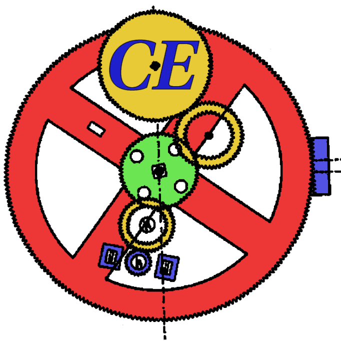

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Fine-Grain Runtime Fault Diagnosis for Reconfigurable Logic Blocks

Stavros Tzilis

CE-MS-2010-13

Abstract

The ever-shrinking technology features have as a direct consequence the increase of defect density in VLSI chips. Going into the nano-scale era, the fabrication procedures cannot keep improving at the pace of the aforementioned shrinking of technology features. Fault Tolerance emerges as a much cheaper solution and it is imperative in the future to be able to build a reliable system with unreliable components. Reconfigurable realization platforms offer the ideal substrate for such approaches, because of their regularity and reconfigurability, which allow for the basic resources to be substitutable, relaxing the defect-free requirement for the whole chip. Sparing and matching techniques allow for substitution and alternative utilization of resources respectively, paving the way to the nano-scale era.
Although a significant number of research works have focused on sparing, very few actually go on to reusing the defective resources and even in these cases, the characterization is conservative, sacrificing more functionality than it needs to. We focus on improving the particular drawback, by proposing two distinct methods for high resolution fault diagnosis of reconfigurable logic resources. The methods are based on the function generator and shift register modes of operation of an FPGA slice. We choose to decouple the diagnosis problem from those of fault detection and localization that have been extensively researched and in this way relax the fault coverage requirements for our methods: It is critical to rescue the core functionality of a defective resource with minimal cost, rather than cover 100% of its possible faults. Substitutable Resource Characterization is performed based on the diagnosis result in a modular manner. Both diagnostic testers are prototyped on FPGA and applied to a real Circuit Under Test, with the help of fault injection. The experimental results show that our approach offers the basis for a viable, low-overhead integrated fault tolerance strategy, which we hope to continue developing in the near future.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Fine-Grain Runtime Fault Diagnosis for Reconfigurable Logic Blocks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Stavros Tzilis
born in Heraklion, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Fine-Grain Runtime Fault Diagnosis for Reconfigurable Logic Blocks

by Stavros Tzilis

Abstract

The ever-shrinking technology features have as a direct consequence the increase of defect density in VLSI chips. Going into the nano-scale era, the fabrication procedures cannot keep improving at the pace of the aforementioned shrinking of technology features. Fault Tolerance emerges as a much cheaper solution and it is imperative in the future to be able to build a reliable system with unreliable components. Reconfigurable realization platforms offer the ideal substrate for such approaches, because of their regularity and reconfigurability, which allow for the basic resources to be substitutable, relaxing the defect-free requirement for the whole chip. Sparing and matching techniques allow for substitution and alternative utilization of resources respectively, paving the way to the nano-scale era.

Although a significant number of research works have focused on sparing, very few actually go on to reusing the defective resources and even in these cases, the characterization is conservative, sacrificing more functionality than it needs to. We focus on improving the particular drawback, by proposing two distinct methods for high resolution fault diagnosis of reconfigurable logic resources. The methods are based on the function generator and shift register modes of operation of an FPGA slice. We choose to decouple the diagnosis problem from those of fault detection and localization that have been extensively researched and in this way relax the fault coverage requirements for our methods: It is critical to rescue the core functionality of a defective resource with minimal cost, rather than cover 100% of its possible faults. Substitutable Resource Characterization is performed based on the diagnosis result in a modular manner. Both diagnostic testers are prototyped on FPGA and applied to a real Circuit Under Test, with the help of fault injection. The experimental results show that our approach offers the basis for a viable, low-overhead integrated fault tolerance strategy, which we hope to continue developing in the near future.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-13

Committee Members :

Advisor:	Dr. Ioannis Sourdis, Assistant Prof., CE, TU Delft
Advisor:	Dr.ir. Georgi N. Gaydadjiev, Assistant Prof., CE, TU Delft
Chairperson:	Dr. Koen Bertels, Associate Prof., CE, TU Delft
Member:	Dr. K.G. Langendoen, Prof., Software Technology, TU Delft

In loving memory of Stavros P. Tzilis Senior (1923-2009)

Contents

List of Figures viii

List of Tables ix

Acknowledgements xi

Achievements xiii

1 Introduction 1
- 1.1 Thesis Incentive . . . 2
- 1.2 Problem Statement . . . 4
- 1.3 Thesis Goals . . . 5
- 1.4 Thesis Overview . . . 6

2 Background 7
- 2.1 The Testing Procedure . . . 7
  - 2.1.1 Fault Models . . . 7
  - 2.1.2 Testing, Localization, Diagnosis and Repair . . . 10
  - 2.1.3 FPGA Testing . . . 15
- 2.2 Fault Tolerance . . . 19
  - 2.2.1 Sparing and Matching . . . 19
  - 2.2.2 The FaTES project framework . . . 21
- 2.3 Summary and Thesis Significance . . . 22

3 Method 25
- 3.1 The Virtex-II FPGA Slice . . . 26
  - 3.1.1 Slice simplified model . . . 27
- 3.2 The Function Generator Based Method . . . 35
  - 3.2.1 Partitioning of the Faults . . . 35
  - 3.2.2 Deriving the LUT Configurations . . . 37
  - 3.2.3 Deriving the Diagnostic Trees . . . 39
- 3.3 The Shift Register Based Method . . . 49
  - 3.3.1 Partitioning of the Faults . . . 50
  - 3.3.2 The Memory Test . . . 51
  - 3.3.3 Deriving the Diagnostic Trees . . . 55
- 3.4 Characterization and Graceful Degradation . . . 61
- 3.5 Summary . . . 63

4 Hardware Implementation **67**
4.1 System Organization . . . 67
4.2 Function and Implementation of modules . . . 68
4.2.1 Response Analyzers . . . 68
4.2.2 System Controllers . . . 70
4.3 Other Implementation Details . . . 72
4.3.1 Wrapping Identical Response Analyzers . . . 72
4.3.2 Fault and Mode Encoding . . . 74
4.3.3 System pipeline . . . 75
4.4 Summary . . . 75

5 Evaluation **79**
5.1 Experimental Setup . . . 79
5.1.1 Simulation . . . 79
5.1.2 Emulation . . . 80
5.1.3 Realization and Prototyping . . . 81
5.2 Quantitative Measures . . . 82
5.3 Qualitative Measures . . . 89
5.3.1 Scalability . . . 89
5.3.2 Adaptability . . . 90
5.3.3 Applicability . . . 90
5.3.4 Graceful Degradation . . . 91
5.4 Design for Testability Guidelines . . . 91
5.5 Summary . . . 93

6 Conclusion **95**
6.1 Summary . . . 95
6.2 Thesis Contributions . . . 97
6.3 Future Work Suggestions . . . 98
6.4 Conclusion . . . 99

Bibliography **104**

List of Figures

1.1 The elementary majority voter. 2
1.2 Shrinking technology features will affect process variation in the future . . 3
1.3 Aging Effects are going to become more intense in the future 3
1.4 Concepts of sparing and matching. 3
1.5 Concept of Substitutable Resource characterization. 4

2.1 Example circuit for Section 2.1 8
2.2 State diagrams for fault free and faulty memory cell 10
2.3 Verification and Testing 11
2.4 The BIST principle for FPGA testing. 17
2.5 A simple sparing strategy 20
2.6 The matching problem 21

3.1 Adaptive diagnosis strategy according to the diagnostic tree approach . . 26
3.2 The Virtex-II slice simplified diagram 28
3.3 The CUT partitioning for phase 1 of the function generator method . . . 36
3.4 The CUT partitioning for phase 2 of the function generator method . . . 37
3.5 The FX/Y diagnostic tree for the first phase 40
3.6 The F5 and X diagnostic tree for the first phase 41
3.7 The YQ diagnostic tree for the first phase 43
3.8 The XQ diagnostic tree for the first phase 44
3.9 The Y diagnostic tree for the second phase 47
3.10 The X diagnostic tree for the second phase 48
3.11 Partitioning of the CUT for the shift register method 51
3.12 The FX diagnostic tree for the shift register method 56
3.13 The Y and X diagnostic tree for the shift register method 57
3.14 Subtree 1 of the Y and X diagnostic tree for the shift register method . . 58
3.15 Subtree 2 of the Y and X diagnostic tree for the shift register method . . 59
3.16 Subtree 3 of the Y and X diagnostic tree for the shift register method . . 60
3.17 Subtree 4 of the Y and X diagnostic tree for the shift register method . . 61
3.18 The YQ and XQ diagnostic tree for the shift register method 62
3.19 The F5 diagnostic tree for the shift register method 63

4.1 Tester Block Diagram 68
4.2 Response Analyzer Implementation 69
4.3 System Controller Implementation 71
4.4 The RA Wrapper 73
4.5 Simple conversion of diagnosis result to mode of degradation 76
4.6 The System Pipeline 77

5.1 Response of the first phase for a fault free CUT 80
5.2 Response of the second phase for a fault free CUT 80
5.3 Detection of fault f21 with the function generator method 80

5.4 Detection of fault f61 with the function generator method 81
5.5 Response of the shift register method for a fault free CUT 81
5.6 Detection of fault f34 with the shift register method 81
5.7 Debugging scheme for the emulation and prototyping stages 83
5.8 Cumulative percentage of diagnosed faults on every cycle 86
5.9 Comparative Latency graph for the two methods 87
5.10 Comparative Latency graph for the two methods applied on a whole column of slices 87
5.11 Diagnostic Accuracy for both methods and various values of P_{faulty} . . . 88
5.12 Conceptual Graph comparing sparing and matching with different degrees of diagnostic resolution 92

List of Tables

2.1 Test Vectors and Faulty Responses for the example circuit 9
2.2 Memory faults considered in this thesis 10
2.3 Difference between faulty response and error response 14

3.1 Effects of address line faults on the effective read address 31
3.2 Multiplexer truth table 31
3.3 Complete list of faults 34
3.4 Detection of the address line faults according to the LUT configuration . 38
3.5 The shift register state on every cycle of the shift register method 54
3.6 Modes of graceful degradation 64

4.1 F5/X Wrapper logic truth table 74
4.2 X/Y Wrapper logic truth table 75
4.3 Fault and mode encoding for the F5/X Response Analyzer of the first phase 76

5.1 Distribution of diagnosed faults between cycles for the function generator method 84
5.2 Distribution of diagnosed faults between cycles for the shift register method 85
5.3 Average Latency for both methods and various values of P_{faulty} 86
5.4 Diagnostic Accuracy for both methods and various values of P_{faulty} . . . 87
5.5 Summary of quantitative measurements 88
5.6 Comparison of the two proposed methods 93

Acknowledgements

First of all, I would like to thank my supervisors for this thesis, Ioannis Sourdis and Georgi Gaydadjiev, for their help and support, as well as the faith they had in me over the whole long period that this thesis lasted. I was lucky and honored to work with you and hope to do so again in the future.

I am grateful to my family: My mother Lena, my father Prokopis and my beloved sister Maria for all their support and patience throughout these years. I hope they are proud of me as much as I am of them. Also I want to thank all my close relatives both in Giannitsa and Irakleio, for their thoughts and the times we had together, whenever I was able to visit throughout these 4 years.

I want to once more thank my previous diploma thesis supervisor, Dimitrios Soundris, for the chance he gave me at a difficult time and for his guidance that had as a result for me to come and study in Delft. Also, my thanks go to all the Professors of the Computer Engineering laboratory and other departments of TU Delft. I learned a little from every one of you and this work is the culmination of this knowledge.

A very special thanks goes to all the close friends I made during my stay in Delft: Angelos, Aris, Dimos, Eunice, Georgios, Katerina, Maria, Nikos Schalekis, Nikos Skalis, Petros and everyone else with whom I honestly and consciously shared happy and sad moments throughout this journey. They will all allow me to especially express my gratitude to Dimitris, George Smaragdos, George Stefanakis and Turhan for happening to be there, each one in his own way, when I needed it the most.

My greetings to all my CE colleagues of the year that entered in September 2006. I hope they are all happy wherever each of them is.

To all my friends in Greece, I don't know if you miss me as much as I do, but I will be seeing you for a while in two weeks from now. Let's hope that the future has in store more time for us in the same country.

Finally, I would like to thank the CE secretaries, Lidwina Tromp and Monique Tromp, as well as all the rest of the EWI administrative personnel for their assistance during this busy period of time.

Stavros Tzilis
Delft, The Netherlands
June 30, 2010

Achievements

The following conference paper was submitted and pending acceptance as a result of this MSc thesis:

- Stavros Tzilis, Ioannis Sourdis, Georgi N. Gaydadjiev, "*Fine-Grain Fault Diagnosis for FPGA Logic Blocks*", International Conference on Field-Programmable Technology (FPT), December 2010

1 Introduction

The reliance of today's world on computer systems is not something to argue about. The spread of personal computers is not as important a reason for the above fact, as the range of applications of Embedded Systems: Ranging from household applications to space exploration and from gaming consoles to biomedical systems, Embedded Computers are a dominant factor in shaping the character of modern societies.

The market of Embedded Systems is extremely diverse. Depending on the specific application, a multitude of factors change, resulting in variations of design and implementation strategies. That might increase the importance of finding engineers that are specialized in the target application, but at the same time makes engineers that have a solid knowledge of general design principles invaluable.

A very good example of applications with special design and implementation requirements is the set of applications that have at least one of the following characteristics:

- The system functions in a place where it is difficult to reach, in order to check, repair or substitute with a new one.
- The surrounding environment is hostile for digital systems, which affects the results of calculations.
- The application is safety-critical, meaning that the occurrence of an error, regardless of how probable this error is, can have catastrophic consequences.

Interestingly enough, two, or even all three of the above characteristics often coexist. For example, many environments that are hostile to electronics, are also hardly reachable by humans. Whether a system works in the bottom of the sea, or in space, it is subject to the above hindrances. Moreover, if the navigation system of a spacecraft fails, it might mean the premature end of the mission. The umbrella term that engulfs all strategies and technologies to overcome these difficulties is Fault Tolerance.

Some elementary examples of fault tolerance are error-correcting codes [23] and majority voting [38]. Error-correcting codes append each basic element of data with a few extra bits in such a way that one error among the bits of this data element can be detected and corrected. Majority voting allows a percentage of the components that compose the system to produce wrong results, temporarily or permanently, by performing the operations on three or more identical modules and choosing the result produced by most of them (Figure 1.1).

Fault tolerance techniques evolved over the years and also affected (and were affected by) the realization platforms. This thesis is particularly interested in realization platforms characterized by regularity, the most widespread example of which are FPGAs. FPGAs first appeared in 1985 and dominated the market, starting in the '90s and still going on until today [1]. The use of regular structures like FPGAs triggered the

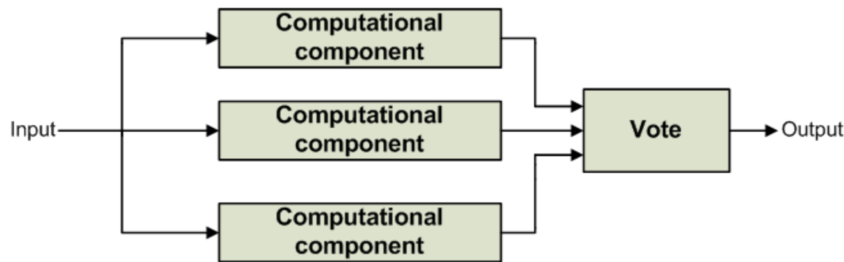

Figure 1.1: The elementary majority voter.

development of techniques that focused on the basic block of these structures. These techniques are called sparing and matching (see next section). Matching in particular is the motivation behind this thesis, as will be shortly explained. Driven by the matching paradigm, we develop and compare two different methods for characterizing faulty FPGA basic blocks, which means deciding which functions they can still perform, despite their faults.

The rest of this introductory chapter is organized as follows: In Section 1.1 we briefly explain the techniques of sparing and matching, the second of which is the incentive of this thesis. In Section 1.2 we state with precision the problem that we target. In Section 1.3 we list the goals we had when we started working on the particular problem. Finally, in Section 1.4 we present an outline of the rest of the thesis chapters.

1.1 Thesis Incentive

Fault Tolerance has gained a lot of ground in recent years. Except from the special characteristics of some applications, an important factor for this development is the shrinking of technology features. Borkar in [9] talks about designing reliable systems from unreliable components and argues about the outmost importance of orienting engineers towards this paradigm. It is mentioned that in the years to come it will be increasingly difficult to fabricate a defect-free chip, because the fabrication procedures cannot decrease the defect density to keep up with the increase of the number of transistors in a given area. It is also pointed out that aging effects (that is, elements that are not defective in fabrication time but will stop functioning during the system lifetime) will have as a result the degradation of the system with time, which has to be dealt with. These two trends are illustrated in Figures 1.2 and 1.3.

To face these new challenges and accomplish to build reliable systems, the main quality that researchers try to exploit is regularity. As already mentioned, researchers and engineers working on Fault Tolerance for regular realization platforms, have to also consider how important the regularity property is. The reason for that is that regularity makes it very easy to focus any method on the basic block, since this block is always the same. FPGAs in particular are based on Logic Cells, Slices and Configurable Logic Blocks (CLBs) and in principal, all blocks (i.e., all Slices) of the same device are identical. These basic blocks, in the fault tolerance context, are generally called *Substitutable Resources*.

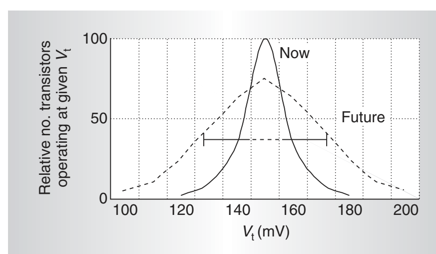

Figure 1.2: The shrinking of technology features means that in the future a smaller percentage of transistors on a given area will function with the desired voltage [9].

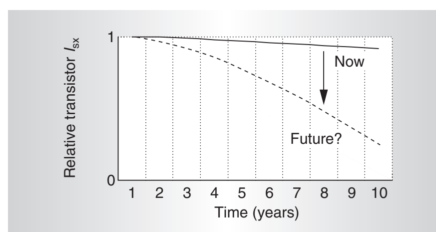

Figure 1.3: Aging Effects are going to become more intense in the future [9].

Both sparing and matching focus on these substitutable resources and are based on the observation that not all of them have to be functional in order for the whole chip to work. Indeed, everyone who has designed a system targeted for FPGA, knows that in the general case not all slices (or CLBs) of the device are utilized. This means that, even if all the non-utilized blocks happened to be defective, there would be no problem with the functionality of the system. Thus, both methods start by testing the Substitutable Resources for permanent defects. They subsequently differentiate themselves by dealing with these defects in different ways: With *sparing*, every defective resource is marked as unusable, while on the contrary, with *matching*, every defective resource is also characterized (Figure 1.4). Characterizing means finding out the subset of functions that the defective component can still perform, despite its defect (see Figure 1.5), in order to still be able to use it [36].

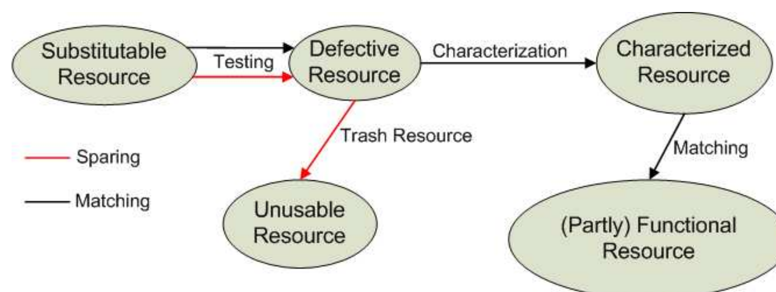

Figure 1.4: Concepts of sparing and matching.

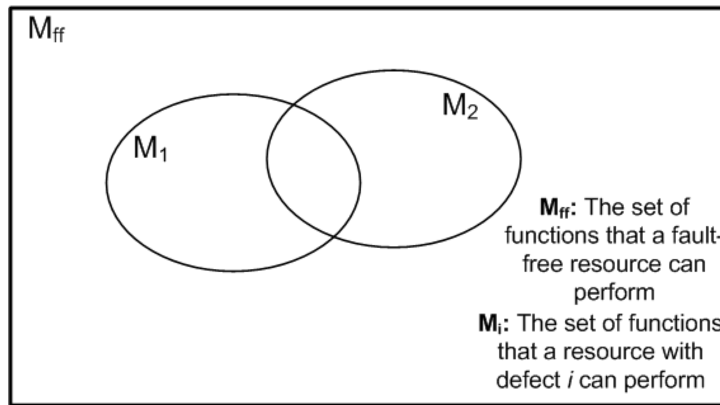

Figure 1.5: Concept of Substitutable Resource characterization.

It is important to note the difference between the requirements of testing a component in general and of testing it for characterization: In the former case, it is enough to produce a go/no go verdict for the whole component, while in the latter we have to find out what exactly is wrong with it, in order to perform the characterization. The process of finding the particular fault that caused a component to malfunction is called fault diagnosis and is naturally more complex than that of go/no go testing. Thus, the tradeoff that exists between sparing and matching is clear: Sparing is simpler to implement but matching wastes less potential functionality of the whole device. It should also be noted that diagnosis can take place either after testing has revealed a component to be faulty, or in parallel with testing.

1.2 Problem Statement

As it became clear in the previous section, FPGAs are ideal for sparing- and matching-based techniques. In this section, we will define the particular problem targeted by this thesis. Through the study of existing literature, we obtained quite a complete picture of various families of approaches for testing FPGAs, either partly or as a whole, either with or without Fault Tolerance being considered. We concluded that the dominant factor in FPGA testing efficiency is that of testing phases [14], meaning the number of reconfigurations required to fulfill the testing goal, whether this is a go/no go decision or any degree of fault localization or diagnosis. We came across a significant number of works that didn't stop at testing whether the FPGA is fault-free or not, but went on to locate the basic block in which the fault occurs. The vast majority of these works, though, considered this level sufficient and did not go on to characterize the faulty resource. This observation naturally triggered our interest about how easy and how useful would characterizing be, especially knowing that because of established technology trends, defect density is going to increase and even small substitutable resources will have a bigger probability of being faulty.

Another factor that affected the direction of our research, was the definition of the

FaTES (Fault Tolerant Embedded Systems) research project by the Computer Engineering group of TU Delft. This thesis came up as an idea during the same meetings from which the FaTES project originated. Being still in its infancy, FaTES actually aims for an integrated SoC design methodology that engulfs implementation of Fault Tolerance strategies at various levels of the system. It features a number of different Fault Tolerance scenarios, in which the notion of Substitutable Resources plays an important role and it is crucial to make as efficient use of these resources as possible. One of our goals for this thesis is to fit the framework defined by that project.

For the above reasons, we aimed for characterization of FPGA basic blocks. To be more precise, the exact formulation of the problem is as follows: Assuming that we have an FPGA slice that we know through previous test results to be faulty, or to have a high probability of being faulty, in what degree can we characterize it and which set of configurations can it support despite its fault? We also opted to limit ourselves to only one or two testing phases (that means none or only one reconfiguration). The relevance of the aforementioned problem will be even more apparent after we present the pre-existing related work.

1.3 Thesis Goals

In this section we will define in more detail the goals with which we started this thesis, in accordance with the problem we defined in Section 1.2. These goals are the following:

- To develop, implement and realize two different methods for conducting fault diagnosis on an FPGA slice. The difference of the two methods lies in the core of the slice, the 2 LUTs, which can be configured, among others, as function generators or as shift registers. We aim to use each of these configurations in one of the two methods and observe the advantages and disadvantages that come with each.
- To decouple the problem of fault diagnosis from the previous stages of testing, as opposed to the vast majority of works that treat testing, localization and diagnosis as one problem.
- To keep the number of testing phases very low, namely two phases for the function generator based method and one phase for the shift register based method. In these 1 to 2 phases we want to achieve as high diagnostic resolution as possible.
- According to the result of the diagnosis, we aim to characterize the slice and decide which set of functions it can still perform, in such a way that it can be reused with specific configurations. The characterization should prepare the ground for the development of an efficient matching algorithm in the near future.
- To apply the methods on a real, physical slice, using fault injection. This is the best possible verification for our methods.
- To use the lessons learned from the whole process in order to propose a set of guidelines for a more testable and diagnosis-friendly implementation of the basic block of the FaTES SoC.

1.4 Thesis Overview

In this section we present a brief outline of the rest of the thesis chapters.

In Chapter 2 we summarize existing research on topics closely related to this thesis. Any themes that were merely mentioned in this introduction to help us define the problem at hand, will also be adequately developed in that chapter. This presentation will solidify the relevance and significance of the problem we solve.

In Chapter 3 we explain the two methods that we developed in order to achieve the thesis goals. We will show, regardless of implementation details, that our approaches can solve the problem that we target.

In Chapter 4, we will focus on the hardware design, analyzing the process of transforming conceptual ideas to real working circuits. We will present the organization of our system from an engineering point of view and explain how various components work together to produce the correct results.

In Chapter 5 we will describe the experimental setup, which was based on a working prototype of the proposed methods, present our experimental results and compare the two methods using both quantitative and qualitative criteria.

Finally, in Chapter 6, we will summarize the ideas presented in the previous chapters, list the thesis contributions, state our conclusions from the whole study and share with the community our ideas for future work.

2 Background

In Chapter 1 we stated the problem that this thesis targets, along with the incentive that led us to research this topic. As it is easily understandable, there are a lot of topics relevant to the work at hand: Fault diagnosis, FPGA-specific testing methods and fault tolerance are a few of them. The purpose of this chapter is twofold: First, to define some basic notions that are required to read the rest of this document and second, to summarize existing research, the results of which we used for our work.

The presentation of all necessary background information and related work in this chapter is organized as follows: In Section 2.1 we mention a few important facts about fault models and testing and go on to present some existing research particularly targeted for FPGAs. In Section 2.2 we begin by expanding a little on the notion of fault tolerance as it was introduced in Chapter 1 and go on to present some relevant and useful facts about the techniques of sparing and (mainly) matching. Before the end of Section 2.2, we also briefly refer to the FaTES paradigm, a newly-defined project relevant to this thesis. Finally, in Section 2.3, we summarize the whole chapter and also briefly go back to the problem statement and justify its significance and relevance.

2.1 The Testing Procedure

The necessary basis for all fault tolerant systems is being able to notice the occurrence of faults, whether they are temporary or permanent. Only then can the fault subsequently be contained and dealt with. In this section, we will give an overview of the various stages of this procedure and clarify the position of our solution in that framework. We will also elaborate a bit more on fault diagnosis, which is the main objective of this thesis.

2.1.1 Fault Models

The basic term used for every malfunction of a digital system, like stopping to work altogether, working outside of its specifications (i.e., working slower), or producing wrong results is *system failure.* A system failure is caused by the occurrence of an error, meaning a misbehavior of a part of the system, i.e. a counter not counting up when it is supposed to. The root causes of such errors are, as long as this thesis is concerned, physical defects in the underlying structure of the system. The exact nature of these defects cannot be easily determined without observing the digital system in depth, i.e. with a microscope [32].

In order for these defects to be detected and comprehended efficiently, they have to first be abstracted by more easily approachable descriptions. *Fault Models* are lists of possible misbehaviors (called faults) of a digital system that occur as the result of the

presence of a defect [30]. The quality of a fault model is determined by how completely it covers the possible defects that can occur in the system. Fault models are also differentiated by the level of abstraction they introduce in the description of the misbehavior. In this section we will mention a few categories of fault models and elaborate on two of them with examples relevant to this thesis.

The general categories of fault models according to the abstraction level are [30]:

- **Electrical Faults:** This is the kind of faults that is closer to the level of actual defects. They are also known as technology-dependent faults. An example electrical fault is a transistor stuck-open fault, meaning that a transistor always behaves as an open switch.
- **Functional Faults:** This kind of faults is defined in the Register-Transfer Level (RTL). The best example of a functional fault is the stuck-at fault for wires (see next section).
- **Behavioral Faults:** These faults do not take into account any internal information of the system or component under test, but just the expected behavior.

2.1.1.1 The single stuck-at fault

A very popular RTL fault model is the single stuck-at fault [30]. It focuses on wires of a combinational circuit diagram. It models underlying defects as wires of the circuit that always have the value '0' or the value '1', hence we have the faults "signal" s.a. 0 and "signal" s.a.1. It is not a very abstract description, in the sense that it closely resembles the wire defects of short and open.

An example application of the single stuck-at fault model is illustrated in Figure 2.1. This simple circuit has 10 possible faults under the specific fault model, listed in the first column of Table 2.1.

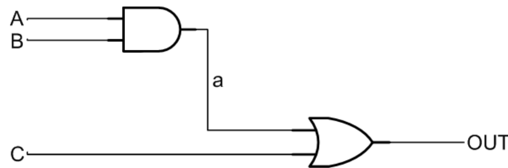

Figure 2.1: Example circuit for Section 2.1.

2.1.1.2 Functional Fault Models for Memory

The regularity of memories allows us to use functional fault models to check each memory cell for a number of possible malfunctions. In this context, a number of single-cell memory faults are described as follows:

Each fault is described through a fault primitive, which is actually a tuple of 3 elements: $< S/F/R >$ [7]. An example fault primitive is $< 0w1/0/- >$. The meaning of each of the tuple members is explained below:

Table 2.1: Test Vectors and Faulty Responses for the example circuit.

Test Vector	000	110	001	100
Fault	Faulty Responses			
fault-free	0	1	1	0
f0: A s.a. 0	0	0	1	0
f1: A s.a. 1	0	1	1	0
f2: B s.a. 0	0	0	1	0
f3: B s.a. 1	0	1	1	1
f4: C s.a. 0	0	1	0	0
f5: C s.a. 1	1	1	1	1
f6: a s.a. 0	0	0	1	0
f7: a s.a. 1	1	1	1	1
f8: OUT s.a. 0	0	0	0	0
f9: OUT s.a. 1	1	1	1	1

- S is the sensitizing sequence of the fault. In our example, $0w1$ means we should write an '1' in a cell that now contains a '0' in order to sensitize (trigger the appearence of) the specific fault.
- F is the fault effect. In our example, 0 means that, even after the write operation, the faulty memory cell still contains a '0'.
- R is a (possible) incorrect read value. In our example, the erroneous '0' in the memory cell is read correctly, thus R = '-'. The R field is meaningful in faults that describe a malfunction of the read operation.

According to the above analysis, our example fault $< 0w1/0/- >$ represents a memory cell that cannot transition from 0 to 1. The state diagram of this fault is shown in comparison to the state diagram of a fault-free cell in Figure 2.2. More single cell faults are defined and categorized in the same way as our example fault, by Al Ars, van de Goor et al in [7] and [40]. The fault list that we used in this thesis is shown in Table 2.2.

In addition to the single-cell memory faults, there is also the category of 2-cell faults, modeling the possible malfunctions of a memory cell because of interference with a neighboring cell. Not having been applied on FPGA testing according to existing literature, 2-cell faults are out of the scope of this thesis.

The most popular family of tests for detecting memory faults is that of march tests [32]. March tests consist of elements that correspond to one or more memory operations to be conducted on all memory cells under test. An elementary march test is the following:

$$\{\uparrow w0; \uparrow r0; \uparrow w1; \uparrow r1; \}$$

This is a march test of 4 elements. The first element (w0) dictates that the value 0 should be written in all cells and the second (r0) that the value 0 that was just written has to be read from each cell. Subsequently, the value 1 is also written to and read from all cells.

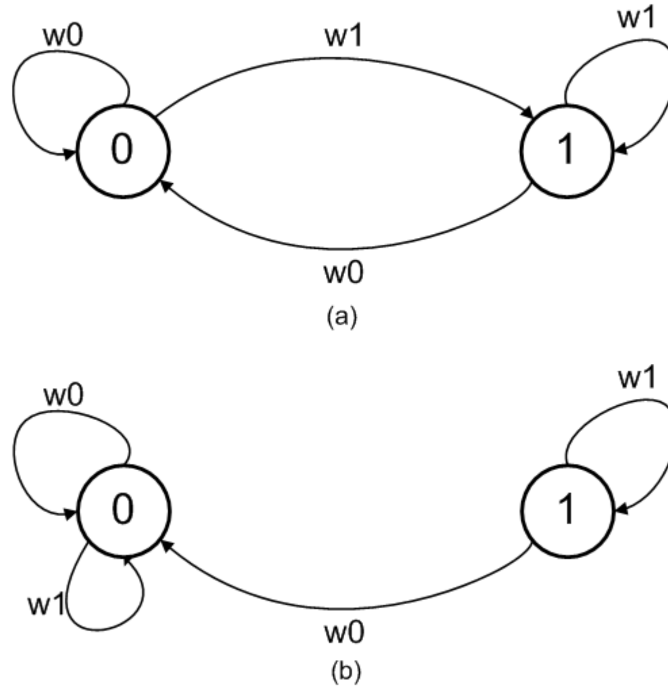

Figure 2.2: State diagrams for fault free and faulty memory cell.

Table 2.2: Memory faults considered in this thesis.

1	$< 0/1/- >$	State Fault 1
2	$< 1/0/- >$	State Fault 0
3	$< 0w0/1/- >$	Write Disturb Fault 1
4	$< 1w1/0/- >$	Write Disturb Fault 0
5	$< 0w1/0/- >$	Transition Fault $0 \rightarrow 1$
6	$< 1w0/1/- >$	Transition Fault $1 \rightarrow 0$
7	$< 0r0/0/1 >$	Incorrect Read Fault 1
8	$< 1r1/1/0 >$	Incorrect Read Fault 0
9	$< 0r0/1/1 >$	Read Destructive Fault 1
10	$< 1r1/0/0 >$	Read Destructive Fault 0

2.1.2 Testing, Localization, Diagnosis and Repair

The general definition of Digital Circuit Testing is the process through which we can decide whether or not the device under test was fabricated without problems (called defects) and can provide the expected function. It is not to be confused with design verification, which happens before fabrication and ensures that the device was *designed* correctly in order to perform the said function (see Figure 2.3).

Taking the notion of testing a step further, we observe that, depending on the ap-

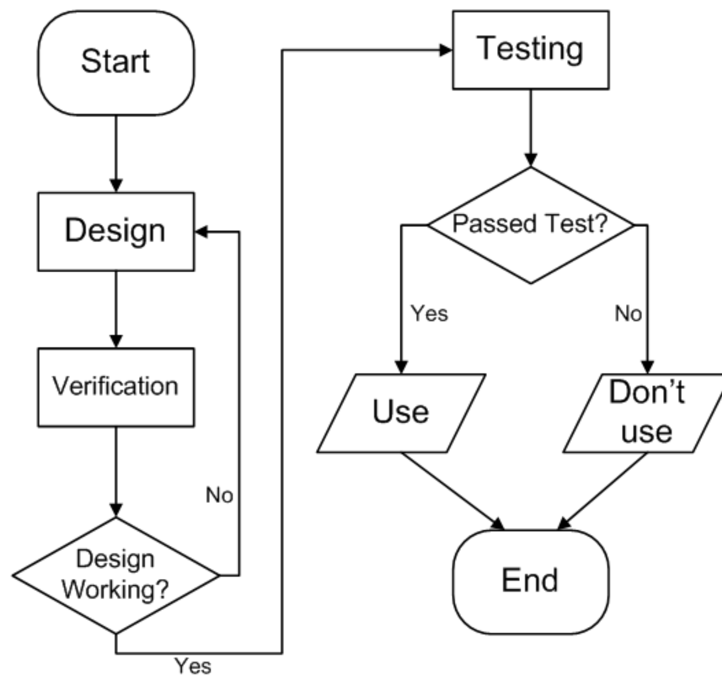

Figure 2.3: Verification and Testing.

plication, the specifics of testing (the method used and the results expected) vary in ways that will be shortly explained. A chain of four steps exists, after each of which the procedure can stop if the application expectations are met. These steps are Testing, Localization, Diagnosis and Repair. The term "testing" is (usually) used to describe the first step of this procedure, but also (sometimes) to describe the whole procedure, or at least the first three steps. In this report we try to make use of the former. In this section we define these four phases and focus a little more on diagnosis, which is the specific interest of this thesis.

2.1.2.1 Testing

Testing is the most straightforward of the four steps: Its sole purpose is to make a go/no go decision about a system or subsystem, called *Circuit Under Test (CUT)*. This is actually sufficient for the vast majority of applications today: When we talk about testing in general, we usually mean checking a chip directly after fabrication, to decide if it is to be sold or thrown away. In order for a correct decision to be achieved, the actual defects are abstracted by faults, as has already been described. Testing methods are out of scope for this thesis. More about them can be found in [32] and [30].

One important for this thesis notion related to testing, is that of on-line testing [46]. Testing a CUT on-line, means testing it without having to stop the normal operation of the system to which it belongs. Apparently, this is very important in the case of the CUT being just a system component (i.e., an FPGA slice) instead of the whole chip.

The term on-line also applies to the next three steps (localization, diagnosis and repair), if the application requires them.

2.1.2.2 Localization

Testing, as described above, is sufficient only for systems without any Fault Tolerance requirements. When dealing with fault tolerant systems, it is required that faults can be dealt with in such a way that the system will continue to function, as defined by its specifications. Fault Localization is the first step towards assuring this: Localizing the fault means determining the component of the system in which the fault in question resides. Examples of fault localization are determining the faulty processor in a failing multiprocessor system [10], discovering the fault site in redundant memories [18] and locating the faulty basic block in a failing FPGA [41].

It is obvious that the term "Localization" is tightly bound with the notion of granularity. For example, "faulty component" in the context of a system realized on multiple FPGAs, could mean the faulty FPGA, the faulty CLB, the faulty slice, the faulty LUT or even finer-grain components. Fault localization is also of outmost importance in all sparing-based Fault Tolerance strategies. As it has been already explained, determining the faulty component is sufficient if the strategy involves throwing away the substitutable resource and using a spare resource in its place. If, however, we want a less conservative approach, fault diagnosis is in order.

2.1.2.3 Diagnosis

Fault Diagnosis involves spotting the exact cause of a failure of the system to accomplish the specified function. This can mean either the exact fault that triggered the erroneous response, or even the actual physical defect that caused this fault. The original motivation behind fault diagnosis was finding the root-cause of chip failures in order to improve the fabrication procedure and enhance the yield. In this case, the actual physical defect should be diagnosed, which even involved observation of the chip with a microscope. Such deep diagnosis is, obviously, out of the scope of this thesis.

In the Fault Tolerance context, diagnosis focuses on detecting the exact fault that caused the CUT to fail. In accordance with localization being a requirement for sparing-based Fault Tolerance strategies, diagnosis is necessary for the characterization of a substitutable resource, which makes matching-based strategies possible. After the specific fault that caused a substitutable resource to fail becomes known, the functions that this component cannot perform anymore can be determined, which is the definition of characterization. After that is accomplished, a suitable function can be mapped on the component.

It is natural that the terms "localization" and "diagnosis" are interchangeable in existing literature. One reason for this is that the actual meaning of both depends heavily on the granularity, as already has been established for localization. Also for diagnosis, its meaning is different when we talk about spotting the physical defect and determining the fault caused by that defect. Even when focusing on the fault, the abstraction level of the fault model affects the meaning of "diagnosis". As a result, either of the two terms can be used in place of both, combined with the term "*diagnostic resolution*" [35].

As fault diagnosis is the object of this thesis, we will now categorize some general diagnosis methods and define some important relevant notions, as they are presented in books [32] and [30].

The most important, for this thesis, definitions on fault diagnosis are listed below:

- **Diagnostic Accuracy** is defined as the proportion of all CUTs that are diagnosed accurately.
- **Diagnostic Resolution** is defined as the average number of faults that are identified by the diagnosis process as possible causes of the CUT failure. By this definition, the minimum and ideal value for diagnostic resolution is 1, meaning that the diagnostic method reduced the set of possible causes of failure to only one fault of the fault list.
- **Fault-free and faulty response:** The fault-free response is the response of a version of the CUT without any faults, if all the test vectors are applied to it. For the example circuit of Figure 2.1 and for the sequence of test vectors listed in Table 2.1, the fault-free response is "0110". Also, for every fault (f_i) in the fault list, a faulty response is defined as the response of a version of the CUT with f_i in it. For example, for the same circuit and test vectors, the faulty response of the fault "a s.a. 1" is "1111".
- **Error response:** Based on the faulty response for every fault f_i in the fault list, the error response is defined to be equal to '1' when the faulty response for f_i is different than the fault-free response and equal to '0' otherwise. For example, the error response of the fault "a s.a. 1" is "1001". See also Table 2.3 for the difference between faulty and error response.
- **Fault Dictionary:** A fault dictionary is a list with one entry for each fault in the fault list. Depending on the size of the fault list, the size of the responses and the available memory, the format of this entry varies, defining the following kinds of fault dictionaries:
  - Complete fault dictionary: The entry corresponding to each fault is the fault's error response (or equivalently, its faulty response).
  - Reduced fault dictionaries: The entry corresponding to each fault is more compact than the corresponding entry of a complete dictionary. A good example of a reduced dictionary is the pass-fail dictionary, which lists the vectors that cause failures for each fault, instead of the complete faulty response on all outputs of the CUT for each vector.
- **Diagnostic Tree:** The diagnostic tree is a structure that keeps the decisions needed to be made by the diagnostic method after the appliance of every test vector. When the CUT fails a vector, the corresponding branch contains the set of faults that are suspects for having caused the particular failure. The leaves of the tree correspond to final results of the diagnosis. They enclose the same information with the respective entry of the fault dictionary, but in general the result is produced faster.

Fault Diagnosis methods are categorized as follows [32]:

Table 2.3: Difference between faulty response and error response. The error response is '1' whenever the faulty response is different than the fault-free response.

Fault-free response	0110
Faulty response	1111
Error response	1001

- **Cause-effect diagnosis:** This strategy for fault diagnosis relies on the CUT being simulated once with the presence of each fault of the fault model. That means that diagnosis starts by predicting the faulty behavior (effect) of the CUT for each one of the faults that is present in the chosen fault model (cause). This is the strategy that we choose for our diagnosis method, because of the small size of our CUT, which meant that running all the fault simulations was manageable. Cause-effect diagnosis is further categorized as follows:
  - **Post-test diagnostic fault simulation:** This strategy starts by applying all the test vectors to the CUT, comparing the response to that of a fault-free CUT and locating the bits of the response that are erroneous. Subsequently, fault simulation takes place for every fault for the first test vector only. The faults that do not fail or fail at different bits than the CUT, are dropped from the fault list. This procedure continues until the required diagnostic resolution is achieved.
  - **Fault dictionary based diagnosis:** This strategy starts by simulating the CUT for every fault in the fault list and storing the faulty response, along with the fault-free response, in a fault dictionary, as it was defined before. For every CUT, the test vectors are then applied and the responses gathered and compared with the entries of the fault dictionary, in order to make a decision about which fault (if any) is present in the CUT.
  - **Diagnostic tree based diagnosis:** Simulations for faulty versions of the CUT happen before the method is applied, just like in dictionary based diagnosis. Afterwards, the test vectors are applied to the CUT, but when the CUT produces a faulty response, the set of possible faults is reduced to the ones that could cause the particular failure. The procedure continues, possibly even adapting the next test vectors to the particular set of suspected faults, until the desired diagnostic resolution is achieved. We chose to use the strategy of a diagnostic tree, because it was relatively simple to perform all fault simulations in design-time and because it generally terminates sooner than the dictionary strategy and produces the same results.

- **Effect-cause diagnosis:** This fault diagnosis strategy is more suitable for fault models that contain a bigger number of faults. Diagnosis starts from the erroneous response of the CUT (the effect) and, according to that, the faults of the fault model (possible causes) that couldn't be present in the CUT are eliminated. This procedure is continued until the required diagnostic resolution is achieved.

2.1.2.4 Repair

The final step of every Fault Tolerance strategy is actually dealing with the fault at hand. Of course, the actual defect cannot be fixed (or in the best case fixing it is very expensive), since the technology features are very small to interfere with, but it can be worked around in ways that we have already mentioned. For example, in the case of a sparing-based Fault Tolerance strategy, repair happens after fault localization by throwing away the faulty resource and mapping the functionality that it used to perform on a spare resource. On the contrary, in the case of a matching-based strategy, repair happens after diagnosis by mapping a suitable function on the faulty resource. In any case, after the stage of repair, the system has to keep functioning under the given specifications.

2.1.3 FPGA Testing

In this section, we will summarize existing solutions for testing FPGAs. We will focus on methods relevant to the work done in this thesis. In general, FPGA testing methods tend to cover more than one stage of the testing procedure, as it has already been described. Fault localization in particular is achievable by the majority of existing works, making use of the FPGA regular structure.

Since before 2000, researchers treat the problem of FPGA testing as different than chip testing in general, because of the FPGA special structure. Some basic characteristics of FPGAs that affect testing are summarized below [14]:

- FPGAs are regular structures: The fact that FPGAs consist of repeating the same functional unit, hints researchers to separate the problems of:
  - Testing one such basic unit.
  - Finding a scheme to feed the test vectors to the maximum possible number of basic units and analyze the results in the fastest and/or cheapest possible way.
- FPGAs are heterogeneous structures: For this reason researchers treat testing of different parts of the FPGA as different problems. The basic parts that various methods target independently are the following:
  - The basic logic block.
  - The interconnection resources.
  - The special parts, such as I/O terminals and programming circuitry.
- FPGAs are reconfigurable devices. Research on FPGA testing cannot ignore the impact of every specific configuration on the testing process. In particular:
  - Every specific configuration limits the set of faults that can be tested, by making some faults redundant, meaning that they do not affect the output of the circuit and thus, they cannot be tested. The redundant faults cannot be ignored, though, because under another configuration they will not be redundant.

– On the other hand, testing engineers can shape the configuration in such a way that it simplifies the testing problem. The smart choice of testing configuration is, actually, the core of all proposed methods.

This thesis focuses on testing and diagnosing the FPGA basic blocks. For this reason, testing of interconnection resources and of special circuitry is out of scope. Some good techniques for testing the interconnection resources are presented by Lombardi, Huang et al in [44] and [29].

Returning to the subproblem of testing the FPGA logic resources, there is a great number of publications before and until 2003, on testing SRAM-based FPGAs. All of these methods are summarized in this 2003 survey by Doumar and Ito [14]. Various methods are enumerated, most of which share the following common characteristics:

- Every approach starts with a basic method of testing a single logic block. This is based on configuring the LUT appropriately and controlling the data flow by configuring the multiplexers appropriately. These characteristics change from testing configuration to testing configuration, in order to check different sets of faults and cover the entire fault model. Every testing configuration defines a *testing phase* [24], with the number of phases being the decisive factor for the time needed to test the whole logic block, since reconfiguration is needed to go from one phase to the next.

- During each of these phases, LUTs are configured as function generators, with the XOR and XNOT functions. These functions allow checking of the memory cell faults as well as address line faults [24].

- Every approach uses a clever scheme in order to use the basic method on as many basic blocks as possible at the same time. This defines a number of *testing sessions* needed to test the whole FPGA. It is clear that the number of sessions is a decisive factor for the time needed to test the whole FPGA [24] [41].

- Most approaches started as mere fault detection methods, aiming for a go/no go decision for the whole FPGA, but were later appended with new publications and upgraded to diagnosis methods. Diagnosis in this context means spotting the faulty basic block, having in mind yield improvement by introducing redundant FPGAs in the same manner as redundant memories. As an example, [15] is a diagnosis version of [13]. In the context of this thesis, this level of diagnostic resolution corresponds to fault localization, since it can support sparing-based fault tolerance strategies.

The aforementioned methods have also important differences, resulting to the following categorizations:

- In some methods the tests are applied from outside the chip, while on others testing is conducted in a Built-In Self-Test (BIST) manner. The advantage of BIST in FPGAs is that it does not have any area overhead, since part of the FPGA can be configured as the tester, while the rest is the unit under test. The parts can be

interchanged between different testing sessions. A good example of this approach for fault detection and localization is presented by Wang and Tsai in [41]. The basic principle of BIST is illustrated in Figure 2.4.

- Some methods choose to be restricted by the existing FPGA structure, while others propose changes that can be made to this structure in order to make it more testing-friendly (Design for Testability, DFT). An example DFT method by Liu et al is presented in [28]. The authors use two extra transistors between each pair of successive basic blocks and one extra primary pin, to test whole rows and columns of the FPGA. In [14], the authors observe that this technique can be used also for fault localization, if the row and column results are combined.

- Fault models differ from approach to approach and from component to component. For example, at the time of the publications, internal implementation details of multiplexers inside the basic blocks were not known to the researchers, thus most of them adopted a functional fault model for them. For this reason, hybrid fault models are a very popular choice, usually combining the stuck-at fault model for wires with functional fault models for the rest of the components. A good example of using a hybrid fault model is [24] by Huang et al.

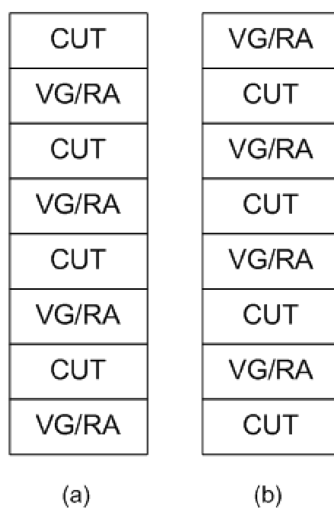

Figure 2.4: The BIST principle for FPGA testing: During the first testing session some blocks are tested (CUTs), while others accommodate Vector Generators and Response Analyzers (VGs and RAs). During the second session, the roles of the blocks are interchanged.

An important observation made by the authors of the 2003 survey [14], is that researchers tend to factitiously reduce the number of testing phases required, by reducing the circuitry that is tested, making the comparison of different methods difficult. They specifically claim that methods which really test the basic block exhaustively require at least 8 and up to 21 test phases, while methods that claim to reduce the number of phases to 6 or even 4, do not really check the functionality of the block. In this thesis,

we consciously reduce the functionality to be checked, since we are performing diagnosis and can afford to sometimes mark the CUT unusable when diagnosis fails to spot the exact fault. Thus, even with a simplified list of faults, our methods are applicable and useful.

One particular work of the design for testability family proposes that the testing configurations are shifted inside the FPGA, in order to bridge the different testing phases. In order to achieve that, the configuration memory should be modified in such a way that configuration data can be freely shifted through it. Note that, although not explicitly mentioned, this doesn't only include LUT bits, but also multiplexer configuration bits and other parts of the configuration. The method is expanded for testing also routing resources [13], and for spotting the faulty basic block [15].

FPGA testing research naturally went on also after 2003. Researchers tried to make use of modern partial reconfiguration capabilities of the devices, in order to make the procedure more efficient. Niamat et al in [31] and Sundararajan et al in [39] both made use of the Xilinx JBits toolkit for partial reconfiguration, to provide defect tolerance by avoiding at run-time the blocks that were discovered to be defective at test-time.

Furthermore, Dutton and Stroud in [17], used the lessons of more than 10 years of experience on the research field to completely test a Xilinx Virtex-5 FPGA in 17 configuration phases and also locate the fault on the LUT or flip-flop that it occurs. It is interesting to note that, even with fewer phases than other relevant works, the configuration time remains the dominant factor of the total testing time.

One of the most interesting works on FPGA testing, localization and diagnosis was done by Abramovici, Stroud, Emmert et al in the framework of the Roving STARs paradigm [4]. We choose to present more about this in Section 2.2.1.

From our literature review on FPGA testing, we drew the following useful conclusions for designing our methods:

- The number of testing phases, due to reconfiguration time, is still dominant in deriving the total testing time, even with partial reconfiguration in play. We decided to diagnose as many faults as possible with only 1 or 2 configurations.
- Built-in self-test approaches for FPGAs have reduced cost, since the area used for the Test Pattern Generator and the Response Analyzer can be used for normal system functionality once the testing/diagnosis is complete. We chose, for this reason, to accommodate our tester on-chip.
- Results of Design for Testability methods cannot be directly compared with results of methods that are limited by real device features, since they have no common frame of reference. In our opinion, DFT should come as a result of the difficulties we face when we design a method for real devices and this is what we chose to do in this thesis: We develop methods restricted by the real device characteristics and record our observations as DFT proposals.
- LUTs of the same slice should be configured with complementary contents during the testing configurations, in order to set up testing of the rest of the slice, mainly multiplexers. Also, realizing functions as XOR and XNOT on the LUTs helps the detection of address faults. We adopted the first of these rules and used the second as a guideline for building our own LUT configuration for testing.

2.2 Fault Tolerance

Fault Tolerance is, as it has already been established in Chapter 1, a cornerstone for standing up to the challenges of many Embedded Systems applications. Examples of such applications are:

- Space applications.
- Avionics.
- Biomedical applications, i.e. medical implants.
- Safety systems in transportation and racing.
- Nuclear plant control systems.

In general, fault tolerance refers to containment of both temporary and permanent faults. It is the shrinking of IC features combined with the increasing popularity of regular realization platforms that triggered the spreading of the term “defect tolerance”, used explicitly for permanent faults, like in [36]. In this thesis we deal only with permanent faults, thus both “fault tolerance” and “defect tolerance” will be used with the same meaning. Some solutions on dealing with temporary faults on FPGAs can be found in [27], [34], [21] and [45].

In this section, we will elaborate on the notions of sparing and matching through some example publications and also briefly refer to the newly defined FaTES paradigm.

2.2.1 Sparing and Matching

The majority of the FPGA testing works presented in section 2.1.3 set the stage for the development of system-level sparing-based fault tolerance techniques. A simple example of a strategy to make use of the fault localization capabilities of these methods, is presented in [26], where groups of four substitutable resources form one tile. Any function requiring 3 of these resources can be mapped on this tile and use the fourth basic block as a spare. The technique is illustrated in Figure 2.5. Note that the four blocks that form a tile can very well be the four slices that form a CLB in many FPGAs, like Virtex-II.

Many sparing implementation strategies revolve around shifting the placement of substitutable resources altogether, in order to avoid the defective part. Different methods are proposed for that, varying in respect with the spread of spare resources on the resource grid and the choice of the shifting direction. For example, in [11], Hanchek and Dutt propose two different distributions of spares, called king and horse distribution, which allow the configuration to be shifted in 8 or 4 different directions respectively, in order to avoid the faulty resource.

Despite the impressive range of existing sparing techniques, few researchers actually move to the next step of reusing defective FPGA resources. *Outside the FPGA domain*, a good example of a coarse-grain matching-based technique is presented by Kim et al in [25]. They perform low-resolution diagnosis on programmable processors and use them only for suitable applications. The actual matching problem is treated as a scheduling problem, since any given application can be mapped on a subset of processing units only.

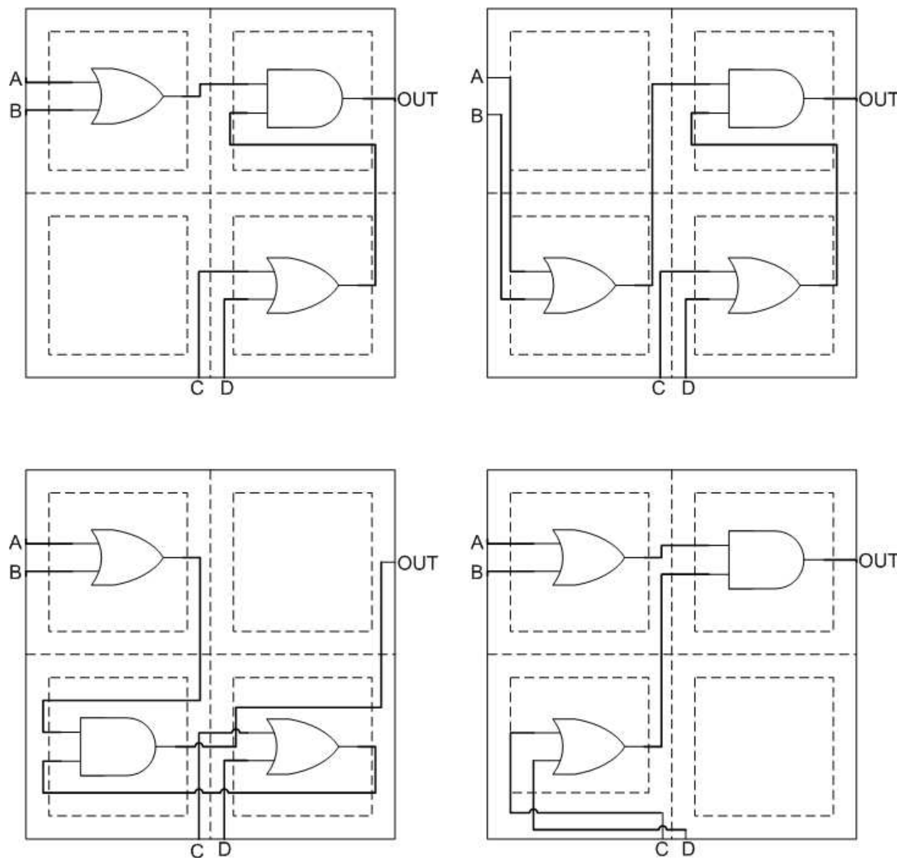

Figure 2.5: A simple sparing strategy. The 4 basic blocks of the tile are used as 3 effective blocks and one spare. If one of the blocks is defective, the function is mapped on the other 3. The 4 possible mappings of a simple function, depending on which of the 4 blocks is defective are depicted [26].

The best example of a work that actually focuses on reusing FPGA faulty resources, is the roving STARs (Self-Testing Areas) paradigm by Abramovici, Stroud, Emmert et al [4]. The basic idea of their approach is to have a small area of a few basic blocks roving around the whole FPGA for testing, diagnosis and fault tolerance purposes. These few logic blocks test each other through 15 different configuration phases and also test the surrounding interconnect resources. In [4] they define this basic idea and present the different testing phases, each of which corresponds also to one mode of operation for the basic block. By observing with which modes of operation the CUT actually failed, they also implicitly perform some characterization, such that the resource can subsequently be used as a spare for a suitable function.

The roving STARs paradigm was further developed through more publications. In [20], the way to deal with detected faults is described by using a spare from the neighborhood of the faulty block. Also, there is the option of the faulty block continuing to perform the originally intended function, if it can still support it. Also, different

distributions of spares are tried. In [6] the testing strategy is enhanced to make testing faster and increase diagnostic resolution. In [5] the interconnection testing approach is presented. In [2], all the above improvements are brought together in an integrated approach, and more effort is put in reusing a faulty resource, both for its original intended function or as a spare for another suitable function. In [3], delay faults are also dealt with. Finally, in [19] the effect of partially reusing faulty blocks is tested through benchmarks and proven to provide a significant improvement.

It has to be noted that the biggest difficulty that matching-based strategies face, is the actual matching phase itself [36]. The set of functions to be mapped and the set of the available substitutable resources (fault-free and faulty) form two distinct sets of nodes, constituting a bipartite graph. Each function is able to be mapped on a number of the available resources. A link is present in the graph between a function and a resource, if the function can be mapped on this resource (Figure 2.6a). In order for the mapping to be completed, a matching of this graph has to be calculated (Figure 2.6b).

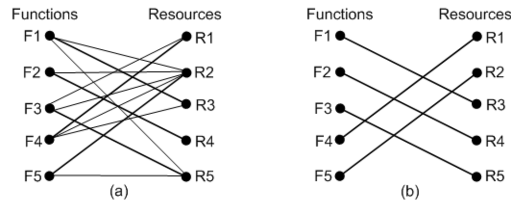

Figure 2.6: The matching problem: (a) is the bipartite graph of functions to be mapped and the available resources. Notice that resource R2 is fault-free. (b) is a successful matching of the graph.

The matching problem can be solved in $O(R^{2.5})$ time [22]. The matching problem is relevant also in other research fields, like network switch scheduling [12], for which it is solved according to the characteristics of each application. A suitable heuristic may be feasible for the case of matching functions to resources, reducing the complexity of the problem.

2.2.2 The FaTES project framework

An important driving force behind this thesis was the definition of the FaTES (Fault Tolerant Embedded Systems) research project. In fact, the thesis topic was defined in the same meetings from which the project definition originated.

The FaTES SoC is designed from scratch with the notion of fault tolerance governing all design decisions. It is assumed that the bigger part of the SoC is fault-prone and a reliable system has to be built by using the unreliable resources appropriately. To achieve that, existing fault tolerance techniques are going to be combined with new ones and integrated in a system that will implement fault tolerance at different levels.

In the above context, many different fault tolerance scenarios are going to be considered. For example, consider the existence of a 32-bit integer adder in the fault-prone part of the SoC. The following are two possible scenarios:

- The adder is discovered to be faulty by software-implemented tests (for example, through duplicated instructions [33]). Subsequently, the operating system continues to apply a pre-defined set of inputs to the adder, in order to determine which part of the adder (i.e., which 4-bit part) is to blame for the fault. This 4-bit part corresponds to a few reconfigurable substitutable resources, one of which is guaranteed to contain a fault.
- The adder is self-checking, based on the idea of each functional cell (essentially, each bit position of the adder) checking its neighboring functional cell [16]. One of the functional cells is discovered by its neighbor to be faulty. This cell corresponds to a few (probably to one) reconfigurable substitutable resource(s), (at least one of) which is guaranteed to be faulty.

The above scenarios are both good examples of the usefulness of the method we develop in this thesis: The local diagnosis method can be applied to the faulty substitutable resource(s), which can be characterized and reused by a different kind of component.

2.3 Summary and Thesis Significance

In this chapter we gave an overview of notions that are required in order to understand the content of this thesis and of research related to our work. A few basic facts about fault models and specifically the functional memory models that we will use were explained. The sequence of testing, localization, diagnosis and repair was overviewed and our method was categorized as a cause-effect, tree-based scheme. Enough research on FPGA testing and fault localization was presented and related to our work

After the discussion of this chapter, the relevance and significance of the problem targeted by this thesis is more apparent. The specific areas in which the present work aims to improve over existing ones are summarized below:

- All FPGA-targeted methods begin from testing and expand up to the point they can. Some of them achieve only testing, many of them achieve fault localization and a few go on to diagnosis and characterization. This has as a result the obligation to cover the entire functionality of the basic logic block, primary and secondary. By decoupling the diagnosis problem by the rest, we are allowed a more relaxed approach: There is no need for our methods to cover 100% of the basic block functionality. It is sufficient to check the core functionality with a minimal cost, since this is enough for accommodating the vast majority of possible functions to be mapped on the resource. Checking the rest of the functionality would induce a cost too big for the benefit it will provide. A direct advantage of this relaxed approach is the reduction of testing phases required.

- Since we focus on one basic logic block, our characterization scheme does not stop at detecting the faulty component within the basic block, i.e. the faulty LUT or flip-flop. Our diagnosis reduces the set of possible faults to only one, or in the worst case, a few faults, allowing us to define modes of degradation that sacrifice as little functionality as possible to partially reuse the faulty block. For example,

we detect specifically the faulty bit of a LUT, which in the FPGA domain is very important for reusing it.

- Since modern fault tolerance strategies have to be applied in runtime, we have to make sure that they burden the system as little as possible. Using this observation, we aimed for diagnosing as many faults as possible in a few cycles only, because this will be very significant in real circumstances, within the framework of a fault tolerance oriented system, like the one defined in the FaTES research project.

- Although we are completely bound by the characteristics of present day devices, we propose a set of guidelines that a fault tolerance oriented system should follow in order to be more testing- and diagnosis-friendly. This is significant because especially design for diagnosis is not considered in devices that are in the market today.

In the following chapter it is time to analyze the core of this work, which is the two different fault diagnosis methods.

3 Method

With all necessary information adequately presented, the stage is set for the description of our diagnosis methods. In order to have a frame of reference, we worked on the basic block of a real device, the Virtex-II Pro FPGA. The methods are targeted for a Virtex-II Pro slice that is known to be faulty, but can also be applied on a small number of slices, at least one of which is faulty. The goal is to reduce the set of possible faults to a few, even only one, except from the case that further disambiguation induces extra costs without actually serving graceful degradation.

It is important to note that most methods described in Chapter 2 that actually perform basic block characterization, treat testing, localization and diagnosis as a single problem. Testing by itself requires a multitude of phases (at least 6 for testing, but 15 in the case that testing and diagnosis are treated as a joint process). Diagnosis is performed by observing which of these phases caused the CUT to fail. Each phase defines a mode of operation, and the result of diagnosis is a list of failing such modes. Example modes of operation are:

- The slice configured as shift register.
- The slice configured as function generator with flip-flop.

These modes will be better explained later.

As opposed to existing works, our method decouples the problem of diagnosis from that of testing and localization. The motivation for that was partly the fact that in modern fault tolerant systems, localization does not necessarily happen through conventional testing, but also through alternative means, like the cases of software-implemented fault tolerance and self-testing modules. The existence of scenarios the result of which is one or a few substitutable resources with at least one fault, means that there is usefulness in our approach.

Before we go on to the detailed description of the 2 methods, we remind the reader that our approach is that of cause-effect, diagnostic tree based diagnosis. Cause-effect means that we performed fault simulations for all possible faults before the actual diagnosis procedure and the error responses are known in run-time. Diagnostic tree based means that, after every step of the process, the set of possible faults is reduced according to the known error responses and the testing strategy adapted in order to yield the best possible performance results. The reasons for which we chose this approach are the following:

- Because the CUT is relatively small, as is the corresponding list of faults, it was manageable to perform all simulations of faulty versions beforehand. We started designing the system with the knowledge of all faulty responses.

- Our method is targeted to be applied on-line, meaning without interrupting the normal operation of the system. For this purpose, it is critical that the result of diagnosis is known as early as possible. This is the reason we preferred the diagnostic tree approach, which allows diagnosis of faults as early as possible.

The diagnosis scheme is illustrated in Figure 3.1.

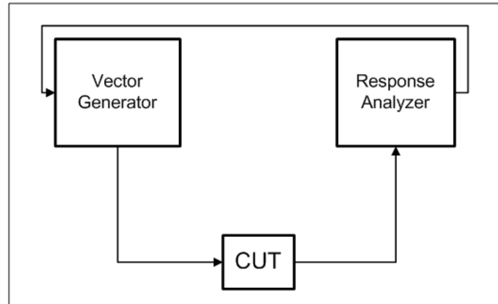

Figure 3.1: Adaptive diagnosis strategy according to the diagnostic tree approach. Based on the CUT response, the response analyzer decides on the next test vectors to be applied.

This chapter is organized as follows: In Section 3.1 the CUT, which is the central object of this thesis, is described in detail. In the subsections, each component of the CUT is described individually and the choice of fault model for each one of them explained and justified. In Section 3.2 the method based on configuring the slice as a function generator is explained, while in Section 3.3 the method based on configuring the slice as a shift register is described. Finally, in Section 3.5 the basic characteristics of both approaches are summarized and a general comparison is performed. The complete comparison, however, will take place in Chapter 5.

3.1 The Virtex-II FPGA Slice

As was already mentioned, the substitutable resource chosen to be diagnosed and characterized in this thesis is the slice of the Xilinx Virtex-II Pro FPGA device. The criteria behind this choice are listed below:

- All devices have similar basic structure and the characteristics for a fault diagnosis method are the same for different devices. Thus, we chose a device that is simpler compared to, i.e. Virtex-5. This allowed us to focus on the methodology concepts, without adding unneeded complexity that wouldn't increase the value of the research.
- It is our opinion that a basic reconfigurable block implemented for testability and easy characterization and matching, should be closer to the smaller implementation of the Virtex-II Pro basic block, compared to other devices mentioned above. Our initial opinion proved to be correct, as will be explained in time. This, of course, refers solely to testability. There is a reason for devices going towards LUTs with more inputs, but this is for purposes of efficient design mapping and not testability.

In this section the basic structure of the Virtex-II slice will first be presented. Subsequently, each component will be analyzed separately and the choice of fault model for it explained. The way that each of these fault models is covered by both the configuration and the input test vectors will be shown and how all of them come together through a bottom-up approach.

3.1.1 Slice simplified model

It is important to mention once more that the purpose of this work is to make fault-prone component degrade more gracefully than the conventional approach of altogether marking the component as unusable when one fault is detected on it. In this context, not every sub-component of the substitutable resource has to be extensively tested. It is, however, important to verify the core functionality of the resource, in order to make a sound claim of reusability. Indeed, anyone who has designed and implemented systems on an FPGA, knows that most slices are used in the "normal" way, meaning that some logic is mapped on their Look-Up Tables and, optionally, the output of this logic is registered in the slice flip-flops.

The simplified version of the slice that was used in this thesis, which is essentially our CUT, is shown in Figure 3.2 [42]. The functionality of each sub-component, from which the realization capabilities of the slice are derived [43], is briefly explained below.

- The two Look-Up Tables (G-LUT and F-LUT) are the heart of every slice. Each of them is basically composed of 16 1-bit entries and 4 address lines that can be used to asynchronously read and synchronously write each of the 16 data. The Look-Up Table is a versatile component that can realize the following functions:
  - Any arbitrarily defined boolean function of four inputs.
  - A shift register of up to 16 bits.
  - A RAM block of 16 1-bit elements, which can be one of the N slices of a RAM block of 16 $N - bit$ elements.
- The MUXF5 and MUXFX multiplexers. The MUXF5 is used to combine the outputs of the 2 LUTs. In this way the slice can realize a boolean function of 5 inputs, the fifth one being the control signal of the multiplexer. The MUXFX, depending on the location of the slice, can be MUXF6, MUXF7 or MUXF8. A MUXF6 combines the outputs of 2 different MUXF5 etc, forming a boolean function of 6, 7 or 8 inputs respectively.
- The two storage elements (G REGISTER and F REGISTER). They can be used as standalone flip-flops receiving and storing any 1-bit data that is needed, or more specifically to register the outputs of the LUTs, terminating a combinational logic path.
- The configuration multiplexers, labeled ROUTING Y, ROUTING X, DYMUX and DXMUX. ROUTING Y and ROUTING X choose which data to propagate to the Y and X outputs respectively:

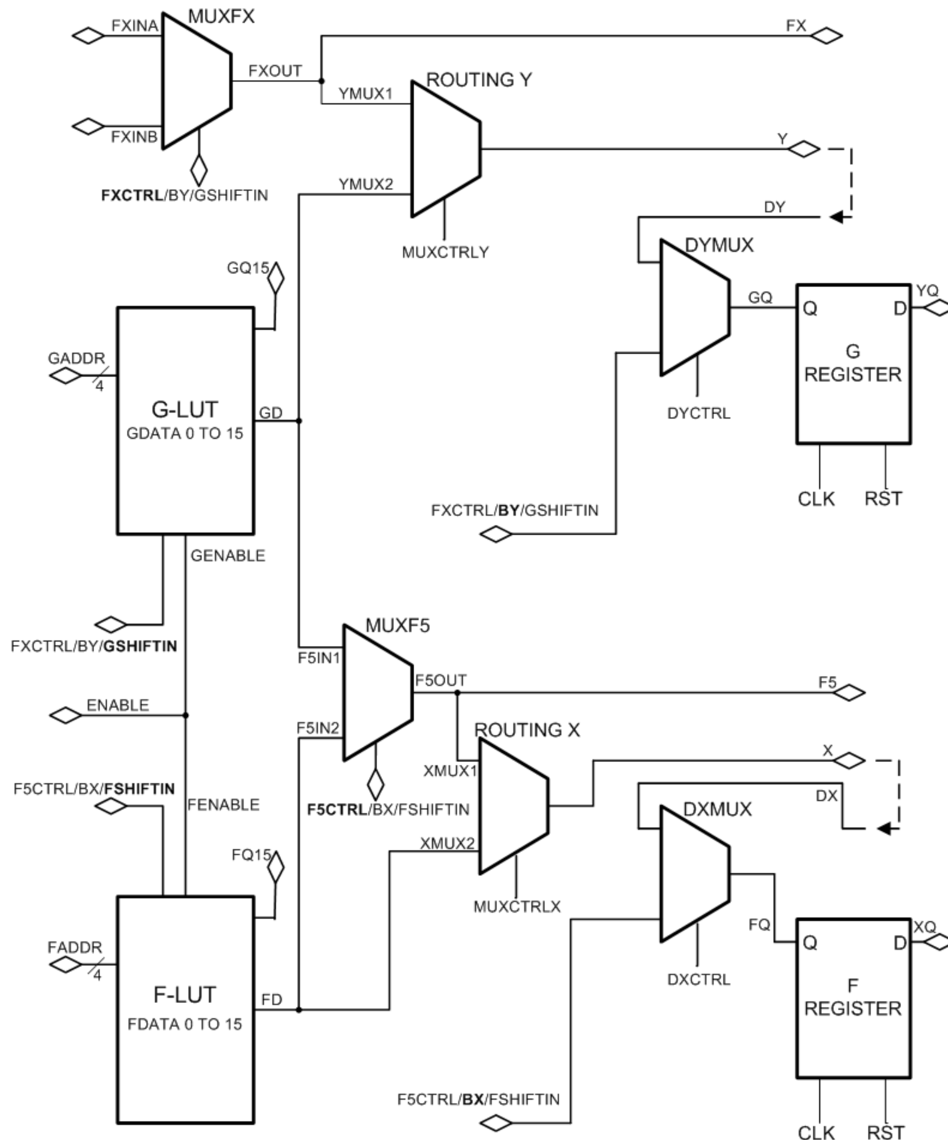

Figure 3.2: The Virtex-II slice simplified diagram.

– The output of the respective LUT, which is the output of a 4-input boolean function, or

– the combination of two or more LUTs, which is essentially the output of a 5-, 6-, 7- or 8-input boolean function.

The DYMUX and DXMUX multiplexers choose which data to input to the Y and X storage elements respectively:

– The output of the respective ROUTING multiplexer, which is essentially the logic product of the combinational part of the slice, or

– a datum directly input to the slice, through the BY and BX inputs.

Apart from the components present in the simplified slice diagram presented above, there are a few special features that we do not test in this thesis. These features include:

- Fast carry chain support.
- Arithmetic dedicated gates.
- A second set of address lines for each LUT, allowing it to realize dual-port memory.

Excluding the out-of-scope features of the slice, the rest will one by one be approached in the following sections, from a diagnosis point of view, with the purpose of choosing a fault model for each of them.

3.1.1.1 The Look-Up Tables

The two Look-Up Tables are the basic components of every slice and generally the basis of the FPGA principle: They can implement any combinational logic of up to 4 inputs and one output. They are composed of 16 memory cells (GDATA 0 to 15 and FDATA 0 to 15), 4 address lines that can address each of the memory cells independently (GADDR 0 to 3 and FADDR 0 to 3), and one output (GD and FD) on which the bit that is read appears. Additionally, when configured as a shift register, each of the LUTs also receives a bit to be shifted in (GSHIFTIN and FSHIFTIN) and a shift enable control signal (GENABLE and FENABLE). Note that the enable signal is actually common, meaning that when both G-LUT and F-LUT are configured as shift registers, they allow (or deny) a shift operation on the same cycles. Also, note that on the GD or FD output still the value of the cell determined by the address lines appears. This creates the option of a serial-in, serial-out shift register of fixed length (by keeping the address locked at a constant value) and of a serial-in shift register which can output any of its bits on any cycle (by changing the value of the address). There is also an output on each shift register (GQ15 and FQ15) on which the contents of DATA15 always appear. This is mainly used to connect more than one 16-bit shift registers in order to form a bigger one.

The 16 LUT entries are tested using the memory fault models listed in Table 2.2. As the entries are essentially SRAM cells, we have chosen this functional fault model, in order to check whether or not they can perform the basic operations that they are supposed to (read and write) without problems. In this way, we don't have to know more about the realization details of every cell. Note that one of the basic differences of the two proposed methods is the way in which they deal with the memory faults, as well as the ability to cover all 10 of them. Further details about this are presented in the method description and evaluation.

There are two reasons why testing of memory cells is the determining factor for the number of clock cycles needed to check the whole slice:

- There are 16 memory cells in each LUT and on every clock cycle we can only read one of them.

- In order to cover the faults listed in Table 2.2, we need to at least read both possible values from each cell.

Thus, at least 32 cycles are needed, plus the time required to switch between the two values, which varies between the two proposed methods. For this reason, the challenge of both methods was to be able to complete all other necessary tests in the time frame imposed by the memory cells test.

The addressing of the memory is checked by treating the 4 address lines as wires and checking each of them for the s.a. 0 and the s.a. 1 fault. It is important to note that these faults are checked by inspecting the read output and comparing it with the contents of the memory cell that was supposedly read during that clock cycle. In that sense, the resources used to diagnose the address line faults and the memory cell faults are the same. The two methods deal with this limitation in different ways. The starting point of both of them is the effect that each address line fault has on the behavior of the read operation. These effects are summarized in Table 3.1. We assume that the 16 memory cells are read in order, from 0 to 15. For each address line fault, the actual order of reads that eventually happen is listed. Notice that ADDR(X) s.a. 0 faults cause the actual address to be smaller than the intended address, for values of the intended address that include ADDR(X) = '1'. Respectively, ADDR(X) s.a. 1 faults cause the actual address to be bigger than the intended address, for values of the intended address that include ADDR(X) = '0'. The key to distinguishing memory cell faults from address line faults is that the former kind always causes only one incorrect result when reading all contents of the LUT, whereas the latter kind causes more than one incorrect result with a suitable configuration.

3.1.1.2 The Multiplexers

Multiplexers of the Virtex-II slice have the duty of managing the outputs of the logic element (the LUT). As mentioned before, there are two different kinds of multiplexers in the slice, the difference being the availability of the control signal in run-time. The control inputs of MUXF5 and MUXFX are accessible during run-time, since they are the extra inputs of a more-than-4-input boolean function. On the other hand, the configuration multiplexers' control input is defined in configuration time and will never change unless a reconfiguration is performed. From the diagnosis standpoint, though, and more specifically regarding fault models, all these multiplexers are treated in the same way.

Our starting point for choosing a suitable fault model for the multiplexers was the fact that we didn't have any information about its internal structure. For this reason, we opted for a functional fault model. The truth table of a multiplexer is seen in Table 3.2. A complete functional fault model could be used, according to which all 8 combinations of inputs should be applied to the multiplexer and the output checked to verify the correct function. In this way, we could make sure that the multiplexer works properly, regardless of its internal structure.

However, by orienting our thought towards graceful degradation, we opted for a more relaxed functional fault model. Notice that in this section we talk only about faults coming from within the multiplexer. The inputs and outputs are tested as wires, as will be explained in Section 3.1.1.4. Our first observation was about the faults denoted

Table 3.1: Effects of address line faults on the effective read address.

Fault	Actual Read Address															
Fault-free	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ADDR(0) s.a. 0	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
ADDR(0) s.a. 1	1	1	3	3	5	5	7	7	9	9	11	11	13	13	15	15
ADDR(1) s.a. 0	0	1	0	1	4	5	4	5	8	9	8	9	12	13	12	13
ADDR(1) s.a. 1	2	3	2	3	6	7	6	7	10	11	10	11	14	15	14	15
ADDR(2) s.a. 0	0	1	2	3	0	1	2	3	8	9	10	11	8	9	10	11
ADDR(2) s.a. 1	4	5	6	7	4	5	6	7	12	13	14	15	12	13	14	15
ADDR(3) s.a. 0	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
ADDR(3) s.a. 1	8	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15

Table 3.2: Multiplexer truth table. Faults considered explicitly by the multiplexer fault model are in bold.

IN1	IN2	CTRL	OUT (fault-free)	OUT (faulty)
0	0	0	0	1
0	0	1	0	1
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	1	0
1	1	1	1	0

in rows 1 and 2 of Table 3.2. It is extremely unlikely for known realizations that only one of these faulty behaviors appears. The same is true for faults of rows 7 and 8. Additionally, in the more likely case of faults 1 and 2 (or 7 and 8) to exist at the same time, the misbehavior will be detected on the output wire. From a graceful degradation standpoint, the presence of any of the above faults would probably render the multiplexer unusable, thus they don't have to be explicitly diagnosed. We decided not to consider input combinations 1, 2, 7 and 8.

Following the above reasoning, we focused on the cases that the two inputs of the

multiplexer are different, corresponding to rows 3, 4, 5 and 6 of Table 3.2. In other words, we check whether or not the multiplexer is able to choose the correct input. As a result, we consider the case that the multiplexer is unable to select input IN1 (corresponding to the control signal being stuck at 1) and the case that it is unable to select input IN2 (corresponding to the control signal being stuck at 0). The majority of all other misbehaviors is covered by checking the input and output wires of the multiplexer.

3.1.1.3 The Storage Elements

The next element that needed to be examined and a fault model chosen for it, was the storage element (flip-flop) of every half of the slice. As with every other slice component, we opted for a functional model to check if the flip-flops can function as expected. The approach we chose in this case was pretty straightforward: We fed the flip-flops with the sequence 01100 (or 10011), to check if all possible transitions ($0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$ and $1 \rightarrow 1$) of the flip-flop are problem-free. This approach for flip-flops is also used in scan design testing [30], to check the scan chain before testing the surrounding logic. It is also important to note that if the flip-flop is viewed as a single memory cell, the above sequence covers all faults of Table 2.2.

Also notice that it is not important which transition the flip-flop fails to support. The input sequence of the flip-flop in run-time cannot be predicted. Thus, from a graceful degradation standpoint, every flip-flop fault renders the flip-flop unusable.

3.1.1.4 The Wires and Fanout Branches

Except from the important components of the slice, there are wires connecting them. It is very important to also consider these wires in formulating the diagnosis method. From a functional point of view, a faulty wire can have serious effects on the slice correct functioning. From a graceful degradation standpoint, it is important to spot the faulty wire, since it can give us information about which parts of the slice have to be marked as unusable. Focusing on the wires is also important because they cover some misbehaviors not covered by the individual components' fault models, because their internal realization details are unknown.

For diagnosing wire faults, we chose the very popular model of the single stuck-at fault. Thus, two faults are defined for each of the circuit wires, namely "wire" s.a. 0 and "wire" s.a. 1. When a fanout branch is encountered, it is treated as 3 separate wires.

After all fault models were chosen, we could proceed to create the list of faults to be checked (Table 3.3). Let us note at this point that the 10 memory faults for each individual cell have been grouped together in 2 faults: MF0 means that a memory fault occurred resulting in a '0' being read from a cell that is supposed to contain an '1' and MF1 is a memory fault resulting in an '1' being read from a cell that is supposed to contain a '0'. In the general case, it is not important to know exactly which memory fault occurred. The memory cell, in any case, is only usable if it is configured to the value that will eventually be read from it (more on that in Section 3.4). There is a small exception to this rule, which will be explained in Section 3.3. Also note that, as is visible in Figure 3.2, the SHIFTIN (G or F), bypass input of the flip-flop (BY or BX) and the

control signal of the function-combining multiplexer (FXCTRL and F5CTRL) of each half of the slice are essentially the same input and as such they are diagnosed.

With the diagram of the CUT and the set of possible faults known, we can devise a strategy to diagnose each one of them. Since different groups of faults have different characteristics, they have different effects on the CUT function. We will now explain how each fault appears on the outputs of the CUT, assuming that only a single fault is present at a time, as is common practice in many related FPGA testing works. These unique properties are the starting point for developing the two methods, each time adapting to the specifics of the situation:

- Memory faults appear on at least one output only when we read the contents of the faulty memory cell, which means *only during one cycle*. This singular effect of memory faults is what makes them easy to diagnose.

- Address line faults potentially appear on an output of the CUT on *half the cycles of the process*, as explained in Table 3.1. Whether or not this potential appearance manifests, depends on the contents of the intended-to-be-read cell and the actually-being-read cell. When they are different, the fault appears on the output. This special property is used in different ways for each of the two methods and allows the formulation of distinct faulty responses depending on the specific address line fault.

- Faults of multiplexers, have been reduced, as explained in Section 3.1.1.2, to faults of their control signals. By trying to keep the inputs of each multiplexer complementary, we make sure that they appear on the output *on every cycle*. Even when it is not always possible to keep the multiplexer inputs complementary, the response is faulty on every cycle that this is possible, still defining a unique property for multiplexer faults.

- Faults of wires force the output that lies on the transitive fanout of the faulty wire to be *fixed to a steady value*. Even when the wires pass through a multiplexer, we know during which cycles they are selected and it is during these cycles that the respective outputs assume a steady value. Although it is trivial to conclude that the fault is on some wire, it is often harder to tell the exact wire. This is achieved by observing which outputs are affected by the fault.

- Faults of storage elements are diagnosed on the cycle that the faulty transition is supposed to occur.

- Any conflict between different faults, meaning that different faults have the same faulty response is, according to the above properties, temporary. The specific fault can eventually be diagnosed by applying the correct test vectors.

After the framework description has been completed, it is time to proceed to the description of the methods themselves. Each of them targets the faults of Table 3.3, aiming to yield the best achievable diagnostic resolution in the least possible time. Although our designs perform fault diagnosis, from this point henceforth, we are going to refer to them as “testers” for convenience.

Table 3.3: Complete list of faults

Complete list of faults			
f0: FXINA s.a.0	f38: GDATA5 MF0	f76: FADDR(1) s.a.0	f114: FENABLE s.a.0
f1: FXINA s.a.1	f39: GDATA5 MF1	f77: FADDR(1) s.a.1	f115: FENABLE s.a.1
f2: FXINB s.a.0	f40: GDATA6 MF0	f78: FADDR(2) s.a.0	f116: FD s.a.0
f3: FXINB s.a.1	f41: GDATA6 MF1	f79: FADDR(2) s.a.1	f117: FD s.a.1
f4: FXIN/ BY/GSHIFTIN s.a.0	f42: GDATA7 MF0	f80: FADDR(3) s.a.0	f118: XMUX1 s.a.0
f5: FXIN/ BY/GSHIFTIN s.a.1	f43: GDATA7 MF1	f81: FADDR(3) s.a.1	f119: XMUX1 s.a.1
f6: FXOUT s.a.0	f44: GDATA8 MF0	f82: FDATA0 MF0	f120: XMUX2 s.a.0
f7: FXOUT s.a.1	f45: GDATA8 MF1	f83: FDATA0 MF1	f121: XMUX2 s.a.1
f8: FX s.a.0	f46: GDATA9 MF0	f84: FDATA1 MF0	f122: MUXCTRLX s.a.0
f9: FX s.a.1	f47: GDATA9 MF1	f85: FDATA1 MF1	f123: MUXCTRLX s.a.1
f10: YMUX1 s.a.0	f48: GDATA10 MF0	f86: FDATA2 MF0	f124: X s.a.0
f11: YMUX1 s.a.1	f49: GDATA10 MF1	f87: FDATA2 MF1	f125: X s.a.1
f12: YMUX2 s.a.0	f50: GDATA11 MF0	f88: FDATA3 MF0	f126: DY s.a.0
f13: YMUX2 s.a.1	f51: GDATA11 MF1	f89: FDATA3 MF1	f127: DY s.a.1
f14: MUXCTRLY s.a.0	f52: GDATA12 MF0	f90: FDATA4 MF0	f128: DYCTRL s.a.0
f15: MUXCTRLY s.a.1	f53: GDATA12 MF1	f91: FDATA4 MF1	f129: DYCTRL s.a.1
f16: Y s.a.0	f54: GDATA13 MF0	f92: FDATA5 MF0	f130: GQ s.a.0
f17: Y s.a.1	f55: GDATA13 MF1	f93: FDATA5 MF1	f131: GQ s.a.1
f18: GENABLE s.a.0	f56: GDATA14 MF0	f94: FDATA6 MF0	f132: YQ s.a.0
f19: GENABLE s.a.1	f57: GDATA14 MF1	f95: FDATA6 MF1	f133: YQ s.a.1
f20: GADDR(0) s.a.0	f58: GDATA15 MF0	f96: FDATA7 MF0	f134: Y REG 0 $\rightarrow$ 0
f21: GADDR(0) s.a.1	f59: GDATA15 MF1	f97: FDATA7 MF1	f135: Y REG 0 $\rightarrow$ 1
f22: GADDR(1) s.a.0	f60: GD s.a.0	f98: FDATA8 MF0	f136: Y REG 1 $\rightarrow$ 0
f23: GADDR(1) s.a.1	f61: GD s.a.1	f99: FDATA8 MF1	f137: Y REG 1 $\rightarrow$ 1
f24: GADDR(2) s.a.0	f62: ENABLE s.a.0	f100: FDATA9 MF0	f138: DX s.a.0
f25: GADDR(2) s.a.1	f63: ENABLE s.a.1	f101: FDATA9 MF1	f139: DX s.a.1
f26: GADDR(3) s.a.0	f64: F5IN1 s.a.0	f102: FDATA10 MF0	f140: DXCTRL s.a.0
f27: GADDR(3) s.a.1	f65: F5IN1 s.a.1	f103: FDATA10 MF1	f141: DXCTRL s.a.1
f28: GDATA0 MF0	f66: F5IN2 s.a.0	f104: FDATA11 MF0	f142: FQ s.a.0
f29: GDATA0 MF1	f67: F5IN2 s.a.1	f105: FDATA11 MF1	f143: FQ s.a.1
f30: GDATA1 MF0	f68: F5CTRL/ BX/FSHIFTIN s.a.0	f106: FDATA12 MF0	f144: XQ s.a.0
f31: GDATA1 MF1	f69: F5CTRL/ BX/FSHIFTIN s.a.1	f107: FDATA12 MF1	f145: XQ s.a.1
f32: GDATA2 MF0	f70: F5OUT s.a.0	f108: FDATA13 MF0	f146: X REG 0 $\rightarrow$ 0
f33: GDATA2 MF1	f71: F5OUT s.a.1	f109: FDATA13 MF1	f147: X REG 0 $\rightarrow$ 1
f34: GDATA3 MF0	f72: F5 s.a.0	f110: FDATA14 MF0	f148: X REG 1 $\rightarrow$ 0
f35: GDATA3 MF1	f73: F5 s.a.1	f111: FDATA14 MF1	f149: X REG 1 $\rightarrow$ 1
f36: GDATA4 MF0	f74: FADDR(0) s.a.0	f112: FDATA15 MF0	
f37: GDATA4 MF1	f75: FADDR(0) s.a.1	f113: FDATA15 MF1	

3.2 The Function Generator Based Method

As it has already been mentioned, the difference between the two methods we developed, is mainly the manner in which the "heart" of the CUT, namely the two LUTs, are configured. In the first method, as its name suggests, the LUTs are configured as function generators, each of them realizing a 4-input boolean function. This actually is the most usual approach throughout the existing literature, since multi-phase approaches configure the LUTs as function generators during most of the phases. This assumingly happens because it is the most frequent manner of configuration for the LUT. Testing the LUT in the mode that it will usually be used is considered by researchers to be a better indication on whether or not it works properly.

Following this trend, we also started our efforts to solve the problem at hand by configuring the LUT as a function generator. We took into account the fault models that need to be covered in order to derive a suitable 16-bit configuration for each LUT. As already mentioned, we used a bottom-up approach to derive the LUT configurations. Since our tester is able to terminate when drawing a definite conclusion, we want to push the diagnosis of as many faults as possible to as early cycles as possible, in order to reduce the average running time. Thus, the first test patterns are going to be critical, since they have to expose many faults in different parts of the CUT. In general, the function generator based method is based on techniques that are popular in existing literature, but adapted to the needs of our fault model and to the goal of diagnosis.

The configuration of the LUT as a function generator has as a direct consequence the need of two testing phases. Indeed, to fulfill the requirement of reading both possible values from each memory cell of the LUT, we need, after reading the original values, to reconfigure it to the exact complementary values. The need for a second phase is also what incited the development of the second method, to eliminate this need. This will become obvious after Section 3.3. The need for reconfiguration also urged us to diagnose as many faults as possible in the first phase, in order to reduce the percentage of CUTs that require a second phase.

3.2.1 Partitioning of the Faults

As it can be seen in Figure 3.2, the CUT has 6 outputs. Each fault on a component of the circuit can affect a subset of these outputs. In general, the set of fault sites that a fault on a given site can affect is called the *transitive fanout* of this fault. For example, output Y is in the transitive fanout of (the faults of) wire YMUX2, whereas output X is not. By observing the natural partitioning of functionality among the parts of the slice, we decided to use it in order to have each output be responsible to diagnose a well-defined subset of the faults. In other words, we partitioned the faults among the outputs of the circuit under test.

Another tool we used to achieve the partitioning was the control signal of the slice multiplexers. By keeping the control signal of each multiplexer to a fixed value, we essentially transform it to a short-circuit of the selected input to the output, at the same time cutting off the non-selected input. By this cutting-off, the circuit is separated in independent parts. In Figure 3.3 the partitioning of the CUT during the first phase can

be seen, while in Figure 3.4 the respective partitioning during phase 2 is depicted.

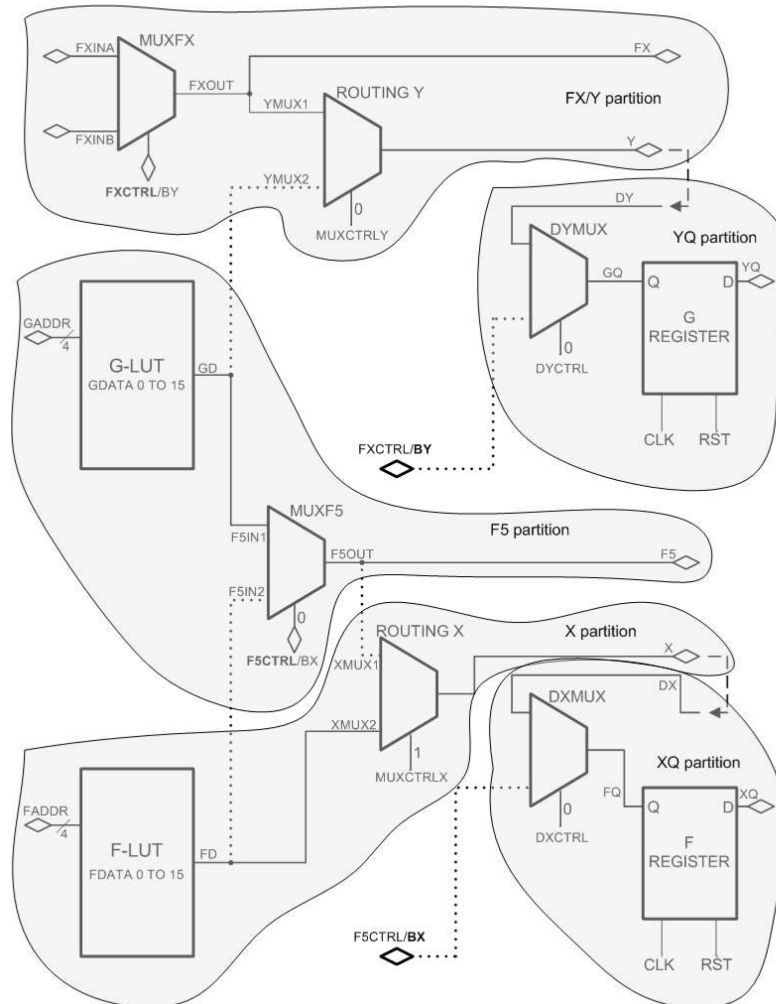

Figure 3.3: The CUT partitioning for phase 1 of the function generator method.

A consequence of keeping the multiplexer signals to a fixed value during the whole phase, is that the non-selected input wire of the multiplexer becomes untestable for this phase. Also, from the two possible faults of the control signals, one becomes untestable. For example, by fixing the value of the MUXF5 control signal to '0' during phase 1, the faults of signal F5IN2 and F5CTRL s.a. 0 become untestable. Thus, the testing of the multiplexers that are used for the partitioning (that is all except the MUXFX), have to be divided between the two phases. Additionally, in order to actually be able to detect the testable fault of the control signal (i.e. the F5CTRL s.a. 1 in the example case of phase 1) we have to keep the two inputs of the multiplexer complementary as often as possible. This guided us to opt for complementary configurations between the two LUTs during the same phase, since they are both inputs of MUXF5.

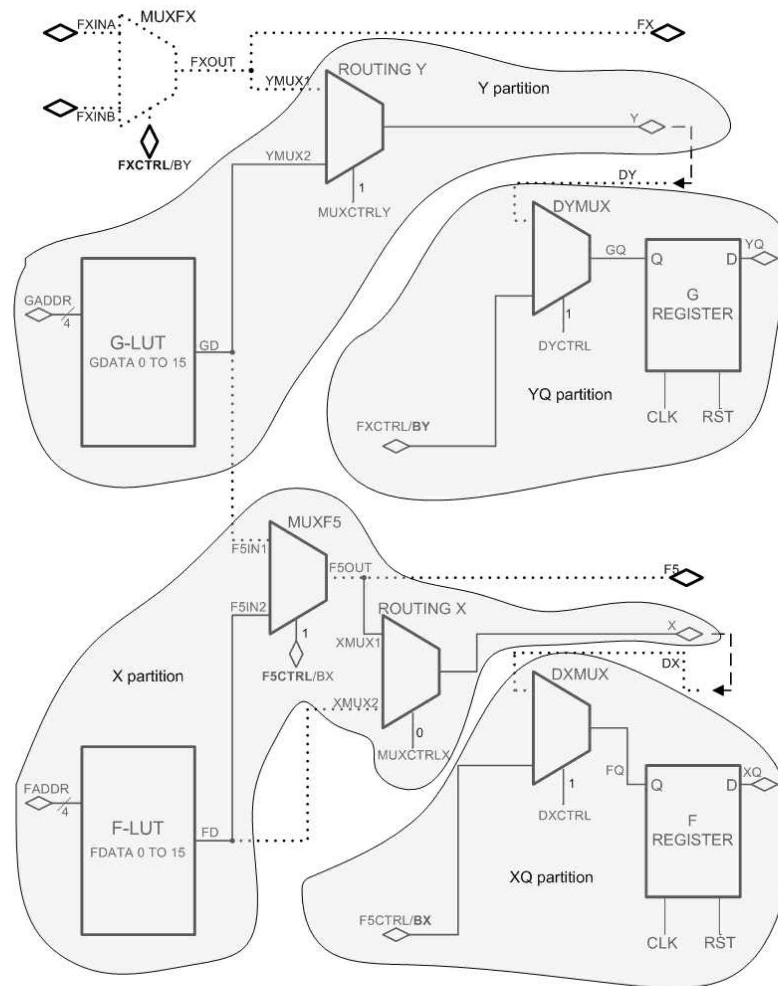

Figure 3.4: The CUT partitioning for phase 2 of the function generator method.

3.2.2 Deriving the LUT Configurations

The decision of the two LUTs being configured with complementary contents was the starting point for deriving their actual contents. The second step was defining the addressing scheme. Since all contents of each LUT have to be read exactly once during each phase, our decision was to address the LUTs with a 4-bit counter, as is common practice in many function generator based techniques. Following these decisions, we had to move on to the actual contents.

Focusing on the F-LUT, the first constraint came from the fact that the test for the F-REGISTER has to be applied as early as possible. Since in the first phase the F-register is tested with the output of the F-LUT, the lower-addressed bits of the configuration sequence (bits 0 to 4) should be 10011. Bits 0 to 4 of the G-LUT were accordingly determined to be 01100.

The most important purpose, though, of the LUT configuration is to help diagnose the address line faults. Remember that, since memory faults cause one bit of the response

Table 3.4: Detection of the address line faults according to the chosen LUT configuration. A red entry denotes a difference between the intended and actual read address that appears on the LUT output, while a green entry denotes a difference between the intended and actual read address that goes unnoticed.

Contents	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1
Fault	Actual Read Address															
Fault-free	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ADDR(0) s.a. 0	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
ADDR(0) s.a. 1	1	1	3	3	5	5	7	7	9	9	11	11	13	13	15	15
ADDR(1) s.a. 0	0	1	0	1	4	5	4	5	8	9	8	9	12	13	12	13
ADDR(1) s.a. 1	2	3	2	3	6	7	6	7	10	11	10	11	14	15	14	15
ADDR(2) s.a. 0	0	1	2	3	0	1	2	3	8	9	10	11	8	9	10	11
ADDR(2) s.a. 1	4	5	6	7	4	5	6	7	12	13	14	15	12	13	14	15
ADDR(3) s.a. 0	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
ADDR(3) s.a. 1	8	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15

to be erroneous, we have to make sure that each address line fault causes at least two erroneous responses. An erroneous response occurs when the contents of the cell intended to be read are different than the contents of the cell eventually being read. Referring to Table 3.1, it can be verified that the first 5 bits of the configuration sequence already offer adequate information to diagnose the faults of ADDR(0), since $DATA(0) \neq DATA(1)$ and $DATA(2) \neq DATA(3)$. The same holds true for the faults of ADDR(1), since $DATA(0) \neq DATA(2)$ and $DATA(1) \neq DATA(3)$. To trigger the appearance of faults of ADDR(2), we set $DATA(5) \neq DATA(1)$ and $DATA(7) \neq DATA(3)$. Finally, to sensitize the faults of ADDR(3) as early as possible and also not confuse them with the ADDR(1) faults, we set $DATA(8) \neq DATA(0)$, $DATA(9) \neq DATA(1)$ and $DATA(10) \neq DATA(2)$. The rest of the bits are not important and are set to the same value for convenience (0 for the G-LUT and 1 for the F-LUT). In Table 3.4 we revisit Table 3.1, this time marking in red a read operation that happens on the wrong address and becomes apparent in the output and in green a wrong read operation that goes unnoticed with the particular LUT contents. According to this, all address line faults produce different faulty responses.

The result of the process described above was the complete configurations for the G- and F-LUTs. To sum up, during the first phase the G-LUT is configured to

the value (bit 0 to 15) "0110001110000000" and the F-LUT respectively to the value "1001110001111111". During the second phase, the two configurations are interchanged. Having completed the formulation of the testing configuration (consisting of the LUT configurations and the multiplexer control signals), we have to list specifically the faults diagnosed on each output during each of the phases.

3.2.3 Deriving the Diagnostic Trees

Partitioning of the CUT allowed us to distribute the complexity of diagnosis over the different outputs. Indeed, as will be in detail explained in Chapter 4, instead of having one complex Response Analyzer (RA) to process all the outputs of the CUT, there is one relatively simple RA attached to every output, processing this single-bit output. This also simplifies the diagnostic trees, which otherwise would be unmanageable to formulate manually.

In this section we will offer an overview of the duties that correspond to each response analyzer of the function generator based method.

3.2.3.1 Phase 1 Response Analysis

During the first phase, there are essentially 5 partitions of the CUT, corresponding to the outputs FX and Y, X, F5, YQ and XQ.

The FX and Y Response Analyzer: This is actually the only RA that examines two outputs of the CUT instead of 1. This does not critically affect the complexity, since the FX/Y RA performs only a few checks: It is responsible for completely checking the MUXFX multiplexer, which is the only one that doesn't need to have a fixed value for its control signal, also for checking the ROUTING Y multiplexer with its control signal set to '0' and finally for the faults of the following wires: FXINA, FXINB, FXOUT, FX, YMUX1 and Y.

To manage these tests, the inputs of this partition (FXINA, FXINB and FXCTRL) are given the values listed in Table 3.2, corresponding to the multiplexer test as it was defined in Section 3.1.1.2. This is done in order to complete the MUXFX test, but it is proven enough to diagnose every other fault as well. In detail, the different responses of every fault are explained below:

- Both FXCTRL faults appear on FX and Y, as always choosing the FXINA or always choosing the FXINB input. Namely, FXCTRL s.a. 0 will cause the output to be "0011", while the presence of FXCTRL s.a. 1 will cause the output "1100".

- Faults of FXINA and FXINB will appear on both outputs only when FXCTRL chooses the faulty input. For example, the fault FXINA s.a. 0 will cause an output of "1000", while FXINB s.a. 0 will cause "0001".

- FXOUT faults will appear on both outputs as a constant '0' or constant '1' output.

- FX faults will appear in the same way only on output FX itself.

- YMUX1 and Y faults will appear only on output Y. Note that faults of these 2 wires can not be distinguished with the setup of phase 1.
- The fault MUXCTRLY s.a. 1 appears on the Y output, since the YMUX2 input of the ROUTING Y multiplexer is always complementary to YMUX1.

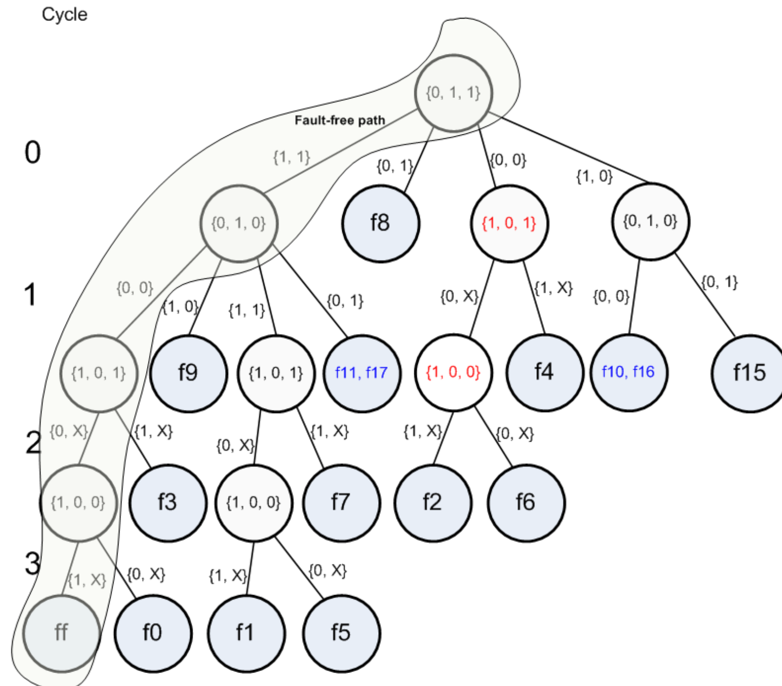

Figure 3.5: The FX/Y diagnostic tree for the first phase. Test vectors are visible on the non-leaf nodes, while the CUT responses annotate the branches. Alternative vectors to distinguish between faults are shown in red. Leaf nodes contain the RA conclusion, which is shown in blue when a 2nd phase is required for further disambiguation.

The resulting diagnostic tree can be seen in Figure 3.5. Every level of the diagnostic tree corresponds to one clock cycle of the testing procedure. The nodes correspond to test vectors in the order FXINA, FXINB, FXCTRL, while the links between them are annotated by the response of the particular partition to this test vector. The leaf nodes, that are also shown in different color, correspond to conclusions of the RA. When reaching a leaf node, no further diagnosis can be performed during this phase. Notice that when a faulty response occurs, for which it can not immediately be concluded which the fault that caused it was, an alternative test vector has been occasionally determined, which is suitable for distinguishing between the faults that actually could have caused the faulty response. These alternative vectors are shown in red. We also show the fault-free path, which corresponds to always receiving a fault-free response and continuing with the predefined vectors. This is in general the form and structure of all diagnostic trees that will be presented in this chapter, excluding the fact that only for this tree the response is 2-bit, while for every other tree it is single-bit.

The F5 Response Analyzer: This RA is responsible for diagnosing all address faults of the G-LUT and also half the memory faults, corresponding to the configuration of phase 1. It is also responsible for testing only the s.a. 0 fault for the control signal of the MUXF5 multiplexer and, finally, for the faults of wires GD, F5IN1, F5OUT and F5. To achieve that, a 4-bit counter addresses the G-LUT, resulting in a read operation on

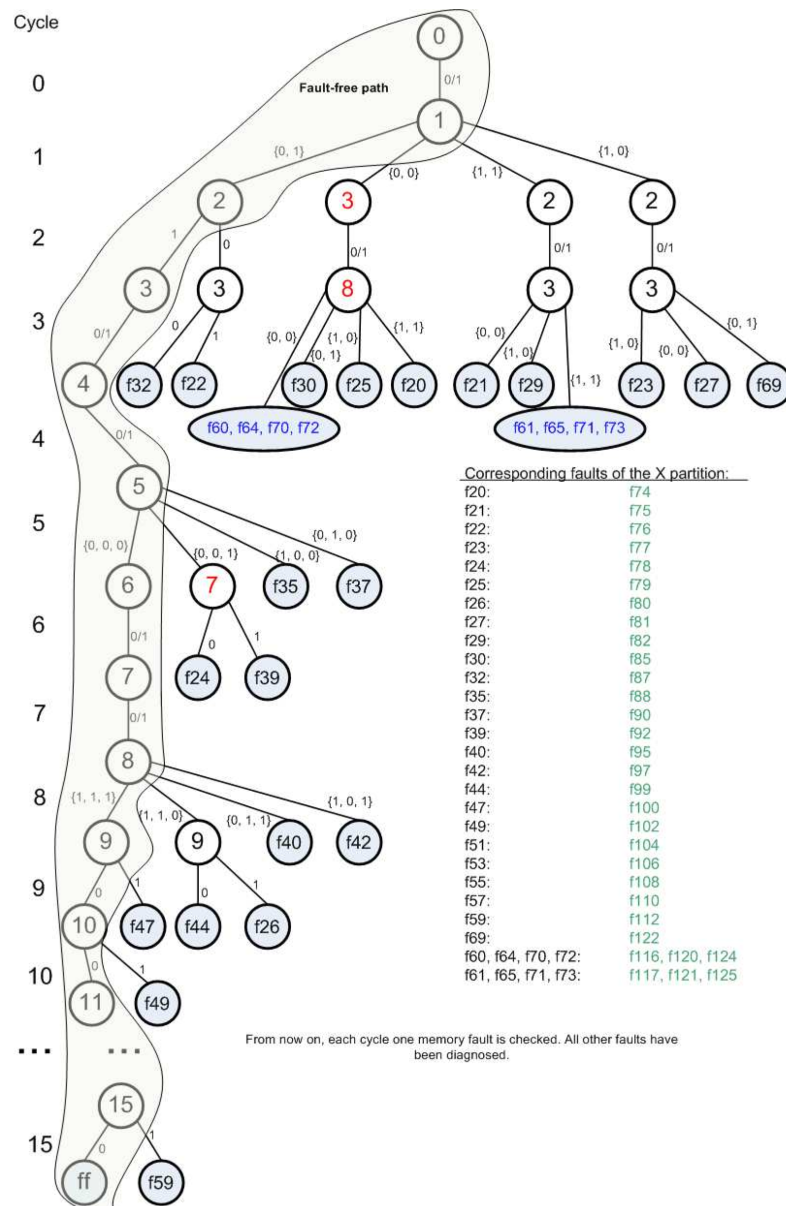

Figure 3.6: The F5 and X diagnostic tree for the first phase. The tree is annotated with the values relevant for the F5 partition. The tree for the X partition would be annotated with the exact complimentary values on the branches and every leaf node would diagnose the corresponding fault of the X partition, as they are listed in green color.

all its entries. The different faults will appear on output F5 in the following ways:

- The address line faults appear on the output on the cycles marked in red for each of them in Table 3.4.
- The memory faults cause exactly one bit of the output to be faulty. The clock cycle on which the faulty response appears denotes the LUT memory cell that is faulty.
- The fault F5CTRL s.a. 1 makes the exact complementary of the fault-free response to appear on the output. It is, thus, very easily recognizable.
- The faults of wires GD, F5IN1, F5OUT and F5 appear on the output as a response of constant '0' or constant '1'. No further clarification is possible with the setup of phase 1.

The resulting diagnostic tree can be seen in Figure 3.6. The numbers in the non-leaf nodes are the read address for the particular cycle, which is the test vector for this partition. Notice that a decision about the presence of a fault or about possible alternative vectors is not made on every clock cycle. Such choices are made on cycles 1, 2, 5 and 8, because these are the cycles that address line faults of line 0, 1, 2 and 3 respectively appear. In this sense, those cycles are a kind of checkpoints for the diagnosis process. On each checkpoint the RA decides, by inspecting a window of the response, whether the response is fault-free, or there was some faulty response which has to be resolved to a specific diagnosis. The tree is denser during the first cycles, since we made every possible effort to push as many outcomes as possible closer to the beginning of the process. In fact, starting on cycle 10, every faulty response is automatically resolved to be caused by a memory fault, since memory faults have to be checked one at a time.

The X Response Analyzer: This RA actually has identical duties with the F5 RA. It has to diagnose all address line faults and half the memory faults of the F-LUT, the MUXCTRLX s.a. 0 fault for the ROUTING X multiplexer and the faults of wires FD, XMUX2 and X. All of these faults are diagnosed on output X in exactly the same way that the corresponding faults on output F5 were diagnosed. Notice that faults FD, XMUX2 and X all have the same faulty responses and cannot be further clarified with the setup of phase 1.

As a result of this symmetry, the diagnostic tree for output X is exactly identical to that of output F5. In Figure 3.6, the faults corresponding to the X output on every cycle are shown in green color.

The YQ Response Analyzer: This RA is responsible mainly for checking the faults of the G REGISTER. Additionally, it has to check the fault DYCTRL s.a. 1 of the DYMUX multiplexer and the faults of wires DY, GQ and YQ. These faults are differentiated on the YQ output in the following way:

- In the presence of DYCTRL s.a. 1, the F5CTRL/BY input appears on the YQ output, making it easy to recognize.

- The $0 \rightarrow 0$ and $1 \rightarrow 1$ transition faults produce distinct faulty responses on the YQ output.
- The s.a. 0 fault of DY, GQ and YQ, as well as the $0 \rightarrow 1$ transition fault of the flip flop coincide regarding their faulty responses. The same happens with the s.a. 1 faults of DY, GQ and YQ, as well as the $1 \rightarrow 0$ transition fault of the flip flop. These fault groups can not be further clarified with the setup of phase 1.

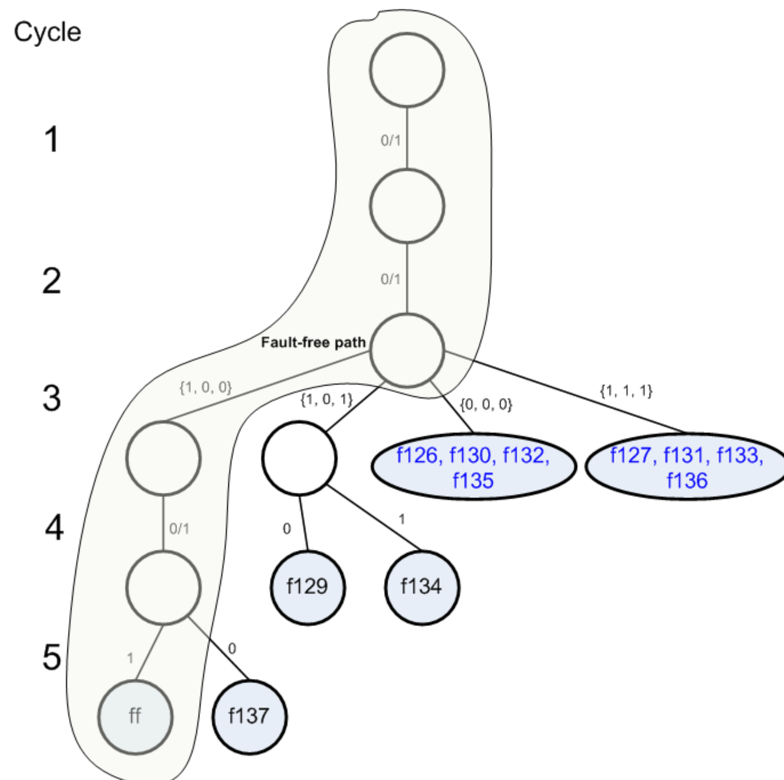

Figure 3.7: The YQ diagnostic tree for the first phase. As YQ is a passive partition, meaning that it does not contain any primary inputs of the CUT, there are no test vectors in the non-leaf nodes.

The resulting diagnostic tree is shown in Figure 3.7. Notice that this tree does not have a test vector value in each node, since this partition does not contain any primary inputs of the CUT. Also, changing the inputs that eventually propagate to the DY input of the DYMUX multiplexer would not offer any significant gain in efficiency, since the test takes only a few cycles anyway. Also notice that the tree starts on cycle 1 instead of cycle 0, because the flip-flop delays its input for one cycle, causing no meaningful output during the first cycle (cycle 0).

The XQ Response Analyzer: The XQ RA is almost identical to the YQ RA. The only difference is that, since F5CTRL/BX is kept at value '0' for the whole phase 1, the fault DXCTRL s.a. 1 coincides with the s.a. 0 fault of wires DX, FQ and XQ.

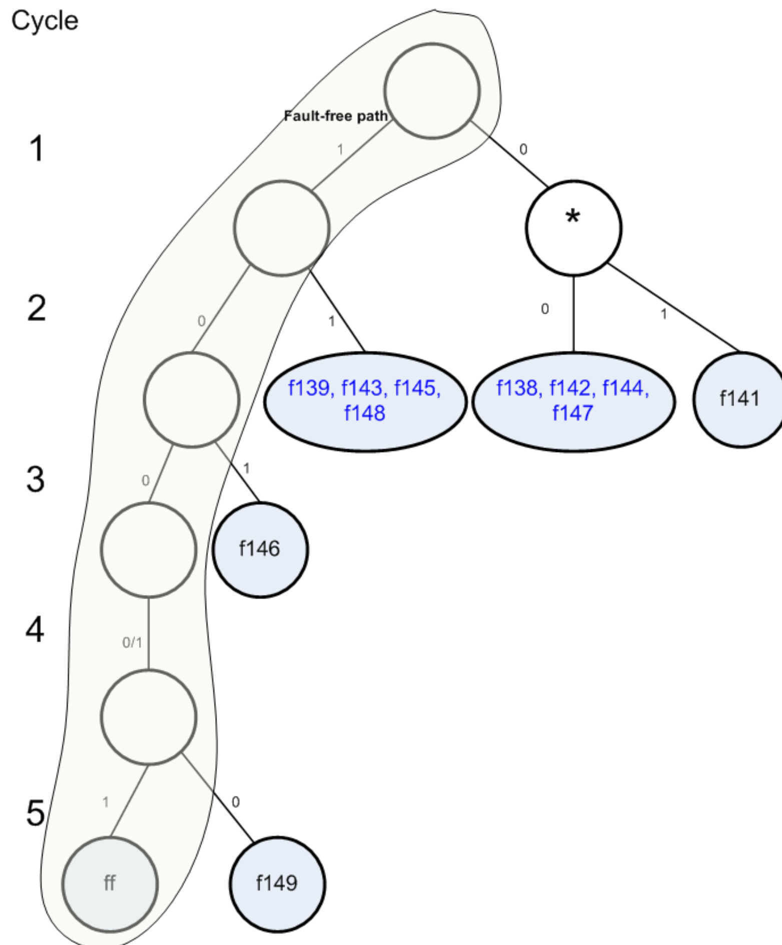

Figure 3.8: The XQ diagnostic tree for the first phase. The node annotated with a star denotes the change of F5CTRL/BX to '1' for the next clock cycle, in order to distinguish the DXCTRL s.a. 1 fault from others that otherwise have the same response.

For this purpose, when the response of this fault group appears on the XQ output, the input F5CTRL/BX is changed to '1' for 1 clock cycle, to diagnose the fault DXCTRL s.a. 1, since this would be impossible during phase 2. This results in the diagnostic tree of Figure 3.8.

Summary of phase 1: Before we go on to derive the diagnostic trees for phase 2, we will recap the result of phase 1. Each fault of our fault list belongs to one of the following categories after phase1:

- **Completely diagnosed:** These faults are diagnosed with certainty. In the occurrence of one of these faults, there is no need for a second phase. The completely diagnosed faults are:
  - All address line faults of both LUTs.
  - Half of the memory faults of each LUT.

  - Both faults of the MUXFX multiplexer.
  - One of the two faults of every other multiplexer.
  - The faults of wires FXINA, FXINB, FXOUT and FX.
  - Half of the storage element faults.

- **Partially Diagnosed:** Some groups of faults, though detected, produce the same response and it cannot be clarified which of the faults in the group caused this faulty response. These fault groups have to be revisited during phase 2:
  - The faults of wires YMUX1 and Y.
  - The faults of wires GD, F5IN1, F5OUT and F5.
  - The faults of wires FD, XMUX2 and X.
  - The faults of wires DY, GQ and YQ, as well as two of the four transition faults for the G REGISTER.
  - The faults of wires DX, FQ and XQ, as well as two of the four transition faults for the F REGISTER.

- **Not Sensitized:** These faults are redundant, hence untestable, under the current configuration. They have to be sensitized and diagnosed during phase 2:
  - One of the two faults of every multiplexer except MUXFX.
  - The faults of the wires YMUX2, F5IN2 and XMUX1.
  - Half of the memory faults of each LUT, corresponding to the configuration values of phase 2.

The way that phase 2 deals with the faults of the last 2 categories will be explained in the next section.

3.2.3.2 Phase 2 Response Analysis

The efficiency of the function generator based method depends heavily on whether or not a conclusion can be drawn during the first phase. If this does not happen, the consequence is a big time penalty. The diagnosis, though, has to proceed anyway and this is the role of phase 2. The setup for phase 2 is summarized in the following points:

- The two LUT configurations are interchanged, resulting in the LUT contents being complementary compared to them of phase 1.
- The control bit of every multiplexer excluding MUXFX, is set to a fixed value complementary to that of phase 1. We refer the reader to Figure 3.4 for the resulting partitioning of the circuit. As a result of this partitioning, phase 2 consists of 4 response analyzers, processing the data of outputs Y, YQ, X and XQ.
- MUXFX and all its inputs and outputs as well as output FX are ignored through this phase, since all their faults have been completely diagnosed.

- During phase 2, almost every faulty response can easily be resolved in the few following cycles, without the need of calculating alternative input vectors. For this reason, we decided to keep the vector generator simple and not use any alternative vectors.

- In the case that phase 1 terminated with partial diagnosis, a *seed* is given to the circuitry of phase 2, denoting that the only duty of phase 2 is to resolve the ambiguity, instead of checking for the whole pool of faults it otherwise has to.

The diagnosis process of the second phase is in detail explained through the description of duties of the 4 response analyzers:

The Y Response Analyzer: In the case of phase 1 resulting to partial diagnosis, this RA has a duty in the following two cases:

- Phase 1 terminated denoting the presence of fault on wire YMUX1 or wire Y: In this case, a faulty response on Y means that the fault is actually on Y, whereas a fault-free response means that the fault is on YMUX1, which is untestable in phase 2.

- Phase 1 terminated denoting the presence of fault on one of wires GD, F5IN1, F5OUT and F5. In this case, faulty response on Y means that the fault is on wire GD, while fault-free response means that it is in one of the other wires.

In the case of no partial diagnosis produced by phase 1, the Y RA has to diagnose the following faults:

- The second half of the G-LUT memory faults, in exactly the same way that the first half is diagnosed during phase 1.

- The fault MUXCTRLY s.a. 0, which appears as an output complementary to the expected one on output Y.

- The faults of wire YMUX2, which are obvious on output Y combined with the absence of seed from phase 1.

The resulting diagnostic tree is shown in Figure 3.9.

The X Response Analyzer: If phase 1 produced partial diagnosis of the faults on wires FD, XMUX2 and X, the X RA has to clarify it during phase 2. If the response during phase 2 is fault-free, it means that the fault is on wire XMUX2, which is untestable during phase 2. If a faulty response is observed on X, it means that the fault was on one of the wires FD and X. In this case, after the 16 first cycles, the control signal of MUXF5 is changed to '0' and the output is observed for 2 more cycles. If the output is still faulty, the fault is on X itself, else it is on wire FD.

In case the first phase didn't produce partial diagnosis, the duties of the X RA are identical to those of the Y RA. Specifically, the X RA has to diagnose the second half of the F-LUT memory faults, the fault MUXCTRLX s.a. 1 and the faults of wire XMUX1.

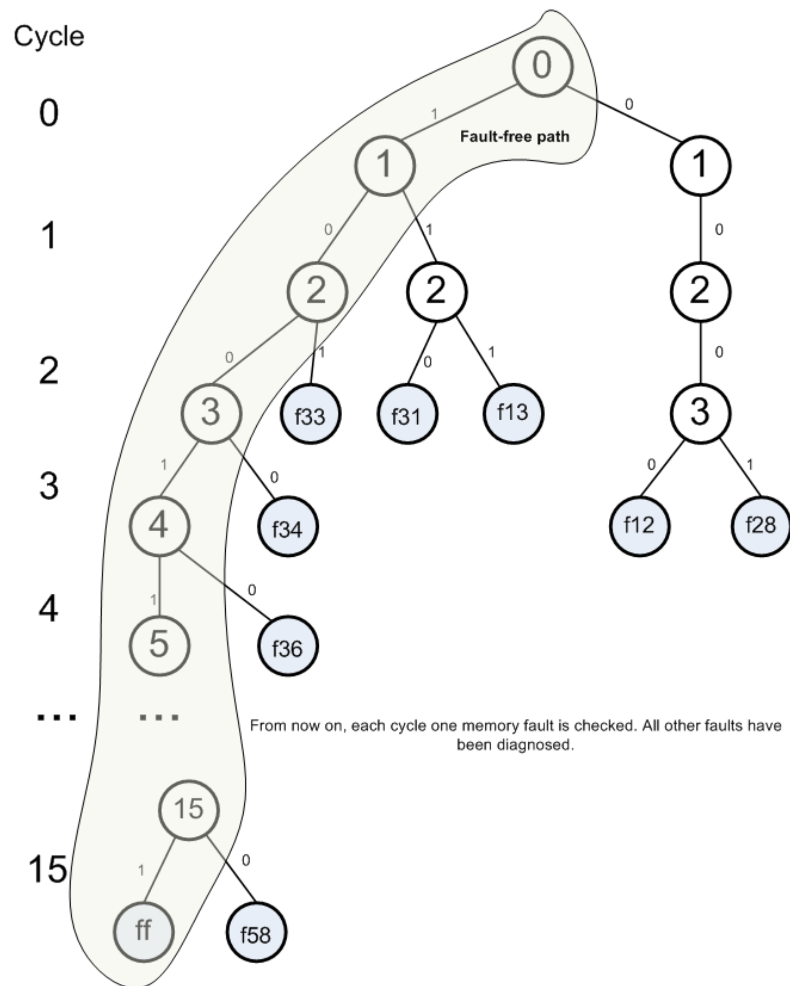

Figure 3.9: The Y diagnostic tree for the second phase, in the case that no partial diagnosis seed is produced by phase 1.

The only difference to the Y RA is that the MUXCTRLX s.a. 1 is not as easy to diagnose as the MUXCTRLY s.a. 0, since both inputs of the ROUTING X multiplexer are driven by the F-LUT during phase 2. For this reason, after the first 16 cycles, the control signal of MUXF5 changes to 0 and propagates the G-LUT output to XMUX1, which is complementary to the F-LUT output that is fed to XMUX2 and allows the diagnosis of MUXCTRLX s.a. 1.

The resulting diagnostic tree is shown in Figure 3.10.

The YQ Response Analyzer: In case phase 1 terminated with a partial diagnosis related to the G REGISTER component, this RA has to decide if the flip-flop is usable through the direct input FXCTRL/BY or unusable at all. If the response is also faulty in phase 2, it means that the fault detected by phase 1 is either a flip-flop transition fault or a fault on one of the wires GQ and YQ. In this case, the flip-flop is unusable altogether. If the response during phase 2 is fault-free, the fault is on wire DY and the

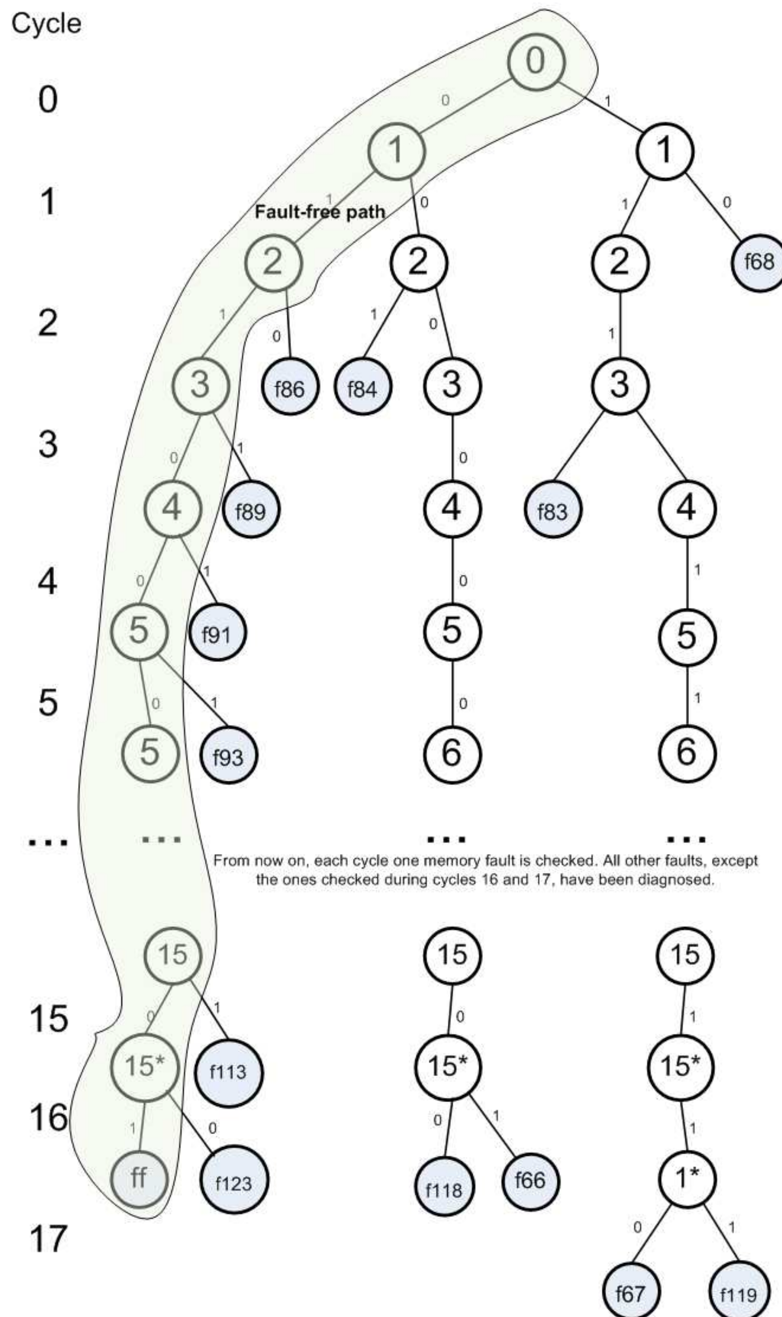

Figure 3.10: The X diagnostic tree for the second phase in the case that no partial diagnosis seed is produced by phase 1. The starred address test vectors after cycle 15 denote that the G-LUT is addressed instead of the F-LUT.

flip-flop is usable through the direct input.

In case there was no partial diagnosis during phase 1, this RA only has to check the fault DYCTRL s.a. 0. This is easily done by observing the response, since we have set

the FXCTRL/BY input to be complementary to DY. No diagnostic tree is required for this single decision.

The XQ Response Analyzer: In case phase 1 terminated with a partial diagnosis related to the F REGISTER component, this RA has to decide if the flip-flop is completely unusable or only usable through the F5CTRL/BX input. This check is performed in exactly the same way as in the YQ RA.

In case there was no partial diagnosis during phase 1, this RA only has to check the fault DXCTRL s.a. 0. This is done by observing the XQ output on a cycle that DX is 0, in order for the DXMUX inputs to be complementary, since F5CTRL/BX is set constantly to 1. No diagnostic tree is required for this single decision.

3.3 The Shift Register Based Method

The function generator based method, as it was completely explained in the previous section, was our first attempt to solve the problem of fault diagnosis and characterization of the Virtex-II FPGA slice. Although it offered good diagnostic resolution, especially on the wires and multiplexers, it had a number of drawbacks, summarized below:

- Its performance is bound by the reconfiguration stage, which, as shown even in recent works, dominates the total testing time when there are different phases.
- It does not detect all 10 memory faults in our fault model for each cell. Namely, although both state faults, incorrect read faults and read destructive faults are covered, only one of the write disturb faults and one of the transition faults are checked, because of the static manner in which data is stored in the LUT when it is configured as a function generator. In order to achieve detection of all 10 faults, more than two phases would be needed, since the function generator configuration does not allow writing on the memory cells during the testing.
- It does not check all the functionality supported by the simplified slice diagram shown in Figure 3.2. More specifically, it does not check the ENABLE wires, thus not making sure that the slice can be configured as a shift register.

The above drawbacks hinted us to look for an alternative solution to our problem. We chose to try the other basic mode of operation of the slice, namely the shift register. According to the drawbacks mentioned above, this new approach obviously had the following advantages:

- Since it allows for the memory cells to be written during the normal operation and not only through configuration, it is free from reconfiguration overhead.
- The ability to change the LUT contents with the shift operation, also means that a memory test can be devised to cover all 10 memory faults of the chosen model easily and quickly.
- Since the slice is configured as a shift register, this mode of operation is obviously checked. On top of that, the shift register mode completely overlaps the function

generator mode, since it uses the same memory cells, the same asynchronous read operation and the same address lines for it.

Of course, the solutions to the problems mentioned above, also come together with some new complications, that make the comparison of the two methods interesting. The most important such complications are listed below:

- Although the shift register mode allows us to modify the LUT contents during the normal testing operation, it does not allow the same for the configuration multiplexers' control signals. Thus, if we keep the testing procedure in one phase, we cannot test one of the two faults for each of the configuration multiplexers. We also cannot test the input that is not selected.
- The presence of two phases in the function generator based method also allowed us to distinguish between some wire faults that appear on the same output in the same way. With the shift register method, this is not always possible.
- Stacking the diagnosis of all faults in one phase somewhat complicates the diagnostic trees, resulting in increased hardware complexity.

The basic guidelines of the method development were the same as the function generator based method: The circuit is partitioned by using the configuration multiplexers' control signals and a different response analyzer detects the faults that can be observed on each output. Upon encountering a faulty response, the response analyzer behaves adaptively, changing the subsequent test vectors whenever an efficiency gain is possible. Since these characteristics are common among the 2 methods, we will present the shift register method by focusing mainly on the differences between them. We will illustrate the partitioning for the shift register method, then explain the memory test that we developed and finally, list the duties of each response analyzer and derive the corresponding diagnostic trees.

3.3.1 Partitioning of the Faults

The partitioning we performed to setup for the shift register method is shown in Figure 3.11. Notice that the main difference from the first method is that the multiplexer MUXF5 is not used for the partitioning, but its control signal (F5CTRL) is used as a variable input, to allow complete test of the MUXF5, since this is possible. Since we can test the flip-flops only in one mode (driven by the LUTs or driven directly), we choose to drive them by the LUT outputs, since this is the way we have seen them used more often in FPGA realizations. Also, the G and F LUT outputs are propagated to the Y and X slice outputs respectively. This way, the basic combinational logic functionality of the slice is checked, which is realizing 4-input boolean functions. Together with the complete MUXF5 test, this means that also the ability to accommodate 5-input boolean functions is checked.

The above partitioning defines the existence of 6 response analyzers. The duties and resulting diagnostic trees of each of them will be explained in Section 3.3.3. Before that, the test patterns of the two shift registers will be derived in Section 3.3.2, to formulate the memory test.

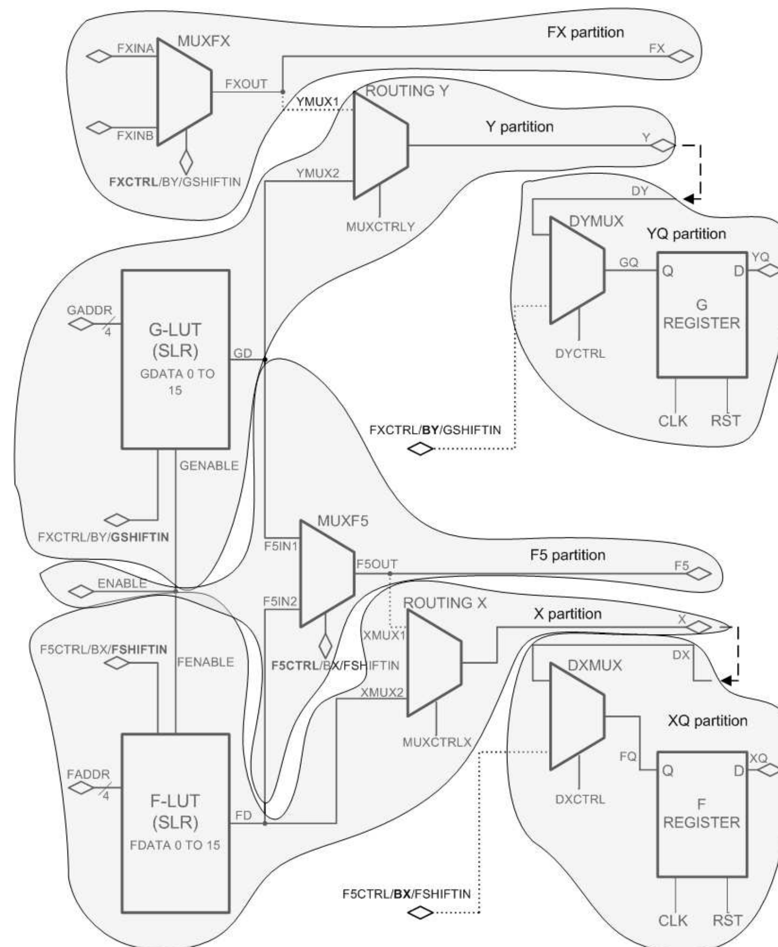

Figure 3.11: Partitioning of the CUT for the shift register method.

3.3.2 The Memory Test

The most important aspect of the shift register method is the way in which it deals with memory faults. It was observed after designing the function generator method that it can not cover all 10 memory faults defined by our fault model. One of the main incentives for developing the shift register method was to improve in this respect.

Since now we have access to the contents of the LUTs, we opted for developing a variation of a *march test* [40], to cover the memory faults. Our write access to the cells is not conventional memory write access, but rather shift-in access. That meant that we had to come up with a way to apply the march test by using the shift operation. Details about those developments will be presented in the following three sections. In these sections we will focus on the F-LUT, but the G-LUT test is exactly identical, with all bit values complementary

3.3.2.1 March Test Formulation and Application

The march test on which we relied to cover all memory faults is the following:

$$\{\uparrow (w0, w1, w1, r1); \uparrow (w0, w0, r0)\}$$

The march test covers all 10 faults of the fault model. To be more specific, the first half of the test detects half of the faults as follows:

- The state fault which causes constant state ‘0’ is detected when reading 1 from each cell, because the cell will always have the value ‘0’.
- The write disturb fault $1w1$ is detected because the second write of 1 will cause the cell to flip to 0 before being read.
- The transition fault $0w1$ will be detected because neither “w1” operation will succeed and the contents of the cell will remain 0 until the read operation.
- The read destructive fault $< 1r1/0/0 >$ is detected because the read operation will flip the contents of the cell to 0.
- The incorrect read fault $< 1r1/1/0 >$ is detected because the read operation gives wrong result on the shift register output.

The other half of the faults is detected in exactly the same way by the second half of the test.

The application of the aforementioned march test has to be managed through the combination of the synchronous shift-in operation and the asynchronous read operation. Actually, the shift-in operation is more suitable than a normal write operation for implementing the “w0” or “w1” elements of the test, because it is enough to shift in a single ‘0’ or ‘1’ and this value will be written in every cell in succession, as the shifting of more bits takes place. It is then enough to read the value from each cell on the correct clock cycle that it arrives. Thus, to implement the first half of the march test, we follow the steps listed below:

- We set the ENABLE input of the shift register to ‘1’.
- We shift in successively the values ‘0’, ‘1’ and ‘1’, to implement the “w0, w1, w1” elements of the test.
- On the cycle that the second ‘1’ is shifted in position 0 of the shift register, we read this position to implement the “r1” element of the test.
- Every next cycle, we increase the read address by 1 and keep reading the ‘1’ from all positions of the shift register.

Notice that it is not important what values we shift in the register after the three that we explicitly mentioned. We determined these values accordingly, in order to efficiently diagnose the address line faults. This technique, along with the way we checked the rest of the LUT faults, will be illustrated in the next section.

3.3.2.2 Finalizing the LUT test

Except from the memory cell faults, the method has to cover all other faults related to the LUT, which is now configured as a shift register. Specifically, there are address line faults as well as faults of the ENABLE and SHIFTIN inputs of the LUT. Remember that we want to push the diagnosis of these faults as early as possible within the procedure.

In order to efficiently cover the address line faults, we played with the initialization of the shift register. The contents were initialized to ‘0’, except from position 15 (the highest position), which was initialized to ‘1’. Before the shifting of new contents commences, we read position 15 once, this way sensitizing all the ADDR(X) s.a. 0 faults. Of course, a faulty response of ‘0’ at this point does not necessarily mean an address line fault (it can mean a memory fault of cell 15, or a s.a. 0 fault of the LUT output), but what is important at this point is to uncover as many faults as possible. Clarifying between them is possible in the next few cycles.

After sensitizing half the address faults, we focus on the ENABLE s.a. 1 fault. To uncover the existence of this fault, we have to keep ENABLE = ‘0’ and SHIFTIN = ’1’. In this way, if ENABLE is s.a. 1, we will read ‘1’ from position 0, instead of the ‘0’ to which we initialized it. After this operation, we keep ENABLE = ‘0’ for two more cycles, which helps us clarify any possible faulty response during this first part of the test. Subsequently, we switch ENABLE to ‘1’ and start shifting in the appropriate values to commence with the march test. When reading the first ‘1’ from position 0, we also uncover the second half of the address line faults, that is ADDR(X) s.a. 1. In the first cycles of the march test, we also uncover the existence of the ENABLE s.a. 0 fault, since it will cause the state of the shift register to always remain the same. Finally, the faults of SHIFTIN will also be uncovered during this phase.

The above procedure is illustrated in Table 3.5. The state of the shift register is shown for each cycle and the read address noted. Notice that behind the two 1s that we shift in for starting the march test, we occasionally shift in more 1s to help clarify between different address line faults. Also, on cycle 21, the second half of the march test begins. The whole test lasts a maximum of 38 cycles, which is a little more than $2 * 16$.

The insertion of these special few cycles in the beginning of the procedure, somewhat complicates the march test in respect with cell 0 and cell 15. The way to overcome this problem is explained in the next section.

3.3.2.3 Boundary Conditions

We borrow the term *Boundary Conditions* from the study of electromagnetic fields. It is used to describe the special equations that are true on the boundaries of a region that is under the effect of an electromagnetic field. For example, to calculate the intensity of a field on a rectangle, some equations are used, but some extra equations are needed to calculate the intensity on the edges of the rectangle. Likewise, in our case, by reading cell 15 and cell 0 of the LUTs before starting the march test, we might have uncovered some of the memory faults of these cells, a fact that has to be taken into account before starting the actual march test. These special cases are analyzed in this section.

Notice that the actual march test response starts being available on cycle 5, when the “w0, w1, w1” elements of the test have been completed for cell 0 and the “r1” operation

Table 3.5: The shift register state on every cycle of the shift register method.

Read Address	15, 0, 0, 0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
State(15)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
State(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
State(13)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
State(12)	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
State(11)	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
State(10)	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
State(9)	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
State(8)	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0
State(7)	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
State(6)	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0
State(5)	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0
State(4)	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0
State(3)	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0
State(2)	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0
State(1)	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
State(0)	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Cycle	0, 1, 2, 3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Read Address	15	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
State(15)	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
State(14)	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
State(13)	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
State(12)	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
State(11)	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
State(10)	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
State(9)	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
State(8)	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
State(7)	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
State(6)	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
State(5)	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
State(4)	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
State(3)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State(2)	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State(1)	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
State(0)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cycle	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37

starts. But cell 0 has been read also during cycles 1 to 4. Thus, every distinct memory fault might have been already uncovered and they have to be examined independently:

- The state fault which causes constant state '0' is sensitized on cycle 4, when the first '1' is read from cell 0.
- The incorrect read $< 1r1/1/0 >$ and read destructive $< 1r1/0/0 >$ faults are also triggered during cycle 4, since we try to read '1' from cell 0.
- The transition fault $0w1$ also causes the response of cycle 4 to be faulty, since '1' has been written on top of '0' and is subsequently read.
- The write disturb fault $1w1$ appears on cycle 5, as normally scheduled by the march

test.

- The state fault which causes constant state '1', is sensitized on cycle 1, when we try to read '0' from the cell.
- The incorrect read $< 0r0/0/1 >$ and read destructive $< 0r0/1/1 >$ faults are also uncovered on cycle 1 in the same way.
- The transition fault $1w0$ is detected on cycle 21, since we write '1' on top of '0' and subsequently read the cell.
- The write disturb fault $0w0$ is detected on cycle 22, as normally scheduled by the march test.

The case of cell 15 is somewhat simpler, because only one extra read is performed on it in the beginning of the procedure:

- The state fault which causes constant state '0' is sensitized on cycle 0, when we try to read '1' from cell 15.
- The incorrect read $< 1r1/1/0 >$ and read destructive $< 1r1/0/0 >$ faults are also uncovered on cycle 0, when we read '1' from the cell.
- The rest of the faults are detected on the normal schedule of the march test.

With the above analysis available, it was possible to derive the diagnostic trees. The rest of the cells are only read within the framework of the march test, thus being tested exactly in the way described in Section 3.3.2.1.

One final observation about the memory test is the different effect of the incorrect read faults $< 0r0/0/1 >$ and $< 1r1/1/0 >$ compared to all other faults. While every other fault actually corrupts the contents of the memory cell, the incorrect read only affects the output. According to this, the cell is potentially capable to take part in a shift register, as long as it is never read, since it will correctly allow the contents to shift through. This statement only holds in the case of a shift register with fixed input address. The shift register method is able to distinguish the incorrect read fault from all other faults, by performing an extra read operation in the next cell when a memory fault is detected. If the fault propagates from cell to cell, then it is not an incorrect read fault.

3.3.3 Deriving the Diagnostic Trees

After the complete formulation of the method, it was time to proceed to the design of the diagnostic trees for each response analyzer. The shift register method uses 6 independent RAs, one for every primary output of the circuit. The duties and the way that each RA draws a valid conclusion will be summarized in this section.

The FX Response Analyzer: This RA is almost identical to the FX/Y RA of phase 1 of the function generator method, but it observes only output FX. It checks both faults of the MUXFX multiplexer, and the wires FXINA, FXINB, FXOUT and FX

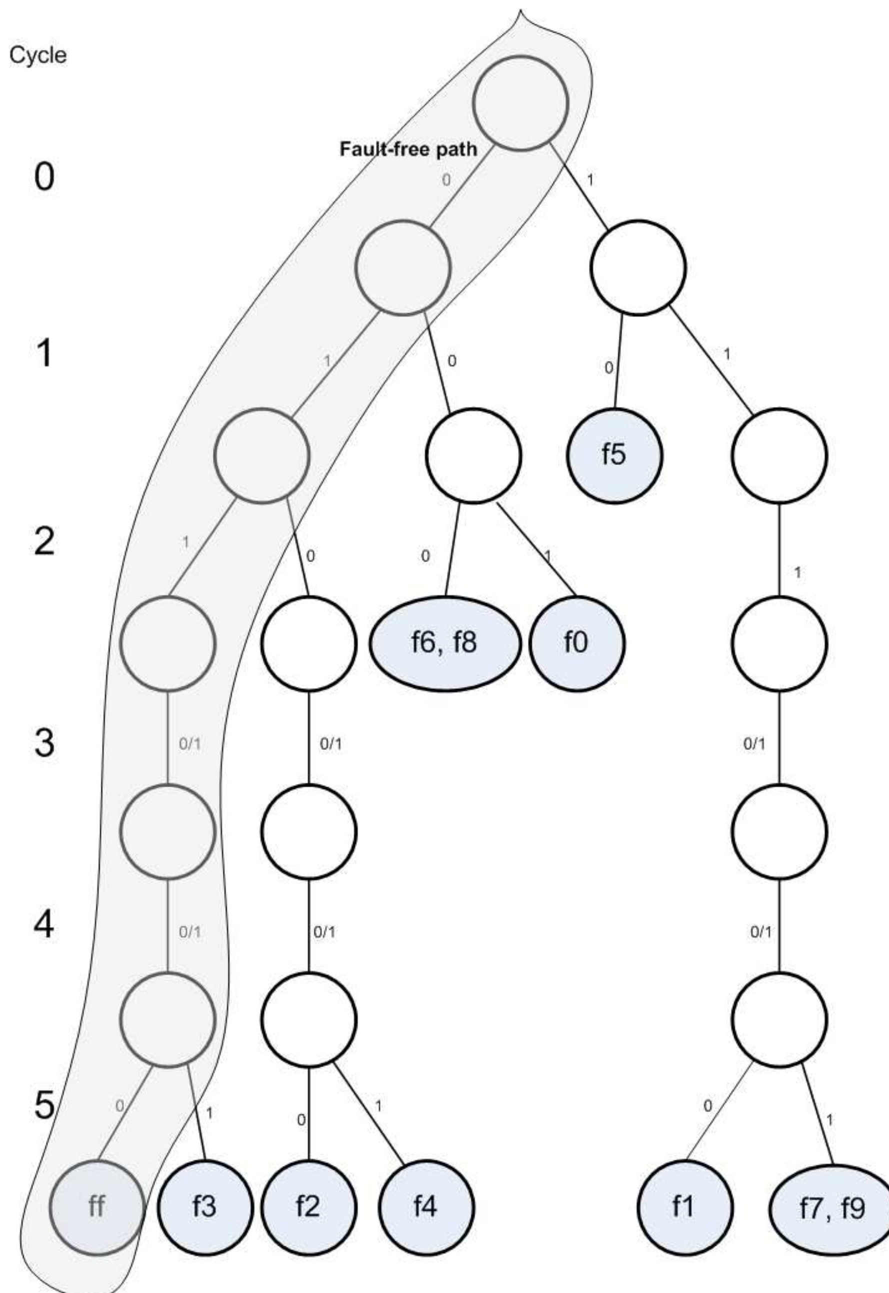

Figure 3.12: The FX diagnostic tree for the shift register method.

in exactly the same way. The only differences are that it can not distinguish between FXOUT and FX faults and it is not responsible for YMUX2 and Y faults. The resulting diagnostic tree is shown in Figure 3.12.

The Y Response Analyzer: This RA's main duty is to check all features of the G-LUT. Additionally, it has to check the MUXCTRLY s.a. 0 fault of the ROUTING

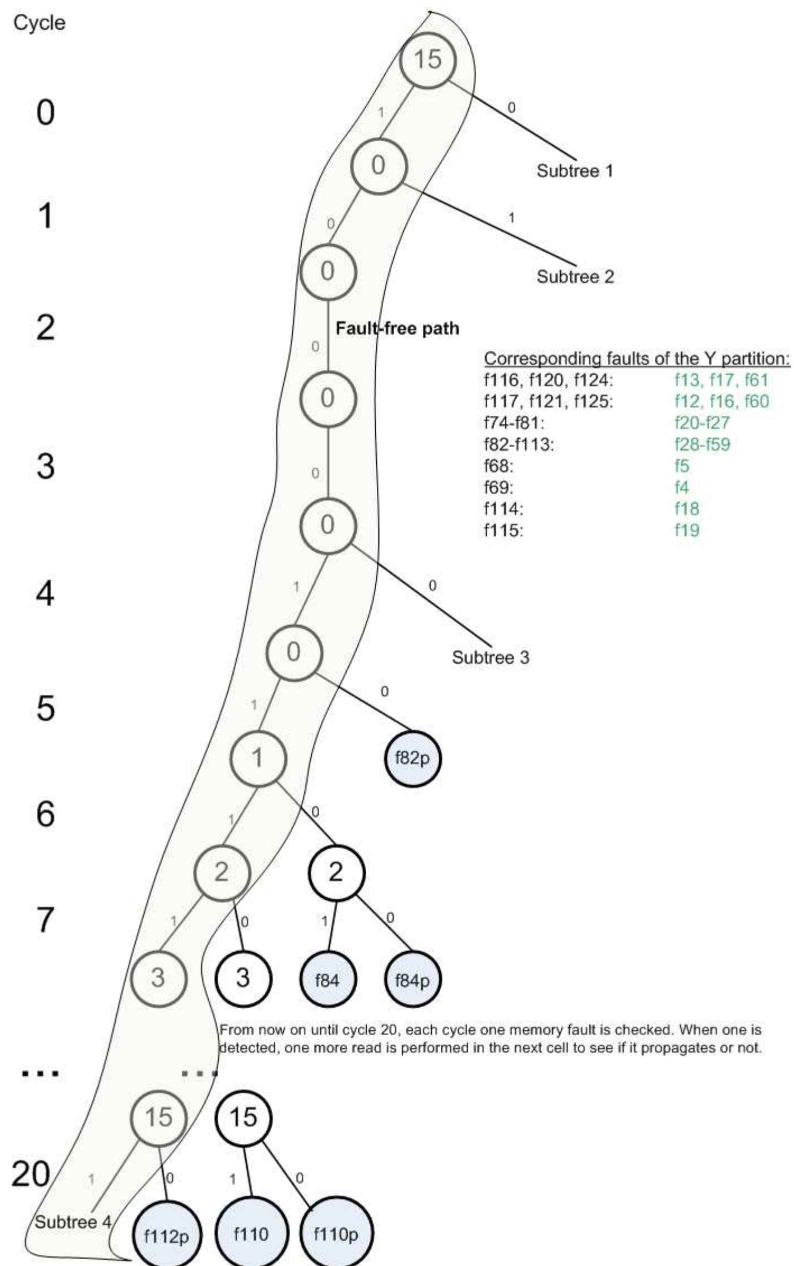

Figure 3.13: The Y and X diagnostic tree for the shift register method. The tree for the X partition is shown, while the corresponding faults of the Y partition, listed in green color, are diagnosed with the exactly complimentary responses. Also see Figures 3.14 to 3.17 for subtrees 1 to 4.

Y multiplexer and the faults of wires GD, YMUX2 and Y. Each category of faults is diagnosed as follows, based on the method characteristics that have been explained in the previous sections:

- The memory faults are detected by the march test, one cell every cycle. Upon

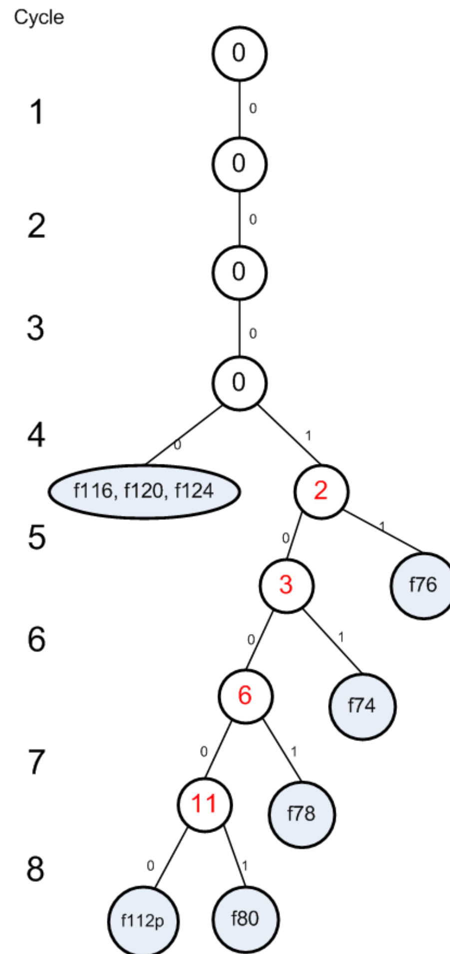

Figure 3.14: Subtree 1 of the Y and X diagnostic tree for the shift register method.

diagnosis of a memory fault, one more read operation is performed in the next cell, to decide whether or not the fault propagates.

- Address line faults are detected on cycle 0 (the ADDR(X) s.a. 0 faults) and on cycle 4 (the ADDR(X) s.a. 1 faults). The specific fault is clarified in the following cycles, by shifting in the appropriate data and reading the appropriate positions.

- The ENABLE s.a. 0 fault is diagnosed because the state of the shift register remains always the same, while the ENABLE s.a. 1 is diagnosed in the first few cycles, if the state of the register changes although it is not supposed to.

- The SHIFTIN faults are diagnosed because the output is fixed to a steady value after ENABLE is set to '1' and the shifting operation starts.

- The faults of all wires appear as a steady value on the output from the beginning of the process.

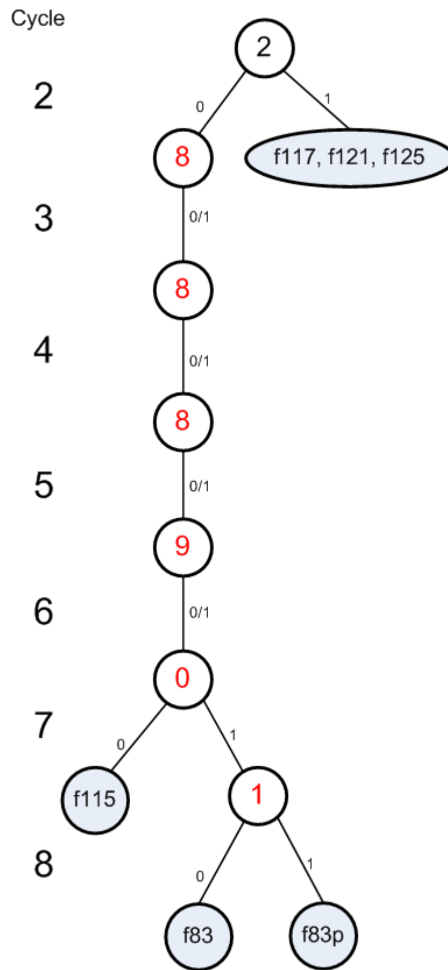

Figure 3.15: Subtree 2 of the Y and X diagnostic tree for the shift register method.

The resulting diagnostic tree is shown in Figures 3.13 to 3.17. This is the most complicated tree we had to build.

The X Response Analyzer: This RA has duties exactly identical to the Y RA: Checking every feature of the F-LUT, the MUXCTRLX s.a. 0 fault of the ROUTING X multiplexer and the faults of wires FD, XMUX2 and X. Each category of these faults is checked in exactly the same way as in the Y RA, thus the X RA uses the same diagnostic tree of Figures 3.13 to 3.17.

The YQ Response Analyzer: This RA is responsible for checking faults of the G REGISTER storage element, the DYCTRL s.a. 1 fault of the DYMUX multiplexer and the faults of wires DY, FQ and YQ. Each of these faults is diagnosed as follows:

- The faults of G REGISTER are diagnosed on the first occurrence of each of the four possible transitions of the flip-flop.

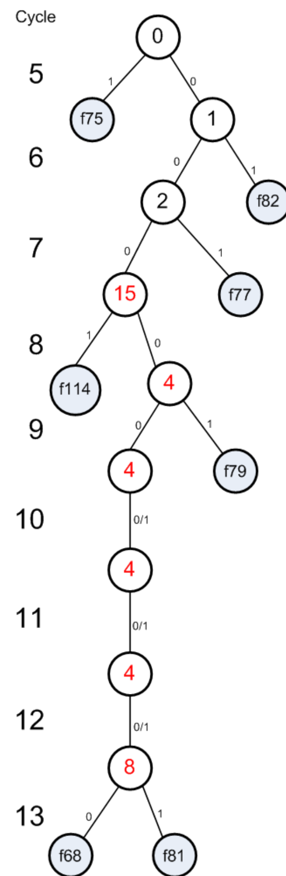

Figure 3.16: Subtree 3 of the Y and X diagnostic tree for the shift register method.

- DYCTRL s.a. 1 is checked on the cycles that FXCTRL/BY is different than DY.
- The faults of all wires are observed directly on YQ, as a steady value.

The resulting diagnostic tree is shown in Figure 3.18.

The XQ Response Analyzer: This RA has duties exactly identical to the YQ RA: It has to check faults of the F REGISTER storage element, the DXCTRL s.a. 1 fault of the DXMUX multiplexer and the faults of wires DX, FQ and XQ. All the faults are diagnosed in exactly the same way as in the YQ RA, thus the XQ RA uses the same diagnostic tree presented in Figure 3.18.

The F5 Response Analyzer: This RA is responsible to check both faults of the MUXF5 multiplexer and the faults of wires F5IN1, F5IN2, F5OUT and F5. It achieves this as follows:

- The inputs of MUXF5 are always complementary, thus checking this multiplexer is trivial.

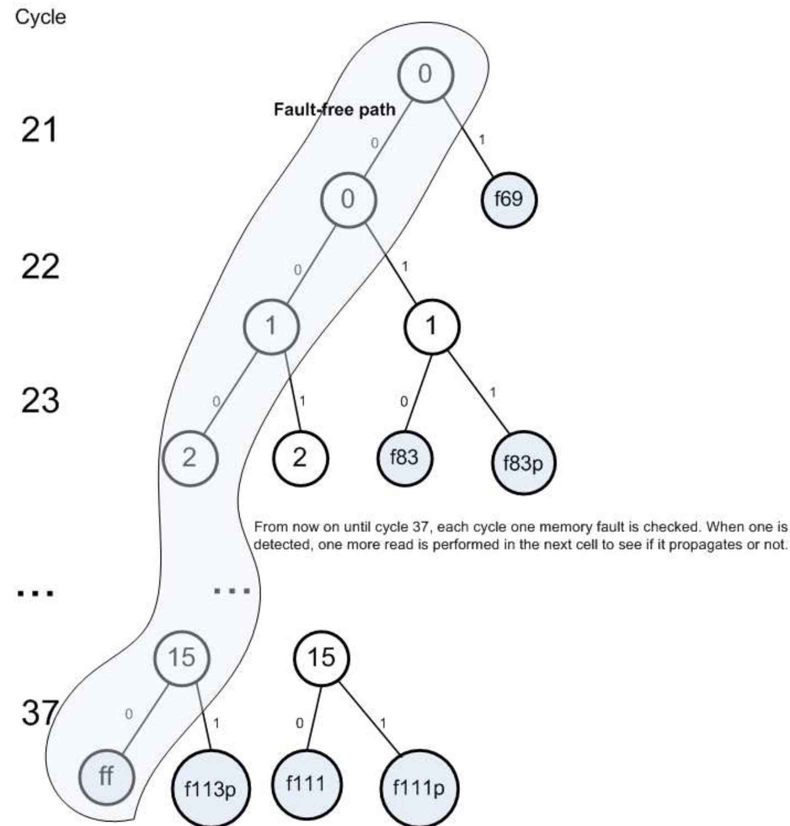

Figure 3.17: Subtree 4 of the Y and X diagnostic tree for the shift register method.

- Faults of F5IN1 and F5IN2 are visible on the output only during cycles that the faulty wire is chosen.
- Faults of F5OUT and F5 are detected as a steady value of the output.

The resulting diagnostic tree is shown in 3.19. This RA also helps in a special case: If it detects a fault on F5IN1 and the Y RA has detected a fault on one of the wires GD, YMUX2 and Y, then the actual fault is surely on GD. This is the only case that results of different RAs are combined to make a final diagnosis. The same is true for faults of FD and F5IN2.

This concludes our complete description of both developed methods. In the final portion of this chapter, we will explain how diagnosis results are used to characterize the slice.

3.4 Characterization and Graceful Degradation

As was mentioned many times throughout this thesis, the purpose of fault diagnosis is to eventually characterize the faulty FPGA slice, so that it can be reused in a suitable

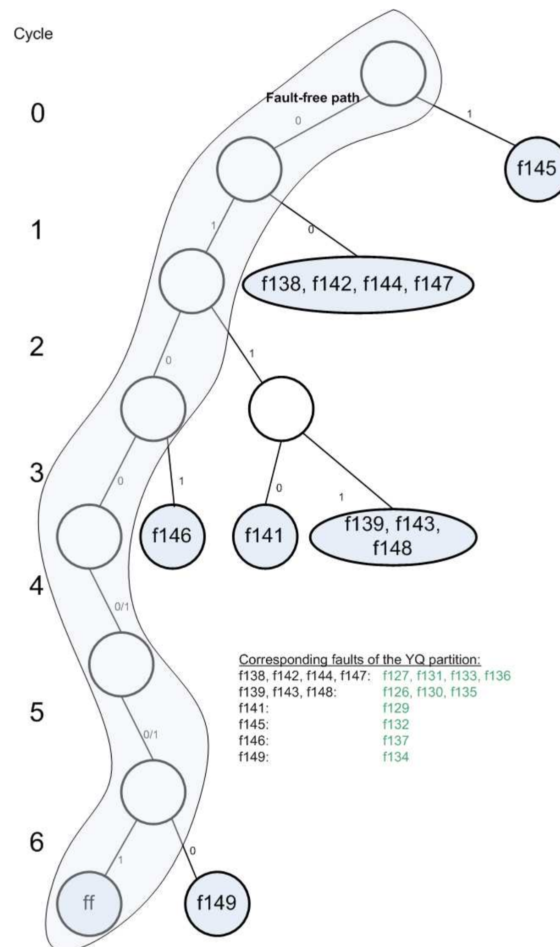

Figure 3.18: The YQ and XQ diagnostic tree for the shift register method. The tree for partition XQ is shown, while the corresponding faults of partition YQ, listed in green color, are diagnosed with the exactly complimentary responses.

mode of operation. In this section we will explain how this is done.

In previous works regarding graceful degradation, a list of modes of operation is defined. For example, a mode of operation is to configure the slice as a function generator, without the use of the storage elements (in this case, it is purely combinational). According to the fault that is diagnosed, each of these modes of operation is marked as suitable or not for the basic block.

In this work, we opted for a slightly different approach. We defined a set of modes of degradation, each of which describes which function(s) the basic block can **not** perform. In our opinion, in this way matching can be performed relatively easily, by inspecting one, or a few bits of the intended configuration bit stream. All modes of degradation and the faults that result to each of them according to both methods are listed in Table 3.6. As will be explained in the next chapter, the characterization is performed independently from the diagnosis. If someone decides that another set of modes of degradation is needed, or another manner of characterization, he can still use the result

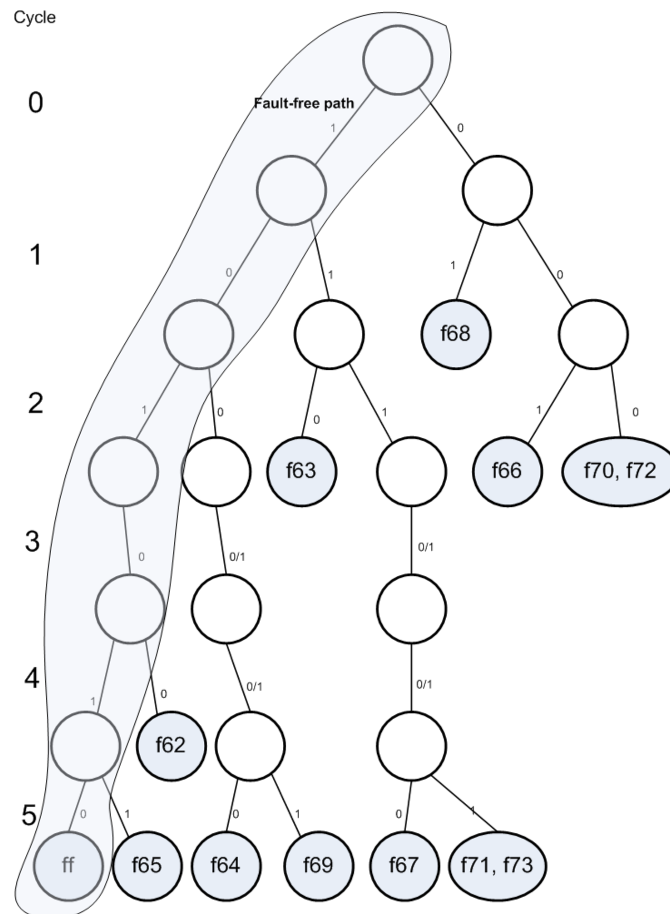

Figure 3.19: The F5 diagnostic tree for the shift register method.

of our diagnosis methods to do so.

3.5 Summary

In this chapter we approached the core concepts of this thesis. We analyzed the two methods that were developed in the process, by explaining in detail how each of them was built step by step.

Our first method was based on the function generator configuration of the slice. This is the most frequent configuration, both in testing methods and during the normal operation of the FPGA. The resulting diagnostic trees were relatively simple and the diagnostic resolution very good. The lack of access to the configuration bits unfortunately meant that at least 2 phases were needed to complete the test and even more if all memory faults are expected to be covered. These drawbacks guided us to the development of our second method.

Our second method revolved around a shift register configuration for the slice. By doing that, we gained access to the configuration bits of the LUTs, which allowed us to

Table 3.6: Modes of graceful degradation.

MODE: CODE	METHOD 1 FAULTS	METHOD 2 FAULTS	DESCRIPTION
M0(a): 0000000	Fault-free	-	The CUT is fault-free in respect with the faults that are checked with the function generator method. Thus, it can support all functions that are tested by this method.
M0(b): 0000000	-	Fault-free	The CUT is fault-free in respect with the faults that are checked with the shift register method. Thus, it can support all functions that are tested by this method.
M1: 0100100	f0, f1, f2, f3, f4, f5, f6, f7	f0, f1, f2, f3, f4, f5, f6, f7, f8, f9	MUXFX not functional, output FX not usable. Signal YMUX1 not usable, hence ROUTING Y multiplexer usable only with MUXCTRLY = '1'.
M2: 0100101	f8, f9	-	Output FX not usable.
M3: 0100110	f10, f11, f15	-	ROUTING Y multiplexer usable only with MUXCTRLY = '1'.
M4: 0100011	f12, f13, f14	-	ROUTING Y multiplexer usable only with MUXCTRLY = '0'.
M5: 0100111	f16, f17	f12, f13, f16, f17	Output Y not usable. Y flip-flop only usable directly through input BY (DYCTRL = '1').
M6: $001M_3M_2M_10$	G-LUT 8 address faults	G-LUT 8 address faults	G-LUT only usable as a 3-variable function generator, excluding the faulty address line. 8 different modes are encoded. M_3M_2 encode the faulty line and M_1 denotes s.a. 0 or s.a. 1 fault.
M7: $1M_50M_3M_2M_1M_0$	G-LUT 32 memory faults	G-LUT 32 memory faults	G-LUT entry needs to be configured to the faulty value. 32 different modes are encoded. $M_5M_3M_2M_1$ encode the faulty memory cell and M_0 denotes the forced configuration value.
M8: 0100000	f60, f61	f60, f61	G-LUT not usable. MUXCTRLY has to be configured to '0'.
M9: 0100001	f64, f65, f66, f67, f68, f69, f70, f71	f64, f65, f66, f67, f68, f69, f70, f71, f72, f73	MUXF5 not usable. Output F5 not usable. Output X can only be driven by the F-LUT (MUXCTRLX = '1').
M10: 0100010	f72, f73	-	Output F5 not usable.
M11: $001M_3M_2M_11$	F-LUT 8 address faults	F-LUT 8 address faults	F-LUT only usable as a 3-variable function generator, excluding the faulty address line. 8 different modes are encoded. M_3M_2 encode the faulty line and M_1 denotes s.a. 0 or s.a. 1 fault.
M12: $1M_51M_3M_2M_1M_0$	F-LUT 32 memory faults	F-LUT 32 memory faults	F-LUT entry needs to be configured to the faulty value. 32 different modes are encoded. $M_5M_3M_2M_1$ encode the faulty memory cell and M_0 denotes the forced configuration value.
M13: 0110000	f116, f117	f116, f117	F-LUT not usable. Outputs F5 and X not usable. X flip-flop only usable directly (DXCTRL = '1').
M14: 0110011	f118, f119, f123	-	ROUTING MUX X can only propagate the F-LUT output to output X (MUXCTRLX = '1').
M15: 0110001	f120, f121, f122	-	ROUTING MUX X can only propagate MUXF5 output to output X (MUXCTRLX = '0').
M16: 0110010	f124, f125	f120, f121, f124, f125	Output X not usable. X flip-flop only usable directly, through input BX (DXCTRL = '1').
M17: 0101010	f126, f127, f129	f129	Y flip-flop only usable directly, through input BY (DYCTRL = '1').
M18: 0101000	f128	-	Y flip-flop only usable through logic output Y (DYCTRL = '0').
M19: 0101011	f130, f131, f132, f133, f134, f135, f136, f137	f126, f127, f130, f131, f132, f133, f134, f135, f136, f137	Y flip-flop not usable. YQ output not usable.
M20: 0111000	f138, f139, f141	f141	X flip-flop only usable directly, through input BX (DXCTRL = '1').
M21: 0111010	f140	-	X flip-flop only usable through logic output X (DXCTRL = '0').
M22: 0111001	f142, f143, f144, f145, f146, f147, f148, f149	f138, f139, f142, f143, f144, f145, f146, f147, f148, f149	X flip-flop not usable. XQ output not usable.
M23: 0000001	-	f18, f19	G-LUT unable to be used as shift register.
M24: 0000010	-	f114, f115	F-LUT unable to be used as shift register.
M25: 0000011	-	f62, f63	Both LUTs unable to be used as shift registers.
M26: N/A	-	-	Slice not usable at all.

produce very good memory tests in one phase and in a number of cycles almost identical to that of the first method. The limitation of only one phase meant that we couldn't get as good diagnostic resolution on the wire faults as the first method. The importance of this drawback depends on the characterization scheme that is chosen. Also, this method can draw a definite conclusion on the ability of the slice to be used as a function generator, whereas the first method can not do the same for the shift register mode.

In the following chapter we will explain various implementation details through block and circuit diagrams, before we proceed to actually evaluating and comparing the 2 methods in Chapter 5.

4 Hardware Implementation

Up to now in this report we tried to as adequately as possible set the research framework, define the problem at hand and describe our solution to it. Our description of the solution was completely independent of technology and implementation details. One can decide to apply it in a completely different manner than we did in this thesis. For example, we opted for an on-chip implementation (in other words, a BIST approach), but this is not imperative, just a better choice in our point of view. That said, the time has come to explain how we transitioned from the diagnosis method design (the process through which we derived the diagnostic trees) to actual hardware implementation (a working system that produces the correct results). That is the purpose of this chapter, which represents the engineering part of our work, while Chapter 3 was dedicated to the research aspect.

This chapter is organized as follows: We will begin by presenting a high-level organization of the system in Section 4.1 and go on to describe the role and internal structure of each main component in Section 4.2. In Section 4.3 we will approach some miscellaneous topics that required our attention throughout the development and in Section 4.4, we will summarize the chapter and prepare the ground for the next phase, which is the method evaluation.

4.1 System Organization

As we already pointed out a few times, the core concept of implementing fault diagnosis in our approach is that of a diagnostic tree. The basic advantage of this approach is that it can support an adaptive diagnosis strategy, by identifying the set of suspect faults after each fragment of the CUT response becomes known and making a decision as soon as possible. In some cases, it is even profitable to modify the upcoming test vectors in order to target the suspect faults specifically and reach the conclusion even sooner. This means that the response analyzer should give some feedback to the vector generator, in order to determine the subsequent steps. This was illustrated in Figure 3.1.

In this section we provide a more detailed, but still high-level description of the system. In Figure 4.1 the general block diagram of the system is given. This is a general version, valid for both methods that were described in Chapter 3. The system consists of a set of response analyzers, each being responsible for a partition of the circuit under test, as was explained adequately in Section 3.2.1 and a system controller, coordinating the operation of the RAs. The duties and internal structure of the RAs and the controller will be explained in the next section.

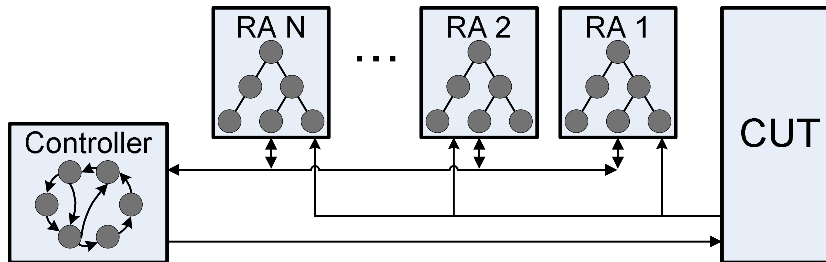

Figure 4.1: Tester Block Diagram.

4.2 Function and Implementation of modules

As was established in the previous section, there are two core components in our system: The response analyzers and the system controller. In this section, we will go on to list the duties and illustrate the implementation of each of them separately.

4.2.1 Response Analyzers

From a computational standpoint, the response analyzers are the core of our testers. They have to assess the response of the CUT and draw conclusions about the presence or absence of faults. The complete list of tasks appointed to the RAs is the following:

- Collecting the response of the respective CUT partition.
- Comparing the response to the expected one and deciding, on every step of the process, if the response of the partition is fault-free, or a fault is present in it.
- Using a window of the partition's response (1 to 4 bits, depending on the specific RA) to adapt the diagnosis strategy by choosing an alternative vector to be applied to the CUT.
- Ultimately producing a diagnosis result according to the diagnostic tree. This result can either be that the partition is fault-free, or the specific fault (or set of few faults) that the partition suffers from.
- Keeping the controller updated about its state: That includes notifying the controller when it is done and the result it produces is valid, but also when it detects the presence of *some* fault in the partition, before actually diagnosing which fault exactly it is. In the latter case, the controller will be able to notify the rest of the RAs that the fault is detected and there is no need for them to continue the process.

The general implementation of the response analyzer is shown in Figure 4.2. Note that not all features are present in all RAs. For example, the RAs that correspond to passive CUT partitions (partitions without any primary input), never produce alternative test vectors. On every cycle, one bit of CUT response is added to the response window, and the oldest bit is thrown away. The response window register, when more than one

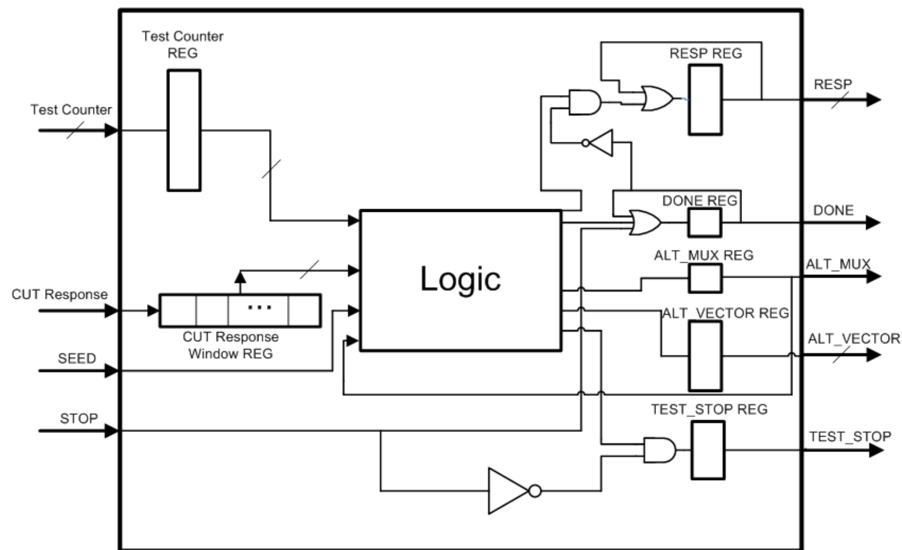

Figure 4.2: Response Analyzer Implementation.

bit, is realized as a small serial-in parallel-out shift register. Subsequently, the value of the test counter (which denotes the step of the process that is currently commencing) together with the response window, are input to a combinational logic block that takes all decisions listed above. The results of this logic module are stored in the output registers and reach the system controller within the next clock cycle. To complete the description of the RA, the following remarks have to be made, referring to Figure 4.2:

- The TEST_STOP output is raised as soon as the very first faulty response of the CUT partition is observed. This is enough to conclude that the fault lies within this partition and that the rest of the RAs can stop. Notice that a RA that has been ordered to stop, cannot order others to stop, because that would include the RA that issued the original STOP command. Also notice that the STOP input is fed forward to this combinational stage, without being registered in the RA input. It is important that RAs are ordered to stop as soon as possible, because there are a few faults that eventually cause faulty responses to other partitions. A good example of such an instance is a fault in the Y (or X) partition causing a faulty response to be observed also on the YQ (or XQ) partition, since it will probably propagate through the flip-flop.

- The ALT_VECTOR output carries potential alternative vectors to the system controller, according to faulty responses observed here. The ALT_MUX output actually notifies the controller that it should use the alternative vectors instead of the predefined ones. The present value of ALT_MUX is used by the combinational logic part, because it is an easy way to tell if the process continues on the fault-free path (as seen on the diagnostic trees of Chapter 3) or not.

- Once the RA decides that it is done, it raises the DONE output. DONE will always stay raised after that time. Except from reaching a diagnosis conclusion for the

partition in question, a RA can also be DONE by being told to STOP.

- The fault-free response for every RA is a vector of 0s. In this way, when the RA is done, the response can be locked to the valid value by the simple scheme shown in Figure 4.2. As soon as DONE is raised, RESP cannot change its value anymore.

As it was already mentioned in this section, all outputs of each RA go to the system controller. The way that the controller processes them is discussed in the following section.

4.2.2 System Controllers

While the RAs are the computational core of the system, the system controllers assume the lead role in organizational matters. The list of tasks that each controller is responsible for is as follows:

- Collecting all outputs of various RAs.
- Keeping a test counter that indicates the step that the diagnosis process is presently at and feeding that value (or part of it, when not all is necessary) to the RAs.
- Feeding the CUT with test vectors. The controller also has to substitute the predefined test vectors with alternative test vectors, when some RA dictates a change of diagnosis strategy.
- Ordering RAs to stop when another RA detected the presence of a fault in its partition.
- Detecting that the diagnosis process or phase is done, when all RAs raise their DONE signals.
- Combining all RESP outputs of the RAs, to calculate a mode of degradation, thus characterizing the CUT.
- (Only for phase 1 of the function generator method): Deciding whether or not a second phase is needed. If it is, check for possible partial diagnosis and produce a seed to be passed to phase 2.
- (Only for phase 2 of the function generator method:) Receiving the seed, if any, from phase 1 and determining the duties of the various RAs according to it.

The hardware implementation of the system controller is shown in 4.3. The following remarks complete the description of the system controller implementation:

- The predefined test vectors are multiplexed with the alternative test vectors with the help of the VECTOR_MUX input coming from the RAs. This happens for each response analyzer that produces alternative vectors. Note that the predefined test vectors are stored in advance, except from the address vectors, that are derived from the value of the test counter as follows:

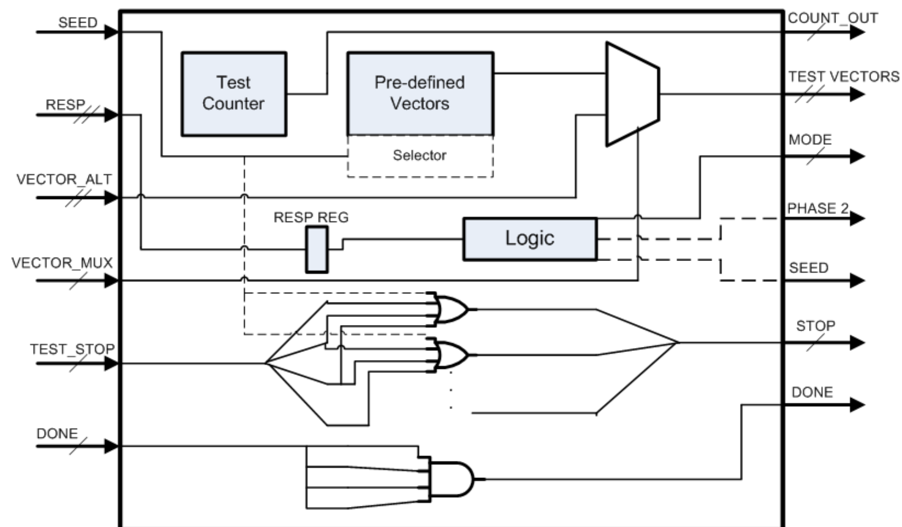

Figure 4.3: System Controller Implementation.

– During phase 1 of the function generator method, the standard address vectors are the value of the test counter on every cycle, since the counter counts from 0 to 15 and on every cycle we read the corresponding LUT bit.

– During phase 2 of the function generator method, the standard address vectors are the value of the test counter from cycle 0 to cycle 15. The 4 remaining vectors for cycles 16 and 17 (2 for each LUT) are stored like any other vector.

– Regarding the shift register method, we observed that the address vector starts from value 15, then becomes 0 and after a few cycles it starts incrementing by 1 on every cycle. When it reaches 15, it returns to 0 and after 1 more cycle it starts counting up to 15 again. Based on this behavior, we addressed the LUTs with a 4-bit cyclic counter, initialized to 15, with a count enable signal. When the address vector remains 0 for some cycles, the count enable signal has to be kept to '0'. In this way, we had to store the values of the count enable signal instead of the 4-bit values of the address vectors.

- The STOP signal for each RA is produced by an OR operation between the TEST_STOP signals of all other RAs.

- The PHASE/PROCESS_DONE signal is produced by an AND operation between all DONE signals coming from the RAs.

- The characterization stage does not interfere at all with any other calculation and is easily substitutable by any other characterization scheme. In our approach, we calculate one value for every RA. If the RA's partition was fault-free, the resulting value is a vector of 0s. The eventual value for MODE is the bitwise OR between all individual values corresponding to a different RA each. The encoding that was chosen for the modes resulted in a very low complexity logic module for calculating them. When phase 1 of the function generator method terminates with partial

diagnosis, a partial, temporary mode is also produced, with only 1 or 2 bits being under dispute, to be resolved during phase 2. When phase 1 terminates without partial diagnosis, the value of MODE is also kept to a vector of 0s.

- In the presence of a seed from phase 1, the controller of phase 2 of the function generator method, uses that seed to order the unneeded RAs to stop directly and also to adjust the test vectors for the needed RAs accordingly.

This concludes the implementation description of the system in general. In the next section, we focus on a few more implementation topics that were significant in realizing the system.

4.3 Other Implementation Details

There were a few points of the implementation process that are significant and require some attention, but do not fall into the categories of the previous sections. In this section we will cover these last details, in order to give a complete picture of the system implementation. In Section 4.3.1 we will describe how we wrapped identical RAs in order to reduce hardware complexity, in Section 4.3.2 we will explain the choice of fault and mode encoding, while in Section 4.3.3, we will illustrate the system pipeline, according to what has been explained so far.

4.3.1 Wrapping Identical Response Analyzers

Going back to Sections 3.2.3 and 3.3.3, we remind the reader that there were a few pairs of RAs with completely symmetrical sets of faults to check and an one-to-one correspondence of faults that had exactly complementary faulty responses. In other words, for RAs A and B, for every fault in the list of RA A there was exactly one fault in the list of RA B with exactly complementary faulty response and vice versa. Also, their fault-free responses were exactly identical. These RAs actually shared the same diagnostic trees.

This fact, combined with our assumption for a single fault per slice, incited us to use the symmetry in order to reduce hardware complexity. Since both RAs share the same diagnostic tree, there is no need for both of them to be realized. One alone can resolve the tree, with the help of a small wrapper that has the following duties:

- Check if the responses of both partitions that are tested are fault-free. This can be done with an XOR operation between the two responses, since if a fault occurs, it will occur only in one of them, causing them to be identical instead of complementary.

- When a response is faulty, determine the partition that produced it and notify the single RA about it.

- Always feed to the RA the appropriate response in order to make the correct diagnosis. Essentially, the single RA always checks one of the two partitions in question, say partition A. When partition A fails, the faulty response has to be

given to the RA, whereas when partition B fails, the response of the corresponding failure of partition A has to be produced.

The three instances of pairs of symmetrical RAs that were wrapped and combined in one, are explained below. The reader is advised to return to Sections 3.2.3 and 3.3.3 if he doesn't remember the symmetry of duties between the two members of each pair.

The F5 and X RAs during phase 1 of the function generator method: The wrapper's inputs are the responses of the two partitions and the address test vector. The wrapper consists of:

- A generator of the fault-free value of F5 for the given test vector.
- A multiplexer choosing between the actual F5 response and its complement.
- A small combinational logic block that checks the 2 responses and the fault-free value of F5 and chooses accordingly which input of the multiplexer to propagate to the RA.

The block diagram of the wrapper can be seen in Figure 4.4.

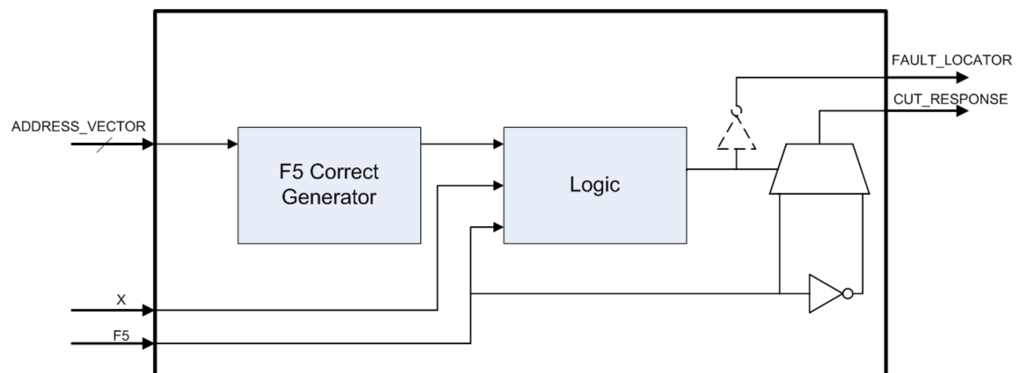

Figure 4.4: The RA Wrapper

The combinational logic block drives the multiplexer according to Table 4.1. For each combination of F5, X and F5_CORRECT, the corresponding choice is explained in the last column. The control signal of the multiplexer is also the FAULT_LOCATOR output of the wrapper, denoting the partition of the fault. The complement of this value can be used instead, if it is more convenient for the fault encoding.

The Y and X RAs of the shift register method: This RA is special, because on top of the other duties it takes care of a special case. The Y and X partitions of the shift register method contain the faults of wires GENABLE and FENABLE respectively, which have complementary faulty responses and can be detected in the usual way. However, if the origin wire ENABLE is faulty, it is going to affect both X and Y outputs, causing the XORing between them to signify the absence of a fault.

The way we chose to deal with this problem was to transfer responsibility for it to the F5 RA. Indeed, the ENABLE faults will affect both F5IN1 and F5IN2. For verification about the F5 RA detecting this fault see the leaf nodes f62 and f63 in Figure 3.19.

Table 4.1: F5/X Wrapper logic truth table. The meaning of the occurrence of every input combination is explained in the last column.

F5	X	F5_CORRECT	LOCATOR	TO RA	EXPLANATION
0	0	0	0	$\overline{F5}$	F5 is correct, X is faulty. Send $\overline{F5}$ to RA, to represent the corresponding fault of X.
0	0	1	1	F5	X is correct, F5 is faulty. Send F5 to RA, because it is faulty.
0	1	0	0	F5	Both F5 and X are correct. Send F5 normally.
0	1	1	DON'T CARE		Will never happen.
1	0	0	DON'T CARE		Will never happen.
1	0	1	0	F5	Both F5 and X are correct. Send F5 normally.
1	1	0	1	F5	X is correct, F5 is faulty. Send F5 to RA, because it is faulty.
1	1	1	0	$\overline{F5}$	F5 is correct, X is faulty. Send $\overline{F5}$ to RA, to represent the corresponding fault of X.

This solution also meant that the truth table for the FAULT_LOCATOR logic in this particular wrapper has to be slightly changed to take care of this special case. The revised truth table is shown in Table 4.2. The wrapper is otherwise identical to the rest, with X and Y being in place of F5 and X respectively, and the block labeled F5_CORRECT GENERATOR now being a X_CORRECT GENERATOR

The YQ and XQ RAs of the shift register method: This wrapper is identical to the first one, with XQ in the place of F5 and YQ in the place of X. Of course, the block labeled F5_CORRECT GENERATOR in Figure 4.1 is now a XQ_CORRECT GENERATOR.

4.3.2 Fault and Mode Encoding

Another critical aspect for the system performance is the manner in which every response analyzer encodes the diagnosis result and also the scheme with which the modes of degradation are encoded. These decisions affect the complexity of the logic that decides on the output of the RAs, as well as the conversion of the diagnosis result to mode of degradation.

A good example of encoding choice is the F5/X RA for phase 1 of the function generator method. In Table 4.3 we show the diagnosis result response, along with the resulting mode of degradation for each outcome. According to that correspondence, the logic that converts diagnosis response to mode of degradation is very simple. This is illustrated in the code fragment of Figure 4.5. We see that most of the bits of the mode can be calculated with one LUT, which is the minimum logic complexity for FPGA

Table 4.2: X/Y Wrapper logic truth table. The meaning of the occurrence of every input combination is explained in the last column.

X	Y	X_CORRECT	LOCATOR	TO RA	EXPLANATION
0	0	0	0	$\overline{X}$	X is correct, Y is faulty. Send $\overline{X}$ to RA, to represent the corresponding fault of Y.
0	0	1	1	X	Y is correct, X is faulty. Send X to RA, because it is faulty.
0	1	0	0	X	Both X and Y are correct. Send X normally.
0	1	1	0	$\overline{X}$	ENABLE fault present. Send $\overline{X}$ to RA to absorb the fault, since another RA will find it.
1	0	0	0	$\overline{X}$	ENABLE fault present. Send $\overline{X}$ to RA to absorb the fault, since another RA will find it.
1	0	1	0	X	Both X and Y are correct. Send X normally.
1	1	0	1	X	Y is correct, X is faulty. Send X to RA, because it is faulty.
1	1	1	0	$\overline{X}$	X is correct, Y is faulty. Send $\overline{X}$ to RA, to represent the corresponding fault of Y.

realizations.

The same simplicity is preserved for the encoding of every RA's results.

4.3.3 System pipeline

In the block diagrams presented in Section 4.2, the places where data is stored are clearly visible. These registers divide the whole circuit in a number of combinational paths. The integrated system pipeline is shown in Figure 4.6.

Notice the importance of feeding forward the STOP signals, so we can instruct RAs to stop in time, in order for them not to corrupt the final response by erroneously detecting a fault that has propagated in their partition from a site outside of it. The critical path of this pipeline in both methods is the logic of the RAs, producing the diagnosis result that is subsequently converted to a mode of degradation.

4.4 Summary

In this chapter, we focused on the engineering qualities of this work. We described the process of going from a generic method description that is shown to potentially work in Chapter 3, to real, working circuits. We first described the system organization in a

Table 4.3: Fault and mode encoding for the F5/X Response Analyzer of the first phase of the function generator method.

Fault	Fault Code	Resulting Mode and Code
Memory faults	$L1A_3A_2A_1A_0$	M7/M12: $1A_3LA_2A_1A_0V$ (A_3 to A_0 denote the faulty cell, L denotes the LUT containing the faulty cell and V denotes the value of the memory fault)
Address faults	$L01L_1L_0V$	M6/M11: $001L_1L_0VL$ (L_1L_0 denotes the faulty address line, V denotes the faulty value and L denotes the LUT that has the defective address line.)
f69/f122	$L00001$	M9/M15: $01L0001$ (L denotes the partition that contains the fault)
{f60, f64, f70, f72} / {f116, f120, f124}	$L00100$	M8/M13: $01L0000$ (L denotes the partition that contains the fault)
{f61, f65, f71, f73} / {f117, f121, f125}	$L00100$	M8/M13: $01L0000$ (L denotes the partition that contains the fault)
fault-free case	000000	M0(a): 0000000

```
F5X_MODE(6) <= I_F5X_RESPONSE(4); -- 0 LUTs
F5X_MODE(5) <= (I_F5X_RESPONSE(4) XNOR I_F5X_RESPONSE(3)) AND
(I_F5X_RESPONSE(4) OR I_F5X_RESPONSE(2) OR I_F5X_RESPONSE(1) OR I_F5X_RESPONSE(0));
-- 2 LUTs
F5X_MODE(4) <= ‘1’WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "01") ELSE
I_F5X_RESPONSE(5); -- 1 LUT
F5X_MODE(3) <= ‘0’WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "00") ELSE
I_F5X_RESPONSE(2); -- 1 LUT
F5X_MODE(2) <= ‘0’WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "00") ELSE
I_F5X_RESPONSE(1); -- 1 LUT
F5X_MODE(1) <= ‘0’WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "00") ELSE
I_F5X_RESPONSE(0); -- 1 LUT
F5X_MODE(0) <= I_F5X_RESPONSE(0) WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "00")
ELSE
I_F5X_RESPONSE(5) WHEN (I_F5X_RESPONSE(4 DOWNTO 3) = "01") ELSE
F_LUT_CORRECT XNOR I_F5X_RESPONSE(5); -- Complement of LUT expected value
-- 2 LUTs
```

Figure 4.5: Simple conversion of diagnosis result to mode of degradation according to Table 4.3

high level and subsequently focused on each component and went into as much detail as necessary to give the reader a complete picture of how we implemented the testers. We tried to focus on some of the problems we faced during the implementation phase and

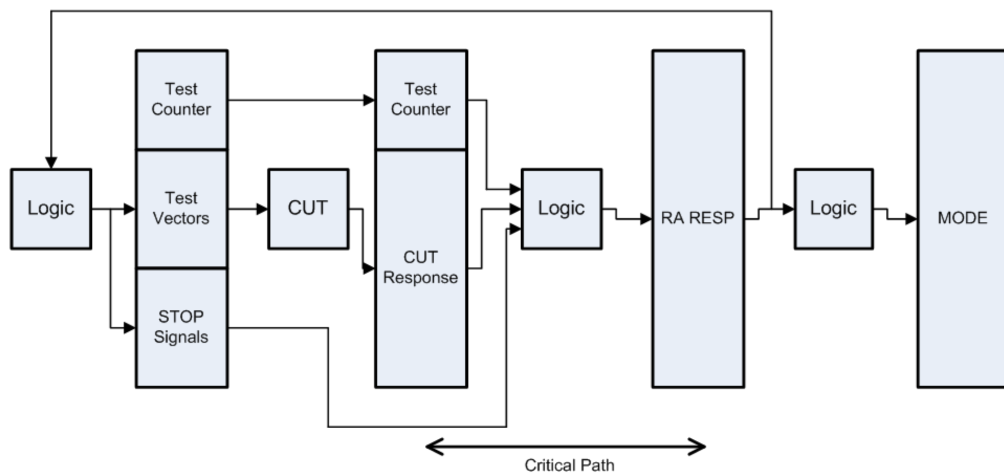

Figure 4.6: The System Pipeline

the solutions we devised to get over them.

In the next chapter, the two methods will be evaluated, mainly compared to each other. We will use both qualitative (i.e. scalability) and quantitative (i.e. performance) criteria in order to draw conclusions about their usefulness in the defined framework.

5 Evaluation

Having completed the detailed description of our two methods, in terms of both design and implementation, it is time to proceed to the evaluation of the work. As will be shortly explained, we went on to create a prototype of our system, working on an FPGA board that is available in the Computer Engineering laboratory. Through this prototype, our realized tester is applied to a real Virtex-II Pro slice. This prototype was a major goal of this thesis from the beginning, since we wanted to approach a real-life scenario as much as possible with the available equipment.

Except from the results of these experiments, it is also important to assess the work in respect to a number of qualitative criteria, such as scalability and adaptability. Together with the quantitative experiment results, after this chapter a complete picture of the thesis results will have been formed.

This chapter is organized as follows: In Section 5.1, we describe our experimental setup and how we went all the way from simulations to prototyping to verify the functionality of our testers. In Section 5.2, we list and interpret the numbers that came as a result of the aforementioned experiments. In Section 5.3 we assess both methods in respect with specific qualitative criteria: Scalability, adaptability, applicability and graceful degradation potential. In Section 5.4, we list a set of guidelines for a more diagnosis-friendly implementation of the basic reconfigurable logic block. Finally, in Section 5.5 we summarize this chapter.

5.1 Experimental Setup

Our verification procedure naturally started from simulations before going on to the realization of a working prototype. This whole process is presented in this section.

5.1.1 Simulation

Our tester was described in VHDL. The first step of design verification had to be performed through extensive simulations. We created a model of the fault-free CUT and modified it accordingly for every fault, thus producing 150 models, one for the presence of every fault. Subsequently, we connected the tester to the CUT model and ran simulations for the fault-free and each of the faulty models. Any uncovered problems were corrected and each time that the tester was modified, the simulation procedure had to start from the beginning, because the modification might have created a problem in an already simulated case. When 151 consecutive simulations for each of the 2 testers were completed, we considered the simulation complete. Some indicative simulation waveforms are shown in Figures 5.1 to 5.6.

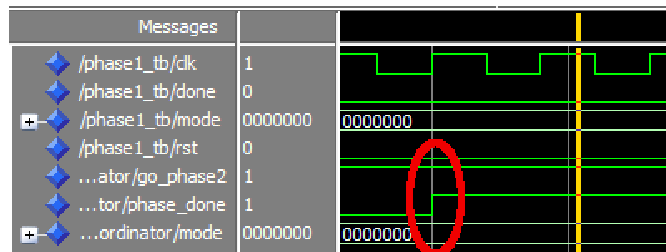

Figure 5.1: Response of the first phase for a fault free CUT. When the PHASE_DONE signal is raised, the GO_PHASE2 signal is '1', causing a second phase to occur.

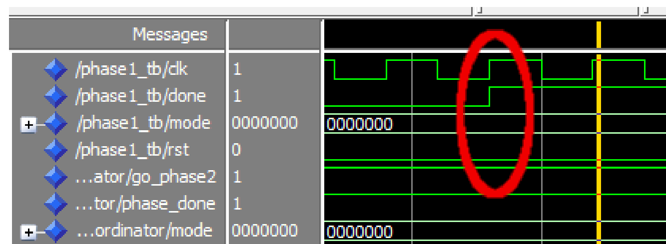

Figure 5.2: Response of the second phase for a fault free CUT. When the DONE signal is raised, the RESPONSE is a vector of zeroes, denoting a fault-free CUT.

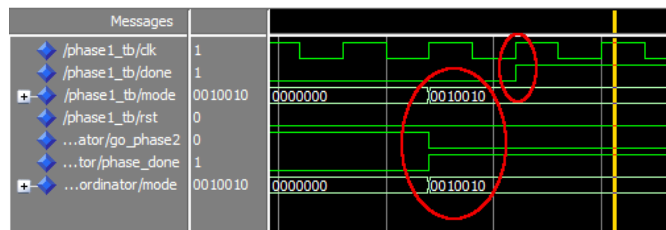

Figure 5.3: Detection of fault f21 with the function generator method. This fault is diagnosed during phase 1. Thus, when PHASE_DONE is raised, GO_PHASE2 is '0' and subsequently the DONE signal is raised one clock cycle later. The correct MODE is also produced.

5.1.2 Emulation

After completing the simulation stage we were confident that the testers were designed correctly. The next step was to make sure that our VHDL code is synthesizable and functions as intended on an actual FPGA.

Using the Xilinx University Program Virtex-II Pro Development System (XUP Virtex-II Pro), we realized the code that was used for simulations. That meant that we had the final realization of the testers connected to a *model of* the CUT, since the synthesis and implementation tools were not able to translate our VHDL model of the slice to an actual, physical slice on the Virtex-II Pro. We used both a fault-free CUT model and selected faulty ones. Thus, in this emulation stage we made sure that our

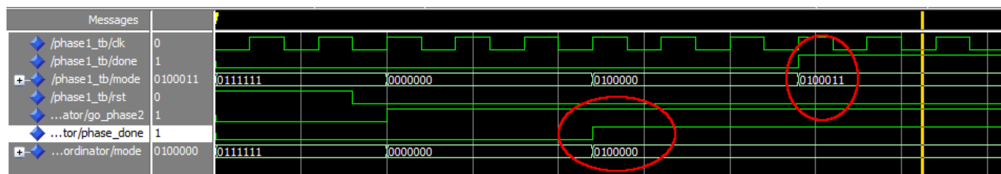

Figure 5.4: Detection of fault f61 with the function generator method. This fault is diagnosed in phase 2, thus when PHASE_DONE is raised, GO_PHASE2 is '1' and subsequently a second phase starts. A temporary MODE is also produced, which is slightly modified during phase 2 to take its final value. Note that both phases take only a few cycles, because the diagnostic trees resolve the particular fault very quickly.

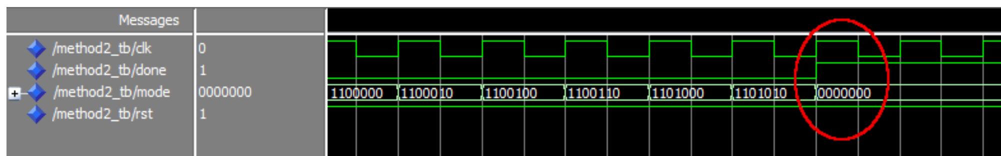

Figure 5.5: Response of the shift register method for a fault free CUT. When the DONE signal is raised, the MODE is a vector of zeroes, denoting the absence of a fault.

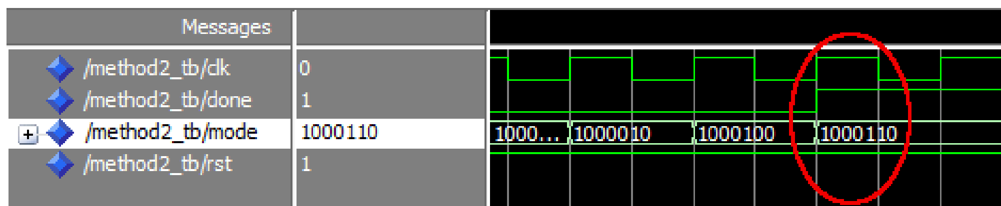

Figure 5.6: Detection of fault f34 with the shift register method. When the DONE signal is raised, the correct MODE is also produced.

tester was synthesizable and functional. The final step was to use the tester to test a real CUT, that is a physical Virtex-II slice instead of a hardware model of it.

5.1.3 Realization and Prototyping

The missing piece to have a complete prototype of the tester working in natural conditions, was to form a CUT exactly like we defined it in Section 3.1 and apply our tester to it.

To build the CUT we used the Xilinx FPGA Editor tool. We designed the CUT completely manually, by inserting the desired configuration for the function generator method (phases 1 and 2) and for the shift register method to one of the slices of the Virtex-II FPGA. In this way, we produced a hard macro for each case, which is a complete, synthesized, placed and routed design, that can be used in bigger designs as a black box component. We instantiated this correct version of the CUT inside our VHDL code,

in place of the VHDL CUT model and realized the complete system on the device. After fixing some minor problems related to the correct manual configuration of the CUT and interfacing with the tester, we got the correct tester response for the fault-free case.

In order to apply the tester also on real faulty versions of the CUT, we had to perform fault injection. We deliberately configured the CUT with faults, essentially performing fault injection through the bitstream [8]. Examples of faults that are easily injected are mentioned below:

- Faults of the CUT inputs can be injected by feeding an input with a constant '0' or '1', instead of the correct test pattern. Note that these faults include the all-important address line faults.
- LUT contents can be configured to the wrong value. The tester still expects to read the correct value from it, so we are able to check if the correct faulty outcome is produced.
- Configuration multiplexers can be easily configured to select their other input than the one they are supposed to, thus forcing the tester to diagnose the fault.

Selected faults were injected in all these ways and the tester produced the correct response in all cases. By this stage, we considered the prototyping complete.

During both the emulation and realization stages, we used some features of the XUP-Virtex-II board to perform debugging: Four LEDs and four general purpose switches used for user input. Our debugging scheme is illustrated in Figure 5.7. We used the switches to multiplex 16 different 4-bit elements and observe any of them on the LEDs. If we needed to probe some transitive state of the system, we had to store it in special registers at the correct time during the system operation. In this way, we created a 64-bit debugging file which was sufficient to give us adequate information without the need to often reconfigure the FPGA, a process that took approximately 7 minutes.

Based on the above experiments, we gathered results for both testers. These results will be presented in the following section.

5.2 Quantitative Measures

The purpose of the experiments described in the above section was not only to verify the system's correct function, but also to collect some measurements in order to evaluate the methods quantitatively. The metrics that we used in order to do that are listed below:

- The area cost of the system.
- The performance of the system, measured through both the achieved frequency and the average latency.
- The quality of the performed diagnosis, by calculating the diagnostic accuracy and diagnostic resolution of each method.

The area cost of the system is measured in Virtex-II slices. Our methods are based on the Built-in Self-Test (BIST) principle, thus they are realized on slices completely

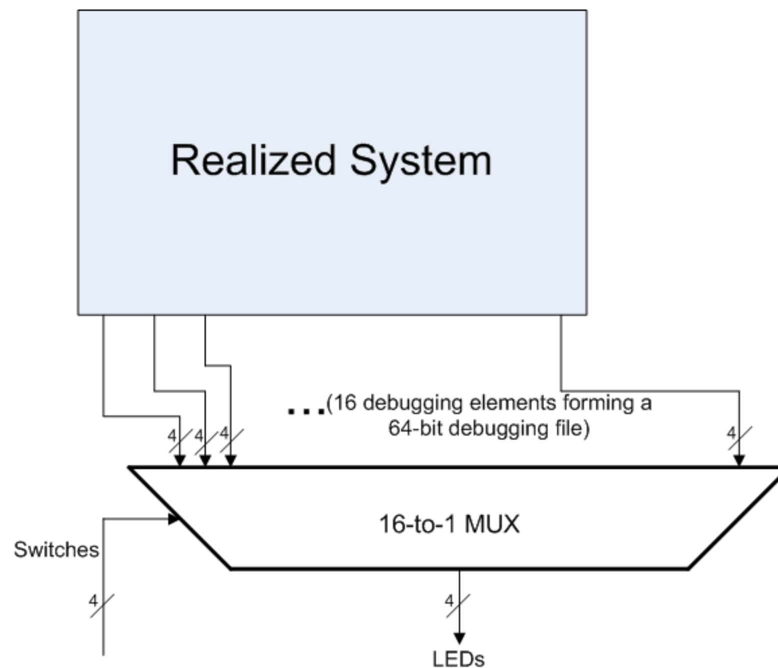

Figure 5.7: Debugging scheme for the emulation and prototyping stages.

identical as the one they test. It is interesting to see how many such units are needed to diagnose the faults of one. The area cost of the first method is 250 slices, while that of the second is 272 slices. The difference is due to the fact that the diagnostic trees of the second method are in average more complex, because there is no distribution of the faults between two phases.

For a moderately sized device, like the xc2vp30 that we used, this is slightly under the 2% of the available logic resources, and that is without packing unrelated logic on one slice, which can reduce the area cost if there is such a need. The tester is supposed to be used repeatedly, each time a problematic slice is spotted by any means. As one of the important components of implementing fault tolerance, we think this is a reasonable overhead cost. The key to keeping the fault tolerance overhead to a low level in the future is a cheap module performing function-to-resource matching.

The best achievable clock frequency for the function generator method is 109.9 Mhz, while for the shift register method it is 102.8Mhz. This difference is again due to the fact that the diagnostic trees of the second method are somewhat more complex, requiring logic expressions with more inputs.

The latency for each method depends on the outcome, so it has to be calculated for the average case. In Table 5.1 we list how many faults are diagnosed on every cycle for the function generator method and in Table 5.2 we do the same for the shift register method. The cumulative percentage of faults diagnosed on every cycle, as it derives from these tables, is illustrated in Figure 5.8 The cycle on which each fault is diagnosed can be taken from the respective diagnostic trees, increased by 4 cycles due to the system 4-stage pipeline. To calculate the average latency for a faulty CUT, we consider all fault

Table 5.1: Distribution of diagnosed faults between cycles for the function generator method.

Cycle	Faults	Cycle	Faults
5	1	RC + 6	17
6	3	RC + 7	8
7	5	RC + 8	8
8	24	RC + 9	2
9	2	RC + 10	2
10	6	RC + 11	2
11	4	RC + 12	2
12	0	RC + 13	2
13	4	RC + 14	2
14	6	RC + 15	2
15	2	RC + 16	2
16	2	RC + 17	2
17	2	RC + 18	2
18	2	RC + 19	2
19	2	RC + 20	2
20	2	RC + 21	10
RC + 5	6	RC + 22	4

sites (wires, multiplexer control signals, memory cells and flip-flops) to have the same probability to be faulty. Consequently, to calculate the latency for a faulty CUT in cycles, we use the following formula:

$$L_{faulty} = \frac{\sum_{i=1}^{N} d_i * (i + Trc_i * f)}{D}$$

Whereas, to calculate the latency for a faulty CUT in nanoseconds, we use the following:

$$L_{faulty} = \frac{\sum_{i=1}^{N} d_i * (i/f + Trc_i)}{D}$$

where:

- i is the cycle index, ignoring the possible reconfiguration phase.
- N is the total number of cycles each method takes, ignoring the possible reconfiguration time.
- d_i is the number of faults diagnosed on cycle i.
- D is the total number of faults diagnosed by each method.
- Trc_i is 0 for cycles before the reconfiguration. This includes all cycles of the shift register method. For cycles following the reconfiguration, it is equal to the reconfiguration time (T_{RC}) in nanoseconds.

Table 5.2: Distribution of diagnosed faults between cycles for the shift register method.

Cycle	Faults	Cycle	Faults
5	2	24	2
6	10	25	4
7	13	26	2
8	10	27	0
9	7	28	2
10	14	29	2
11	6	30	2
12	8	31	2
13	6	32	2
14	4	33	2
15	2	34	2
16	2	35	2
17	2	36	2
18	6	37	2
19	2	38	2
20	2	39	2
21	2	40	2
22	2	41	2
23	2	42	4

- f is the operating frequency of each method.

The Reconfiguration time was estimated according to [37] to be 8547ns. According to that and the above formulas, the average latency for a faulty CUT (L_{faulty}) is calculated to be $0.5 * 8547 * f + 21.8$ cycles, or 4769ns for the function generator method and 17.5 cycles or 170ns for the shift register method. The critical impact of the reconfiguration on the latency of the function generator method is obvious. In the case of a fault-free CUT, the methods will have to run until the end, thus the latency in this case (L_{ff}) is $8547 * f + 42$ cycles or 8929ns for the function generator method and 42 cycles or 407.4ns for the shift register method. The overall average latency depends on the probability of the CUT to be faulty (P_{faulty}) or fault-free (P_{ff}) and is calculated as follows:

$$L = P_{ff} * L_{ff} + P_{faulty} * L_{faulty}$$

The average latency in nanoseconds for both methods and for $P_{faulty} = 1, 1/2, 1/4, 1/8$ *and* $1/16$ are summarized in Table 5.3 and illustrated in Figure 5.9. In addition, in Figure 5.10, we illustrate the same results for the case of applying the method sequentially to all 160 the slices of a column of the FPGA with only one reconfiguration, which is a more fair measure for the function generator method, since one column is the minimum partial reconfiguration fragment for Virtex-II Pro.

The diagnostic accuracy for the function generator method and a faulty CUT (ACC_{faulty}) is 0.96 (96%), according to the assumption that each fault site has the

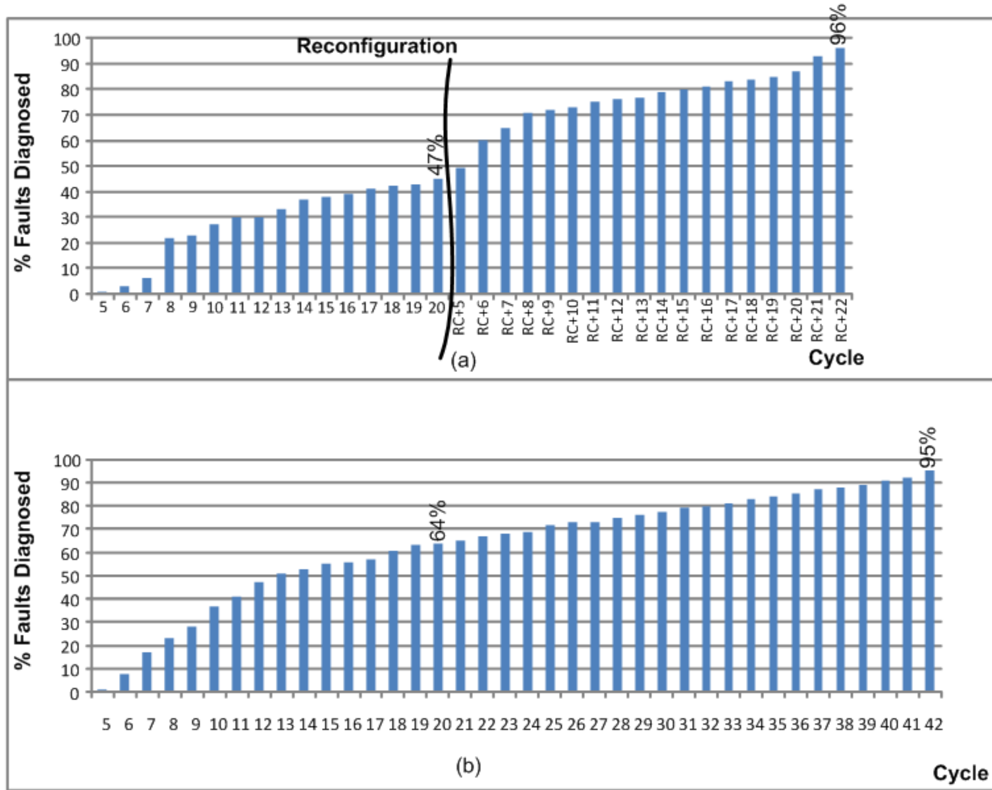

Figure 5.8: Cumulative percentage of diagnosed faults on every cycle for the function generator method (a) and for the shift register method (b).

Table 5.3: Average Latency for both methods and various values of P_{faulty}.

P_{faulty}	**Method 1 Latency (ns)**	**Method 2 Latency (ns)**
1	4769	170
1/2	6835	289
1/4	7869	348
1/8	8385	377
1/16	8667	393

same probability to produce a fault. The missing accuracy is due to the inability to diagnose faults of the ENABLE signals of the shift registers with this method. The overall accuracy depends again on the probability of the CUT to be faulty, since a fault-free CUT is always accurately diagnosed. Thus, the diagnostic accuracy is calculated as follows:

$$ACC = P_{ff} * 1 + P_{faulty} * ACC_{faulty}$$

For the shift register method, the value of ACC_{faulty} is 0.95 (95%). The missing accuracy is due to the inability of this method to detect half the faults of configuration multiplexers. The diagnostic accuracy for both methods and P_{faulty} =

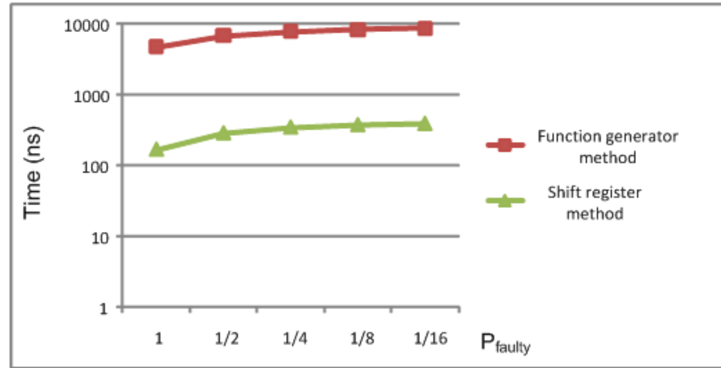

Figure 5.9: Comparative Latency graph for the two methods.

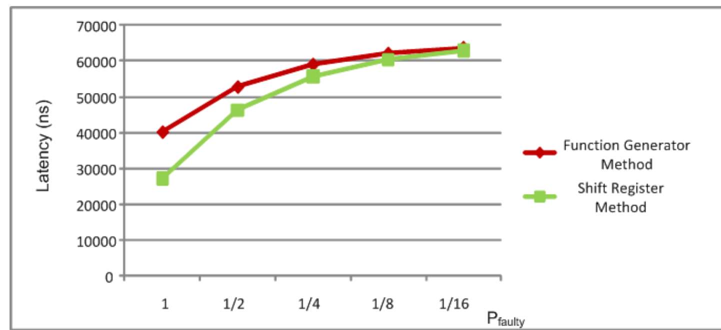

Figure 5.10: Comparative Latency graph for the two methods applied on a whole column of slices. Because of the big number of slices, the effect of the reconfiguration diminishes.

$1, 1/2, 1/4, 1/8$ *and* $1/16$ are summarized in Table 5.4 and illustrated in Figure 5.11.

The diagnostic resolution for the function generator method is 1.25. This value is derived as follows:

- 126 faults are fully diagnosed.
- 18 faults are diagnosed in groups of 3.

Thus, there are 126 cases that the set of possible faults contains only one element and 18 cases that it contains 3 elements. The average number of elements in the set of possible

Table 5.4: Diagnostic Accuracy for both methods and various values of P_{faulty}.

P_{faulty}	**Method 1 Accuracy**	**Method 2 Accuracy**
1	0.96	0.947
1/2	0.98	0.973
1/4	0.99	0.987
1/8	0.995	0.993
1/16	0.9975	0.997

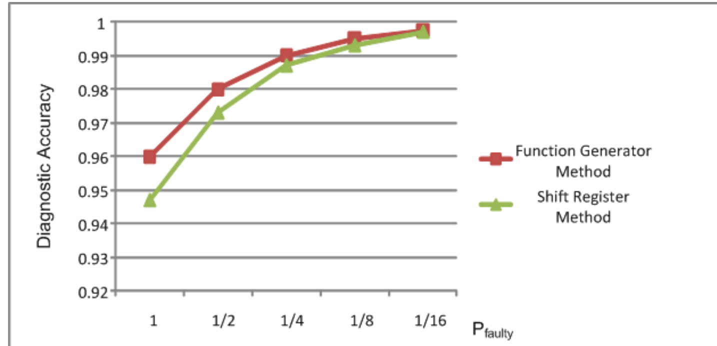

Figure 5.11: Diagnostic Accuracy for both methods and various values of P_{faulty}.

Table 5.5: Summary of quantitative measurements.

Measure	Method 1	Method 2
Area	250 slices	272 slices
Frequency	109.9Mhz	102.8Mhz
Latency (faulty CUT)	4,769 ns	170 ns
Latency ($P_{faulty} = 1/4$)	7,869 ns	348 ns
Diagnostic Accuracy (faulty CUT)	0.96	0.947
Diagnostic Accuracy ($P_{faulty} = 1/4$)	0.99	0.987
Diagnostic Resolution	1.25	1.37

faults is 1.25. Note that almost all missing resolution is due to grouping together of faults that wouldn't provide any extra graceful degradation potential if further distinguished.

For the shift register method, the diagnostic resolution is derived as follows:

- 112 faults are fully diagnosed.
- 16 faults are diagnosed in sets of 2.
- 6 faults are diagnosed in sets of 3.
- 8 faults are diagnosed in sets of 4.

The resulting value, calculated in the same way that was explained for the function generator method, is 1.37. This method has worse diagnostic resolution, because it is not able to distinguish between some wire faults in one phase.

All quantitative measurements are summarized in Table 5.5. For the latency and diagnostic accuracy, we use the value $P_{faulty} = 1$, which corresponds to the original formulation of the problem (1 slice that is known to be faulty) and $P_{faulty} = 1/4$, which corresponds to applying the methods on the 4 slices that constitute one CLB in Virtex-II Pro and one of them turns out to be faulty.

5.3 Qualitative Measures

Except from the measurements that we took during the experiments described in Section 5.1, the two proposed methods have also to be assessed according to qualitative criteria. The criteria we use are scalability, adaptability, applicability and graceful degradation potential.

5.3.1 Scalability

As it has been clearly stated very early in this report, the proposed methods target a single FPGA basic logic block that is known to be faulty, or on a set of a few such blocks at least one of which is faulty. In other words, it follows up the stages of fault detection and localization. It is thus very important to explore the potential scalability of the methods when applied on n slices. It is especially important to consider the case of 4 slices, since they form one CLB, which is the next level block in the FPGA hierarchy.

Both methods scale with identical complexity, since they are based on the same general structure and design principles. In the somewhat optimistic case that we know exactly one of the n slices to be faulty, scaling of the methods can be performed with constant complexity ($O(1)$) for both area cost and latency. Indeed, a tester like the ones implemented in this thesis can be used, along with a wrapper that examines the responses of the n basic blocks to detect the presence of a fault in one of them and also to inform the tester in which block the fault is. The functioning principle of the wrapper is identical to the ones presented in Section 4.3.1. The area cost of the wrappers of course depends on n, but for reasonable values of n it is smaller than that of the tester. The latency of such a setup is identical to that of the method being applied to one slice only.

It is more important, however, to consider the more consistent, according to the framework we set for this thesis, case of every slice under test being potentially faulty, since we started our study with an assumed fault density of one fault per slice. In this case we cannot assume that the first occurrence of a fault in one of the slices means that the rest of the slices are fault-free. In this case, there are the following two extreme options for scaling the tester, which define a design space:

- Only one tester can be used, together with a wrapper that detects the presence of a fault in one of the slices. Note that this wrapper is similar to the one used in the previous case, but not identical, since it has to support the occurrence of faults in more than one slices simultaneously. Whenever a fault is detected, the testing procedure for the rest of the slices must be paused and the fault in question resolved, before the process is resumed for the rest of the slices. In this case, the area cost remains constant ($O(1)$), but the latency is $O(n)$ for n slices to be diagnosed.

- A number of testers equal to n can be used, one for each slice to be diagnosed. In this case the area cost of the system is obviously $O(n)$, but the latency is constant.

In this design space, any number of independent testers can be used, each of which is in charge of an equal number of slices. In any case, the *Area-Latency Product* scales with $O(n)$ complexity, with n being the number of slices to be diagnosed.

5.3.2 Adaptability

Another important characteristic of the proposed methods is their good adaptability to different reconfigurable devices. The guidelines used to diagnose different faults are applicable to the common basic block structure of most commercial FPGAs. The basic components of all such blocks are the same: LUTs, multiplexers and flip-flops. Also, the roles of the components are identical: LUTs can implement (among others) logic functions and shift registers, the multiplexers are used to combine and route the results of the logic, while the flip-flops are used to implement the sequential part of every system.

As far as the function generator method is concerned, a suitable configuration for the LUTs can be formulated in order to check memory faults, address line faults and also help with the diagnosis of the rest of the slice components, except from the LUT. In fact, for bigger LUT sizes (i.e. the ones of the Virtex-5 devices), it is easier to accommodate all tests, but the worst-case latency is bigger, since it is determined by the size of the LUT.

For the shift register method, the faults of address lines, wires and multiplexers are taken care of using the read operation on the boundary memory cells of the shift register, which can be performed regardless of the register length. The march test for the rest of the memory cells does not depend at all on the slice structure. Note that every non-controllable multiplexer in the slice, means 3 faults that cannot be detected by the shift register method, one control signal fault and 2 faults of the non-selected multiplexer input, just like it has been explained for the non-controllable multiplexers of the Virtex-II slice.

5.3.3 Applicability

While designing the methods, we made every effort possible to be completely bounded by the characteristics of real today's devices. This meant that both methods are directly applicable on actual FPGA slices with minimal effort, as proven by our working prototypes.

There was, however, one issue we encountered during the realization stage, that has to be mentioned and addressed. It turns out that the inputs FXINA and FXINB of the slice are not directly, globally controllable, while the outputs FX and F5 are not directly, globally observable. This happens because these inputs and outputs, together with the FXMUX and F5MUX multiplexers, are used to combine 2 to 16 LUTs, in order to implement functions of 5 to 8 inputs. For this purpose, the FX and F5 outputs are connected to FXINA and FXINB inputs of neighboring slices, depending on the position of each slice within the CLB. Thus, the aforementioned outputs have to be propagated through neighboring slices in order to eventually be globally observable, while the inputs have to be accessed through neighboring slices in order to be globally controllable.

In our prototype we used neighboring slices to recreate the exact method as it is described in Chapters 3 and 4. To achieve that, we needed 7 more slices to be activated except from the one under test. Except from the area overhead, this can mean that the method cannot be applied on more than one slices simultaneously as explained in Section 5.3.1, since in that case the neighboring slices cannot be used for propagating the non-observable and access the non-controllable signals. This can be solved by excluding

from the tests the parts defined by these inputs and outputs, which are the following:

- The signals themselves, FXINA, FXINB, FX and F5.
- The FXMUX multiplexer.

These features correspond to the upper part of the slice and the F5 output. Note that the F5MUX multiplexer can still be tested through the X output, since its inputs are controllable and also that the F5 output needs only one neighboring slice to be propagated through, which is not prohibitive. The really problematic part of the slice in this respect is the one defined by FXINA, FXINB and FX.

The issue explained above hinted us that the chain of F5MUX and FXMUX multiplexers of neighboring slices has to be checked separately from the rest of the features. Our next approach will surely be in this direction.

5.3.4 Graceful Degradation

The graceful degradation potential of each method has already been summarized in Section 3.4 and specifically in Table 3.6. In this section we briefly assess this potential of both methods in comparison to existing works and to each other.

Characterization in related work is still, in our opinion, on a conservative level. After testing, the failing configurations are used with the possible addition of a few more diagnosis configurations to determine which component of the basic block actually is faulty. Subsequently, a function is mapped on this basic block that does not use the faulty LUT or storage element [2].

In our approach, we opt for a different approach. We diagnose the fault that is present with as good resolution as possible and, for each fault, we define a mode of degradation that describes what the slice can *not* do because of the presence of this fault. In this way, we do not sacrifice more functionality than it is actually necessary. The difference between sparing, conservative matching and our approach is conceptually illustrated in 5.12. As explained in Section 3.4, we are optimistic that our approach is not going to critically complicate the stage of matching functions to resources.

Examining the two methods in comparison to each other, it has to be noted that the faults diagnosed by the function generator method are more efficiently distributed among the modes of degradation in Table 3.6 than the ones diagnosed by the shift register method. The reason for this is the inability of the second method to distinguish between some wire faults, in which case the existence of all faults in the resulting group have to be assumed present. This, however, is less important than the inability of the first method to test the slice configured as a shift register, combined with the reduced coverage of memory faults.

5.4 Design for Testability Guidelines

From the beginning of this thesis, one of our basic goals was to use the lessons we took throughout the development of the diagnosis methods, in order to provide some guidelines for a more diagnosis-friendly implementation of the substitutable reconfigurable

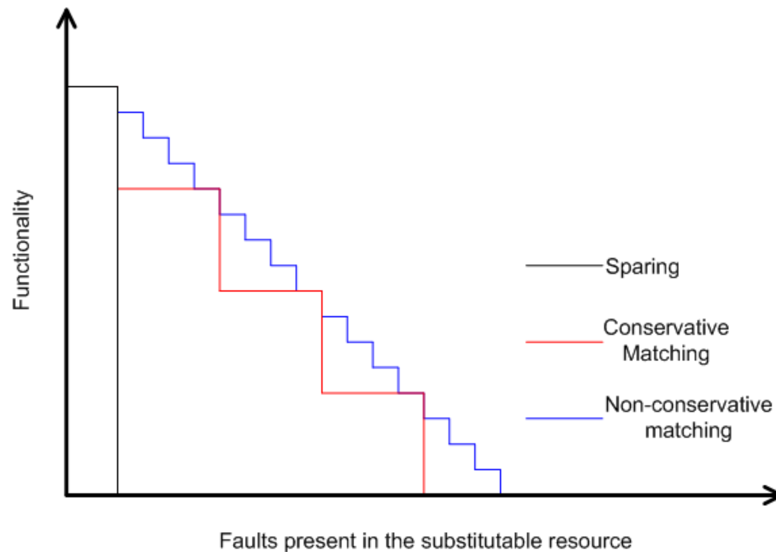

Figure 5.12: Conceptual Graph comparing sparing and matching with different degrees of diagnostic resolution.

resource. These guidelines are applicable to all reconfigurable hardware devices, but they are mainly meant to be used in the framework of the FaTES research project. Our conclusions are summarized below:

- The slice LUTs should be smaller than 16 bits, which means 8 bits, since the size has to be a power of 2 and 4 bits would be too small. Of course, 8-bit LUTs were never considered for commercial FPGAs, but that was not due to reasons associated with fault tolerance, but for reasons of efficient implementation of boolean functions. From a diagnosis (and actually, also testing) standpoint, however, we have established that a determining factor for the overall latency, especially for methods of one or a few phases, is the size of the LUTs, since their contents have to be read one by one. In our methods, the surrounding logic and storage elements have been almost completely tested in the first 8 cycles.
- There should be write access to the whole configuration bitstream, at least during the testing and diagnosis procedure. This is particularly important for multiplexer control signals, which have to be kept to a steady value, unless time-consuming reconfiguration takes place. If these control signals behaved as variable inputs of the CUT, the shift register method in particular could achieve a diagnostic accuracy of 100% and resolution almost equal to 1.
- All primary inputs of the basic block should be directly controllable and all the outputs should be directly observable from any other logic resource of the device. This modification would solve the applicability problems that we faced in our methods and were described in Section 5.3.3.
- It would help the testing and diagnosis process if there was the ability of swapping the contents of the two LUTs, or complementing all of their contents, without

Table 5.6: Comparison of the two proposed methods.

Measure	Method 1	Method 2
Area	+	
Frequency	+	
Latency		++
Diagnostic Accuracy	≈	≈
Diagnostic Resolution	≈	≈
Scalability	=	=
Adaptability	=	=
Applicability	=	=
Graceful Degradation		+

partial reconfiguration. This would eliminate the need for reconfiguration that is present in the function generator method.

5.5 Summary

In this chapter we provided complete evaluation of both proposed methods, in respect to all relevant criteria. We concluded that the function generator method scores slightly better in terms of area cost and highest achievable frequency, but it falls back critically in terms of average latency, also because of the reconfiguration cost. The two methods have similar diagnostic accuracy and resolution, but the missing diagnostic accuracy of the first method is more detrimental to fault tolerance than this of the second method. On the qualitative side, both methods are equally scalable, adaptable and applicable. As for the graceful degradation potential, the shift register method is preferable due to the fact that it covers both the shift register and function generator modes of operation, while diagnosing the slice with the function generator method means that, regardless of the result, the shift register functionality has to be sacrificed. In Table 5.6 we summarize the strong and weak points of each method. According to the preceding analysis, if we were to choose one of them for basing an integrated fault tolerance technique on, we would choose the shift register method.

In comparison to existing techniques, our methods are certainly more costly in absolute area numbers, but because of their reusability throughout the whole lifetime of the device, the area percentage of a moderately-sized device that they occupy is quite low, a bit less than 2% of the xc2vp30 device. In terms of the quality of diagnosis, we are in most cases able to determine the exact fault that is present, contrary to existing approaches that determine the faulty component. This, in turn, uncovers a lot of graceful degradation potential that was not present in previous works. Also, by decoupling the diagnosis problem from that of fault detection and localization, we relaxed the requirement for 100% fault coverage and decided that rescuing the most frequently used functionality of the basic block is sufficient for efficient fault tolerance. This allowed us to restrain ourselves to 1 or 2 testing phases only, critically reducing the time needed for

diagnosis.

Overall, we are confident that we provided a viable approach for efficient fault tolerance. More problems that have to be independently addressed in the future, in order to complete the fault tolerance framework that we started defining in this thesis will be briefly presented in the next chapter, along with a summary of the whole report.

6 Conclusion

Throughout this report, we attempted to provide an as complete as possible picture of the work that we did in the MSc thesis framework. The problem at hand is well defined based on existing work and current research trends; the proposed methods are adequately explained and shown to produce a solution for the problem at hand; the experiments we conducted are described and relevant results gathered and illustrated. We made every effort possible to be consistent, accurate and complete, through explaining the reasons for every choice we made, stating advantages and drawbacks of all our decisions and generally placing our research in the framework already defined by already established knowledge.

This chapter provides the conclusion to this thesis. In Section 6.1 a chapter by chapter summary of this report is attempted, which can also be used as a source for locating a particular topic within the report. In Section 6.2 the main contributions of the thesis are listed, according to both our starting expectations and the actual results. Finally, in Section 6.3 interesting future prospects are introduced, to complete the interfacing of the present work with other relevant ones.

6.1 Summary

In this section, we remind the reader of the most important facts stated in this report and provide a detailed revision of its structure.

We began in Chapter 1, by justifying the importance of fault tolerance for a wide range of computing applications. We went on to stress the anticipation of researchers that this importance will increase in the near future because of the ever-shrinking technology features in Section 1.1. The difference between the widespread, but relatively conservative, sparing and the more efficient matching techniques is also explained as motivation for the work.

Subsequently, the problem targeted by this thesis is defined in Section 1.2. Specifically, the proposed methods target to diagnose the existing fault in an FPGA basic logic block that is known, through prior testing, to be faulty. They can also be applied on a relatively small number of such blocks, at least one of which is known to be faulty. Also, according to the diagnosis result, characterization of the logic block has to be performed. In other words, the subset of functions that the faulty block can still accommodate despite its defect is determined. A detailed list of the thesis goals, based on this problem statement is given in Section 1.3.

The purpose of Chapter 2 was twofold: First, to define basic notions and terms, essential for understanding the rest of the report, i.e. terms relevant to fault diagnosis and second, to summarize existing research that is relevant with the problem at hand. In Section 2.1.1, the notion of fault models as abstractions of actual defects was introduced,

and a few specific models important for this thesis (for wires and memory cells) presented. The family of *march tests* to deal with the memory faults was also briefly introduced.

Subsequently, in Section 2.1.2, the tasks relevant to fault detection and tolerance were explained in the order they actually happen: Testing, Localization, Diagnosis and Repair. Special attention was paid to diagnosis, as it is the actual object of this thesis. Necessary relevant terminology is defined: The meaning of diagnostic accuracy and resolution, fault-free, faulty end error responses, fault dictionaries and diagnostic trees is explained. Also, a categorization of existing fault diagnosis methods is provided, and our approach categorized as cause-effect, diagnostic tree based diagnosis. The basic reason for this choice is the relatively small size of the CUT, which allows for all the fault simulations to be performed a priori.

In Section 2.1.3, related work on FPGA testing is summarized. The special characteristics of FPGAs (regularity, heterogeneous nature and reconfigurability) are listed and their effect on the testing strategies explained. It is stated that Built-in Self-Test approaches are profitable when applied on FPGA testing, because of the non-existing area overhead. Useful conclusions are drawn from the whole analysis and the advantages of different methods are subsequently combined, whenever possible, to develop our methods. The number of testing phases (reconfigurations) is spotted as the dominant factor in determining testing time, which hinted us to reduce the number of phases as much as possible. That was possible by decoupling the diagnosis problem from that of fault detection and localization, which are extensively addressed by previous works and effectively dropping the requirement of full fault coverage.

In Section 2.2 we attempt to solidify the fault tolerance framework, which enhances the significance of the thesis. The suitability of regular, reconfigurable realization platforms for fault tolerance is stressed and the techniques of sparing and matching explained in more detail, also through related research. The relevance of the targeted problem is further supported by the existence of systems built from scratch with fault tolerance in mind, one of which is the newly-defined paradigm of the FaTES research project. Finally, is Section 2.3, all arguments for the thesis significance and relevance are summarized.

After setting the framework, it was time to proceed to the development details of the proposed methods themselves, in Chapter 3. In Section 3.1, the simplified model of our CUT, which is the Virtex-II Pro slice, was described. Each component was individually approached to define its functionality and a suitable fault model was determined for each of them, based on realistic assumptions and diagnosis requirements. From these fault models, the complete list of faults to be checked was derived. The section concluded with the guidelines that help us distinguish faults of different kinds by observing the effect they have on the CUT outputs. Based on these guidelines, our methods could subsequently be explained.

Section 3.2 is about the development of the function generator method. In Section 3.2.1 the partitioning of the CUT, in order to simplify the diagnosis procedure, was explained. The most important part was determining the suitable LUT configurations in order to accommodate the diagnosis of memory, address line, multiplexer, storage elements and wire faults. The chosen fault models posed restrictions to the configuration contents and an appropriate solution was found in Section 3.2.2. Based on this analysis and on the guidelines for diagnosing different fault that were already determined in the

previous section, the procedure of deriving the diagnostic trees was explained in detail in Section 3.2.3.

The drawbacks of the function generator method were our motivation for developing a different method, based on a shift register configuration of the slice. This procedure is explained in the beginning of Section 3.3, followed by the partitioning of the faults for this new method in Section 3.3.1. Subsequently, the basic characteristic of the shift register method is unfolded in Section 3.3.2, which is the formulation and application of a march test to check the memory faults. Finally, in Section 3.3.3 all the above analysis is particularized to specific diagnostic trees.

Chapter 3 is concluded by explaining how a diagnosis outcome results in characterization of the slice in Section 3.4.

In Chapter 4, we present the hardware implementation of the resulting testers that realize the proposed methods. In Section 4.1, the high-level organization of the system is presented, based on two substantial components: The response analyzers and the system controller. In Section 4.2, internal details of those two components are discussed, by mentioning the duties of each one and explaining how they are carried out. Finally, in Section 4.3, miscellaneous implementation topics are discussed: How identical response analyzers are implemented only once and wrapped to accommodate more than one partition; how the diagnosed faults and resulting modes of degradation are efficiently encoded; and how the system is pipelined to allow for a higher operating frequency and better distribution of calculations within it.

Finally, in Chapter 5, the presented work is evaluated and the results illustrated and analyzed. In Section 5.1, the experimental setup is described and all conducted experiments listed, beginning from simulations up to the realization of a working prototype. Subsequently, in Section 5.2, all measurements taken during the aforementioned experiments are organized and interpreted. It is shown that the methods achieve at least 95% diagnostic accuracy and a diagnostic resolution close to 1. Also, the defective slice is successfully characterized in 170 to 8500 ns. The area cost of the realized testers is not prohibitive for a system with hard fault tolerance requirements. Eventually, in Section 5.3, the proposed methods are assessed in respect to a set of qualitative criteria: Scalability, adaptability, applicability and graceful degradation potential. The proposed methods are shown to score well on all these measures, constituting a basis for a viable integrated fault tolerance approach, which we hope to complete with future works.

6.2 Thesis Contributions

The main problem targeted by this thesis was the fast, high-resolution fault diagnosis and subsequent characterization of an FPGA slice that is known to be faulty, or belongs to a set of a few slices at least one of which is known to be faulty. Through working on this well defined framework, we achieved the following main contributions:

- **The design and implementation of two distinct methods for fault diagnosis:** Both methods provide high-resolution fault diagnosis of a Virtex-II Pro FPGA slice, based on a hybrid fault model which consists of suitable fault models for every component of the slice.

- **The decoupling of the diagnosis problem from those of fault detection and localization:** Contrary to existing research which tends to address these problems in an integrated manner, we solved the diagnosis problem separately, taking advantage of the fact that an abundance of existing methods is able to spot the defective component. In this way, we were able to restrict ourselves to only one or two diagnostic configurations of the CUT, since missing some faults or not testing some special features of the slice can not affect its correct functioning.

- **A matching-aware characterization scheme:** We performed characterization of the slice based on the diagnosis result, in a modular and efficient manner. Indeed, if a different characterization scheme is chosen in the future, it can seamlessly replace the existing one and use the same diagnosis result. Our characterization scheme, in turn, sets the stage for the development of an efficient matching heuristic, that can assign functions to defective slices by inspecting only a few bits of the intended configuration bitstream.

- **The preparatory work for an integrated fault tolerance scheme, possibly in the framework of the FaTES research project:** The resulting testers were prototyped on an actual Virtex-II Pro device and applied on one of its slices. Fault injection through the bitstream was used to verify the correctness of the developed methods. Applicability issues were encountered, which taught us valuable lessons, both for developing more efficient methods in the future and for proposing a set of guidelines to design a more diagnosis-friendly basic logic block for the FaTES project (Design for Testability).

6.3 Future Work Suggestions

In the process of working on this thesis, we came across a number of interesting problems and prospects, which could be good starting points for future works. The main such prospects are summarized below:

- **Extension of the proposed methods:** The proposed methods can be extended, in order to perform fault diagnosis on all the slice features. For the purposes of this thesis, we excluded some special features, like fast carry chains and arithmetic dedicated gates. Although this was a conscious choice, it is still interesting to explore the extra cost of checking the complete functionality.

- **Development of new methods based on the drawbacks of the proposed ones:** While applying the proposed methods on a real slice, we encountered some problems due to the realization details of the slice. We are confident that it will be profitable to develop a special method for testing the chain of MUXF5 and MUXFX multiplexers, which are used to realize boolean functions of 5 to 8 inputs. This method has to be constrained by the LUT configurations described in this thesis, in order to be able to be combined with either of the proposed methods for the rest of the slice features.

- **A complete design of an easily testable substitutable reconfigurable resource:** Based on the guidelines that came as a conclusion from this thesis, a complete implementation of a substitutable resource should be proposed, to be used in the framework of the FaTES research project. This work should address the same problems that we did in this thesis, from a Design for Testability standpoint, instead of being restricted by the characteristics of existing devices.

- **The development of an efficient heuristic for matching functions to defective resources:** Our characterization scheme is aware of the fact that it will be followed-up by an algorithm which will efficiently facilitate the reuse of defective blocks. This matching algorithm has to be based on adaptability and precomputation, in order to achieve matching times that are not prohibitive for the application fault tolerance requirements.

6.4 Conclusion

As a conclusion, we would like to state our strong belief that the proposed methods, especially the shift register method, have great potential of being a critical component in an integrated fault tolerance scheme. Fault tolerance by using regular reconfigurable structures is a very popular research field and we are happy to offer even the minimum contribution to it. We hope to have the chance in the near future to continue the work that we started in this thesis, possibly by solving some of the problems stated in Section 6.3.

Bibliography

[1] *FPGA basic facts*, `http://en.wikipedia.org/wiki/FPGA`.

[2] Miron Abramovici, John M. Emmert, and Charles E. Stroud, *Roving stars: An integrated approach to on-line testing, diagnosis, and fault tolerance for fpgas in adaptive computing systems*, Evolvable Hardware, 2001, pp. 73–92.

[3] Miron Abramovici and Charles E. Stroud, *BIST-based delay-fault testing in FPGAs*, IOLTW, 2002, pp. 131–134.

[4] Miron Abramovici, Charles E. Stroud, Carter Hamilton, Sajitha Wijesuriya, and Vinay Verma, *Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications*, ITC, 1999, pp. 973–982.

[5] Miron Abramovici, Charles E. Stroud, Matthew Lashinsky, Jeremy Nall, and John M. Emmert, *On-line BIST and diagnosis of FPGA interconnect using roving STARs*, IOLTW, 2001, pp. 27–33.

[6] Miron Abramovici, Charles E. Stroud, Brandon Skaggs, and John M. Emmert, *Improving on-line BIST-based diagnosis for roving STARs*, IOLTW, 2000, pp. 31–39.

[7] Zaid Al-Ars, A. J. van de Goor, Jens Braun, and Detlev Richter, *A memory specific notation for fault modeling*, Asian Test Symposium, 2001.

[8] Monica Alderighi, Sergio D'Angelo, Marcello Mancini, and Giacomo R. Sechi, *A fault injection tool for SRAM-based FPGAs*, IOLTS, 2003, pp. 129–.

[9] S. Borkar, *Designing reliable systems from unreliable components: the challenges of transistor variability and degradation*, Micro, IEEE **25** (2005), no. 6, 10 – 16.

[10] G Chang, *(t, k)-diagnosability for regular networks*, Computers, IEEE Transactions on (2010).

[11] Jason A. Cheatham, John M. Emmert, and Stanley Baumgart, *A survey of fault tolerant methodologies for FPGAs*, ACM Trans. Design Autom. Electr. Syst. **11** (2006), no. 2, 501–533.

[12] Supratim Deb, D. Shah, and S. Shakkottai, *Fast matching algorithms for repetitive optimization: An application to switch scheduling*, Information Sciences and Systems, 2006 40th Annual Conference on, 22-24 2006, pp. 1266 –1271.

[13] Abderrahim Doumar and Hideo Ito, *Testing the logic cells and interconnect resources for FPGAs*, Asian Test Symposium, 1999, pp. 369–374.

[14] ———, *Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: A survey*, IEEE Trans. VLSI Syst **11** (2003), no. 3, 386–405.

[15] Abderrahim Doumar, Satoshi Kaneko, and Hideo Ito, *Defect and fault tolerance FPGAs by shifting the configuration data*, DFT, 1999, pp. 377–385.

[16] S. Dutt and F. Hanchek, *REMOD: a new methodology for designing fault-tolerant arithmetic circuits*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **5** (1997), no. 1, 34 –56.

[17] B.F. Dutton and C.E. Stroud, *Built-in self-test of configurable logic blocks in virtex-5 FPGAs*, System Theory, 2009. SSST 2009. 41st Southeastern Symposium on, 15-17 2009, pp. 230 –234.

[18] C. Elm and D. Tavangarian, *Associative search based test algorithms for test acceleration in FAST-RAMs*, Memory Testing, 1993., Records of the 1993 IEEE International Workshop on, 9-10 1993, pp. 38 –43.

[19] John M. Emmert, Charles E. Stroud, and Miron Abramovici, *Online fault tolerance for FPGA logic blocks*, IEEE Trans. VLSI Syst. **15** (2007), no. 2, 216–226.

[20] John M. Emmert, Charles E. Stroud, Brandon Skaggs, and Miron Abramovici, *Dynamic fault tolerance in FPGAs via partial reconfiguration*, FCCM, 2000, pp. 165–174.

[21] Maya Gokhale, Paul Graham, Darrel Eric Johnson, Nathan Rollins, and Michael J. Wirthlin, *Dynamic reconfiguration for management of radiation-induced faults in FPGAs*, IPDPS, 2004.

[22] John E. Hopcroft and Richard M. Karp, *A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, FOCS, 1971, pp. 122–125.

[23] Mu-Yue Hsiao and Julius T. Tou, *Application of error-correcting codes in computer reliability studies*, Reliability, IEEE Transactions on **R-18** (1969), no. 3, 108 –118.

[24] Wei-Kang Huang, Fred J. Meyer, Xiao-Tao Chen, and Fabrizio Lombardi, *Testing configurable LUT-based FPGA's*, IEEE Trans. VLSI Syst. **6** (1998), no. 2, 276–283.

[25] Kyosun Kim, Ramesh Karri, and Miodrag Potkonjak, *Heterogeneous built-in resiliency of application specific programmable processors*, ICCAD, 1996, pp. 406–411.

[26] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak, *Algorithms for efficient runtime fault recovery on diverse FPGA architectures*, DFT, 1999, pp. 386–394.

[27] Fernanda Lima, Luigi Carro, and Ricardo Augusto da Luz Reis, *Designing fault tolerant systems into SRAM-based FPGAs*, DAC, 2003, pp. 650–655.

[28] Tong Liu, Wei-Kang Huang, and Fabrizio Lombardi, *Testing of uncustomized segmented channel field programmable gate arrays*, FPGA, 1995, pp. 125–131.

[29] Fabrizio Lombardi, David Ashen, Xiao-Tao Chen, and Wei-Kang Huang, *Diagnosing programmable interconnect systems for FPGAs*, FPGA, 1996, pp. 100–106.

[30] Vishwani D. Agrawal Michael L. Bushnell, *Essentials of electronic testing*, Springer Science + Business Media, LLC, 2000.

[31] M.Y. Niamat, K.M. Attravanam, and M. Alam, *Testing FPGAs using JBits RTP cores*, Circuits and Systems, 2005. 48th Midwest Symposium on, 7-10 2005, pp. 1131 –1134 Vol. 2.

[32] Sndeep Gupta Niraj Jha, *Testing of digital systems*, The Press Syndicate of the University of Cambridge, 2003.

[33] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey, *ED4I: Error detection by diverse data and duplicated instructions*, IEEE Trans. Computers **51** (2002), no. 2, 180–199.

[34] E. Syam Sundar Reddy, Vikram Chandrasekhar, Milagros Sashikánth, V. Kamakoti, and Narayanan Vijaykrishnan, *Detecting SEU-caused routing errors in SRAM-based FPGAs*, VLSI Design, 2005, pp. 736–741.

[35] P.G. Ryan and W. Kent Fuchs, *Dynamic fault dictionaries and two-stage fault isolation*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **6** (1998), no. 1, 176 –180.

[36] André DeHon Scott Hauck, *Reconfigurable computing, the theory and practice of FPGA-based computing*, ch. 37, Denise E.M. Penrose, 2008.

[37] N. Pete Sedcole, Brandon Blodget, Tobias Becker, James Anderson, and Patrick Lysaght, *Modular partial reconfiguration in virtex FPGAs*, FPL, 2005, pp. 211–216.

[38] Charles E. Stroud, *Reliability of majority voting based VLSI fault-tolerant circuits*, IEEE Trans. VLSI Syst **2** (1994), no. 4, 516–521.

[39] Prasanna Sundararajan and Steven A. Guccione, *Run-time defect tolerance using JBits*, FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays (New York, NY, USA), ACM, 2001, pp. 193–198.

[40] A. J. van de Goor and Zaid Al-Ars, *Functional memory faults: A formal notation and a taxonomy*, VTS, 2000, pp. 281–290.

[41] Sying-Jyan Wang and Tsi-Ming Tsai, *Test and diagnosis of fault logic blocks in FPGAs*, ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design (Washington, DC, USA), IEEE Computer Society, 1997, pp. 722–727.

[42] Xilinx, *Virtex-ii pro and virtex-ii pro x FPGA user guide.*

[43] Xilinx, *Virtex-ii pro and virtex-ii pro x platform FPGAs: Complete data sheet.*

[44] Yinlei Yu, Jian Xu, Wei-Kang Huang, and Fabrizio Lombardi, *A diagnosis method for interconnects in SRAM based FPGAs*, Asian Test Symposium, 1998, pp. 278–282.

[45] Hamid R. Zarandi, Seyed Ghassem Miremadi, Costas Argyrides, and Dhiraj K. Pradhan, *Fast SEU detection and correction in LUT configuration bits of SRAM-based FPGAs*, IPDPS, 2007, pp. 1–6.

[46] Yi Zhao, Li Chen, and Sujit Dey, *On-line testing of multi-source noise-induced errors on the interconnects and buses of system-on-chips*, ITC, 2002, pp. 491–499.

Curriculum Vitae

Stavros Tzilis was born in Heraklion, Crete, Greece, on August 30th of 1982. His family moved to the town of Giannitsa 2 years later. He lived there until the age of 12, when he moved to Thessaloniki and Anatolia College, to complete his secondary education. During this time, in 1997, he was awarded with the bronze medal in the junior national Greek mathematics competition. After finishing high school in 2000, he moved to the town of Xanthi, in order to follow the 5-year Electrical, Electronics and Computer Engineering Diploma of the Democritus University of Thrace. He concluded his studies by defending his diploma thesis in June 2006, at which time he was already packing his bags for Delft, The Netherlands.

In TU Delft he followed the Computer Engineering M.Sc. program, while at the same time preparing the ground for a possible Ph.D. His main research interests include Fault Tolerance on Reconfigurable Hardware, and Computer Arithmetic on Reconfigurable Hardware, with or without Fault Tolerance considerations. Academic research, however, will have to wait for at least a while, because the upcoming year Stavros will be working for the European Space Agency, in the ESTEC installation in the Netherlands.

Stavros is very interested in all mind-stimulating activities, including cooking, role-playing games and team sports. However, he doesn't enjoy anything more than a nice, relaxing evening with persons that he feels comfortable with. It is very important for him to be surrounded by people with good will and to like what he is doing, in order to be able to happily invest a big amount of time, energy and attention to it.