierarchical test methodology involving multiple embedded cores having different dft mechanisms. In *Digest of Papers of IEEE International Workshop on Testing Embedded Core-Based SyTems*, 1998.

[63] Akira Tanabe, Toshio Takeshima, Hiroki Koike, Yoshiharu Aimoto, Masahide Takada, Toshiyuki Ishijima, Naoki Kasai, Hiromitsu Hada, Kentaro Shibahara, Tahemitsu Kunio, et al. A 30-ns 64-mb dram with built-in self-test and self-repair function. *IEEE Journal of Solid-State Circuits*, 27(11):1525–1533, 1992.

[64] Edward Tau, Derrick Chen, Ian Eslick, and Jeremy Brown. A first generation dpga implementation. In *In Proceedings of the Third Canadian Workshop on Field-Programmable Devices*. Citeseer, 1995.

[65] Nur A Touba and Bahram Pouya. Testing embedded cores using partial isolation rings. In *VLSI Test Symposium, 1997., 15th IEEE*, pages 10–16. IEEE, 1997.

[66] Robert Treuer and Vinod K Agarwal. Built-in self-diagnosis for repairable embedded rams. *IEEE Design & Test of Computers*, 10(2):24–33, 1993.

[67] VSI `http://www.vsi.org/library/specs.htm`. Vsi virtual socket interface web site, architecture document. [Online], Mar. 1997.

[68] A. J. van de Goor and I. B. S. Tlili. March tests for word-oriented memories. In *Proceedings Design, Automation and Test in Europe*, pages 501–508, Feb 1998.

[69] Ad J Van de Goor. *Testing semiconductor memories: theory and practice*. John Wiley & Sons, Inc., 1991.

[70] Ad J Van De Goor. Using march tests to test srams. *IEEE Design & Test of Computers*, 10(1):8–14, 1993.

[71] Ad J Van de Goor and Issam BS Tlili. March tests for word-oriented memories. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 501–508. IEEE, 1998.

[72] AJ Van De Goor and B Smit. Automatic verification of march tests (srams). In *Memory Testing, 1993., Records of the 1993 IEEE International Workshop on*, pages 131–136. IEEE, 1993.

[73] AJ Van De Goor and B Smit. The automatic generation of march tests. In *Memory Technology, Design and Testing, 1994., Records of the IEEE International Workshop on*, pages 86–91. IEEE, 1994.

[74] AJ Van de Goor and B Smit. Generating march tests automatically. In *Test Conference, 1994. Proceedings., International*, pages 870–878. IEEE, 1994.

[75] Prab Varma. Evolving strategies for testing systems on silicon. *Integrated System Design-Virtual Chip Design Supplement*, 1997.

[76] Prab Varma and B Bhatia. A structured test re-use methodology for core-based system chips. In *Test Conference, 1998. Proceedings., International*, pages 294–302.

IEEE, 1998.

[77] Lee Whetsel. An ieee 1149.1-based test access architecture for ics with embeddded cores. In *Proceedings of the 1997 IEEE International Test Conference*, page 69. IEEE Computer Society, 1997.

[78] Michael J Wirthlin and Brad L Hutchings. A dynamic instruction set computer. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 99–107. IEEE, 1995.

[79] Chi-Feng Wu, Chih-Tsun Huang, and Cheng-Wen Wu. Ramses: a fast memory fault simulator. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*, pages 165–173, Nov 1999.

[80] Kamran Zarrineh, Shambhu J Upadhyaya, and Sreejit Chakravarty. A new framework for generating optimal march tests for memory arrays. In *Test Conference, 1998. Proceedings., International*, pages 73–82. IEEE, 1998.

[81] Yervant Zorian. A distributed bist control scheme for complex vlsi devices. In *VLSI Test Symposium, 1993. Digest of Papers., Eleventh Annual 1993 IEEE*, pages 4–9. IEEE, 1993.

[82] Yervant Zorian. Test requirements for embedded core-based systems and ieee p1500. In *Test Conference, 1997. Proceedings., International*, pages 191–199. IEEE, 1997.

[83] Yervant Zorian. Testing the monster chip. *IEEE Spectrum*, 36(7):54–60, 1999.

[84] Yervant Zorian. Emerging trends in vlsi test and diagnosis. In *VLSI, 2000. Proceedings. IEEE Computer Society Workshop on*, pages 21–27. IEEE, 2000.

[85] Yervant Zorian. Embedding infrastructure ip for soc yield improvement. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 709–712. IEEE, 2002.

[86] Yervant Zorian, Sujit Dey, and Michael J Rodgers. Test of future system-on-chips. In *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, pages 392–398. IEEE, 2000.

[87] Yervant Zorian, Erik Jan Marinissen, and Sujit Dey. Testing embedded-core based system chips. In *Test Conference, 1998. Proceedings., International*, pages 130–143. IEEE, 1998.