POLITECNICO DI TORINO
Repository ISTITUZIONALE

New techniques for functional testing of microprocessor based systems

*Original*
New techniques for functional testing of microprocessor based systems / Cantoro, Riccardo. - (2017).

*Availability:*
This version is available at: 11583/2680228 since: 2017-09-14T15:27:23Z

*Publisher:*
Politecnico di Torino

*Published*
DOI:10.6092/polito/porto/2680228

*Terms of use:*
Altro tipo di accesso

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

*Publisher copyright*

(Article begins on next page)

04 August 2020

ScuDo

Scuola di Dottorato ~ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Computer and Control Engineering ($29^{th}$cycle)

# New techniques for functional testing of microprocessor based systems

By

**Riccardo Cantoro**

******

**Supervisor(s):**

Prof. Ernesto Sanchez

**Doctoral Examination Committee:**

Dr. Maksim Jenihhin, Referee, Tallinn University of Technology

Prof. Dr. Mario Schölzel, Referee, Universität Potsdam

Prof. Paolo Bernardi, Politecnico di Torino

Prof. Alberto Bosio, LIRMM

Prof. Michele Portolan, Laboratoire TIMA

Politecnico di Torino

2017

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Riccardo Cantoro
2017

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Doctoral School of Politecnico di Torino (ScuDo).

# Acknowledgements

I would like to acknowledge all the people who made the accomplishment of this thesis possible by guiding, supporting and encouraging me.

My thanks goes especially to Ernesto Sanchez, Paolo Bernardi and Matteo Sonza Reorda. I feel very grateful for the privilege of working with them and I hope our collaboration will continue.

I also acknowledge all the other people I have worked with in the Politecnico di Torino research group. A special thanks goes to Davide Piumatti, whose works has been fundamental to the experimental part of this thesis.

I acknowledge all the researchers I have been in contact with over these years. Thanks to the fruitful collaboration and the time spent together in conferences and meetings.

Finally, I would also like to thank Katia, my family and my friends for their constant encouragement.

# Abstract

Electronic devices may be affected by failures, for example due to physical defects. These defects may be introduced during the manufacturing process, as well as during the normal operating life of the device due to aging. How to detect all these defects is not a trivial task, especially in complex systems such as processor cores. Nevertheless, safety-critical applications do not tolerate failures, this is the reason why testing such devices is needed so to guarantee a correct behavior at any time. Moreover, testing is a key parameter for assessing the quality of a manufactured product.

Consolidated testing techniques are based on special Design for Testability (DfT) features added in the original design to facilitate test effectiveness. Design, integration, and usage of the available DfT for testing purposes are fully supported by commercial EDA tools, hence approaches based on DfT are the standard solutions adopted by silicon vendors for testing their devices. Tests exploiting the available DfT such as scan-chains manipulate the internal state of the system, differently to the normal functional mode, passing through unreachable configurations. Alternative solutions that do not violate such functional mode are defined as functional tests.

In microprocessor based systems, functional testing techniques include software-based self-test (SBST), i.e., a piece of software (referred to as test program) which is uploaded in the system available memory and executed, with the purpose of exciting a specific part of the system and observing the effects of possible defects affecting it. SBST has been widely-studies by the research community for years, but its adoption by the industry is quite recent.

My research activities have been mainly focused on the industrial perspective of SBST. The problem of providing an effective development flow and guidelines for integrating SBST in the available operating systems have been tackled and results have been provided on microprocessor based systems for the automotive domain.

Remarkably, new algorithms have been also introduced with respect to state-of-the-art approaches, which can be systematically implemented to enrich SBST suites of test programs for modern microprocessor based systems. The proposed development flow and algorithms are being currently employed in real electronic control units for automotive products.

Moreover, a special hardware infrastructure purposely embedded in modern devices for interconnecting the numerous on-board instruments has been interest of my research as well. This solution is known as reconfigurable scan networks (RSNs) and its practical adoption is growing fast as new standards have been created. Test and diagnosis methodologies have been proposed targeting specific RSN features, aimed at checking whether the reconfigurability of such networks has not been corrupted by defects and, in this case, at identifying the defective elements of the network. The contribution of my work in this field has also been included in the first suite of public-domain benchmark networks.

# Contents

**List of Figures** **xi**

**List of Tables** **xiv**

**Introduction** **1**

Problem Formulation . . . . . 4

**I Software-Based Self-Test Enhancements** **5**

**1 Background** **7**

1.1 Basic Concepts . . . . . 8

1.2 Software-Based Self-Test (SBST) . . . . . 10

1.3 Fault Grading Process for SBST . . . . . 14

1.4 Observability Solutions . . . . . 15

1.4.1 Module-Level . . . . . 16

1.4.2 Processor-Level . . . . . 17

1.4.3 System Bus . . . . . 17

1.4.4 Memory Content . . . . . 19

1.4.5 Performance Counters . . . . . 20

1.5 Generation Flows . . . . . 21

1.5.1 ATPG-based . . . . . 22

1.5.2 Deterministic . . . 22
1.5.3 Feedback-based . . . 23

**2 SBST Algorithms 26**

2.1 Decode Unit . . . 27
2.1.1 Architectural Overview . . . 28
2.1.2 Instruction Set Analysis . . . 29
2.1.3 Signature Mechanisms . . . 31
2.1.4 Proposed Test Strategies . . . 35
2.1.5 Experimental Results . . . 40
2.2 Register Forwarding and Pipeline Interlocking . . . 41
2.2.1 Architectural Overview . . . 42
2.2.2 Proposed Test Strategies . . . 45
2.2.3 Experimental Results . . . 52
2.3 Dual-Issue Processors . . . 53
2.3.1 Architectural Overview . . . 54
2.3.2 Scheduling Issues . . . 56
2.3.3 Duplicated Computational Modules . . . 57
2.3.4 Multi-Port Register File . . . 59
2.3.5 Feed-Forward Paths . . . 65
2.3.6 Pipeline Interlocking . . . 70
2.3.7 Instruction Prefetch Buffer . . . 75
2.3.8 Case Studies . . . 78
2.4 Floating Point Unit . . . 83
2.4.1 Architectural Overview . . . 84
2.4.2 Proposed Test Strategies . . . 85
2.4.3 Experimental Results . . . 88

2.5 Chapter Summary . . . . . 90

**3 Development Flow for On-Line SBST 91**

3.1 On-Line Constraints . . . . . 92

3.2 Execution Management . . . . . 93

3.2.1 Test Encapsulation . . . . . 94

3.2.2 Context Switching to Test Procedure . . . . . 95

3.2.3 Interruption Management and Robustness . . . . . 97

3.3 Development Flow . . . . . 99

3.3.1 Resources Partitioning . . . . . 101

3.3.2 Optimized Test Programs Generation Order . . . . . 102

3.4 Case Studies . . . . . 106

3.4.1 Automotive Microprocessor . . . . . 106

3.4.2 Embedded Floating-Point Unit . . . . . 114

3.4.3 Cumulative Results . . . . . 117

3.5 Chapter Summary . . . . . 119

**4 Summary of Part I 120**

**II Test and Diagnosis of Reconfigurable Scan Networks 122**

**5 Background 124**

5.1 Network Constructs . . . . . 125

5.1.1 IEEE Std 1149.1-2013 . . . . . 125

5.1.2 IEEE Std 1687 . . . . . 127

5.1.3 Example Network . . . . . 129

5.2 Related Works . . . . . 130

5.3 IEEE 1687 Benchmark Networks . . . . . 131

**6 Testing 133**

6.1 Terminology and Fault Model . . . . . 134

6.1.1 Configurations, Vectors, and Test Time . . . . . 134

6.1.2 Fault Model for Reconfigurable Modules . . . . . 137

6.2 Network Representation . . . . . 141

6.2.1 Topology Graph . . . . . 142

6.2.2 Configuration graph . . . . . 143

6.3 Proposed Test Strategies . . . . . 145

6.3.1 Optimal Approach . . . . . 146

6.3.2 Enhanced Version . . . . . 151

6.3.3 Sub-Optimal Approaches . . . . . 155

6.4 Experimental results . . . . . 158

6.4.1 Experiments with Known Benchmarks . . . . . 159

6.4.2 Experiments with Synthesized Benchmarks . . . . . 162

6.5 Chapter Summary . . . . . 165

**7 Diagnosis 166**

7.1 Terminology and Fault Model . . . . . 167

7.1.1 Fault Dictionary and Fault Classes . . . . . 168

7.1.2 Fault Model . . . . . 169

7.2 Proposed Diagnostic Approach . . . . . 171

7.2.1 Diagnostic Analysis . . . . . 171

7.2.2 Generation of New Patterns . . . . . 175

7.3 Experimental Results . . . . . 176

7.4 Chapter Summary . . . . . 179

**8 Summary of Part II 180**

**Conclusions** **181**

List of Research Contributions . . . . . 182

Future Works . . . . . 184

**References** **185**

# List of Figures


1.1 Stuck-at fault and possible test pattern for a sample combinational circuit (a) and sequential circuit (b) . . . . . 8

1.2 Conceptual representation of a sequential circuit including a scan chain 9

1.3 Conceptual representation of a pipelined microprocessor architecture and example of SBST for a stuck-at 0 fault in the Arithmetic Logic Unit (ALU) . . . . . 11

1.4 Fault grading process for SBST . . . . . 14

1.5 Generic system under test: the observation points adopted by the techniques described in the text are highlighted . . . . . 16

1.6 Taxonomy of routine development styles for SBST, taken from [60] 21

1.7 Feedback-based approach based on random code generator . . . . . 24

1.8 Feedback-based approach based on evolutionary engine . . . . . 24

2.1 Example of signature mechanism for ALU instructions . . . . . 33

2.2 Example of signature mechanism for memory instructions . . . . . 33

2.3 Example of signature mechanism for execution flow related instructions 34

2.4 Example of opcodes in the neighborhood of the `add` instruction . . 36

2.5 Test procedure and ISRs for breakpoint instructions . . . . . 39

2.6 Graph of the possible forwarding paths between pipeline stages . . . 43

2.7 The Register Forwarding and Pipeline Interlock unit and its interaction with the processor pipeline and Register File module. . . . . 44

2.8 Test program fragment for testing the MUX for the EXE stage . . . 47
2.9 Vector 0 application to the MUX for the EXE stage . . . . . 49
2.10 Comparator schema and test patterns . . . . . 50
2.11 Test program fragment for testing the CMP in the EXE stage . . . . 51
2.12 Example block diagram of in-order dual issue processors . . . . . 55
2.13 Single-issue (a) and dual-issue (b) versions of a snippet procedure to test adder units. . . . . 58
2.14 Example of dual-issue execution of instructions that access to the register file ports in parallel. . . . . 59
2.15 Effect of data-dependencies on the access to the register file read ports. Wrong implementation (a) and correct version by means of nop instructions (b). . . . . 63
2.16 Example feed-forward paths of in-order dual issue processors . . . . 66
2.17 Example of multiplexer feeding one of the operands of the execute stage. . . . . 67
2.18 Propagation of a test pattern through feed-forward paths . . . . . 68
2.19 Test sequence that applies the pattern 0 of Table 2.6 to $EX_A$ *OP1* MUX 69
2.20 Example of instruction schedule on a dual-issue processor . . . . . 71
2.21 Example of instruction schedule including a multi-cycle instruction 72
2.22 Test sequence that applies the first pattern of Fig. 2.22 to two CMPs involved in data-dependency check . . . . . 73
2.23 Implementation of the test algorithm for the $EX_A$ *OP1* CMP of an example in-order dual-issue processor with 8 registers . . . . . 74
2.24 Dual-issue prefetch buffer . . . . . 76
2.25 Test sequence that propagates a pattern through the prefetch buffer slots . . . . . 77
2.26 Implementation of the pipeline reset in the case studies . . . . . 79
2.27 Example of ATPG loop-based approach applied to FPU . . . . . 85

3.1 Test program encapsulation and loading for execution phase . . . . 94
3.2 Expected and unexpected exception management scenario . . . . . 98
3.3 Sub-module identification and visualization of the coverage figure evolution along the proposed generation steps. . . . . 100
3.4 Proposed test program development order for CPUs organized in levels and branches, and synchronization steps . . . . . 104
3.5 Proposed test program development order for FPUs . . . . . 104

5.1 Concept of Reconfigurable Scan Network (RSN) . . . . . 125
5.2 Excludable TDR segment described in IEEE Std 1149.1-2013. . . . 126
5.3 Selectable TDR segments described in IEEE Std 1149.1-2013 . . . 126
5.4 Segment Insertion Bit (SIB) described in IEEE Std 1687 . . . . . 127
5.5 ScanMux Control Bit (SCB) described in IEEE Std 1687 . . . . . 128
5.6 Example of IEEE Std 1687 reconfigurable scan network. . . . . 129

6.1 Topology graph of the example network in Fig. 5.6. . . . . 144
6.2 Pseudo-code of the optimal approach based on the $A^*$ algorithm. . . 149
6.3 Topology graph of the example network in Fig. 5.6 annotated for heuristic optimization. . . . . 154
6.4 Pseudo-code of the sub-optimal approach based on the depth-first algorithm. . . . . 156
6.5 Pseudo-code of the sub-optimal approach based on the breadth-first algorithm. . . . . 158
6.6 Normal cumulative distribution function (CDF) of the ratio between sub-optimal approaches (depth-first in black, breadth-first in gray) and $A^*$ on the randomly generated networks. . . . . 163
6.7 Progression in time of the maximum fitness value for the evolutionary-based experiments. . . . . 164

7.1 Examples of active path in a RSN and related faulty path lengths . . 174

# List of Tables


1.1 SBST generation techniques comparison . . . . . 25

2.1 Example of instruction format and bitmask . . . . . 30

2.2 Example list of instructions . . . . . 30

2.3 Operand configurations for instructions working on three registers . 38

2.4 Incremental results of the application of the proposed approach . . . 41

2.5 Experimental results on random opcodes . . . . . 41

2.6 Test vectors for a 8-to-1 MUX . . . . . 46

2.7 Input values for the 3-to-1 MUX feeding the first operand input of the EXE stage in a pipelined processor . . . . . 47

2.8 Characteristics of the test program for the RF&PI unit . . . . . 53

2.9 Implementation of the single-issue version of the basic test algorithm for a register file with 8 registers . . . . . 62

2.10 Implementation of the dual-issue version of the basic test algorithm for a register file with 8 registers . . . . . 64

2.11 Fault simulation results on the duplicated computational modules of e200z448 . . . . . 81

2.12 Fault simulation results on the register file of e200z448 . . . . . 81

2.13 Fault simulation results on the feed-forward paths, interlocking logic, and prefetch buffer of e200z448 . . . . . 83

2.14 Experimental results on e200z448 and e200z425 . . . . . 83

2.15 Proposed special operations to detect faults related to erroneous operations . . . 87
2.16 Experimental results on the FPU . . . 89
2.17 Duration and code size of test programs for the embedded FPU . . . 90

3.1 SBST strategies used along the generation process for the automotive microprocessor . . . 108
3.2 Coverage evolution along the development flow for the automotive microprocessor . . . 110
3.3 Number of evaluated test programs, duration, and code size along development flow for the automotive microprocessor . . . 112
3.4 Fault simulation time comparison for approaches without and with synchronization . . . 113
3.5 SBST strategies used along the generation process for the embedded FPU . . . 115
3.6 Coverage evolution along the development flow for the embedded FPU 116
3.7 Number of test programs, duration, and code size for the embedded FPU . . . 117
3.8 Cumulative experimental results on industrial processors . . . 118
3.9 Cumulative experimental results on academic processors . . . 119

5.1 List of possible configurations for the network in Fig. 5.6. . . . 130

6.1 Effect of the functional fault on the ScanMux of Fig. 5.6, which always selects the input 1, when selecting different active paths. . . . 139
6.2 Adjacency matrix of the configuration graph built on network in Fig. 5.6. . . . 145
6.3 Characteristics of the ITC'16 benchmark networks . . . 159
6.4 Experimental results on the ITC'16 benchmark networks . . . 161
6.5 Characteristics of the selected networks . . . 165

6.6 Experimental results on the selected networks . . . . . 165

7.1 Characteristics of the new synthesized networks . . . . . 177

7.2 Test and diagnostic sequences characteristics . . . . . 178

# Introduction

A general definition of *functional testing of microprocessor based systems* is hard to give. I will try to explain the concepts separately and then put everything together. The key concept is testing, which in this thesis is the activity that checks whether a given entity works as expected, according to an ideal model [1]. When referring to a manufactured electronic product, several terms are commonly used, such as *device under test* (DUT) or *unit under test* (UUT) [2]. DUTs in this thesis are systems that include microprocessor cores and the focus of testing is given to physical defects possibly affecting such systems. Testing flows that are currently applied to electronic devices are based on the application of suitable stimuli to some specific test points of the DUT, while other test points are used to perform measurements of the DUT responses to the applied stimuli [3]. Different definitions of the term functional testing have been given, for example that it corresponds to checking whether the device is able to perform the function which has been designed for. Another definition is that, during functional test, only functional input signals of a circuit are stimulated (we will see that special logic is commonly included in the electronic device, e.g., for testing purposes) and only functional output signals are observed [4]. A common definition borrowed from software testing is that functional testing is one type of black box testing, in which functions are tested by feeding them input and examining the output, while internal structure is rarely considered [5]; in this context, only functional models are provided and the evaluation of functional tests requires the definition of new metrics [6, 7]. Finally, functional testing is often referred to as the test applied to the DUT while this is embedded in the overall system [3].

In the field of functional testing of microprocessor based systems, several research aspects are still open, such as concerning the generation of high-quality tests. The quality of a test is defined by well specified metrics and identifies the effectivess of the test in detecting defects. Hence, the more defects a test is able to detect the

higher the test quality. In this thesis, several systematic techniques will be introduced in the first part that are able to produce high-quality functional tests for common microprocessor based systems, while the second part will concentrate on special logic embedded in modern (and next-generation) designs. This logic is purposely inserted in a system to access to on-board instruments, such as sensors, debug-related features, special hardware used for testing, and so on. Other the testing the on-board instruments, the problem of testing the accessing logic itself is still open and is one the major achievements of this thesis.

The first part includes Chapters 1 to 4 and is about a well-known functional testing technique for microprocessor based systems, named software-based self-test (SBST). SBST is a self-test methodology performed by a processor available in the system, which executes a test program stored in an accessible memory. We will see in Chapter 1 the basic concepts and the state-of-the art of SBST. This subject is not new in literature, as many publications can be found proposing heterogeneous generation techniques and targeting aspects such as quality of the test, on-line execution of SBST during the normal system operations, or diagnostic capability of SBST.

SBST presents several advantages when compared to hardware approaches. Being a software-based technique, its development can be done in parallel with the manufacturing process of the hardware and eventual modification to test programs does not affect the hardware, while a modification to the overall system where the DUT is inserted in may affect the development of SBST. To understand the scenario, let us consider the test of a microprocessor. The manufacturing of the microprocessor passes through several phases, starting from the design of a microprocessor model up to the physical manufacturing. SBST is developed based on a model (at a given level of granularity), hence when the needed model has been consolidated, test programs can be generated. However, the test engineer (or the tool) implementing SBST requires the knowledge of the system in which the microprocessor is inserted in, such as the memory map and the attached peripheral cores. Since software operations may interact with such peripherals (or may produce different behaviors by changing the memory map), any modification to such an environment may require changing SBST programs, even by leaving the microprocessor untouched. We will see in Chapter 3 some of these problems, that are even more emphasized for on-line testing, where additional constraints exist due to the coexistence of SBST and operating systems.

Clearly, there are drawbacks that are limiting the adoption of SBST as the standard test solution for end-of-manufacturing, when compared with hardware approaches that are fully supported by commercial EDA tools. The current average capability of detecting defects by SBST is considerably lower and not acceptable to guarantee the expected level of test quality. Despite this, there are specific applications in which this coverage level is acceptable, such as for safety-critical domain, where standard regulations demand for constant monitoring of the system status, intended as faulty or not. These periodic tests should address the most relevant parts on the processor core, allowing in some cases low coverage levels in marginal processor parts. The industrial interest to SBST as a periodical self-test methodology for safety critical applications has moved the scope from the mere academic experience to reality. Although some of the techniques proposed in the literature in the last decades, which were trying to leverage the manual effort required to the test engineer, are valid and demonstrate that the automatic generation of SBST is possible, such techniques are still not applied in industry, where the work of test engineers is based on the personal experience and standardized commercial tools are demanded for quality aspects.

Even though part of my research work has been focused on automatizing the SBST generation, this part of the thesis is more oriented to the work of industrial test engineers, rather than to EDA tool vendors. An effective development flow based on manual processes will be described in detail in Chapter 3, which mainly targets in-field testing and on-line application of SBST. This process is preceded by a set of systematic algorithms presented in Chapter 2, which have been intended to cover special sub-modules existing in modern microprocessor based systems and have outperformed the state-of-the-art methodologies. Finally, some conclusive discussions on the topic will be given in Chapter 4.

The second part of this thesis consists of Chapters 5 to 8 and is focused on test and diagnosis of reconfigurable scan networks (RSNs). A RSN is a kind of infrastructure for interconnecting the numerous instruments that are available in the modern microprocessor based system. Such an infrastructure has the ability of being reconfigured to interact with different sets of instruments, dynamically. More details about RSNs will be presented in Chapter 5, which describes the basic concepts concerning instruments and reconfigurable modules, the main functional operations of RSNs, and the related work on this topic.

Given the recent approval of standards describing the specifications for RSNs, their adoption in electronic designs is growing fast and tools are being developed targeting several aspects related to their design, integration, and usage. Being the topic relatively new, few works can be found in the literature about the test of RSNs. A new functional testing approach will be presented in Chapter 6, based on performing functional operations on the network to excite possible defects, which affect the ability of the network of being dynamically reconfigured. One of the main goals of the presented methodology is to minimize the test time and different approaches will be presented, which are able to find both optimal solutions, highly demanding for CPU time and memory occupation, and sub-optimal, this time using very few resources. Diagnosis of RSNs will be discussed in Chapter 7. The proposed diagnostic algorithm is able to apply functional operations to identify the faulty element of the network. To the best of my knowledge, the presented work is the first one targeting diagnosis of RSNs. Finally, Chapter 8 will give some conclusions on this topic. Contrarily to the first part, the algorithms presented for RSNs are mainly oriented to EDA tool vendors.

# Problem Formulation

This thesis deals with the enhancement of state-of-the art SBST techniques and with the test and diagnosis of RSNs.

Concerning SBST, the research question this thesis is trying to answer is whether SBST can be successfully adopted for the test of real-world processor based systems. This means providing algorithms and techniques that a test engineer can systematic implement for the specific DUT with reasonable effort. Moreover, when dealing with real-world processors, the problem of integrating SBST in the on-board operating system represents a further challenge.

Concerning RSNs, at time of writing there are very few works existing in literature about testing, while contributions about diagnosis are missing. The research question tackled by this thesis is whether RSNs can be systematically tested, once the functional description is known and, in this case, in the minimum amount of time. Moreover, whether it is possible to diagnose which is the faulty part of a specific RSN, systematically and in a reasonable time.

# Part I

# Software-Based Self-Test Enhancements

# Chapter 1

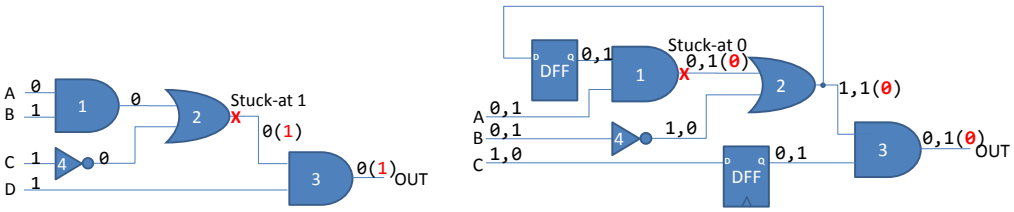
# Background

The use of microprocessor based systems in safety- and mission-critical applications, calls for total system dependability. This requirement translates in a series of system audit processes to be applied throughout the product lifecycle. Nowadays, some of these processes are common in industrial design and manufacturing flows. These include risk analysis, design verification and validation, performed since the early phases of product development, as well as various test operations both during and at the end of manufacturing or assembly steps. Increasingly often, test operations need to be applied during the product's mission life, such as periodic on-line testing and concurrent error detection. The reliability requirements are met by trading off test quality with admissible implementation costs of the selected solutions.

This chapter briefly introduces the topic of Software-Based Selft-Test, which is a consolidated technique for the functional test of microprocessor based systems, together with basic concepts needed for the understanding of this thesis.

The rest of this chapter is structured as follows. Section 1.1 presents the basic terms and concepts. Section 1.2 gives a definition of SBST and discusses the related work. The methodology used to evaluate of the SBST fault coverage is briefly described in Section 1.3. Section 1.4 discusses about possible observability solutions used to evaluate SBST. The chapter is concluded in Section 1.5 by a discussion of the most used SBST generation techniques.

Parts of this chapter have been published in [17], concerning observability solutions for SBST.




(a) Combinational (b) Sequential

Fig. 1.1 Stuck-at fault and possible test pattern for a sample combinational circuit (a) and sequential circuit (b)

## 1.1 Basic Concepts

The purpose of *Testing* is to identify defective products. When electronic circuits are considered, it consists in the application of suitable patterns to certain test points of the actual circuit, such as the input ports, while other points, such as the output ports, are sampled and compared to expected values, that are computed on an ideal device. If there is a mismatch between measured and expected values, a certain defect affecting the device has been identified [3].

Generating test patterns for assessing the correctness of the structure of a circuit is a critical task. In order to make it feasible and to quantify the efficacy of a test, the possible circuital defects are mathematically modeled as *Faults* [1]. Testing usually refers to faults instead of defects, since the first can be enumerated. Fault coverage is defined as the fraction of faults detected (or covered) by a set of test patterns over all possible faults according to a fault model.

Test pattern generation for combinational circuits is quite straightforward. The pattern has to activate (or excite) the fault at its location, which means to introduce a logical difference between the fault-free machine and the one affected by that fault. Then, it has to propagate the fault effect to an observable point. An example of test pattern for a stuck-at-1 fault affecting a combinational circuit is shown in Fig. 1.1a. In the example, the test pattern excites the stuck-at-1 fault by forcing the value of the faulty location to be 0 and propagates the fault effect up to the OUT signal. The problem of test pattern generation for sequential circuits is more complex, due to the presence of memory elements (e.g., flip-flops). In this scenario, additional effort is needed to move the fault effect towards an observable point. In the example shown in Fig. 1.1b, the first pattern applied to the primary inputs of the sequential circuit loads

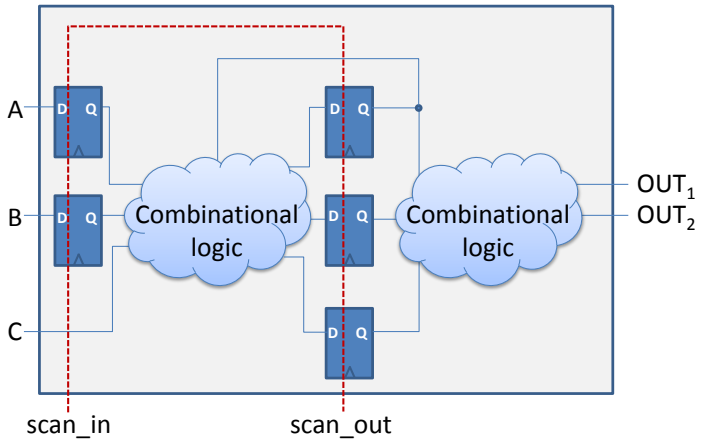



Fig. 1.2 Conceptual representation of a sequential circuit including a scan chain

the flip-flops with suitable values; then, in the next clock cycle, the fault excitation and propagation are performed by both the second pattern and the values previously stored in flip-flops.

Automated algorithms for test pattern generation are widely known and employed in industry. This technique is defined as Automatic Test Pattern Generation (ATPG). The complexity of the ATPG grows exponentially with the sequential depth of the circuit.

To circumvent the complexity of sequential test pattern generation, which is the case of microprocessor based systems, hardware solutions are applied to the original design, specifically devised for testing purposes. Such solutions are traditionally known as Design-for-Testability or DfT. A de-facto standard DfT solution implemented in all currently available industrial microprocessors is the inclusion of one or more scan chains. A scan chain is a serial connection of the available flip-flops in a circuit and is accessible by additional dedicated pins (see Fig. 1.2). By serially shifting test patterns through the scan chains, the testing problem is restricted to the complexity of the combinational logic, thus the efficiency of the ATPG is dramatically increased. Other DfT solutions are based on Logic Built-In Self-Test or Logic BIST, which are able to apply stimuli to the primary inputs of the device under test and to compact the primary output values into a so-called *test signature* [3].

Testing solutions exploiting the available DfT such as scan-chains may present some problems: firstly, they are poor in detecting certain faults (e.g., dynamic faults related to timing issues) since DfT tests are performed at low frequencies;

additionally, they manipulate the internal state of the system, differently to the normal functional mode, passing through unreachable configurations. When the testing is performed at the nominal frequency of the circuit, without using special hardware not guaranteeing its functional mode of operation, then it is said to be *functional*. This means, for example, that a functional test does not make use of scan-chains and is typically used for in-field testing, when an excessive stress due to unreachable configurations is neglected or scan chains are not accessible anymore.

## 1.2 Software-Based Self-Test (SBST)

The term Software-Based Self-Test (SBST) was first proposed by Chen and Dey in [18], but the approach itself has been proposed few years before under the name *Native Mode Functional Test* in [19, 20]. SBST broadly identifies all test methodologies based on forcing a microprocessor/microcontroller to execute a program and checking the results to detect the presence of possible defects affecting the hardware. Indeed, the pioneering idea of testing a microprocessor with a program dates back to 1980. In [21], Thatte and Abraham devised fault models and procedures for building test programs able to detect permanent defects in different functional units of a simple processor. A wide adoption of their methodology was hindered by the difficulties in automating the generation of such test programs, especially when targeting complex processors.

The basic concept of SBST is graphically depicted in Fig. 1.3. In the figure, a stuck-at-0 fault affecting the ALU is excited by a specifically crafted sequence of instructions and by suitable data. When the sequence of instructions is executed, the result of the arithmetic computation is stored in the data memory. The contents of the memory are then compared with the expected values, hereinafter referred to as *golden* references. Alternative ways to observe the fault effect are discussed in Section 1.4.

In general, the usage of SBST requires:

1. Generating a suitable test program. This is typically a hard job, which is still mainly performed by hand. Moreover, the complexity and effectiveness of this task depends on the adopted metric, which in turns depends on the available information: in some cases, both RTL and gate-level models of the

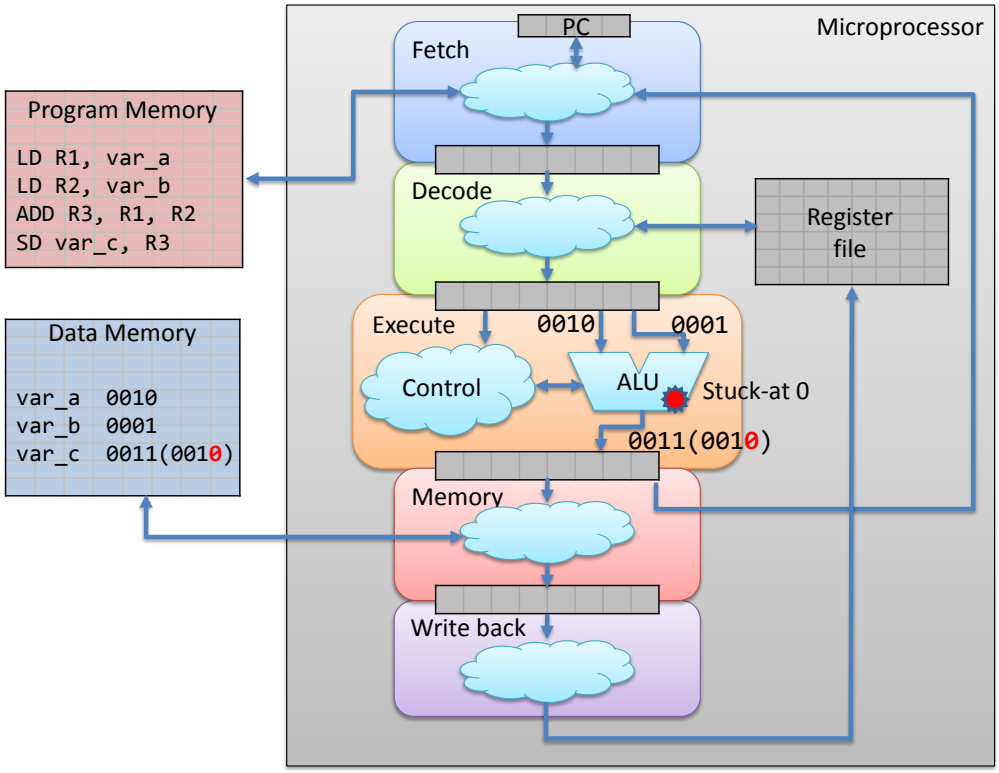



Fig. 1.3 Conceptual representation of a pipelined microprocessor architecture and example of SBST for a stuck-at 0 fault in the Arithmetic Logic Unit (ALU)

target system are available, while in others functional information is available, only. When the gate-level netlist is available, then it is possible to compute the fault coverage achieved by the generated test program with respect to the most common structural fault models (e.g., stuck-at). Details about the most common generation techniques are given in Section 1.5.

2. Creating an environment to support its execution. Once the test program is available, it must be stored in some memory accessible by the processor. The processor must be triggered to execute the test program at the due time. Finally, the results produced by the processor during its execution must be observed. More details are given in Section 1.3.

Nowadays, the complexity of processors has significantly increased; the micro-architectural details play a fundamental role, and devices cannot be accurately modeled using information about the Instruction Set Architecture (ISA) alone. However, SBST is getting more and more important: it commonly supplements other kinds of tests, as functional programs may detect unmodeled defects that escape

traditional structural tests (the so-called "collateral coverage" [22]). By definition, the functional approach tests the system in its operational mode, without activating a test mode and without reconfiguring the system; hence, it is guaranteed not to cause overtesting (or overkilling). Moreover, several producers exploit functional stimuli to validate their design or to run post- silicon verification.

In some cases, test programs are generated pseudo-randomly [23], possibly using simulation feedback, and may even use some hardware support to make the test phase more efficient [24].

Among the several recent works focusing on SBST, some aim at developing algorithms to generate effective test programs for common modules starting from the knowledge of its ISA alone (e.g., for an OpenSPARC T1 processor [25] or a MIPS-like ISA [26]), eventually combined with RTL description (e.g., [27] for two different implementations of the MIPS ISA).

Several works focus on the possible automation of the test program generation procedure. Details on some of the most used generation techniques are given in Section 1.5. The work in [28] proposes a genetic-algorithm-based evolution framework that enables small test programs to evolve using a LEON2 processor. High-level decision diagrams (HLDDs) are used by the authors of [29]. The work in [30] applies bounded model checking tools on a global extended finite state machine (EFSM) model of the processor under test. SAT-based test program generation is also proposed by several other works [31, 32], including a personal collaborative work [33], which is able to derive test programs from the gate-level microprocessor description. The SAT-based framework of such work is described in details in [34] and not included in this thesis. Other works reuses existing procedures to automatically generate effective test programs. For example, [35] remaps generic test program on a target processor, as experimented in a VLIW processor, while [36] uses the execution trace collected during executing training programs on the processor under test to create constraints for the test program generation.

The possible usage of SBST for diagnostic purposes has also been explored, e.g., in [37] using an 8-bit accumulator-based microprocessor, and [38] for transition-delay faults in an i8051-compatible microcontroller. The work in [39] is able to produce diagnostic test programs, which specifically target for faults that can be handled with an available self-repair method. The same authors described how to integrate such programs into a superscalar processor [40]. A personal collaborative work [41]

tackled the automatic generation of test programs with diagnostic capabilities and provides results gathered on a MIPS-like processor. Also in this case, the work is not further discussed in this thesis and details can be found in [41].

Finally, a number of works study how to apply SBST for in-field test (e.g., [42] for a MIPS architecture processor).

In this last domain, regulations and standards mandate the adoption of effective solutions to early detect permanent faults, and SBST has the big advantage that it does not require access to any systematic DfT solution, whose usage details are often considered as proprietary by the manufacturer. SBST can be used not only to test the CPU, but also the other components in a microcontroller or SoC: for example, several works addressed its adoption for the test of peripherals [43], memories [44] (possibly implementing transparent test [45]), and cache memories [46, 47]. SBST usage can also be extended to the test of multi-core systems [48]. In some cases, the usage of existing hardware resources introduced for non-test-related purposes (e.g., for debug, design validation, performance assessment) allows significantly reducing the size and duration of SBST test programs [49].

Moreover, SBST can more easily match the constraints of the environment where the processor is employed. When adopted for in-field test, SBST typically means activating a test program either at the system power-on, or during the application idle times. In the latter case, additional constraints about the duration of the test exist, due to the limited duration of the available time slots. Unfortunately, the constraints posed by the application environment may severely impair the effectiveness of the method when applied to test a processor. When functional test is used for end-of-manufacturing test, processor inputs and outputs can often be fully controlled and observed by an *automatic test equipment* (ATE). Nevertheless, during in-field test some parts of the processor (e.g., the test and debug structures) might not be accessible by the test procedure, thus resulting in untestable faults [50], i.e., faults for which no input stimuli exists, able to detect them. In other words, some parts of a processor which are not used anymore during the operational phase may contain faults, which cannot be tested in this phase, but are also guaranteed not to affect the system behavior. Besides, not all the processor inputs may be freely controllable in the in-field scenario: for example, activating the reset signal is hardly possible, thus preventing the test of the reset logic. More in general, possible Control/Status input signals coming from other devices may be hard to control [20]. Finally, observability

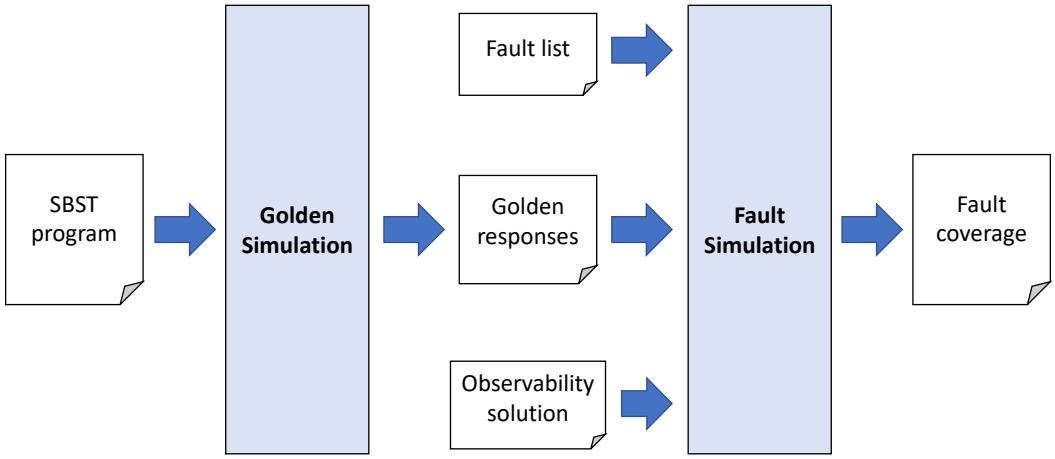



Fig. 1.4 Fault grading process for SBST

during SBST in-field test could be limited, since only the produced results (e.g., in memory) can be observed. The set of faults which cannot be tested in the in-field environment due to these additional constraints are known as *functionally untestable faults* [50]. As previously mentioned, it is important to be able to identify untestable faults, since they limit the achievable fault coverage.

## 1.3 Fault Grading Process for SBST

Fault grading corresponds to the task of measuring fault coverage of a certain SBST procedure. This process, that evaluate the effectiveness of SBST programs, passes through two main steps: the golden simulation and the fault simulation (see Fig. 1.4).

In microprocessor based systems, all the environment concurs to the test program execution. The environment includes the microprocessor itself, together with accessible peripheral cores and memory modules. A proper testbench has to be able to control such a system, load a compiled program in th available memory, and finally forces the microprocessor to executes the program.

A logic (golden) simulation of the testbench uses the SBST program to collect the golden run responses, e.g., the values of the microprocessor primary output ports, or the memory content at the end of the test program.

The fault simulation uses the golden run responses and compare them with the faulty circuit responses, according to a given observability solution (refer to Section 1.4). The fault list used in fault simulation contains all faulty locations to be checked in the gate-level netlist of the system.

At the end of the grading process a fault coverage report is produced. Further analysis on the circuit can be performed to show effective test coverage not taking into account functionally untestable faults. As an example, some scan signals may become faulty but never affect the in-field SoC behavior, or modules used to perform software and silicon debug will be no more used during the mission, thus may be skipped during the grading process. Details about how to identify part of these faults are given in [50]. In generation approaches based on formal methods, it is possible to prove that a given fault is functionally untestable during test generation, as shown in a collaborative work based on SAT solvers [34].

More details on a fault grading process for dependable automotive applications are presented in [51].

## 1.4 Observability Solutions

In the following, the main solutions that can be adopted to observe the effects of possible faults during the SBST testing of a processor-based system are described, namely: module-level, processor-level, system bus, memory content, and performance counters. The above solutions are referred to bare metal systems, i.e., solutions that would require the presence of an Operating System (e.g., based on monitoring its performance, or analyzing the event logs) are not included.

The assumption is that the targeted faults are those inside a given module within the processor. For every solution, the adopted mechanism as well as the main advantages and disadvantages are detailed, and a preliminary analysis about the forecasted coverage is reported. Fig. 1.5 graphically summarizes the considered solutions.

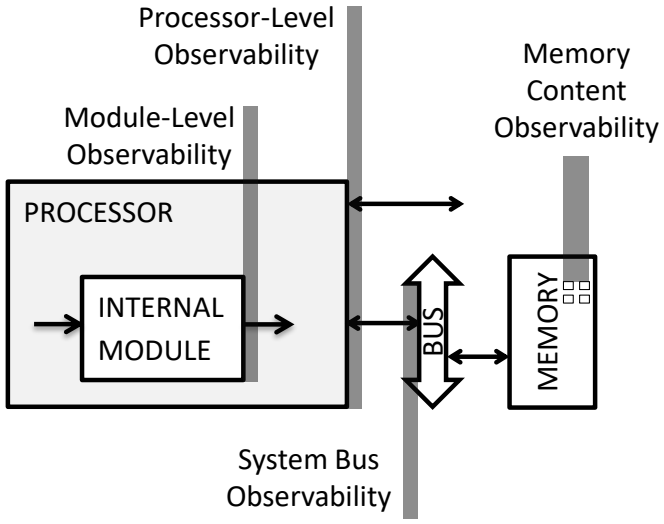



Fig. 1.5 Generic system under test: the observation points adopted by the techniques described in the text are highlighted

### 1.4.1 Module-Level

When a generic module inside the processor is considered for the test, the *ideal* level of observability is the boundary of such a module, i.e., it is assumed that all the output ports are available for observation.

The test program is supposed to be able to properly stimulate the input ports of the considered module, to excite the faults and to propagate them towards the module output ports, which are test observation points.

This observability approach can be adopted during simulation and fault simulation processes. However, when working on real chips, for a number of reasons it is hardly feasible neither in-field nor, in most cases, at end-of-manufacturing. These reasons are, firstly, that the module output ports usually do not coincide with the circuit pinout, and even if they do, it is normally not affordable to continuously observe the circuit behavior without resorting to additional hardware; secondly, when an instrument is attached to the observation points, the observed signals can only be read in test mode through a dedicated tester.

Therefore, this observability solution is introduced here only as a reference, because it establishes an upper bound to the fault coverage results obtainable in simulation through SBST test approaches.

### 1.4.2 Processor-Level

This solution assumes that fault effects can be observed at the processor level, i.e., that all the processor outputs can be continuously monitored. While module-level observability is very specific and may be not feasible in practice, observability at processor-level represents one of the scenarios that are sometimes adopted for end-of-manufacturing test. Considering an internal module which has to be tested, the test program must not only propagate the fault effects up to the module output ports, but must be able to propagate them also to the processor output ports.

According to these considerations, the observability we can get with this solution is generally lower than the one obtained at module-level. In most of the cases, propagating the faulty behavior requires an additional effort in order to reach the processor outputs. In the case of a functional testing approach based on test programs, this additional effort may imply the addition in the code of specific instructions able to propagate the fault effects to the processor outputs. As an example, faults within an arithmetic unit can be easily activated by executing suitable arithmetic instructions (thus propagating their effects on the module outputs), and can then be made observable on the processor outputs via Store instructions that propagate the result of the arithmetic operation up to the processor output ports.

Faults may also exist that, even with the addition of instructions, cannot be observed on the processor outputs. This situation may happen when the processor design includes some redundant circuitry, for example left from previous releases or included for future extensions of the design. Clearly, the related faults can be classified as untestable. However, the identification of untestable faults may often represent a relevant problem.

Due to the need of constant monitoring of all the processor outputs, this observability solution requires the use of an ATE and thus, cannot be adopted by in-field SBST.

### 1.4.3 System Bus

This solution mandates that the control, data and address signals of the system bus are continuously monitored. When comparing this solution with the previous, all the processor outputs not related with the system bus are excluded from observation.

End-of-manufacturing scenarios may offer a high-level of observability, when the constant monitoring of the output ports of the processor is possible. Such a powerful scenario is not representative of an observation mechanism for in-field testing. However, more and more processors (especially for embedded systems) are equipped with specific components in charge of monitoring the interconnections between the processor and the memory subsystem, in some cases including external caches. Examples of such modules are MISRs attached to the bus, that update a signature every time new data are going to be written to the memory. This solution has been adopted by commercial microcontrollers, e.g., from Freescale [52]. In other cases, dedicated programmable embedded cores are in charge of tracing specific bus transactions (e.g., ARM HTM [53]) and of storing a history of processor execution in a local memory, which is accessible through a dedicated port (e.g., for debug purposes). An example of IP core specially devised for SoC testing is described and demonstrated for a processor compliant with the SPARC v8 architecture in [54]. The presence of caches significantly limits the amount of data flowing through the bus, and hence the number of faults whose effects can be observed by observing it.

SBST programs using this observability solution should include specific sequences of instructions that permit the propagation of the fault effects up to the system bus.

As an example, the faults affecting the circuitry for supporting an external coprocessor are considered. If the external coprocessor is connected to the processor with dedicated ports, the effects of such faults –propagated up to this interface in the original program– need to be read back from the coprocessor and stored to the system memory in order to become observable in the system bus. This solution adds complexity to the test program and is not always feasible. In case of faults whose effects can only be observed on non-functional output signals and never read back, the fault coverage reduction cannot be recovered, thus resulting in a potentially less-effective observation mechanism in general.

Also, if an SBST program developed for processor-level observability is evaluated with this observability solution, a significant fault coverage reduction could be observed. This fault coverage loss is motivated by the reduction of the observed signals, as they are a subset of the output signals of the processor.

### 1.4.4 Memory Content

According to this solution, a fault is marked as detected if the content of the system memory is different than the expected one at the end of the execution of the SBST program.

All the previously presented observation mechanisms rely on the fact that some output ports of the circuit can be constantly monitored, e.g., by a dedicated tester which is physically connected to test points or to the interface with on-board instruments. This is not the usual case of SBST in general. In a manufacturing at-speed SBST scenario, the functional program is often uploaded in the system memory (e.g., a cache, or a dedicated flash) and run at-speed, storing its responses in some available memory elements, such as internal registers, caches, or main memory, and hence permitting a low-cost tester to access them at the end of the execution. Similarly, during in-field SBST, at the end of the test program run the processor itself or another module (e.g., another processor) may perform an access to the specific memory cells in order to compare their values with the expected ones.

According to the presented scenario, this observation mechanism assumes that the test program collects in some way the information about test results and saves this information in the system memory. The information collected by the test program may be compacted by the test program itself and then (at the end of the test process) saved in few selected memory cells. Alternatively, the information saved by the program may be written to a set of memory cells, according to the targeted module characteristics as described in [55] for a MIPS-like processor and an industrial System-on-Chip.

Since the test results correspond to the values generated by the test program, which are checked only at the end of the test program execution without taking into account when these results are produced, some performance faults may escape when using this observability mechanism. For example, in the case of Branch Prediction Units, some performance faults may not modify the final test program results, but only delay the actual execution time [56], e.g., by turning a correctly predicted branch into a mispredicted one.

### 1.4.5 Performance Counters

Performance Counters (PeCs) measure the number of occurrences of different internal events, making their observation easier from the outside. They exist in many processors, mainly for design validation, performance evaluation and to support silicon debug. Their values can normally be accessed via software. Hence, a test program may read the value of a given PeC, execute a sequence of instructions exercising a given module, and then read again the value of the PeC comparing it to the expected one. Possible differences may allow the detection of faults inside the module.

The most common types of PeCs include those that count internal events related to:

- caches, counting the number of miss and hit events;
- Branch Prediction Units (BPUs), counting the number of correctly or incorrectly predicted branches;
- pipeline stages, counting the different types of stalls;
- Memory Management Units (MMUs), counting the number of hit/miss accesses to the Translation Lookaside Buffer (TLB);
- exception units, counting the number of triggered exceptions, often divided by type.
- bus interfaces, counting the number of performed bus transfers, also often divided by type.

These counters are already quite common in general-purpose high performance processors, and their adoption is growing in microcontrollers for embedded applications.

The usage of these counters as part of the observability mechanism adopted by a testing procedure was proposed in several works, such as [56, 26] that use variants of the MIPS architecture, or [57, 58] working with the OpenSPARC T1 processor. The PeCs have also been proposed as feedback in automatic test programs generation [59] and in [47]to simplify the test programs aimed at detecting faults in caches. They are crucial for the detection of some specific types of faults, such as performance faults. Moreover, they can facilitate the test of faults belonging to some modules, such as Branch Prediction Units, Cache Controllers, TLBs. They may also be used

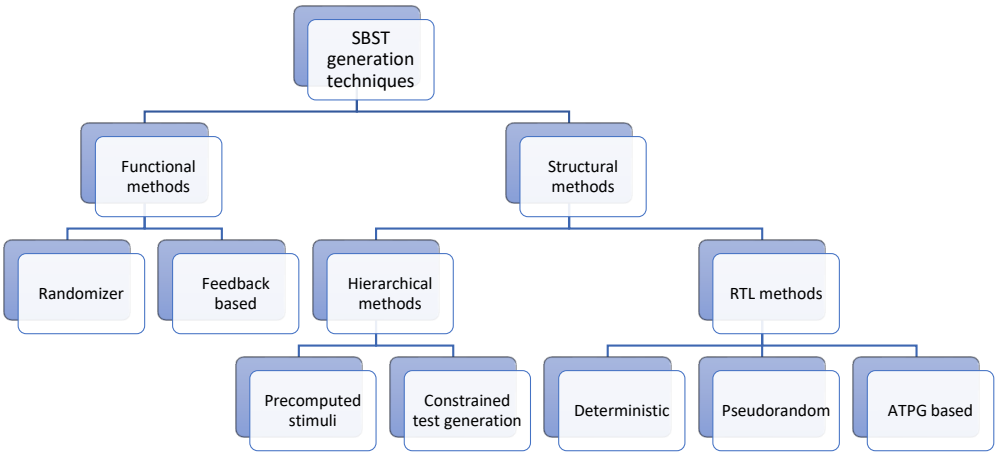



Fig. 1.6 Taxonomy of routine development styles for SBST, taken from [60]

to support the test of specific modules within the pipeline, such as those controlling the activation of stalls.

Regarding observability issues, the PeCs may provide deeper details on internal events affecting the module that may not reach the output ports, and allow the detection of several performance faults. Thus, exploitation of PeCs and propagation of performance values to system memory increases observability and may represent a valuable solution during in-field SBST.

## 1.5 Generation Flows

This section presents some of the most common methodologies for the development of SBST programs. As a reference, the nomenclature presented in [60] is used. The main taxonomy of different development styles is also shown in Fig. 1.6. The authors includes all possible generation techniques into two main categories:

**Functional methods** that exploit only functional information, such as the microprocessor ISA. They include both methods that rely on *code randomizers*, sometimes guided with suitable constraints, and methods that adopt a *feedback-based* strategy, meaning they evaluate generated test programs according to suitable metrics (often computed through simulation) and try to progressively improve them.

**Structural methods** that uses also structural information, such as gate-level are RTL descriptions. These methods are further divided into two categories: *hierarchical* approaches, that focus on a processor's module one at a time, generating stimuli for each module and then extending those stimuli to the processor level; and *RTL* approaches, that exploits structural RTL information along with the ISA information to generate instruction sequence templates for justifying and propagating faults of the module under test.

Above all the methodologies listed in Fig. 1.6, in the following three representative techniques are discussed in details.

### 1.5.1 ATPG-based

This methodology guarantees the highest possible coverage, given the fact that test patterns for a specific module are automatically generated by means of an ATPG engine. Especially for combinational blocks, ATPG is able to reach 100% test coverage, while it may be harder (in terms of computational effort) for complex sequential circuits.

These patterns are intended to be applied to the primary input signals of the block under test and are usually provided in text format (e.g., STIL). Further work is needed to parse the text file and transform the signal values to a sequence of assembly instructions. Such step is trivial for simple blocks, but can be more challenging in other cases, where selection signals have to be properly interpreted. This requires the knowledge of the processor's microarchitecture.

A few coverage loss is typically observed when transforming ATPG patterns to SBST, due to masking effects and aliasing during the signature computation.

*Effort*: medium-low (minutes to hours), depending the functional block complexity. In the best case, it requires a single fault simulation.

### 1.5.2 Deterministic

It consists in the implementation of a documented algorithm or methodology. This is a functional methodology, which does not highly depend on the internal structure of

the module under test. Some new deterministic algorithm targeting specific modules of modern microprocessors are presented in Chapter 2.

The expected fault coverage level is medium to high. However, the resulting program can be highly redundant in terms of test patterns applied to the module under test. This redundancy is mainly present to effectively adapt the program to the whichever implementation of the module.

These kind of techniques are really useful when the netlist is missing or not hierarchical (e.g., flattened, obfuscated). Moreover, they are suitable for such modules that are too complex for ATPG-based methods.

*Effort*: low (minutes to hours). In some cases, it requires several adjustments and fault simulations, until an appropriate level of fault coverage is reached.

### 1.5.3 Feedback-based

This methodology is based on an iterative generation, in which the fault-coverage of the current test program is used as a feedback for the next program generation.

A sample scheme resorting on random code generator is graphically depicted in Fig. 1.7. According to this scheme, a test program is incrementally generated, by adding new instructions at each iteration. Every time new instructions are added to the test program, a fault simulation is performed. If new faults are not detected by the test program, then the added instruction are removed by the test program and new ones are generated in the following loop.

An alternative scheme is based on an evolutionary engine to continuously generate test programs, as shown in Fig. 1.8. The effectiveness of this technique depends both on the number of test programs generated and the skills of the test engineer, which has to guide the evolutionary engine. Briefly, the test engineer has to build the basic blocks of the test programs, where some parameters such as register contents are unknown. The evolutionary engine starts by assigning random values to such parameters. Later on, after the fault coverage level of the generated programs is evaluated, the engine is able to combine previously generated programs and to generate new ones [61].

The fault coverage can be very high, but it requires that many programs are generated. This technique is highly useful for distributed parts of the processor, for

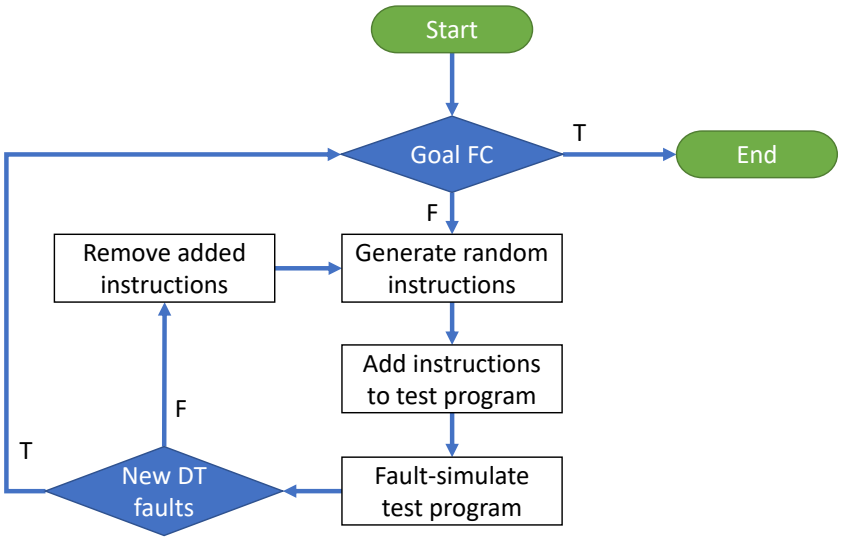



Fig. 1.7 Feedback-based approach based on random code generator

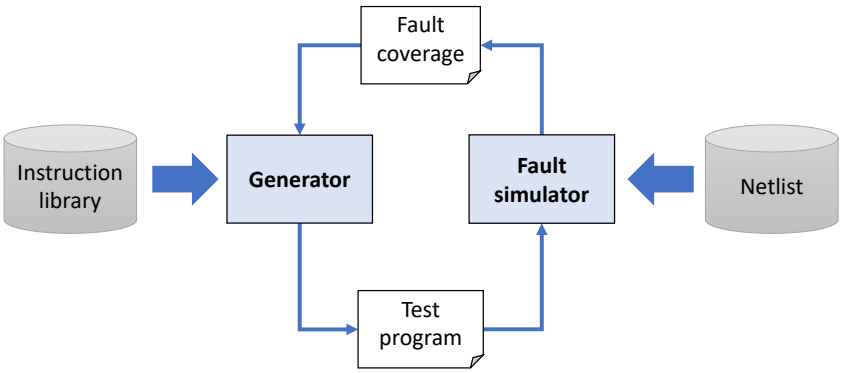



Fig. 1.8 Feedback-based approach based on evolutionary engine

which is quite difficult to group the logic into a well-defined module. In this case, the fault list is enough to guide the evolution. Even for flattened or obfuscated circuits, it can be used to cover the corner cases as the last generation process (i.e., after deterministic approaches).

*Effort*: medium-high (hours to days). Since each program is fault simulated, the time required for a single fault simulation is a key parameter for the effectiveness of this method. The preliminary steps of preparing the program skeletons require additional effort, but typically this is not very high.

Table 1.1 SBST generation techniques comparison

| Technique | #lines | #clock cycles | SA FC% |
|---|---|---|---|
| ATPG-based | 110/708 | 3,549 | 98.0 |
| Deterministic | 42/533 | 41,326 | 90.1 |
| Feedback-based | 164/188 | 1,651 | 91.9 |

Table 1.1 provides the results in terms of number of code lines, clock cycles required to execute the programs, and stuck-at (SA) fault coverage while using the described techniques, targeting the shifter module included in the ALU of an industrial processor core. The shifter counts with 4,196 gates. The column labeled as *#lines* provides two values (A/B) showing the number of lines of the obtained test program. The A and B values count the number of lines of a loop-based version of the final program and its unrolled version, respectively.

# Chapter 2

# SBST Algorithms

Test programs included in a SBST test suite for microprocessors are typically developed following heterogeneous approaches. Although an automatic way to generate them would be the dream of every test engineer, similarly to the ATPG for test patterns, at the moment most of the work is done by manually implement systematic test algorithm on the target processor, especially in industry. This is mainly due to the lack of automatic SBST support by commercial EDA tools.

A negative aspect of these approaches is that the effectiveness of manually developed test programs is highly influenced by the skills of the test engineers and the availability of test algorithms for different parts of the processor. The more the processors become complex, the harder it is to reach high fault-coverage levels especially on those modules that are not directly controllable by single assembly instructions.

This chapter purposely tackles the testing of particular sub-modules of modern processors that are deeply embedded in the pipeline and whose faults are historically considered hard to test.

In Section 2.1 the focus is given to the Decode Unit, which in RISC processors is responsible for decoding each instruction fetched from the program memory and generating the control signals needed by the other units.

Section 2.2 covers faults on hardware modules responsible for resolving data hazards. A data hazard is the situation in which the result of a previous instruction is needed in the pipeline before it is available. Hardware modules devoted to

resolve such a situation implement the Register Forwarding and Pipeline Interlock mechanisms.

Section 2.3 deals with superscalar processors that are able to issue multiple instructions at a time. A set of rules and strategies are given for the re-design of traditional SBST techniques to cope with multi-issue aspects. Although the presented algorithms specifically target dual-issue processors, a test engineer can easily adapt them to complex in-order multi-issue processors.

Finally, a classical computational module such as the Floating-Point Unit (FPU) is analyzed in Section 2.4. Although the FPU differ from the previous modules in terms of controllability by means of specific assembly instructions, its test is not a trivial task. Specific strategies are shown that are able to cover possible test escapes of state-of-the-art techniques.

Parts of this chapter have been also published in [9] (about Register Forwarding and Pipeline Interlocking), [8] (about Decode Unit), and [11] (about FPU).

The presented algorithms are examples of *free* test program generation, which does not take into account online testing contraints, such as test duration and memory occupation of the test suite. Details about those aspects are analyzed in Chapter 3.

Experimental results based on fault-simulation are given in each section for each of the presented algorithms. Results refer to both academic and industrial processors manufactured by STMicroelectronics.

## 2.1 Decode Unit

In this section, a functional methodology to test common decode units by means of SBST is presented. The method uses the instruction set architecture (ISA) of the target processor and its pipeline functional behavior as described in the user manual, only. The physical implementation of the processor is only used for evaluating the fault coverage of the proposed methodology on the actual processor netlist.

Details about the proposed methods are given in the following subsections. A brief architectural overview is given in Section 2.1.1. The overall approach starts analyzing the instruction set and dividing the list of assembly instructions into homogeneous groups, as presented in Section 2.1.2. Depending on the class of

instructions, Section 2.1.3 shows different implementations of *signature* mechanisms, i.e., the way to observe the effect of faults affecting the decode unit after the target instruction has been decoded. Such signature mechanisms have to be included in the synthesized SBST procedures, which are specific of each group of instructions, as detailed in Section 2.1.4. Finally, experimental results are presented in Section 2.1.5.

### 2.1.1 Architectural Overview

In a typical pipelined RISC processor, the instruction Fetch Unit fetches instructions from memory and provide them to the Decode Unit. Usually, instruction buffers are placed in between these two units for performance purposes. The decode unit interprets bits in the operational code (*opcode*) of each instruction and generates the control signals for further units in the processor pipeline. In details, the decoding logic performs the following functions:

- opcode decoding to determine the instruction class and resource requirements for each instruction being decoded;
- source and destination register dependency checking;
- execution unit assignment;
- determine and decode instruction serialization, and inhibit subsequent instruction decoding.

In case of multi-issue processors (i.e., in which a pipeline stage is able to process multiple instructions at the time), additional dependency checking among different issues is performed [62, 63]. In processors such as ARM Cortex A-Series instruction decode unit also performs register renaming to facilitate out-of-order execution by removing Write-After-Write (WAW) and Write-After-Read (WAR) hazards.

From the architectural point of view, the decode unit is composed of combinational logic. It implements a table which is addressed by the instruction opcode together with some configuration and status bits of the processor. Numerous signals are provided by this module to the later pipeline stages.

### 2.1.2 Instruction Set Analysis

This step aims to create groups of similar assembly instructions according to the analysis of the processor user manuals. The needed information related to the decode stage that has to be extracted is the following:

- the list of all assembly instructions, their instruction format and operands (e.g., affected registers);
- the mechanisms to enable/disable different features of the processor (e.g., caches, FPU, high-privileges instructions);
- the mechanisms to switch between multiple ISAs (e.g., between Power ISA and VLE for Power Architecture) or to enable extensions (e.g., Thumb for ARM).

The collected data are used to group together instructions according to their functional category, instruction opcode format, and sensibility to control and status bits.

#### Functional Category

Each assembly instruction is included into a group corresponding to a specific functional category, such as:

**ALU:** Instructions of this kind make computations over registers and eventually modify specific status bits of the processor; examples are multiply, shift, and bitwise operations.

**Memory:** This category includes Load and Store instructions.

**Execution flow:** All such instructions that modify the sequential execution flow, such as branches, calls to subroutines, and interrupts-related operations (e.g., system call).

**Processor control:** Instructions that act on special-purpose registers.

By having instructions grouped in this way, an appropriate signature mechanism can be associated to each of the groups.

Table 2.1 Example of instruction format and bitmask

| | Opcode bits | Operand rD | Operand rA | Operand rB | Opcode bits |
|---|---|---|---|---|---|
| | 0 5 | 6 10 | 11 15 | 16 20 | 21 31 |
| Instruction | 011111 | 00011 | 00001 | 00010 | 01000010100 |
| Bitmask | 111111 | 00000 | 00000 | 00000 | 11111111111 |

Table 2.2 Example list of instructions

| Instruction | Opcode | Bitmask | Operands |
|---|---|---|---|
| add | 7C000214 | FC0007FF | *rD*, *rA*, *rB* |
| and | 7C000038 | FC0007FF | *rD*, *rA*, *rB* |
| andi | 70000000 | FC000000 | *rD*, *rA*, *SIMM* |

### Instruction Format

The format of each assembly instruction is reported in the processor ISA. The instruction format determines which of the instruction bits are the operands and which are the fixed values that determine the specific instruction opcode.

A bitmask can be used to discriminate between operand and fixed opcode bits, as in the example shown in Table 2.1. The example represents a bitmask for a set of instructions operating on three registers: *rA* and *rB* are the source registers, while *rD* is the destination register. The bitmask contains the value 1 in the bits 0-5 and 21-31 to indicate the part reserved for the opcode, and the value 0 in the bits 6-20, that are associated to operands.

By following the same criteria, a bitmask and a list of operands can be associated to each assembly instuction. Instructions that have the same bitmap and operands are then placed in the same group. In the example of Table 2.2, the *add* and *and* instructions (extracted from [64]) have the same bitmask and operands and are placed in the same group, while *andi* has a different format.

In case of processor supporting multiple ISAs, the grouping is repeated for all of them. For each group of instructions created in this way, a test procedure is devised, as shown in the next subsections.

#### Control and Status Bits

Some assembly instructions are sensible to control and status bits of the processors (i.e., bits that are part of certain special purpose registers, such as the Program Status Word). The processor datasheets provide information about each of these bits. In particular, typical control bits affecting the decode stage are related to:

- privilege level: part of the instruction set could be disabled when the processor is running with low privileges (e.g., instructions handling special purpose registers);
- processor features: they permit to disable certain classes of assembly instructions associated to specific functional units (e.g., floating-point operations).

For each of these control bits, one group is created, which includes all the instructions that are sensible to the value of that bit. The effect of such a bit is that instructions in the group behave differently according to its value. For example, they are normally executed in one case, while raise an exception in the other (or they behave as a no-operation instruction).

Instruction may affect the status bits of the processor, as the effect of a special computational result (e.g., an overflow). Status bits are typically modified in the following pipeline stages (e.g., during the execution stage, for the overflow bit) and reported in the processor ISA. Although a group for each of these bits may be created, it is typically unnecessary. It is up to the signature mechanism associated to each functional category to observe the values of status bits.

### 2.1.3 Signature Mechanisms

Instructions that are processed by a faulty decode unit are likely to be corrupted. The effect of executing legal instructions (i.e., assembly instructions that are compliant with the processor ISA) is a potential corruption of a data element or the execution flow. The specific faulty behavior highly depends on the functional category of the decoded instruction. Consequently, the mechanism that collects the fault effect into a result value has to be implemented differently depending on the functional category. This mechanism is also referred to as *signature*. In the following paragraphs, details about different signature mechanisms are given.

Finally, the case in which a legal instruction is decoded as unimplemented or illegal opcode has to be handled. This can be done by including an interrupt service routine (ISR) that modifies the signature value every time such a situation happens.

### ALU

The arithmetical and logical instructions make use of the pipeline ALU. Operations of this category involves processor registers (and immediate values). The signature mechanism for ALU instructions consists in the following steps:

1. the register file carefully is loaded with appropriate values;
2. the ALU instruction is executed;
3. the signature value is computed over the destination registers (including status bits such as overflow an comparison flags).

An effective approach consists in identifying registers whose encoding is adjacent to the destination register with respect to the Hamming distance. Such neighbor registers have to be loaded with a values that differ from the expected result of the computation. Moreover, they have to concur in the update of signature value. Similarly, also the neighborhood of source registers has to be loaded with values that differ from the related source register.

An example of the proposed mechanism is shown in Fig. 2.1. In particular, Fig. 2.1a reports the test procedure that includes a signature mechanism for an adding operation. The neighborhoods of the source and destination registers of the adding operation are listed in Fig. 2.1b.

### Memory

Instructions belonging to the *memory* category make use of memory addresses (or offsets) as operands, typically stored into base and index registers, or as immediate values. A fault affecting the decode unit and corrupting the target address of a certain memory instruction can be observed by means of a successive memory access. A signature mechanism is implemented by coupling together two memory instructions, which respectively excite and observe the effect of a possible fault. For example, a Store operation into a certain memory location is followed by a Load operation from the same location. In order to avoid a possible corruption of the instruction operands

```
begin
    foreach reg ∈ Neighborhood(r4) do
        reg ← K ≠ 3;
    end
    foreach reg ∈ Neighborhood(r21) do
        reg ← K ≠ (3+2);
    end
    r4 ← 3;
    r21 ← r4+2;
    signature ← update with r21;
    foreach reg ∈ Neighborhood(r21) do
        signature ← update with reg;
    end
end
```

(a) Test procedure

| | | |
|---|---|---|
| Src: | $r4$ | (encoding 00100) |
| | $r5$ | (encoding 0010**1**) |
| | $r6$ | (encoding 0011**0**) |
| | $r0$ | (encoding 00**0**00) |
| | $r12$ | (encoding 0**1**100) |
| | $r20$ | (encoding **1**0100) |
| Dst: | $r21$ | (encoding 10101) |
| | $r20$ | (encoding 1010**0**) |
| | $r23$ | (encoding 1011**1**) |
| | $r17$ | (encoding 10**0**01) |
| | $r30$ | (encoding 1**1**101) |
| | $r5$ | (encoding **0**0101) |

(b) Neighborhood wrt Hamming

Fig. 2.1 Example of signature mechanism for ALU instructions

```
begin
    r7 ← test value;
    Store r7 with base 40000000h and offset 1234h;
    Load r8 with base 40001234h and offset 0h;
    signature ← update with r8;
end
```

Fig. 2.2 Example of signature mechanism for memory instructions

at the same way, the memory location is expressed differently in the two operations, as analyzed in [55]. An example of the proposed mechanism is shown in Fig. 2.2.

Additionally, faulty memory accesses can create unexpected exceptions that have to be handles. In order to protect the memory outside the testing area, special capabilities offered by the system have to be exploited whenever possible. For example:

- by configuring the memory protection unit (MPU) before executing memory instructions;
- by implementing an ISR that modify the signature value in case of memory violations.

```
begin
    r5 ← 123;
    if r5 ≠ 123 then
        r5 ← 0;
    end
    signature ← update with r5;
end
```

Fig. 2.3 Example of signature mechanism for execution flow related instructions

### Execution Flow

The operands of instructions belonging to the *execution flow* category are memory offsets, table indexes, and condition registers (used by conditional branch instructions). This category requires special attention, since faults affecting the decode unit may result in a corrupted control flow of the program. In order to guarantee a controlled execution flow, the following guidelines to detect has to be followed:

- for each conditional branch instruction, the unexpected case (e.g., according to a test result, the branch should be taken, but it is not) has to be handled by means of instructions able to restore the execution flow and to modify the signature value;
- unexpected memory violations due to possible faults that corrupt the branch target addresses has to be handled by implementing an ISR (and eventually by configuring the MPU);
- on-board timers (e.g., watchdogs) has to be configured whenever possible in order to recover from possible endless loops.

A simple example of signature mechanism for conditional branch instructions is presented in Fig. 2.3.

### Processor Control

Instructions belonging to the *processor control* category are used to move data between the special purposes registers (SPR). Operands of such instructions are typically register codes. A fault affecting the decode unit and corrupting the register code can be observed with the same mechanism implemented for the memory

category. In order to excite and observe the effect of a possible fault, two processor control related instructions are coupled together. In this case, an instruction that modifies a special purpose register is followed by a read operation from the same register. However, the values transferred between the special purpose registers have to be carefully evaluated to avoid corruptions of the system integrity.

An effective signature mechanism consists in the implementation of a set of ISRs that handle wrong SPR manipulations. In details, a corruption of the register code results in a manipulation of an unexpected SPR, e.g., an SPR encoded near with respect to the Hamming distance. ISRs have to modify the signature value whenever such a situation happens.

### 2.1.4 Proposed Test Strategies

In the following paragraphs, the signature mechanisms are integrated in the overall test procedure. Different strategies are presented, each one focusing on a specific functionality of the decode unit, such as determining the instruction opcode, the operands, and the interaction with control and status bits.

#### Opcodes

One of the features performed by the decode unit is the identification of the kind of instruction that is going to be issued. The proposed test strategy handles all single bit corruptions of the instruction opcode. The principle is that a faulty decode unit, after processing a legal opcode, will result into one of the following alternatives:

1. the expected behavior;
2. the behavior of a different legal opcode;
3. an exception due to illegal opcode.

If the first case occurs, it means that the specific opcode is not able to excite a certain fault, while in the other two cases, the fault is excited and has to be properly handled. In order to be effective, the test procedure has to include a sufficient number of opcodes and to excite all possible faults. Moreover,the test procedure has to implement a proper signature mechanism and to observe the effect of all excited faults.

| | |
|---|---|
| Test opcode: | 7C000214 – add r0,r0,r0 |
| Bitmask: | FC0007FF |
| Neighbor legal: | 7C000215 – add. r0,r0,r0 |
| | 7C000210 – doz r0,r0,r0 |
| | 7C000014 – addc r0,r0,r0 |
| | 7C000614 – addo r0,r0,r0 |
| | 78000214 – rldcr r0,r0,r0,8 |
| | 74000214 – andis. r0,r0,0x214 |
| | 6C000214 – xoris r0,r0,0x214 |
| | 5C000214 – rlwnm r0,r0,r0,8,10 |
| | 3C000214 – lis r0,0x214 |
| Neighbor illegal: | 7C000216, 7C00021C, 7C000204, |
| | 7C000234, 7C000254, 7C000294, |
| | 7C000314, FC000214 |

Fig. 2.4 Example of opcodes in the neighborhood of the `add` instruction

The proposed strategy is composed of two phases. The first phase executes all kind of instructions of the processor ISA. The second phase makes use of carefully selected illegal opcodes. In order to execute all the available instructions, the processor flags have to be set appropriately (e.g., the FPU has to be enabled).

During the first phase, the fixed bits of a legal opcode are manipulated and all neighbor opcodes are identified, i.e., those having Hamming distance equal to 1. According to the functional category of each of the neighbor opcodes, the test procedure has to implement a signature mechanism that handles the case in which the target opcode has been decoded as the corresponding neighbor. If illegal opcodes are part of the neighborhood, they have to be handled by an ISR. Fig. 2.4 reports the neighborhood of the *add* instruction from [64]. The bitmask indicates the fixed bits of the opcode. In the example, the signature value has to be updated after the execution of the target instruction also with the content of status bits, that are potentially modified by neighbor instructions (e.g., *addc* sets the carry flag). Moreover, the register *r0* has to be loaded carefully (e.g., in order for a neighbor instruction to set the overflow flag).

The second phase of the proposed strategy makes use the illegal opcodes gathered during the first phase. Each of these opcodes is inserted in the test procedure. In the fault-free scenario, the processor raises an exception each time an illegal instruction is executed. However, in presence of a fault, the opcode may be decoded as a legal

instruction. In this case, the test procedure has to implement a mechanism able to modify the signature value when such a situation happens.

Please note that illegal instructions may reach the decode unit also during the normal system operations. A typical case is the presence of software bugs. For this reason, functional strategies targeting both legal and illegal instructions are equally necessary.

### Operands

The decode unit selects the operands for next pipeline stages and eventually checks dependencies between source and destination registers. Each operand specified in the instruction bits represents one of the following:

- a general/special purpose register code
- the index of a bit/field in a control/status register
- an immediate value
- a memory address/offset.

The proposed test strategy handles all single bit corruptions of the operands. For each group of instruction formats (see Section 2.1.2), the test procedure selects an instruction and instantiates it multiple times, each time with different operands. The values used as operands are the following:

1. complementary patterns (e.g., all 0s and all 1s);
2. minimum, maximum, middle values (e.g., the first, the last, and an intermediate register encoding value);
3. marching bit (e.g., 100..., 010..., 001...);
4. random values.

The first test guarantees that both 0 and 1 values are assigned to each bit of the operands. The second test is more functional based and should cover possible corner cases. These two tests are suitable for operands such as immediate values and memory offsets.

The third test is more suitable for instructions that perform operations among registers and is intended to cover possible faults affecting the comparators that implement dependency checking between source and destination registers (if coupled with the complementary patterns). In details, the test consists in propagating the

Table 2.3 Operand configurations for instructions working on three registers

| Configuration | Operand rD | Operand rA | Operand rB |
|---|---|---|---|
| 1 | 00000 | 00000 | 00000 |
| 2 | 11111 | 11111 | 11111 |
| 3 | 10000 | 00000 | 00000 |
| 4 | 01000 | 00000 | 00000 |
| 5 | 00100 | 00000 | 00000 |
| 6 | 00010 | 00000 | 00000 |
| 7 | 00001 | 00000 | 00000 |
| 8 | 00000 | 10000 | 00000 |
| 9 | 00000 | 01000 | 00000 |
| 10 | 00000 | 00100 | 00000 |
| 11 | 00000 | 00010 | 00000 |
| 12 | 00000 | 00001 | 00000 |
| 13 | 00000 | 00000 | 10000 |
| 14 | 00000 | 00000 | 01000 |
| 15 | 00000 | 00000 | 00100 |
| 16 | 00000 | 00000 | 00010 |
| 17 | 00000 | 00000 | 00001 |

bit 1 in the variable bits (i.e., the operand bits) of the instruction. An example of configurations for instructions working on three registers is listed in Table 2.3.

The last test uses random values as operands. It is suggested to include a certain amount of instructions with random values in the test procedure to increase the overall fault coverage. A reasonable number is the same amount of instructions used in the third test.

Finally, please note that the just presented test strategies targeting the operands can be combined with the test on the opcodes presented in the previous paragraph. For each group of instruction formats, every one of the opcodes in the group can be bound to a different configuration on the operands.

### Control and Status Bits

The decode unit is influenced by the values of configuration bits that enable/disable certain processor capabilities. The proposed test strategy handles all single bit corruptions of such bits. For each group of instructions that are sensible to control

```
begin test procedure
    r5 ← 0;
    r6 ← 0;
    enable debug features;
    breakpoint;
    disable debug features;
    breakpoint;
    signature ← update with r5;
    signature ← update with r6;
end
```

```
begin ISR debug
    r5 ← r5 + 1;
    return;
end

begin ISR illegal
    r5 ← r6 + 1;
    return;
end
```

Fig. 2.5 Test procedure and ISRs for breakpoint instructions

bits (see Section 2.1.2), the test procedure selects an instruction and executes it when the related configuration bit is set to the two values. Before executing the instruction, the related configuration bit is programmed. In one case, an exception is raised when the instruction is decoded. This case has to be handled by an ISR. In the other case, the test procedure has to implement a suitable signature mechanism, depending of the functional category of the instruction.

As an example, let us consider a debug related instruction, such as a breakpoint. The ISR that handles the breakpoint has to update the signature value and restore the program execution flow. If the breakpoint instruction is executed after disabling the debug features, it will results in a different behavior (typically, it raises a different kind of interrupt, or it is ignored). In a faulty condition, it may raise a debug interrupt as well, thus corrupting the signature value due to the unexpected execution of the ISR. The opposite situation may happen in the other case, when the debug features have been enabled. A simple signature mechanism for these scenarios is presented in Fig. 2.5.

Please note that each one of the legal instructions are executed by the test procedure of the opcodes, so it is enough to execute the selected instructions one more time with the opposite control bit values.

### 2.1.5 Experimental Results

The proposed test methodology has been applied to a SoC including a 32-bit pipelined microprocessor based on the Power Architecture™. The SoC is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers and is currently being manufactured by STMicroelectronics. The decode unit module is composed of 2,440 gates and counts 11,454 stuck-at faults. Overall, the decode unit occupy the 1.55% of the entire processor area.

In the experiments, an existing test program *TP Instruction Coverage*, written for verification purposes, has been first fault-simulated. Starting from such a program, the progression in the coverage achieved by implementing the proposed methods is shown. Note that the actual fault coverage reached by applying the proposed method is the last one presented. In order to show the test programs contribution on the final test set, the fault simulation results are reported step by step.

The *Instruction Coverage TP* set was built by simply including all instructions in the ISA with no additional requirements on opcodes, operands, flags, and illegal instructions. Such an approach is widely used in the industry for both verification and test. The program reaches a quite low coverage level (62.71%).

The first enhancement applied to the initial test program was the test of the opcodes, by following the method presented in the fist part of Section 2.1.4. In the specific case study, it leaded to the generation of 30 groups of instructions (and related masks), whose size ranges from 1 to 125 instructions (12 in the average). As described in the proposed approach, illegal opcodes were also included. The fault coverage reached the 80% with this enhancement.

Next, the proposed strategy considers the generation of programs for operand improved coverage. This method is explained in the second part of Section 2.1.4. The fault coverage was improved to about 90%.

Finally, as presented in the last part of Section 2.1.4, the control and status bits have been handled. As a result, the final fault coverage was 91.08%. Table 2.4 reports the incremental fault simulation results obtained by applying the proposed strategy.

Table 2.4 Incremental results of the application of the proposed approach

| | Size [kB] | Duration [cc] | SA FC% |
|---|---|---|---|
| Instruction Coverage TP (initial) | ~14 | 12,456 | 62.71 |
| + Opcodes | ~19 | 21,688 | 80.63 |
| + Opcodes + Operands | ~23 | 23,230 | 90.14 |
| + Opcodes + Operands + C/S Bits | ~24 | 23,652 | 91.08 |

Table 2.5 Experimental results on random opcodes

| | Size [kB] | Duration [cc] | SA FC% |
|---|---|---|---|
| 1k random opcodes | ~10 | ~11k | 49.19 |
| 10k random opcodes | ~100 | ~142k | 52.33 |
| 100k random opcodes | ~1,000 | ~1.5M | 53.50 |

For the sake of completeness, a random generation process was implemented, where random opcodes (legal and illegal) were produced and used as the test program code. Table 2.5 shows the related results.

Remarkably, only about 8% of SA faults on the module escape when the proposed flow is implemented. Please note that the proposed flow is not based on the netlist description of the processor, but just on the knowledge of the ISA and user manuals. Indeed, the coverage level reached is a particularly useful measure telling that around 91% of faults of the decode unit can be systematically detected without any design knowledge.

## 2.2 Register Forwarding and Pipeline Interlocking

A *Data Hazard* is defined as a situation where a processor pipeline produces a wrong output due to data dependency relations between instructions [62]. This may happen when the input of one instruction coincides with the output of a previous instruction, and this output has not yet been written into the proper register when the instruction execution phase takes place (i.e., the two instructions have data dependence). Experiments performed on a MIPS-like pipelined processor with programs from the SPEC92 benchmarks show that almost 50% of the executed instructions have some kind of data dependence [65]. A common strategy to deal with data hazards reducing executing time penalties is to handle them by hardware, relying on two

basic mechanisms: Register Forwarding (or data bypass) and Pipeline Interlock (RF&PI); usually, for optimizing the processor performance, both mechanisms are implemented.

A fault present in the hardware structures implementing the Register Forwarding and Pipeline Interlock mechanisms may result in their malfunctioning and therefore in the processor producing a wrong output; alternatively, the fault may cause an unneeded pipeline stall and consequently a performance penalty. Hence, some of the faults affecting the RF&PI logic fall in the class of *performance faults* [66].

In the next sections, a SBST strategy to test the hardware mechanisms that handle data hazards is presented. In Section 2.2.1, these mechanisms are first analyzed, describing common principles present in different implementations. An algorithm to develop a suitable test program is then provided in Section 2.2.2. Finally, experimental results are presented in Section 2.2.3.

### 2.2.1 Architectural Overview

Register Forwarding (or data bypass) and Pipeline Interlock are functions managed by some combinational logic units included in a pipelined microprocessor to avoid data hazards.

The methods employed to avoid data hazards mainly consist in checking if there is a data dependency between two instructions simultaneously present in different pipeline stages, and take suitable counteractions accordingly. Typically, when an instruction enters the pipeline, the system checks if its input operands correspond to the same register which any other instruction already present in the pipeline is using as output. If this is true, there is a data dependency between the two instructions and some actions have to be taken. For clarity purposes, let us call the first instruction to enter the pipeline instruction 1 and the instruction arriving later, which has a data dependency with the first one, instruction 2. In this case Register Forwarding must be activated: the input data for instruction 2, instead of coming from the register file, is directly taken from the stage where instruction 1 produces it. In case instruction 1 is not yet in that stage, Pipeline Interlock is used. Pipeline Interlock implies the pipeline is stalled until instruction 1 reaches the stage in which the data is produced. At that moment, Register Forwarding is used to send the result of instruction 1 to the stage where instruction 2 needs its operands.

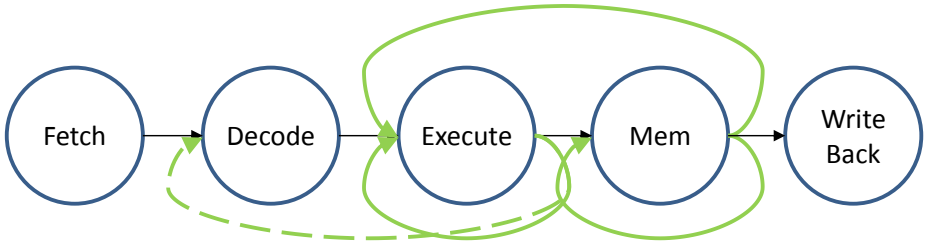



Fig. 2.6 Graph of the possible forwarding paths between pipeline stages

According to this working mechanism there are different possible forwarding paths (shown in Fig. 2.6 for a MIPS-like processor) and their activation depends not only on the existence of some data dependency between two instructions but also on the stage in which the instructions produce/require the data.

Interestingly, forwarding paths may vary according to the processor architecture. For example, if the branch instructions are completely resolved during the decode stage, the processor core may include forwarding paths able to feed the decode stage, as represented by the dashed line in Fig. 2.6, and shown as input of de decode stage in Fig. 2.7.

To implement the described working mechanism, the different stages of the pipeline and the Register File module interact with the RF&PI unit within the Data Hazards handling module, as shown in Fig. 2.7 using the MIPS architecture as an example.

Due to the required behavior of the unit and to its architecture, usually located within the pipeline control logic, specific sequences of instructions only are able to trigger its action. Obtaining the full activation of the module, required for its test, is a relatively difficult task; also, due to the intensive interaction of the unit with the register file, proper register initialization is needed to build an effective test program.

The architecture of the data hazards handling mechanisms can be divided in the three structural sub-blocks: multiplexers, comparators, and bonding logic. Their purposes are detailed in the next paragraphs.

### Multiplexers

A multiplexer (or MUX) is a combinational block able to select one input out of multiple ones and connect it to the output signals.

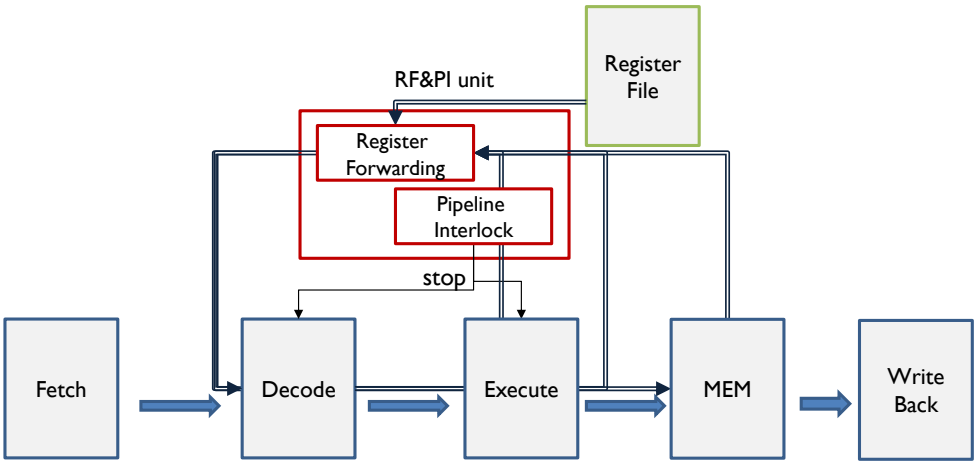



Fig. 2.7 The Register Forwarding and Pipeline Interlock unit and its interaction with the processor pipeline and Register File module.

The Register Forwarding logic uses a number of multiplexers to send the appropriate data to each stage. As many MUXs are used as possible destinations exist (e.g., one for operand 1 of the execute stage and a second one for operand 2 of the execute stage, etc.), each one with a number of inputs equal to the number of possible data origins (register file, write back stage, execute stage, etc.).

### Comparators

Comparators (CMPs) are mainly used for two purposes: first, to detect if there is a data dependency present in the pipeline, i.e., if two instructions present concurrently in different pipeline stages use the same registers as input and output, respectively; in this case the CMPs compare register identifiers (each register owns a specific identifier which is usually encoded in the instruction operating code). The other use is within the pipeline interlock mechanism to detect potentially unresolved hazards: when an instruction has a data dependency, they check if that dependency is resolved in a later stage than the one when the data is needed and accordingly halt the pipeline when necessary.

#### Bonding logic

It integrates the logic controlling all the data hazard avoidance mechanisms, determining at any given moment the activation of register forwarding and pipeline interlock based on the processor status.

### 2.2.2 Proposed Test Strategies

In this section, a SBST strategy for the main structures of RF&PI unit is presented. The overall test is developed bearing in mind that the RF&PI unit is mainly composed of MUXes and CMPs. A proper test algorithms for these types of modules have been considered in [67] and [68], respectively.

The sequence of values mandated by the above mentioned algorithms is applied to each MUX ore CMP embedded within the architecture of the addressed unit. The produced results is made observable in a functional manner. The testing of the bonding logic is done as a consequence of applying the testing strategies of the multiplexers and comparators. The proposed strategies address stuck-at faults, only.

#### Test of the Multiplexers

In [67] the authors consider a number of different implementations for a MUX, and prove that they can all be tested by the same set of input vectors. In particular, they prove that when considering a generic $n$-to-1 multiplexer a set composed of $2n$ test patterns can achieve full stuck-at fault coverage.

The set of necessary test vectors for an 8-to-1 MUX with 1 bit parallelism (i.e., 8 inputs and 1 output, each one composed of 1 bit) is reported in Table I. $S_i$ denotes a selection signal, $D_i$ an input data signal, while $Z$ is the output.

When moving to MUXs having a parallelism greater than 1, the same set of vectors can be used, substituting the 1/0 value with as many 1/0 bits as the parallelism is.

The proposed test algorithm applies the input vectors into a sequence of instructions to be executed by the processor able to

- apply the same set of values to the generic MUX within the RF&PI unit;
- make the MUX output observable.

Table 2.6 Test vectors for a 8-to-1 MUX

| Vector | $S_2$ | $S_1$ | $S_0$ | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 5 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 6 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 8 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 9 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 11 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 12 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 13 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 14 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 15 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

A specific MUX channel is activated when a data dependence between two instructions is detected, and the result produced by the first (i.e., stored in a given pipeline register) must be forwarded to the second (i.e., to the input of a given stage). Hence, the considered MUX inputs are the data registers existing within the pipeline registers. This means that for every MUX in the RF&PI unit (i.e., for each possible data dependence configuration) a program fragment can be created, including a couple of data dependent instructions that trigger the Register Forwarding. Ideally, the fragment should be replicated as many times as the number of test vectors required to test the MUX. For each replica the values of the pipeline registers corresponding to the inputs of the MUX should hold the all 0s or all 1s value, according to what mandated by Table 2.6. Finally, the fragment should include an instruction to make the output of the MUX observable.

As an example, an 8-bit wide 3-to-1 MUX for feeding the first operand of the execution stage in a pipelined processor is considered. Assuming that the processor stages are fetch (*IF*), decode (*ID*), execute (*EXE*), memory (*MEM*), and write back (*WB*), as described in [62], and the considered operands are provided to the MUX inputs from *ID*, when no forwarding is required, and from *EXE*, and *MEM* stages, when forwarding is needed.

Table 2.7 Input values for the 3-to-1 MUX feeding the first operand input of the EXE stage in a pipelined processor

| Vector | $S_1$ | $S_0$ | $D_0$ (*ID*) | $D_1$ (*EXE*) | $D_2$(*MEM*) | Z |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 00 | FF | FF | 00 |
| 1 | 0 | 1 | 00 | FF | 00 | FF |
| ... | ... | ... | ... | ... | ... | ... |

```
1  li   R1, 0xff          ; prepare for observability
2  li   R2, 0x00
3  ld   R3, FF_val(R0)
4  li   R4, 0xff
5  add  R1, R2, R0        ; vector 0
6  nop
7  nop
8  nop
9  sd   R1, data1(R0)     ; observability instruction
10 ld   R3, 00_val(R0)
11 li   R2, 0xff
12 add  R1, R2, R0        ; vector 1
13 nop
14 nop
15 nop
16 sd   R1, data2(R0)     ; observability instruction
```

Fig. 2.8 Test program fragment for testing the MUX for the EXE stage

In this example, the first MUX input comes from the decode stage, the second one from the execution stage, and the last one from the memory stage. An assembly-like sequence is provided, that could be used for applying the first 2 vectors proposed in [67] and reported in Table 2.7.

Table 2.7 reports the required input values in the MUX under test. As the reader can notice, the different values in the MUX inputs at every clock cycle depend on the instructions that traverse the processor pipeline during the considered instance of time. The program in Fig. 2.8 reports a sequence of instructions able to apply the mentioned vectors in the MUX under evaluation.

Assuming that there are no cache misses and that the LD/SD instructions require only one clock cycle during the MEM stage, the first vector in Table 2.7 is applied

during the EXE stage of instruction 5. In the considered clock cycle, the input value of the MUX under test is actually provided by the decode stage that reads from the register file the value of R2, already set by instruction 2. Instructions 3 and 4 propagate through the pipeline the rest of the values (FF for both cases) required by the test vector. Additionally, instructions 1 and 9 are devoted to add observability capacities to the considered code fragment.

The second vector is applied by the block of instructions 10-16. In details, instruction 12 depends on instruction 11 that forwards its output value (the one for R2) from the output of the EXE stage to the input of the MUX under evaluation. Once again, the rest of the instructions set appropriate values in the pipeline and support observability.

Fig. 2.9a describes the normal behavior for the sample program at the time in which Vector 0 is applied. The different values are propagated exploiting the block of instructions 3-5 that guarantees the values 00, FF, and FF in the MUX inputs.

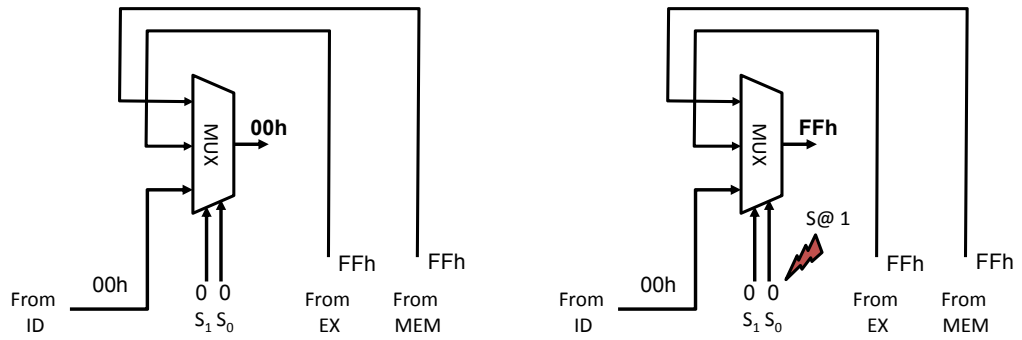
If a structural fault occurs in one of the selection wires (as in Fig. 2.9b), the MUX outputs assume a different value (in this case FF) easily detected thanks to the observability instructions. Similarly, if a stuck-at fault is located in one of the forwarding paths, the error effect will result in a MUX output different from the expected 00 (or FF).

Please note that the rest of the patterns of Table 2.6 (vectors 2-15, in the case of an 8-to-1 MUX) can be easily translated to assembly instructions following the same scheme provided for Vectors 0 and 1. With this algorithm a total coverage of faults in the decode logic of multiplexer is assured, with an optimum scalability in terms of bit-wise parallelism and a short SBST code footprint.

### Test of the Comparators

To thoroughly test an $m$-bit wide comparator (i.e., 2 inputs, with $m$ lines each), independently on its low-level implementation, it was stated in [68] that one can use a set of $2m+2$ patterns. In the same article it is shown that the following patterns allow achieving complete fault coverage.

Out of the $2m+2$ vectors, two correspond to the situation in which the two CMP inputs match: this means that all the corresponding bits in the two input operands are equal. Each bit holds an opposite value in the two vectors.



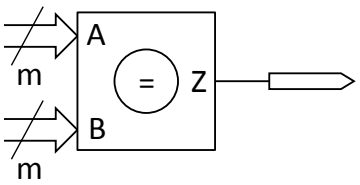

(a) Normal behavior

(b) Wrong result in case of stuck-at-1 in one selector

Fig. 2.9 Vector 0 application to the MUX for the EXE stage

Each of the other $2m$ patterns generates a mismatch in only one bit of the pair of words fed in parallel into the comparator. Hence, this set of patterns corresponds to a walking 1, starting from the most significant bit (MSB) of one of its inputs going all the way to the least significant bit (LSB) of the other input. Fig. 2.10 shows the schema of a comparator and the patterns to be applied to test it. Anyhow, being combinational logic, the order in which these patterns are applied is irrelevant.

As for the multiplexers, the approach aims at developing a program fragment able to apply the above set of test vectors to each CMP in the RF&IP unit. As already mentioned, the detection of data dependencies in the pipeline is done with comparators. For this use there is one comparator per input operand, per each stage involved in the RF&PI mechanism. In common processor architectures having two operands per instruction this means a series of comparators that check the two operands against the possible sources in every one of the pipeline stages where these values may be produced (see Fig. 2.6). Accordingly, one of the inputs of these comparators is connected to the operand identifier (i.e., a register) encoded in the instruction and used by the instruction in one stage, while the other input is the identifier of the output register on one of the possible source stages where the other instruction in the pair assumed to have a data dependence is placed. The same applies for all relevant pipeline stages. In this way any data dependency can be detected. In order to excite these comparators with the required patterns, registers with identifiers following the patterns previously described should be addressed by consecutive instructions.




| Pattern | A | B | Z |
|---|---|---|---|
| 1 | **1**0...00 | 00...00 | 0 |
| 2 | 0**1**...00 | 00...00 | 0 |
| ... | | | |
| $m-1$ | 00...**1**0 | 00...00 | 0 |
| $m$ | 00...0**1** | 00...00 | 0 |
| $m+1$ | 00...00 | **1**0...00 | 0 |
| $m+2$ | 00...00 | 0**1**...00 | 0 |
| ... | | | |
| $2m-1$ | 00...00 | 00...**1**0 | 0 |
| $2m$ | 00...00 | 00...0**1** | 0 |
| $2m+1$ | 11...11 | 11...11 | 1 |
| $2m+2$ | 00...00 | 00...00 | 1 |

Fig. 2.10 Comparator schema and test patterns

The example in Fig. 2.11 shows a piece of code intended to excite the comparator for operand 1 in the EXE stage, considering the processor architecture described in Section 2.2.1. One input of the comparator is connected to the pipeline register storing the identifier of the output register of the instruction in the EXE stage; the other input is connected to the pipeline register storing the identifier of the operand of the following instruction. The red arrows show the performed comparisons. For this example the register file is assumed to be composed of 8 registers, so that 3 bits are used to represent the register identifier; consequently, $2 \times 3 = 8$ comparisons are needed to apply the sequence of test vectors proposed above.

The used registers must be properly initialized, in order to make possible faults observable. In particular, please note that R0 and R7 are initialized with values different from all the other registers. Similarly, the value for $k$ should be carefully chosen to avoid any overflow (e.g., $k = 0$).

The proposed sequence of 17 instructions thoroughly excites the comparator of the example. The only purpose of the *nop* instructions is to create the necessary distance between the relevant instructions.

The same sequence of instructions can be easily tailored to test the other comparators. For example, it can be modified to target both comparators related to the EXE stage, just using an instruction with two input operands (e.g., *add* $R_A, R_B, R_C$ – which performs $R_A \leftarrow R_B + R_C$) instead of an immediate operand. Additionally,

```
1  addi R0, R1, k   ; R0 <= R1 + k, k being a constant
2  nop      ↘
3  addi R0, R4, k   ; A := R4 (100), B := R0 (000)
4  nop      ↘
5  addi R0, R2, k   ; A := R2 (010), B := R0 (000)
6  nop      ↘
7  addi R0, R1, k   ; A := R1 (001), B := R0 (000)
8  nop      ↘
9  addi R4, R0, k   ; A := R0 (000), B := R0 (000)
10 nop      ↘
11 addi R2, R0, k   ; A := R0 (000), B := R4 (100)
12 nop      ↘
13 addi R1, R0, k   ; A := R0 (000), B := R2 (010)
14 nop      ↘
15 addi R7, R0, k   ; A := R0 (000), B := R1 (001)
16 nop      ↘
17 addi R0, R7, k   ; A := R7 (111), B := R7 (111)
```

Fig. 2.11 Test program fragment for testing the CMP in the EXE stage

placing a useful instruction (for testing purposes) instead of the *nop* instruction can allow us to also test the comparators of other stages at the same time. By generalizing this solution, the addition of one instruction per involved stage allows testing all comparators.

Another use of comparators within the data hazard mechanism is in the pipeline interlock activation. Let us define the *level* of an instruction as the stage in which the data produced by that instruction is ready, or when the effects of the instruction are seen; for example, in the miniMIPS architecture [69] the level of an ADD is the EXE stage, while the level of a LOAD from memory instruction is the MEM stage. This information is hardcoded within the microprocessor and gathered when each instruction is decoded (i.e., at the decode stage). When a data dependency is identified, the level of the instruction producing the result to be forwarded is compared with the stage in which the instruction is, in order to identify potentially unresolved hazards. In the case of the data hazard handling module, there is one comparator per stage devoted to this task, and its inputs are connected to a constant indicating the stage itself and to a signal indicating the level of the instruction present in that stage.

#### Observation Mechanism

The previous subsections describe how to write test programs able to test the RF&PI unit. This test can be performed by suitably activating the different components of this unit, and then making the produced results observable.

However, some the faults affecting the RF&PI unit may not produce any wrong results, but rather a change in the performance of the processor, e.g., by introducing unnecessary stalls. For the purpose of detecting these faults, some mechanism to measure the time required to execute the test program (or some of its parts) can be exploited.

This can be accomplished in several ways:

- using the performance counters [70] existing in many processors and able to count the number of stalls;
- resorting to some timer able to measure the performance of the processor when executing a given piece of code;
- adding some ad hoc module to the system able to monitor the bus activity [71].

### 2.2.3 Experimental Results

The feasibility and effectiveness of the proposed approach have been evaluated on an academic processor module and on a commercial microprocessor.

The selected academic case study is the miniMIPS processor [69] synthesized using an in-house developed library and resulting in a 16,236-gate circuit (without multipliers). The miniMIPS architecture is based on 32-bit buses and includes a 5-stage pipeline. The RF&PI unit occupies around 3.4% of the total number of gates of the processor, accounting for a total of 3,738 stuck-at faults.

In order to experimentally validate the proposed approach, an initial test set of programs has been considered. This set tackles the testing of the whole miniMIPS processor core and achieves about 91% fault coverage with respect to stuck-at faults. The test programs contained into the test set were developed following state of the art strategies such as [72]. However, the stuck-at fault coverage achieved on the RF&PI unit reached only about 66%, thus proving that specific test algorithms are required for it.

Table 2.8 Characteristics of the test program for the RF&PI unit

| | Size [kB] | Duration [cc] | SA FC% |
|---|---|---|---|
| RF&PI TP | 4 | 2,084 | 98.89 |

Secondly, a test program (RF&PI TP) was developed following the algorithm described in Section 2.2.2 and specifically targeted to the RF&PI unit. Its main characteristics are summarized in Table 2.8 in terms of size, duration, and stuck-at Fault Coverage on the RF&PI unit. The few untested faults (39 out of 3,520) are mainly related to signals and features that cannot be activated using a functional approach, such as the interrupt signal; other faults remain untested because the module can support a coprocessor, which was not used in the experiments.

The proposed technique was also applied to a commercial System-on-Chip including a 32-bit pipelined microprocessor based on the Power Architecture™ and manufactured by STMicroelectronics. The device contains over 2 million logic gates and is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers.

In this case, the module playing the role of the RF&PI unit accounts for about 14k faults. The functional test program developed for the whole circuit reached only about 62% stuck-at fault coverage on the RF&PI unit. After adding to it some further test fragments developed according to the proposed test algorithm (and accounting for about 2,000 instructions and less than 5,000 clock cycles), the fault coverage on the same unit raised to about 92%. Obviously, implementation of the proposed algorithm on this architecture is harder than for the miniMIPS, due to the higher complexity of the pipeline and the higher number of functional units.

## 2.3 Dual-Issue Processors

This section deals specifically with the test of dual-issue processors with in-order execution of instructions strategy. Such processors cannot be tested with available techniques, as the overall fault-coverage will be low. This is mainly due to specific classes of sub-modules, or features, which are not well covered by general methodologies. Such modules have a considerable impact on the final result, since they strongly characterize this class of processors.

In the following, after a brief architectural overview (Section 2.3.1), typical issues due to instruction scheduling that characterize this class of processors are presented in Section 2.3.2. Then, a systematic methodology to develop an effective suite of test programs for dual-issue processors is proposed in the following sections. In particular, the methodology covers the following modules:

1. replicated computational modules in the execution units and, in particular, the set of multiplexers that propagate operands and results through the data-path (these modules are covered in Section 2.3.3);
2. register file and the replicated set of multiplexers used to read/write in the register bank (Section 2.3.4);
3. duplicated pipeline registers, and feed-forward paths, which are interconnected by multiplexers allowing pipeline stages to share data and dealing with data dependencies (Section 2.3.5);
4. interlocking logic, mainly composed of comparators detecting possible hazards and control logic managing pipeline stalls and routing feed-forward paths (Section 2.3.6);
5. instruction pre-fetch buffer logic, which behavior is strongly influenced by the occurrences of pipeline stalls (Section 2.3.7).

Finally, Section 2.3.8 reports about experimental results carried out on industrial processor cores manufactured by STMicroelectronics.

### 2.3.1 Architectural Overview

In the classical processor pipelined architectures, a sequence of instructions is fetched from the memory and fed to the pipeline that ideally processes one instruction per clock cycle. Such architectures are known as *scalar* or *single-issue*. On the contrary, modern processors are able to process many instructions at the same time, by leveraging on more complicated pipeline mechanisms and replicated computational modules. Such architectures are called *multiple-issue*, depending how many instructions are able to process for each clock cycle, e.g., two instructions in dual-issue processors. When the processor relies on special hardware rather than on the compiler to check whether multiple instructions can be issued (i.e., if they do not produce structural hazards or data dependencies), it is traditionally referred to as a *superscalar* processor. Such processors are classified either as *in-order* or *out-of-order*, whether

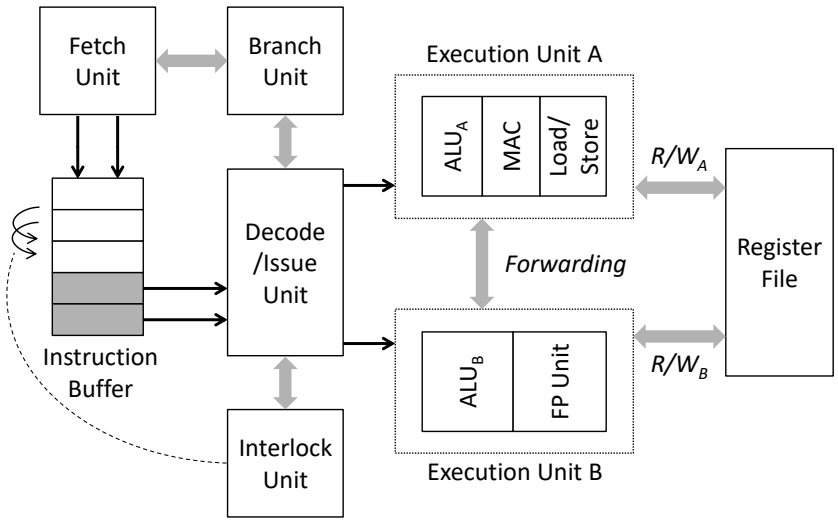



Fig. 2.12 Example block diagram of in-order dual issue processors

they keep the instruction order unchanged, or permit instruction reordering through specialized hardware modules (e.g., register renaming and re-order buffer).

This section deals with testing of in-order dual-issue processors. From a structural point of view, the processor's CPU contains two execution units that permit to treat two instructions at a time, allowing to the replicated computational modules to run in parallel. Depending on the specific processor's implementation, the execution units can be: *symmetric*, when both of them contain the same functional units; *asymmetric*, when some functional units are missing in one of them. From a logical point of view, the processor contains two pipelines (referred as *A* and *B*). At each clock cycle, two instructions are fetched from the memory and enter in the two pipelines where they are processed in-order.

An example block diagram of in-order dual-issue processor is shown in Fig. 2.12. In this processor, instructions are retrieved from the memory by a *Fetch Unit* and placed into an *Instruction Buffer*, which is composed of a certain amount of slots and behaves as a queue. The fetch unit interacts with a *Branch Unit*, which usually implements a speculative prediction of the program counter's value in case of branch instructions, e.g., by means of a Branch Target Buffer (BTB). Two instructions at a time are withdrawn from the instruction buffer by a *Decode/Issue Unit*, which can be implemented in several ways according to the issue and dispatch strategies of the analyzed processor. In case of stalls due to dependencies detected during instructions

issue, an *Interlock Unit* manages the pipeline stages, for example by limiting the propagation of instructions through the instruction buffer, or by interlocking the decode/issue unit until completion of the critical instruction (e.g., because requiring multiple clock cycles to be issued). The two instructions issued by the decode/issue unit are processed by two *Execution Units* (named *A* and *B* in the figure), each one collecting several functional units and interfacing with a multi-port *Register File*.

### 2.3.2 Scheduling Issues

When dealing with multi-issue execution, the internal status of the logical pipelines must be known at the beginning of the test program, before moving to the test operations. This permits to test the targeted module in a deterministic way, without counter effects caused by an unknown state. It is possible to identify two categories of uncertainty: uncertain signature and uncertain coverage.

A test program may present an *uncertain signature*, in case the non-deterministic scheduling of the instructions among the multiple logical pipelines can influence the signature value computed at run-time by the test program itself. If the test program does not to compromise its signature in presence of unknown conditions, still may present an *uncertain coverage*, i.e., an oscillating fault-coverage value resulting from different fault simulations of the same test program under different conditions. Such conditions are imposed by the instructions processed by the pipelines prior the test program's start. This is the typical situation arising when the test program is executed on-line, i.e., scheduled by the mission application.

The non-deterministic behavior represents an issue especially for those test programs implementing algorithms requiring that certain sequences of instructions are precisely scheduled in the two pipelines; previous pipelines status may influence the effectiveness of a test program. To solve this problem a viable solution needs to be implemented and embedded in each of the test programs. This process will be referred to as the *reset* of the pipelines. General guidelines for implementing the pipeline reset can be given.

First of all, the alignment of the instructions in the code memory may determine which pipeline processes which instruction. In this case, the test engineer should put the code of the test programs in the instruction memory taking into account that alignment (e.g., 8-byte aligned instructions are dual-issued). Clearly, other

architectures may not have such a static scheme, for example in case of a variable-length instruction set. However, the mechanisms governing the order in which the instructions are fetched from the memory and scheduled among pipelines are reported in the user manuals.

Secondly, since test program instructions entering in the pipelines can be influenced by the instructions entered previously and currently populating the different stages of the pipelines, as well as by flushing operations of the instruction buffer, a fixed number of *nop* instructions can be inserted to avoid a non-deterministic signature.

The described concerns are overcome by architectures provided with special assembly instructions in charge of flushing the instruction buffer, or performing the synchronization of the pipelines, by inserting a certain numbers of stalls in the earlier stages preventing the operands needed by the following instructions to be retrieved from feed-forward paths. As well, these instructions can be used in the implementation of the pipelines reset, coupled with the requirement about the alignment of instructions in memory.

### 2.3.3 Duplicated Computational Modules

Some implementations of in-order dual-issue processors adopt a duplication of certain computational modules, thus permitting the simultaneous execution of two instructions of the same kind. Examples of these modules are adder units, barrel shifters, and bit-wise operators.

Computational modules are tested by means of specific patterns that are brought to the module primary inputs by clearly identifiable assembly instructions. For example, an *add* instruction clearly exercises an adder unit.

In single-issue processors, once the patterns required to test a specific computational module are identified (e.g., from ATPG, or by a deterministic method), a test program can be composed, which loads the operands with the patterns, performs the instruction exercising the module, and updates the signature value with the results [73], without any importance given to the scheduling of the instructions. Also in VLIW architectures, the single computational module can be precisely exercised by mapping each operation to a specific part of a VLIW instruction at compile-time [35].

(a) Single-issue version

```
1 load   r1 , immediate1
2 load   r2 , immediate2
3 add    r3 , r2 , r1
4 store  R3 , memory
```

(b) Dual-issue version

```
1 load   r1 , immediate1
2 load   r2 , immediate2
3 add    r3 , r2 , r1
4 add    r4 , r2 , r1
5 store  r3 , memory
6 store  r4 , memory
```

Fig. 2.13 Single-issue (a) and dual-issue (b) versions of a snippet procedure to test adder units.

The case of dual-issue architectures is different, whereas a sequence of two instructions may be dispatched to different modules. For instance, two *add* instructions can use two adder units and be executed in parallel. The direct consequence is that a given sequence of test instructions that thoroughly test a single module will be split among the replicated modules if executed on a dual-issue architecture. Such mechanism intended to improve processor throughput is negatively impacting the fault-coverage as each unit executes only a subset of test instructions.

To illustrate the problem, we can consider the test of the adder units in the processor (see Fig. 2.13), and a list of effective patterns obtained by an ATPG run for one of the adders, which are supposed to have an equal net-list. As previously mentioned, each ATPG pattern can be translated to a sequence of instructions (see Fig. 2.13a); in the illustrated scenario instructions 1 and 2 load two registers with immediate values that represent the ATPG pattern, instruction 3 executes an adding instruction (*add*) which applies the pattern to the adder unit's primary inputs, and finally, instruction 4 stores the resulting value in a memory area which will be used for computing the test signature. If this code is executed by a dual-issue processor, the *add* instruction will be scheduled to excite just one of the available adders.

The first proposed modification of the code will guarantee high fault-coverage on both adders (see Fig. 2.13b). The pattern is applied on both modules by having a couple of equivalent target instructions executed consecutively. Once the pattern is loaded in the two registers (instructions 1 and 2), the target *add* instruction is replicated (3,4), with the second one using the same registers as inputs and storing results in different one; finally, the two results can be stored in memory (5,6).

Just duplicating the instructions is not enough. As analyzed in Section 2.3.2, the reset of the pipelines is important to reach the desired schedule of instructions. Additionally, some processors are able to recognize when two consecutive *add* instructions are using the same operands; in this case, a single adding operation is executed and the result is feed-forwarded. To face this potential issue, the solution is to use a disjoined set of registers for the second *add*; in this way, the processor does not optimize the execution and the same pattern is applied to both adders.

Finally, the strategy can be effectively applied on a more complex multi-issue processor. For example, in a 4-way in-order core, each pattern is replicated four times, and the set of utilized registers is larger.

### 2.3.4 Multi-Port Register File

The register file of dual-issue processors is unique. However, since two instructions need to be executed in parallel, the logic permitting reading and writing of registers is replicated. Thus, a typical register file for a dual-issue processor is composed of: 1 memory bank, 2 writing ports, and 4 reading ports.

Each writing port is structurally implemented as a decoder, which propagates the resulting value from the execution unit towards the memory bank; conversely, each reading port is implemented as a multiplexer that retrieves a value stored in the memory bank to be used as operand by the execution unit.

Given a duplicated ALU unit, two adding instructions can be processed in parallel by the stages of the pipelines *A* and *B*, as in the example of Fig. 2.14. In this example, the instruction 1 is supposed to be processed by the pipeline *A*, and the three operands are registers, which encoding is used as the selection for the decoder and the multiplexers dedicated to the pipeline *A*, referred as $DEC_A$ (in $Write_A$ port), $MUX1_A$ (in $Read1_A$ port), and $MUX2_A$ (in $Read2_A$ port), respectively. In the same way, the instruction 2 is processed by the pipeline *B*.

```
1 add r1,r2,r3      ; ALU in pipeline A
2 add r4,r5,r6      ; ALU in pipeline B
```

Fig. 2.14 Example of dual-issue execution of instructions that access to the register file ports in parallel.

Given this specific register-bank access method, a test strategy is needed to take into account the parallel access to the register file. As for replicated computational units, a methodology that enhances an algorithm for single-issue processors is proposed. The purpose of the algorithm is to test both the memory bank and the access ports.

### Single-issue Version

The basic algorithm for single-issue processors follows the strategy presented in [74], which proposes to take into consideration the Hamming distance among the bit string of the register encodings when testing the access ports. The algorithm for single-issue is described in the following steps. Successively it is discussed how to adapt it for dual-issue processors:

1. Preliminary setup operations:
   (a) Selection of 2 register groups, where in each group the encodings of all registers have a hamming distance higher than one bit with respect to each other. As an example two possible groups of two possible groups considering 8 accessible registers, r0-r7 is:
   **Group A:** r1, r2, r4, r7
   **Group B:** r0, r3, r5, r6
   (b) Identification of instructions exciting the register file read and write ports, referred as *Read1*, *Read2*, and *Write*, respectively. As an example, the *add* instruction between registers presented above accesses *Write* with the first operand, *Read1* with the second, and *Read2* with the third. Other instructions can be used as well.
   (c) Definition of two complementary patterns, which are propagated through access ports. An example is:
   **Pattern A:** 0101...01
   **Pattern B:** 1010...10
2. Propagate *Pattern A* through the logic connected to the registers in *Group A*, and *Pattern B* through *Group B*:
   (a) load all registers in *Group A* with instructions that write the *Pattern A* value through *Write* port;
   (b) load all registers in *Group B* with instructions that write the *Pattern B* value through *Write* port;

   (c) read all registers with instructions that read from *Read1* port, and accumulate a signature value;
   (d) read all registers with instructions that read from *Read2* port, and accumulate a signature value.
3. Propagate *Pattern B* through the *Group A* logic, and *Pattern A* through the *Group B* logic, the same as step 2 but with inverted values.

A possible implementation of the proposed algorithm for single-issue processors is presented in Table 2.9, where *add* instructions are used to bring data to the ALU through the register file read ports.

### Dual-issue Version

The proposed algorithm is modified into a dual-issue version, which is parallelized on the two pipelines with a careful attention given to the scheduling of each instruction (see Section 2.3.2). The proposed modifications are the following:

1. In the setup phase, the two groups of registers presented in the single-issue version (which are split according to the Hamming distance) are further split into two sub-groups, randomly. For example, the 8 registers in *Group A* and *Group B* of the previous example can be split as follows:

   **Group $A_1$:** r1, r2
   **Group $A_2$:** r4, r7
   **Group $B_1$:** r0, r3
   **Group $B_2$:** r5, r6
2. The propagation steps of the single-issue version of the algorithm are split and parallelized on the two pipelines, each one acting on a sub-group, accordingly:

   **Pipeline A:** *Pattern A* in *Group* $A_1$, *Pattern B* in *Group* $B_1$
   **Pipeline B:** *Pattern A* in *Group* $A_2$, *Pattern B* in *Group* $B_2$

   Since each pipeline covers half the total amount of registers, the step is then repeated with the groups inverted on the two pipelines:

   **Pipeline A:** *Pattern A* in *Group* $A_2$, *Pattern B* in *Group* $B_2$
   **Pipeline B:** *Pattern A* in *Group* $A_1$, *Pattern B* in *Group* $B_1$
3. As in the algorithm described for single-issue processors, to complete the test, the propagation steps are finally repeated with inverted patterns, i.e., *Pattern B* in the two *Group A* sub-groups, and *Pattern B* in *Group A* ones.

Table 2.9 Implementation of the single-issue version of the basic test algorithm for a register file with 8 registers

| Steps | Instruction | *Read1* input | *Read2* input | *Write* output |
|---|---|---|---|---|
| (1) | Set *patternA* equal to 55555555h | | | |
| | Set *patternB* equal to AAAAAAAAh | | | |
| (2a-2b) | load r1 , patternA | – | – | 0101...01 |
| | load r2 , patternA | – | – | 0101...01 |
| | load r4 , patternA | – | – | 0101...01 |
| | load r7 , patternA | – | – | 0101...01 |
| | load r0 , patternB | – | – | 1010...10 |
| | load r3 , patternB | – | – | 1010...10 |
| | load r5 , patternB | – | – | 1010...10 |
| | load r6 , patternB | – | – | 1010...10 |
| (2c-2d) | add r1 , **r1** , **r1** | 0101...01 | 0101...01 | – |
| | add r1 , **r2** , r1 | 0101...01 | – | – |
| | add r1 , **r4** , r1 | 0101...01 | – | – |
| | add r1 , **r7** , r1 | 0101...01 | – | – |
| | add r1 , r1 , **r2** | – | 0101...01 | – |
| | add r1 , r1 , **r4** | – | 0101...01 | – |
| | add r1 , r1 , **r7** | – | 0101...01 | – |
| | add r0 , **r0** , **r0** | 1010...10 | 1010...10 | – |
| | add r0 , **r3** , r0 | 1010...10 | – | – |
| | add r0 , **r5** , r0 | 1010...10 | – | – |
| | add r0 , **r6** , r0 | 1010...10 | – | – |
| | add r0 , r0 , **r3** | – | 1010...10 | – |
| | add r0 , r0 , **r5** | – | 1010...10 | – |
| | add r0 , r0 , **r6** | – | 1010...10 | – |
| | store r1 , memory | | | |
| | store r0 , memory | | | |
| (3a-3b) | Repeat (2a-2b) with inverted patterns[(a)] | | | |
| (3c-3d) | Repeat (2c-2d) | | | |

[(a)] Instructions using *patternA* use *patternB* and vice-versa.

(a) Wrong implementation

```
1 load  r1 , patternA          ; Write port
2 add   r7 , r1 , r7           ; EX -to-EX forwarding
3 add   r7 , r7 , r1           ; MEM -to-EX forwarding
```

(b) Correct implementation

```
1 load  r1 , patternA          ; Write port
2 nop                          ; Avoid EX -to-EX forwarding
3 nop                          ; Avoid MEM -to-EX forwarding
4 nop                          ; Avoid WB -to-EX forwarding
5 add   r7 , r1 , r7           ; Read1 port
6 add   r7 , r7 , r1           ; Read2 port
```

Fig. 2.15 Effect of data-dependencies on the access to the register file read ports. Wrong implementation (a) and correct version by means of nop instructions (b).

A possible implementation of the proposed algorithm for dual-issue processors is presented in Table 2.10, as an extension of the single-issue version in Table 2.9.

To correctly implement the presented algorithms, even for single-issue processors, data dependencies among instructions should be avoided. In fact, each of the specific test patterns is supposed to pass through the access ports of the register file, rather than through feed-forward paths, which is the case when the instruction is affected by data dependencies with the previous instructions. Even though the final signature is the same, the effectiveness of the test program is compromised.

To comply with this undesirable situation, a sequence of instructions may be introduced to wait for register write-back. These extra instructions will be referred as *padding* and can be implemented as *nop* instructions.

As an example, a 5-stage pipeline is considered and a code snippet implementing the propagation of a pattern through a single register (see Fig. 2.15). In an implementation not taking into consideration data dependencies, as in Fig. 2.15a, the register *r1* is loaded with a the pattern value (1), which is then used as input operand of the following two instructions (2,3); however, instead of retrieving the value from the register file, the processor feed-forwards the result from the EX stage to the instruction 2 and from the MEM stage to the instruction 3.

The correct implementation of the desired code, as listed in Fig. 2.15b, requires to add a *padding* composed of three *nop* instructions (2-4) after the instruction

Table 2.10 Implementation of the dual-issue version of the basic test algorithm for a register file with 8 registers

| Steps | Instruction | Pipel. | *Read1* input | *Read2* input | *Write* output |
|---|---|---|---|---|---|
| (1) | Set *patternA* equal to 55555555h | | | | |
| | Set *patternB* equal to AAAAAAAAh | | | | |
| $(2a\text{-}2b)_1$ | load r1 , patternA | A | – | – | 0101...01 |
| | load r4 , patternA | B | – | – | 0101...01 |
| | load r2 , patternA | A | – | – | 0101...01 |
| | load r7 , patternA | B | – | – | 0101...01 |
| | load r0 , patternB | A | – | – | 1010...10 |
| | load r5 , patternB | B | – | – | 1010...10 |
| | load r3 , patternB | A | – | – | 1010...10 |
| | load r6 , patternB | B | – | – | 1010...10 |
| $(2c\text{-}2d)_1$ | add r1 , **r1** , **r1** | A | 0101...01 | 0101...01 | – |
| | add r4 , r4 , r4 | B | 0101...01 | 0101...01 | – |
| | add r1 , **r2** , r1 | A | 0101...01 | – | – |
| | add r4 , r7 , r4 | B | 0101...01 | – | – |
| | add r1 , r1 , **r2** | A | – | 0101...01 | – |
| | add r4 , r4 , r7 | B | – | 0101...01 | – |
| | add r0 , **r0** , **r0** | A | 1010...10 | 1010...10 | – |
| | add r5 , r5 , r5 | B | 1010...10 | 1010...10 | – |
| | add r0 , **r3** , r0 | A | 1010...10 | – | – |
| | add r5 , r6 , r5 | B | 1010...10 | – | – |
| | add r0 , r0 , **r3** | A | – | 1010...10 | – |
| | add r5 , r5 , r6 | B | – | 1010...10 | – |
| | store r1 , memory | A | | | |
| | store r4 , memory | B | | | |
| | store r0 , memory | A | | | |
| | store r5 , memory | B | | | |
| $(2a\text{-}2b)_2$ | Repeat $(2a\text{-}2b)_1$ with inverted pipelines[(a)] | | | | |
| $(2c\text{-}2d)_2$ | Repeat $(2c\text{-}2d)_1$ with inverted pipelines[(a)] | | | | |
| $(3a\text{-}3b)_1$ | Repeat $(2a\text{-}2b)_1$ with inverted patterns[(b)] | | | | |
| $(3c\text{-}3d)_1$ | Repeat $(2c\text{-}2d)_1$ | | | | |
| $(3a\text{-}3b)_2$ | Repeat $(2a\text{-}2b)_2$ with inverted patterns[(b)] | | | | |
| $(3c\text{-}3d)_2$ | Repeat $(2c\text{-}2d)_2$ | | | | |

[(a)] instructions on *Pipeline A* are executed on *Pipeline B* and vice-versa

[(b)] instructions using *patternA* use *patternB* and vice-versa.

that loads the pattern value in *r1* (1); the value is then read back by the following test instructions (5,6) through the register file read ports. Clearly, the extra *nop* instructions could have been replaced by other test instructions not making use of the register *r1*. Please note that the feed-forward of register *r7* (5,6) does not influence the expected propagation of the pattern, which is handled by the register *r1*.

Finally, note that the implementation of the proposed algorithms is influenced by the specific processor architecture, as in the special case of processors that implemented asymmetric pipelines. In such processors, certain instructions can be processed only by the first pipeline, which contains the necessary functional units, and others by the second one.

As an example, a processor is considered, in which the pipeline *A* processes arithmetical and logical instructions, except for the multiply (*mul*) instructions, that are processed by the pipeline *B*. In this case, the algorithm presented for single-issue processors can be implemented in two versions: the first implementation uses ALU-related instructions to propagate patterns through $Read1_A$, $Read2_A$, and $Write_A$; the second implementation uses *mul* instructions to propagate patterns through $Read1_B$, $Read2_B$, and $Write_B$.

For the sake of completeness, the test engineer can adapt the presented algorithm for multi-issue processors. The principle of parallelizing the single-issue algorithm on the different pipelines is still valid. In a 4-way core, one solution is to further split the group of registers ($A_1$ to $A_4$, $B_1$ to $B_4$) and apply the propagation steps on the four pipelines (the number of steps is doubled with respect to the dual-issue). Another solution is to use the same algorithm of the dual-issue using two pipelines at the time, while *nop* instructions are executed by the others.

### 2.3.5 Feed-Forward Paths

In dual-issue processors, the instructions are processed in parallel by two logical pipelines, which stages are interconnected with several forwarding paths, in charge of managing the potential propagation of operands of data dependent instructions. Feed-forwarding paths could be categorized in *intra-pipeline* and *inter-pipeline* forwarding paths. For example, in a dual-issue processor with two 5-stage pipelines (see Fig. 2.16), referred to as *A* and *B*, *intra-pipeline* paths can be the following:

1. $EX_A$-to-$EX_A$ and $EX_B$-to-$EX_B$;

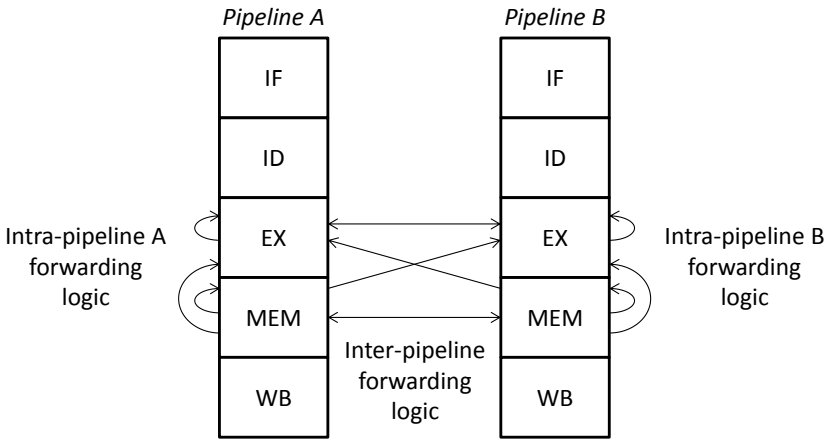



Fig. 2.16 Example feed-forward paths of in-order dual issue processors

2. $MEM_A$-to-$EX_A$ and $MEM_B$-to-$EX_B$;
3. $MEM_A$-to-$MEM_A$ and $MEM_B$-to-$MEM_B$.

Such forwarding paths only propagate operands among stages of the same pipeline. In case the processor only adopts *intra-pipeline* forwarding mechanism, data dependencies occurring within instruction issued to different pipelines are resolved in different ways, according to the processor implementation. In some processors, an interlock unit resolves data-dependencies between different pipelines by stalling both of them until the write-back of the needed data completes. Alternatively, other processors act at the scheduling time, so that the dual-issue capabilities are temporary disabled and data-depending instructions scheduled for execution in the same logical pipeline; the other is fed with *nop* instructions.

On the contrary, processors having *inter-pipeline* forwarding paths do not present such problems and propagate the operands through additional paths. Examples of *inter-pipeline* paths (see Fig. 2.16) in 5-stage pipeline processors are:

4. $EX_A$-to-$EX_B$ and $EX_B$-to-$EX_A$;
5. $MEM_A$-to-$EX_B$ and $MEM_B$-to-$EX_A$;
6. $MEM_A$-to-$MEM_B$ and $MEM_B$-to-$MEM_A$.

In all the cases, there must be no data-dependencies between two consecutive instructions that normally would be executed in parallel. In such a case, the second instruction cannot be scheduled in the second pipeline, i.e., both are processed by

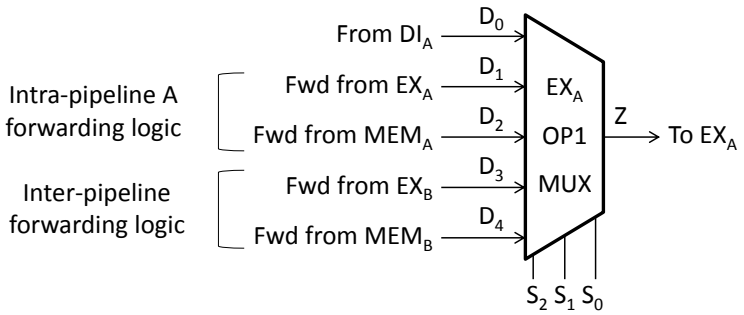



Fig. 2.17 Example of multiplexer feeding one of the operands of the execute stage.

the first pipeline, while the second processes a *nop* instruction (further details are given in Section 2.3.6).

The forwarding paths propagate data between pipeline stages through multiplexers, as already analyzed in Section 2.2. As many MUXes are used as possible destinations exist, each of them with a number of inputs equal to the number of possible data origins.

As an example, we can consider one of the two execute stages, referred as $EX_A$, which requires two operands, *OP1* and *OP2*. Each of the operands is brought to $EX_A$ through a MUX, which receives as input both the normal propagation path, selected in case no data-dependencies exist, and forwarding paths, including *intra-pipeline* and *inter-pipeline* paths. Other than the path coming from $ID_A$ stage ($D_0$ input), the *OP1* MUX may include two additional inputs (see Fig. 2.17), both from *intra-pipeline* forwarding paths:

1. from $EX_A$ stage ($D_1$ input);
2. from $MEM_A$ stage ($D_2$ input);

as well as other two, related to *inter-pipeline* paths:

3. from $EX_B$ stage ($D_3$ input);
4. from $MEM_B$ stage ($D_4$ input).

More complex multi-issue processors present larger MUXes, due to the higher number of forwarding paths.

Specific sequences of instructions are needed to cover these circuitries, including test instruction addressing forwarding paths and multiplexers.

**Setup phase**
1 Set *patternA* equal to 55555555h
2 Set *patternB* equal to AAAAAAAAh
3 `load r0, 0`
4 `load r1, patternA`

**Reset of pipelines** See Section 2.3.2

**Test phase**
```
5  add r2,r1,r0      ; Pipeline A
6  nop               ; Pipeline B
7  add r3,r1,r0      ; Pipeline A
8  add r4,r2,r0      ; Pipeline B: EX_A -to- EX_B (OP1)
9  nop               ; Pipeline A
10 add r5,r0,r3      ; Pipeline B: EX_A -to- EX_B (OP2)
```

Fig. 2.18 Propagation of a test pattern through feed-forward paths

The forwarding paths can be tested by propagating complementary values, similarly to those shown in Section 2.3.4. For instance, the snippet of code in Fig. 2.18 is able to test the $EX_A$-to-$EX_B$ forwarding paths of a processor that allows dual-issue of two arithmetic instructions.

In the example, a setup phase (1-4) prepares two registers with the *A* pattern, and then the pipelines are synchronized. The following instructions are dual-issued in the two pipelines since the data-dependencies are resolved by propagating the operands through $EX_A$-to-$EX_B$ forwarding paths. In details, the second operand of the instruction 8 requires the result of 5, as well as the third operand of 10 uses the result of 7. Some *nop* instructions are inserted in order to control the instruction scheduling; eventually, these instructions can be replaced by other (*test*) instructions not affecting the registers already considered. The test of the forwarding paths is concluded by repeating the same scheme of instructions with the pattern *B* and finally including the resulting registers in the signature computation.

An effective test requires to repeat such a procedure for each of the forwarding paths and it depends on the specific processor's implementation; however, the principle is the same of the above example. Moreover, it is applicable also to more complex multi-issue processors.

Even though the strategy is effective to test the forwarding paths, additional work is still required to test the MUXes connecting such paths to the appropriate pipeline stages (as in the example of Fig. 2.17).

**Reset of pipelines** See Section 2.3.2

**Test phase**

```
1 load  r0,0          ; IF ID EX ME WB
2 nop                 ; IF ID EX ME WB
3 xnor  r1,r0,r0      ;    IF ID EX ME WB ; D1=11...11
4 xnor  r2,r0,r0      ;    IF ID EX ME WB ; D4=11...11
5 xnor  r3,r0,r0      ;       IF ID EX ME ; D1=11...11
6 xnor  r4,r0,r0      ;       IF ID EX ME ; D3=11...11
7 add   r5,r0,r5      ;          IF ID EX ; D0=00...00
8 nop                 ; end pattern 0
```

Fig. 2.19 Test sequence that applies the pattern 0 of Table 2.6 to $EX_A$ *OP1* MUX

The test of MUXes follows the strategy described in Section 2.2.2, which is extended for dual-issue processors, where the number of MUXes is greater due to the two logical pipelines.

The $EX_A$ *OP1* MUX in Fig. 2.17 can be tested with a subset of patterns presented in Table 2.6 (patterns 4-6 and 9-11 are discarded due to the selection of unavailable inputs). Each test pattern can be transformed into a sequence of assembly instructions bringing the needed value to each of the MUX inputs (i.e., pipeline stage), and finally selecting the appropriate input (i.e., register) for the signature computation. An example of sequence applying the first test pattern is reported in Fig. 2.19.

In the example, the test instruction 7 acts on the $EX_A$ *OP1* MUX, by selecting the $D_0$ input as operand, which value is set by the instruction 1 and retrieved from the register file. The other MUX inputs take the values imposed by the other instructions (3-6). Additional *nop* instructions (2,8) are added to respect the needed instruction scheduling.

A similar scheme can be used also for the other patterns, by replacing some of the *xnor* with *xor* instructions, when the opposite value is needed, and modifying the instructions 1 and 8 with the appropriate register (e.g., *r3* register to select the $D_0$ input as operand). Other schemes can be easily constructed for each of the other MUXes. Again, the strategy can be adapted to multi-issue processors.

For the sake of completeness, it is also possible to obtain alternative test patterns from those listed in Table 2.6 by an ATPG.

### 2.3.6 Pipeline Interlocking

Pipeline interlocking is a mechanism to detect a hazard and resolve it. This mechanism is necessary to preserve original data dependencies specified in a sequence of the instructions. Moreover, an interlock prevents instructions from being executed in a wrong order.

Pipeline interlock is usually implemented by the *interlock unit* of a processor, which constantly monitors the processor's internal resources, such as pipeline's registers and computational units, required by the instructions processed in each of the pipeline stages, and performs several operations, such as the following related to in-order dual-issue processors:

**Dual-issue detection:** the interlock unit detects whether two decoded instructions can be dual-issued, issuing a *nop* instruction together with the first instruction in the negative case.

**Multi-cycle instructions:** the interlock unit assures in-order execution of instructions, by eventually stalling the pipeline in case instructions requiring more than one clock cycle are being processed.

**Structural dependency check:** the interlock unit checks functional units required by decoded instructions and eventually stalls the pipeline until the needed functional unit is available.

**Data dependency check:** the interlock unit checks data required by decoded instructions and eventually:

1. activates appropriate selection of forwarding paths, if the data is available in a register of the pipeline;
2. stalls the pipeline until the needed data is available, in case the data is being processed.

*Dual-issue detection* is typically implemented, at higher level, as a condition composed of several clauses, each one corresponding to a specific case where the dual-issue of instructions is possible (or not possible). All possible combinations of instructions that cannot be dual-issued are reported in the processor's documentation.

When a condition precluding dual-issue is detected, the control logic activates suitable control signals that propagate a *nop* instructions in the pipeline and prevent one of the instructions to be issued (it will be issued at the following time slot).

```
1 mul r1,r2,r3      ; Pipeline A
2 mul r4,r5,r6      ; Pipeline A (no dual-issue)
...
3 add r7, r1,r4
```

Fig. 2.20 Example of instruction schedule on a dual-issue processor

In case of faults affecting the dual-issue detection, this behavior can be disturbed, resulting in the following unexpected effects:

1. instructions that can be dual-issued are actually delayed and unnecessary stalls are introduced;
2. instructions, that are normally delayed, are issued.

In the first case, the performances of the processor are affected, and faults causing this are classified as "performance faults". Such faults can be detected by using some performance counters that are able for example to compute the number of stalls; these modules exist in many processors.

In the second case, the faults may change the data-flow of the instructions. In some situations, the instruction is skipped, thus the following instructions that retrieve data produced by the skipped instruction is wrong. For example, let us consider two consecutive *mul* instructions executed by a dual-issue processor equipped with a single multiplication unit, as in the code snippet of Fig. 2.20. The *mul* instructions (1,2) cannot be dual-issued; however, in case of fault, the instruction 2 may be skipped. In this specific case, the following *add* instruction (3) needs the result of 2 and will use an old and most likely wrong value in its computation. The same happens in case the instruction 1 is skipped.

In other situations, the internal registers of the pipeline may be changed, thus corrupting the results of other instructions in the pipeline that use data from feed-forward paths. This cascade effect may finally bring to unexpected run-time exceptions.

According to these considerations, the proposed functional test strategy aimed at detecting faults affecting the *dual-issue detection* feature is the following:

1. Preliminary setup operations:
   (a) to initialize (or deactivate) modules affecting the performances, such as Branch Prediction Unit (BPU), and Caches, in order to create a deterministic environment with respect to the execution time;

```
1 div  r1,r2,r3      ; Pipeline A
2 xor  r4,r5,r6      ; Pipeline B
3 xor  r2,r3,r5      ; Pipeline A (delayed)
  ...
4 add  r7,  r1,r4
5 add  r7,  r7,r2
```

Fig. 2.21 Example of instruction schedule including a multi-cycle instruction

(b) to initialize performance counters when available in the processor;
(c) to create and setup Instruction Service Routines (ISRs) to trap unexpected exceptions during test.

2. Test each pair of instructions that cannot be dual-issued:
   (a) to prepare the operands with values suitable to observe the result in the signature computation;
   (b) to perform a reset of the pipelines (refer to Section 2.3.2);
   (c) to execute the two instructions;
   (d) to use the data produces by the two instructions in the signature computation.
3. Read the value of the timer or performance counters and update the signature.

In case of in-order multi-issue processors, each pair of instructions is executed on all combinations of pipelines.

The proposed strategy can be extended to other interlocking features, such as the management of *multi-cycle instructions* and *structural dependency check* (or hazards).

In many processors, specific execution hardware is not fully pipelined, thus executing particular instructions may require additional clock cycles. This is the case, for example, of multiplications and especially divisions. In case a multi-cycle instruction is being processed, the interlock unit stalls the pipeline until the instruction (and the other one dual-issued) completes. We can consider a *div* instruction followed by other single-cycle instructions as a significant scenario, as reported in Fig. 2.21.

In the example, let us consider the *div* instruction 1 executed on the *A* pipeline and the instruction 2 that is dual-issued. Normally, the following instruction (3) is delayed to wait for instruction 1. In case of fault, the result of 1, 2, or 3 may change, thus corrupting the results of the following instructions (4,5).

```
1 load  r4 , patternA          ; Pipeline A
2 add   r3 , r0 , r0           ; Pipeline B (r0 != r4)
```

Fig. 2.22 Test sequence that applies the first pattern of Fig. 2.22 to two CMPs involved in data-dependency check

Structural hazards may exist in a dual-issue processor with asymmetric pipelines, where one of the execution units contains certain functional units. As an example, in case of a single multiplier, a multiply instruction can only be issued in one of the pipelines; thus, it is delayed in case the order is not respected. In case of fault, the instructions are corrupted similarly to the previous examples.

The step 2 of the presented algorithm can be extended by:

1. a test for each multi-cycle instruction;
2. a test for each possible structural dependency.

The last feature which is interesting to be analyzed is the *data dependency check*. The test of such a feature follows the strategy described in Section 2.2.2, which is extended for dual-issue processors. To detect if there is a data-dependency present in the pipeline, the register identifiers, which are encoded in the instruction operating code, are compared between different pipeline stages, by means of comparators (CMPs). In details, the amount of CMPs is equal to the number of input operands per each pipeline stage involved in the forwarding mechanism. For example, the $EX_A$ *OP1* (analyzed in Section 2.3.5) includes a series of CMPs that check the operand against the possible sources in every one of the pipeline stages where these values may be produced (see Fig. 2.17). Accordingly, one of the inputs of these CMPs is connected to the operand identifier (i.e., a register) encoded in the instruction and used by the instruction in one stage, while the other is the identifier of the output register on one of the possible source stages where the other instruction in the pair assumed to have a data dependence is placed.

In Section 2.3.5, systematic patterns for CMPs (see Fig. 2.10) are transformed in a sequence of assembly instructions. In the following, the strategy is adapted to dual-issue processors.

As an example, it is possible to apply the first pattern of Fig. 2.10 to the two CMPs that verify data-dependency between two consecutive instructions, which are supposed to be dual-issued, with the instructions reported in Fig. 2.22.

**Setup phase** Load all registers with different values

**Reset of pipelines** See Section 2.3.2

**Test phase**

| | | Pipe | CMP input1 | CMP input2 |
|---|---|---|---|---|
| load | r4, value | A | — | — |
| nop | | B | — | — |
| add | r2, r0, r4 | A | r4(100) | r0(000) |
| nop | | B | — | — |
| add | r1, r0, r2 | A | r2(010) | r0(000) |
| nop | | B | — | — |
| add | r0, r0, r1 | A | r1(001) | r0(000) |
| nop | | B | — | — |
| add | r0, r4, r0 | A | r0(000) | r4(100) |
| nop | | B | — | — |
| add | r0, r2, r0 | A | r0(000) | r2(010) |
| nop | | B | — | — |
| add | r7, r1, r0 | A | r0(000) | r1(001) |
| nop | | B | — | — |
| add | r0, r7, r0 | A | r7(111) | r7(111) |
| nop | | B | — | — |
| add | r0, r0, r0 | A | r0(000) | r0(000) |
| nop | | B | — | — |

Compact registers and compute test signature.

Fig. 2.23 Implementation of the test algorithm for the $EX_A$ *OP1* CMP of an example in-order dual-issue processor with 8 registers

In the example, the first input of the CMP is *r4* ($100_2$), while the second input is *r0* ($000_2$). Normally, the two instructions do not present any data-dependency, thus they are dual-issued. In case of fault, the CMP may signal a data-dependency and a stall may be inserted in the pipeline. Again, the effect of the fault may be observed by means of performance counters.

A complete application of test pattern to the $EX_A$ *OP1* CMP can be performed with a suitable sequence of instructions interleaved with *nop* instructions, aimed at scheduling the test instructions only on the pipeline A, as depicted in Fig. 2.22.

### 2.3.7 Instruction Prefetch Buffer

Another critical module in dual-issue processors is the instruction prefetch buffer. This module, included in the fetch (or prefetch) stage, fetches instructions in advance and forwards the relative Program Counter. In case of conditional branches, fetches are speculative, in accordance with the *Branch Unit*.

In principle, the *Bus Interface Unit* (BIU) loads the prefetch buffer by reading instructions from the code memory or instruction cache if available. On the other hand, instructions are read from the prefetch buffer and then decoded and issued accordingly. From the logical point of view, the prefetch buffer is a FIFO, which is composed of several slots (or entries), each one possibly holding one instruction.

The number of slots that can be filled in a cycle by the BIU depends on the bus width and, in dual-issue processors, two instructions at each cycle can be retrieved from the prefetch buffer. Dependencies among instructions and stalls in the pipeline influence this behavior.

For example, when data-dependencies exist, so that the processor issues a *nop* instruction (as analyzed in Sections 2.3.5 and 2.3.6), only one instruction is retrieved from the prefetch buffer at the successive cycle.

Moreover, in case of multi-cycle instruction, when in-order processors stall the pipelines until the instruction completes, no instructions are retrieved from the prefetch buffer, which is eventually filled with other instructions. The longer is the stall, the higher is the amount of prefetch slots that are filled.

According to these principles, instructions are moved from the higher slots (near to the BIU) towards lower slots (near to the issue stages) as follows:

- instructions are hold in the slots, in case of stalls present in the pipelines;
- one instruction is retrieved and the other slots are shifted by one position, in case the previous instruction has been single-issued;
- two instructions are retrieved and the other slots are shifted by two positions, in case the previous two instructions have been dual-issued.

Multi-issue processors (e.g., 4-way) are able to shift instructions by a higher number of positions (e.g., 4).

Structurally, a slot of the prefetch buffer includes a register, where an instruction is hold, and a MUX, which selects the source of the instruction to be hold depending

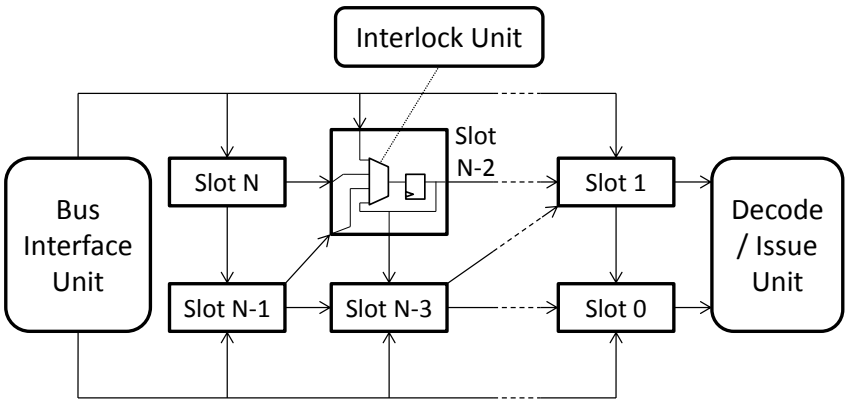



Fig. 2.24 Dual-issue prefetch buffer

on the selection signals coming from the *Interlock Unit* (see Fig. 2.24). The possible sources of instructions are:

- the content of the previous slot (single-issue case);
- the content of the second previous slot (dual-issue case);
- the content of the slot (stall case);
- the value coming from the BIU.

The proposed strategy to effectively test this module is based on two phases:

1. An appropriate sequence of instructions is executed able to stall the pipelines; this allows to load further instructions up to the higher slots.
2. A test sequence is loaded by the BIU in the slots during the stalling cycles; at the end of the stall, this sequence is propagated in the slots exciting some of the paths.

As an example, a prefetch buffer composed of 4 slots is considered, and a dual-issue processor where a division instruction stalls the pipeline for enough clock cycles to fill the prefetch buffer. In this situation, a *div* instruction is considered, followed by a sequence of test instructions, as reported in Fig. 2.25.

In the example, the *div* instruction (1) introduces a stall in the pipelines, long enough to fill all the slots of the prefetch buffer with the following instructions. The number of stalls introduced by the *div* is usually documented in the user manual, however, in case a single *div* is not able to guarantee enough stall time, then multiple *div* instructions have to be used. The *nop* instruction (2) is dual-issued with the instruction 1 and is hold in the pipeline until that instruction completes. Meanwhile,

```
1 div  r1 , r2 , r3          ; introduce a long stall
2 nop                        ; dual-issued with 1
3 nop                        ; slot 0
4 nop                        ; slot 1
5 nop                        ; slot 2
6 nop                        ; slot 3
7 load  r4 , value           ; slot 4
```

Fig. 2.25 Test sequence that propagates a pattern through the prefetch buffer slots

the prefetch buffer is loaded with the following instructions. In particular, the opcode of the *load* instruction (7) is used as the test pattern, which is then propagated in some of the prefetch buffer's paths. The other *nop* instructions (3-6) are dual-issued and do not insert any stall in the pipelines, thus the pattern is shifted by two slots at the time. At the end of the sequence, the test signature has to be updated in order to detect a possible fault. In particular, a fault in the propagation path corrupts the test instruction's opcode, while an error in the MUX selectors may produce an unexpected sequence of instructions.

By carefully modifying the example in Fig. 2.25, another path can be excited. For example, the extra *nop* instructions (3-6) can be replaced with other instructions introducing data-dependencies, in order to shift the instruction 7 by one slot at the time.

Moreover, other than simply filling the whole buffer, even the content needs to be carefully selected. In Fig. 2.25, the instruction 7 can be replaced with other opcodes, referred as *opcode A* and *opcode B*. To be effective, the two opcodes should be: 1) complementary; 2) valid instructions.

If no combination is feasible for *opcode A* and *opcode B* in the processor's instruction set, then more than two opcodes are needed, in order to propagate complementary values in each bit of the propagation path.

Finally, although most of the faults in the prefetch buffer can be covered by the above strategy, further work is needed to cover the faults on the MUXes selection circuitries. As analyzed in Sections 2.2.2 and 2.3.5, well known pattern can be used in this case (see Table 2.6). The sequence of instructions can be modified accordingly, by replacing the *nop* instructions with specific opcodes implementing the specific pattern.

### 2.3.8 Case Studies

The proposed methodology has been applied to two 32-bit in-order dual-issue processors of the family named e200, which is based on the Power Architecture™.

The first processor (codename e200z448) is embedded into a 90nm SoC manufactured by STMicroelectronics. Such a SoC is employed in safety-critical automotive embedded systems, and serves two main application areas: mid-range engine management, and automotive transmission control. The other processor (codename e200z425) is embedded into a 40nm multi-processor SoC also manufactured by STMicroelectronics. This SoC is targeted at automotive powertrain, chassis and body applications.

The e200z448 and e200z425 processors count for about 500 thousand equivalent gates each and are representative for typical industrial dual-issue processors. They share the same pipelined architecture, but present some differences. They utilize an in-order dual-issue five-stage pipeline for instruction execution, which includes two integer execution units, a branch control unit, instruction fetch unit and load/store unit, and a multi-ported register file.

In the following, the details about the implemented reset of the pipelines will be given, followed by the experimental results on the processors; a comparative analysis with non-optimized test programs will also be given.

#### Reset of the Pipelines

A set of synchronization instructions available in the ISA has been exploited. Such instructions stall the pipelines for a number of clock cycles that is enough to verify certain conditions, which are: *memory synchronization*, performed to assure that accesses to memory are ordered, and *instructions synchronization*, performed to assure that fetches to memory occur in a synchronized context.

The solution implemented in the case study (see Fig. 2.26) is the following: a memory barrier (1) orders all memory accesses; a context serialization instruction (2) waits for previous instructions (including any interrupts they generate) to complete before it executes, which purges all instructions from the processor (i.e., the prefetch buffer is flushed) and re-fetches the next instruction. Thus, the processor starts a new fetch phase, where a couple of instructions are fetched only in the case when the

```
1 msync                    ; memory synchronize
2 isync                    ; instruction synchronize
3 .align 3                 ; align multiple of 8 bytes
4 <instruction>            ; Pipeline A
5 <instruction>            ; Pipeline B
```

Fig. 2.26 Implementation of the pipeline reset in the case studies

program counter contains a double-word aligned address, otherwise it only fetches one instruction at first, and then keeps fetching two instructions at the time; for this reason, an assembler directive (3) has been used, in order to assure that the following instruction (4) is stored at a double-word aligned address; if needed, at compile-time the assembler includes additional *nop* instructions between 2 and 4. Finally, instructions 4 and 5 are fetched simultaneously.

Such a mechanism has been embedded whenever needed in the implemented algorithms, eventually simplifying it when the previous instructions do not perform load/store operations; in such cases, memory synchronization can be avoided.

### Experimental Results

The evaluation of the proposed methodology has been carried out by selecting the appropriate processor modules from the complete netlists. The Stuck-At (SA) fault model has been considered in this work.

The entire fault lists have been split and organized in groups according to the discussed algorithms. Modules under analysis are tested by adapting techniques thought for single-issue processors; thus, a comparative analysis of different test programs is presented. The e200z448 processor was extracted by the netlist of the 90nm SoC. The entire CPU module consists in 756,789 SA faults. It includes two execution units, with several computational modules used for both integer execution and vector processing. The two execution units are not symmetric, in fact only one of them includes and a division unit; moreover, two 32×32 hardware multiplier arrays are used for both two 32-bit multiply instructions (in dual-issue) and a vector multiply instruction, thus it can be fully tested without a specific dual-issue approach; such modules are not considered in this work. The total amount of SA faults that have been evaluated for the purposes of this work is 406,666.

For each computational module, test patterns were generated with ATPG and then transformed in suitable sequences of assembly instructions that are able to bring patterns towards the module's primary input and time by time compute the test signature with the values on the primary outputs. In order to highlight the importance of having a deterministic behavior, several modified versions of the final test program have been created (referred to as *no reset*), where the mechanism which implements the reset of the pipelines is disabled. In order to influence the scheduling of the instructions among the two pipelines, some *nop* instructions have been randomly added in the *no reset* programs. Experimental results obtained via fault simulation have shown a significant fault-coverage drop (up to 37 points) on the modules behaving to one of the two pipelines. Actually, it is possible to note that in the case the testing instructions are issued in only one pipeline, the obtained results in this pipeline are similar than the expected ones. However, since the processor issue is statistically unbalanced, it is possible to notice high differences in the fault-coverage results on the different pipelines (see Table 2.11). Up to 15 percentile units of fault-coverage drop are observed in the worst case on the complete fault list of computational modules. In details, about 6 points are the worst case loss for the pipeline *A* (less than 1 the best case), while for the pipeline *B* it is more than 25 (more than 6 the base case). The details of the fault simulation on the duplicated computational modules can be found in Table 2.11.

The register file contains 32 64-bit GPRs, each one accessible by 32-bit ports; thus, there are 4 write ports (2 for each pipeline), and 12 read ports (6 for each pipeline). Also in this case, the impact of the methodology specifically devised for the dual-issue capabilities has been evaluated by comparing three different test programs. The *dual-issue final* test program (see Table 2.12) represents the methodology herein explained: the idea is to recur to the reset of the pipelines, and to the propagation algorithm that runs two times, because of the two pipelines. In addition, two alternative versions of the test program have been created: the new versions do not recur to the reset of the pipelines; additionally, one of the versions only executes a single run of the propagation algorithm, as if the processor were a single-issue one. The details of the fault simulation on the register file can be found in Table 2.12. The experiments show that all the three test programs are able to test the memory bank, thus no fault-coverage loss is observed on this module. Concerning the access ports that include about 90% of the faults of the register file, a slight drop (about 5 percentile units) can be observed on the version not implementing the reset

Table 2.11 Fault simulation results on the duplicated computational modules of e200z448

| Module | Pipeline | #faults | SA FC% no reset | SA FC% final |
|---|---|---|---|---|
| Arithmetic Units | A | 2,928 | 98.58–99.37 | 99.73 |
| | B | 3,040 | 60.85–95.75 | 97.34 |
| Logic Units | A | 2,922 | 82.50–89.69 | 89.94 |
| | B | 2,914 | 59.74–80.88 | 89.57 |
| Priority encoders | A | 2,072 | 88.67–92.80 | 92.81 |
| | B | 2,046 | 54.85–77.98 | 92.42 |
| Barrel shifters | A | 7,002 | 93.99–96.22 | 96.99 |
| | B | 7,170 | 73.74–91.12 | 95.68 |
| Mask unit | A | 4,169 | 76.70–83.32 | 85.20 |
| | B | 4,204 | 58.65–69.02 | 76.07 |
| *(total)* | *A* | *19,124* | *88.58–92.52* | *93.33* |
| | *B* | *19,374* | *64.34–84.12* | *90.42* |
| | *A+B* | *38,498* | *76.37–88.29* | *91.88* |

of the pipelines (column 4), while a significant loss of about 25 percentile units is visible on the version devised for single-issue processors (column 3).

Each stage of the two pipelines stores the resulting data into specific registers, which are interconnected with several multiplexers implementing data feed-forward. Each of the two execution units retrieves data from 7 operand registers and updates a status register. Each operand registers is fed by a dedicated multiplexer, which is connected to several computational modules. Each status registers is fed by a dedicated multiplexer, which implements the feed-forward paths; both *intra-pipeline*

Table 2.12 Fault simulation results on the register file of e200z448

| Module | #Faults | SA FC% single-issue no reset | SA FC% dual-issue no reset | SA FC% dual-issue final |
|---|---|---|---|---|
| Memory Bank | 17,521 | 99.94 | 99.94 | 99.94 |
| Access Ports | 165,099 | 69.56 | 91.39 | 96.38 |
| *(total)* | *182,620* | *72.40* | *92.14* | *96.72* |

and *inter-pipeline* paths are implemented. Other dedicated registers and multiplexers exist for the memory stages, but load/store operations cannot be dual-issued in this processor, thus they are not considered in this work. Test programs for these modules have been able to reach about 77% of fault-coverage. The missing coverage is due to the presence of logic managing exceptions, which has resulted hard to control.

The pipeline interlocking unit is implemented by a specific module of the netlist, in charge of comparing the pipeline registers, identifying data-dependency and routing the feed-forward paths, identifying structural dependency, and eventually stalling the pipelines. As in the previous case, the final fault-coverage is not very high (around 69%) for the exception-related logic.

Finally, the fetch stage includes two instruction registers that are preceded a prefetch buffer composed of 20 slots. Other than the instruction opcode, each slot holds the address, which is used by the debug unit of the processor for hardware breakpoints, and prediction bits added by the branch prediction unit. The prefetch logic also include other information for an externally accessible debug module, which has limited the fault-coverage to 76%.

In order to evaluate the effectiveness of the results on forwarding, interlocking, and prefetch unit, which are strongly influenced by the presence of exception-handling and debug logic, alternative versions of the original test programs have been created, which are thought for single-issue processors. Concerning feed-forward paths, only *intra-pipeline* forwarding logic have been systematically tested. Test program for interlocking logic has been relaxed, by removing the test of faults affecting the *dual-issue detection* feature and by restricting data dependency check to single-issue. A test program devised for a single-issue processor of the same family has been finally used to test the prefetch buffer. Experimental results have shown a significant fault-coverage drop (15 to 30 percentile units) obtained with single-issue versions, as summarized in Table 2.13.

The complete test suite has a size of 50 kB and its execution takes about 62k clock cycles. The average fault-coverage is 86.98%.

Especially in controlling units, some of the logic is not directly accessible with SBST (or general purpose programs), resulting as functionally untestable. Unfortunately, it is very hard to isolate the faults related to such logic without ad-hoc formal tools, thus it is difficult to determine an upper bound value of fault-coverage. This is a typical limitation of the state-of-the-art SBST.

Table 2.13 Fault simulation results on the feed-forward paths, interlocking logic, and prefetch buffer of e200z448

| Module | #Faults | SA FC% single-issue | SA FC% dual-issue |
|---|---|---|---|
| Feed-forward paths | 97,821 | 47.22 | 77.53 |
| Interlocking logic | 20,214 | 54.48 | 69.15 |
| Prefetch buffer | 67,513 | 61.03 | 76.79 |

Table 2.14 Experimental results on e200z448 and e200z425

| Module | e200z448 | | e200z425 | |
|---|---|---|---|---|
| | #Faults | SA FC% | #Faults | SA FC% |
| Dupl. comp. modules | 38,498 | 91.88 | 42,976 | 89.94 |
| Register file | 182,620 | 96.72 | 103,456 | 99.65 |
| Forwarding | 97,821 | 77.53 | 133,523 | 75.80 |
| Interlocking | 20,214 | 69.15 | 22,114 | 67.83 |
| Prefetch | 67,513 | 76.79 | 56,915 | 85.19 |
| *(total)* | *406,666* | *86.98* | *358,980* | *85.37* |

For the sake of completeness, the same methodology has been applied to the e200z425 processor, which has been extracted from the netlist of a 45nm SoC. This processor includes a CPU consisting in 715,777 SA faults. The architecture is similar to the previous processor, with some differences due to the lack of vector extensions, and the addition of signal processing capabilities. The total amount of SA faults that have been identified for the purposes of this work is 358,980.

By following the same strategy, a set of test programs for the interested modules has been created and the average fault-coverage of 85.37% has been reached. The complete test suite occupies 47 kB and is executed in about 51k clock cycles. The cumulative results on the two processors are summarized in Table 2.14.

## 2.4 Floating Point Unit

Current critical systems commonly use floating-point computations. A floating-point unit (FPU) is the module that implements the floating-point arithmetic as defined by the IEEE standard 754 [75, 76]. This module is highly integrated in the pipeline of

general purpose processors, thus SBST techniques used to test microprocessor cores are valid also for FPUs.

In this Section, a SBST methodology to test the FPU in modern embedded processors is presented. Well-known techniques are used to test some of the features of the FPU that mainly implement the classical Floating-Point (FP) operations, e.g., FP multiplication. High fault detection can be achieved on the components that implement such features by converting Automatic Test Pattern Generation (ATPG) patterns in assembly instructions that finally compose a test program. However, other parts are better tested by using a systematic functional approach.

In the following, a FPU architectural overview is given (Section 2.4.1), follower by the proposed test strategies (Section 2.4.2). Finally, Section 2.4.3 shows some experimental results gathered on a FPU embedded in an industrial SoC.

### 2.4.1 Architectural Overview

Floating-point numbers consist of a mantissa, exponent, and sign bit. The ANSI/IEEE Standard 754-1985 [75] and IEEE 754-2008 [76] specify various FP number formats, such as single precision, expressed on 32 bits, with 1 sign bit, 8-bit exponent, and 23-bit mantissa. The standards also define values for positive and negative infinity, a "negative zero", exceptions to handle invalid results like division by zero, special values called "not a number" (NaNs) for representing those exceptions, denormalized numbers to represent numbers lower than the minimum, and rounding modes.

In modern microprocessor designs, FP arithmetic is either integrated in the main CPU (e.g., [77]) or part of a specialized core (e.g., [78, 79]).

FPUs in embedded microcontrollers usually implement a single-precision FP system, which is in some cases a reduced subset with software support in order to fully conform to the standard while minimizing the area occupation.

From a structural point of view, FPUs consist in several integer functional units, such as adders, multipliers, shifters, etc., coupled with control modules that perform normalization, rounding, and truncation. A careful analysis of testability of such modules is presented in [80].

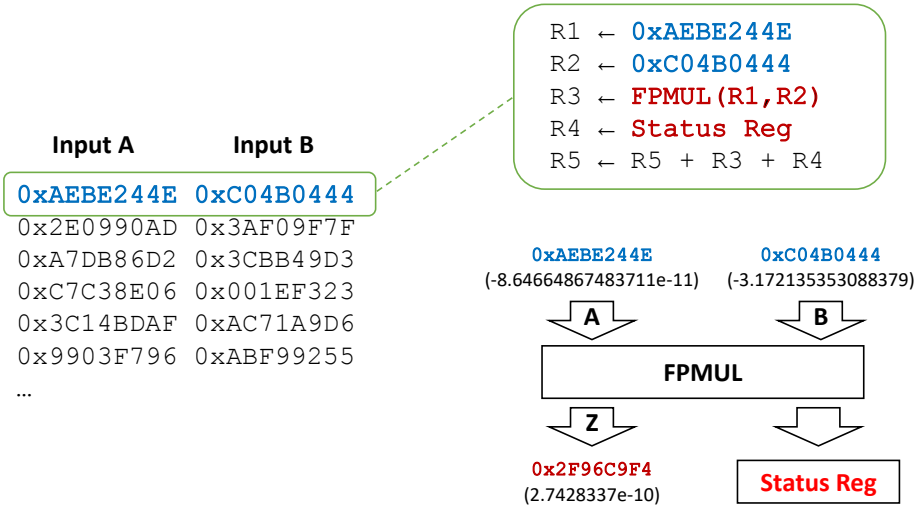



Fig. 2.27 Example of ATPG loop-based approach applied to FPU

## 2.4.2 Proposed Test Strategies

An extensive test of the floating-point unit (FPU) requires the development of a set of test programs, each one specialized in a particular feature or sub-module. Well-known techniques can be applied to different modules, while there are specific features (e.g., format conversion) that require a more accurate analysis.

### Computational Modules

Typical blocks embedded in a FPU are the functional units that implement the FP arithmetical operations, such as FP adders, FP multipliers, FP dividers, etc. Some of these modules are combinational blocks and can be extensively tested by means of ATPG loop-based approaches. These techniques rely on ATPG tools for generating effective test patterns, which are then converted into loop-based sequences of instructions [73, 81].

A simple example is shown in Fig. 2.27, which reports a list of ATPG patterns (left part of the figure) for a FP multiplier. Such patterns are converted to a sequence of instructions (right part of the figure) that loads the pattern into two general purpose registers, then executes the FP multiply instruction, retrieves the result and the status bits (from a status register), and finally computes the test signature.

A loop is typically implemented that applies all the generated patterns to the target module. A few coverage loss is typically observed when transforming ATPG patterns to SBST, due to masking effects and aliasing during the signature computation.

The conversion step is trivial for simple blocks, but can be more challenging in other cases, where selection signals have to be properly interpreted. Moreover, for sequential blocks such as FP multipliers, sequential ATPG is required. If an effective sequential ATPG engine is not available, alternative approaches such as deterministic algorithms for these modules have to be used to cover the test escapes.

### Format Conversions

Due to the different formats described in the standard, FPU in modern processors can be equipped with specialized modules in charge of dealing with format conversion.

The test of these modules can be performed by applying the conversion of the following values to different formats:

- well-known patterns, such as 0101..., 1010..., 0011..., 1100..., etc.;
- some denormalized and NaN values;
- positive and negative zero and infinite;
- several legal values, for example by reusing some values from the test of functional modules (e.g., operands of FP adder test instructions).

For example, a single-precision FPU may also support the half-precision format, as defined in IEEE 754-2008. Thus, the FPU instruction set should include specific assembly instructions that convert a value to other formats. In this example, the test of the conversion capabilities consists to convert each of the above values to half-precision, and then convert the result again to single-precision, as shown in Eq. (2.1).

$$A \xrightarrow{Single-to-Half} B \xrightarrow{Half-to-Single} C \tag{2.1}$$

In the above example, the FP value $A$, which is expressed in single-precision, is converted in the value $B$, which is expressed in half-precision. Finally, $B$ is converted in the value $C$, which is again a single-precision number. Due to the fact that the half-precision format has less bits in the exponent and mantissa fields, $B$ can become a denormalized number (and thus also $C$). Both $B$ and $C$ are used to update the

Table 2.15 Proposed special operations to detect faults related to erroneous operations

| Operations | Value in A | Value in B |
|---|---|---|
| A/B, A*B | Denormalized | Valid $\neq 0$ |
| A/B, A*B | Valid $\neq 0$ | Denormalized |
| A/B | Zero | Valid $\neq 0$ |
| A/B | Valid $\neq 0$ | Zero |
| A+B | Infinite | Valid or Zero |
| A+B | Valid or Zero | Infinite |
| A*B | Denormalized | Zero |
| A*B | Zero | Denormalized |
| A+B | Hamming 1 non-conventional | Valid or Zero |
| A+B | Valid or Zero | Hamming 1 non-conventional |

signature, which should compact the effects of possible faults altering the correct conversions.

The proposal approach is to apply the scheme of Eq. (2.1) to all the conversions that are supported by the instruction set, for example:

- FP to integer, with truncation;
- FP to integer, with rounding.

### Erroneous Operations

Operands involved in a FP operation can have different exponents, and as a consequence the operation may result in overflow or underflow conditions. Moreover, the standards specify the format for non-conventional values such as two zeros (positive and negative), two infinite values (positive and negative), NaN values, and denormalized values. FP operations using such values may also result in erroneous values.

FPUs usually signal these situations by using a status register, which can be read after a FP operation to determine its correctness. In some implementations, the FPU raises exceptions in order to alert the CPU that consequently handles errors via specialized interrupt service routines.

In order to test the circuitry involved in checking and alerting erroneous operations, the test program has to include some well-defined FP operations, as specified

in Table 2.15. The presented operations are intended to detect all faults that create a different erroneous result than the expected one (in a fault-free scenario). After each operation, the signature has to be updated not only with the resulting value, but also with the effect of the erroneous operation, i.e., by reading the status register or handling the corresponding interrupt.

Each of the operations listed in Table 2.15 uses single values for the operands A and B, except for the last two operations. For these operations, all the values with Hamming distance 1 from non-conventional values are used, in order to consider the dual cases with respect to the others in the table.

As an example, the first FP division in the table is considered, using a denormalized value (A) as the dividend and a valid value different than zero (B) as the divider. In faulty cases, the operand A may be interpreted as a valid value, thus the erroneous result is not signaled. The final signature is then corrupted.

### 2.4.3 Experimental Results

The methodology herein introduced has been applied to a SoC including a 32-bit pipelined microprocessor based on the Power Architecture™equipped with a floating-point unit (FPU). The SoC is employed in safety-critical automotive embedded systems and is currently being manufactured by STMicroelectronics.

The FPU integrates many execution units, a control unit, and few registers. Concerning execution units, the FPU includes a floating-point adder unit, a floating-point divider unit, a square root computation unit, and conversion units. The control unit absolves many duties: it receives the instruction to be decoded and sent to the right execution unit. As well, it is in charge of stalling the system if arithmetic calculation is getting longer that a clock cycle. Few registers and flip-plops permit the management of the execution flow.

In the experiments, an initial set of test programs (*CPU-SBST*) has been first evaluated. These programs were developed focusing on the main processor. The FPU was exploited for processor testing purposes, such as:

- to rise a special category of processor exceptions that are triggered by the FPU;

Table 2.16 Experimental results on the FPU

| Sub-modules | #Faults | CPU-SBST SA FC% | FPU-SBST SA FC% | Size [byte] | Duration [cc] |
|---|---|---|---|---|---|
| Add/Mul/Conv. | 53,095 | 32.4 | 89.4 | 2,684 | 3,360 |
| Divider | 12,794 | 25.2 | 87.2 | 2,848 | 21,985 |
| Square root | 6,621 | 19.5 | 96.2 | 648 | 508 |
| Exceptions | 418 | 30.1 | 86.3 | 5,500 | 4,420 |
| Status register | 1,014 | 23.0 | 100 | 2,352 | 10,776 |
| Decode | 556 | 78.4 | 100 | 460 | 216 |
| Control logic | 751 | 15.3 | 94.8 | 920 | 2,897 |
| Glue logic | 2,169 | 29.1 | 82.3 | – | – |
| *TOTAL* | *77,418* | *30.2* | *90.0* | *15,412* | *44,162* |

- to excite the processor decode unit, which identifies floating-point instructions when read from the code memory, and sends them to the FPU through the FPU interface, for further decoding and execution.

The fault simulation of the CPU-SBST resulted in the low coverage level of around 30% of Stuck-at (SA) faults.

The CPU-SBST test set has been enhanced following the strategy presented in Section 2.4.2. Final value reached by the enhanced test set (*FPU-SBST*) is around 90% of SA fault coverage. The fault coverage values on the FPU sub-modules for both test sets is resported in Table 2.16.

Table 2.17 also shows additional data, such as the size of memory occupied by the FPU-SBST and its execution time on chip, expressed in clock cycles. It is relevant to note that divider-related programs are very long because each division instruction requires an average of 14 clock cycles to complete in the considered architecture.

Finally, the test escapes can be attributed to the lower coverage of large computational modules like multiplier and divider. These sequential modules are not easy to tackle either by using an ATPG based strategy, if a powerful sequential ATPG engine is not available. Exhaustive search of operation can permit higher coverages at the cost of a long fault simulation process aimed at discovering more effective patterns.

Table 2.17 Duration and code size of test programs for the embedded FPU

| Sub-modules | Duration [cc] | Code size [byte] |
|---|---|---|
| Add/Mul/Conversions | 2,684 | 3,360 |
| Divider | 2.848 | 21,985 |
| Square root | 648 | 508 |
| Exceptions | 5,500 | 4,420 |
| Status register | 2,352 | 10,776 |
| Decode | 460 | 216 |
| Control logic | 920 | 2,897 |
| *TOTAL* | *15,412* | *44,162* |

## 2.5 Chapter Summary

This chapter presented several methodologies for the development of a suite of SBST programs specialized in the test of specific components of modern microprocessors. The proposed test strategies are systematic and their implementation has to be adapted to the specific case study.

Some of the modules tackled by this chapter, such as the decode unit, the register forwarding unit, and the pipeline interlocking unit, strongly characterize the pipeline of modern processors. These modules are not classical functional units that are mapped on well-defined assembly instructions, but require specific sequences of instructions to be tested. Moreover, the complex features characterizing multi-issue processors have been analyzed, with special focus given to in-order dual-issue processors. In this case, the hardness is due to the non-deterministic scheduling of instructions among replicated functional units. Finally, the test of a complex computational unit such as the FPU has been tackled. In this case, other than state-of-the-art techniques based on ATPG, hints have been given about how to cover hard-to-test faults.

Experimental results on both academic and industrial processor have shown the feasibility and the effectiveness of the proposed test strategies. In all the presented case studies, the stuck-at fault coverage has been higher than 85%.

# Chapter 3

# Development Flow for On-Line SBST

When SBST programs have to be integrated into safety-critical systems, the coexistence with on-board operating system (OS) impacts on the development phase with additional effort by the test engineer. In systems of this kind, such as for automotive or avionics, safety regulations impose the periodical execution of test phases during the mission, thus SBST programs are suitable to integrated in the system and executed as OS tasks.

This chapter tackles the problems of the development of SBST programs and their integration into safety-critical systems, with special emphasis given to the automotive field.

Microprocessor-based systems are employed in cars for a great variety of applications, ranging from infotainment to engine and vehicle dynamics control, including safety-related systems such as airbag and braking control. The use of microprocessor systems in safety-critical and mission-critical applications, calls for total system dependability. This requirement translates in a series of system audit processes to be applied throughout the product lifecycle, including on-line testing. The reliability requirements are met by trading off fault coverage capabilities with admissible implementation costs of the selected solutions.

The technical content of this chapter deals with the most relevant aspects of on-line test programs characteristics and their development flow.

Section 3.1 illustrates the several issues that need to be taken into account when generating test programs for on-line execution.

Section 3.3 proposed an overall development flow based on ordered generation of test programs that is minimizing the computational efforts. Algorithms proposed in the previous chapter are integrated in the generation flow and combined to minimize the overall development time.

Section 3.2 provides guidelines for allowing the coexistence of SBST with the mission application while guaranteeing execution robustness.

Finally, Section 3.4 shows the results that have been collected on industrial case studies. The impact of on-line requirements is evaluated on automotive Systems-on-Chip (SoCs) manufactured by STMicrolectronics. Experimental results also demonstrate that the development of SBST becomes unfeasible on processors with a significant dimension, without careful planning for proper resource partitioning and ordering.

The concepts and the results presented in this chapter have been published in [12] (about the methodology and partial results for microprocessors) and [11] (about FPU).

## 3.1 On-Line Constraints

SBST is widely perceived as a proper method for accurate and non-invasive autonomous test. In a few words, a test program runs and eventually detects misbehavior by simply exercising the processor functionalities. This process intrinsically respects power constraints, since test programs are executed under the same conditions of the mission mode. SBST does not ask for additional test circuitries, and is quite cheap in terms of features and commodities required to the test equipment.

When dealing with on-line SBST, test programs have to share processor resources with the mission application, i.e., the OS which is managing mission's tasks; this coexistence introduces very strong limitations compared to manufacturing tests:

1. SBST programs need to be compliant with a standard interface, enabling the OS to handle them as normal processes. This interface must guarantee processor status preservation and restoration, even in case of higher priority requests (e.g., preemption).
2. SBST programs need to be generated following execution time constraints, due to the resources occupation that can be afforded by the mission environment.

In particular, this is strictly required when a test cannot be interrupted because using critical resources (e.g., special purpose registers).

3. There is a strong limitation in terms of memory resources, due to the mission code and data characterizing the OS. To face this issue, it is usually recommended to implement the following guidelines:
   (a) to develop SBST as a set of precompiled programs stored in binary images to be run along mission mode, possibly scheduled and loaded by the operating system;
   (b) not to refer to any absolute addresses when branching, as a result, the test code can be stored in the memory, copied, and launched from any location without any functional or coverage drawback (this is usually referred to as *code relocation*);
   (c) not to refer to absolute addresses when accessing the data memory (i.e., *data relocation*);
   (d) to identify possible memory constraints from the point of view of the OS restrictions, and indispensable locations to be reserved for test purposes.

Moreover, targeting effort reduction, the test should be created also taking into account the characteristics of the general processor family, in order to reduce code modifications when transferring SBST programs to another processor core belonging to the same family. The next sections face these questions and provide some guidelines for easily taking early decisions.

On-line execution constraints may in same cases limit the effectiveness of the usual methodologies for writing test programs, which has to be rethought to fit to the real system.

## 3.2 Execution Management

The inclusion of SBST routines in the mission environment is a critical issue. To face the problematic aspects of this integration, after the generation, three major points related to test program execution are considered:

1. cooperation with other software modules, usually related to the mission environment such as the OS;
2. context switching and result monitoring;

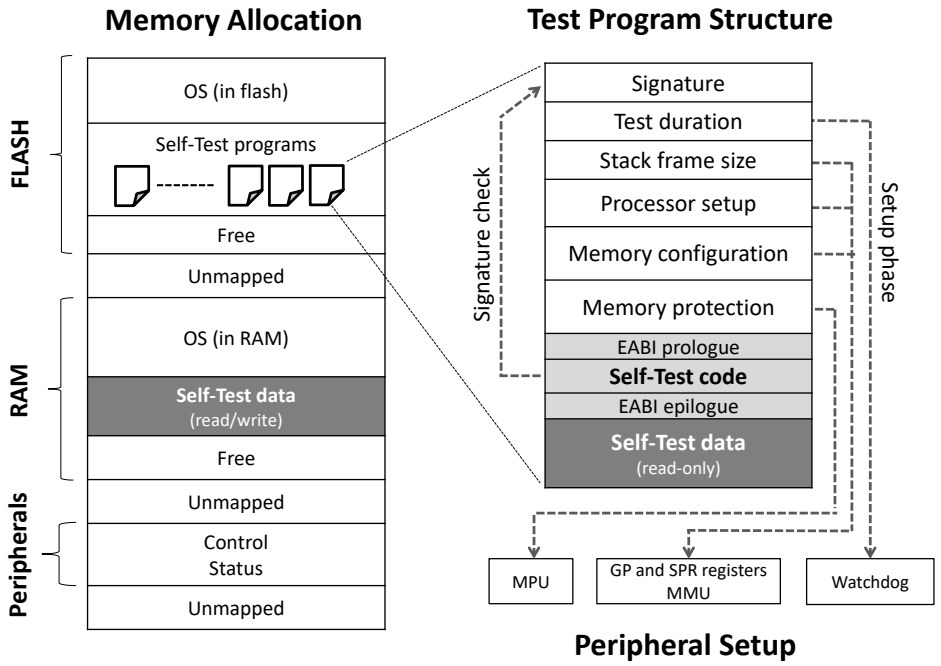



Fig. 3.1 Test program encapsulation and loading for execution phase

3. robustness in case of faulty behavior, which is strictly related to interruption management.

### 3.2.1 Test Encapsulation

Considering the cooperation with other software modules, such as the threads launched by the OS, the test program suite needs to be constructed by including key features enabling the test to be launched, monitored, and eventually interrupted by higher priority processes of the mission management system.

The test program has to be carefully structured, in order to configure memory areas and peripheral resources for test purposes, as also graphically depicted in Fig. 3.1. All the test programs are normally stored and executed in the flash memory.

In order to be compliant with the mission software environment, a viable and strongly suggested solution for the development of the OS and software modules for automotive embedded microcontrollers is the adoption of the Embedded-Application Binary Interface (EABI). Widely used EABIs include PowerPC [82], ARM EABI2 [83], and MIPS EABI [84]. EABIs specify standard conventions for file formats,

data types, register usage, stack frame organization, and function parameter passing of a software program. Thus, every test program includes an EABI prologue and epilogue (see Fig. 3.1), in charge of saving and restoring the mission status.

The EABI frame is created by the test code as soon as it begins. Thus, any scheduler can launch the test execution, e.g., the scheduler available in the OS hosting the test routine. Moreover, extra information should be included to allow a test scheduler to perform a proper running environmental setup.

Such additional information, or metadata, encompasses:

- stack frame size;
- special purpose register setup;
- memory protection setup;
- test duration.

Metadata are used by the test program during different phases, some at the setup:

1. duration time (i.e., watchdog setup);
2. stack frame size (i.e., space available for mission configurations to be saved and local variables of the test program);
3. processor setup (i.e., special purpose register ad-hoc values);
4. memory configuration (i.e., virtual memory initialization);
5. memory protection (i.e., to manage wrong memory accesses through exceptions);

and others at the execution end:

6. signature (i.e., test results check).

Such a memory structure can be also stored in the mass memory until it is loaded to be run from any portion of the available memory, according to the code relocation feature already described in Section 3.1.

### 3.2.2 Context Switching to Test Procedure

Test programs structured as described Section 3.2.1 can be integrated into any mission OS as normal system tasks. Proper context switching is managed by the EABI interface; additional setup may be required, according to the characteristics of

the test program, and it is managed internally by the test programs themselves, by means of the additional metadata.

Three general cases can be identified, each one demanding for proper metadata to be used in setup procedures:

**Run-time tests** They can be interrupted by mission requests; usually used to cover computational modules such as arithmetic modules.

**Non-exceptive tests** They require the manipulation of SPR register, such as for testing the Register File.

**Critical tests** They intentionally raise interrupts and make use of peripheral cores, such as a procedure for testing modules managing software exceptions.

*Run-time tests* are the easiest to manage: they only require creating a stack frame according to EABI compliancy; stack frame size is minimal. It is suggested to execute this kind of tests with low privileges, i.e., user mode, because they shall never request interruptions or privileged instruction execution in the fault-free scenario. No other special setup is required. EABI compliancy can be satisfied during the execution, and other OS threads can preempt the test execution.

*Non-exceptive tests* are slightly less easy to manage because these require resources that are not allowed to be directly used in the EABI context. For this category, additional test setup steps have to be executed before running:

- disable the external interrupts to avoid preemption;
- save all special and general purpose registers in a larger stack frame memory area;
- modify the content of special and general purpose registers according to the processor setup information.

Moreover, some closing operations are needed at the conclusion of the test execution to restore the initial configuration. During the execution of these tests, no preemption is allowed because the compliancy with the EABI standard cannot be guaranteed.

When considering *Critical tests*, other than saving-restore all registers and disable external interrupt sources, more restrictive requests have to be accomplished:

- the Interrupt Vector Table (IVT) and the related registers in case an alternative IVT is required for testing purposes;

- the current control and status registers of the used peripheral modules, such as the interrupt controller configuration and the MMU.

### 3.2.3 Interruption Management and Robustness

Interrupt mechanisms, which are managing synchronous and asynchronous exceptions, need to be handled with extreme care, because they are not only intentionally raised for testing purposes.

In the SBST scenario, three types of exception can be identified:

- intentionally provoked exceptions, i.e., to test processor exceptions;
- unexpected, induced by an erroneous execution that is provoked by a faulty configurations;
- mission mode interruptions.

Intentionally provoked interruptions can be both synchronous or asynchronous. Situations like system calls, illegal memory access, illegal instructions, and privilege related operations are synchronous, as they are forced by the code itself. On the other hand, asynchronous interruptions are raised through operations with peripheral cores.

To test exceptions it is therefore necessary to both rise and manage them (see Fig. 3.2). If the logic managing the interrupt has not been corrupted by a fault, each single forced exception is correctly managed, meaning that a test Interrupt Service Routine (ISR) is accessed. Such an ISR is configured at the time when the scheduler prepares the environment for the test program execution and replaces the mission ISR with the test ISR.

The code included in the test ISR is also used to add significant contents into the signature, e.g., the processor's status registers. In presence of a fault, the execution flow may be altered so that an exception, which is intentionally scheduled, actually is not raised. In this case, the signature is not updated by the test ISR, and the right signature value is not produced.

Furthermore, the exception management is also crucial for detecting faults producing execution flow deviations that lead to an unexpected processor's internal status or cause unexpected synchronous interruptions. Typical cases are a legal instruction wrongly decoded as an illegal instruction format, or an illegal memory

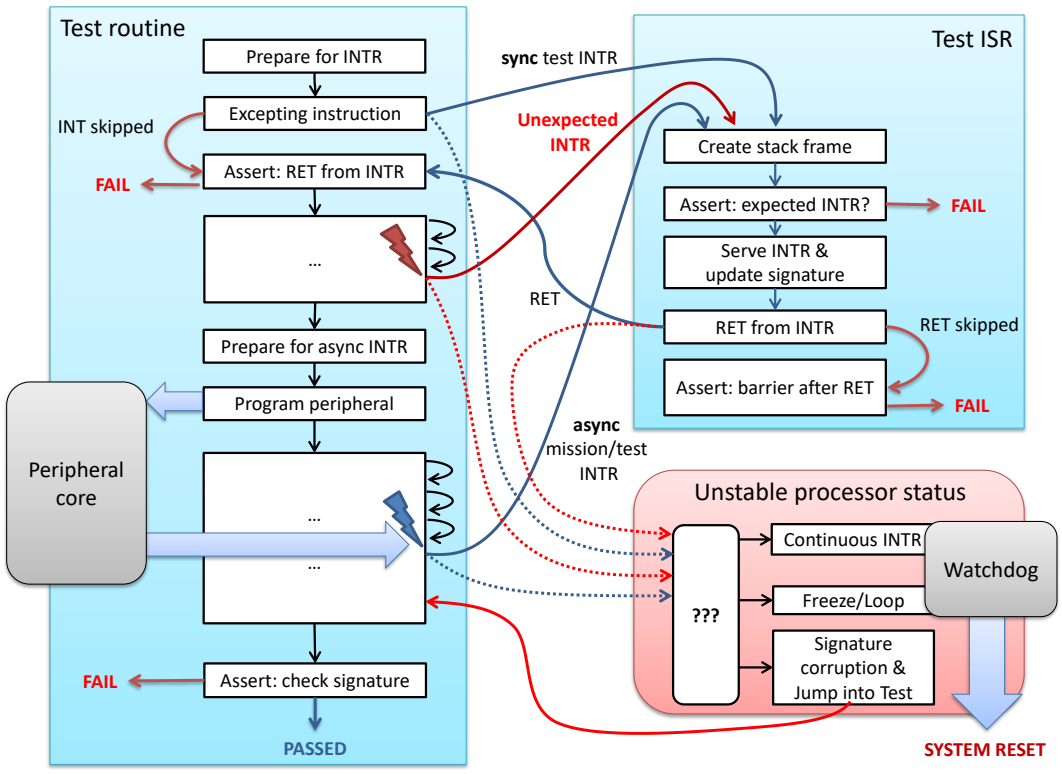



Fig. 3.2 Expected and unexpected exception management scenario

access raising an interruption by the memory protection unit mechanism (such cases have been also analyzed in Section 2.1). If this situation occurs during the execution of any test program, the test ISR should ideally be able to recover such an erroneous (due to the fault) execution flow and to record the wrong behavior observed. Some countermeasure can be adopted to identify unexpected interrupt requests, such as performing an assertion in the ISR prologue to check a password stored into a GPR before the interrupt is intentionally raised. A similar method is implemented for checking the correct return from interrupt, e.g., by completing the test execution with an assertion.

This technique is making the test code quite robust, but further work is needed if the processor status becomes unstable, resulting in spurious and repeated exceptions or endless loops. In the latter case, an external mechanism have to be implemented in order to bring the system into a safe status, i.e., by programming a watchdog timer before running the test.

These cases are graphically shown in Fig. 3.2, where solid lines are showing expected interrupts while dashed ones are showing the effect in case the processor status is unstable. During the execution of run-time test programs, mission interrupts need to be identified and served as soon as possible, i.e., passed to the mission OS. The correct implementation of the EABI permits to easily manage this case.

## 3.3 Development Flow

The major cost in the development of SBST is the computational effort required to generate test programs: the fault grading process [51], which evaluates the effectiveness of a test program (by assessing the fault coverage), represents a severe bottleneck. Moreover, the cost for developing the test program infrastructure described in the previous sections and required by the on-line execution is not negligible. For instance, for a medium sized embedded processor with about 200k stuck-at faults, fault simulating 1 ms program may require some 3 days on a 2GHz processor.

Such a cost becomes unsustainable if the generation process is iterative and produces many programs before achieving a good coverage [28]. The proposed methodology is able to achieve a significant optimization of development time and resources and is based on the following principles:

**Modularity** The processor under test is not tackled as a unique module, but its sub-modules are considered separately (e.g., ALU, Control Unit, etc.); this means that the processor's fault universe is selectively divided into several smaller fault lists that are more effectively managed along the test program generation.

**Parallelization** By facing modules separately, it facilitates the distribution of the development process on the available workstations/test-engineers (see more details in Section 3.3.1);

**Side-effect** By developing a test for a specific sub-module, additional faults belonging to other sub-modules can be covered.

The strategy is based on these principles and consists on the iterated execution of two steps until all sub-modules are considered:

1. to generate, possibly in parallel, test programs for a set of carefully selected sub-modules until these are sufficiently covered (more details in Section 3.3.2);

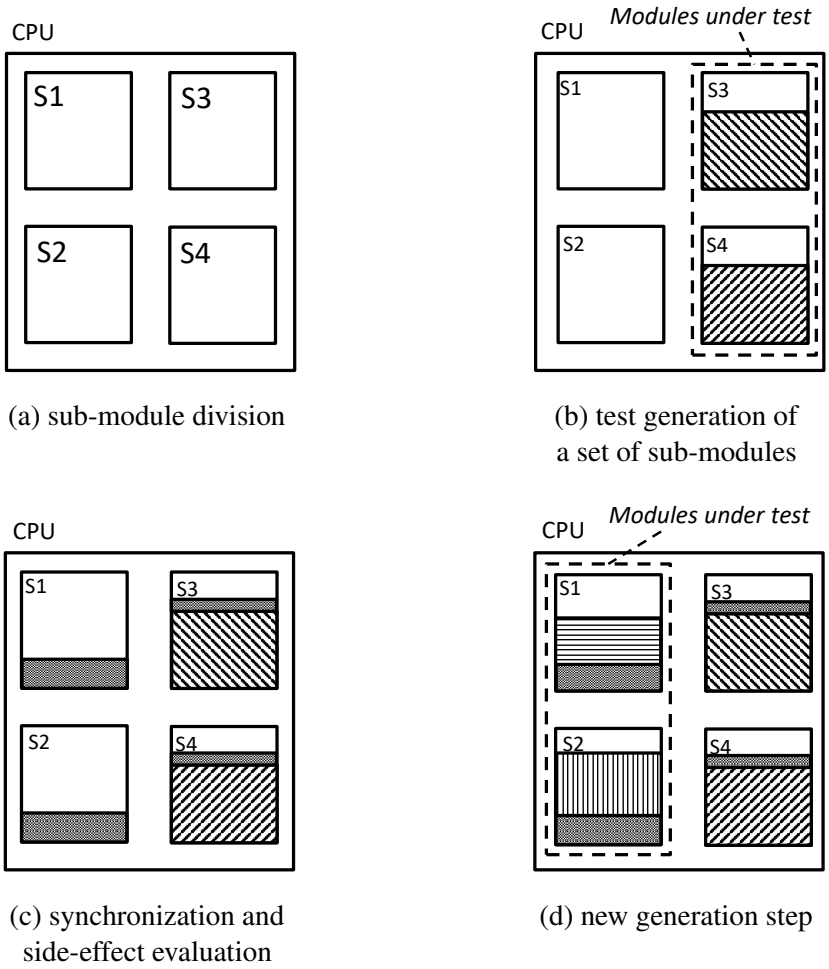



Fig. 3.3 Sub-module identification and visualization of the coverage figure evolution along the proposed generation steps.

2. to perform a *synchronization* among all the previously generated test programs, which evaluates their effectiveness over a larger fault list; to synchronize means to fault grade test programs that are generated for a specific sub-module over a different (larger) list of sub-modules.

In order to better explain the proposed flow, an example device is considered, which is split into four sub-modules: *S1*, *S2*, *S3*, and *S4* (Fig. 3.3a).

In the first step (Fig. 3.3b), two modules (*S3* and *S4*) are considered in parallel and graded separately, i.e., during the generation process for module *S3* only its own faults are considered.

As soon as the coverage of these sub-modules is satisfactory, the generated test programs are graded over the other parts of the device (Fig. 3.3c); this synchronization step brings to observe a positive side-effect on the coverage of modules *S1* and *S2*, as well as on *S3* when grading the test program for *S4*, and vice versa.

A new generation step is then started on *S1* and *S2* (Fig. 3.3d); it is worth to mention that the previous steps have been beneficial because the starting fault lists of *S1* and *S2* have been pruned before facing their generation process.

The main advantage of using this approach is a faster development of the test suite because the fault lists to be considered are significantly smaller than the complete fault list, resulting in a faster fault grading that usually takes in the order of minutes to complete. Moreover, after each synchronization step, the number of active faults in new sub-modules is reduced, again leading to a speed-up of the fault simulation process.

### 3.3.1 Resources Partitioning

According to the modularity principle, the processor is divided into sub-modules to consider *independent* fault lists in parallel, and their selection is the major issue in the preliminary phase preceding the generation effort.

For being *independent*, fault lists need to be:

- *non-overlapping*, i.e., one fault has to belong to only one fault list;
- *functionally orthogonal*, i.e., faults in the same independent fault list need to belong to modules related to one specific functionality.

The non-overlapping criterion dictates that the same fault must be considered only one time in the process; on the other hand, when dealing with orthogonality, a fault must be included in the most relevant fault list from the point of view of the functionality of the related gate.

The process of selecting the set of fault lists is not trivial and relies on:

1. manuals and documentation of the device with specific indications about the micro-architecture;
2. hierarchy of the device netlist;
3. test engineer expertise.

To identify independent fault lists, the process requires the analysis of the processor functions. Such functions are then mapped over the processor hierarchical netlist.

Most of the fault lists directly derive from a specific module, but frequently several sub-modules are combined into a single fault list if related to the same processing functionality. As an example, the faults of a multiplication unit usually constitute an independent fault list. On the contrary, there are several multiplexers that seem to be independent netlist modules, but actually compose the feed-forwarding logic in the processor's pipeline. Thus, faults belonging to these multiplexers have are grouped into the same fault list, which is functionally orthogonal and non-overlapping with other modules.

Concerning computational resources allocation, once the independent fault lists have been identified, the following rules aim at maximizing the number of fault lists to be considered in parallel:

1. fault coverage calculation for sub-modules may be allocated on a single or many threads according to:
   (a) number of available threads per CPU;
   (b) number of EDA tool licenses;
   (c) fault list size;
2. fault coverage calculation for synchronizing the results of multiple test programs among multiple sub-modules should be allocated on many threads.

### 3.3.2 Optimized Test Programs Generation Order

The side-effect principle is crucial to select the most promising order to proceed in the test program generation. The decision needs to be tailored on the specific architecture under analysis.

In the following, general guidelines for determining the generation order are provided for the most common microprocessor architectures used in automotive. In the proposed development flow, the generation order is organized according to horizontal and vertical flow rules.

Vertical flow rules demand to split the flow into consecutive *levels*, such that by testing all the modules into a given level, a large positive side-effect in terms of fault coverage is observed when moving to the next level.

The proposed methodology divides the flow into levels according to the following rules:

1. to consider first those units that can be mapped on specific assembly instructions or specific architectural programming mechanisms;
2. to continue with memorization and control flow resources;
3. to conclude with modules which functions are transparent to the programmer.

According to the presented strategy, a synchronization step is needed after completion of the currently considered level before moving to the next. This synchronization step involves all sub-modules of the next level, i.e., to reduce the size of the fault lists to be considered successively.

By looking at the problem in a horizontal manner, it is also possible to identify many parallel *branches* which are still complying with vertical requirements. This horizontal view consists in identifying branches, so that a negligible side-effect crosses branches belonging to different horizontal views.

Based on the previous rules, for a typical automotive-oriented CPU architecture, an effective development flow is based on 3 levels and 2 branches (Fig. 3.4), as in the following scheme:

**Level 1 – Branch A (1A)** *ALU*: easy to test, sub-modules that ask for the execution of specific arithmetic and logical instructions. Side effect will be maximized towards the REGISTER FILE by an accurate selection of registers to be used as operands and in the control flow management.

**Level 1 – Branch B (2B)** *SPECIAL*: sub-modules that encompass Exceptions Management, Branch Prediction, and Virtual Memory-related modules, e.g., the Memory Management Unit (MMU) module. These sub-modules are hard to cover, requiring specific instructions and sequences of instructions. They will produce a very large positive side-effect on ADDRESS-related modules.

**Syncronization 1** Once a sufficient coverage on sub-modules belonging to Level 1 is reached, the set of programs developed for (1A) are evaluated on the REGISTER FILE fault lists, while (1B) is graded on the ADDRESS-related modules.

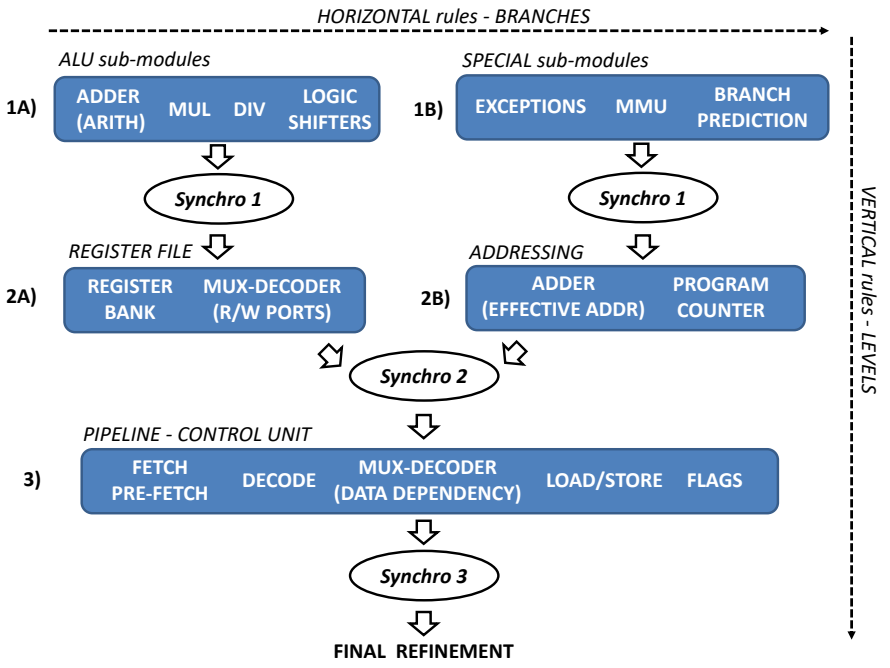



Fig. 3.4 Proposed test program development order for CPUs organized in levels and branches, and synchronization steps

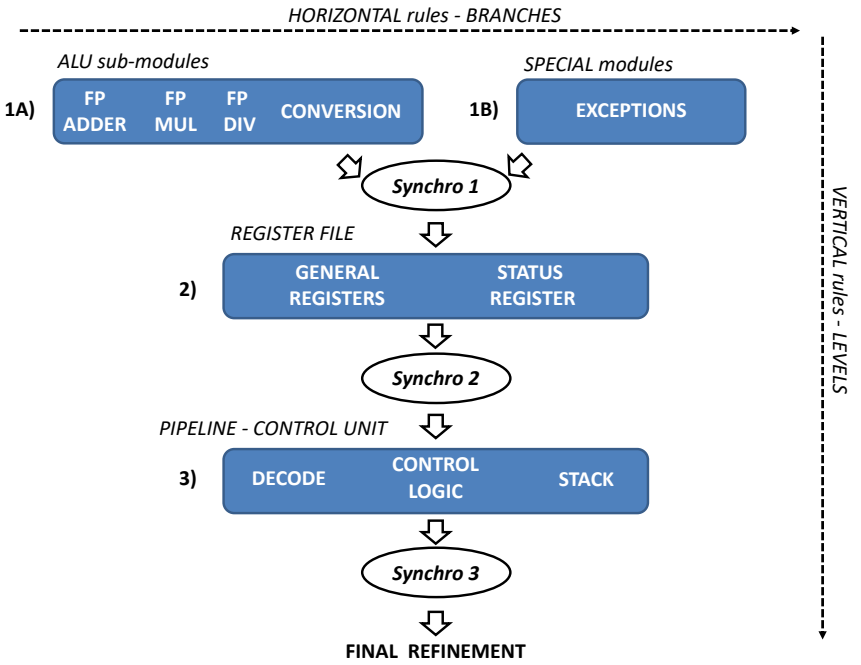



Fig. 3.5 Proposed test program development order for FPUs

As a result, the number of active faults to be then considered in Level 2 will be greatly reduced.

**Level 2 – Branch A (2A)** *REGISTER FILE*: the test of the register file module is straightforward, being several papers describing effective sequences to test (including Section 2.3.4). In the proposed generation method, it is suggested to reorder instructions and operands in order to induce the usage of DATA DEPENDENCY structures in the PIPELINE.

**Level 2 – Branch B (2B)** *ADDRESSING*: by having completed (1B), the most of the faults included in the ADDRESS-related modules, such as Branch unit, Effective Address calculation, and Program Counter, will result as already covered. This step is therefore a completion of the previous one, which is done mainly by adding memory operation and branches to specific addresses.

**Syncronization 2** After reaching a sufficient fault coverage on Level 2, the test programs developed for Levels 1 and 2 are evaluated on the PIPELINE and CONTROL UNIT modules.

**Level 3** *PIPELINE – CONTROL UNIT*: these modules are typically considered hard-to-test, but the fault lists are highly pruned by previous generation steps. Algorithms for modules such as Decode Unit, Data Dependency-related logic, or Prefetch Buffer are presented in Chapter 2.

**Syncronization 3** The entire test suite obtained along this process is evaluated on the whole processor fault universe.

The missing faults are analyzed at the end of the process and final refinement programs are added to cover corner cases and specific configurations not considered along the previous steps.

In the previous scheme, additional branches can be added when considering microprocessors equipped with special features, e.g., caches, shared memory schemes, and Floating-Point unit (FPU). Alternatively, a similar approach can be applied to the special feature under analysis, if this is complex enough to require multiple generation steps. As an example, a complex FPU can be considered. The 3-level 2-branch scheme presented above can be adapted to such a module, as shown in Fig. 3.5. In this case, ADDRESSING modules have not been considered, but the actual implementation depends on the specific case under analysis.

## 3.4 Case Studies

The methodology herein introduced has been applied to several industrial SoCs, in the frame of a collaboration with STMicroelectronics. Such SoCs are employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers and is currently being manufactured by STMicroelectronics. On-line SBST programs have been developed mainly targeting the CPU and FPU cores.

Details about the development flow for a 32-bit microprocessor based on the Power Architecture™ are given in Section 3.4.1. Results of the development flow for an embedded FPU are presented in Section 3.4.2. Finally, Section 3.4.3 shows cumulative experimental results gathered among the different processors.

### 3.4.1 Automotive Microprocessor

The case study is a 32-bit pipelined microprocessor based on the Power Architecture™ technology and designed specifically for embedded applications. The processor integrates a pair of integer execution units, a branch control unit, instruction fetch unit and load/store unit, and a multi-ported register file capable of sustaining six read and three write operations per clock cycle. Most integer instructions execute in a single clock cycle. Branch target prefetching is performed by the branch unit to allow single-cycle branches in many cases. It contains a Memory Management Unit (MMU) and a Nexus Class 3 module is also integrated for external debug purposes.

The 32-bit processor utilizes an in-order dual-issue five-stage pipeline for instruction execution. These stages are:

1. Instruction Fetch
2. Instruction Decode/Register file Read/Effective Address Calculation
3. Execute 0/Memory Access 0
4. Execute 1/Memory Access 1
5. Register Write-Back

The stages operate in an overlapped fashion, allowing single clock instruction execution for most of the available instructions.

The integer execution unit consists of a 32-bit Arithmetic Unit (AU), a Logic Unit (LU), a 32-bit Barrel shifter (Shifter), a Mask-Insertion Unit (MIU), a Condition

Register manipulation Unit (CRU), a Count-Leading-Zeros unit (CLZ), a 32x32 Hardware Multiplier array, and result feed-forward hardware. Integer EU1 also supports hardware division. Most arithmetic and logical operations are executed in a single cycle with the exception of multiply, which is implemented with a 2-cycle pipelined hardware array, and the divide instructions. A Count-Leading-Zeros unit operates in a single clock cycle. Two execution units are provided to allow dual-issue of most instructions. Only a single load/store unit is provided, and only a single integer divide unit is provided, thus a pair of divide instructions cannot issue simultaneously.

### Experimental Results

To test processor through software is a deeply explored field; therefore, in many cases the technique utilized has been borrowed from the literature and adapted to cover the specific modules of the considered processor core. Table 3.1 reports the list of generation techniques employed to achieve the high fault coverage of each processor sub-module. On selecting these techniques, some of the most important proposals regarding test program generation available in today's literature has been used.

Concerning automatic approaches, both *ATPG*-based and optimization techniques based on *Evolutionary Algorithms* have been utilized (refer to Section 1.5). *Deterministic* techniques refer to available solutions that exploit the sub-module regularity in order to propose a well-defined test algorithm. Finally, rows labeled as *Manual* refer to pure manual strategies performed by the test engineer exploiting the processor user manual, the ISA, as well as the available HDL processor descriptions.

As discussed in Section 3.1, both the duration and the size of each test program are on-line requirements that may vary depending on the mission application and physical limits of the device (e.g., the available memory space). In the specific case study, the limitations were given both in terms of duration of single programs, and of overall occupation of the complete test suite. In particular, the maximum duration of a single program labeled as *run-time test* should have not exceeded 512 clock cycles, while the Flash memory area reserved for test purposes was limited to 256kB.

To match these constraints, every method needs to be tailored opportunely:

Table 3.1 SBST strategies used along the generation process for the automotive microprocessor

| Sub-module | Techniques | References |
|---|---|---|
| Arithmetic adders<br>Divider<br>Logic unit<br>Multiplier<br>Shifters | Deterministic +<br>Constrained ATPG +<br>Evolutionary | [28, 27, 85, 39],<br>Section 2.3.3 |
| Exceptions | Deterministic | [86] |
| Branch prediction unit | Deterministic | [26] |
| Timers | Manual | |
| Register bank<br>Register ports | Deterministic | Section 2.3.4 |
| EA adder<br>Load store unit<br>Program counter | Deterministic +<br>Evolutionary +<br>Manual | [55] |
| Forward/Interlock unit | Deterministic | Sections 2.3.5 and 2.3.6 |
| Decode unit<br>Control registers<br>Status registers | Deterministic | Section 2.1 |
| Fetch unit | Deterministic | Section 2.3.7 |

- *ATPG*-based generation methods can be constrained by asking the automatic engine for high compression and limiting the generation to a maximum number of patterns; the generated patterns may be eventually transformed into many test programs compliant with duration constraints;
- fitness values used in the *Evolutionary* computation experiments include program size and length measurements; in such a way, the programs exceeding the imposed limitations can be discarded;
- *Deterministic* and *Manual* generation methods require additional efforts by the test engineer to fit the programs length and size into the required constraints; more easily, if too long, they can be split into several shorter programs.

Code characteristics fitting on-line requirements were also considered in all cases, such as having relocatable code (absolute branches and access by absolute addresses to memory locations were not allowed) and resorting to a limited portion of memory space (1kB) reserved for test.

As described in Section 3.2, each generated program was encapsulated into the EABI standard frame and included the additional code sequences that guarantee the test robustness. For the specific case of study, the EABI compliant frame is accounting for very few instructions at the beginning of procedures (e.g., 3-5 instructions); this number increases whereas:

- extra registers have to be saved before being used and finally restored to their original values (e.g., non-volatile registers, or special purpose registers, such as the Microprocessor Status Register);
- memory resources need to be protected (e.g., Memory Protection Unit is exploited);
- on-board peripherals need to be programmed for test robustness (e.g., watch-dog timers, performance counters).

The number of additional instructions required to afford robustness other than mere compliancy with EABI standards was equal to about 20. Additionally, to further enforce robustness, additional instructions were added when a context switching is forced via exception. Concerning the development flow, the fault list generation and the adopted generation order followed the generic indications provided in Section 3.3.

The fault lists were generated mainly according to the processor functionalities which are directly related to specific modules in the netlist hierarchy. There were some exceptions, due to the fact that the considered device is a dual-issue processor. Thus, replicated arithmetic modules, such as adders, were grouped together in a single fault list; in a different way, the data-forwarding unit is composed of several multiplexers, which faults are jointly considered. Another interesting case of resource partitioning is related to the multi-port register file that is contributing with two fault lists, the register bank and the register ports (decoders and multiplexers); this is due to both the fault list size, and the different functionalities.

Table 3.2 shows the evolution of the stuck-at (SA) fault coverage along the development flow. The final fault coverage is 87.23% on the fault list that includes around 750k SA faults.

For different reasons, there are modules not well covered:

1. modules managing exceptions: because it is not possible to purposely exercise all of them (e.g., it is not possible to force a bus error which asks the exception unit to intervene);

Table 3.2 Coverage evolution along the development flow for the automotive microprocessor

| Sub-module | #Faults | Single 1A SA FC% | Synchro 1A SA FC% | Single 1B SA FC% | Synchro 1B SA FC% | Single 2A SA FC% | Single 2B SA FC% | Synchro 2A+2B SA FC% | Single 3 SA FC% | Synchro 3 SA FC% |
|---|---|---|---|---|---|---|---|---|---|---|
| Arith. adders | 5,996 | **95.03** | 97.93 | – | – | – | – | 98.27 | – | 98.52 |
| Divider | 19,018 | **83.98** | 83.98 | – | – | – | – | 83.99 | – | 84.82 |
| Logic unit | 22,294 | **76.32** | 78.57 | – | – | – | – | 78.70 | – | 83.34 |
| Multiplier | 78,094 | **91.18** | 92.62 | – | – | – | – | 92.62 | – | 95.90 |
| Shifters | 14,172 | **87.95** | 92.96 | – | – | – | – | 93.97 | – | 96.32 |
| Exceptions | 40,718 | – | – | **66.08** | 67.17 | – | – | 68.16 | – | 72.48 |
| Branch pred. | 24,489 | – | – | **70.91** | 70.95 | – | – | 72.67 | – | 72.67 |
| Timers | 7,683 | – | – | **88.21** | 88.43 | – | – | 88.46 | – | 89.70 |
| Register bank | 83,764 | – | *71.21* | – | – | **84.15** | – | 89.38 | – | 92.66 |
| Register ports | 126,329 | – | *69.17* | – | – | **94.93** | – | 97.67 | – | 98.09 |
| Program cnt | 26,060 | – | – | – | *66.07* | – | **68.66** | 69.42 | – | 70.09 |
| EA adder | 5,228 | – | – | – | *66.51* | – | **92.02** | 93.75 | – | 94.57 |
| Fetch unit | 71,582 | – | – | – | – | – | – | *69.45* | **82.39** | 83.54 |
| Forw./Interl. | 84,758 | – | – | – | – | – | – | *70.95* | **84.29** | 84.82 |
| Status flags | 33,277 | – | – | – | – | – | – | *59.31* | **78.08** | 78.61 |
| Control flags | 10,328 | – | – | – | – | – | – | *64.21* | **66.83** | 69.83 |
| Decode unit | 62,876 | – | – | – | – | – | – | *50.12* | **92.46** | 93.08 |
| Load/Store | 15,971 | – | – | – | – | – | – | *73.50* | **75.42** | 76.73 |
| Glue logic | 19,425 | – | – | – | – | – | – | – | – | *63.36* |
| *TOTAL* | *756,789* | – | *36.07* | – | *9.74* | – | – | *76.87* | – | *87.23* |

2. branch prediction, program counter, and load/store units: due to the memory mapping configuration of the specific SoC, not all bits in the addressing registers can be functionally excited;
3. processor status and control registers: because many bits cannot be excited because controlling circuitries outside the processor core.

As a complement to the fault coverage measurements, the size, duration, and number test programs along the entire development flow are included in Table 3.3. At the working frequency of 150 MHz, the overall time required for executing all 73 developed tests is about 0.8ms.

It is interesting to note, how the synchronization phases produce a very strong positive side-effect over the modules not yet considered; at least the half of the faults of the modules that are going to be considered during the next generation steps were pruned from the list without any additional effort. Table 3.2 also permits to remark that the synchronization steps cause coverage improvements also for modules of the current and previous levels of the same branch, as described in Section 3.3.2.

A significant advantage in terms of grading time reduction is achieved by a proper development order which is maximizing the cascade effect. As an example of effectiveness, by adopting the proposed order, the generated test programs over the 139,574 faults of arithmetic modules included in 1A (Level 1 – Branch A) led to a positive side effects on 2A consisting in 147,029 over 210,093 faults (corresponding to about 70%), i.e., these faults are already covered without any specific generation effort for 2A. In other words, the fault simulation experiments carried on level 2A need to consider only 63,064 faults.

For the sake of completeness, the proposed generation order have been evaluated against an alternative one. Such an alternative order first considers the modules of 2A and then the ones of 1A. The 210,093 faults of the register file (2A) have been tackled by obtaining a fault coverage comparable with the results in Table 3.2 and the side-effect of such programs over the ALU (1A) has been evaluated. As a result, only 10,318 faults (corresponding to 7.4%) were already covered over the total amount of 139,574 faults of the ALU sub-modules.

The reduction in the fault list cardinality, achieved by properly ordering sub-modules and synchronizing fault lists, induces a significant time gain due to a large reduction of fault simulation efforts. The effect is not limited to the successive

Table 3.3 Number of evaluated test programs, duration, and code size along development flow for the automotive microprocessor

| Generation step | #Test programs | Duration [cc] | Code size [kB] |
|---|---|---|---|
| Single 1A | 29 | 8,840 | 23 |
| Synchro 1A | 29 | 8,840 | 23 |
| Single 1B | 8 | 19,716 | 11 |
| Synchro 1B | 8 | 19,716 | 11 |
| Single 2A | 10 | 32,634 | 36 |
| Single 2B | 8 | 28,212 | 17 |
| Synchro 2A+2B | 55 | 89,402 | 87 |
| Single 3 | 18 | 26,700 | 32 |
| Synchro 3 | 73 | 116,102 | 119 |

synchronization, but permits faster generation iterations as required by loop-based approaches (e.g., based on evolutionary algorithms).

Table 3.4 shows the elapsed time for fault simulations in two cases:

1. a simple development flow not using synchronization but simply considering sub-modules separately;
2. the proposed development flow implementing levels/branches based generation order and synchronization between levels.

All the experiments were executed on a single core of a 2 GHz processor. The resulting times would be reduced by running multi-process fault-simulations.

It is worth to notice that the fault simulation time becomes excessive if not implementing synchronization. As well, the development order is important to minimize the fault simulation efforts. In the alternative scenario in which 2A (Level 2 – Branch A) is considered before 1A, the CPU time reduction for fault simulations is only 3 hours (from about 37 to 34), which is a negligible gain if compared to that obtained by the proper ordering.

### SBST Integration into Mission OS

The suite of test programs resulting from the development phases described above was integrated in an industrial demo project for STMicroelectronics. The project handles the whole test set of programs by means of two software modules, and

Table 3.4 Fault simulation time comparison for approaches without and with synchronization

| Fault-list | Fault simulation time [hours] | | |
|---|---|---|---|
| | without synchro | with synchro | Speed-up |
| Level 1 – Branch A | 37 | 37 | – |
| Level 1 – Branch B | 122 | 122 | – |
| Level 2 – Branch A | 217 | 55 | 4× |
| Level 2 – Branch B | 72 | 23 | 3× |
| Level 3 | 630 | 195 | 3× |

provides the project integrator with a software Application Programming Interface (API), in order to include them in the mission application:

- tests for power-on: 44 test program, including *Non-exceptive* and *Critical* tests**,** scheduled by an ad-hoc software module named Boot Time Self-Test Module (BTSTM);
- tests for run-time: 29 *Run-Time* test programs handled by an AUTOSAR 4.0 Complex Driver [87] named Core Self-Test (CST) Library.

Both CST Library and BTSTM provide configuration capabilities at compile time, in such a way that the project integrator can selectively activate all programs or a subset of the entire suite. It is up to the user of the API to choose suitable test combinations and a scheduled execution order to fulfill the safety requirements of the system.

In the demo setup, after the execution of tests for power-on that takes about 0.7ms, the run-time tests were scheduled according to some specific on-line requirements:

- self-test chunks must be less than 5$\mu$s long;
- self-test interrupts the mission application every 500$\mu$s.

Along mission, using the proposed demo setup, the overall self-test length does not exceed 100$\mu$s and a complete self-test is performed in less than 2ms. Availability of the mission application is reduced by around 1% even though that self-test can be preempted at any time.

Along development, the demo test suite was encompassing several verification and validation stages towards software maturity, which were including embedded documentation of the code by means of Doxygen tags [88] that are parsed by external tool for automatically generating user manuals.

Test programs also provide services for returning test results, i.e., error codes such as AUTOSAR DEM errors and malfunctioning signatures computed by the test programs for successive inspection of failing chips. BTSTM assumes that all the available processor functionalities can be exclusively accessed for testing purposes; on the contrary, CST Library has more restrictive requirements.

For validation sakes, additional experiments aimed at emulating the in-field behavior of the system:

- to verify the fault-free behavior, a sample OS was considered that was intensively triggering mission interrupts while SBST was executed at regular intervals as described above; a physical target was programmed with such a complete software environment and left running for several hours, tracing the correctness of the test responses and liveness of the system;
- to investigate on the robustness in case of faults, a specific fault injection campaign was performed by means of complete simulation (i.e., without fault dropping) in order to classify erroneous behaviors.

Erroneous situations were classified as following:

1. self-test ends with a wrong signature;
2. self-test is not ending due to deadlock configuration;
3. self-test ends with unattended exception management, due to:
   (a) illegal instruction execution;
   (b) wrong branches to protected memory areas.

### 3.4.2 Embedded Floating-Point Unit

The case study is the embedded FPU of a SoC for safety-critical automotive application (which also includes the case study processor of Section 3.4.1) manufactured by STMicroelectronics.

In this section, the experimental results presented in Section 2.4.3 are extended with details about the development flow.

Table 3.5 provides a list of generation techniques employed to reach high coverage for each sub-module of the considered FPU.

Table 3.5 SBST strategies used along the generation process for the embedded FPU

| Sub-module | Techniques | References |
|---|---|---|
| Add/Mul/Conversion<br>Divider<br>Square root | Deterministic +<br>Constrained ATPG | Section 2.4.2 |
| Exceptions | Deterministic | [86] |
| Decode | Deterministic | Section 2.1 |
| Status register<br>Control logic | Deterministic +<br>Manual | Section 2.4.2 |

Where applicable, the constrained ATPG-based technique guarantees very good coverage values; as a drawback, a high number of patterns may be required.

For others units, it is not trivial to use the ATPG. In these cases, it is necessary to implement specific test strategy, unfortunately with manual efforts.

Similarly to the microprocessor case, a development flow composed of 3 levels and 2 branches (see Fig. 3.5) was implemented.

Table 3.6 reports the evolution of the stuck-at (SA) fault coverage along the several phases of the development flow, with side-effects evaluated over the synchronization phases. Final value reached is around 90% of SA fault coverage.

Table 3.7 shows the number of test programs developed for each FPU sub-module, as well its code memory size and execution time on chip, expressed in clock cycles.

As for the microprocessor case study, all mentioned programs are encapsulated in EABI standard frame that includes general and special purpose register settings and additional information specifically aimed at guaranteeing robustness. For instance, many faults may cause unexpected exceptions, thus a specialized vector table and interrupt service routines are implemented to intercept such defective consequences. As well, test programs that execute arithmetic floating-point operations and do not freely make use of RAM memory (i.e., the only memory they use is in the stack and in the global data sections) or intentionally raised exceptions, can be interrupted at any moment along their execution by the mission application.

Table 3.6 Coverage evolution along the development flow for the embedded FPU

| Sub-module | #Faults | Single 1A FC% | Single 1B SA FC% | Synchro 1A+1B SA FC% | Single 2 SA FC% | Synchro 2 SA FC% | Single 3 SA FC% | Synchro 3 SA FC% |
|---|---|---|---|---|---|---|---|---|
| Add/Mul/Conv. | 53,095 | **85.2** | – | 85.2 | – | 87.9 | – | 89.4 |
| Divider | 12,794 | **86.4** | – | 86.4 | – | 86.6 | – | 87.2 |
| Square root | 6,621 | **96.2** | – | 96.2 | – | 96.2 | – | 96.2 |
| Exceptions | 418 | – | 82.6 | 85.2 | **–** | 85.9 | – | 86.3 |
| Status register | 1014 | – | – | *80.0* | **95.5** | 95.7 | – | 100.0 |
| Decode | 556 | – | – | – | **–** | *89.2* | **99.3** | 100.0 |
| Control logic | 751 | – | – | – | **–** | *70.8* | **93.1** | 94.8 |
| Glue logic | 2,169 | – | – | – | – | – | – | *82.3* |
| *TOTAL* | *77,418* | – | | *85.1* | – | *88.8* | – | *90.0* |

Table 3.7 Number of test programs, duration, and code size for the embedded FPU

| Sub-modules | #Test programs | Duration [cc] | Code size [byte] |
|---|---|---|---|
| Add/Mul/Conversions | 3 | 2,684 | 3,360 |
| Divider | 3 | 2.848 | 21,985 |
| Square root | 1 | 648 | 508 |
| Exceptions | 1 | 5,500 | 4,420 |
| Status register | 2 | 2,352 | 10,776 |
| Decode | 1 | 460 | 216 |
| Control logic | 1 | 920 | 2,897 |
| *TOTAL* | *12* | *15,412* | *44,162* |

### 3.4.3 Cumulative Results

In the frame of a collaboration with STMicroelectronics, during the recent years the proposed development flow has been applied to several industrial devices for safety-critical automotive applications. The fault coverage level of the resulting suite of SBST programs concurs to the overall functional safety regulated by the ISO 26262 standard [89].

The cumulative results gathered in these years are reported in Table 3.8. The e200 processor family is a set of CPU cores that implement low-cost versions of the Power Architecture™ Book E architecture. A brief description of the processors is given in the following:

**e200z0h** is a single-issue 32-bit CPU embedded in a single-core 90nm SoC for automotive targeting chassis market segment, specifically the Electrical Hydraulic Power Steering (EHPS) and the lower end of Electrical Power Steering (EPS), and the airbag application segment.

**e200z448** is the case study processor of Section 3.4.1. It is embedded into a 90-nm single-core SoC for safety-critical automotive applications.

**e200z215** is a single-issue 32-bit CPU operating up to 80 MHz. It is embedded into a 40-nm single-core SoC that targets automotive vehicle body and gateway applications, such as standalone gateway, simple body control module, and satellite body application like door or lighting module.

**e200z420** is a dual-issue five-stage pipeline 32-bit CPU operating up to 180 MHz. It is embedded into a 40-nm multi-core SoC (including two symmetrical z420 CPUs and a third processor running at 100 MHz) that targets automotive

Table 3.8 Cumulative experimental results on industrial processors

| Processor | #Faults | SA FC% | Duration [cc] | Code size [kB] |
|---|---|---|---|---|
| e200z0h | 198,622 | 82.55 | 45,091 | 76 |
| e200z448 | 756,789 | 87.23 | 116,102 | 119 |
| e200z215 | 282,370 | 80.40 | 156,363 | 86 |
| e200z420 | 422,419 | 82.64 | 205,003 | 113 |
| e200z425 | 792,647 | 81.04 | 222,855 | 139 |

vehicle body and gateway applications, such as central body controller, smart junction box, and mid-end and high-end gateway.

**e200z425** is a dual-issue five-stage pipeline 32-bit CPU operating up to 180 MHz. It is embedded in a 40-nm multi-core SoC that targets any safety-critical application requiring a very high level of safety integrity, such as automotive powertrain controllers, steering, breaking, and others.

In order to comply with the overall functional safety, the required fault coverage level on the CPU module of a given device was equal to 80%. As shown in Table 3.8, this level was reached in all the considered case studies. However, the final fault coverage reached on the processors is not very high and in same cases is quite near to 80%. The reason for this is partially due to the maturity level of the test suite, that is still in development for some processors. Moreover, stringent on-line requirements were limiting the effectiveness of some test procedures, as well as functionally untestable faults which are hard to identify and quantify.

Finally, cumulative results have been gathered on a set of academic processors and reported in Table 3.8. The RT-level descriptions of such processors have been synthesized using Synopsys Design Compiler targeting different open-source technology libraries. No Design-for-Testability features have been included in the final gate-level netlists, thus the fraction of functionally untestable faults in these processors is significantly lower than in industrial ones, whose netlist are post-layout. Also, the reason why the final fault coverage on academic processors is considerably higher is due the less complexity and size of the fault lists. Finally, on-line constraints have not been considered in the test program generation for these processors.

Table 3.9 Cumulative experimental results on academic processors

| Processor | #Faults | SA FC% | Duration [cc] | Code size [kB] |
|---|---|---|---|---|
| miniMIPS | 110,024 | 95.02 | 17,162 | 45 |
| Cortex-M0 | 75,936 | 90.47 | 205,765 | 153 |
| OpenRISC 1200 | 112,144 | 85.60 | 32,516 | 54 |
| OpenMSP 430 | 29,424 | 94.87 | 118,450 | 48 |

## 3.5 Chapter Summary

This chapter described an effective development flow for the SBST generation of a library of test programs to be integrated in the mission operating system. The automotive domain was used as reference in the discussion.

In details, the chapter discussed about:

1. identification of on-line constraints and implemented solutions;
2. resources distribution and generation order for a most efficient and fast test program generation along the various sub-modules of the device under test;
3. robust execution and management of SBST programs.

As case of studies, the chapter presented results related to the development flow for an industrial 32-bit processor core and a floating-point unit (FPU) included in automotive SoCs manufactured by STMicroelectronics.

The fault coverage obtained on the case study processor is more than 87% over around 750k stuck-at faults, while 90% is the FPU coverage. Cumulative results on several academic and industrial processors demonstrated that the development flow can be systematically applied to reach the target fault coverage level with reduced generation time.

# Chapter 4

# Summary of Part I

This part of the thesis presented SBST techniques for microprocessor based systems. Within this work, a set of systematic techniques for hard-to-test modules in modern microprocessors has been presented: decode unit, register forwarding and pipeline interlocking unit, special modules that implement dual-issue execution, and floating-point unit.

Although this subject has been studied for years, its practical adoption by the industry is still not mature. An effective development flow has been presented and actually implemented for a family of industrial processors manufactured by STMicroelecronics. Previous approaches found in the literature mainly target SBST generation for academic processors, without stringent on-line requirements due to the integration of SBST into mission operating systems. Such requirements are not only related to time and memory budgets, which challenge the possibility by the test engineer to reach high levels of fault coverage, but also concern the robustness of the testing environment, meaning that SBST programs must not corrupt resources used by the mission application.

Referring to the industrial domain, testing of microprocessor cores is traditionally afforded using automatic techniques that are highly supported by commercial EDA tools. This is the case of scan testing, which relies on ATPG and retargeting techniques for embedding patterns into built-in self-test modules. Conversely, SBST generation is mainly implemented with manual effort, by recurring to deterministic algorithms or feedback-based techniques, which in most cases are highly

time-consuming tasks. The usage of commercial EDA tools by the test engineer is restricted to fault simulations, used for the fault grading or withing the test program refinement loops, and (mainly combinational) ATPG for sub-modules that can be easily controlled by means of software routines. This thesis has tried to go in the direction of collecting some state-of-the art SBST techniques and new proposed ones, with the purpose of creating a repeatable generation flow for complex designs. Nevertheless, the proposed flow is based on manual effort, but is trying to maximize the generation efficiency, compared to non-organized alternative approaches.

# Part II

# Test and Diagnosis of Reconfigurable Scan Networks

# Chapter 5

# Background

Due to the complexity of new electronic devices, several features are embedded in such systems beside the core functional logic. Examples of such features are Built-In Self-Test (BIST) included for test and diagnostic sake, interfaces to core testing (e.g., based on the IEEE Std 1500), analog components (e.g., temperature sensors) accessed during the chip calibration, or debug related components (e.g., trace buffers). These features are hereinafter called *instruments*. Creating a mechanism to access instruments has led to many different legacy solutions, facing the complex task of integrating all of them in the system, especially when they come from different designers. In order to mitigate these issues, new standards have been created.

The traditional boundary scan chains are not efficient solutions for connecting the considerable number of instruments present in next generation designs for different reasons. Some of the instruments may need to be accessed only in particular situations, e.g., for security-related issues. This chapter introduces the basic concepts of Reconfigurable Scan Networks (RSN in Fig. 5.1), which extend the concept of reconfigurable scan chains and are extensively used in the IEEE Std 1687.

The rest of this chapter is structured as follows: Section 5.1 presents the main structures characterizing RSNs, in particular introduced by IEEE Std 1149.1-2013 (cf. Section 5.1.1) and IEEE Std 1687 (cf. Section 5.1.2). Section 5.2 discusses about the related work. Finally, Section 5.3 introduces a suite of benchmarks used in the experimental results of this thesis.

Parts of this chapter have been derived from published works [90, 13, 14].

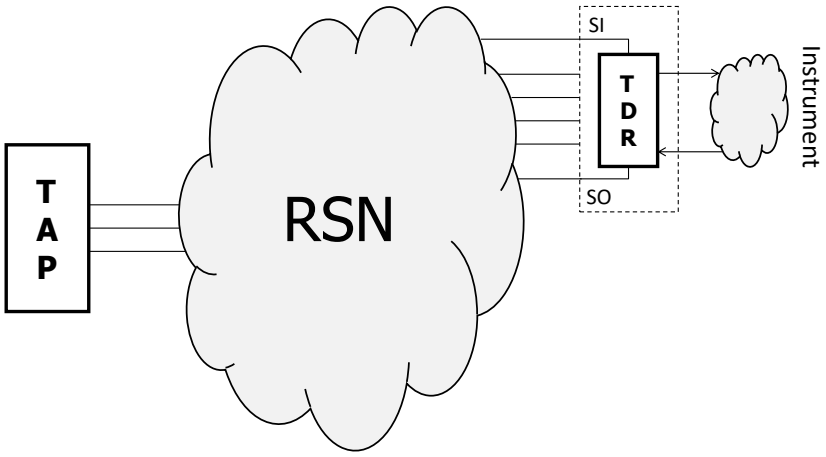



Fig. 5.1 Concept of Reconfigurable Scan Network (RSN)

## 5.1 Network Constructs

IEEE Std 1149.1-2013 and IEEE Std 1687 have introduced the concept of *reconfigurable scan chains*. This kind of chains are segmented in several parts, hereinafter referred to as *segments*, which are interleaved with special elements, hereinafter referred to as *reconfigurable modules*. Each segment can include one or more instruments. The interface with an instrument is the *test data register* (TDR), which can include capture logic (in case of reading capability) and update logic (in the case when writing is allowed). According to the configuration of reconfigurable modules, certain segments are connected together in the so called *active path*, i.e., the path connected between the scan input and scan output pins of the reconfigurable scan chain. Since the complexity of these reconfigurable scan chains can be high (i.e., many possible active paths may exist), the standards refer to them as *networks*.

### 5.1.1 IEEE Std 1149.1-2013

In the IEEE Std 1149.1, TDRs are composed of a number of bits, each bit containing a capture/scan cell (C) and an optional update cell (U). According to the standard, a set of TDRs can be defined as part of the boundary-scan architecture, each one with a register code specified at design time, and selected by acting on the *test access port* (TAP) controller. In the earlier versions of the standard, each TDR must have a fixed length, chosen by the designer. In the latest revision of the standard (i.e., IEEE Std

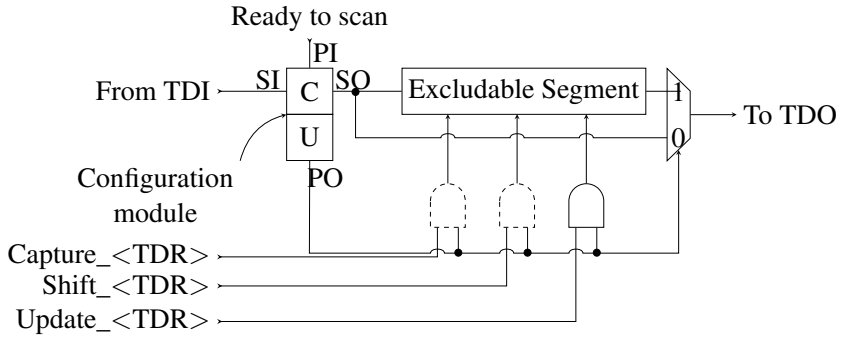



Fig. 5.2 Excludable TDR segment described in IEEE Std 1149.1-2013.

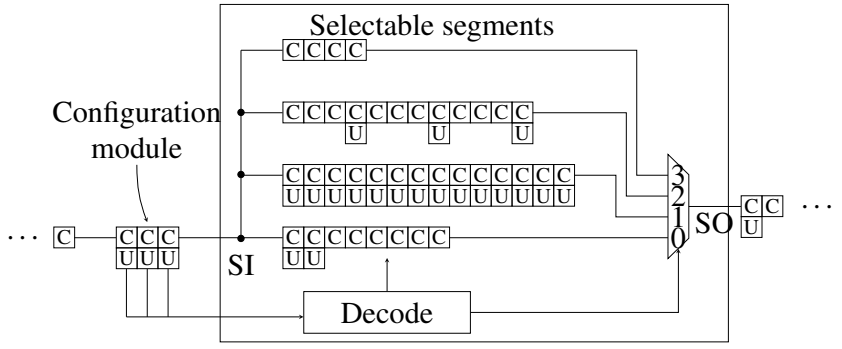
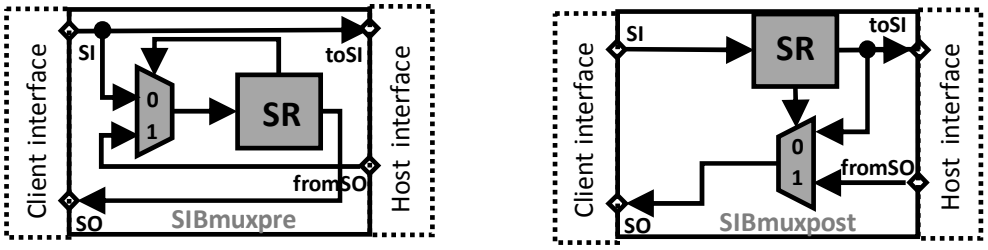



Fig. 5.3 Selectable TDR segments described in IEEE Std 1149.1-2013

1149.1-2013 [91]), each TDR can be constructed as a chain of multiple segments, some of which are always scanned while others, called *excludable segments* and *selectable segments*, are scanned only in particular situations.

The excludable segment of a TDR is controlled by a configuration module composed of one bit, which eventually disables the update/capture capabilities of the segment, and is followed by a switching element controlled by the configuration module (see Fig. 5.2). The configuration module is also a boundary scan cell, composed of a C cell and an U cell. When a logical 0 is scanned into the configuration bit, the segment will be excluded from the *active path* of the network.

The selectable segments of a TDR are segments, even of different lengths, which are connected to a selection circuit (e.g., a multiplexer). According to the value of the configuration module (this time composed of one or more bits), only one segment at a time is selected as the scanout for the set (see Fig. 5.3).




(a) pre-MUX SIB (b) post-MUX SIB

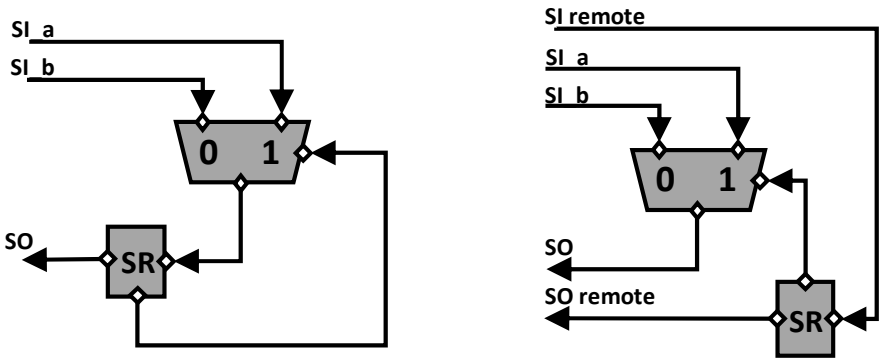
Fig. 5.4 Segment Insertion Bit (SIB) described in IEEE Std 1687

### 5.1.2 IEEE Std 1687

The new IEEE Std 1687 [92] tackles the same problem of accessing instruments by means of RSNs, also referred to as *IJTAG* (Internal JTAG) networks. Actually, the standard offers more general access mechanisms with respect to the IEEE Std 1149.1, and the reconfigurable scan networks are only a subset of the possibilities allowed. For example, the standard does not force the usage of the TAP. The structure of each network including configurable blocks is described using a standard format called *Instrument-Connectivity Language* (ICL) [92].

One of the most important structures the stardard defines is called *Segment Insertion Bit* (SIB), which is similar in concept to an excludable segment. Selectable segments are instead implemented by means of scan multiplexer (ScanMux) modules. Each segment controlled by a SIB or a ScanMux can be a complex network itself.

A SIB includes a multiplexer preceding the flip-flop storing the control bit, as shown in Fig. 5.4a, or the multiplexer can follow the control bit (Fig. 5.4b). Control bit is implemented as a Scan Register (SR) of a size of one bit. With the host scan interface, SIB is connected to the nested scan chain segment. With the client scan interface of the SIB, active scan chain has access to the nested scan chain segment located behind the host interface of the SIB. In Fig. 5.4, the right-hand side ports (toSI and fromSO) belong to host type and the left-hand side ports (SI and SO) belong to the client scan interfaces type. For the sake of simplicity, the CE, SE, UE, SEL, RST, and TCK ports of the scan interface are omitted in the figure. When a SIB is said to be *asserted*, the nested scan chain segment is included in the active path; otherwise, it is said to be *de-asserted*.




(a) SCB in-line (b) SCB remote

Fig. 5.5 ScanMux Control Bit (SCB) described in IEEE Std 1687

The second type of network configuration constructs is for selecting a split, mutually-exclusive, selectable scan chain. For this task the standard defines an inline ScanMux control bit (SCB) (Fig. 5.5a) and a ScanMux with remote control bit (Fig. 5.5b). In Fig. 5.5, SI_a, SI_b and SI_remote indicate an end-points of the scan chains that belong to the separate scan interfaces, where SI_a and SI_b share the SO port. The inline and remote SCBs have only client type scan interfaces. Thus, the scan chains SI_a and SI_b (Fig. 5.5) are selected by the dedicated signals (not shown in Fig. 5.5) composed with and operation between SEL signal from the respective host interface and the SCB data signal. Same SCB data signal is selecting the multiplexer inputs.

In order to bring a reconfigurable scan network into a certain network configuration, vectors have to be shifted through the scan input port. Then, an update operation moves the vector from the shift flip-flops (C cells) to the update latches (U cells) of the configuration module. This operation changes the active path of the network. Since a RSN can have a hierarchical structure, the operation of making an instrument, placed deep into the network, part of the active path may require multiple configuration phases. The standard defines the language to describe such instrument-specific and network-independent procedures together with necessary data (including test patterns), called *Procedure Description Language* (PDL) [92].

Finally, the access to instruments may include combinational logic (e.g., included for security purposes).

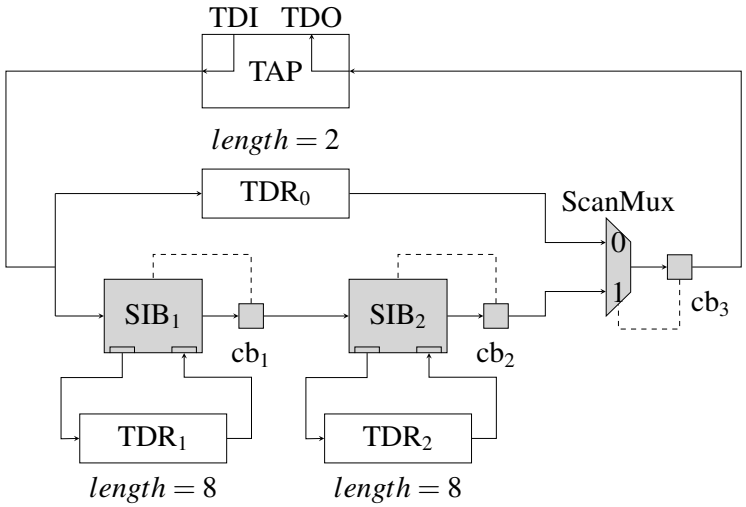



Fig. 5.6 Example of IEEE Std 1687 reconfigurable scan network.

### 5.1.3 Example Network

In order to clarify the scenario and to explain the methodologies proposed in the next sections, a simple IEEE Std 1687 scan network is shown in Fig. 5.6. The example network is accessed through an IEEE 1149 TAP interface and is composed of two selectable segments: the first one with a single TDR, and the other one with two TDRs, each one controlled by a SIB. In Fig. 5.6, the bit length of each TDR is also reported. The SIB modules and the multiplexer (ScanMux) are associated each one with a control bit ($cb_1$, $cb_2$, and $cb_3$) and are highlighted in grey. Depending on the configuration (i.e., the value of the control bits of SIBs and ScanMux), the network presents one of five possible active paths, each one including different subsets of TDRs, as listed in Table 5.1. In this table, 'A' means the SIB is in the asserted position, 'D' means de-asserted, 0 and 1 correspond to the two possible positions of the ScanMux, and 'X' appears when a module belongs to an inaccessible segment (i.e., don't care value). During the system reset, a known configuration is selected. The status of the reconfigurable modules upon reset determines the network *reset configuration*. In the example network, the reset configuration is assumed to be 0,D,D (using the same module ordering of Table 5.1).

Table 5.1 List of possible configurations for the network in Fig. 5.6.

| ScanMux | SIB1 | SIB2 | Scan length | Active path |
|---|---|---|---|---|
| 0 | X | X | 3 | TDI→$TDR_0$→$cb_3$→TDO |
| 1 | D | D | 3 | TDI→$cb_1$→$cb_2$→$cb_3$→TDO |
| 1 | D | A | 11 | TDI→$cb_1$→$TDR_2$→$cb_2$→$cb_3$→TDO |
| 1 | A | D | 11 | TDI→$TDR_1$→$cb_1$→$cb_2$→$cb_3$→TDO |
| 1 | A | A | 19 | TDI→$TDR_1$→$cb_1$→ $TDR_2$→$cb_2$→$cb_3$→TDO |

## 5.2 Related Works

RSNs have been a hot research topic for years. Due to the recent adoption of the IEEE Std 1687 by commercial tools, several issues have arisen regarding design, validation, and test of such structures, as well as their usage in the field.

Given a generic circuit with several instruments, different access networks can be created, e.g., by using IEEE Std 1687 SIB modules. The authors of [93] have analyzed different possible scenarios and evaluated the overall access time (i.e., the time required to access all instruments in the circuit).

The design automation of IEEE Std 1687 reconfigurable scan networks has been targeted by different works. In [94], a method has been proposed, which is able to construct networks composed of multiple SIBs that optimize parameters such as area overhead, total access time, or average access time. Additional scenarios have been analyzed in [95], such as different positions of the configuration module with respect to the related ScanMuxes; moreover, a formal analysis of the modeled networks has been presented in the same paper for verification sake. Test scheduling has been analyzed in [96, 97].

A formal analysis of generic RSNs has been also presented by the authors of [98, 99]; RSNs have been modeled as satisfiability (SAT) problems, upon which several structural and functional properties have been verified. The works in [98] and [99] offer a deep analysis about the related modeling, verification, and pattern generation problems. However, the authors of [98] and [99] do not face the issue of minimizing the test time, which is one of the goals of this article.

Pattern retargeting is a well-known problem in the area of RSNs. This problem consists of the operations that are needed to transport the required data bits to/from

the instruments' registers [100]. IEEE Std 1687 provides a way to define instrument-specific operations. Each operation targets a specific segment of the network, thus the specific issue arises of configuring the network such that the time to access that segment is minimized. This problem has been tackled by different works, e.g., [101, 100]. The benefit of automatic retargeting has been discussed in [102].

The test of RSNs has been targeted in different works. Dependability issues of RSNs including verification, test, and security have been the objective of [103]. The authors of [104] have tested structural faults and investigated the test quality of different test strategies. The combination of the evaluated test strategies achieves high fault coverage even in synthetic difficult to test circuits. More than evaluating the test time efficiency, which is the main objective of the work in Chapter 6, the work in [104] aims to maximize the structural fault coverage. In the work presented in Chapter 6, the knowledge of the system is limited to the functional description of the network, thus controllability of the attached instruments is neglected, meaning that faults affecting such a logic are not considered.

In-field reuse of IJTAG for system-health monitoring has been proposed in [115, 116] and further elaborated in [117, 118, 119, 120, 121] including a prescriptive regular architectural extension proposal [117] and introducing a new concept of self-reconfiguring IJTAG networks [121]. Recent works describing on-chip monitor wrapping case-study [122], and focusing on implementation challenges of on-chip IJTAG retargeting engines [123], as well as on-the-fly dynamic retargeting framework [124]. Reliability and fault tolerance of IJTAG during online operation has been considered in [125].

Finally, IJTAG has also been proposed as a complementary solution for 3D-IC test [126], which opens up another dimension in research and application in the domain of IJTAG.

## 5.3 IEEE 1687 Benchmark Networks

The lack of specific IEEE Std 1687 open examples, forced several research groups to derive own IEEE Std 1687 networks from the ITC'02 set [127] – all yielding different results. Different authors have made different assumptions, which makes it difficult now to objectively compare their results.

Different ways have been used by authors to generate IEEE Std 1687 networks from ITC'02 benchmarks. First is the way how scan-chains are associated to instruments. Each instrument may either receive a dedicated scan register or all instruments at a core could be connected together. Second, the inputs, outputs and bi-directionals in the ITC'02 benchmarks can be handled differently. Finally, there are different ways to handle the hierarchy resulting in different network depth. This makes comparison across different methods difficult and represents an additional argument towards a new holistic benchmark set.

In order to stop uncontrolled multiplication of numerous similar but non-identical experimental IJTAG networks, a collaborative work within the frame of the EU-funded FP7 research project BASTION put together a holistic benchmark suite. Source files (including ICL and VHDL) as well as respective documentation, including textual and graphical descriptions allowing to characterize each network are publicly available through the BASTION web-site [128]. Each benchmark corresponds to the IJTAG circuitry that connects the device interface (e.g., the IEEE 1149.1 TAP, or the IEEE 1500 interface) to the instrument interfaces. The benchmarks do not include any information about the connected instruments, apart from the size of the related TDR.

Details about the benchmarks, referred to as ITC'16 in this thesis, are given in [90]. The paper also reports some results gathered by applying a few algorithms developed to face some problems related to IJTAG networks, such as retargeting, test and diagnosis (these last ones are discussed in the next chapters of this thesis).

# Chapter 6

# Testing

This chapter discusses about the test of possible faults affecting reconfigurable modules in RSNs.

Testing a standard (non-reconfigurable) scan chain for permanent faults has been a widely studied subject for years. Several techniques exists, e.g., shifting a suitable sequence of 0s and 1s through the scan chain, such as the sequence "00110011" that applies all possible transitions in two cycles [129]. In order to cover scan cells internal defects (see [130]), the previous sequence is not enough, but has to be enhanced with additional tests, e.g, for stuck-open faults [131, 132], or for bridging faults [133]. Diagnosis for intermittent faults in scan-chains has been discussed in [134].

RSNs are however more complicated to test. In addition to flip-flops composing the TDRs, which have to be tested to check whether they can correctly shift values when included in the active path, the reconfigurable modules (e.g., the IEEE Std 1687 SIBs and ScanMuxes) also need to be tested in order to check whether they are able to move the network into all its possible configurations.

The main motivation of this work is not to cover faults affecting TDR flip-flops. These modules have been widely studied for years and state-of-the-art techniques can be used for this purpose. Moreover, the update and capture logic is not taken into consideration in this work, since its controllability and observability depend on the instruments attached to the TDRs.

The main motivation of this work is checking that the capability of a network to change its configuration is not corrupted by a fault. This means that reconfigurable modules, such as SIBs and ScanMuxes, have to be fully tested. Moreover, the configuration bits associated to these modules have to be also tested. In details, the main objective is to present the problem of testing reconfigurable modules and to offer alternative ways to solve it. Different techniques in terms of memory requirements and computational time are presented in the following sections. This chapter extends the basic idea already presented in [13] and [14] with a detailed description of the proposed algorithms and more extensively supported by experimental results.

The rest of this chapter is structured as follows: Section 6.1 presents the basic notions about network testability, the functional fault model introduced in this work, and test vectors. Section 6.2 illustrates the high-level representation of RSNs by means of graphs, which are used by the proposed algorithm. Section 6.3 presents different test algorithms, based on graph exploration, including an optimal approach in terms on test time. Some experimental results on benchmark RSNs are reported in Section 6.4. Finally, Section 6.5 draws some conclusions.

## 6.1 Terminology and Fault Model

In this section, the basic terminology introduced in this chapter is presented. Moreover, in order to generalize as much as possible the problem of testing the reconfigurable modules in a network, a functional fault model is introduced and specified for each type of reconfigurable modules.

### 6.1.1 Configurations, Vectors, and Test Time

A generic reconfigurable module, hereinafter indicated with $M_i$, is able to control parts of the network defined as *segments*, according to the values stored in its configuration bits. Moving backward from the $k$-th input pin of $M_i$ (corresponding to an input to the multiplexer in $M_i$), the segment $s_k$ ends in the fork that connect $s_k$ to the other segments connected to $M_i$. In the example of Fig. 5.6, ScanMux controls two segments: one including $TDR_0$, and the other including the two SIBs. Moreover, each SIB in the figure controls a segment that include a TDR and an empty segment (when de-asserted). In general, a certain $M_i$ has $k$ controllable (or

*selectable*) segments. In this thesis, each element of the network is associated to the most specific segment possible. For example, $TRD_1$ lies in the segment controlled by $SIB_1$, while $cb_1$ is in the segment controlled by the ScanMux. The length of a segment is equal to the number of bits it includes.

A generic configuration of the network (i.e., the value of all configuration bits) is referred to as $\sigma_i$. The term $\sigma_0$ indicates the reset configuration. Each $\sigma_i$ can be associated to a record, which contains an identifier and the following information:

- for each reconfigurable module $M_i$, the configuration bit values (e.g., asserted/de-asserted for SIBs, an input identifier for ScanMuxes);
- the active path length;
- the list of possible faults (each referred to as $F_i$) affecting the network, that can be detected by performing test operations while the network is configured with $\sigma_i$.

Such test operations use *test vectors* to verify whether the expected path has been inserted between the scan input and scan output pins, i.e., whether the right instrument can be accessed during the normal operation. An example test vector $tv_i$ consists of the following operations:

1. a suitable sequence (as long as the active path length) is shifted in, forcing it to travel along the active path and to appear on its other end;
2. scan output pins (e.g., TDO) are monitored: the sequence previously loaded is expected to come out; based on the fact that the observed sequence matches the expected one or not, possible faults can be detected; we will see in the following that, according to the proposed fault model, the effect of a fault affecting a reconfigurable module is to change the active path; in this case, the expected output sequence will appear on scan output pins after a wrong number of clock cycles.

A network *transition* is defined as a change in the configuration, by means of one or more *configuration vectors*. A generic configuration vector $cv_i$ consists of the following operations:

1. as many shift operations as the active path length, so that the next configuration is stored in the C flip-flops of the reconfigurable modules' configuration bits, while the other bits are don't care ('X' in this thesis);

2. an update operation, so that the next configuration is applied to the network and the active path changes.

If transitioning from the configuration $\sigma_i$ to $\sigma_j$ requires a single configuration vector, then $\sigma_j$ is a *neighbor* configuration of $\sigma_i$. In this case, the *transition cost* in terms of clock cycles is equal to the active path length of $\sigma_i$ plus one (the update operation). Please note that the neighborhood relation is not reversible. For example, let us consider the network in Fig. 5.6, whose configurations are listed in Table 5.1. In this network, the configuration bits are placed in the same segment of the related reconfigurable module (i.e., right after each SIB and ScanMux), then the network can be moved from the configuration $\sigma_1 = \{1, A, A\}$ to $\sigma_2 = \{0, D, D\}$ by shifting a single vector. On the contrary, when the network is in $\sigma_2$, two vectors are needed to reach $\sigma_1$, passing through the intermediate configuration $\sigma_3 = \{1, D, D\}$.

The neighborhood $\Sigma_i$ of a certain configuration $\sigma_i$ is obtained by generating all permutations on the reconfigurable modules' configuration bits that are part of the active path (i.e., they can be changed by shifting a single vector). In the configuration $\sigma_1$ of the previous example, all configuration bits are part of the active path, thus the neighborhood of $\sigma_1$ includes all other configurations. On the contrary, $\sigma_2$ only exposes the element ScanMux, while $\text{SIB}_1$ and $\text{SIB}_2$ are not included in the active path; thus, the neighborhood of $\sigma_2$ is obtained by changing the configuration of ScanMux, i.e., it only includes $\sigma_3$.

Configuration and test vectors are used by the proposed test techniques and organized in *sessions*. A generic session, referred to as $S_i$, is composed of two phases:

1. a *configuration phase* (*Cfg*), corresponding to a network transition, in which a certain number of configuration vectors are applied, until the target configuration is reached;
2. a *test phase* (*Tst*), in which test vectors are applied.

The sequence of test vectors to be used in the test phase depends on the kind of defects to be tested. More details are given in the following subsection, where the specific fault model and test vectors for reconfigurable modules are presented.

For every session $S_i$, the related *session fault set* ($SFS_i$) is defined as the set of all faults related to reconfigurable modules excited by the session.

The term $t_i^c$ is used to denote the duration (in clock cycles) of the configuration phase $Cfg_i$ and $t_i^t$ indicates the duration of the test phase $Tst_i$. The configuration time is the time needed to apply all the configuration vectors of the session. Each vector requires a certain time to be shifted in, plus a few clock cycles (the exact number of which is implementation dependent) to update it into the U cells of the corresponding path (this time is denoted as *JTAG protocol overhead* in [97]). The active path changes after each update operation, thus each vector may have a different length. The duration of the test phase ($t_i^t$) depends on the active path length $l$ of the target configuration (i.e., after the last configuration vector). The total test duration for a network that needs $N$ sessions to cover each testable fault is thus given by:

$$T = T_c + T_t = \sum_{i=1}^{N} t_i^c + \sum_{i=1}^{N} t_i^t \tag{6.1}$$

where $T_c$ is the sum of clock cycles of each $Cnf_i$ and $T_t$ is the sum of the clock cycles of each $Tst_i$.

During test generation, a *fault list* is used, which is composed of all possible faults affecting the network, according to the proposed fault model. The fault list includes an indication for each fault, hereinafter indicated with $F_i$, about whether $F_i$ is tested, still untested, or untestable in any possible network configuration. If $F_i$ is still untested, is said to be *active*.

### 6.1.2 Fault Model for Reconfigurable Modules

Reconfigurable modules are used with the purpose of including (excluding) segments into (from) the scan path. In case of faults affecting such modules, the network configuration may become different than the expected one, or unknown. In this situation, the network becomes unusable, hence testing such modules is a real need.

In order to be independent from the implementation, a functional fault model is used. A certain fault $F_i$ affecting reconfigurable modules is modeled such that a different configuration is forced by $F_i$ rather than the expected one. $F_i$ leads to a different active path (called *faulty path*) than the expected one, and the two are likely to have a different length. For example, in Fig. 5.6 the multiplexer (ScanMux) may be affected by a permanent fault whose effect is that the segment connected to the input 0 is always selected, no matter the value in the configuration bit. The

same may arise for the SIBs, which may be stuck at their asserted (or de-asserted) configuration.

Stuck-at faults in the shift flip-flops (C cells) of the configuration modules are considered as detected by implication by testing such functional faults, which also cover the faults affecting the update logic of reconfigurable modules. Moreover, such faults also cover those faults affecting the reset logic, whose effect is to keep the module stuck at its reset value. Other reset faults (i.e., those that make the reset ineffective) are not considered, but can be easily targeted if we grant the possibility to act on the asynchronous reset signal.

A proper test for functional faults of a reconfigurable module compares, for a configuration able to excite a given fault, the expected path length against the length of the faulty path. This comparison is performed by looking at the number of clock cycles required by the input sequence to appear on the scan output pin. As an example, the functional fault on the module ScanMux of Fig. 5.6, which always selects the segment connected to the input 1, can be excited by a configuration which selects the input 0; all such possible configurations are listed in Table 6.1, which also reports the length of the selected faulty paths. In the table, the first configuration is not able to detect the fault, due to the fact that the faulty path length is equal to the active path length. Thus, one of the remaining three configurations can be selected for the test.

Some situations may exist, in which all faulty paths have the same length of the active path, i.e., no configuration able to excite the fault exists. In this case, the fault is *untestable*.

In the following subsections, the basic concepts presented above are applied in a test procedure for SIB modules and ScanMuxes. Configuration and test vectors are applied in the test procedure. Since test vectors aim at checking whether the active paths are as long as expected, an *initialization vector* is included in the test phase, which forces the scan paths to a known value. A sequence of 0s can be used for this purpose. The length of the initialization sequence is equal to the longest path in the network, rather than to the active path length. The reason for this is that a fault affecting a reconfigurable module results in a faulty path, whose length is (generally) different than the length of the expected path. Thus, the length of the longest path in the network is a suitable value to be used before any test phase. In this case, each $t_i^t$ contribution in Eq. (6.1) includes the length of the longest path. An alternative

Table 6.1 Effect of the functional fault on the ScanMux of Fig. 5.6, which always selects the input 1, when selecting different active paths.

| ScanMux | SIB1 | SIB2 | Path length (faulty/active) | Faulty path |
|---|---|---|---|---|
| 0 | D | D | 3/3 | TDI→$cb_1$→$cb_2$→$cb_3$→TDO |
| 0 | D | A | 11/3 | TDI→$cb_1$→$TDR_2$→$cb_2$→$cb_3$→TDO |
| 0 | A | D | 11/3 | TDI→$TDR_1$→$cb_1$→$cb_2$→$cb_3$→TDO |
| 0 | A | A | 19/3 | TDI→$TDR_1$→$cb_1$→ $TDR_2$→$cb_2$→$cb_3$→TDO |

solution is to use the maximum length among all faulty paths and the expected path, but this value is fault-dependent. In the example of Fig. 5.6, an initialization vector composed of 19 0s can be used before each test phase.

### SIBs

Given a SIB stuck-at asserted (de-asserted) fault, the test procedure has to force the SIB to be de-asserted (asserted) and then check whether the SIB works correctly. In details, the test procedure is the following:

1. apply a certain number of configuration vectors, until the SIB is part of the active path;
2. shift in an initialization vector whose length is equal to the one of the longest path in the network;
3. shift in a test vector as long as the expected path length;
4. check whether the expected sequence appears on the output of the path;
5. reconfigure the network, so that the SIB is part of the active path and at the opposite configuration;
6. shift in an initialization vector whose length is equal to the one of the longest path in the network;
7. shift in a test vector whose length is equal to the one of the expected path length;
8. check whether the expected sequence appears on the output of the path.

As an example, the test of $SIB_1$ of the reference network in Fig. 5.6 is presented, assuming that in the reset configuration the SIBs are de-asserted and the ScanMux selects the input 0. As test vectors, a sequence of alternated 0s and 1s is used,

followed by two consecutive 1s, which are used as sequence terminator. The output pin is monitored until the sequence terminator is shifted out: this permits to calculate the active path length and to compare it against the expected one. In details:

1. Reset – active path (AP): TDI→TDR$_0$ →s$_3$→TDO
2. Configuration vector 1:
   (a) Shift in `XX1` (lenght = 3)
   (b) Update – AP: TDI→cb$_1$ →cb$_2$ →cb$_3$→TDO
3. Initialization vector 1:
   (a) Shift in `0000000000000000000` (lenght = 19)
4. Test vector 1:
   (a) Shift in `010` (lenght = 3)
   (b) Shift in `11` (lenght = 2)
5. Check
6. Configuration vector 2:
   (a) Shift in `101` (lenght = 3)
   (b) Update – AP: TDI→TDR$_1$ →cb$_1$ →cb$_2$ →cb$_3$→TDO
7. Initialization vector 2:
   (a) Shift in `0000000000000000000` (lenght = 19)
8. Test vector 2:
   (a) Shift in `01010101010` (lenght = 11)
   (b) Shift in `11` (lenght = 2)
9. Check.

In the last check (step 9), extra bits are shifted in until the last sequence terminator comes out from the output pin (at maximum, as long as the longest path plus the length of the sequence terminator, i.e., $19+2=21$ in the example). The lengths of test vectors shifted in at steps 4 and 8 are checked when the next configuration vector is shifted.

### Scan Multiplexers

The same test procedure can be extended to scan multiplexers. The basic idea is once again to first configure the network so that the ScanMux is switched to a given configuration, thus making a given path accessible. The difference with respect to the SIB is that the faulty path of a SIB is always longer (or shorter) than the active path. On the contrary, that is not the case of faulty paths of ScanMuxes (see Table 6.1). In

that case, the length of the faulty path may even vary depending on the configuration of other modules in the active path. Moreover, the faulty paths can be more than one (e.g., in a 4-to-1 multiplexer, each configuration has 3 faulty paths). In order for the fault to be testable, the length of each faulty path has to be different than the active path. In details, the test procedure for a testable scan multiplexer fault is the following:

1. apply a certain number of configuration vectors, until:
   (a) the multiplexer is part of the active path and set at a certain configuration, and
   (b) the other modules are configured such that the faulty paths have different length than the active path;
2. shift in an initialization vector as long as the longest path in the network;
3. shift in a test vector as long as the expected path length;
4. check whether the expected sequence appears on the output of the path;
5. repeat the previous steps for all the multiplexer's configurations.

## 6.2 Network Representation

RSNs can be represented in different formats, such as the *Instrument Connectivity Language* (ICL) for IEEE Std 1687 networks. Structural representations are hard to handle and extracting the main properties of a network requires to pass through internal representations. In order to abstract the main functions of a reconfigurable scan network to be tested, two formal graphs representations are proposed.

These representations are traversed by the proposed test algorithms, as discussed in the following section, in order to generate a sequence of configuration and test patterns for the target network.

The first graph, named *Topology Graph*, is based on the network topology and consists of all possible scan paths of the network. This graph is a topological view of the network.

The second graph, named *Configuration Graph*, is based on the list of possible network configurations. A path in this graph represents a sequence of configurations in which the network can be sequentially placed.

### 6.2.1 Topology Graph

The topology graph is a simplified representation of the scan network providing a topological view. The elements of the scan network (TDRs, SIBs, ScanMuxes, configuration bits) are associated with vertices, while the connections between elements are represented by edges. The set of vertices also includes those associated to the input and output pins of the scan network, e.g., TDI and TDO.

In case of a single pair of input/output pins, the graph has a single source vertex (e.g., TDI) and a single sink vertex (e.g., TDO). In general, it is a directed acyclic graph (DAG), whose topological ordering has the following properties (according to the graph theory):

- the network input pins are in the first level (their corresponding vertices are called *source* in the graph theory notation);
- the reconfigurable modules and registers are in the intermediate levels (their corresponding vertices are called *internal*);
- the network output pins are in the last level (their corresponding vertices are called *sink*).

Each vertex of the graph is annotated with the hierarchical *depth* of the corresponding element in the scan network. As an example, let us consider the scan network of Fig. 5.6, whose topology graph is shown in Fig. 6.1. In such a graph, the reconfigurable modules' vertices are highlighted in grey. For each vertex, the depth $k$ is also reported in Fig. 6.1 using the notation $d = k$. In details, the elements ScanMux and $cb_3$ are placed at the top-level ($depth = 1$), i.e., they are always part of the active path. The elements $TDR_0$, $SIB_1$, $cb_1$, $SIB_2$, and $cb_2$ are placed in the segments controlled by ScanMux, thus their vertices are annotated with $depth = 2$. Finally, the vertices $TDR_1$ and $TDR_2$ are annotated with $depth = 3$, since they are placed in the segments controlled by $SIB_1$ and $SIB_2$, respectively. In general, the depth of a module placed in a certain segment is the depth of the configurable module controlling that segment incremented by one.

Each vertex has as many outgoing edges as the number of alternative paths it produces in all its configurations. Configuration values are used to annotate edges outgoing from a given vertex. A simple case is the SIB, which can be either asserted or de-asserted. This means that each SIB vertex has two outgoing edges, labeled *asserted* and *de-asserted*, respectively. In the scan network of Fig. 5.6, the vertex for

$SIB_1$ has two outgoing edges: the first edge is connected to $TDR_1$ and corresponds to the case in which $SIB_1$ is asserted, while the other one is connected to $cb_1$.

Similarly, a vertex associated to a ScanMux has as many outgoing edges as the number of inputs to the multiplexer. Each edge is labeled with the values of the multiplexer's configuration bits that select the corresponding segment. For example, the vertex associated to the ScanMux in Fig. 5.3 has four outgoing edges, each one connected to a TDR. It is important to note that the vertex associated to a ScanMux precedes the vertices of each element on its segments in the topological ordering of the graph. When building the topology graph, the vertex of a multiplexer is associated to the fork of its segments, rather than to the element itself. Moreover, the last elements on the segments are directly connected to the vertex associated to the element that succeeds the ScanMux in the network. In the reference network, starting from TDI, the scan path is forked into two segments, one including $TDR_0$, and the other with $SIB_1$ and $SIB_2$ (and their configuration bits $cb_1$ and $cb_2$). The two segments are connected to the element ScanMux. In the corresponding topology graph (see Fig. 6.1), the vertex associated to TDI is directly connected to ScanMux, which has two outgoing edges: one connected to $TDR_0$ and one to $SIB_1$. The vertex $TDR_0$ is directly connected to $cb_3$, as well as the vertex $cb_2$. Finally, a vertex associated to a TDR has a single outgoing edge. In the topology graph of Fig. 6.1, edges that are activated upon reset are highlighted.

Every possible path in the scan network is represented by a path in the topology graph from source to sink vertices. When the network is in a certain configuration, each reconfigurable module selects a given segment. Similarly, one *active edge* comes out from the related vertex in the topology graph. Moreover, each source vertex is connected to a sink vertex by means of an active path.

### 6.2.2 Configuration graph

The topology graph offers a view of the interconnections between scan modules of the network. Such a representation, however, does not include information about the time (in terms of scan clock cycles) required to move the network from one configuration to another. Thus, an alternative representation is needed.

The list of configurations and the neighborhood relation are used to build a directed graph $G = (V, E)$. Each vertex $V_i$ corresponds to a network configuration

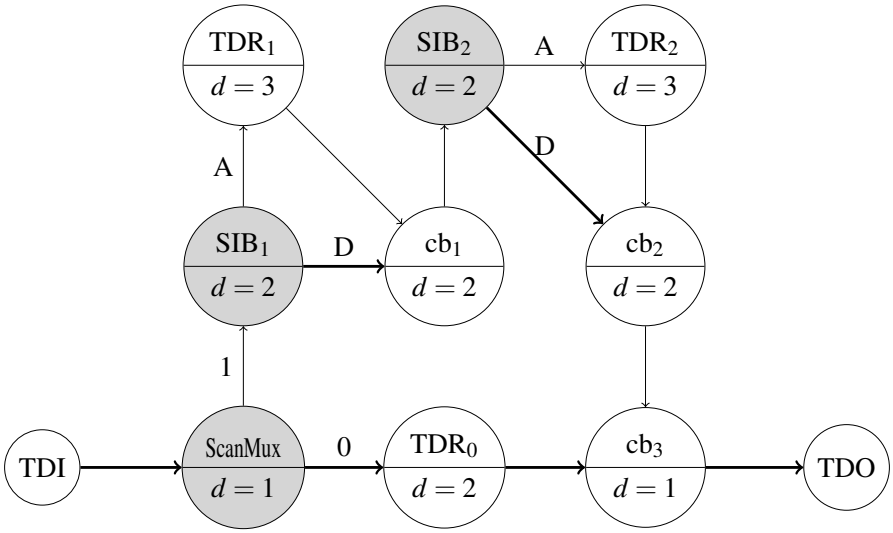



Fig. 6.1 Topology graph of the example network in Fig. 5.6.

$\sigma_i$. The reset state $V_0$ is used to refer to $\sigma_0$. Each edge $(V_i, V_j)$ represents a transition from $\sigma_i$ to $\sigma_j$ with $\sigma_j$ neighbor of $\sigma_i$ and it is labeled with its transition cost, equal to the active path length of $\sigma_i$ (if need be incremented by one, i.e., the extra clock cycle for the update operation). The active path of a vertex is obtained by applying its configuration to the topology graph.

The configuration graph can be built by applying the following procedure. As many vertices are created as the number of possible network configurations. In details, for each configuration $\sigma_i$, the following steps are performed:

1. a vertex $V_i$ is created, if not existing;
2. the neighborhood $\Sigma_i$ is identified;
3. for each configuration $\sigma_j \in \Sigma_i$, a vertex $V_j$ is created, if not existing, and an edge $(V_i, V\,j)$ is created and labeled with the active path length of $\sigma_i$.

The process can be implemented as a recursive procedure that starts from the reset configuration and returns when all neighbor configurations are extracted from the neighborhood set.

As an example, the procedure has been applied on the network represented in Fig. 5.6. The adjacency matrix of the resulting graph is shown in Table 6.2. Each row of the matrix reports the transition cost of the outgoing edges from a vertex to other vertices, or '–' when the two vertices are not connected by an edge. It can be noticed that the first four configurations have only one outgoing edge each.

Table 6.2 Adjacency matrix of the configuration graph built on network in Fig. 5.6.

| | 0,D,D | 0,D,A | 0,A,D | 0,A,A | 1,D,D | 1,D,A | 1,A,D | 1,A,A |
|---|---|---|---|---|---|---|---|---|
| 0,D,D | – | – | – | – | 4 | – | – | – |
| 0,D,A | – | – | – | – | – | 4 | – | – |
| 0,A,D | – | – | – | – | – | – | 4 | – |
| 0,A,A | – | – | – | – | – | – | – | 4 |
| 1,D,D | 4 | 4 | 4 | 4 | – | 4 | 4 | 4 |
| 1,D,A | 12 | 12 | 12 | 12 | 12 | – | 12 | 12 |
| 1,A,D | 12 | 12 | 12 | 12 | 12 | 12 | – | 12 |
| 1,A,A | 20 | 20 | 20 | 20 | 20 | 20 | 20 | – |

In such configurations, in fact, the ScanMux is configured to the value 0, thus the other configuration bits are not part of the active path. On the contrary, all the other configurations can reach all the other vertices of the graph. Moreover, all outgoing edges from a certain vertex are labeled with the same transition cost, equal to the active path of the vertex plus one.

## 6.3 Proposed Test Strategies

The complete test of a reconfigurable network must pass through a certain number of configurations, each one able to include in the active path a subset of the registers and the reconfigurable modules. Once each target configuration is reached, the active path of the network is tested and the response is observed by monitoring the scan output values. In Section 6.1, the concepts of configuration and test vectors and sessions have been introduced.

After the system reset, the network is set to its initial configuration (which is known). The overall test procedure requires a certain amount of sessions. After each session, the network target configuration is changed and the target configuration of the previous session becomes the starting configuration.

During the test phase, the active path includes a certain number of reconfigurable modules to be tested. The test vectors to be applied in this phase depend on the specific defects under analysis. Since the main focus of this work is not to improve state-of-the-art scan cell testing techniques but to support minimum-length test of reconfigurable modules, each $Tst_i$ simply consists of an initialization vector composed of as many 0s as the longest path length, followed by a test vector

composed of an alternate sequence of 0s and 1s. Clearly, more complex sets of vectors can be used in this phase.

Since the amount of possible configurations of a network grows exponentially with the number of reconfigurable modules, the problem of identifying a sequence of sessions which guarantees the full network test coverage while minimizing the overall test time is not trivial. This work achieves the intended goal with a tractability limitation of the approach on large circuits, due to the size of the search space. In the next sections, an approach implementing a minimum cost search on the configuration graph is presented. An enhancement of the proposed approach based on a space pruning heuristic is then shown. Such a variant is able, in some cases, to reduce dramatically the search space, thus speeding up the search algorithm.

The proposed optimal approach is also useful for evaluating the effectiveness, in terms of total test time, of alternative solutions to the same problem, e.g., based on heuristics or optimization techniques. Later on in this section, a pair of sub-optimal approaches based on the topology graph traversal are presented. The main advantage of these solutions is that they are easy to implement and successfully applicable to very large circuits.

### 6.3.1 Optimal Approach

The approach proposed in this section formulates the problem as a graph search at the minimum cost. The cost to be minimized is the total test time, as expressed in Eq. (6.1). The configuration graph, enriched with test information, is traversed using a modified version of the $A^*$ algorithm [135]. In the following, the classical $A^*$ formulation is presented. Later on, the proposed modifications to the algorithm are shown in details.

$A^*$ is an informed search algorithm, which operates by searching among all possible paths to the solutions (goals) for the one that incurs the smallest cost. In a labeled graph, vertices are the problem states, while each edge represents the transition cost to move from a state to another. The goal is typically identified by a certain vertex. Starting from a specific vertex of the graph, $A^*$ constructs a tree of paths rooted in that vertex, expanding paths one step at a time, until one of its paths ends at the predetermined goal vertex.

At each iteration of its main loop, $A^*$ determines which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to reach the goal vertex. Specifically, $A^*$ selects the path that minimizes the following function:

$$f(n) = g(n) + h(n) \tag{6.2}$$

where $n$ is the last vertex on the path, $g(n)$ is the cost of the path from the source vertex to $n$, and $\mathrm{h}(n)$ is a heuristic that estimates the cost of the cheapest path from $n$ to the goal. The heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the goal vertex.

The problem at hand is different from the classic $A^*$ formulation in which there is not a predetermined goal state. The goal is to reach the full test coverage of the network, thus it cannot be associated to a certain state of the configuration graph. Instead, whether or not a given state is the goal depends on the path followed to reach it. In other words, the algorithm is looking for a goal path instead of a goal state.

The algorithm traverses the configuration graph. In such a graph, each vertex is associated to a network configuration. By moving to the next vertex, a configuration pattern is applied. The optimal test procedure has to determine whether or not a test phase is necessary before moving to the next configuration (in the latter case, the configuration is intermediate). Since $A^*$ is a graph traversal algorithm, such information has to be embedded into the graph representation. The proposed solution is to add new vertices to the configuration graph. In details, for each vertex $V_i$, a new vertex $T_i$ is created. The vertex $T_i$ can be reached only from $V_i$ through an edge whose transition cost is the number of clock cycles needed for the test phase. After the test phase, the network configuration is not changed, thus it is the same of $V_i$. In order to keep the representation able to reach other configurations after the test phase, $T_i$ is connected to all $V_i$'s neighbor vertices through edges with the same transition cost.

The graph is traversed starting from the network's reset configuration state $V_0$. When a $T_i$ vertex of the graph is visited, the fault list is updated, by removing faults that are covered by the corresponding test phase. Starting from $V_0$, a path represents an ordered list of configuration and test phases. A fault $F$ affecting the network

modules might deviate the expected behavior of the path, and one or more test phases may fail. If such a situation happens, the path is a test for $F$.

The proposed formulation of the $A^*$ algorithm minimizes the following function:

$$f(n) = g(n, \mathscr{F}) + h(p) \tag{6.3}$$

where $p$ is a path, $n$ is the last vertex on $p$, and $\mathscr{F}$ is the list of faults that are detected by $p$. Moreover, a goal function $G(p)$ is introduced, that states whether or not $p$ is a goal path (i.e., if it covers the whole list of testable faults). This means that the estimates of the cost from a state to the goal must be a function of the current path as well. The heuristic function that computes those estimates is therefore denoted as $h(p)$ in Eq. (6.3) as well. Details about the heuristic function are given later in this section. In classic $A^*$ we keep a frontier of open vertices that corresponds to a tree of partial paths rooted in the initial state. Each of these vertices is labeled with information that is used to keep track of the current best path (in terms of actual cost plus estimated cost to the goal) from the initial state to such a vertex. Every time a path to a vertex with a lower actual cost is found, such information (predecessor vertex) is updated. This approach is sound because the estimated cost for the vertices does not change. Therefore, a lower actual cost makes the new path always preferable. In our case, depending on the path we follow to reach a given vertex, the set of faults that are covered by the path may vary. Therefore, we need to keep multiple instances of each vertex open at any given time, in order to take into account the different paths to reach the vertex along with their different set of detected faults. The proposed solution is to keep a frontier of open vertices like in classic $A^*$, however each of these vertices maintains a hash table of paths with keys equal to the set of detected faults. When a new path to a vertex is found, if it has the same set of detected faults of a previously found path, its actual cost is checked and, if lower that the previous path, the latter is updated. This contribution is indicated with $g(n, \mathscr{F})$ in Eq. (6.3).

In other words, each vertex keeps track of a set of paths, each with different set of detected faults. A new path overwrites a previously stored one if it detects the same set of faults but has a lower actual cost. For each state we also keep track of the current best path to reach it, that is the currently open path to the state that has the lower combined cost (i.e., actual cost plus estimated cost to the goal). The estimated cost to the goal is computed by means of an admissible heuristic function.

```
begin A-star
    OpenQueue ← ∅;
    ClosedPaths ← ∅;
    insert reset configuration state V_0 into OpenQueue;
    while OpenQueue ≠ ∅ do
        extract V_i with the lowest f(V_i) from OpenQueue;
        p ← best path to V_i;
        put p into ClosedPaths;
        if G(p) is true then
            return p;
        end
        N ← neighbors of V_i;
        foreach V_j ∈ N do
            q ← path p connected to V_j;
            if q ∉ ClosedPaths then
                key ← faults covered by q;
                cost_p ← actual cost of path p;
                cost_ij ← transition cost from V_i to V_j;
                if cost_p + cost_ij < g(V_j, key) then
                    update g(V_j, key);
                    update f(V_j) with h(q);
                end
                put V_j into OpenQueue;
            end
        end
    end
end
```

Fig. 6.2 Pseudo-code of the optimal approach based on the $A^*$ algorithm.

The pseudo-code of the algorithm is reported in Fig. 6.2, where the $f$, $g$, and $h$ functions are referred to Eq. (6.3), $G$ is the goal function, *key* corresponds to $\mathscr{F}$, *OpenQueue* is the frontier, and *ClosedPaths* contains the list of paths that are proved to be non-optimal (i.e., they either do not detect all faults or they do but with a higher test time than the current best path). Briefly, the algorithm iteratively extracts the current best open path from the frontier, visits its neighborhood by updating/adding open paths until the goal is reached or no more open paths remain. When a best path to a node is extracted from the frontier, so that the goal function is satisfied, it represents the optimal solution (the algorithm is exhaustive in its search).

The performance in terms of run-time needed to reach the goal highly depends on the heuristic function. The proposed heuristic uses the length of the segments connected to each configurable element to estimate the cost of the remaining tests required to fully cover the network faults. Given a reconfigurable module $M$ with $k$ selectable segments, a fault $F$ that forces $M$ to select the segment $s_i$ can be detected by configuring $M$ so that a segment $s_j$, with $j$ different than $i$, is included in the active path, and then shifting a test vector into the network. Such test vector is at least as long as $s_j$. According to this reasoning, the contribution of $F$ to the heuristic function is comparable to the length of the shortest segment other than $s_i$, plus the number of configuration bits of the $F-$related module; if such a segment includes other reconfigurable modules, such modules and the segments they control are not counted; this permits not to take a segment into consideration multiple times in the heuristic function computation. For example, the fault that forces the ScanMux module of Fig. 5.6 to the value 1 is detected by a configuration in which ScanMux is set to 0 and selects the segment that includes $\text{TDR}_0$. The contribution to the heuristic function of such a fault is the $\text{TDR}_0$ length plus the configuration bit (i.e., $2+1=3$). The cost of the opposite ScanMux fault is estimated as the length of a modified version of the other segment (i.e., the one which includes the two SIBs), in which the inner SIBs have been removed; thus, the cost is zero for the segment plus the configuration bit (i.e., $0+1=1$).

The heuristic function value is computed while considering each of the remaining untested faults (i.e., the active faults) of the fault list. When the active fault list is empty, the goal is reached (i.e., the path is a test sequence). The application of the algorithm on the example network in Fig. 5.6 produces the following test sequence after reset:

- Session 1
  1. Configuration 1,D,D
  2. Test: $\text{SIB}_1$ stuck-at-A, $\text{SIB}_2$ stuck-at-A
- Session 2
  1. Configuration 1,A,A
  2. Test: $\text{SIB}_1$ stuck-at-D, $\text{SIB}_2$ stuck-at-D, ScanMux stuck-at-0
- Session 3
  1. Configuration 0,A,A
  2. Test: ScanMux stuck-at-1

### 6.3.2 Enhanced Version

The time and memory requirements of the complete algorithm proposed in the previous subsection depend on the total number of possible configurations in the network. In general, with $n$ configuration bits (one or more for every reconfigurable module), there are $2^n$ configurations. In this section, an heuristic strategy able to reduce the search space is proposed. This goal is achieved by reducing the list of admitted (i.e., considered by the search algorithm) network configurations, through an analysis of the network topology. The strategy is composed of two phases. In the first phase, the configuration list is reduced, by applying some constraints. The collateral effect of such a reduction is that some faults may become untestable using the admitted configurations, only. Thus, the second phase consists of adding some of the removed configurations back into the configuration list up until the full network testability is guaranteed again. As a result, the optimal approach performs the minimum cost search on a reduced configuration graph, improving both run-time and memory requirements.

#### First Phase

The main idea behind the first phase is to remove all the configurations such that, given a certain reconfigurable module $M_i$:

1. $M_i$ is not *controllable*, i.e., it cannot be programmed with a single configuration vector, because its configuration bits are not part of the active path;
2. $M_i$ is not *observable*, i.e., it is not part of the active path, thus the effects of its faults cannot be observed.

If the above conditions are both true, $M_i$ is forced to its reset state (i.e., it is configured to select the same segment of the network reset configuration). The same reasoning is applied for all reconfigurable modules in the network. This phase is represented as a Boolean problem. For each reconfigurable module, the above predicates are represented with controllability, observability, and reset Boolean constraints.

In the following, the procedure is explained in details. The topology graph (introduced in Section 6.2.1) is annotated with Boolean variables, according to the following procedure:

1. each vertex associated to a reconfigurable module $M_i$ is annotated with a set of *controlling variables*, i.e., the configuration of the reconfigurable module controlling the segment of the network that includes the configuration bits of $M_i$;
2. each edge is annotated with a set of *observing variables*, i.e., the configuration of the reconfigurable module controlling the segment of the network associated to the edge;
3. for each reconfigurable module vertex, the outgoing edge that corresponds to the segment selected on the network reset is marked.

A controlling variable typically is equal to an observing variable of the graph. As an example, let us consider the annotated graph reported in Fig. 6.3, which corresponds to the example network in Fig. 5.6 (the original topology graph is shown in Fig. 6.1). In this graph, Boolean variables $s_{ij}$ are used to represent the state of each segment. The controlling variable of module ScanMux is equal to the observing variable of the egde connecting TDI to ScanMux. In the network topology, this edge is associated to the top-level segment (i.e., the segment outgoing from TDI, before the fork, and then outgoing from ScanMux up to TDO), which includes the configuration bit of ScanMux. In the same way, the modules $\text{SIB}_1$ and $\text{SIB}_2$ are controlled by the variable associated to one of the outgoing edges of ScanMux. In particular, this edge corresponds to the segment of the network that includes the configuration bits of $\text{SIB}_1$ and $\text{SIB}_2$ (they are referred to as SIBs with local control in [92]).

Given a reconfigurable module $M_i$, the following constraints are defined:

- $C(M_i)$ is the controllability constraint. It corresponds to the logical conjunction of each controlling variable that annotates the vertex associated to $M_i$. A module $M_i$ is controlled iff $C(M_i)$ is satisfied, i.e., its configuration bits are part of the active path, thus it can be programmed with a single configuration vector.
- $O(M_i)$ is the observability constraint. It corresponds to the logical conjunction of each observing variable associated to the segment that includes $M_i$. In the graph, this variable annotates the edge outgoing from the reconfigurable module that controls such a segment. A module $M_i$ is observed iff $O(M_i)$ is satisfied, i.e., it is part of the active path. For the modules at top level, the observability constraint is assumed to be always true ($\top$).

- $R(M_i)$ is the reset constraint. It corresponds to the logical conjunction of each observing variable of the marked outgoing edge of the vertex associated to $M_i$. It denotes the reset configuration of $M_i$.

In case of a module $M_i$ with local control (as defined in [92]), the constraints $C(M_i)$ and $O(M_i)$ are coincident. As an example, the constraints derived from the graph in Fig. 6.3, which only contains modules with local control, are the following:

$$\begin{aligned} C(ScanMux) &= \top & O(ScanMux) &= \top \\ C(SIB_1) &= s_{12} & O(SIB_1) &= s_{12} \\ C(SIB_2) &= s_{12} & O(SIB_2) &= s_{12} \end{aligned}$$

From such representation we can derive a Boolean formula that has to be satisfied by all admitted configurations, as follows:

$$\bigwedge_{M_i} \big(R(M_i) \vee C(M_i) \vee O(M_i)\big) \tag{6.4}$$

For the network of the graph in Fig. 6.3, the resulting Boolean formula is the following:

$$(s_{22} \vee s_{12}) \wedge (s_{32} \vee s_{12}) \tag{6.5}$$

**Second Phase**

For each configuration, the list of testable faults can be identified. Hence, the undetected faults of the admitted configuration list (i.e., the subset of configurations satisfying Eq. (6.4)) can be derived. Each escaping fault can be either untestable (by any possible configuration) or testable by some configuration not satisfying the Boolean formula. For each of such faults, an iterative process starts from the reconfigurable module affected by the fault, and at each iteration relaxes the constraints on the modules in the controlled segment (each fault on a reconfigurable module can be associated to a controlled segment). The process stops when either the fault becomes testable or is proved to be untestable. If the fault is untestable, the original constraints are resumed.

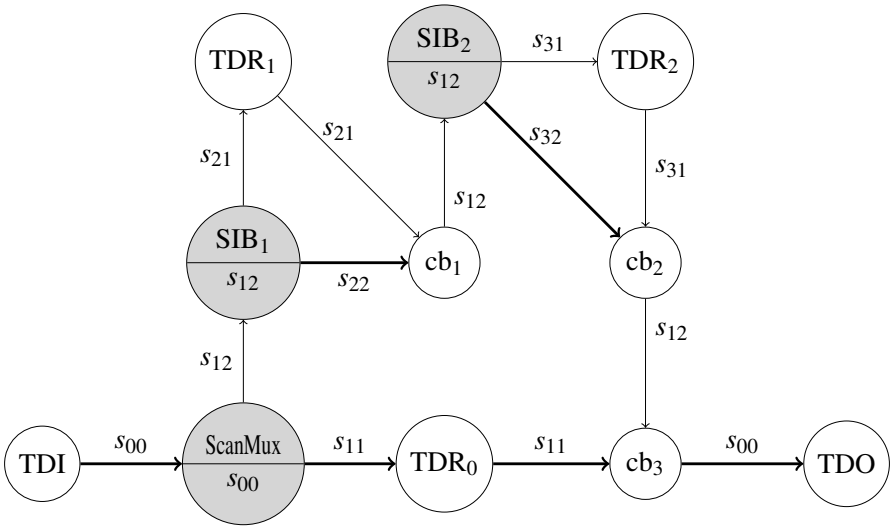



Fig. 6.3 Topology graph of the example network in Fig. 5.6 annotated for heuristic optimization.

In the previous example of the graph in Fig. 6.3, as already discussed in Section 6.1.2, the fault that forces the ScanMux to always select the segment connected to input 1 (see Fig. 5.6) can be detected only by a subset of configurations in which the ScanMux is part of the active path. In particular, according to Table 6.1, the configuration in which the ScanMux selects the segment connected to input 0 (i.e., the segment which excites the fault) and the modules $\text{SIB}_1$ and $\text{SIB}_2$ are de-asserted creates a faulty path with the same length of the active path (see Table 5.1). Thus, this configuration is not able to detect the fault, while all others are. Since $\text{SIB}_1$ and $\text{SIB}_2$ are neither controllable nor observable in such a configuration of the ScanMux, they are constrained to their reset configurations by the Boolean formula. In order to make such a fault detectable, an iterative process is started. By relaxing the constraint that forces $\text{SIB}_1$ to its reset state, an admitted configuration in which the active path and the faulty path have different length is found (configuration 0,A,D in the table). The formula becomes the following:

$$s_{32} \vee s_{12} \tag{6.6}$$

After the application of the heuristic optimization on the example network, the optimal approach based on $A^*$ uses a graph composed of 12 vertices only (6 for configuration and 6 for test) instead of 16. The optimal solution is not compromised and the test sequence found by the algorithm is the same.

### 6.3.3 Sub-Optimal Approaches

For large networks, the optimal approach based on $A^*$ is hardly applicable due to the excessive search space size. The proposed heuristic optimization is helpful to dramatically reduce it, but it does not solve the problem for all networks. In such situations, a scalable approach is preferred, even if the total test time obtainable is sub-optimal. In the following, two alternative approaches are shown, both based on topology graph traversal: a depth-first approach and a breadth-first approach.

When a test phase is performed, the fault list can be updated by removing the faults that have been tested. Faults affecting reconfigurable modules are associated to outgoing edges of their corresponding vertices.

The proposed strategies apply a sequence of test sessions by traversing the topology graph. At each step, a vertex associated to a reconfigurable module $M_i$ is found in the graph. In order to change the configuration of $M_i$, a configuration phase is performed, which consists of one or more configuration vectors. After the configuration phase, a different outgoing edge of vertex $M_i$ becomes active (as defined in Section 6.2.1), while the previously selected edge becomes inactive. As a result, the active path of the network changes. Then, a test phase is performed and the fault list is updated. By carefully identifying target edges on the graph, the process alternates these two phases, until the full test coverage is reached. The two strategies differ in the way these edges are selected.

#### Depth-First

The topology graph is traversed by following a depth-first approach. At each step of the graph traversal, a subset of reconfigurable modules is selected, that are part of the current active path. In the set of selected modules, the configuration is changed only for each module that is able to excite some of the untested faults and lies at the maximum depth. The depth of each module is found as an annotation of the topology graph vertices (see Section 6.2.1). All new configurations are applied together by means of a single configuration pattern, in case all the configuration bits are part of the active path (i.e., reconfigurable modules with local control), otherwise multiple configuration patterns are needed. A configuration in which all excited faults become observable is reached (i.e., in which the selected modules are part of the active path),

```
begin Depth-first
    FL ← all testable faults;
    while FL ≠ ∅ do
        M ← reconfigurable modules in the active path;
        d ← maximum depth of m ∈ M which is able to excite a fault in any
          configuration;
        foreach m ∈ M do
            if depth(m) = d and m is able to excite a fault then
                activate an outgoing edge from m that excites a fault;
            end
        end
        foreach just configured m ∈ M do
            reach a configuration in which m is part of the active path;
        end
        apply a test pattern;
        remove observed faults from the fault list;
        if all m ∈ M are fully tested then
            reach a configuration in which untested modules are part of the
              active path;
        end
    end
end
```

Fig. 6.4 Pseudo-code of the sub-optimal approach based on the depth-first algorithm.

then a test pattern is applied. The process is repeated until all faults are covered. The pseudo-code of the depth-first approach is reported in Fig. 6.4.

As an example, the application of the depth-first strategy on the graph in Fig. 6.1 produces the following test sequence after reset:

- Session 1
  1. Configuration 1,D,D
  2. Test: $SIB_1$ stuck-at-A, $SIB_2$ stuck-at-A
- Session 2
  1. Configuration 1,A,A
  2. Test: $SIB_1$ stuck-at-D, $SIB_2$ stuck-at-D,
     SMux stuck-at-0
- Session 3

1. Configuration 0,A,A
2. Test: SMux stuck-at-1

In the case of the example, the algorithm is able to produce the same test sequence produced by $A^*$.

### Breadth-First

The topology graph is traversed by following a breadth-first approach. The algorithm groups reconfigurable modules into levels, according to their hierarchical depth. Starting from the top-level, modules are tested level by level. At each iteration, the network is configured such that one or more modules of the target level are part of the active path and new faults can be excited. Then, a test pattern is applied and new faults that are excited and observed are removed from the fault list. Once all reconfigurable modules of the target level have been fully tested, the next level is considered, until the maximum depth of the network has been reached or all faults have been detected. The pseudo-code of the breadth-first approach is reported in Fig. 6.5.

As an example, the application of the breadth-first strategy on the graph in Fig. 6.1 produces the following test sequence after reset:

- Session 1
  1. Configuration 1,D,D
  2. Configuration 0,A,D
  3. Test: $\text{SMux}_1$ stuck-at-1
- Session 2
  1. Configuration 1,A,D
  2. Test: $\text{SIB}_1$ stuck-at-D, $\text{SIB}_2$ stuck-at-A, SMux stuck-at-0
- Session 3
  1. Configuration 1,D,A
  2. Test: $\text{SIB}_1$ stuck-at-A, $\text{SIB}_2$ stuck-at-D

```
begin Breadth-first
    FL ← all testable faults;
    d ← 1;
    while FL ≠ ∅ do
        M ← reconfigurable modules of level d;
        while all m ∈ M are not fully tested do
            foreach m ∈ M do
                if m is able to excite a fault then
                    activate an outgoing edge from m that excites a fault;
                end
            end
            repeat
                reach a configuration in which one or more m are part of the
                  active path;
                apply a test pattern;
                remove observed faults from the fault list;
            until all previously configured m ∈ M have been considered;
        end
        d ← d + 1;
    end
end
```

Fig. 6.5 Pseudo-code of the sub-optimal approach based on the breadth-first algorithm.

## 6.4 Experimental results

The effectiveness in terms of test duration of the proposed algorithms has been evaluated with an in-house tool on the ITC'16 benchmarks of IEEE Std 1687 scan networks [90]. Moreover, additional networks have been synthesized, which show the main differences between the proposed algorithms. The $A^*$ algorithm has been compared against the depth-first search and the breadth-first search algorithms.

The tool, written in Java, is able to read the network topology described in different formats. The *ICL Tools Software library* from [128] has been included in the developed tool. The tool also allows to configure the cost parameters $t_i^c$ and $t_i^t$ presented in Section 6.3.

The experiments were run on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. Each benchmark network has been tested using

Table 6.3 Characteristics of the ITC'16 benchmark networks

| Network | #SIBs | #ScanMuxes | #Config. bits | Max depth | Longest path | Total scan cells |
|---|---|---|---|---|---|---|
| Mingle | 10 | 3 | 13 | 4 | 171 | 270 |
| TreeBalanced | 43 | 3 | 48 | 7 | 5,219 | 5,581 |
| TreeFlat_Ex | 57 | 3 | 62 | 5 | 5,100 | 5,195 |
| TreeUnbalanc. | 28 | – | 28 | 11 | 42,630 | 42,630 |
| a586710 | – | 32 | 32 | 4 | 42,381 | 42,410 |
| p22810 | 270 | – | 270 | 2 | 30,356 | 30,356 |
| p34392 | – | 96 | 96 | 4 | 27,899 | 27,990 |
| p93791 | – | 596 | 596 | 4 | 100,709 | 101,291 |
| q12710 | 27 | – | 27 | 2 | 26,185 | 26,185 |
| t512505 | 159 | – | 159 | 2 | 77,005 | 77,005 |
| N132D4 | 39 | 40 | 79 | 5 | 2,555 | 2,991 |
| N17D3 | 7 | 8 | 15 | 4 | 372 | 462 |
| N32D6 | 13 | 10 | 23 | 4 | 84,039 | 96,158 |
| N73D14 | 29 | 17 | 46 | 12 | 190,526 | 218,869 |
| NE1200P430 | 381 | 430 | 811 | 127 | 88,471 | 108,148 |
| NE600P150 | 207 | 194 | 401 | 78 | 23,423 | 28,250 |

$A^*$ (in the optimized version) and both sub-optimal approaches (depth-first and breadth-first). Each experiment used a maximum heap size of 64 GB.

### 6.4.1 Experiments with Known Benchmarks

The key characteristics of the ITC'16 benchmark networks are detailed in Table 6.3. For each network, the table reports first the number of SIBs and ScanMuxes. The fourth column refers to the number of configuration bits that control the configuration of SIBs and ScanMuxes. The column *Max depth* indicates the maximum hierarchical depth of each network (for SIB-based networks this value equals to the maximum number of nested SIBs, according to [90]). Finally, the column *Longest path* reports the maximum possible number of scan cells on active path, while *Total scan cells* is the sum of lengths of all scan registers in each network.

In the experiments, the cost for a configuration pattern has been set to the active path length plus the JTAG protocol overhead (to move from shift to update, see [97]). The cost for a test pattern has been set to the sum of the following contributions:

1. the JTAG protocol overhead (to move from update to shift), which has been set to 5;
2. the longest path length (initialization vector, see Section 6.1.2);
3. the active path length plus two (a sequence of alternated 0s and 1s as long as the active path followed by two consecutive 1s).

Due to tractability limitations (given the size of the search space), $A^*$ resulted in out-of-memory failures for most of the benchmarks, while the sub-optimal algorithms were occupying few memory resources even for very large networks (no out-of-memory has been experienced by reducing the heap size up to 1GB). Experimental results on ITC'16 benchmarks are shown in Table 6.4. For each algorithm (depth-first is referred to a DF, while BF indicates breadth-first), the table reports the number of sessions (#*S* in column 3), each one composed of one or more configuration vectors and a test vector. The total number of configuration vectors is also reported (#*cv* in column 4). The table also indicates the number of clock cycles required by configuration patterns ($T_c$ in column 5) and test patterns ($T_t$ in colum 6), as well as their sum (column 7). Finally, for sub-optimal approaches, the ratio of the total time over the $A^*$ total time is reported (column 8), when $A^*$ succeeded, i.e., only for the networks *Mingle* and *N17D3*. All modeled faults have been covered in each experiment (i.e., test coverage is 100%).

An analysis of Table 6.4 shows that the two sub-optimal approaches produced the same results (i.e., the total time) for most of the benchmark networks. The reason for this is due to the topology of the benchmarks: all networks but *TreeBalanced* and *TreeFlat_Ex* contain SIBs and 2-to-1 ScanMuxes (i.e., with one configuration bit). Moreover, in all such networks that have only 2-to-1 ScanMuxes, one of the two segments controlled by a ScanMux does not include any other nested SIB or ScanMux, with the only exception of the network *Mingle*. In the following, ScanMuxes of this kind are referred to as *Unbalanced ScanMuxes*, while ScanMuxes that have nested reconfigurable modules on both segments (more than one segment, in case of ScanMuxes larger than 2-to-1) are referred to as *Balanced ScanMuxes*. Balanced ScanMuxes are one category of the modules that determine a different result between depth-first and bread-first, as for the network Mingle. The other factor is the presence of ScanMuxes with more than one configuration bits (i.e., larger than 2-to-1) not placed at the maximum hierarchical depth (level) of the network. In the benchmarks, the networks TreeBalanced and TreeFlat_Ex both have one ScanMux with 3 configuration bits. The only difference between them is that the

Table 6.4 Experimental results on the ITC'16 benchmark networks

| Network | Alg. | #$S$ | #$cv$ | $T_c$ [cc] | $T_t$ [cc] | $T$ [cc] | $T/T_{A^*}$ |
|---|---|---|---|---|---|---|---|
| Mingle | $A^*$ | 7 | 7 | 337 | 1,684 | 2,021 | – |
| | DF | 7 | 6 | 362 | 1,920 | 2,282 | 1.13 |
| | BF | 8 | 8 | 453 | 2,173 | 2,626 | 1.30 |
| TreeB. | DF | 8 | 7 | 8,580 | 60,789 | 69,369 | – |
| | BF | 8 | 7 | 8,580 | 60,789 | 69,369 | – |
| TreeFl. | DF | 6 | 5 | 15,263 | 56,078 | 71,341 | – |
| | BF | 7 | 8 | 30,250 | 66,177 | 96,427 | – |
| TreeUnb. | DF | 12 | 11 | 237,475 | 834,324 | 1,071,799 | – |
| | BF | 12 | 11 | 237,475 | 834,324 | 1,071,799 | – |
| a586710 | DF | 5 | 4 | 1471 | 298,153 | 299,624 | – |
| | BF | 5 | 4 | 1,471 | 298,153 | 299,624 | – |
| p22810 | DF | 3 | 2 | 573 | 152,364 | 152,937 | – |
| | BF | 3 | 2 | 573 | 152,364 | 152,937 | – |
| p34392 | DF | 5 | 4 | 697 | 196,005 | 196,702 | – |
| | BF | 5 | 4 | 697 | 196,005 | 196,702 | – |
| p93791 | DF | 5 | 4 | 1,950 | 706,928 | 708,878 | – |
| | BF | 5 | 4 | 1,950 | 706,928 | 708,878 | – |
| q12710 | DF | 3 | 2 | 43 | 130,979 | 131,022 | – |
| | BF | 3 | 2 | 43 | 130,979 | 131,022 | – |
| t512505 | DF | 3 | 2 | 494 | 385,530 | 386,024 | – |
| | BF | 3 | 2 | 494 | 385,530 | 386,024 | – |
| N132D4 | DF | 6 | 5 | 9,332 | 29,399 | 38,731 | – |
| | BF | 6 | 5 | 9,332 | 29,399 | 38,731 | – |
| N17D3 | $A^*$ | 5 | 5 | 900 | 3,007 | 3,907 | – |
| | DF | 5 | 4 | 802 | 3,341 | 4,143 | 1.06 |
| | BF | 5 | 4 | 802 | 3,341 | 4,143 | 1.06 |
| N32D6 | DF | 5 | 4 | 183,439 | 759,031 | 942,470 | – |
| | BF | 5 | 4 | 183,439 | 759,031 | 942,470 | – |
| N73D14 | DF | 13 | 12 | 1,577,674 | 4,400,373 | 5,978,047 | – |
| | BF | 13 | 12 | 1,577,674 | 4,400,373 | 5,978,047 | – |
| NE1200P. | DF | 128 | 127 | 5,014,931 | 16,500,774 | 21,515,705 | – |
| | BF | 128 | 127 | 5,014,931 | 16,500,774 | 21,515,705 | – |
| NE600P. | DF | 79 | 78 | 916,829 | 2,809,897 | 3,726,726 | – |
| | BF | 79 | 78 | 916,829 | 2,809,897 | 3,726,726 | – |

ScanMux of network TreeBalanced is placed in the bottom hierarchical level, while in TreeFlat_Ex it is placed in an intermediate level. For this reason, only the network TreeFlat_Ex presents different results between depth-first and breadth-first.

### 6.4.2 Experiments with Synthesized Benchmarks

The effectiveness of the proposed algorithms has been also evaluated on new synthesized networks. A tool able to generate random networks (constrained with some parameters that affect the topology shape) has been purposely devised. The tool has been used in an evaluation experiment, where around 20k networks have been generated. Networks include both unbalanced and balanced ScanMuxes, also larger than 2-to-1, and are manageable with the $A^*$ approach. The characteristics of the generated networks are the following:

- number of ScanMuxes between 2 and 16;
- number of configuration bits between 2 and 19;
- longest path between 22 and 55,428 scan cells;
- cumulative path between 22 and 70,088 scan cells;
- maximum depth between 2 and 9 levels.

For each network, the total time of the test developed by depth-first and breadth-first approaches have been divided by the total time of the test generated by $A^*$ (as for the column $T/T_{A^*}$ of Table 6.4). The normal distribution of the results of the depth-first approach has a mean of 1.08 and a standard deviation of 0.11, while the breadth-first has mean equal to 1.15 and standard deviation equal to 0.15. The maximum values are 1.99 for depth-first and 2.15 for breadth-first. The cumulative distribution functions (CDF) of the two algorithms are reported in Fig. 6.6. The figure shows that depth-first and breadth-first approaches have been able to find the optimal solution (ratio over $A^*$ equal to 1) in 23% and 17% of the cases, respectively, while in 90% of the cases the test sequence produced by the two algorithms is long 1.23 and 1.34 times with respect to $A^*$ or shorter.

In order to understand how good these results are with respect to the worst case, another set of experiments have been performed. The purpose of these experiments has been to find some networks that are critical (in terms of test sequence duration) for depth-first and breadth-first. We used an evolutionary-based approach, using the

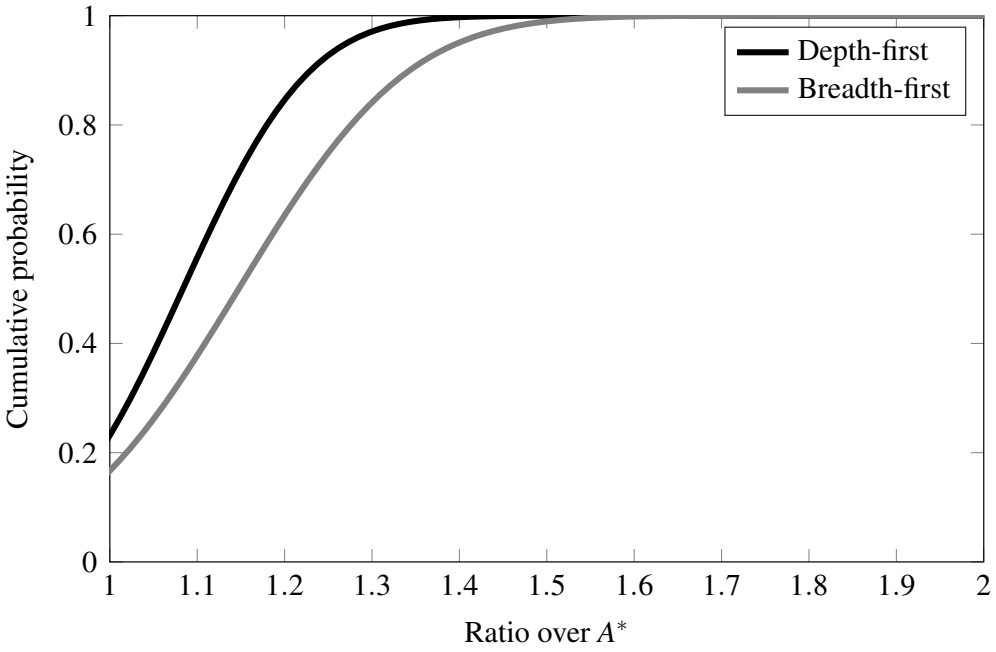



Fig. 6.6 Normal cumulative distribution function (CDF) of the ratio between sub-optimal approaches (depth-first in black, breadth-first in gray) and $A^*$ on the randomly generated networks.

evolutionary engine $\mu$GP (MicroGP) [136]. The approach is based on a loop, in which every iteration consists of the following steps:

1. $\mu$GP generates a scan network topology, according to some constraints;
2. the topology is read by the tool implementing the test algorithms and the fitness value of the network is computed;
3. $\mu$GP reads the fitness value and associate it to the corresponding network.

$\mu$GP maintains a population of networks and each time generates a new one by applying well-known genetic operators (based and mutation and crossover). After reading the fitness of the new generated network, $\mu$GP updates the population and other internal parameters. The loop is repeated until a stop condition is met (e.g., the fitness reaches a certain maximum value, or after reaching a fixed amount of wall-clock time or generated networks).

We have performed two evolutionary-based experiments. In the first one, the fitness value is the total time ratio between depth-first and $A^*$. In the second one, the ratio is between breadth-first and $A^*$. The experiments have been parallelized over 30 cores. After 45 days, 46,066 networks have been generated in the first experiment

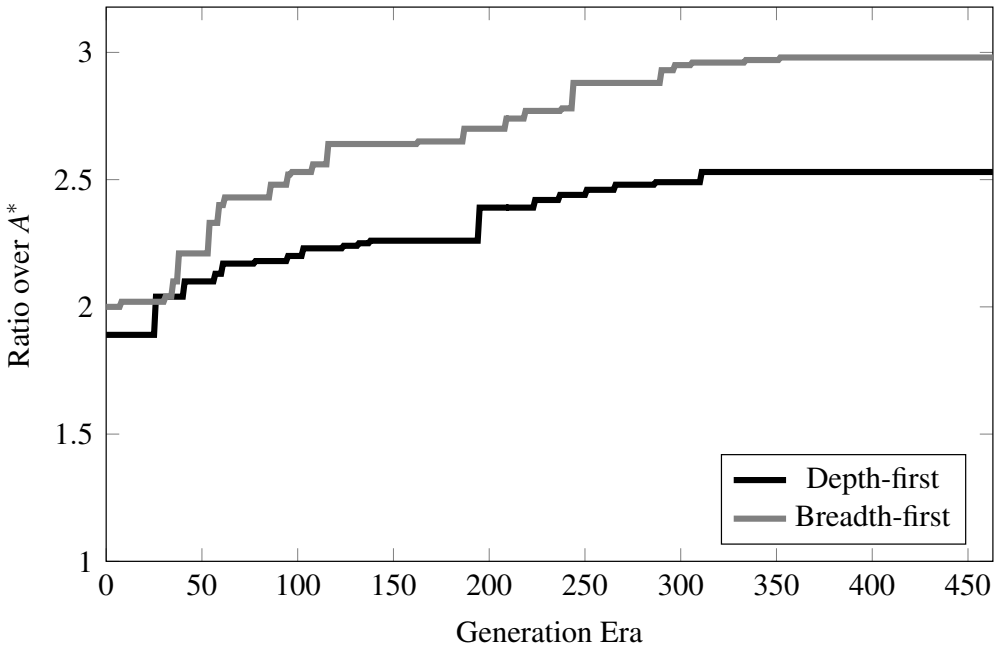



Fig. 6.7 Progression in time of the maximum fitness value for the evolutionary-based experiments.

and 46,823 networks in the second. The fitness in the depth-first experiment started from a maximum value of 1.89 after the first generation *era* (i.e., the first population of networks) and reached 2.53 after 315 eras. In the breadth-first experiment the maximum fitness started from 2 and reached 2.98 after 354 eras. The fitness evolution of the best network in the population is shown in Fig. 6.7.

Some interesting networks have been extracted from the population and their characteristics are reported in Table 6.5. The experimental results on the selected network are reported in Table 6.6. In the first network, depth-first resulted as efficient as $A^*$, while breadth-first has slightly worst performances. The opposite behavior can be observed for the second network, where breadth-first resulted to be highly efficient. Finally, the last two networks are the worst ones generated by the first and the second experiments, respectively.

It is important to highlight that evolutionary experiments do not guarantee that the global optimum is found, thus fitness values have not to be considered as worst-cases. However, evolutionary experiments outperformed random network generation in terms of maximum fitness values.

Table 6.5 Characteristics of the selected networks

| Network | #SIBs | #ScanMuxes | #Config. bits | Max depth | Longest path | Total scan cells |
|---|---|---|---|---|---|---|
| Network 1 | – | 4 | 6 | 3 | 8,142 | 8,884 |
| Network 2 | – | 5 | 8 | 3 | 6,325 | 6,557 |
| Network 3 | – | 6 | 8 | 3 | 24,580 | 24,851 |
| Network 4 | – | 7 | 10 | 3 | 34,169 | 34,543 |

Table 6.6 Experimental results on the selected networks

| Network | Alg. | $\#S$ | $\#cv$ | $T_c$ [cc] | $T_t$ [cc] | $T$ [cc] | $T/T_{A^*}$ |
|---|---|---|---|---|---|---|---|
| Network 1 | $A^*$ | 4 | 3 | 909 | 49,500 | 50,409 | – |
| | DF | 4 | 3 | 772 | 49,637 | 50,409 | 1.00 |
| | BF | 6 | 6 | 24,716 | 74,827 | 99,543 | 1.97 |
| Network 2 | $A^*$ | 5 | 4 | 6,796 | 38,650 | 45,446 | – |
| | DF | 4 | 5 | 12,717 | 56,413 | 69,130 | 1.52 |
| | BF | 5 | 5 | 6,838 | 38,811 | 45,649 | 1.004 |
| Network 3 | $A^*$ | 9 | 8 | 24,968 | 246,299 | 271,267 | – |
| | DF | 10 | 9 | 196,237 | 491,192 | 687,429 | 2.53 |
| | BF | 10 | 12 | 269,832 | 491,192 | 761,024 | 2.81 |
| Network 4 | $A^*$ | 5 | 4 | 34,586 | 205,458 | 240,044 | – |
| | DF | 5 | 4 | 101,916 | 340,616 | 442,532 | 1.84 |
| | BF | 7 | 8 | 237,434 | 476,999 | 714,433 | 2.98 |

## 6.5 Chapter Summary

The chapter presented several methods for the functional test of reconfigurable modules of a scan network.

The proposed methodology represents the network topology and configurations as graphs. An optimal test sequence in terms of total test time is generated by applying the $A^*$ search algorithm on the configuration graph, with a suitable heuristic to reduce the search space without compromising the final results. Sub-optimal approaches traverse the topology graph and are based on depth-first and breadth-first algorithms. Such approaches scale well on large networks, when the optimal approach is not applicable, while still producing a test set whose duration compares well with the optimal one.

# Chapter 7

# Diagnosis

Once a RSN has been found faulty, it may be important to identify the faulty elements. The objective of the work presented in this chapter is to extend the testing approach to diagnosis, i.e., providing a solution for the generation of a sequence of stimuli allowing to identify the faulty element in a faulty RSN. This work uses the same fault model adopted in Chapter 6, with some extensions to TDR faults.

While several methods have been proposed to identify the faults possibly affecting a scan chain, such as [137, 138, 134, 139, 140, 141, 142, 143], to the best of my knowledge this is the first attempt to attack the issue of diagnosis in a RSN. The main difference with respect to regular scan-chain diagnosis is that RSNs also include configurable modules: hence, from one side we need to deal with different possible paths, from the other we need to take faults affecting reconfigurable modules into account.

The approach proposed in this work is based on the usage of a *fault dictionary*, where the behavior of the network when each of the faults belonging to a given set of possible faults arises is stored. By comparing the observed faulty behavior with those in the dictionary, one can identify the fault which may be responsible for the observed behavior. Since only scan output signals (e.g., TDO) can be observed, the different faulty behaviors only differ for the values appearing on these signals.

Based on the analysis of the effects of permanent faults affecting the different elements in a network (TDRs, SIBs, ScanMuxes), one can determine the diagnostic properties of an existing test sequence. If the achieved diagnostic capabilities are not

enough, one could try to generate further stimuli able to achieve improved diagnostic capabilities. The proposed techniques allow the generation of test stimuli able to distinguish between any possible pair of faults in a network (if existing). The main contribution of the work presented in this chapter lies in describing a method to generate some distinguishing input stimuli for any pair of single permanent faults. Moreover, rules for the identification of classes of *undistinguishable faults* are provided. By definition, all faults in the same class always behave in exactly the same manner, no matter which input stimuli is applied to the network.

Experimental results gathered on a set of benchmark networks show the feasibility of the approach, and provide figures allowing to quantitatively estimate the length of the stimuli for the diagnosis of an IEEE 1687 network.

Depending on the application scenarios (and in particular on when the instruments are accessed), the diagnostic approach presented here can be used either at the end of manufacturing, or in the field (or in both cases).

The rest of this chapter is organized as follows: Section 7.1 introduces the basic vocabulary and notions about diagnosis, and briefly summarizes the functional fault model. Section 7.2 describes the proposed approach, which is first based on a diagnostic analysis (Section 7.2.1) followed by the test patterns generation (Section 7.2.2). Some experimental results an benchmark RSNs are presented in Section 7.3. Finally, Section 7.4 draws some conclusions.

The concepts and the results presented in this chapter have been also published in [15, 144, 90].

## 7.1 Terminology and Fault Model

The concepts related to configurations, vectors, and test time have been presented Section 6.1. In this section, the basic terminology is extended to diagnosis.

A given session $S_i$ is composed of two phases: a configuration phase $Cnf_i$, in which configuration vectors are applied, and a test phase $Tst_i$, in which test vectors are applied. A session $S_i$ is able to excite a set of faults $SFS_i$.

The active path (or *session path*) is the path connected between the scan input and scan output pins, while the faulty path is the path that would be selected if a

certain reconfigurable module $M_i$ in the active path was in the opposite state to the expected one. Details are given in Section 6.1.2, where faulty paths due to the faulty ScanMux in Fig. 5.6 are shown in Table 6.1.

Given two reconfigurable modules $M_i$ and $M_j$, they are defined to be *independent* whether a path between the scan input and scan output ports can be found, to which only one of the two modules belongs.

In the following, the concepts of fault dictionary and fault classes are introduced. Moreover, the functional fault model already introduced for reconfigurable modules (cf. Section 6.1.2) is summarized and extended to TDRs.

### 7.1.1 Fault Dictionary and Fault Classes

The approach to diagnosis proposed in this work is based on a pre-computed fault dictionary [3]: for each possible fault in the network, it is assumed that the corresponding behavior of the network when a given set of input stimuli (called *diagnostic sequence*) is applied has been stored in a suitable database, named *fault dictionary*. In the ideal case, the input stimuli devised for diagnosis are able to force the network to produce a different behavior for every possible fault. Hence, by looking at the observed behavior and by accessing to the database, one can identify the fault responsible for the misbehavior. In this case, taking any pair of faults $F_i$, $F_j$ from the list of $N$ possible faults, it is guaranteed that the behavior of the network under $F_i$ is different than the one produced under $F_j$. In this case $F_i$ and $F_j$ are said to be *distinguished*. In practice, it is possible that a diagnostic sequence is not able to distinguish all fault pairs, i.e., it is possible that some fault pairs remain *undistinguished*.

A common procedure used to generate the diagnostic sequence is based on starting from an existing test sequence, i.e., a set of input stimuli generated for test purpose. By performing some *diagnostic analysis* of the test sequence it is possible to compute the behavior of each of the $N$ faults in the fault list, and identify those fault pairs that are still undistinguished. For each of these pairs a *diagnostic procedure* is performed, which is able to generate an improved diagnostic sequence (adding new stimuli to the existing ones). The generated diagnostic sequence is able to distinguish between the faults in the target pair.

In some cases the diagnostic procedure may be unable to generate any distinguishing sequence for the target fault pair, either for excessive computational or

memory efforts (in which case the fault pair is marked as *aborted*), or because the procedure identified the two faults as *undistinguishable* (i.e., it proved that the network will always produce the same output in the presence of one or the other faults, no matter the input stimuli). Such faults are referred to as *Undistinguished Pairs* (UPs), in particular $UP_{TDR}$ are the UPs affecting TDRs, while $UP_{RM}$ are UPs on the reconfigurable modules.

All faults which are undistinguishable pairwise are said to belong to the same *equivalence class*. If a new test sequence has been generated, it is once again fault simulated, possibly labeling new fault pairs as distinguished; the process is then repeated again until a given stopping condition is reached, corresponding either to the case in which a maximum computational effort has been spent, or to the one in which all fault pairs have been considered, marking them either as aborted, or as distinguished, or as undistinguishable.

### 7.1.2 Fault Model

Following the approach already adopted in Chapter 6 for testing reconfigurable modules, this work tries to be as independent as possible on the implementation details of the RSN. Hence, the topology graph presented in Section 6.2.1 is used, in which the network is modeled as an interconnection of TDRs and switches (corresponding to SIBs and ScanMuxes).

Moreover, the possible defects affecting each element of the network are modeled at the highest possible level. In the following, details about the fault model for each kind of elements are given. For the purpose of this work, only single and permanent faults are considered.

#### Test Data Registers

It is assumed that each shift flip-flop (FF) in a TDR can be affected by a pair of stuck-at faults on its output (TDR stuck-at-0 and stuck-at-1). When one such fault exists in a TDR, the output of the network will produce a sequence of 0s or 1s (respectively) when the TDR is accessed. Since a stuck-at fault affecting one of the FFs in a TDR cannot be distinguished from a stuck-at fault in another FF in the same TDR without recurring to known responses of the attached instruments (as for

regular scan-chain diagnosis), only two *TDR faults* are defined, that group together stuck-at-0 and stuck-at-1 faults of all the TDR's FFs, respectively.

Other faults affecting FFs are not considered in this work. For example, a given FF may be stuck at the transparent mode. In such a situation, the length of the active path is reduced.

### Reconfigurable Modules

The fault model for reconfigurable modules is the same described in Section 6.1.2 and briefly summarized in the following:

- the high-level fault model considers the faulty SIB to be always asserted (SIB stuck-at-asserted) or de-asserted (SIB stuck-at-de-asserted), independently on the configuration of the network. The effect of such a fault is that the path connecting scan input (e.g., TDI) to scan output (e.g., TDO) pins after the network has been configured is not the expected one. Hence, a given sequence (e.g., composed of alternated 0s and 1s) is forced on the scan input, it will appear on scan output after a number of clock pulses different than expected, whose value depends on the path which has been erroneously selected due to the fault (i.e., the faulty path).
- the high-level fault model corresponds to forcing the ScanMux to select as input the wrong one. Considering 2-to-1 ScanMuxes for simplicity (the same will be done in the rest of this section), this fault model corresponds to a ScanMux always selecting the upper (ScanMux stuck-at-up) or lower (ScanMux stuck-at-down) input, independently on the value on the control input (and hence on the current configuration of the network). When one fault of this type affects a ScanMux in the network, the path introduced between scan input and scan output pins is different than the correct one. Hence, if a given sequence (e.g., composed of alternated 0s and 1s) is forced on the scan input, the sequence will appear on the scan output after a number of clock pulses which may be different than expected, and this number corresponds to the length of the faulty path which has been erroneously selected due to the fault (see example in Table 6.1).

## 7.2 Proposed Diagnostic Approach

When considering the test and diagnosis of an IEEE 1687 network, the used input stimuli correspond to a sequence of test sessions. Assuming that a test sequence has been properly generated, according to the guidelines given in Chapter 6, the following conditions hold:

1. each TDR is accessed at least once;
2. each SIB appears as asserted in at least one session, and as deasserted in at least one session
3. Each ScanMux appears configured in each possible configuration in at least one session.

All possible untestable faults in the network (e.g., faults affecting a ScanMux which switches between two TDRs having the same length) are ignored. All such untestable faults belong to the same equivalence class of the fault-free network[1].

### 7.2.1 Diagnostic Analysis

The proposed approach at first identifies the diagnostic properties of the existing test sequence, i.e., which fault pairs can be distinguished by it, and which cannot. The diagnostic analysis is based on simulation and determines the behavior of the network when each possible fault is present and some given input stimuli are applied.

Please note that during test application, each session may result in one of the three possible outcomes:

1. The expected output is produced: the first bit of the alternated sequence appears at the expected clock cycle on the scan output.
2. The output sequence does not include the alternated sequence: a fixed value (0 or 1) is produced on the scan output. This means that one of the TDRs in the path is faulty (stuck-at-0 or stuck-at-1).
3. The alternated sequence appears on the scan output at an unexpected time. This means that a SIB or a ScanMux belonging to the path is faulty, i.e., works

[1]These faults could be tested in functional mode if suitable assumptions on the instruments can be made (e.g., if they are input instruments and the values they produce are known). However, for sake of generality, this approach is not considered in this work.

in the opposite mode than expected (e.g., a SIB is configured as asserted, but behaves as de-asserted). Hence, the path between scan input and scan output is different than the expected one.

The goal of the diagnostic analysis phase is to compute for each possible fault which is the behavior of the network, and then to determine which pairs of faults are distinguished, and which are not.

### Diagnostic Analysis on TDRs

According to the presented functional fault model, faults affecting TDRs produce a different network behavior than faults affecting reconfigurable modules. Hence, all faults affecting TDRs are by definition distinguished from faults affecting reconfigurable modules.

In order to determine which pair of TDR faults are distinguished by a given test session, one may note that if a faulty TDR exists in the RSN, at least one session will fall into case 2 (among the three outcomes mentioned above). The faulty TDR can thus be identified by checking which sessions fail. All sessions whose path includes it should fail. In other words, the following properties can be stated:

- If two TDRs belong to the same segment, they will always appear together in any session, and all the related faults are undistinguishable.
- If two TDRs belong to different segments, but appeared together in all sessions of the current diagnostic sequence, the related faults are undistinguished.

Hence, the algorithm for diagnostic analysis should:

1. identify all TDRs tested in each session;
2. for each pair of TDRs, check whether there is at least one session, in which only one of the two TDRs appear:
   (a) if this is the case, the two TDRs are marked as distinguished;
   (b) otherwise, the two TDRs are marked as undistinguished.

### Diagnostic Analysis on Reconfigurable Modules

In order to determine which fault pairs composed of faults affecting SIBs and ScanMuxes are distinguished by a given test session, one may note that if a faulty

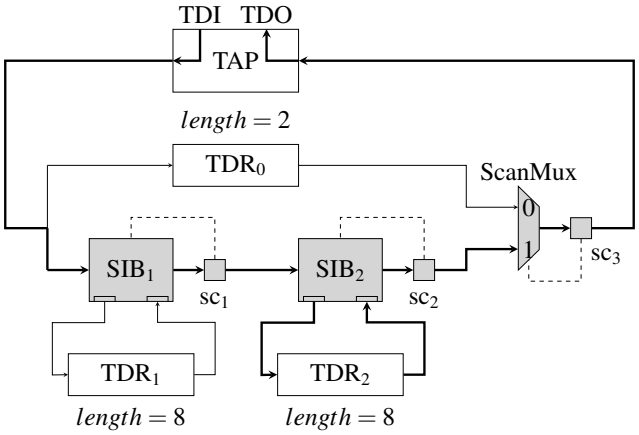
SIB or ScanMux exists in the network (according to the presented fault model), at least one session will fall into case 3. By looking at when the alternated sequence appears on the scan output, the length of the path existing between the scan intput and scan output pins is obtained. Using this length, a diagnostic procedure can try to identify the faulty SIB or ScanMux. In practice, the faulty paths should be preliminary computed for every session $S_i$.

By computing, for every session $S_i$, the related fault set $SFS_i$, the following properties can be stated:

1. given a session $S_i$ in the existing test sequence, all faults belonging to $SFS_i$ are distinguished from all faults related to SIBs and ScanMuxes which do not belong to the session path; in fact, if one of the faults in $SFS_i$ arises, $S_i$ will produce a failure, and a diagnostic procedure will be able to state that the fault responsible for the failure belongs to $SFS_i$;
2. given $S_i$ in the existing test sequence, and for every pair of faults both belonging to $SFS_i$, they can be distinguished one from the other if the two faulty paths have different lengths.

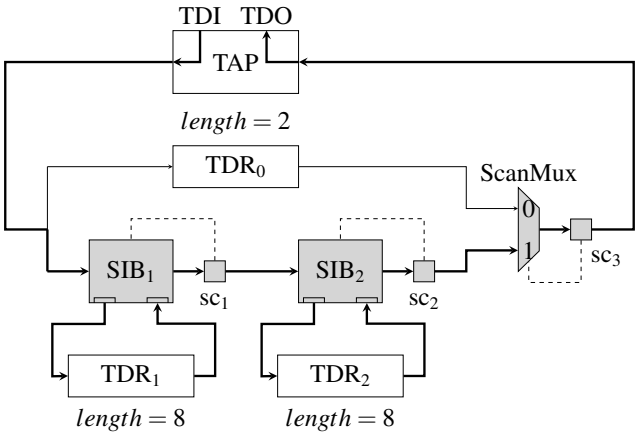
Using the example RSN of Fig. 5.6, and focusing a session $S_i$ which selects the path highlighted in Fig. 7.1a, the diagnostic analysis will come to the following conclusions:

- in case the network is fault free, the first bit of the alternated sequence will appear on TDO 11 clock cycles after it has been shifted in through TDI;
- in case any FF in $\text{TDR}_2$ is affected by a fault, the alternated sequence will not appear on TDO; by looking at the fixed value on TDO one can understand whether the fault is a stuck-at-0 or stuck-at-1; faults on $\text{TDR}_2$ are distinguished from all the other TDR faults, since any other TDR fault will not cause any effect during this session;
- in case a stuck-at-asserted fault in $\text{SIB}_1$ exists in the network, the segment controlled by $\text{SIB}_1$ will be added to the active path; thus, the faulty path length will be increased by the length of $\text{TDI}_1$ (i.e., $11+8=19$);
- in case of $\text{SIB}_2$ stuck-at-de-asserted fault, the segment it controls will be removed from the active path, which simply becomes $\text{TDI}\rightarrow\text{sc}_1\rightarrow\text{sc}_2\rightarrow\text{sc}_3\rightarrow\text{TDO}$; the faulty path length will be reduced by the length of $\text{TDR}_2$ (i.e., $11-8=3$): the first bit of the alternated sequence will appear on TDO three clock cycles after having being shifted-in;




| Fault | Len |
|---|---|
| – | 11 |
| $SIB_1$ stuck-at-ass. | 19 |
| $SIB_2$ stuck-at-de-ass. | 3 |
| ScanMux stuck-at-0 | 3 |

(a) Active path: TDI→$sc_1$→$TDR_2$→$sc_2$→$sc_3$→TDO




| Fault | Len |
|---|---|
| – | 19 |
| $SIB_1$ stuck-at-de-ass. | 11 |
| $SIB_2$ stuck-at-de-ass. | 11 |
| ScanMux stuck-at-0 | 3 |

(b) Active path: TDI→$TDR_1$→$sc_1$→$TDR_2$→$sc_2$→$sc_3$→TDO

Fig. 7.1 Examples of active path in a RSN and related faulty path lengths

- in case of ScanMux stuck-at-0 (or stuck-at-up), the selected segment will be the one including $TDR_0$, thus the active path length becomes 3 (i.e., the length of $TDR_0$ plus one);
- the faults $SIB_1$ stuck-at-asserted is distinguished from $SIB_2$ stuck-at-de-asserted and ScanMux stuck-at-0, since the latter will produce different observable misbehavior during $S_i$; moreover, $SIB_2$ stuck-at-de-asserted and ScanMux stuck-at-0 are undistinguished, since their faulty path lengths will be the same.

### 7.2.2 Generation of New Patterns

The second phase in the proposed approach aims at generating new test sessions to be added to the existing test sequence, whose purpose is to increase its diagnostic capabilities.

At the end of the diagnostic analysis, a given number of fault pairs will have been marked as distinguished, while others will still be undistinguished. The diagnostic test session generation phase will perform the following operations:

1. select one pair of undistinguished faults;
2. generate a new test session able to distinguish them, if any; alternatively, the fault pair may be identified as a undistinguishable one;
3. if a new test session has been generated, the whole set of newly distinguished fault pairs is identified;
4. repeat from step 1 until either all distinguishable fault pairs have been distinguished, or a given threshold of computational effort has been reached.

Let us now focus on step 2. If the selected fault pair is composed of two undistinguished TDR faults, they can be distinguished by adding one further session to the diagnostic sequence, in which one of the two TDRs appears, while the other does not. Clearly, if the two TDRs belong to the same segment, the related faults are undistinguishable: no additional session can be found in this case.

If the selected fault pair is composed of two undistinguished SIB or ScanMux faults, the diagnostic procedure should first check whether the corresponding reconfigurable modules $M_i$ and $M_j$ are independent: if this is the case, the proposed method will generate a new test session, which will correspond to a path in which only one of $M_i$ and $M_j$ belongs. The new session will be able to distinguish the two faults, which can thus be marked as distinguished.

If the two configurable modules are not independent, the diagnostic procedure needs to generate a test session whose path includes both reconfiguration modules. In this case the network should be configured in such a way that the faulty paths of the two faults have different lengths. The new session (if any) will be able to distinguish the two faults, which can thus be marked as distinguished.

To clarify the proposed procedure, the RNS in Fig. 5.6 is used. In the previous example, the faults $SIB_2$ stuck-at-de-asserted and and ScanMux stuck-at-0 have been

declared undistinguished by applying a session whose active path is highlighted in Fig. 7.1a. $SIB_1$ and the ScanMux are clearly not independent, since any path from TDI to TDO includes or excludes both. Hence, the diagnostic procedure will look for a session, corresponding to a path including both. The new session should be characterized by the fact that the faulty path for the fault $SIB_2$ stuck-at-de-asserted should have a different length than the faulty path for ScanMux stuck-at-0, as for the active path highlighted in Fig. 7.1b, in which the two faulty paths have lengths 11 and 3, respectively, thus the two faults are distinguished.

If a session able to distinguish two faults does not exist, the two faults are marked as undistinguishable.

At the moment, the identification of the required configuration is performed by exhaustive enumeration of all possible configurations until one matching the target requirements is found. Clearly, this is not the optimum solution from the point of view of the total length required for applying the whole diagnostic sequence (which may be a very important parameter especially for on-line applications), but it still represents a computationally feasible solution, as far as networks with some tens of TDRs and reconfigurable elements are considered. More effective solutions could be adopted (e.g., based on the techniques proposed in [100] or in Chapter 6) if the identification of the shortest diagnostic sequence is targeted.

## 7.3 Experimental Results

The correctness of the above algorithms has been experimentally validated with a prototypical tool of about 1200 lines of C# code, implementing the proposed technique.

A subset of the ITC'16 benchmarks [90] has been used, together with a few other synthesized RSNs. The new generated networks include both SIBs and 2-to-1 ScanMuxes. The characteristics of the ITC'16 benchmarks are reported in Table 6.3, while Table 7.1 reports the characteristics of the new synthesized networks.

For each of the benchmarks, a first set of test sessions has been derived and used as the original test sequence. The number of test sessions $S$ and the duration $T^T$ (in terms of number of clock cycles) of this sequence is reported in columns 2 and 3 of Table 7.2. The sequences have been encoded in a database, where each record

Table 7.1 Characteristics of the new synthesized networks

| Network | #SIBs | #ScanMuxes | #TDRs | Max depth | Total scan cells |
|---|---|---|---|---|---|
| Synthetic1 | 31 | 37 | 63 | 3 | 2,225 |
| Synthetic2 | 16 | 18 | 31 | 1 | 1,080 |
| Synthetic3 | 11 | 21 | 40 | 2 | 1,354 |
| Synthetic4 | 38 | 21 | 64 | 7 | 247,428 |
| Synthetic5 | 32 | 32 | 56 | 4 | 1,949 |
| Synthetic6 | 31 | 33 | 62 | 6 | 313,890 |
| Synthetic7 | 44 | 50 | 90 | 3 | 427,776 |

contains the scan-path length for each fault; thus, the database size is proportional to the number of test sessions and the amount of faults. Then, the tool implementing the proposed methodology has been run and the number of Undistinguished Pairs (UPs, as defined in Section 7.1.1) produced by the existing test sequence has been derived. Columns 4 and 5 of the table separately provide the number of $UP_{TDR}$ and $UP_{RM}$. The tool was then run to add new sessions to the existing sequence to cover all UPs: as a result, the tool generated a diagnostic sequence, whose characteristics, in terms of number of sessions, duration ($T^D$), duration increase ratio with respect to the original test sequence ($T^D/T^T$), and number of Undistinguished Pairs are reported in columns 6 to 10. The tool was always able to generate new sessions and to distinguish all possible fault pairs. The table also shows that the original test sequence is often already capable of distinguishing the reconfigurable modules fault pairs, while the added sessions are crucial to eliminate the undistinguished TDR pairs. The number of added sessions (as well as the increase in the length of the diagnostic sequence duration with respect to the original test one) depends on the number of possible pairs of faults, and thus exponentially depends on the number of reconfigurable modules.

The significant increase in the duration of the diagnostic sequence with respect to the initial test sequence can be partly explained with the increase in the number of sessions. On the other side, it should also be recalled that every session during the test lasts only until the moment in which the alternated sequence emerges from the scan output (which is expected to happen after a number of clock cycles equal to the selected path), so that it is possible to check if the alternated sequence appears on TDO at the right time. If not, the network is labeled as faulty. Conversely, during

Table 7.2 Test and diagnostic sequences characteristics

| Network | Test sequence | | | | Diagnostic sequence | | | | | Und.able pairs of faults | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\#S$ | $T^T$ [cc] | $UP_{TDR}$ | $UP_{RM}$ | $\#S$ (added) | $T^D$ [cc] | $T^D/T^T$ | $UP_{TDR}$ | $UP_{RM}$ | TDR | RM |
| TreeUnb. | 12 | 709,038 | 306 | 7 | 16 | 113,064,612 | 146.8 | 14 | 0 | 0 | 0 |
| a586710 | 3 | 209,933 | 10 | 0 | 4 | 3,879,326 | 18.5 | 0 | 0 | 0 | 0 |
| p22810 | 3 | 154,734 | 379 | 1 | 28 | 46,601,832 | 301.2 | 0 | 0 | 0 | 0 |
| p34392 | 3 | 117,499 | 174 | 5 | 20 | 20,665,284 | 175.9 | 0 | 0 | 0 | 0 |
| p93791 | 4 | 587,400 | 584 | 5 | 46 | 273,091,791 | 464.9 | 0 | 0 | 0 | 0 |
| N132D4 | 6 | 30,934 | 1,048 | 3 | 56 | 3,900,952 | 128.7 | 0 | 0 | 435 | 0 |
| N17D3 | 5 | 3,795 | 10 | 0 | 3 | 48,099 | 13.0 | 0 | 0 | 4 | 0 |
| N32D6 | 5 | 816,664 | 54 | 0 | 9 | 25,038,071 | 30.7 | 0 | 0 | 16 | 0 |
| N73D14 | 13 | 4,377,101 | 239 | 0 | 17 | 320,437,112 | 73.2 | 0 | 0 | 78 | 0 |
| NE1200P. | 128 | 14,447,050 | 31,906 | 216 | 262 | 66,415,530,376 | 4,164.6 | 643 | 0 | 598 | 0 |
| NE600P. | 79 | 2,495,300 | 9,343 | 104 | 141 | 4,373,190,098 | 1,560.8 | 427 | 0 | 305 | 0 |
| Synthetic1 | 4 | 16,032 | 577 | 1 | 52 | 2,219,397 | 141.9 | 0 | 0 | 206 | 0 |
| Synthetic2 | 2 | 4,746 | 225 | 6 | 37 | 263,866 | 56.9 | 0 | 0 | 105 | 0 |
| Synthetic3 | 4 | 10,978 | 305 | 1 | 20 | 333,280 | 31.0 | 0 | 0 | 260 | 0 |
| Synthetic4 | 9 | 3,312,514 | 452 | 0 | 30 | 353,225,042 | 106.6 | 0 | 0 | 63 | 0 |
| Synthetic5 | 5 | 16,596 | 378 | 8 | 44 | 2,065,800 | 127.5 | 0 | 0 | 68 | 0 |
| Synthetic6 | 7 | 3,577,391 | 434 | 0 | 25 | 267,851,629 | 74.9 | 0 | 0 | 204 | 0 |
| Synthetic7 | 5 | 3,565,527 | 992 | 0 | 40 | 480,683,792 | 134.8 | 0 | 0 | 301 | 0 |

diagnosis, this is not enough, since it is needed to also know when the alternated sequence arrives on the scan output: if the alternated sequence does not come within a given time, then the fault affects a TDR; if it arrives, the time of its arrival on the scan output suggests us which configurable element is faulty. Hence, the duration of each session when performing diagnosis is extended up to the length of the longest faulty path.

In the last two columns, the table finally reports the number of faulty TDR and reconfigurable modules (RM) pairs that the tool proved to be undistinguishable.

It is worth mentioning that the total running time of the tool for performing both the diagnostic analysis and the diagnostic test generation is negligible, accounting to less than a few seconds even for the larger networks.

## 7.4 Chapter Summary

This chapter addressed the issue of generating a sequence of input stimuli able to identify the faults possibly affecting TDRs and reconfigurable modules in IEEE Std 1687 networks. The proposed solution is based on the functional fault model and assumptions already introduced in Chapter 6. The described approach is able to first identify the diagnostic capabilities of an existing sequence (e.g., developed for testing purposes), and then to improve them by adding new sessions to the existing sequence. The resulting sequences can be applied at the end of manufacturing, or in-field, depending on when the instruments made accessible by the network are used. Rules are given, such that their fulfillment allows the generated test sequence to distinguish any pair of distinguishable faults. Experimental results are reported on a set of benchmark networks.

Future works are intended to target more complex fault models (e.g., affecting the reset and clock signals) and on the optimization of the proposed technique, thus reducing the length of the generated sequences.

# Chapter 8

# Summary of Part II

Reconfigurable scan networks have been the subject of this part of the thesis. These networks represent a way of interconnecting on-board instruments, more and more often embedded in modern electronic devices.

Given the existence of standards such as IEEE Std 1687, the design, integration, and usage of instruments and interconnecting infrastructure is attracting the interest of EDA vendor and new tools are being developed. The purpose of this thesis has been presenting the problem of testing the infrastructure offered by RSNs. Different methodologies have been proposed in this thesis, for the test and the diagnosis of possible faults affecting specific RSN structures, according to a high-level fault model. Such algorithms are completely automatic and rely on high-level network description, so their implementation in EDA tools is a real possibility.

One of the problems that has been addressed in this thesis is minimizing test time. It has been shown that this problem can be solved optimally, but it faces a severe scalability limitation as it requires exploring the network configuration space. However, non-optimal approaches have been presented, which explore the network topologically with little computational effort.

# Conclusions

This thesis addressed the problem of generating high-quality functional tests for microprocessor based systems, by combining different techniques, such as Software-Based Self-Test and systematic test pattern generation for Reconfigurable Scan Networks.

The first research question, concerning the adoption of SBST for the test of real-word processor based system has been tackled in the first part of this thesis. Several systematic algorithms have been proposed to support the work of test engineers in the development of a suite of test programs for the specific DUT. Each proposed algorithm has covered a particular sub-module of modern processors, such as the instruction decode unit, the register forwarding and pipeline interlocking unit, and the floating-point unit. Moreover, hard-to-test components managing multiple issue of instructions in RISC processors have been analyzed, such as duplicated computational modules, multi-port register files, and instruction prefetch buffer. An effective development flow for SBST generation has been also proposed, based on various levels, branches, and synchronization steps. The limitations imposed by operating systems have been handled, concerning solutions to deal with on-line constraints, and robust execution and management of SBST programs. Experimental results on several industrial and academic processors have shown the feasibility and the effectiveness of the proposed development flow, thus positively answering to the research question.

The second research question, concerning test and diagnosis of RSNs, has been tackled by providing systematic algorithms able to automatically generate test and diagnostic patterns able to cover functional faults, according to a proposed fault model. Contrarily to SBST, this work is relatively new in literature and represents a reference for future development on the topic. In details, the problem of minimizing the overall test time has been solved for the first time with an optimal solution, but

scalability limitations have been shown. Alternative sub-optimal solutions based on heuristics can be applied to overcome such a limitation. Finally, a diagnostic approach has been proposed for the first time. The devised algorithms have been experimented on a representative set of benchmark networks by means of prototypical tools, thus positively answering to the second research question and showing that such methodology can be integrated in commercial EDA tools.

# List of Research Contributions

A significant amount of work has been spent in devising deterministic algorithms for developing test programs for particular components of modern pipelined processors. In details:

- A new SBST technique based on partitioning the Instruction Set to systematically cover faults possibly affecting the functionality of Decode Units in RISC processors has been proposed and published in [8]. The paper has been slightly rephrased in Section 2.1, which also reports published results.
- Structural components implementing the register forwarding and the pipeline interlocking mechanisms in RISC processors have been tested using a SBST approach, which is based on the conversion of test patterns for such components (such as multiplexers) to sequences of instructions. Algorithms and experimental results published in [9] have been reported in Section 2.2.
- Several SBST algorithms have been proposed in Section 2.3 for the test of processors that implement multiple issue of instructions. At time of writing, only algorithms concerning duplicated computational modules (cf. Section 2.3.3) as well as related experimental results have been published in [10], while the other modules have been tacked for the first time in this thesis only, i.e., multiport register file (cf. Section 2.3.4), feed-forward paths (cf. Section 2.3.5), pipeline interlocking (cf. Section 2.3.6), and instruction prefetch buffer (cf. Section 2.3.7).
- A SBST approach for floating-point units based on both conversion of test patterns and systematic functional floating-point operations has been published in [11] and reported in Section 2.4.

The proposed algorithms have been applied to real-world processors following a systematic development flow. Part of my research has been finding best practices to

make such a development feasible even for large processors. In [12] the practical limitations imposed by the operating system have been introduced and a clever partitioning of the development flow in various levels, branches, and synchronization steps has been presented (and reported in Chapter 3). Moreover, the development flow presented in [12] has been applied to an extensive set of real-world processors from the automotive industry and experimental results have been published for the first time in this thesis only (cf. Section 3.4).

Concerning RSNs, the main contribution has been to address the problem of test (cf. Chapter 6) and diagnosis (cf. Chapter 7) of the main reconfigurable elements of the network. In details:

- the basic concepts of fault model, topology graph used to describe RSNs, and a first heuristic algorithm to test the target faults have been published in [13];
- the problem of minimizing the overall test time has been tacked in [14], based on a minimum-cost search among the possible network configurations;
- for the first time, a diagnostic analysis on RSNs has been published in [15], which introduces the basic concepts of fault pairs;
- a first diagnostic approach based on the enrichment of test vectors to support diagnosis has been published in [144].

At time of working on the above publications, freely-available benchmarks of RSNs were not existing, thus experiments were performed on in-house synthesized RSNs. Some of these network have then been selected and included in the ITC'16 set of benchmarks [90].

Chapter 6 is an extension of the ideas presented in [13, 14]. At time of writing, the amount of material published for the first time in this thesis only is substantial. In details:

- the description of the addressed problem and of all the proposed algorithms have been enhanced, in particular by formally describing the graphs (cf. Section 6.2) and presenting all the algorithms in details with their pseudo-codes (cf. Section 6.3);
- a new heuristic algorithm has been introduced and evaluated on all the selected benchmark networks;
- concerning the experimental results (cf. Section 6.4), the benchmark networks are now taken from the ITC'16 suite and a new set of benchmarks networks

have been added, which have been synthetically generated to explore the limits of the exact algorithms.

## Future Works

Future works in the SBST field are focused on three main aspects: integration of SBST in the design flow supported by EDA tools, optimization of time and memory budgets, and extension to new fault models. Regarding the last point, some works can be already found in the literature targeting dynamic faults (e.g., small delay faults). However, at the time of writing this thesis, the industrial domain where SBST is actually implemented (i.e., safety-critical systems, such as for automotive and avionic applications) only targets static fault models such as stuck-at faults to comply with safety regulations. Finally, commercial fault simulators are currently well-optimized for scan designs, while is not always feasible to implement an efficient fault grading environment for SBST, even worse if fault models different than stuck-at are targeted.

Future works on RSNs are mainly aimed at improving the techniques presented and incorporate them into the test flow and diagnosis of the complete design. Remember that one of the hypothesis of the proposed methodology has been that the knowledge of attached instruments is neglected. This means that part of the network that interfaces with the instruments is not tested systematically. Once this knowledge is granted and if a gate-level description of the system is available, it is possible to target test generation for conventional fault models such as stuck-at or transition delay.

The flexible and standardized way to access on-board instruments via RSNs is also attractive from the point of view of the system's self-test capability. Functional testing techniques such as SBST can benefit from the presence of the IEEE Std 1687 infrastructure. This scenario will also be examined in future activities.

# References


[1] Hans-Joachim Wunderlich. *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 9048132819, 9789048132812.

[2] Technopedia. *Device Under Test (DUT)*. URL: https://www.techopedia.com/definition/25924/device-under-test-dut (visited on 06/08/2017).

[3] M. Bushnell and Vishwani Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013. ISBN: 9781475781427.

[4] A. Touati, A. Bosio, P. Girard, A. Virazel, P. Bernardi, and M. S. Reorda. "Improving the Functional Test Delay Fault Coverage: A Microprocessor Case Study". In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2016, pp. 731–736. DOI: 10.1109/ISVLSI.2016.42.

[5] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. 2006.

[6] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. "Abstraction techniques for validation coverage analysis and test generation". In: *IEEE Transactions on Computers* 47.1 (Jan. 1998), pp. 2–14. ISSN: 0018-9340. DOI: 10.1109/12.656068.

[7] P. Mishra, N. Dutt, N. Krishnamurthy, and M. S. Ababir. "A top-down methodology for microprocessor validation". In: *IEEE Design Test of Computers* 21.2 (Mar. 2004), pp. 122–131. ISSN: 0740-7475. DOI: 10.1109/MDT.2004.1277905.

[8] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. Sonza Reorda, S. De Luca, R. Meregalli, and A. Sansonetti. "On the in-field functional testing of decode units in pipelined RISC processors". In: *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Oct. 2014, pp. 299–304. DOI: 10.1109/DFT.2014.6962090.

[9] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. Sonza Reorda, M. Grosso, and O. Ballan. "On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors". In: *2013 14th International Workshop on Microprocessor Test and Verification*. Dec. 2013, pp. 52–57. DOI: 10.1109/MTV.2013.10.

[10] P. Bernardi, C. Bovi, R. Cantoro, S. De Luca, R. Meregalli, D. Piumatti, E. Sanchez, and A. Sansonetti. "Software-based self-test techniques of computational modules in dual issue embedded processors". In: *2015 20th IEEE European Test Symposium (ETS)*. May 2015, pp. 1–2. DOI: 10.1109/ETS.2015.7138730.

[11] R. Cantoro, D. Piumatti, P. Bernardi, S. De Luca, and A. Sansonetti. "In-field functional test programs development flow for embedded FPUs". In: *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Sept. 2016, pp. 107–110. DOI: 10.1109/DFT.2016.7684079.

[12] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, and A. Sansonetti. "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers". In: *IEEE Transactions on Computers* 65.3 (Mar. 2016), pp. 744–754. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2498546.

[13] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson. "On the testability of IEEE 1687 networks". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. Nov. 2015, pp. 211–216. DOI: 10.1109/ATS.2015.7447934.

[14] R. Cantoro, M. Palena, P. Pasini, and M. Sonza Reorda. "Test Time Minimization in Reconfigurable Scan Networks". In: *2016 IEEE 25th Asian Test Symposium (ATS)*. Nov. 2016, pp. 119–124. DOI: 10.1109/ATS.2016.58.

[15] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson. "On the diagnostic analysis of IEEE 1687 networks". In: *2016 21th IEEE European Test Symposium (ETS)*. May 2016, pp. 1–2. DOI: 10.1109/ETS.2016.7519294.

[16] R. Cantoro, M. Sonza Reorda, A. Rohani, and H. G. Kerkhoff. "On the maximization of the sustained switching activity in a processor". In: *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*. July 2015, pp. 34–35. DOI: 10.1109/IOLTS.2015.7229826.

[17] J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero. "Observability Solutions for In-field Functional Test of Processor-based Systems. a survey and quantitative test case evaluation". In: *Microprocessors and Microsystems* 47.PB (Nov. 2016), pp. 392–403. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2016.09.002.

[18] Li Chen and S. Dey. "Software-based self-testing methodology for processor cores". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.3 (Mar. 2001), pp. 369–380. ISSN: 0278-0070. DOI: 10.1109/43.913755.

[19] Jian Shen and J. A. Abraham. "Native mode functional test generation for processors with applications to self test and design validation". In: *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*. Oct. 1998, pp. 990–999. DOI: 10.1109/TEST.1998.743296.

[20] Jian Shen and Jacob Abraham. “Synthesis of Native Mode Self-Test Programs”. In: *Journal of Electronic Testing* 13.2 (1998), pp. 137–148. ISSN: 1573-0727. DOI: 10.1023/A:1008305820979.

[21] S. M. Thatte and J. A. Abraham. “Test Generation for Microprocessors”. In: *IEEE Transactions on Computers* C-29.6 (June 1980), pp. 429–441. ISSN: 0018-9340. DOI: 10.1109/TC.1980.1675602.

[22] P. Parvathala, K. Maneparambil, and W. Lindsay. “FRITS - a microprocessor functional BIST method”. In: *Proceedings. International Test Conference*. 2002, pp. 590–598. DOI: 10.1109/TEST.2002.1041810.

[23] L. Fournier, Y. Arbetman, and M. Levinger. “Functional verification methodology for microprocessors using the Genesys test-program generator. Application to the x86 microprocessors family”. In: *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*. 1999, pp. 434–441. DOI: 10.1109/DATE.1999.761162.

[24] S. Gurumurthy, M. Pratapgarhwala, C. Gilgan, and J. Rearick. “Comparing the effectiveness of cache-resident tests against cycleaccurate deterministic functional patterns”. In: *2014 International Test Conference*. Oct. 2014, pp. 1–8. DOI: 10.1109/TEST.2014.7035348.

[25] G. Theodorou, S. Chatzopoulos, N. Kranitis, A. Paschalis, and D. Gizopoulos. “A Software-Based Self-Test methodology for on-line testing of data TLBs”. In: *2012 17th IEEE European Test Symposium (ETS)*. May 2012, pp. 1–1. DOI: 10.1109/ETS.2012.6233043.

[26] E. Sanchez and M. Sonza Reorda. “On the Functional Test of Branch Prediction Units”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.9 (Sept. 2015), pp. 1675–1688. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2356612.

[27] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. “Software-based self-testing of embedded processors”. In: *IEEE Transactions on Computers* 54.4 (Apr. 2005), pp. 461–475. ISSN: 0018-9340. DOI: 10.1109/TC.2005.68.

[28] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero. “Automatic test program generation: a case study”. In: *IEEE Design Test of Computers* 21.2 (Mar. 2004), pp. 102–109. ISSN: 0740-7475. DOI: 10.1109/MDT.2004.1277902.

[29] A. Jasnetski, R. Ubar, and A. Tsertov. “On automatic software-based self-test program generation based on high-level decision diagrams”. In: *2016 17th Latin-American Test Symposium (LATS)*. Apr. 2016, pp. 177–177. DOI: 10.1109/LATW.2016.7483357.

[30] Y. Zhang, A. Rezine, P. Eles, and Z. Peng. “Automatic Test Program Generation for Out-of-Order Superscalar Processors”. In: *2012 IEEE 21st Asian Test Symposium*. Nov. 2012, pp. 338–343. DOI: 10.1109/ATS.2012.43.

[31] L. Lingappan and N. K. Jha. “Satisfiability-Based Automatic Test Program Generation and Design for Testability for Microprocessors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.5 (May 2007), pp. 518–530. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2007.896908.

[32] A. Dalirsani, M. E. Imhof, and H. J. Wunderlich. "Structural Software-Based Self-Test of Network-on-Chip". In: *2014 IEEE 32nd VLSI Test Symposium (VTS)*. Apr. 2014, pp. 1–6. DOI: 10.1109/VTS.2014.6818754.

[33] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker. "On the automatic generation of SBST test programs for in-field test". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2015, pp. 1186–1191. DOI: 10.7873/DATE.2015.0271.

[34] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker. "A Flexible Framework for the Automatic Generation of SBST Programs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (Oct. 2016), pp. 3055–3066. DOI: 10.1109/TVLSI.2016.2538800.

[35] D. Sabena, M. Sonza Reorda, and L. Sterpone. "On the Automatic Generation of Optimized Software-Based Self-Test Programs for VLIW Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.4 (Apr. 2014), pp. 813–823. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2252636.

[36] Y. Zhang, H. Li, and X. Li. "Automatic Test Program Generation Using Executing-Trace-Based Constraint Extraction for Embedded Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.7 (July 2013), pp. 1220–1233. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2012.2208130.

[37] Li Chen and S. Dey. "Software-based diagnosis for processors". In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*. 2002, pp. 259–262. DOI: 10.1109/DAC.2002.1012632.

[38] D. Appello, P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda. "Effective Diagnostic Pattern Generation Strategy for Transition-Delay Faults in Full-Scan SOCs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.11 (Nov. 2009), pp. 1654–1659. DOI: 10.1109/TVLSI.2008.2006177.

[39] M. Schölzel, T. Koal, and H. T. Vierhaus. "Systematic generation of diagnostic software-based self-test routines for processor components". In: *2014 19th IEEE European Test Symposium (ETS)*. May 2014, pp. 1–6. DOI: 10.1109/ETS.2014.6847795.

[40] M. Schölzel, T. Koal, S. Müller, S. Scharoba, S. Röder, and H. T. Vierhaus. "A comprehensive software-based self-test and self-repair method for statically scheduled superscalar processors". In: *2016 17th Latin-American Test Symposium (LATS)*. Apr. 2016, pp. 33–38. DOI: 10.1109/LATW.2016.7483336.

[41] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker. "Effective generation and evaluation of diagnostic SBST programs". In: *2016 IEEE 34th VLSI Test Symposium (VTS)*. Apr. 2016, pp. 1–6. DOI: 10.1109/VTS.2016.7477279.

[42] A. Paschalis and D. Gizopoulos. "Effective software-based self-test strategies for on-line periodic testing of embedded processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.1 (Jan. 2005), pp. 88–99. ISSN: 0278-0070. DOI: 10.1109/TCAD.2004.839486.

[43] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda. "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices". In: *IEEE Design Test of Computers* 26.2 (Mar. 2009), pp. 52–63. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.43.

[44] P. Bernardi, L. Ciganda, M. Sonza Reorda, and S. Hamdioui. "An Efficient Method for the Test of Embedded Memory Cores during the Operational Phase". In: *2013 22nd Asian Test Symposium*. Nov. 2013, pp. 227–232. DOI: 10.1109/ATS.2013.50.

[45] M. G. Karpovsky and V. N. Yarmolik. "Transparent memory BIST". In: *Proceedings of IEEE International Workshop on Memory Technology, Design, and Test*. Aug. 1994, pp. 106–111.

[46] S. Di Carlo, P. Prinetto, and A. Savino. "Software-Based Self-Test of Set-Associative Cache Memories". In: *IEEE Transactions on Computers* 60.7 (July 2011), pp. 1030–1044. ISSN: 0018-9340. DOI: 10.1109/TC.2010.166.

[47] J. Sosnowski. "Improving Software Based Self - Testing for Cache Memories". In: *2007 2nd International Design and Test Workshop*. Dec. 2007, pp. 49–54. DOI: 10.1109/IDT.2007.4437427.

[48] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael. "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors". In: *IEEE Transactions on Computers* 65.5 (May 2016), pp. 1453–1466. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2449840.

[49] Janusz Sosnowski. "Software-based Self-testing of Microprocessors". In: *J. Syst. Archit.* 52.5 (May 2006), pp. 257–271. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2005.05.004.

[50] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan. "On-line functionally untestable fault identification in embedded processor cores". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2013, pp. 1462–1467. DOI: 10.7873/DATE.2013.298.

[51] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan. "Fault grading of software-based self-test procedures for dependable automotive applications". In: *2011 Design, Automation Test in Europe*. Mar. 2011, pp. 1–2. DOI: 10.1109/DATE.2011.5763092.

[52] Freescale Semiconductor. *e200z4 Power Architecture Core Reference Manual*. Oct. 2009. URL: http://www.nxp.com/assets/documents/data/en/reference-manuals/e200z4RM.pdf (visited on 05/21/2017).

[53] ARM Limited. *AMBA AHB Trace Macrocell (HTM)*. 2008. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0328e/DDI0328E_amba_htm_r0p4_trm.pdf (visited on 05/21/2017).

[54] P. Bernardi, M. Grosso, M. Rebaudengo, and M. Sonza Reorda. "Exploiting an I-IP for both Test and Silicon Debug of Microprocessor Cores". In: *2005 Sixth International Workshop on Microprocessor Test and Verification*. Nov. 2005, pp. 55–62. DOI: 10.1109/MTV.2005.11.

[55] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, and O. Ballan. "On-line software-based self-test of the Address Calculation Unit in RISC processors". In: *2012 17th IEEE European Test Symposium (ETS)*. May 2012, pp. 1–6. DOI: 10.1109/ETS.2012.6233004.

[56] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis. "A methodology for detecting performance faults in microprocessors via performance monitoring hardware". In: *2007 IEEE International Test Conference*. Oct. 2007, pp. 1–10. DOI: 10.1109/TEST.2007.4437646.

[57] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos. "Software-Based Self-Test for Small Caches in Microprocessors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.12 (Dec. 2014), pp. 1991–2004. ISSN: 0278-0070. DOI: 10.1109/TCAD.2014.2363387.

[58] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos. "Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.4 (Apr. 2013), pp. 786–790. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2012.2191000.

[59] W. Lindsay, E. Sanchez, M. Sonza Reorda, and G. Squillero. "Automatic test programs generation driven by internal performance counters". In: *Microprocessor Test and Verification (MTV'04), Fifth International Workshop on*. 2004, pp. 8–13. DOI: 10.1109/MTV.2004.5.

[60] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda. "Microprocessor Software-Based Self-Testing". In: *IEEE Design Test of Computers* 27.3 (May 2010), pp. 4–19. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.5.

[61] G. Squillero. "Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs". In: *Computing* 93.2 (2011), pp. 103–120. ISSN: 1436-5057. DOI: 10.1007/s00607-011-0157-9.

[62] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 9780123838728.

[63] X. Yang, L. Yu, W. Zhuang, Y. Wu, and L. Hao. "Design of instruction decode logic for dual-issue superscalar processor based on LEON2". In: *2013 IEEE Third International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*. Sept. 2013, pp. 1–4. DOI: 10.1109/ICCE-Berlin.2013.6697986.

[64] Freescale Semiconductor. *EREF 2.0 : A Programmer's Reference Manual for Freescale Power Architecture Processors*. Sept. 2011. URL: http://cache.freescale.com/files/32bit/doc/ref_manual/EREF_RM.pdf (visited on 05/21/2017).

[65] P. S. Ahuja, D. W. Clark, and A. Rogers. "The performance impact of incomplete bypassing in processor pipelines". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. Nov. 1995, pp. 36–45. DOI: 10.1109/MICRO.1995.476809.

[66] T. Y. Hsieh, M. A. Breuer, M. Annavaram, S. K. Gupta, and K. J. Lee. "Tolerance of performance degrading faults for effective yield improvement". In: *IEEE International Test Conference (ITC)*. Nov. 2009, pp. 1–10. DOI: 10.1109/TEST.2009.5355594.

[67] S. R. Makar and E. J. McCluskey. "On the testing of multiplexers". In: *IEEE International Test Conference (ITC)*. Sept. 1988, pp. 669–679. DOI: 10.1109/TEST.1988.207851.

[68] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian. "Generic BIST architecture for testing of content addressable memories". In: *2011 IEEE 17th International On-Line Testing Symposium*. July 2011, pp. 86–91. DOI: 10.1109/IOLTS.2011.5993816.

[69] *miniMIPS :: Overview*. June 2004. URL: https://opencores.org/project,minimips (visited on 04/20/2017).

[70] B. Sprunt. "The basics of performance-monitoring hardware". In: *IEEE Micro* 22.4 (July 2002), pp. 64–71. ISSN: 0272-1732. DOI: 10.1109/MM.2002.1028477.

[71] W. J. Perez, J. Velasco, D. Ravotto, E. Sanchez, and M. Sonza Reorda. "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs". In: *2008 14th IEEE International On-Line Testing Symposium*. July 2008, pp. 143–148. DOI: 10.1109/IOLTS.2008.22.

[72] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi. "Systematic Software-Based Self-Test for Pipelined Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.11 (Nov. 2008), pp. 1441–1453. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2008.2000866.

[73] R. S. Tupuri and J. A. Abraham. "A novel functional test generation method for processors using commercial ATPG". In: *IEEE International Test Conference (ITC)*. Nov. 1997, pp. 743–752. DOI: 10.1109/TEST.1997.639687.

[74] D. Sabena, M. Sonza Reorda, and L. Sterpone. "A new SBST algorithm for testing the register file of VLIW processors". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2012, pp. 412–417. DOI: 10.1109/DATE.2012.6176506.

[75] "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (1985). DOI: 10.1109/IEEESTD.1985.82928.

[76] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[77] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener. "P6 Binary Floating-Point Unit". In: *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*. June 2007, pp. 77–86. DOI: 10.1109/ARITH.2007.26.

[78] J. Rupley, J. King, E. Quinnell, F. Galloway, K. Patton, P. M. Seidel, J. Dinh, H. Bui, and A. Bhowmik. "The Floating-Point Unit of the Jaguar x86 Core". In: *2013 IEEE 21st Symposium on Computer Arithmetic*. Apr. 2013, pp. 7–16. DOI: 10.1109/ARITH.2013.24.

[79] V. Patil, A. Raveendran, P. M. Sobha, A. David Selvakumar, and D. Vivian. "Out of order floating point coprocessor for RISC V ISA". In: *2015 19th International Symposium on VLSI Design and Test*. June 2015, pp. 1–7. DOI: 10.1109/ISVDAT.2015.7208116.

[80] G. Xenoulis, M. Psarakis, D. Gizopoulos, and A. Paschalis. "Testability Analysis and Scalable Test Generation for High-Speed Floating-Point Units". In: *IEEE Transactions on Computers* 55.11 (Nov. 2006), pp. 1449–1457. ISSN: 0018-9340. DOI: 10.1109/TC.2006.187.

[81] I. Bayraktaroglu and M. d'Abreu. "ATPG based functional test for data paths: application to a floating point unit". In: *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*. Nov. 2004, pp. 37–40. DOI: 10.1109/HLDVT.2004.1431230.

[82] S. Sobek and K. Burke. *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*. Ed. by Inc. Freescale Semiconductor. 2004. URL: http://www.nxp.com/assets/documents/data/en/application-notes/PPCEABI.pdf (visited on 05/18/2017).

[83] ARM Limited. *Application Binary Interface for the ARM Architecture*. 2008. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036b/IHI0036B_bsabi.pdf (visited on 05/18/2017).

[84] E. Christopher. *mips eabi documentation...* 2003. URL: http://www.cygwin.com/ml/binutils/2003-06/msg00436.html (visited on 05/18/2017).

[85] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos. "Hybrid-SBST Methodology for Efficient Testing of Processor Cores". In: *IEEE Design Test of Computers* 25.1 (Jan. 2008), pp. 64–75. ISSN: 0740-7475. DOI: 10.1109/MDT.2008.15.

[86] P. Singh, D. L. Landis, and V. Narayanan. "Test Generation for Precise Interrupts on Out-of-Order Microprocessors". In: *2009 10th International Workshop on Microprocessor Test and Verification*. Dec. 2009, pp. 79–82. DOI: 10.1109/MTV.2009.14.

[87] AUTOSAR. *Standard Platform*. URL: https://www.autosar.org/standards/classic-platform/ (visited on 05/20/2017).

[88] *Doxygen Web-site*. URL: http://www.stack.nl/~dimitri/doxygen/ (visited on 05/20/2017).

[89] *ISO/DIS 26262 - Road vehicles — Functional safety*. Tech. rep. Geneva, Switzerland, 2009.

[90] A. Tšertov, A. Jutman, S. Devadze, M. Sonza Reorda, E. Larsson, F. G. Zadegan, R. Cantoro, M. Montazeri, and R. Krenz-Baath. "A suite of IEEE 1687 benchmark networks". In: *IEEE International Test Conference (ITC)*. Nov. 2016, pp. 1–10. DOI: 10.1109/TEST.2016.7805840.

[91] "IEEE Standard for Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (May 2013), pp. 1–444. DOI: 10.1109/IEEESTD.2013.6515989.

[92] "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device". In: *IEEE Std 1687-2014* (Dec. 2014), pp. 1–283. DOI: 10.1109/IEEESTD.2014.6974961.

[93] F. G. Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson. "Access Time Analysis for IEEE P1687". In: *IEEE Transactions on Computers* 61.10 (Oct. 2012), pp. 1459–1472. ISSN: 0018-9340. DOI: 10.1109/TC.2011.155.

[94] F. G. Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson. "Design automation for IEEE P1687". In: *2011 Design, Automation Test in Europe*. Mar. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763228.

[95] F. G. Zadegan, E. Larsson, A. Jutman, S. Devadze, and R. Krenz-Baath. "Design, Verification, and Application of IEEE 1687". In: *2014 IEEE 23rd Asian Test Symposium*. Nov. 2014, pp. 93–100. DOI: 10.1109/ATS.2014.28.

[96] F. G. Zadegan, U. Ingelsson, G. Carlsson, and E. Larsson. "Test Time Analysis for IEEE P1687". In: *2010 19th IEEE Asian Test Symposium*. Dec. 2010, pp. 455–460. DOI: 10.1109/ATS.2010.83.

[97] F. G. Zadegan, U. Ingelsson, G. Asani, G. Carlsson, and E. Larsson. "Test Scheduling in an IEEE P1687 Environment with Resource and Power Constraints". In: *2011 Asian Test Symposium*. Nov. 2011, pp. 525–531. DOI: 10.1109/ATS.2011.80.

[98] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. "Modeling, Verification and Pattern Generation for Reconfigurable Scan Networks". In: *2012 IEEE International Test Conference*. Nov. 2012, pp. 1–9. DOI: 10.1109/TEST.2012.6401555.

[99] Rafal Baranowski, Michael A. Kochte, and Hans-Joachim Wunderlich. "Reconfigurable Scan Networks: Modeling, Verification, and Optimal Pattern Generation". In: *ACM Trans. Des. Autom. Electron. Syst.* 20.2 (Mar. 2015), 30:1–30:27. ISSN: 1084-4309. DOI: 10.1145/2699863.

[100] R. Krenz-Baath, F. G. Zadegan, and E. Larsson. "Access Time Minimization in IEEE 1687 Networks". In: *2015 IEEE International Test Conference (ITC)*. Oct. 2015, pp. 1–10. DOI: 10.1109/TEST.2015.7342408.

[101] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. "Scan Pattern Retargeting and Merging with Reduced Access Time". In: *2013 18th IEEE European Test Symposium (ETS)*. May 2013, pp. 1–7. DOI: 10.1109/ETS.2013.6569354.

[102] F. Ghani Zadegan, U. Ingelsson, E. Larsson, and G. Carlsson. "Reusing and Retargeting On-Chip Instrument Access Procedures in IEEE P1687". In: *IEEE Design Test of Computers* 29.2 (Apr. 2012), pp. 79–88. ISSN: 0740-7475. DOI: 10.1109/MDT.2012.2182984.

[103] M. A. Kochte and H. J. Wunderlich. "Dependable on-chip infrastructure for dependable MPSOCs". In: *2016 17th Latin-American Test Symposium (LATS)*. Apr. 2016, pp. 183–188. DOI: 10.1109/LATW.2016.7483366.

[104] M. A. Kochte, R. Baranowski, M. Schaal, and H. J. Wunderlich. "Test Strategies for Reconfigurable Scan Networks". In: *2016 IEEE 25th Asian Test Symposium (ATS)*. Nov. 2016, pp. 113–118. DOI: 10.1109/ATS.2016.35.

[105] J. Dworak and A. Crouch. "A call to action: Securing IEEE 1687 and the need for an IEEE test Security Standard". In: *2015 IEEE 33rd VLSI Test Symposium (VTS)*. Apr. 2015, pp. 1–4. DOI: 10.1109/VTS.2015.7116256.

[106] C. Clark. "Anti-tamper JTAG TAP design enables DRM to JTAG registers and P1687 on-chip instruments". In: *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. June 2010, pp. 19–24. DOI: 10.1109/HST.2010.5513119.

[107] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. "Securing Access to Reconfigurable Scan Networks". In: *2013 22nd Asian Test Symposium*. Nov. 2013, pp. 295–300. DOI: 10.1109/ATS.2013.61.

[108] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. "Access Port Protection for Reconfigurable Scan Networks". In: *Journal of Electronic Testing* 30.6 (2014), pp. 711–723. ISSN: 1573-0727. DOI: 10.1007/s10836-014-5484-2.

[109] H. Liu and V. D. Agrawal. "Securing IEEE 1687-2014 Standard Instrumentation Access by LFSR Key". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. Nov. 2015, pp. 91–96. DOI: 10.1109/ATS.2015.23.

[110] A. Zygmontowicz, J. Dworak, A. Crouch, and J. Potter. "Making it harder to unlock an LSIB: Honeytraps and misdirection in a P1687 network". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.208.

[111] J. Dworak, A. Crouch, J. Potter, A. Zygmontowicz, and M. Thornton. "Don't forget to lock your SIB: hiding instruments using P1687". In: *2013 IEEE International Test Conference (ITC)*. Sept. 2013, pp. 1–10. DOI: 10.1109/TEST.2013.6651903.

[112] J. Dworak, Z. Conroy, A. Crouch, and J. Potter. "Board security enhancement using new locking SIB-based architectures". In: *2014 International Test Conference*. Oct. 2014, pp. 1–10. DOI: 10.1109/TEST.2014.7035355.

[113] R. Baranowski, M. A. Kochte, and H. J. Wunderlich. "Fine-Grained Access Management in Reconfigurable Scan Networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.6 (July 2015), pp. 937–946. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2391266.

[114] M. A. Kochte, R. Baranowski, M. Sauer, B. Becker, and H. J. Wunderlich. "Formal Verification of Secure Reconfigurable Scan Network Infrastructure". In: *2016 21th IEEE European Test Symposium (ETS)*. May 2016, pp. 1–6. DOI: 10.1109/ETS.2016.7519290.

[115] A. Jutman, S. Devadze, and J. Aleksejev. "Invited paper: System-wide fault management based on IEEE P1687 IJTAG". In: *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. June 2011, pp. 1–4. DOI: 10.1109/ReCoSoC.2011.5981520.

[116] E. Larsson and K. Shibin. "Fault management in an IEEE P1687 (IJTAG) environment". In: *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. Apr. 2012, pp. 7–7. DOI: 10.1109/DDECS.2012.6219013.

[117] A. Jutman, S. Devadze, and K. Shibin. "Effective Scalable IEEE 1687 Instrumentation Network for Fault Management". In: *IEEE Design Test* 30.5 (Oct. 2013), pp. 26–35. ISSN: 2168-2356. DOI: 10.1109/MDAT.2013.2278535.

[118] K. Shibin, S. Devadze, and A. Jutman. "Asynchronous Fault Detection in IEEE P1687 Instrument Network". In: *2014 IEEE 23rd North Atlantic Test Workshop*. May 2014, pp. 73–78. DOI: 10.1109/NATW.2014.24.

[119] K. Petersen, D. Nikolov, U. Ingelsson, G. Carlsson, F. G. Zadegan, and E. Larsson. "Fault injection and fault handling: An MPSoC demonstrator using IEEE P1687". In: *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*. July 2014, pp. 170–175. DOI: 10.1109/IOLTS.2014.6873664.

[120] K. Shibin, S. Devadze, and A. Jutman. "On-line fault classification and handling in IEEE1687 based fault management system for complex SoCs". In: *2016 17th Latin-American Test Symposium (LATS)*. Apr. 2016, pp. 69–74. DOI: 10.1109/LATW.2016.7483342.

[121] F. G. Zadegan, D. Nikolov, and E. Larsson. "A self-reconfiguring IEEE 1687 network for fault monitoring". In: *2016 21th IEEE European Test Symposium (ETS)*. May 2016, pp. 1–6. DOI: 10.1109/ETS.2016.7519288.

[122] G. Ali, A. Badawy, and H. G. Kerkhoff. "Accessing on-chip temperature health monitors using the IEEE 1687 standard". In: *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. Dec. 2016, pp. 776–779. DOI: 10.1109/ICECS.2016.7841317.

[123] A. Ibrahim and H. G. Kerkhoff. "Analysis and design of an on-chip retargeting engine for IEEE 1687 networks". In: *2016 21th IEEE European Test Symposium (ETS)*. May 2016, pp. 1–6. DOI: 10.1109/ETS.2016.7519301.

[124] M. Portolan. "A novel test generation and application flow for functional access to IEEE 1687 instruments". In: *2016 21th IEEE European Test Symposium (ETS)*. May 2016, pp. 1–6. DOI: 10.1109/ETS.2016.7519302.

[125] A. Jutman, K. Shibin, and S. Devadze. "Reliable health monitoring and fault management infrastructure based on embedded instrumentation and IEEE 1687". In: *2016 IEEE AUTOTESTCON*. Sept. 2016, pp. 1–10. DOI: 10.1109/AUTEST.2016.7589605.

[126] J. C. Ye, M. A. Kochte, K. J. Lee, and H. J. Wunderlich. "Autonomous Testing for 3D-ICs with IEEE Std. 1687". In: *2016 IEEE 25th Asian Test Symposium (ATS)*. Nov. 2016, pp. 215–220. DOI: 10.1109/ATS.2016.56.

[127] E. J. Marinissen, V. Iyengar, and K. Chakrabarty. "A set of benchmarks for modular testing of SOCs". In: *Proceedings. International Test Conference*. 2002, pp. 519–528. DOI: 10.1109/TEST.2002.1041802.

[128] *FP7 Project BASTION Web-Site*. 2014. URL: http://fp7-bastion.eu/ (visited on 04/27/2017).

[129] K. J. Lee and M. A. Breuer. "A Universal Test Sequence for CMOS Scan Registers". In: *IEEE Proceedings of the Custom Integrated Circuits Conference*. May 1990, pp. 28.5/1–28.5/4. DOI: 10.1109/CICC.1990.124822.

[130] R. Guo, L. Lai, H. Yu, and W. T. Cheng. "Detection and Diagnosis of Static Scan Cell Internal Defect". In: *2008 IEEE International Test Conference*. Oct. 2008, pp. 1–10. DOI: 10.1109/TEST.2008.4700596.

[131] M. K. Reddy and S. M. Reddy. "Detecting FET Stuck-Open Faults in CMOS Latches And Flip-Flops". In: *IEEE Design Test of Computers* 3.5 (Oct. 1986), pp. 17–26. ISSN: 0740-7475. DOI: 10.1109/MDT.1986.295040.

[132] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz. "Detection of Internal Stuck-Open Faults in Scan Chains". In: *2008 IEEE International Test Conference*. Oct. 2008, pp. 1–10. DOI: 10.1109/TEST.2008.4700577.

[133] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz. "Detectability of Internal Bridging Faults in Scan Chains". In: *2009 Asia and South Pacific Design Automation Conference*. Jan. 2009, pp. 678–683. DOI: 10.1109/ASPDAC.2009.4796558.

[134] D. Adolfsson, J. Siew, E. J. Marinissen, and E. Larsson. "On Scan Chain Diagnosis for Intermittent Faults". In: *2009 Asian Test Symposium*. Nov. 2009, pp. 47–54. DOI: 10.1109/ATS.2009.74.

[135] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[136] *MicroGP Web-Site*. 2002. URL: http://ugp3.sourceforge.net/ (visited on 04/27/2017).

[137] S. Kundu. "On diagnosis of faults in a scan-chain". In: *Digest of Papers Eleventh Annual 1993 IEEE VLSI Test Symposium*. Apr. 1993, pp. 303–308. DOI: 10.1109/VTEST.1993.313363.

[138] Y. Huang, R. Guo, W. T. Cheng, and J. C. M. Li. "Survey of Scan Chain Diagnosis". In: *IEEE Design Test of Computers* 25.3 (May 2008), pp. 240–248. ISSN: 0740-7475. DOI: 10.1109/MDT.2008.83.

[139] Y. Huang, X. Fan, H. Tang, M. Sharma, W. T. Cheng, B. Benware, and S. M. Reddy. "Distributed dynamic partitioning based diagnosis of scan chain". In: *2013 IEEE 31st VLSI Test Symposium (VTS)*. Apr. 2013, pp. 1–6. DOI: 10.1109/VTS.2013.6548916.

[140] W. H. Lo, A. C. Hsieh, C. M. Lan, M. H. Lin, and T. Hwang. "Utilizing Circuit Structure for Scan Chain Diagnosis". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.12 (Dec. 2014), pp. 2766–2778. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2294712.

[141] H. Chen, Z. Qi, L. Wang, and C. Xu. "A scan chain optimization method for diagnosis". In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. Oct. 2015, pp. 613–620. DOI: 10.1109/ICCD.2015.7357172.

[142] S. Kundu, S. Chattopadhyay, I. Sengupta, and R. Kapur. "Scan Chain Masking for Diagnosis of Multiple Chain Failures in a Space Compaction Environment". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.7 (July 2015), pp. 1185–1195. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2333691.

[143] J. Ye, Y. Huang, Y. Hu, W. T. Cheng, R. Guo, L. Lai, T. P. Tai, X. Li, W. Changchien, D. M. Lee, J. J. Chen, S. C. Eruvathi, K. K. Kumara, C. Liu, and S. Pan. "Diagnosis and Layout Aware (DLA) Scan Chain Stitching". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.3 (Mar. 2015), pp. 466–479. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2313563.

[144] R. Cantoro, M. Montazeri, M. Sonza Reorda, F. G. Zadegan, and E. Larsson. "Automatic generation of stimuli for fault diagnosis in IEEE 1687 networks". In: *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. July 2016, pp. 167–172. DOI: 10.1109/IOLTS.2016.7604692.