TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
27/2020

High-Level Implementation-Independent Software-Based Self-Test for RISC Type Microprocessors

ADEBOYE STEPHEN OYENIRAN

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on 1 July 2020

Supervisor: Professor Raimund-Johannes Ubar,
Department of Computer Systems,
School of Information Technologies,
Tallinn University of Technology
Tallinn, Estonia

Opponents: Professor Matteo Sonza Reorda,
Politecnico di Torino,
Torino, Italy

Professor H.G Kerkhoff,
University of Twente,
Enschede, Netherlands

Defence of the thesis: 24 August 2020, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Adeboye Stephen Oyeniran _____
signature

European Union
European Regional
Development Fund

Investing
in your future

ISSN 2585–6898 (publication)
ISBN 978-9949-83-588-1 (publication)
ISSN 2585–6901 (PDF)
ISBN 978-9949-83-589-8 (PDF)
Printed by Auratrükk

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
27/2020

Mikroprotsessorite tarkvarapõhine implementatsioonist mittesõltuv funktsionaalne enesekontroll

ADEBOYE STEPHEN OYENIRAN

TAL TECH
KIRJASTUS

Contents

List of Publications 7

Author's Contributions to the Publications 9

Abbreviations 10

1 Introduction 11
- 1.1 Motivation 12
- 1.2 Problem Formulation 13
- 1.3 Contributions 13
- 1.4 Thesis Organization 14

2 Overview 15
- 2.1 Overview of high-level fault modeling of digital systems 15
- 2.2 Behavioral level fault modeling for microprocessors 17
- 2.3 Overview of Software-Based Self-Test (SBST) methods for microprocessor 18
- 2.4 Summary 22

3 High-level Decision Diagrams(HLDDs) 23
- 3.1 HLDDs as a new model for diagnostic modeling of digital circuits 23
- 3.2 Modeling microprocessors with HLDDs 23
- 3.3 Fault modeling in microprocessors using HLDDs 27
- 3.4 Minimization of Number of Edges in HLDDs 30
- 3.5 Optimization of the HLDD Model 34
- 3.6 Summary 36

4 High-level functional test generation for the control parts of modules 38
- 4.1 Implementation-independent test concept for testing control faults 38
- 4.2 High-level functional control fault model 40
- 4.3 Mapping of high-level faults to gate level 41
- 4.4 Generation of test data for testing the control faults 43
- 4.5 High-level fault simulation and fault coverage 44
- 4.6 Extension of the Fault Class Beyond SAF 45
- 4.7 Identification of redundant faults in microprocessors 49
- 4.8 Case Study of the Control Part Test for Forwarding Unit of MIPS-like RISC processor 50
- 4.9 Summary 52

5 Pseudo-exhaustive testing of data-parts of modules 53
- 5.1 Overview of the pseudo-exhaustive test concept 53
- 5.2 Testing of multipliers with pseudo-exhaustive test patterns 54
- 5.3 Combining pseudo-exhaustive test with deterministic test data 56
- 5.4 Summary 57

6 Software-based test program generation for microprocessors 58
- 6.1 Environment for the SBST synthesis (The flow of tasks) 58
- 6.2 Test templates and the concepts of conformity and scanning tests 61
- 6.3 Organization of the full test program 64
- 6.4 Multiple fault detection in microprocessors 65

6.5 Introducing the result of the thesis into engineering education 70
6.6 Summary 71

7 Experimental Results 72

8 Conclusions 77

List of Figures 80

List of Tables 81

References 82

Acknowledgements 93

Abstract 94

Kokkuvõte 96

Appendix 1 99

Appendix 2 107

Appendix 3 113

Appendix 4 121

Appendix 5 131

Appendix 6 139

Appendix 7 147

Appendix 8 155

Appendix 9 163

Curriculum Vitae 193

Elulookirjeldus 195

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I R. Ubar, S. A. Oyeniran, M. Scholzel, and H. T. Vierhaus, "Multiple fault testing in systems-on-chip with high-level decision diagrams," *2015 10th International Design Test Symposium (IDT)*, pp. 66–71, Dec 2015

II A. S. Oyeniran, U. E. Odozi, and R. Ubar, "A new measure for calculating multiple fault coverage of microprocessor self-test," in *2016 15th Biennial Baltic Electronics Conference (BEC)*, pp. 75–78, Oct 2016

III A. S. Oyeniran, A. Jasnetski, A. Tsertov, and R. Ubar, "High-level test data generation for software-based self-test in microprocessors," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, June 2017

IV A. S. Oyeniran, R. Ubar, S. P. Azad, and J. Raik, "High-level test generation for processing elements in many-core systems," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, July 2017

V A. S. Oyeniran and R. Ubar, "High-level functional test generation for microprocessor modules," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pp. 356–361, June 2019

VI A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "High-level combined deterministic and pseudo-exhuastive test generation for risc processors," in *2019 IEEE European Test Symposium (ETS)*, pp. 1–6, May 2019

VII A. S. Oyeniran, S. P. Azad, and R. Ubar, "Parallel pseudo-exhaustive testing of array multipliers with data-controlled segmentation," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018

VIII A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "Mixed-level identification of fault redundancy in microprocessors," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, March 2019

IX A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors," *Journal of Electronic Testing*, vol. 36, no. 1, pp. 87–103, 2020

Other related publications

X A. S. Oyeniran, S. P. Azad, and R. Ubar, "Combined pseudo-exhaustive and deterministic testing of array multipliers," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, 2018

XI A. Jasnetski, S. A. Oyeniran, A. Tsertov, M. Schölzel, and R. Ubar, "High-level modeling and testing of multiple control faults in digital systems," in *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–6, 2016

XII S. Payandeh Azad, A. S. Oyeniran, and R. Ubar, "Replication-based deterministic testing of 2-dimensional arrays with highly interrelated cells," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 21–26, 2018

XIII A. S. Oyeniran, R. Ubar, and M. Kruus, "Teaching digital system test," in *2017 27th EAEEIE Annual Conference (EAEEIE)*, pp. 1–6, 2017

XIV L. Jürimägi, R. Ubar, M. Jenihhin, J. Raik, S. Devadze, and A. S. Oyeniran, "Application specific true critical paths identification in sequential circuits," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 299–304, 2019

XV A. S. Oyeniran and R. Ubar, "High-level functional test generation for microprocessor modules," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pp. 356–361, 2019

XVI S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in network-on-chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, 2017

XVII R. Ubar, A. S. Oyeniran, and O. Medaiyese, "Minimization of the high-level fault model for microprocessor control parts," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, 2018

XVIII A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "Implementation-independent functional test generation for risc microprocessors," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 82–87, 2019

XIX R. Ubar, L. Jürimägi, E. Orasson, G. Josifovska, and S. A. Oyeniran, "Double phase fault collapsing with linear complexity in digital circuits," in *2015 Euromicro Conference on Digital System Design*, pp. 700–705, 2015

XX R. Ubar and S. A. Oyeniran, "Multiple control fault testing in digital systems with high-level decision diagrams," in *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, 2016

Author's Contributions to the Publications

I In Publication I, the author participated in idea formulation, describing of systems at high-level and preparing paper for presentation.

II In Publication II, I carried out the extraction of modules of processor under test, described the modules with HLDD for testing of the control parts and data parts. The author also carried out experiments, prepared the paper for publication and presented at the conference.

III In Publication III, I implemented a deterministic algorithm for generation of operands for conformity test of microprocessors, took part in several discussions with supervisor and co-authors in paper planning, experimentation and also presentation of the research result at the conference.

IV In Publication IV, I implemented together with one of the co-author, two algorithms for high-level simulation-based test generation for the control path of the case-study processor. I also participated in paper preparation and presentation at the conference.

V In Publication V, I took part in discussion leading to choosing of another microprocessor as case-study, created the HLDD model of the processor using ISA information from processor's documentation, performed experiments and presented the paper at the conference.

VI In Publication VI, I took part in the discussion on extending the test approach to execution unit of the processor. carried out experiments, prepared paper for publication and presented the paper at the conference.

VII In Publication VII, I extracted the multiplier module from the ALU of the processor for experiment. I also prepared the pseudo-exhaustive data that was used for fault detection in the Unit Under Test, carried out the experiment in partnership with one of the co-authors and presented the paper at the conference.

VIII In Publication VIII, I implemented a method for identification of the high-level redundant faults by carrying out experiments which shows that a test, which provides 100% coverage of non-redundant high-level faults, will also guarantee 100% non-redundant SAF coverage. I also contributed to proofs that showed that all gate-level SAF not covered by the test are identified as redundant and presented the paper at the conference.

IX In Publication IX, I developed an implementation independent test program that targets the full modules of the processor, described the test framework and compared experimental result with state-of-the-art methods. I also prepared paper for publication and managed correspondence with the publishers.

Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application specific integrated circuits
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generator
BBD	Binary Decision Diagram
BIST	Built-In Self-Test
CPU	Central Processing Unit
DFT	Design For Testability
DMUX	De-multiplexer
DNF	Disjunctive Normal Form
DOT	Defect-oriented test pattern generator
EDA	Electronic Design Automation
EDNF	Equivalent Disjunctive Normal Form
FRITS	Functional Random Instruction Testing at Speed
HLDD	High-Level Decision Diagrams
I/O	Input or Output
ISA	Instruction Set Architecture
MIPS	Microprocessor without Interlocked Pipelined Stages
MUT	Module Under Test
MUX	Multiplexer
PC	Program Counter
PET	Pseudo-exhaustive Test
RISC	Reduced instruction set computer
RTL	Register-transfer levels
SAF	Stuck-At Fault
SBST	Software-Based Self-Test
SPFM	Single Point Fault Metric
TDF	Transition Delay Faults
TLM	Transaction Level Modelling
UUT	Unit Under Test
NDA	Non-Disclosure Agreement
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very-large-scale integration

1 Introduction

Scaling technology in today's deep-submicron processes produces new failure mechanisms in electronic devices, causing researchers both to create more sophisticated fault models compared to the traditional stuck-at-fault (SAF) model [21] and to explore the possibilities of reasoning the system's faulty behavior using no specific fault models [22, 23].

The traditional solution for solving testing problems for VLSI designs has been to apply Design For Testability (DFT) methods or Built-In Self-Test (BIST) [24]. An example of DFT is the insertion of scan-chains into the design. DFT techniques like scan-chains are an inevitable part of processor testing. However, it requires expensive external test equipment. In BIST, the tasks of test pattern generation and response evaluation are moved from external ATE to processor embedded logic. This facilitates achieving high-level test quality (including testing of dynamic defects and delay faults), it leads to testing cost reduction as well. However, the BIST related testing approaches for microprocessors are found not as feasible as it is for memories or in application-specific integrated circuits (ASICs) [25]. Furthermore, BIST results often in over-testing and over-stressing the circuit because of higher than normal switching activity during the test.

The semiconductor industry has been challenged over the past decade with developing alternative test methods that can be integrated into an existing test flow of microprocessors. As a result, SBST was introduced as an alternative method to hardware-based self-test [25–29]. This approach eliminates the need for expensive external testing equipment, and the testing time is influenced by the processor's performance. The principal subject of the SBST approach is the development of the test program, which must comply with the industry's high-quality standards for fault coverage.

The general idea of Software-Based Self-Test (SBST) is to use the resources of processors to test themselves by running specific test programs. The nature of this method implies such features as non-intrusiveness, low cost and compatibility with at-speed and in-field testing [30, 31]. SBST approach is based on software programs designed to test the functionality of the processor cores and this method is well accepted in the industry. The efficiency of test program generation (quality and speed) highly depends on the abstraction level of representing the system and on the adequacy of fault models. Because of the increasing complexity of digital systems such as microprocessors, the gate-level approaches are time-consuming and high-level approaches have become more attractive.

The lack of efficient formal methods has made self-test programs to be written manually for microprocessors. High-level fault modeling approaches and formal test generation strategies have not been investigated thoroughly enough to support the automated synthesis of self-test programs and to offer fast methods for test quality evaluation.

In this thesis, a novel formal approach for modeling the high-level functionality and possible faulty behaviors of microprocessors is elaborated. The state-of-the-art High-Level Decision Diagrams (HLDD) is adopted as the main modeling framework for microprocessors, which can be considered as a logic level generalization of binary decision diagrams (BDD).

1.1 Motivation

Due the reduction in the dimension of today's transistors, it is possible to integrate more transistors on chips. This is the trend in technology and is referred to as technology shrinking or scaling. This advance in technology makes it possible for microprocessors to be built from billions of transistors and operate at GHz frequencies. However, the probability of different physical defects is also increasing, and the growing complexity of systems makes testing problems extremely difficult to solve. Due to the huge number of possible low-level defect types and complex physical mechanisms in microprocessor systems, high-level fault modeling has become a hot topic in the field of test generation for microprocessors.

Over the years, various designs for testability techniques (such as scan-chain insertion, BIST) have been developed for testing digital systems despite the complexities, and this technique is a common solution during manufacturing testing. However, these techniques come at a cost of hardware overhead and power requirements, which cannot be overlooked in designs like processors. Besides this, recent standards (e.g. ISO 26262 and ASIL standards) make it a necessity to have strategies for on-line testing [32]. According to these standards, there should be a constant observation of a system throughout its life-time for possible faults.

As noted by [30],the test technology challenges prompted the semiconductor industry during the past decade to consider alternative testing methods that can be incorporated in an established microprocessor test flow. This can be seen by the increasing efforts of processor manufacturers in providing SBST libraries that can check whether a fault is affecting the processor cores during normal operation or not.

Despite advances in technology which makes testing an arduous task, new standards in the industry place even more stringent constraints or requirements for fault coverage as previously mentioned. For example, very critical environments in the automotive system such as the airbag require by ISO26262 standard a fault coverage of 99% when considering Single Point Fault Metric (SPFM).

Over the past few years, academia has renewed its interest in software-based self-testing of embedded devices for in-field applications. ISO26262 describes the requirements for online processor core testing in automotive technology. However, additional demand in SBST has increased following the release of IEC 61508 for industrial safety systems, ISO 26262 for automotive applications and DO0254 for safety-critical applications and processor-centric systems. The practical SBST-based solution is exclusive to in-system or in-field testing since commercial products' structural information is intellectual property kept under NDA. Nonetheless, there is significant interest in the automation of SBST approach as the complexity of manual test program generation can be inexcusably high. Automated SBST [33–35] could reduce the cost of test creation and, ultimately, a product's price.

There are still several drawbacks that leave SBST as a complementary test method when compared with established structural and functional testing. One of these drawbacks is that, relative to the scan-based test, SBST is more difficult to automate. Furthermore, there are industrial EDA tools available, which can produce structural tests and achieve high fault coverage compared to SBST. Likewise, the functional test also has good fault coverage and covers defects that have not been found in the structural test. Therefore, this thesis aims at proposing a novel implementation-independent high-level SBST method, which addresses some drawbacks of existing SBST solutions.

1.2 Problem Formulation

The major problem with Software-Based Self-Test (SBST) is related to fault coverage. To improve fault coverage, several methods have been implemented in state-of-the-arts, however, there are drawbacks to these known methods. One of these drawbacks is the need to know the implementation details of the processor to be tested. The second drawback is that fault coverage has often been measured traditionally only concerning single stuck-at faults (SAF). Broader classes of faults are not considered and there has been no attempt to evaluate test coverage regarding multiple faults.

High-level fault models are broadly used in the field of Software-Based Self-Test(SBST) [31, 36–38], however, the main and general problem of high-level faults is the difficulty of proving that the model covers all low non-redundant (detectable) faults. If there would be such a high-level proof, it would be possible to identify the redundancy of gate-level faults exclusively by gate-level fault simulation which is cheaper in terms of cost compared to low-level fault redundancy proof by conventional gate-level ATPGs.

Lastly, according to [39], SBST programs targeting only functional components of the processor suffers from sufficiently testing critical units like the forwarding blocks of a pipelined processor, resulting in low coverage. This implies that despite several advantages of SBST as highlighted later in the thesis, there is a need for an efficient SBST program with methods for testing performance-enhancing and non-functional modules of the processor.

Considering the above problems, the main goal of this thesis is based on addressing them in the following ways:

- Proposing a novel deterministic high-level test generation method for SBST of embedded processors which is based on a novel implementation-free high-level functional fault model. The idea is to represent the information given in the instruction set in the form of High-Level Decision-Diagrams (HLDD).
- Providing a proof that the test based on high-level data constraint-based functional control fault model which produces 100% high-level fault coverage will also guarantee 100% low-level detectable SAF coverage, and that all not detected SAF identified by low-level fault simulation, are redundant.
- Extending the high-level constraint-based functional model to targeting faults in the non-functional units of the processor.
- Automation of implementation-independent high-level SBST generation. I divide this into two parts: Automation of high-level test data generation and automation of SBST test program generation.

1.3 Contributions

This thesis aims at the development of a novel high-level approach for the implementation-independent generation of functional software-based self-test programs for processors with RISC architecture. The approach enables the fast generation of manufacturing tests with high stuck-at fault coverage. The following are the key contributions of the work:

- Development of automatic High-level data generation algorithms and tools for generating quality test data for the control part of functional modules in the processor.
- Development of a pseudo-exhaustive method for generating test data, for testing the data part of the microprocessor.

- Development of a novel method of mixed-level identification of redundant faults.
- Development of a high-level implementation-independent Software-based self-test program generation which targets both functional and non-functional modules of microprocessors.
- Creating a generalization of the logic level test group approach for identifying fault-free sub-circuits in digital systems represented at higher register-transfer levels (RTL) or functional levels using High-Level Decision Diagrams (HLDD).

1.4 Thesis Organization

The thesis is organized as follows; Chapter 2 gives an overview of the state-of-the-art.

In Chapter 3, a novel High-Level Decision Diagram was proposed as an extension of Binary Decision Diagrams(BDDs). The HLDD allows covering of common non-formalized high-level fault types in a formalized form and can be considered as an efficient formal model of self-test generation for microprocessors. The HLDD model allows for straightforward partitioning of faults in the modules of microprocessors into two groups of high-level faults: control parts and data parts. An optimization method for minimizing the size of the HLDD model of a microprocessor which helps in the reduction of the complexity of the test generation process is also presented in this chapter.

In chapter 4, High-level test generation for the control part of processor modules was presented. The proposed high-level implementation-independent test generation concept for microprocessors is based on the partitioning of the modules into control and data parts. Also, foundations are being developed for the development of a new paradigm of a high-level functional fault model in the control part of microprocessor modules. This novel fault model is used for generating implementation-independent tests, using only high-level information about the functionality of the test module.

Pseudo-exhaustive testing of the data-part was explored in chapter 5. This guarantees high coverage of a large class of faults which includes stuck-at-faults(SAF), shorts, conditional SAF and multiple SAF. As a case study, a novel repetition-based pseudo-exhaustive test generation method was developed for different classes of multipliers. A novel mixed level method for identification of low-level redundant faults was developed and is based on only simulating the high-level test for the implementation of the given circuits.

Chapter 6 describes the software-based test program generation approach for microprocessors. A commercial/in-house tool-based SBST synthesis environment for carrying out the experiments with all methods and algorithms developed in this thesis is presented. The SBST concept developed in this thesis represents a novel method of test compaction which is to be unrolled during the execution of the test. Furthermore, the SBST concept is based on the novel architecture of the compacted test represented as a structure composed of the sets of test templates, instructions and test data.

Experimental results were presented in chapter 7 and chapter 8 concludes the thesis.

2 Overview

This chapter summarizes the state-of-the-art which will include an overview of high-level modeling of digital systems, behavioural level fault modeling of microprocessors and an overview of software-based self-test for microprocessors.

2.1 Overview of high-level fault modeling of digital systems

Fault modeling is the central problem between test generation and fault simulation. Although there are similarities, test generation and fault simulation differ in complexity. Fault simulation has linear complexity, and hence, is not so much sensitive to the size of fault lists to be simulated while test generation needs high-level fault modeling approaches because of its high complexity.

There are two important but opposing criteria which should be followed in developing tools for synthesis of tests: efficiency (the cost of test generation) and the quality of generated tests (fault coverage). Both criteria depend on which fault models are used in either test generation or fault simulation for test quality assessment.

For test quality analysis, we need fault models which reflect the physical mechanisms of real defects as accurately as possible. In test generation, where the faults are the measurable targets to be achieved, two expectations should be satisfied: first, the number of faults to be covered should be as low as possible, and second, the generated tests should at the same time achieve as high quality as possible in terms of physical defect coverage. The only way to satisfy these opposite requirements of fault modeling is a multi-level approach. A low-level fault modeling and fault simulation is needed to cope with the need for accuracy in test quality assessment, while a high-level fault model should be used to cope with the complexity of test generation.

Digital circuit fault models have been developed for various types of fault mechanisms, such as signal line bridges [40], transistor stuck-opens [41] or faults due to increased circuit delays [42]. Another trend to develop general fault modeling frameworks and test methods that can evaluate specific types of faults effectively emerged, and the oldest example is the D-calculus [43]. This method has been extended in the input pattern fault model [44] and in the pattern fault model [45] which can reflect any arbitrary change in a circuit block's logic function where a block is specified as any standard gate, complex gate or combinational sub-circuit represented at any level of the abstraction of the design.

For the module level fault diagnosis in combinational circuits, a similar pattern-related modeling method called functional fault model was proposed earlier in [46]. The functional (*or pattern*) fault model allows the grouping of an arbitrary set of signal lines into activation conditions for a single fault location, enabling the simulation of a variety of physical defect types. Based on the functional fault model, a deterministic defect-oriented test pattern generator (DOT) was developed in [47] which resulted in proof of the logic redundancy of not detected physical defects.

In [48] and [49], a template called *conditional faults* was proposed for test generation and diagnostic purposes, respectively. A conditional failure enables the combination of additional signal line objectives with the detection criteria of a particular fault. In [50], a pattern-oriented gate-exhaustive fault model was proposed for the complete exercise of blocks in gate-level combination circuits, which was extended by *region-exhaustive* fault model in [51] to target broader regions (gate collections). The functional, conditional and pattern failure models mentioned offer high flexibility in the modeling of defects beyond the single SAF model. Further developments in low-level fault modeling are

made by implementing the fault tuple fault model [52], the realistic sequential cell fault model [53], or the cell-internal fault model [54], where the last two cases have general ability to handle sequential misbehaviour of circuits.

The conditional SAF model and the other listed models [44–54] support hierarchical test approach where the test pattern (or sequence) that causes a low-level fault (e.g. physical defect) at the lower level can be considered a high-level condition (or constraint) for a higher-level functional fault.

High-level faults represent the effects of physical defects on the operation of a system described on a higher functional or behavioral level. A high-level fault model can be considered a suitable model if the tests generated using this model provide high coverage of SAF or physical defects.

In the abstraction of design, higher-level descriptions have fewer details of implementation, but more explicit functional information than descriptions at lower levels. The various levels of abstraction include behaviour (architecture), register-transfer, logical (gate), and physical (transistor) levels [21]. High-level fault models depend on the level the tests are being generated. Typically, the high-level test generation techniques are divided into structural RTL description-based methods [55, 56] which are distinguished by a more comprehensive fault model and behavioral test generation [57–60], oriented to the analysis of algorithmic descriptions.

In [61], the behavioral fault model is defined as perturbing the constructs of the language, in which the high-level description is presented. For example, the key VHDL constructs supported for behavioral modeling are: if statements, case statements, loop statements; constants, variables and signals, and pre-defined VHDL operators. Behavioral level fault models are related to these VHDL constructs. The problem with behavioral fault models is in a poor link to the hardware. In [61], the functional fault model is defined based on the high-level hardware network, where for each module, a subset of input patterns which provide complete coverage of lower-level faults (gates, transistors) over a broad range of implementations is defined. In [62], faults are also classified into functional and behavioral models. The functional model is used in the top-down design flow, where the system architecture is first described at the functional level, and the system is partitioned into several functional blocks. The behavioral fault model, on the contrary, is used where the system is described at lower levels in more detail, to cover electrical effects (non-linearity, coupling effects, impedance etc.). The behavior level allows fitting blocks interfaces by describing the system with more accuracy.

The fundamental concept of high-level modeling(behavioural or functional) is to extract the high-level description of the system in a formal model and to get various incorrect versions of the system by adding faults into the model. According to [63], this approach is called model perturbation. The models can be perturbed in certain ways: by truth-table modification, micro-operation modification, etc. This idea is implemented in different high-level fault models for different classes of digital systems. For example, in the case of microprocessors, it is implemented in a more dedicated way [64, 65]. In this case, individual functional fault models and corresponding test strategies have been developed for different function classes, such as register decoding, instruction decoding, control, data storage, data transfer and data manipulation. For systems represented in register-transfer languages, it is implemented in a more general way [55, 66, 67], or in a special way for systems described in hardware descriptive languages like VHDL [68–70].

A high-level fault model can be explicit or implicit [71]. An explicit model identifies each fault individually, and every fault in this model will be a target for test generation.

An implicit model identifies classes of faults with similar properties so that all faults in the same class can be detected by similar procedures. The advantage of an implicit fault model is that it does not require explicit enumeration of faults within a class.

Summing up the discussion on high-level fault models, the following classification of fault models is defined and employed in this thesis:

- Behavioral level fault model, given using a hardware description level language, such as VHDL, Verilog or Instruction Set Architecture (ISA) languages;
- Functional level fault models, given at the register transfer level (RTL) and applied for functional components, such as registers, adders, multipliers, and interconnect structures like multiplexers and buses;
- Structural level fault models, given at the logic level, and applied for gates, flip-flops, and interconnection between them
- Physical defect level models, given at the transistor level.

The idea of developing high-level fault models is to depict realistic physical faults in the test circuits efficiently. These faults can be used as targets in high-level test generators or fault simulators, to access the efficiency of low-level (gate or transistor) fault detection. High-level fault models can also be used to test the HDL requirements and validate the design functionality before implementation.

2.2 Behavioral level fault modeling for microprocessors

High-level approaches to fault modeling for test generation and fault simulation in digital systems can be grouped in two different classes:

- High-level fault modeling for structural RTL descriptions [56, 72, 73], which is characterized by a certain relationship between language constructs and the network structure.
- Behavioral level fault modeling [60, 68, 70, 74], which is oriented to the analysis of only algorithmic descriptions.

High-level fault modeling of microprocessors that only uses the information about Instruction Set Architectures (the instruction lists) can also be considered as behavioral approaches.

High-level approaches to fault modeling based on behavioral VHDL descriptions were discussed in [60, 68, 75], and a VHDL error simulator was developed in [68] for the analysis of functional VHDL descriptions. For this simulator, there are two types of errors that can be inserted: bit failures (SAF 0 and 1) at variables, signals or ports, and condition failures that are stuck-to true or stuck-to false. The errors are closely related to RT-level stuck-at faults, but errors in RTL components (VHDL operators) are omitted. High-level tests can be generated using a more detailed high-level fault class for behavioral sequential models; targeting bit coverage, statement coverage, branch coverage, condition coverage and partial path coverage [75]. The test generation itself, however, could be conducted at a low logic level using BDDs, derived from the description of the high-level VHDL as in the case of [75].

High-level microprocessor fault models were usually derived from instruction sets' high-level behavioral descriptions. This development has already had a long history. In [76], an approach was proposed for generating tests using behavioral faults described by disturbing language constructs in HDL descriptions. They defined stuck-at faults (at

inputs, outputs, and state variables), control faults and general function faults in this approach.

In [64], the most comprehensive and mostly cited behavioral fault model was defined directly for microprocessors using unique characteristics of microprocessor models. Faults were defined for decoding functions, control functions, and data functions. This approach is suitable for microprocessors, but cannot be directly applied to general behavioral models. In [72], an approach was proposed for generating tests using Register Transfer Level (RTL) faults defined by perturbing language constructs in RTL structures. This approach is more general than in [64], but a drawback of this model was found in generating tests using ill-structured control constructs of RTLs [60]. In [70], behavioral fault models were proposed based on a subset of the language constructs of C programming language. Faults are defined for variables and control constructs such as for, switch, if, while, assignment, and wait for.

A behavioral test generation algorithm (called B-algorithm) is presented in [60]. It generates tests directly from behavioral VHDL descriptions using three types of behavioral faults (behavioral SAF, behavioral stuck-open and micro-operation faults). Behavioral faults are defined by perturbing VHDL constructs, and it is a single fault model (assuming that only one behavioral fault occurs at a time). The perturbation-based micro-operation fault model is rather restrictive, as only the operations carried out in the microprocessor's instruction set are involved in the perturbation procedure.

The behavioral level faults affecting the operation of a microprocessor can be divided into the following classes [64, 65]:

- addressing faults affecting register decoding.
- addressing faults affecting the instruction decoding and sequencing functions.
- faults in the data-storage function.
- faults in the data-transfer function.
- faults in the data-manipulation function.

The behavioral level fault model developed in [64, 65] for different units of the data processing and the control structures of microprocessors was discussed in more detail in [77].

One drawback of the approaches described so far is that only microprocessors are handled, and the fault classes defined can not be extended to cover the general problem of digital systems testing. Another major drawback is the formalism of the fault models. All the fault models described above require specialized and dedicated test generation procedures, making it difficult to automate the generation of test programs based on these high-level fault models.

An ideal case would be to identify a small, well-defined class of fault with only a few high-level fault models and construct well-standardized and consistent test algorithms around it. This is possible with the concept of a fault model based on the High-level Decision Diagram (HLDD). This fault model has been adopted in this study, as it is well suited to support the development of a consistent and straightforward high-level test generation and simulation of faults.

2.3 Overview of Software-Based Self-Test (SBST) methods for microprocessor

Software-Based Self-Test is an emerging paradigm in the test field that is based on software programs designed for testing the functionality of the processor cores. With

SBST, test functions are moved from external tester to on-chip resources [78]. A typical flow of an SBST is as shown in Fig.33 and summarized as: loading the test program and test data into the chip's memory or processor's memory, executing the test program and storing test response in the data section of the memory, and writing out the test response for fault analysis or grading.

This approach was first introduced by [64] about four decades ago and has gained popularity as a complementary approach to conventional test methods (e.g. scan test) for quality and less costly tests, that can be applied in the processor operational mode at no extra power and hardware cost. The following characteristics among others have contributed to the popularity and acceptance of SBST approach over the years [30, 31]:

1. SBST is non-intrusive. Inclusion of extra hardware for testing could be very expensive for microprocessors. The impact of extra hardware is not limited to cost. There is also an issue of power consumption, area, and performance overhead. With SBST, this constraint or limitation doesn't exist since it does not require extra hardware.
2. Over-testing is avoided. With SBTS, the circuit is tested under normal operation. This implies that faults which cannot be activated during the operational mode of the processor are not detected. This is known to reduce yield loss significantly.
3. SBST allows for at-speed testing. That SBST programs run at the processor's operational frequency makes it possible to cover not just the stuck-at faults, but also delay defects.
4. Applicable in-field. SBST programs loaded into the processor during manufacturing can be used in-field throughout the life of the processor. This feature has placed SBST on a good acceptability position in the industry. For example, [79] noted that car manufacturers are adopting the ISO 26262 standard, which requires that the on-line self-test technique is embraced as an essential test process in critical electronic vehicle parts to ensure high quality and mission safety throughout the product's life-cycle.

Interest in the SBST method is growing in the context of in-field test for processor-centric systems in safety-critical applications. [31, 34]. Recent standards for application, for example, ISO26262, IEC61508, D 0254, set strict minimum fault coverage requirements for embedded microprocessor circuits to ensure the robustness and operational safety of sensitive electronic systems. Therefore, to meet these requirements, more work is being placed on automated SBST for in-field testing. It is interesting to note that one advantage of automated SBST generation is reducing the cost of test development [34, 35].

In the literature, several SBST test approaches have been proposed within the last few decades [25, 27–29, 37, 80–88]. These approaches can be divided into two major categories: structural and functional. Structural approaches [28, 84, 87–89] are based on the generation of tests using lower-level processor design information (gate-level or RTL-level description), while functional approaches [37, 80, 81, 90–92] mainly use instruction set architecture (ISA) information. The structural approaches cannot be used when the structural information about the processors to be tested is not available.

Structural methods are divided into two main subcategories. Methods that follow a hierarchical approach are grouped into the first category as **hierarchical structural SBST** methods. These methods concentrate on the modules of a processor one at a time, generating stimuli for each module and then expanding the stimuli to the

level of the processor. Conversely, the methods by which the test program generation process uses structural Register Transfer Level (RTL) information and ISA information to generate instruction sequence templates for describing and propagating the faults of the module under review are called **RTL structural SBST** methods.

In [84], the first interesting work using hierarchical structural SBST was done. Here, the ATPG method was used to produce stimuli in processor core modules for detecting hard-to-detect faults. Using a bounded model checker, the generated stimuli were then filtered to fit the processor's instruction set under test. Another hierarchical approach presented in [87] is based on satisfiability-based ATPG. A framework that tests the description of the processor's micro-architecture by creating models for each processor module being evaluated was proposed, and test stimuli for each module were created and filtered by the satisfiability solver.

Constraint-based test generation is another hierarchical structural SBST method. As explained by [30], these test methods feed an ATPG tool with a processor core described at different levels of abstraction. The module under test is described at the structural level (e.g. gate level), and the rest of the processor is described at a higher level while enforcing constraints between the module under test and the rest of the processor.

Another hierarchical structural SBST method is based on learning algorithms [89]. This approach can be divided into simulation and generation phase and involves functional test generation, where simulation results are used to direct additional test generation. The simulation I/O data for the module under learning is stored during simulation of the module, after which learned model is derived for each module, and these models replace the actual modules during test pattern generation phase. The test pattern generation phase applies structural ATPG for producing the test which detects the faults in the unit.

For the RTL-based class of structural SBST, the test program generation takes advantage of RTL information and ISA description to generate instruction sequences for controlling and observing the faults in the design [30]. In [28], a component-based divide-and-conquer approach was proposed where only the ISA and RTL description of the processor was used for test generation. In [93], SAF and delay faults were targeted using constraints based on ISA information of components of the processor. [88] presented an approach that explores the development of SBST for various types of circuits based on processor architecture, register level and gate-level design information. The fundamental idea of this work is to use structural or architectural information to improve structural fault coverage.

As mentioned above, functional SBST relies only on the ISA information. They can be applied even when structural information of the unit under test is unavailable.

One of the first ISA-based methods, using pseudo-random test sequences, was proposed in [90]. They developed a framework called "Vertis" which generates test pseudo-randomly using random data. The effectiveness of the tools makes it usable during different stages of production. However, since the tool generates many instruction sequences for every instruction tested, the test program can be considerably large. Another solution, FRITS (Functional Random Instruction Testing at Speed) [80] was based on test program generation on random instruction sequences with pseudo-random data. It suits well for wafer testing due to its cache-resident nature. Alternative cache-resident method for production testing [91] using random generation mechanism proves that high-cost functional testers can be replaced by the low-cost SBST without significant loss in fault coverage. This work shows the usefulness of SBST approach in the production of industrial processors. Another method was introduced in [36]

based on the evolutionary algorithm. Evolutionary in the context of microprocessor testing means re-evaluating the test program and adding to it only the active codes. The test program comprises the most effective code snippets (in a matter of SAF coverage), characterized by constant re-assessment. Nonetheless, the approach is based on structural information.

Later research concentrates on test approaches for specific processor parts like pipeline, branch prediction mechanism [38, 94] or caches [95, 96]. In [37], a method is proposed, which can enhance the SBST program to bring more coverage to pipeline logic and also memory addressing. Another approach for testing the pipeline was made in [97]. The proposed strategy involves the activation of faults related to the data hazards and register forwarding logic in processor core and the research in [31] concentrates on the decode stage of the pipeline. The method in [98] uses the executing trace collected during executing training programs on the processor to generate test programs with gate-level constraints, while the method in [99] is based on VHDL processor models and genetic algorithms using various evolutionary strategies, and in [100], a greedy strategy in which instructions that detect newly identified faults are preserved in the evolutionary cycle to identify hard-to-test faults in a processor was described. In [101], SBST method which is usually used to self-test processors was adopted to test avionics controller circuits, and SBST strategy to test delay faults in computational blocks of an out-of-order super-scalar processor without area overhead or timing cost was described in [102]. The recent advances in reinforcement learning (RL) have lead researchers in [103] to propose an RL-based test program generation technique for transition delay fault detection of some modules of a MIPS processor.

The drawback of some of these established methods is in reliance on the implementation details of the processor being tested. If this is not available, test generation would be impossible using these methods. Secondly, the methods are based on testing the microprocessor for single-stack-at faults. Other classes of faults are not targeted.

In this thesis, a novel deterministic high-level test generation method for SBST of embedded processors is proposed, which differs from the state-of-the-art methods by using a novel implementation-free high-level functional fault model developed directly from the description of the instruction set and high-level architecture of the microprocessor. This description is converted into a form of a structure consisting of high-level functional components called modules under tests (MUT), and the MUTs are represented as High-Level Decision Diagrams (HLDD). The functions of control and data parts of the MUT are described as separate parts of the HLDD model. In contrast to the known approaches, the test data are generated separately for the control and data parts of MUT using the HLDD model. The independence of the method regarding the real implementation is achieved by using a novel high-level functional fault model for the control part, and by applying the pseudo-exhaustive test approach for each function of the data part.

2.4 Summary

In this chapter, the following issues have been outlined:

- State-of-the-art software-based self-test methods have not been sufficiently developed to achieve well-formalized high-level test data generation procedures simultaneously, to achieve high fault coverages of created test programs, for efficient identification of redundant faults and avoidance of multiple faults self-masking.
- Binary Decision Diagrams(BDDs) have been a state-of-the-art data structure in VLSI CAD for several decades. Nonetheless, in complex and well-optimized systems such as microprocessors, they can not be exploited for high-level simulation and fault reasoning.
- To reduce the diagnostic modeling and test generation complexity for microprocessors, generalization of BDDs for modeling complex systems, and their fault behavior at a higher functional level is a challenge selected to be attacked as the target of the thesis.

3 High-level Decision Diagrams(HLDDs)

For almost three(3) decades, different kinds of Decision Diagrams have been applied for design verification and testing. Reduced Ordered Binary Decision Diagrams (BDD) [104] as canonical forms of Boolean functions have their application in equivalence checking and symbolic model checking. In this thesis, I consider a decision diagram representation called High-Level Decision Diagrams(HLDDs), which is considered as a generalization of BDD.

The HLDDs can be used for representing different abstraction levels from RTL (Register-Transfer Level) to TLM (Transaction Level Modelling) and the behavioural level. It has proven to be an efficient model for simulation and diagnosis because they provide for a fast evaluation by graph traversal and easy identification of cause-effect relationships [105] [106].

In HLDDs, the terminal nodes are labelled by high-level constants (vectors), bus or register variables, or by high-level algebraic operations. In essence, the terminal nodes in HLDDs represent the data processing operations, which take place in the digital system, while the non-terminal nodes of HLDDs represent the control variables.

3.1 HLDDs as a new model for diagnostic modeling of digital circuits

Most of the high-level control faults described in the previous chapter can be covered by the addressing fault model [107]. Typical examples include: addressing a word in memory, selecting a register according to a field in the instruction word of a processor and decoding an op-code to determine the instruction to be executed. The common feature of these schemes is the use of an n-bit address (or op-code) to select one of 2^{n} possible items. Whenever an item i is to be selected, the presence of an addressing (or op-code) fault may lead to:

- selecting no item,
- selecting item j instead of i,
- selecting item j in addition to i.

In a more general and summarized case, a set of items $\{j_1, j_2, \ldots, j_k\}$ may be selected instead of, or in addition to, i.

High-level data processing faults related to data manipulation operations are usually left open to be solved on an individual basis. The most common high-level approach, which considers only stuck-at faults at inputs, is not satisfactory for quality reasons. A general implementation free solution is exhaustive (or pseudo-exhaustive) fault model, which however, may prove not to be practicable due to complexity. As a trade-off between quality and complexity, the hierarchical approach could be a solution, where fault simulation and test generation for the operational modules should proceed at a lower (closer to implementation) levels using lower-level fault models, whereas the control faults will be modelled at higher levels.

3.2 Modeling microprocessors with HLDDs

In a general case, the microprocessor as a digital system can be considered as a node network, where each node represents a module that performs either data manipulation, data transfer or data storage functions, in which case, the output of a node serves as input to other nodes. To fully grasp the concept of high-level decision diagram

and it's synthesis for microprocessors, let's assume an instruction set of a hypothetical microprocessor as shown in table 1.

Table 1 – Instruction set of a microprocessor

OP	B	Mnemonic	Semantics and RT level operations
0	0	LDA A1, A	$\mathrm{READ}: R(A1) = M(A), PC = PC + 2$
	1	STA A2, A	WRITE: M(A)=R(A 2), P C=P C+2
1	0	MOV A1,A2	TRANSFER: $R(A1) = R(A2), PC = PC + 1$
	1	CMA A1,A2	COMPLEMENT: $R(A1) = \neg R(A2), PC = PC + 1$
2	0	ADD A1,A2	ADD: R(A 1)=R(A 1)+R(A 2), P C=P C+1
	1	SUB A1,A2	SUBTRACT: R(A 1)=R(A 1)-R(A 2), P C=P C+1
3	0	JMP A	JUMP: P C=A
	1	BRA A	Conditional jump (Branch instruction): IF $C = 1$, THEN $PC = A$, ELSE $PC = PC + 2$

Each of the instructions is represented by a complex instruction variable I as a concatenation of 5 sub-variables ($I = OP.B.A1.A2.A$), where OP and B denote two fields of the operation code, A1 and A2 are register addresses, and A is the memory address. In table 1, the fourth column represents a list of all the functions executed in the system network shown in figure 1.

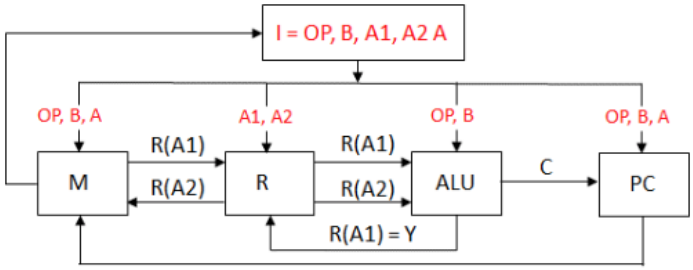


Figure 1 – Network of computing nodes for the instruction set in Table 1

This implies that it is possible to describe the functionality of the Microprocessor based on a set of control and data variables. For the control variables, we could identify OP and B, which represents two fields of the opcode. The addresses A1, A2 and A are also interpreted as control variables describing access to internal registers or memory locations represented by data variables of the set $R_{DATA} = \{\mathrm{R0}, \mathrm{R1}, \mathrm{R2}, \mathrm{R3}, \mathrm{M}\}$. The nodes or modules which represent a list of all the functions executed in the system network shown in Figure 1,are presented as follows:

- I – control module
- M – memory
- R – register block
- ALU – execution module
- PC – program counter

These modules are controlled by the control signals (shown in red color), decoded from the instruction word. The data manipulation functions executed in the modules, are represented by the 5 HLDDs in Fig. 2

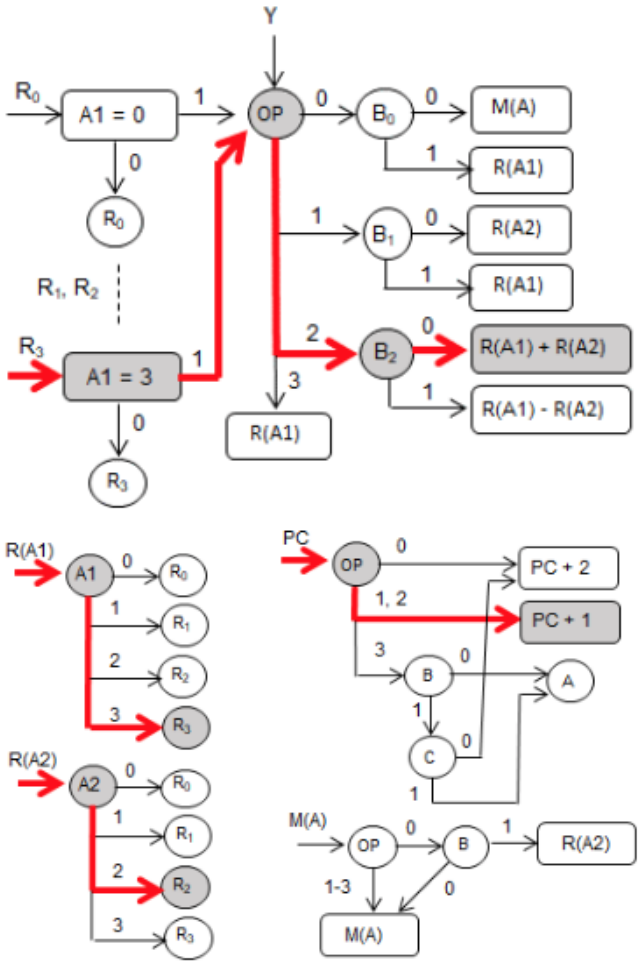


Figure 2 – HLDDs for the processor described in Table 1

The HLDDs represent a data structure used for simulation and test generation purposes. Each module contains the control and data part, which are represented by the internal (non-terminal) nodes and the terminal nodes of the HLDDs, respectively. The HLDDs have entry edges, which are labeled by the functional variable (called HLDD variable) and their values can be calculated using this graph. Simulator traverses the graph, according to the control values decoded from instruction word, and the HLDD variable will be assigned to the value calculated by the function in the terminal node where the traversing procedure is ended.

The red parts of the graphs illustrate the traversed paths for the instruction OP=2, B=0, A1=3, A2=2. The following values are calculated for this instruction on the HLDD model: R(A1) = R3, R(A2) = R2, Y = R(A1) + R(A2) and R3 = R(A1) = Y. The content of the program counter will also be updated as PC = PC + 1, with the assumption that the program counter is incremented by one for this implementation.

For this research, I consider a MIPS-like processor which is a 32-bits RISC based processor with 5 pipeline stages. The data-path and pipeline structure of the processor is shown in Fig. 3 The synthesis of HLDDs for microprocessors is not the focus of this

thesis. This topic has been discussed extensively in [108]. However, the knowledge of synthesis of HLDD is important and it's application has been directly used to model the MIPS-like processor for the purpose of simulation, test data generation and test program generation.

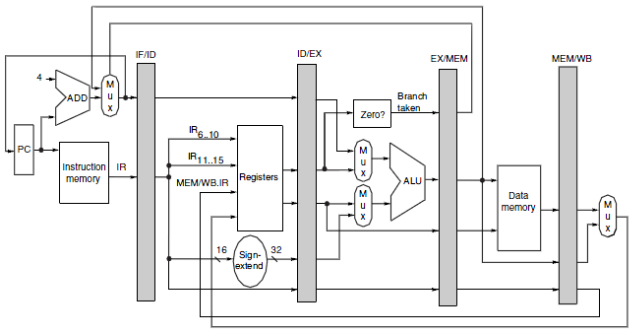


Figure 3 – MIPS-like processor data-path and pipeline

One important point to note about the choice of modeling the processor in HLDDs is that it serves as a basis for which implementation-free methods of SBST are generated. In the next section, I will describe how easy it is to model the faults in a microprocessor at high-level using HLDDs and creating a set of algorithms or methods for detecting these fault classes without relying on the implementation details of the system. The complexity of modeling a system with HLDD is only related to the size of the instruction, which is simple and easily mastered by engineers, rather than size of the system.

In generating the HLDDs for the functional modules of the processor, we rely only on the instruction set information given in the processor's manual. However, the HLDDs may also be used to present specific information that is hidden for manual users. The manual presents instruction-based information that explains what will happen when an instruction is executed, while the HLDDs present functional variable-based information that explains how each variable will behave when different instructions are executed. The instruction-based information is suitable for the microprocessor user, who is interested in exploiting the device's behavior, whereas, the variable-based information is more suitable for testing when the aim is to diagnose the faults or errors in the microprocessor's structure For modules that are not functional, we use the data-path and pipeline structure or high-level behavioral structure of the processor to create the HLDD model.

In Fig.4, a part of the pipelined structure of the microprocessor is depicted. The yellow-colored part highlights the executing unit, while the rest on the figure shows the main components of the pipeline architecture, i.e. pipeline registers,hazard detection circuitry and the forwarding unit. We consider the selected modules as consisting of disjoint control and data parts, presented as hypothetical structures without knowing their implementation details.

As previously mentioned, the HLDDs are well suited for test program planning and test data generation. Each HLLD node represents the structural unit of a microprocessor. For example, the terminal nodes labeled by variables may represent the registers or buses while the terminal nodes labeled by arithmetic or logic expressions may represent the data manipulation sub-units in the ALU. On the other hand, the non-terminal nodes of the HLDDs represent the units for interpretation of control information (decoders, multiplexers or de-multiplexers). The HLDD shown in Fig 6 represents a generalized

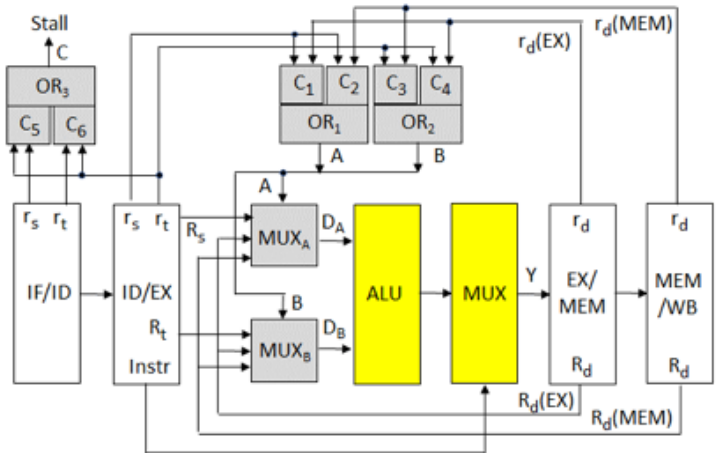


Figure 4 – A part of a RISC type microprocessor with executing unit in the pipeline and data forwarding environment

form of the ALU module of the MIPS-like processor in which case, the non-terminal decision nodes represent the control part while the terminal nodes represent the data path. As will be seen, we not only can model the ALU of the processor, the register decoders, pipeline forwarding unit can also be modelled. The graphs GR0, GR1 and GR2 in Fig. 5 show the general-purpose register decoders for the source and destination registers. Note that there are also special-purpose registers in the processor such as the HI-LO register which stores the results of operations like MULT, MTLO etc. The decoders for these special-purpose registers can also be similarly modeled by HLDD as the general-purpose registers shown in Fig. 5.

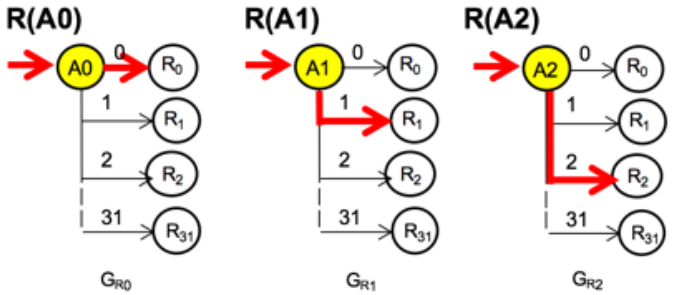


Figure 5 – HLDD for Register Decoders of a MIPS-like Microprocessor

3.3 Fault modeling in microprocessors using HLDDs

The high-level non-terminal node fault model is based on exhaustive testing of the nodes. In the case of the terminal node fault model, a pseudo-exhaustive set of data patterns needed for testing the related function was used. Each path in an HLDD describes the behavior of the system in a specific mode of operation (working mode). The faults which may affect the particular working mode can be associated with nodes along the related path.

Definition 3.1. *A non-terminal node related fault in the HLDD may cause the following corruptions of the model:*

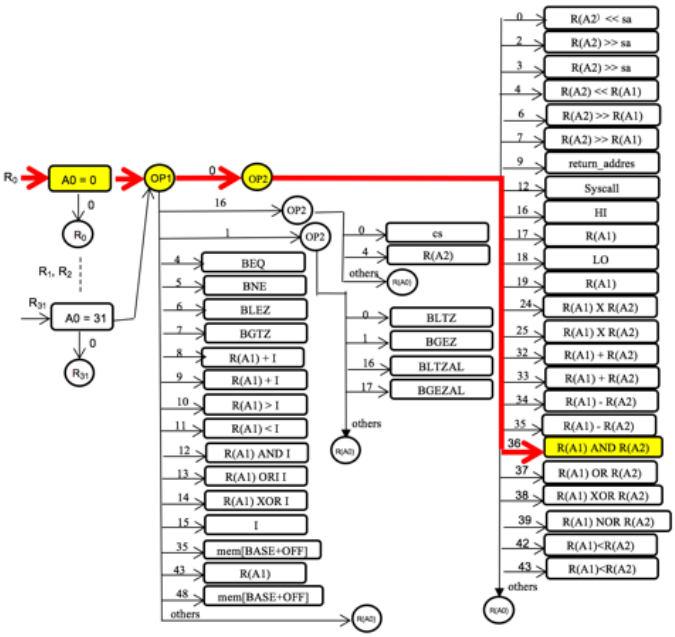


Figure 6 – HLDD for MIPS-like Microprocessor

1. *the output edge of the node is broken;*
2. *the output edge of a node is always activated;*
3. *instead of the activated edge, a combination of other edges is erroneously activated.*

As a fault model for the terminal node, a set of data patterns needed for testing the related functions of the data path is used in this thesis.

Fig. 7 describes how different fault models based on Definition 3.1 may be represented uniformly as node faults on the HLDD model. The graph $\mathbf{G}_{R(A1)}$ illustrates an addressing fault (F1): instead of the edge 3 of the node A1, another edge 0 (or both edges concurrently) are activated. This fault can propagate to other HLDDs of the model. For example, in the ALU graph it can cause either the fault of a wrong source (F2) or a fault of a wrong destination (F3). The fault type F4 (an instruction part erroneously activated) is illustrated by the fault of the node OP as "instead of the edge 2 the edge 1 is activated". All these faults belong to the third class of the HLDD fault model defined by Definition 1.

Representing the internal node under test in the HLDD by m, the set of all terminal nodes which can be reached from the node m at the current instruction under test starting from all erroneously activated output edges of the node m by $M^T(m)$. Let $f\left(m^T\right)$ represents the expression labeling the terminal node $m^T \in M^T(m)$. The following constraints which is a part of the fault model, are introduced for testing the control part of a Microprocessor [109]:

$$\forall m^T \in M^T(m) : \left[f\left(m^T\right) \neq \Omega\right] \tag{1}$$

$$\forall m_i, m_j \in M^T(m), _{i \neq j} : \forall k \left[f_k(m_i) < (f_k(m_i) * f_k(m_j))\right] \tag{2}$$

where $\Omega = ZERO$ (or ONE), and the symbol $*$ stands for logic OR (or logic AND), depending on the technology implemented in the microprocessor [64]. In this case,

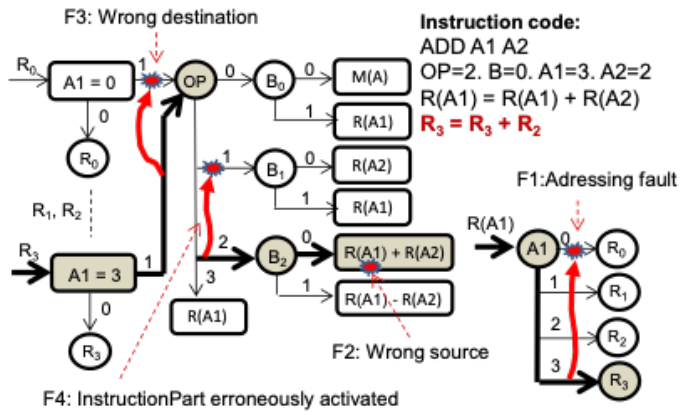


Figure 7 – Illustration of different corruptions of the HLDD by faults in MP

ZERO denotes a binary vector (00...0), and ONE denotes a binary vector (11...1). The index k refers to the bit number of the data words.

Focusing only on the case where $\Omega = ZERO$ and $*$ stands for OR, the constraints (1) and (2) can simplified into a constraint (3) as shown below for the reason that $f_k(m_i) < (f_k(m_i) * f_k(m_j))$ is valid always if $f_k(m_i) < f_k(m_j)$. Also, in the bit-based analysis the constraint (1) results directly from (2).

$$\forall m_i m_j \in M^T(m), _{i \neq j}: \forall k \left[f_k(m_i) < f_k(m_j) \right] \tag{3}$$

The high-level fault model for non-terminal nodes results from Definition 1, leading to exhaustive testing of control nodes of microprocessors in a natural way.

The number of functional faults for the non-terminal node m can be calculated as $N(m) = n_m (n_m - 1) k$ where n_m is the number of output edges of the node m, and k is the length of the data word.

Let us generalize the fault model to microprocessors using the ALU as a case study. It is assumed that the ALU executes n different functions $y = fi(d)$ by a set $F = \{f_i\}$ instructions, where d represents data operand(s) for fi, length of the data word (operand) is m, and the ALU is controlled by p control signals.

According to Fig.17, the control part consists of the multiplexer MUX and p control lines (control inputs to MUX which originate from the opcode field of the instruction register). Each of the n AND blocks in the execute unit's control part have p control and one single m-bit data input, while the OR block has n data word input from the AND block outputs. Each AND block consists of m AND gates with p control inputs, and a single bit data input.

For this case study ALU, two types of high-level functional fault models are classified as control faults (faults related to the control part of the ALU) and data faults (faults related to the data part of the ALU). These fault classes could also be generalized for other modules of the microprocessor. For the control faults, a novel high-level functional control fault model is therefore introduced as follows.

Definition 3.2. *Introduce for the function (instruction) $f_i \in F$, the following high-level control fault model $M(f_i)$ as a set of data operands $M(f) = \{D_i\}$, which satisfy the following constraints at least once for each bit k of y_i:*

$$\forall k \in (1, m) : \left\{ \exists d_i \in M(f_i) \left(y_{i/k} \neq 0 \right) \right\} \tag{4}$$

$$\forall f_j \in F, j \neq i : \forall k \in (1,m) \left\{ \exists d_i \in M(f_i) \left(y_{i/k} < y_{j/k} \right) \right\} \quad (5)$$

Note that y_i represents the data word considered as the result of execution of the function f_i with data operand(s) d_i as $y_i = f_i(d_i)$.

To test that the function f_i can be executed, and the result "$y_i = 1$" can be produced in each bit of the data word for detecting the faults (Stuck-at Fault)SAF/0 on all AND-gates inputs, the constraint (4) is needed. Likewise, the constraint(5) is needed to test that the result "$y_i = 0$" can be generated in each bit of the data word for detecting two types of faults: SAF/1 on all AND-gates inputs relating to the function f_i, and all functional faults that overwrites the value "$y_i = 0$" in each bit due to the control faults of other functions f_j, $j \neq i$.

The high-level functional control fault test, which satisfies the constraints (4, 5) will cover the following fault classes (for each control word bit):

1. For MUX as in (Fig.17): SAF/0 due to constraints (4), and SAF/1 due to constraints (5) on the inputs and related paths to outputs; conditional SAF, shorts, and multiple SAF on the inputs of AND-gates (due to applying exhaustive patterns)

2. Functional faults in the instruction decoder: no function accessed, multiple functions simultaneously accessed – similar to address decoder faults of memory test (due to constraints (5)) [110]

3. Functional microprocessor faults: instead of function f_i, another function f_j accessed, or multiple functions simultaneously accessed (due to constraints (5)) [64].

3.4 Minimization of Number of Edges in HLDDs

The HLDD model described in the previous section can be optimized by minimizing its edges, to reduce the complexity of test generation. In this thesis, two methods have been proposed. These are the greedy algorithm and branch & bound algorithm (B&B). These methods are used for creating HLDDs from instruction-level truth table to minimize the edges on graphs.

Let us introduce the following notations: $e(x_i,k)$ as an entry in TT, $e(k)$ as a vector of entries (the values of variables) in the k-th row in TT, and $E(x_i,v)$ as a group of rows where $x_i = v$, which has the size $|E(x_i,v)|$. For example, in Fig.2a, $e(x_2,5) = 3$, $|E(x_2,3)| = 5$ and $|E(x_3,4)| = 1$.

Definition 3.3. *Let us introduce operation of partitioning a given truth table TT into two sub-tables $TT(x = v)$ and $TT(x \neq v)$, so that $TT(x = v)$ will include the rows where $x = v$, and $TT(x \neq v)$ will include the rest of rows.*

Definition 3.4. *Introduce a term of average size of the values of a variable x in TT:*

$$AV(x) = \sum_{v \in V(x)} \frac{|E(x,v)|}{|V(x)|}$$

where $V(x)$ is the set of possible values of the variable x in TT.
For example, in the TT in Fig.2a, for x_1 we have $|V(x1)| = 3$, and $AV(x_i) = (4+2+2)/3 = 2.6$.

In this research, the following method of HLDD synthesis is proposed for a given TT based on definition 3.4.

Algorithm 1: GREEDY Minimization of Edges

1 First, the given TT is taken as the basis TT (BTT). In each step, a variable x from selected BTT is chosen, which has the biggest AV(x), and a new node is included into HLDD labelled by the same variable x. Then, the selected TT is divided into TT(x=v) and TT(x^1v). To continue synthesis of HLDD from the node x to the right direction, TT(x=v) is taken as the new BTT, and to the direction downwards, TT(x^1v) is taken as the new BTT. Both BTTs are included into a set B of pending BTTs. The described procedure will be repeated until all BTTs in B contain a single row. To each BTT in B, with vector e(k), a path in the HLDD will correspond, which is to be terminated by a terminal node of HLDD labelled with Fk.

An example of applying the first step of Algorithm 1 for TT in Fig.8a, is shown in Fig.8b and Fig.8c. The variable x_1 is selected from TT, because $AV(x_1)$ is the biggest among x_1, x_2, and x_3: $AV(x_1) = 2.6$; $AV(x2) = (3+5)/4 = 2$; $AV(x3) = (5+3)/6 = 1.3$. The HLDD begins with the root node x1, and the sub-table $TT(x = v)$ will be used to continue the HLDD construction to the right direction, whereas the sub-table $TT(x \neq v)$ will be used to continue the HLDD construction to the direction down. The full HLDD, synthesized for TT in Fig.8a, is depicted in Fig.8c.

Theorem 3.1. *Lower bound LB of the edges in the HLDD for the given TT can be calculated as* $\mathrm{LB} = \sum_{x \in X} |V(x)|$, *where* X *is the set of all variables in* TT.

Proof. Lower bound LB cannot be less than the number of different entry values, since each of these values serves as a possible fault location, and hence, must be represented in HLDD as an edge. On the other hand, LB can be achieved if an edge in the HLDD exists for all pairs $\{x \in X, v \in V(x)\}$ □

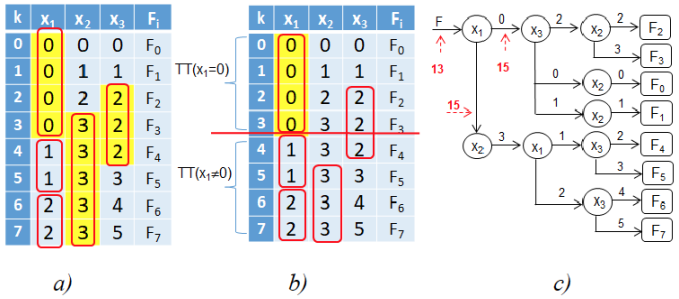


Figure 8 – Control Path TTs, HLDD for a microprocessor with 8 instructions and 3 op-code fields

Corollary 3.1.1. *The lower bound LB of the edges in the HLDD for the given* TT *is always achievable by Algorithm 1, if for the given number* n *of variables, the number of rows* r *in* TT *is equal to* n^2. *The lower bound LB in this case is equal to* $\sum_{k=1}^{n} 2^k$.

Proof. The proof of the corollary results from the non-redundancy of the TT. From this it follows that the full TT and all possible sub-tables of it, created during Algorithm 1, are always divisible into equal parts with preserving all groups $E(x, v)$ from splitting. □

There might be cases where $r < n^2$. In such cases, Algorithm 1 should be considered as a heuristic one and hence cannot guarantee synthesis of minimal HLDDs. In order

to find exact minimal HLDD, the Branch & Bound algorithm which is based on using the lower bound LB defined in Theorem 3.1 is proposed.

Algorithm 2: Branch & Bound Minimization of Edges

1 For each variable $x^* \in X$, the synthesis of a separate HLDD will start with the related root node x^*, according to Algorithm 1, and the TT is divided into $TT(x^* = v)$ and $TT(x^* \neq v)$, for each x^* respectively. Based on these tables, for each HLDD, the lower bound $LB(x^*)$ is calculated. The HLDD synthesis will continue for the HLDD(x_*), which has produced the lowest value of $LB(x_*)$. A new node labelled with the new variable x will be introduced into the HLDD(x_*) and its $LB(x_*)$ is updated. The procedure continues with the HLDD, which has the lowest value of $LB(x_*)$. Algorithm is completed, when an HLDD is found, which involves all variables of X, and the number N of its edges is less or equal to the lowest $LB(x_*)$ of pending HLDDs.

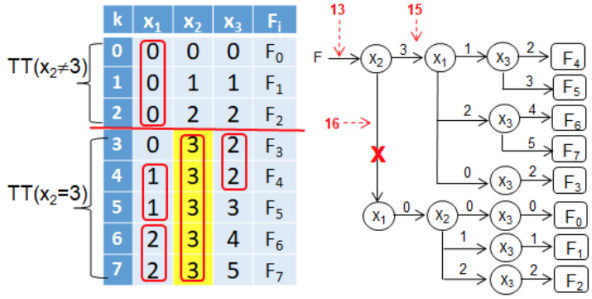


Figure 9 – TT split by selection of x_2 as the root variable in the HLDD

As an example, in Figures 8-10, three HLDDs have been created for the variables x_1, x_2, and x_3 respectively. The red numbers at the edges of the nodes denote the calculated LB-s. The best solution is depicted in Fig 8 with the number of edges $N(x_1) = 15$. In Fig 9, an HLDD with $N(x_2) = 16 > N(x_1)$ is found. The HLDD in Fig 10 is not completed, because we found that $LB(x_3) = 16 > N(x_1)$ at the early stage.

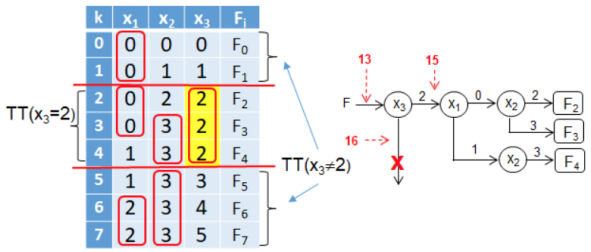


Figure 10 – TT split by selection of x_3 as the root variable in the HLDD

The feasibility of applying the proposed algorithms to formal generation and minimization of the HLDD model for the widely used MIPS-like microprocessor was investigated. Two versions of HLDDs for representing the control part of the processor are depicted below.

Fig.11 shows that the optimized model generated by Algorithm 1 has a size of 55 fault locations or edges. while Fig.12 shows a solution with 100 edges where the optimization criterion defined for Algorithms 1 and 2 was not employed. It is obvious that the reduction in the complexity of the generated high-level fault model based

on the proposed Algorithms is 1.8 times which has great impact on test generation complexity.

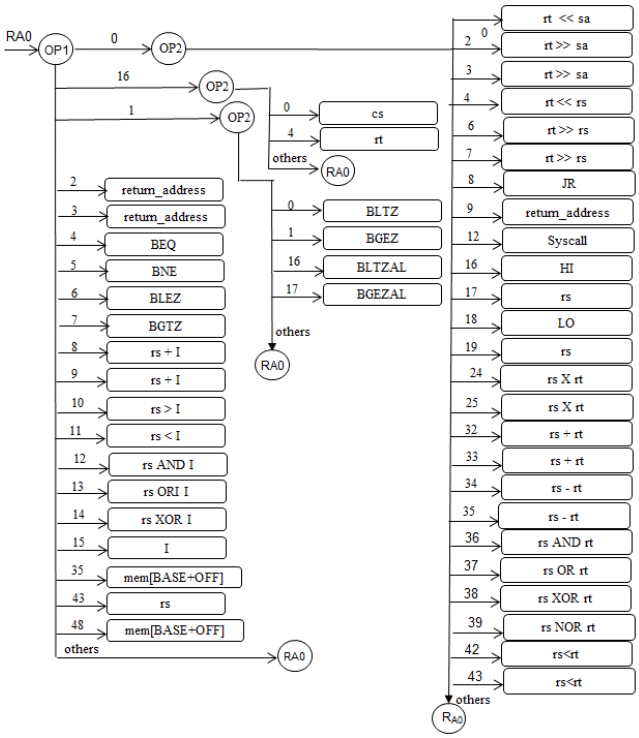


Figure 11 – Minimized HLDD model for the MIPS-like microprocessor

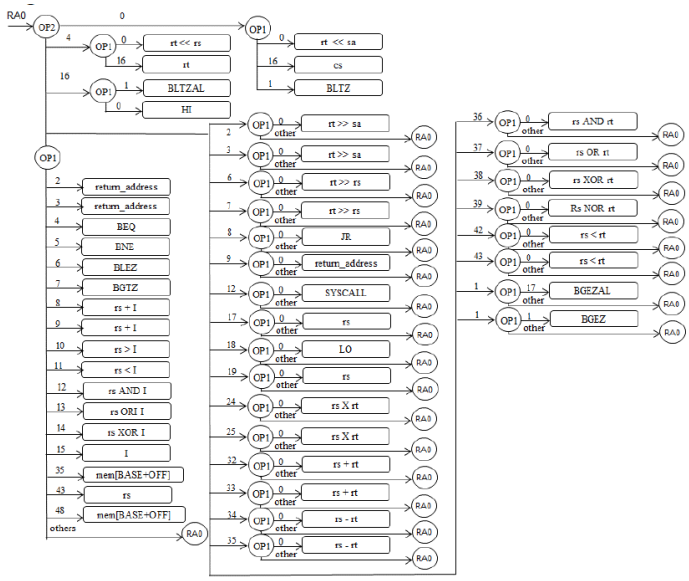


Figure 12 – Not-minimized HLDD model for the MIPS-like microprocessor

3.5 Optimization of the HLDD Model

Minimization of edges doesn't solve all the issues with test generation when considering the HLDD for modeling of faults. It helps with dealing with the reduction of the complexity of test generation. However, there is still a problem of scalability when in relation to the functional fault model proposed in this thesis. As the size of the set of functions grows, the number of high-level faults also grows very fast. This problem is not peculiar to only the method proposed here, it is also a known problem with memory testing. This means that if we desire to cover broader classes of faults, a longer test is needed.

The question of how to cope with the complexity explosion is finding trade-offs between some test characteristics like fault coverage, test length, test generation time, etc. One possibility is to partition the sets of functions F into smaller subsets and consider high-level test generation for subsets of F separately.

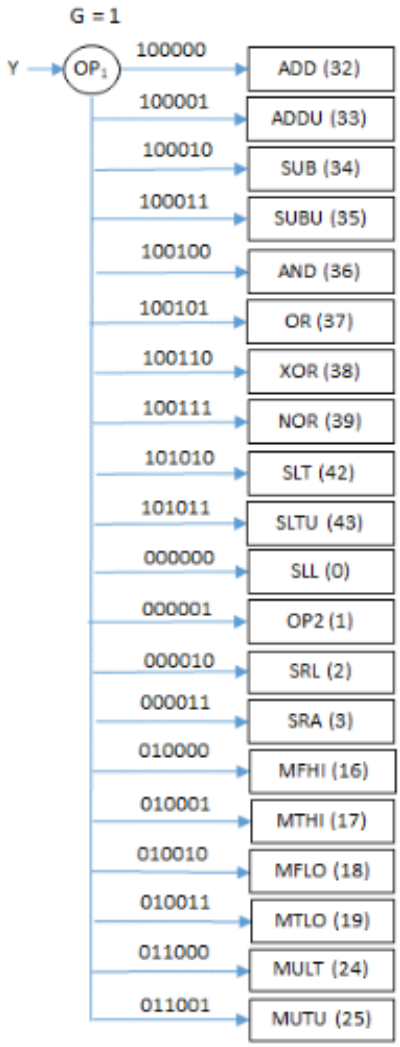


Figure 13 – HLDD with a single decision node for representing 20 MiniMIPS instructions

Consider in Fig.13, an HLDD for a subset of 20 instructions of the MiniMIPS microprocessor [111] which represents a subset of functions of the ALU. The single non-terminal decision node of the HLDD is labelled by the control variable c (denoted by the operation code of the instructions) having n control values labelling the output edges of the node c. The terminal nodes are labelled by data manipulation functions f_i to be used for creating the data constraints (5).

The HLDD in Fig.13 can be regarded as a MUX of the control part of the MUT, whereas terminal nodes describe the functions of the data part. Denote the HLDD as $G = 1$, meaning that the graph has 1 decision node. The size of the fault model for this subset of functions, $n = 20$ is $|CF| = (n-1)^2 \times m = 11552$, assuming the data word length is $m = 32$.

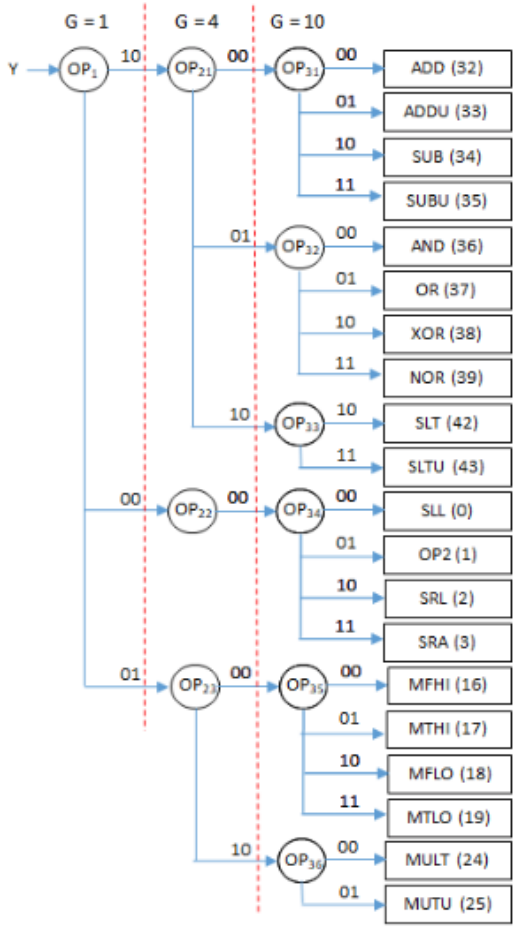


Figure 14 – HLDD for a subset of instructions of MiniMIPS

Depending on different partitioning of the set of control functions, HLDDs may have more than one non-terminal nodes. If the HLDD has more than one internal nodes, then for each non-terminal node m, the test is generated separately, where the subset of functions $F(m) \subset F$ related to the node m under test is set up from the HLDD, so that to each output edge of the node m, a terminal node m^T in HLDD (having a path from m to m^T) with related function f_j is mapped and included into $F(m)$.

To reduce the complexity of the model of the MUT, we can iteratively partition the set of functions by adding internal nodes into the HLDD.

Considering another version of the HLDD in Fig.14, which represents the same subset of 20 instructions of the MiniMIPS, but has 10 decision nodes in this case. Each decision node represents a partition, and the number of edges of the decision node corresponds to the size of the related subset of functions F.

We can imagine three versions of HLDDs for this subset of 20 functions (separated by red dotted lines):

1. $G = 1$ as a HLDD with a single internal node and 20 terminal nodes with $N = 12160$ functional faults (Fig.13) ,
2. $G = 4$ as 4 sub-graphs with decision nodes OP1, OP21, OP22 and OP23, and with 3, 10, 4 and 6 terminal nodes, respectively, resulting in $N = (6+100+12+30) \times 32 = 3904$ functional faults,
3. $G = 10$ i.e. the current HLDD in Fig.21 with 13 decision nodes, resulting in $N = (6+(6+12+2)+(12+12+2)+(12+2)) \times 32 = 2112$ functional faults (Fig.14).

We have generated tests for these three versions of HLDD models with the following results.

Using these results, we see opportunities for optimization of the test, "trading off" different parameters like test length (number of test patterns), achieved SAF fault coverage and test pattern generation time. We see that the result of the minimization of the complexity of the fault model due to the partitioning of the set of functions under test, the number of functional faults taken into account reduces dramatically from 12160 to 2112 (7 times), which has of course impact of the quality of testing the extended class of functional faults. At the same time, the SAF coverage does not change significantly, it decreases only from 99.03% to 99.61%, despite the reduction of the test length from 143 to 79 (nearly 2 times). As the complexity of the fault module decreases, the test generation time decreases as well.

Table 2 – Example of scalabilities for three versions of HLDDs

HLDD	No of Patterns	Number of High-Level faults N	SAF FC(%)	Time(s)
$G = 1$	143	12160	99.03	0.33
$G = 4$	100	3904	98.77	0.27
$G = 10$	79	2112	99.61	0.23

3.6 Summary

The contribution of this chapter includes the following

- A novel High-Level Decision Diagram (HLDD) is proposed as an extension of Binary Decision Diagrams(BDDs) which can be considered as an efficient formal means for functional modeling of the instruction sets of microprocessors, and as a means of self-test generation for microprocessors.
- HLDDs can be used both for high-level structural and functional modeling of microprocessors. In structural approach, the HLDD model allows for straightforward partitioning of faults in the modules of microprocessors into two groups

of high-level faults: the terminal nodes of HLDDs represent the functions of the data-part, and the non-terminal nodes represent the MUXes of instruction decoders.

- For functional modeling of microprocessors, two methods are proposed for creating HLDDs from Truth Tables with minimization of the edges of the HLDD graphs. These results in minimizing the size of the HLDD model of a given microprocessor and helps in reducing the complexity of the test generation process.
- Possible trade-offs between the complexity of the high-level fault model and the characteristics of generated tests such as test length, SAF coverage, and test generation time were investigated through partitioning the set of functions of the microprocessor into subsets of functions.

4 High-level functional test generation for the control parts of modules

In this chapter, a presentation on the control part test generation at high-level is made. It includes implementation-independent test concepts for testing the control faults, high-level functional control fault model, mapping of high-level faults to gate-level faults, generation of control faults test data and high-level fault simulation. An example of the control part test for the forwarding unit of a RISC processor is also presented as a special case.

4.1 Implementation-independent test concept for testing control faults

One of the objectives of this research is proposing a novel method for high-level testing of RISC microprocessors in a functional way, without resorting to the knowledge of low-level implementation details.

The main idea behind the method is based on defining the set of logic functions of the microprocessor as the target of testing. Such a set of functions is defined based on the instruction set and the description of the architecture of the processor as given in the manual. The set of the functions under test is partitioned into a set of groups where the results of the executed functions can be observed at the same node of the high-level structure comparably. For example, one group of functions may consist of logic and arithmetic operations derived from the instruction set, another group may be the forwarding functionality derived from the architectural description, and for both cases the observable node will be the output of ALU unit. Another group of functions is the branching functionality, which differs from the previously mentioned groups by having the observable node in the ALU 1-bit flag, which determines the branch direction of the instructions flow.

The same test template will be built for all functions of a particular group. This includes instructions for initialization, instruction sequence required to sensitize the function, and instructions necessary to observe the results of the test at the same node defined for the entire group of functions under test.

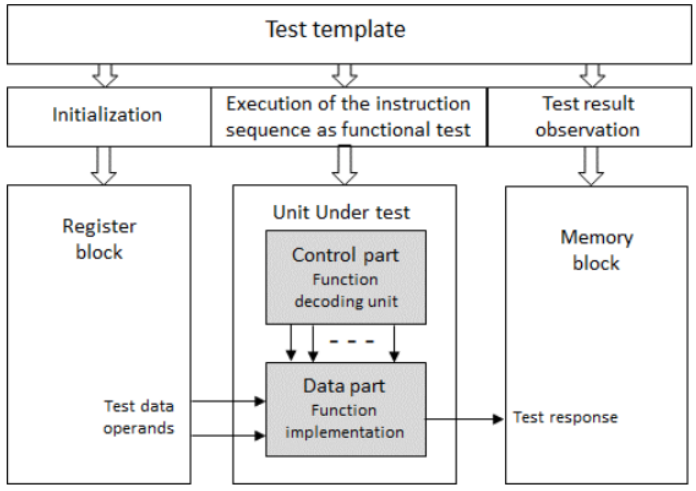


Figure 15 – Test Execution setup

In this thesis, I have focused on testing the execute unit, forwarding units in pipelined RISC processors, the general register bank and branch control unit. For these units, experiments have been carried out and will be presented later in the experimental section. Fig.15 shows the general set-up of the processor's high-level structure for testing the complete logic functionality. Depending on the templates, test data operands are loaded into the registers from the memory during the initialization phase of the test. Some instructions require that the source, target and destination registers are initialized, and some instructions require that only the source and target registers are initialized while some test data are used are immediate data. After initialization, the next phase of testing is the functional execution of instruction sequence. This means that each instruction is executed iteratively for all data operands. For each execution, the functional test response is stored in the memory by some set of instructions for memory manipulations.

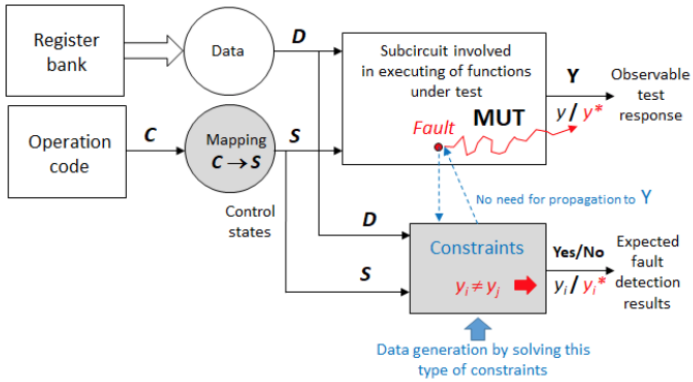


Figure 16 – Illustration of the proposed test concept

The concept of high-level implementation-independent functional test approach is illustrated in Fig.16. As a unit under test (UUT), we consider a sub-circuit of the processor involved in executing of the selected group of functions under test. The UUT has two high-level inputs: data D, control signals C, and an output Y which represents the observable node for observing the behavior of function of the group.

The low-level structure of this sub-circuit under test is unknown. In our model however, we consider that in the UUT the data part and control part can be partitioned. The UUT inputs are partitioned as well into data and control signals. However, the coding of the functions is unknown. For example, how exactly the control information (control signals C) is mapped into the control states S activating the selected functions (the mapping C $\rightarrow$ S) is unknown. Note here that there is a one-to-one mapping between the set of S and the set of functions under test.

Since the internal logic structure of the unit under test is unknown, the traditional methods of the path activation from the fault site to the observable point cannot be used. Rather, we can substitute the traditional fault propagation technique by direct generation of data which are needed to differentiate tested faulty behaviors from the correct behavior since all the functions of the group under test are well-comparable. To have such a high-level fault list, we will introduce a set of data constraints, where each constraint represents a particular high-level control fault. If a particular constraint is satisfied by the data used for activated function, then the related high-level fault, i.e. the respective low level fault is propagated to the observable node Y of the UUT.

In summary, Fig.16 illustrates how the upper black-box model of MUT is replaced with lower constraints based functional model for the control part of ALU. Here $y_i \neq y_j$ means that if there are data generated in D, which satisfy this constraint for expected results of functions f_i and f_j, then these functions are distinguishable, and if this constraint is not satisfied at least in one bit, some faults can remain not detectable by the test.

The control fault model and how it is applied to test generation will be discussed in more detail in the next section.

4.2 High-level functional control fault model

The focus of this research is on testing the executing units, register decoding module, pipeline forwarding module and system co-processor module in a pipelined RISC processors. The executing module for instance consists of a control part and data path as shown in Fig. 15. The method could also be applied to testing other pipeline stages, flags, branch prediction mechanism, caches etc.

Let us start with the execute unit and present it (in implementation-independent generic way) as an equivalent circuit where the control part is highlighted as an AND-OR multiplexer for decoding the instructions and extracting the results of the executed instructions as shown in Fig.17. In this unit, we consider the set of functions represented by logic and arithmetic functions.

The circuit in Fig.17 can be represented by an equivalent disjunctive normal form (EDNF) related to the execution unit, which can be taken as the functional basis of test generation that will be described later on. It is possible to generate test without the knowledge of implementation because a test developed for detecting all non-redundant faults in the EDNF, will also detect all the faults in the original circuit [36].

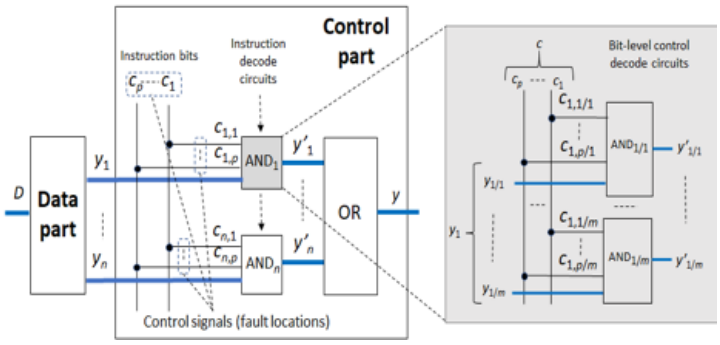


Figure 17 – Generic DNF based control structure of the executing unit

A generic representation of the control part in the form of multiplexer is implemented as described above, to develop the high-level functional control fault model. Assuming a more detailed view of the executing unit in Fig.4 as shown in Fig.17, where specifics of implementation are conceptual and hypothetical, We may assume that the data part performs n different functions $y_i = f_i(D_i)$ controlled by a set of instructions $F = \{f_i\}$, where D_i is the set of f_i-manipulated data operands. The length of the data word is m, and the number of control signals is p. The control part consists of the multiplexer MUX and p control lines controlling the MUX. The n AND blocks in MUX have p control each and a single m-bit data input, whereas the OR block in MUX has n data inputs from the outputs of AND blocks. Each AND block consists of m AND gates with p control inputs, and a single 1-bit data input. Thus, the control module described,

represented as a high-level multiplexer, consists of n different 1-bit logic level AND-OR multiplexers used to decode the instructions and extract the results of the instructions executed.

By implementing the mentioned theoretical MUX-based execution system, we separated the data and control parts and transformed the control block's function from "active" controlling of the manipulations in the data part to "passive" selection of the results of manipulations in the data part. What we mean by this is that all possible optimizations that may have been made during the design of the execution module have been ignored.

Another abstraction concerns the decoding of the control signal, which is highly dependent on implementation details in general. To ensure that test generation for the control part is implementation independent, the decoded control signals sent to the AND gates for selecting the related data manipulation results are represented by symbolic signals. This allows the issue of illegal instruction codes to be solved and helps identify redundancies in the control part.

The cost of having an implementation-independent test is in terms of test length increase. However, the gain is that we do not need to know how the unit-under-test is implemented and we can cover a larger fault class including multiple faults when compared with the traditionally measured single SAF.

The theoretical foundation of the proposed high-level control fault model is presented in the appendix section of [9].

4.3 Mapping of high-level faults to gate level

To understand how the high-level faults can be mapped to low level fault, let us Introduce the following notations of the input information for solving the problem.

Definition 4.1. *Let D_i^* be the set of data operands which satisfy the constraints of the fault model $M(f_i)$, T_i^* is the test for the instruction f_i, which uses the data operands $d \in D_i^*$, and $T^* = \{T_i^*\}$ is the full test, generated for all high-level control faults for the set of instructions $F = \{f_i\}$.*

Theorem 4.1. *The test $T^* = \{T_i^*\}$, which covers all non-redundant high-level faults of the fault model $M(f_i)$, also covers all gate-level non-redundant SAF in the control part of the microprocessor, which controls the set of functions F.*

Proof. The proof can be done in 2 steps. First is to consider the equivalent circuit of ALU control part presented in Fig.17 described as the following DNF.

$$y = c_{1,1}C_{1,2}\cdots c_{1,p}y_1 \vee c_{2,1}c_{2,2}\cdots c_{2,p}y_2 \vee \ldots \vee c_{n,1}c_{n,2}\cdots c_{n,p}y_n \qquad (6)$$

For each bit of the data word in the output of OR block. It can easily be shown that from generation of data which satisfy the constraints (4) and (5) for all functions $f_i \in F$, it follows that in the DNF, all SAF faults will be detected. In this DNF, the variables $c_{i,j}$for selecting the data results $c_j, j = 1,\ldots p$, present the global control signals $c_j, j = 1,\ldots p$ being either inverted or not, and covering all the 2^p combinations exhaustively in general case. Secondly, assume that the control circuit is optimized and is represented as a multi-level combinational circuit instead of the two-level DNF. In this case, we can represent the circuit as an equivalent disjunctive normal form in a similar way as DNF (6). As already mentioned, if there is a test set which detects all non-redundant faults in the EDNF, this test will also detect all faults in the original multi-level circuit [112]. □

The corollaries below result from the Theorem 4.1

Corollary 4.1.1. *If a high-level test is generated, so that the the constraints (4) and (5) are fully satisfied, but if there are some SAF in the related EDNF, which remain not detected by the high-level test, the not detected SAF are redundant.*

Corollary 4.1.2. *If there are some cases in the constraints (5), which cannot be satisfied by selecting data operands, these cases refer to the high-level redundancies in the model* $M(f_i)$.

Corollary 4.1.3. *If the high-level redundancies are removed from* $M(f_i)$*, and a high-level test is generated, so that the non-redundant constraints (4) and (5) are satisfied, but if there are still some SAF in the related EDNF, which remain not detected by the high-level test, the not detected SAF are redundant.*

Example 4.1.1. *Consider a simplified ALU unit which implements the set of three functions* f_1, f_2, f_3*, activated by a set of control signals* $\overline{c_2}c_1, c_2\overline{c_1}, c_2c_1$ *respectively. The ALU can be represented by the DNF:*

$$y = c_2c_1y_1 \vee c_2c_1y_2 \vee c_2c_1y_3 \tag{7}$$

The test $T^* = \{T_1^*, T_2^*, T_3^*\}$ *generated for the control part of ALU that satisfies the constraints (5) is depicted in Table 3.*

Table 3 – Example of a high-level control test

T_i^*	Test		Fault Table			Constraint Satisfied
	c_2c_1	$y_1y_2y_3$	$\overline{c}_2c_1y_1$	$c_2\overline{c_1}y_2$	$c_2c_1y_3$	
1	2		3	4	5	6
T_1^*	0 1	0 1 1	1 1 0	0 0 1	0 1 1	$y_1 < y_2, y_1 < y_3$
T_2^*	1 0	1 0 1	0 0 1	1 1 0	1 0 1	$y_2 < y_1, y_2 < y_3$
T_3^*	1 1	1 1 0	0 1 1	1 0 1	1 1 0	$y_3 < y_1, y_3 < y_2$

The table contains the test patterns in column 2, the fault table in columns 3-5, and the constraints satisfied by generating data in column 6 for the control test patterns. The detected gate-level faults in the fault table are highlighted in red: 0 is the value of a signal triggering the fault SAF/1. For example, the value of the output signal $y = y_1 = 0$ will change from 0 to $y = y_1 \vee y_3 = 1$ for the fault $c_2 \equiv 1$ in column 5. For the SAF/0, 3 more test patterns are needed to detect the faults (this is not shown in the table).

In the fault table we see that there is no detection of faults $c_1 \equiv 1$ in column 3 and $c_2 \equiv 1$ in column 4. These faults are redundant on the basis of Corollary 4.1.1. Minimizing the function (7) we get a new formula: $y = \overline{c_2}y_1 \vee c_2(\overline{c_1}y_2 \vee c_1y_3)$ where redundancies are removed and the test detects all SAF/1

It is worth noting that theorem 4.1 and Corollaries 4.1.1-4.1.3 have been formulated in consideration of the single SAF model. However, the strength of the proposed high-level fault control model extends beyond the single SAF fault class, as shown in the following corollaries.

Corollary 4.1.4. *The test* $T^* = \{T_i^*\}$*, covers all gate-level multiple SAF and bridging faults between control lines in the control part of the microprocessor, which controls the set of functions* $F = \{F_i\}$.

Proof. From (5) it follows that for each function $f \in F$, $\forall k : \left(y_{i/k} < y_{j/k}\right)$ for all $j \neq i$ must hold. This means that not only SAF/1 in a single control signal of a single function $f \in F$, $j \neq i$, can be detected (by overwriting $y_{i/k} = 0$ with $y_{j/k} = 1$), where the control words for f_i and f_j differ in a single bit, rather such overwriting of signals $y_{i/k} = 0$ with 1 can happen, and hence, can be detected due to multiple changes $0 \rightarrow 1$ for $f \in F$, $j \neq i$, leading to detecting multiple faults. On the other hand, from the constraints (4-5), and from the exhaustiveness of testing all the control functions $f \in F$, $j \neq i$, it follows that non-redundant bridging faults between the control lines can be also detected by T^*. □

Algorithm 3: RANDOM test data generation for ALU

Input: Instruction set of the processor
Output: Sets of test operands OP_i for each instruction, and fault table D
Notations: n – number of instructions (functions F_j), op – test operand, OP – current set of selected random test operands, $f_j(op)$ – the result of the instruction I_j for the operand(s) op, D – fault table, D_{ij} – w-bit entry in D (w – length of the data Word).

```
1  initialize OP = ∅;
2  Generate a set of R random operands;
3  for i = 1,...,n do
       // generation of operands for instruction I_i
4      initialize OP_i = ∅;
5      for j = 1,...,n(j ≠ i) do
           // operands for solving constraints f_{i,k} < f_{j,k}
6          Initialize D_ij = 0;
7          for all op ∈ R do
8              while (D_ij ≠ 0) do
                   // adding new operands for covering D_ij
9                  D_ij(op) = f_j(op) ∧ (f_i(op) ⊕ f_j(op));
                   // calculating fault coverage for op
10                 if (D_ij(op) ∨ D_ij) ⊕ D_ij ≠ 0 then
                       // check for the coverage increment
11                     begin;
12                     D_ij = D_ij ∨ D_ij(op);
                       // update of the coverage vector Include op into
                          OP_i
                       // new operand is selected
```

4.4 Generation of test data for testing the control faults

From the discussions above, we can deduce that the fault model defined by the set of constraints in (3) can be interpreted as the definition of the universe of high-level faults. The possibility of evaluating the high-level functional fault coverage as the percentage of constraints in (3) for a given test is a direct impact of this interpretation. Note also that we can overlook the fact that the faults $f_{i,k} \equiv 0$ and $c_{i,k} \equiv 0$ in the control path will not be taken in the fault universe of the control faults because these faults will be covered as a byproduct by the data part test $T(f)$. In the next session of this thesis, I will discuss the approach or method implored for testing the data part of the unit under

test. An algorithm to generate the test data set $T(c)$ for testing the control path in conformity with the constraint $f_{i,k} < f_{j,k}$ is presented.

The result of Algorithm 3 will be a set of operands OP_i for each instruction I_j, and the fault table $D = \left\| D_{ij}^k \right\|$ where $D_{ij}^k = 1$ means that the functional fault described by the constraint $f_{i,k} < f_{j,k}$ is covered at least by one operand $op \in OP_i$, otherwise $D_{ij}^k = 0$. The percentage of $1s$ in D is the high-level functional fault coverage of the test for the control path. We called the Algorithm 3 RANDOM since in each step of 7, the first random operand $op \in R$ that produces an increase in the fault coverage will be chosen, no matter how big it is. In order reduce the test length for testing the control path, another algorithm called GREEDY is implemented.

The GREEDY algorithm differs from the RANDOM at step 7, where the whole search space of random data is ran through to find the best operand that produces the maximum fault coverage for all $op \in R$. The next operand is selected in a similar way and this continues until the goal $D_{ij}^k = 1$ is reached, or no more operands can be selected to satisfy all constraints $f_{i,k} < f_{j,k}$.

It is possible that the constraint $f_{i,k} < f_{j,k}$ may not be solved in some cases. This could be that either the related functional fault is redundant or the search space R is not big enough. This research also makes attempt to provide proofs for high-level fault redundancies.

4.5 High-level fault simulation and fault coverage

In order to measure the fault coverage for the fault model $M(f_i)$, $f \in F$, proposed in Definition 3.2, by a given test T_i^* and a set of operands D_i^*, we introduce the high-level fault table as a matrix $E = \|e_{ij}\|$ with n columns and n rows, where n is the number of functions in F. Each entry $e_{i,j}$ in E is a m-bit vector $e_{i,j} = \left(e_{i,j/1}, e_{i,j/2}, \ldots, e_{i,j/m}\right)$, where m is the number of bits in the data-words $y_i = f_i(d_i)$, $d_i \in D_i^*$. We say that $e_{i,j/k} = 1$, if the constraint $y_{i/k} < y_{j/k}$ for the bit k in the set of constraints (5) is satisfied by the set of data operands in $D_i^* = \{d_i\}$, but if not, $e_{i,j/k} = 0$.

Let's take an example with an hypothetical ALU composed of 5 instructions: OUI, ADD, SUB, SLT and AND. In Table 4, we present a matrix $E = \|e_{ij}\|$ based on a test T_i^* for a set of functions $F = \{f_i\}$ which is executed by the set of instructions I={OUI, ADD, SUB, SLT, AND}. Each i-th row in the table represents the high-level control fault coverage of testing the function $f_i \in F$ and the respective instruction $I_i \in F$.

Table 4 – Example of a High-Level Fault Table

	f_1(OUI)	f_1(ADD)	f_1(SUB)	f_1(SLT)	f_1(AND)
f_1(OUI)		111111	111111	111111	000000
f_1(ADD)	111111		111110	111111	111111
f_1(SUB)	111111	111110		111111	111111
f_1(SLT)	111111	111111	111111		000000
f_1(AND)	111111	111111	111111	111111	

The fault table is the product of a high-level fault simulation for the given set of operands that will be used by the high-level test. For this study, the following high-level control-fault-simulation algorithm has been implemented.

Algorithm 4: Fault Simulation Algorithm

```
for all row instructions f_i, i = 1,...,n do
    for all data operands d_{i,j,1}, d_{i,j,2}, j = 1,...,n_i do
        for all column instructions f_h, h = 1,...,n do
            calculate the value y_h;
            check the relation y_i < y_h, h ≠ i;
            update the vector e_{i,h} ∈ E
```

As a result of algorithm 4, a simulation based high-level test generation method was implemented on the basis of random search for test data to satisfy the constraints(5).

In Table 4, 0s refer either the high-level control faults that are not detected or the possible high-level redundancies of the faults that are related to the constraints $y_{i/k} < y_{j/k}$. Here, i and j correspond to the rows and columns respectively while k corresponds to the bit number. All 0s in e_{ij} refer to the high probability of redundancy of the full set of high-level faults for all bits. This means that the constraints $y_{i/k} < y_{j/k}$ for all k cannot be satisfied for the respective instructions I_i and I_j.

4.6 Extension of the Fault Class Beyond SAF

The ideas of the proposed fault model are adopted from the known methods of memory testing, particularly from March test [110]. The goal of the motivation was to expand the fault class to be covered by the study, to the one used in the case of memories.

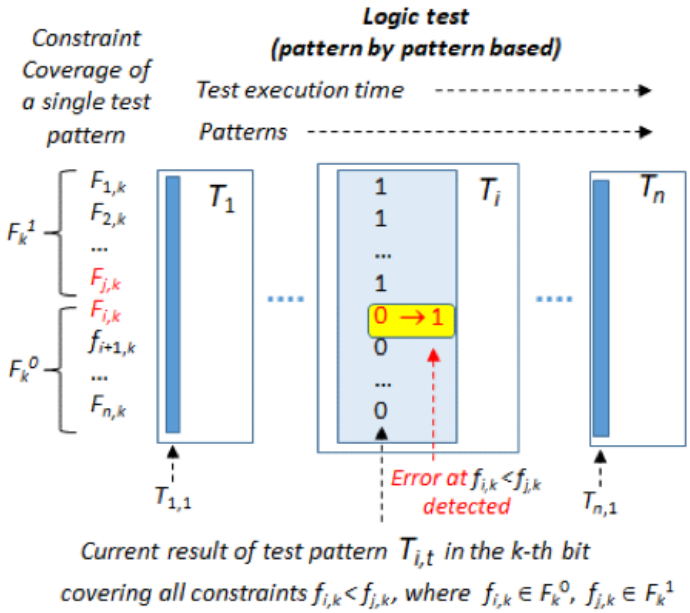


Figure 18 – Unrolled test execution evolving in time

considering an example of the March test depicted in Fig.19, and comparing it with the test flow developed for a logic MUT shown in Fig.18. The comparison between the memory test and the logic test is equivalent in addressing cells in memory and controlling the functions of $f_i \in F$ in logic MUTs. In case of memories, testing of cells

(data part) and the addressing logic (control part) can be easily joined in the same test, whereas in the proposed approach, testing of data part and control part proceeds separately.

In case of memory, the initialization of constraints (writing $1s\,(W1\uparrow)$ into cells) can be done once for all cells in a single cycle. Then, having these constraints stored, the following test cycle $(r/w0\downarrow)$ and observation cycle $(r1)$ can be carried out.

In the proposed method, the constraints cannot be stored, rather they have to be produced "on-line" at each test pattern. In Fig.18, a test pattern $T_{i,t} \in T_i$ including test data $d \in Di$, is illustrated, showing the values it produces on-line for the $k-th$ bit of all functions $f_i \in F$, simultaneously. All functions for the bit k are partitioned by the data $d \in D_i$ into two groups F_k^0 and F_k^1. We see, that this particular test pattern with data d covers only a subset of constraints for $f_{i,k}\,(d) < f_{j,k}\,(d)$, where $f_{i,k} \in F_k^0$ and $f_{j,k} \in F_k^1$.

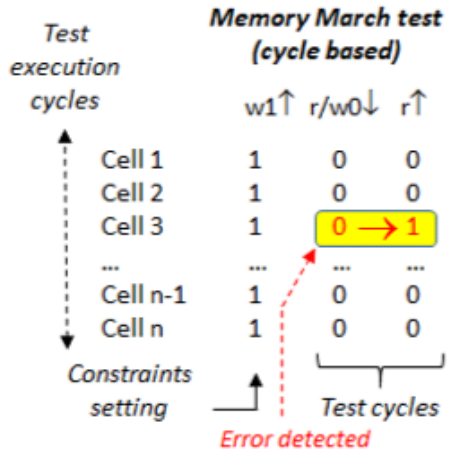


Figure 19 – Illustration of the March test for memories

In case of memory, in each step of the test cycle $(r/w0\downarrow)$, when reading the Cell i, all constraints [Cell i] $<$ [Cell j] are covered by a single run through all the cells. Here, [Cell i] means the value stored in the Cell i. In case of the proposed method of testing a logic MUT, the test for $f_i \in F$, has to be repeated with other data d until all the constraints (5) have been satisfied for all pairs of functions $\{f_{i,k}, f_{j,k}\}$.

The comparison of the proposed data constraints based test method with March test for memories reveals the possibility of applying the proposed approach, not only for the combinational MUTs like ALU but also for sequential MUTs. If in sequential MUTs, a part of data $d \in D$ belongs to the registers or memory, the test must include a proper initialization sequence.

Consider a MUT, represented by a set of mappings:

$$(c_i \in C) \rightarrow (f_i \in F),$$

where C is a set of mutually exclusive control signals (instructions) produced by the control part of MUT, and F is the set of operations (data manipulations) taking place in the data part of MUT.

By test data generation, used in the March test for memories and in the proposed test method for logic MUTs, the coverage of the following functional fault classes by the proposed method results [110]:

CL-1: With a certain instruction $(c_i \in C)$, no activity f_i in F will happen.

CL-2: There is no instruction (c_i), which can activate a function $f_i \in F$. A certain function is never accessed.

CL-3: With a certain instruction (c_i), multiple functions $\{f_i, f_j, \ldots\} \in F$ are activated simultaneously.

CL-4: A certain function $f_i \in F$ can be activated with multiple instructions $\{f_i, f_j, \ldots\} \in F$.

The fault classes CL-1 – CL-4 are illustrated in Fig.7 [110]:

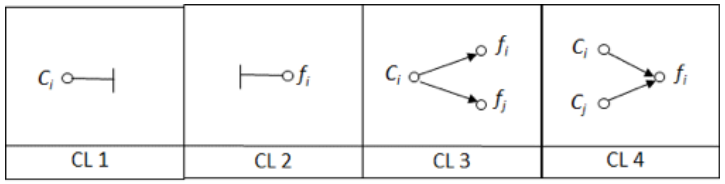


Figure 20 – Functional control fault classes CL 1 – CL 4

It is easy to realize that these high-level functional fault classes also cover SAF (CL-5) and bridging (CL-6) fault classes, i.e. these faults can be collapsed, and do not need to be taken into account any more, except when the fault coverage of these faults for a given implementation is under interest.

Address decoders built out of CMOS gates can exhibit CMOS stuck-open faults [CL-7] as shown in [113]. The effect of such faults is that the combinational instruction decoder will behave as a sequential circuit for certain control signals. The consequence of such a fault is that another instruction will be decoded and executed. However, this fault can also be collapsed, because it will be covered by the faults of CL-4.

Any multiple low-level structural fault CL-8 (SAF or shorts), in a particular implementation, will cause a change of an instruction $c_i \rightarrow c_j$, which in turn can be considered as the fault from class CL-4, and hence, be collapsed.

Other general fault classes, such as cell-internal defects (CL-10) [114] or conditional SAF (CL-9) [49], also known as functional faults [115], pattern faults [45], input pattern faults [44] and fault tuples [52], will manifest themselves as a change of instruction code $c_i \rightarrow c_j$, and are covered by the fault class CL 4.

We have shown, that the structural fault classes CL-5 – CL-10 are collapsed by the implementation-independent high-level functional fault classes CL-1 – CL-4, which are used in memory testing and are covered by the March test [110]. On the other hand, in Section V, we have shown, that the test for microprocessor MUT, which satisfies the constraints (5) will cover the same fault classes CL-1 – CL-4 used in memory testing. Finally, from Definition 3.2, it follows, that the fault classes CL-1 – CL-4 can be represented by a single fault class $CF = \{f_{i,k} \rightarrow (f_{i,k}, f_{j,k})\}$.

The relationships between iterative fault collapsing are shown in Fig.21: first, collapsing of structural faults (CL-5 – CL-9) by functional faults used in memory testing (CL1 – CL-4) [110], and consequently, collapsing of the faults (CL1 – CL4) by the general high-level control fault CF, developed in this thesis.

The proposed method is also extendable to generating a test that detects all transition delay faults (TDF). The TDF model assumes that only one gate in the circuit is affected by the delay fault. Each gate has two TDFs: a slow-to-rise fault and a slow-to-fall fault. To detect TDF in a combinational circuit, two input patterns $V = (v_1, v_2)$ must be applied. The first pattern v_1, initializes the circuit, while the second pattern v_2, activates the fault and propagates its effect to some primary output. The second pattern

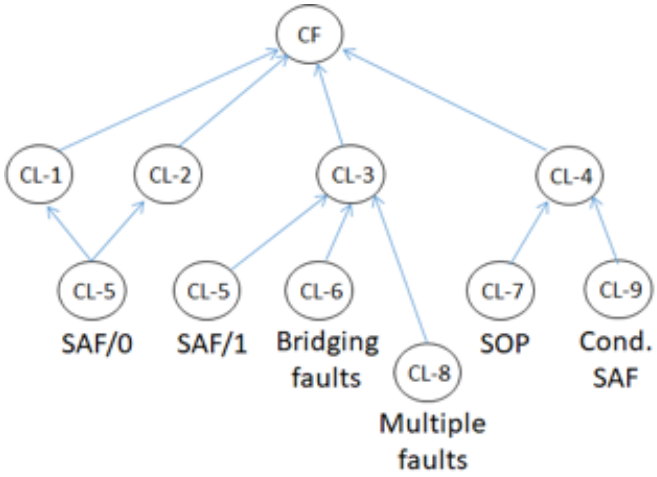


Figure 21 – Fault collapsing relationships

can be found by a SAF test generation tool.

Traditionally, for a given combinational circuit, both patterns are generated by ATPG. For example, for testing a slow-to-rise TDF, the first pattern f_1 initializes the fault site to $v_1 = 0$, and the second pattern f_2 is a test for SAF/0($v_2 = 1$) at the fault site. A TDF is considered detected if a transition occurs at the fault site, and a sensitized path extends from the fault site to some primary output.

In the proposed functional approach, the target TDFs cannot be directly defined, and the pair of signals (v_1, v_2) cannot also be found by sensitizing the respective paths, because the gate-level structure of the circuit is not given. However, according to the proposed fault model, for every testable SAF r of the unknown implementation, there is always an existence of a test pattern f_2 with signal v_2 in the functional test sequence, generated by solving the constraints (5), which detects the fault r. Since we do not know, which SAF is detected by the patterns of the test sequence, we could not generate the first sensitizing patterns f_1 to produce v_1 by conventional TDF testing methods.

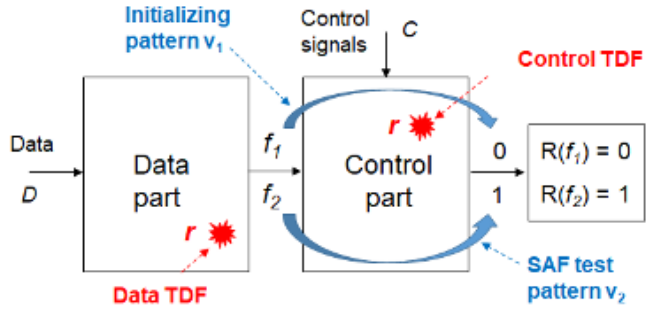


Figure 22 – Transition delay fault testing in modules under test

However, since the functional faults of the proposed model are sensitized by manipulations in the output domains of functions rather than structural sensitization of faults, the same concept can also be extended for finding test pairs (v_1, v_2) for detecting transition delay faults by substituting the task of finding the test pattern f_1 with the

task of finding the output response $R(f_1)$ through simply inverting $R(f_2)$, where the role of the input pattern f_1 is to produce the signal v_1.

Fig.22 illustrates the main difference between the proposed functional approach to testing of transition delay faults and the traditional approach. Instead of targeting the pairs of input test patterns (f_1, f_2) to produce signals (v_1, v_2), the target is to find the observable output values $(R(f_1), R(f_2))$. This eliminates the need to have knowledge of the structure of the circuit for paths sensitizing, thereby making the method implementation-independent.

To explain how this works, let us consider the circuit in Fig.22 as an example. If we have a test pattern $(C.D)$ representing a control word C (instruction) and a data operands D, which executes the function f_2 , $y = f_2(C, D)$ to produce the value $R(f_2) = y$, $y \in \{0,1\}$, and store it in the register R. Assume that this pattern detects a SAF $r \equiv x$, $x \in \{0,1\}$, either in the control part or in the data part of the module under test. To detect the TDF on the fault site r, the value on r must be pre-initialized to $\bar{x}$. If the TDF is present on this fault site, then the fault will propagate through the circuit and produce the faulty value $\bar{y}$ in the register R. To detect this TDF in R, it is sufficient to pre-store this expected faulty value $\bar{y}$ in R. If TDF is missing or exists, the observable value $\bar{y}$ will either change to y, or remain the same as $\bar{y}$ respectively.

The proposed method of test generation for TDF is fundamentally a new approach, easy implementable at the functional level in connection with the new method of test generation for microprocessors. The novelty lies in the substitution of input test data generation by generation of output expected responses.

4.7 Identification of redundant faults in microprocessors

In chapter 4, a high-level fault table was presented based on algorithm 4. According to this, we identified that there could be instances where the constraint $y_i < y_j$ cannot be satisfied for a given test. For these faults, either the test has to be improved, or it should be proven that the not detected faults are functionally redundant. In this section, I would provide redundancy proofs based on a 1-bit truth partial table for any related faults where the constraints mentioned above cannot be solved.

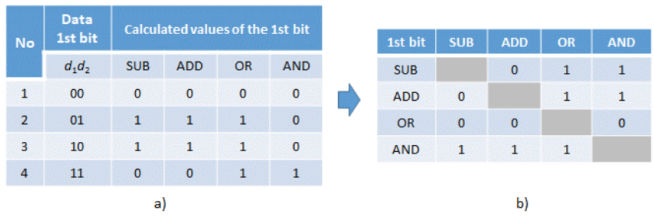
Example 4.1.2. *For example, in most cases of ALU operations, it is very easy to identify this type of redundancy. For example, if a 1-bit function $y_i = f_i(d_1, d_2)$ refers to AND operation and $y_j = f_j(d_1, d_2)$ refers to OR, it is straightforward that the constraint $y_i < y_j$, i.e. $(d_1 \vee d_2) < (d_1 \wedge d_2)$ cannot be satisfied.*

Apart from these easy cases, there are cases where there is no solution for constraint $y_i < y_j$ only in a single bit k, or in a few bits. In these cases, we use the partial truth table method for providing the redundancy proof.

The idea of the approach is to demonstrate the equivalence of partial truth tables or to prove the impossibility of solving the constraints for the functions involved in the constraint relation, to pick a few bits as possible that would be needed for proof.

In Fig. 23, an example of a 1-bit partial truth table for the functions SUB, ADD, OR, AND, and XOR is shown for selected bits k. The pairs $00, 01, 10, 11$ in Fig.23a, represent the values of the data variables (as arguments) under analysis and the 1-bit values in the columns show the results of the bit related operations, proving the functional redundancy of both functions in this bit.

By comparing the values in column 3-6, we can populate the fault table in fig.23b with the values of 0 or 1, where 0 means that the constraint(5) is not satisfied and 1 means that the constraint was satisfied. For example, if we take the functions SUB



No	Data 1st bit	Calculated values of the 1st bit			
	$d_1 d_2$	SUB	ADD	OR	AND
1	00	0	0	0	0
2	01	1	1	1	0
3	10	1	1	1	0
4	11	0	0	1	1

a)

1st bit	SUB	ADD	OR	AND
SUB		0	1	1
ADD	0		1	1
OR	0	0		0
AND	1	1	1	

b)

Figure 23 – Example of redundancy proofs with 1-bit truth table

and ADD, a comparison of these two functions for all possible data d_1, d_2 would never satisfy the constraint (5). By this, we mean that this implies that their equivalence contradicts the constraint. The same analysis goes for the functions OR and AND.

In more detail, if $d_1, d_2 = 00$, the function $y_i = f_i(d_1, d_2)$ for ADD will be 0, And the function $y_i = f_i(d_1, d_2)$ for SUB will also be 0. Therefore $y_i < y_j$ is not satisfied for this data. If $d_1, d_2 = 01$, the function $y_i = f_i(d_1, d_2) = 1$ for ADD, while the function $y_i = f_i(d_1, d_2) = 1$ also for SUB. The constraint $y_i < y_j$ is also not satisfied for this data. Likewise, the data $d_1, d_2 = 10$ and $d_1, d_2 = 11$ cannot produce any value of y_i and y_j for ADD and SUB respectively, such that $y_i < y_j$. This sort of analysis can be carried out for other functions, thereby proving functional redundancies. This can also be regarded as functional untestable faults.

In some cases, the partial truth table method will not work, because the results of operations may substantially depend on all bits of the word like for increment or decrement operations. When this happens, specific corner cases should be found for the proof of redundancy. For example, to prove the equivalence of increment and decrement operations in the least significant bit, the operand $1 \ldots 110$ should be used, where both instructions INC and DEC produce the same result "all 1s".

4.8 Case Study of the Control Part Test for Forwarding Unit of MIPS-like RISC processor

Considering a case where the set of functions F to be tested does not represent the processor's main functional properties described in the instruction set, but other non-functional properties such as performance, for which the forwarding unit is responsible.

Imagine the forwarding system as the high-level circuit in Fig.24, consisting of the pipeline registers; ID/EX, EX/MEM and MEM/WB [111], the forwarding control unit, the pipeline register data-selection multiplexer and ALU with observable output node. The role of the control unit is to compare the addresses r_s used in the current instruction with addresses $r_d(EX)$, $r_d(MEM)$ stored in the respective pipeline registers, and to produce the control signals c_1, c_2 and $c*$ for selecting the data D_1, D_2, and D_3, respectively, from different pipeline registers. The data D_i represents the values of the functions $f_i \in F$, where F represents the functionality of the forwarding unit.

The test patterns generated for testing the forwarding function in Fig. are depicted in Table 5. The test consists of 3 groups of patterns (in each 2 patterns, framed in bold): forward from EX/MEM (c_1, D_1), MEM/WB (c_2, D_2), and no forward (c_*, D_3). Each of the 6 test patterns in Table 4 are applied in a cycle for as many test data needed to solve the constraints (5).

For solving the constraints (5) like $r_d(EX) < r_d(MEM)$, the data that is needed for the register numbers to either produce or not to produce hazards, is generated using pseudo-exhaustive test data for comparators. The patterns "all 1s" and "all 0s" are used

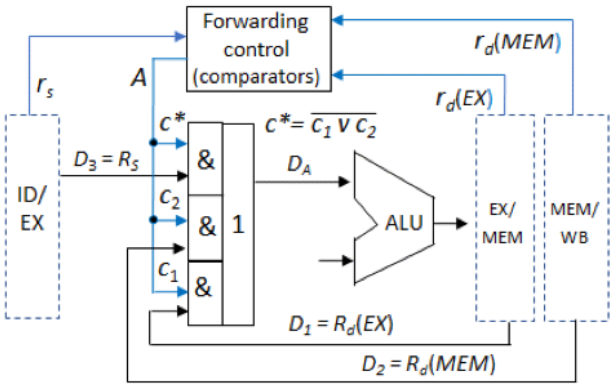


Figure 24 – Example of testing the pipeline forwarding unit

Table 5 – Test generation for pipeline forwarding unit

Var	The values of var(hazard detection condition)	Test Pattern					
		1	2	3	4	5	6
c_1	$rd(EX) = rs$	1	1	0	0	0	0
$f_1 = D_1$	$R_d(EX)$	0	1	1	0	1	0
c_2	$rd(MEM) = r_s$	0	0	1	1	0	0
$f_2 = D_2$	$R_d(MEM)$	1	0	0	1	1	0
c^*	$\overline{c_1 \vee c_2} = 1$	0	0	0	0	1	1
$f_3 = D_3$	R_s	1	0	1	0	0	1

to organize parallel testing (solving the constraints (5)) of all bits of the data words. In contrast to testing the execute unit (where for all instructions, the constraints were solved by data of the same time frame), the constraints are solved by data from different time frames and located in different pipeline registers in the case of the forwarding unit. For this purpose, relevant test templates are produced.

For example, Fig.25 demonstrates two test programs generated for testing the hazard detection and data forwarding functions. In case of hazards in addresses of registers $(r_a = c_b)$, the data from the register r_b is forwarded from EX/MEM stage (for Fig.25a), and from MEM/WB (for Fig.25b). The cases correspond to test patterns 1-2, and 3-4 in Table 5 respectively. For the 1st patterns the values "all 0s" are forwarded, and for the 2nd patterns "all 1s" are forwarded. Two tests are needed for mutually satisfying the constraints (5): $rd(EX) < rd(MEM)$ and $rd(MEM) < rd(EX)$.

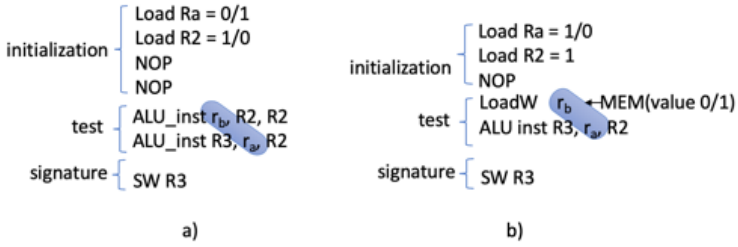


Figure 25 – Examples of testing the pipeline forwarding unit

The red entries in Table 5 correspond to the signals where the SAF faults are sensitized and can be detected. The test patterns 1,3 and 5 are created for testing the

control faults where the constraint (5) is satisfied. These patterns also test the SAF/1 faults in the data part. The test patterns 2,4,6, satisfying the constraint (4), are testing the SAF/0 faults in both the control and data parts.

4.9 Summary

In this chapter,the following contributions are highlighted

1. The proposed high-level implementation-independent test generation concept for microprocessors based on the partitioning of the modules into control and data parts was presented.
2. Development of foundations for the creation of a new paradigm of a high-level functional fault model in the control part of microprocessor modules.
3. The new fault model allows implementation-independent test generation to be carried out using only high-level information on the functionality of the module under test.
4. Methods for generating high-level test data used to test the control part of the microprocessor modules were developed. Also, a high-level measure and a related high-level fault simulation method were developed for evaluating the high-level quality of the test.
5. The new method of test generation avoids time-consuming conventional fault propagation along the true paths in digital circuits, allowing drastic speed-up in calculating the requisite test operands.
6. The new test generation method is the first method which in addition to traditional SAF, allows covering of a large class of low-level faults such as shorts, conditional SAF, Transition Delay Faults(TDF) and multiple SAF using a new dedicated high-level fault model, without knowing the low-level structure.
7. A novel mixed-level method for identification of low-level redundant faults was developed based only on simulating the high-level test for the given circuits implementation.
8. Presentation of control part testing of the non-functional unit such as the forwarding module.

5 Pseudo-exhaustive testing of data-parts of modules

There might be several possibilities for generating test patterns or testing the working mode of the microprocessor. These methods can be classified into hierarchical, exhaustive or pseudo-exhaustive, and functional or heuristic methods.

In the hierarchical approach, the test data is generated at the gate level and mapped to a higher level as a set of constraints for the high-level functional fault model. Several methods using this approach have been proposed in [21, 71, 107]. By using pseudo-exhaustive data to test the data path of the microprocessor, the test procedure is independent of the details of the processor core under test.

In this chapter, the concept of pseudo-exhaustive test strategy, and how it is used for testing the data part of modules of the MIPS-like processor is discussed. I highlight that this concept allows us to detect not only the non-redundant SAF but also other classes of faults.

5.1 Overview of the pseudo-exhaustive test concept

Rather than using the conventional gate-level ATPG [71], we can use the pseudo-exhaustive testing method to test the data path. The benefit of pseudo-exhaustive data is that it does not make the test generation process reliant on the implementation details of the processor cores being tested. The pseudo-exhaustive test generation method is based on the idea of testing the operations in all bits independently of other bits.

To apply a true exhaustive approach for testing logic operations, we can use only four exhaustive patterns { (0,0), (0,1), (1,0), (1,1) } per bit since they are substantially independent in all bits, but for unary operations like shifts or move, only two patterns are sufficient.

Table 6 – Pseudo-exhaustive test data for addition operation

No	...	$4-\text{bit}$	$3-\text{bit}$	$2-\text{bit}$	$1-\text{bit}$	$0-\text{bit}$
		$a_4b_4c_4$	$a_3b_3c_3$	$a_2b_2c_2$	$a_1b_1c_1$	$a_0b_0c_0$
1	...	000	000	000	000	000
2	...	010	010	010	010	001
3	...	100	100	100	100	010
4	...	110	001	110	001	011
5	...	001	110	001	110	100
6	...	011	011	011	011	101
7	...	101	101	101	101	110
8	...	111	111	111	111	111

In Tables 6 and 7, examples of pseudo-exhaustive data are shown for additions and subtractions. In this example, ripple carry is used for addition, and ripple borrow is used for subtraction. In the case of carry-lookahead addition, the pseudo-exhaustive approach will be more complex and more test data will be required. The idea of the pseudo-exhaustive approach shown in Tables 6 and 7 reveals that all bits of all ALU operations f_i, $i = 0, 1, 2, \ldots$, are exhaustively tested. For ADD and SUB operations, 8 data pairs are required to cover each bit of all 3 inputs (two operands and carry/borrow bit) combinations of the adder or subtractor.

To generate patterns as shown in Tables 6 and 7, we begin from the least significant bits, calculate the carry ci for the next bit, and match ai and bi with the calculated carry ci for the next bit, so that all exhaustive combinations for this bit segment have been achieved. In this way, the created patterns for the $2nd$ and $1st$ bits can be copy-pasted

Table 7 – Pseudo-exhaustive test data for subtraction operation

No	...	$4-\text{bit}$	$3-\text{bit}$	$2-\text{bit}$	$1-\text{bit}$	$0-\text{bit}$
		$a_4b_4c_4$	$a_3b_3c_3$	$a_2b_2c_2$	$a_1b_1c_1$	$a_0b_0c_0$
1	...	000	000	000	000	000
2	...	110	011	110	011	001
3	...	001	100	001	100	010
4	...	100	110	100	110	011
5	...	011	001	011	001	100
6	...	101	101	101	101	101
7	...	010	010	010	010	110
8	...	111	111	111	111	111

for the next two-bit sections to the right.

5.2 Testing of multipliers with pseudo-exhaustive test patterns

Another contribution of this work is the design of a new method for bit-parallel testing of array multipliers using a new approach of data-controlled circuit segmentation to transform the task of testing 2-dimensional arrays into testing a set of 1-dimensional array. We combined pseudo-exhaustive and deterministic testing to achieve high coverage of a large class of gate-level faults for this method.

Multiplication in microprocessors' data-path architectures usually occurs by optimized array multipliers of different architectures. An overview and comparison of various types of multipliers are given In [116]. In which case, the unsigned array, Baugh Wooley, modified Booth, and modified Booth Wallace tree multipliers are considered. Most of the test approach for multipliers in the state-of-the-art, need to change the design, which as a rule introduces performance issues. For SBST in general-purpose microprocessors, the added extra control inputs make these approaches not applicable. Moreover, these methods are not developed for detecting delay and Stuck-open(SOP) faults.

Here, we propose a novel testing approach suitable for both BIST and SBST related applications. The innovation is in a regular test data structure, consisting of two pattern sets: multiplier operands and multiplicands. Both can easily be generated algorithmically and can be used for BIST. This pseudo-exhaustive test (PET) method allows multiple combination cell faults and sequential faults to be detected while avoiding fault-masking.

Let us consider a ripple-carry adder which belongs to a class of full dependence (FD) circuits as a 1-dimensional Iterative Logic Arrays(ILA). This can be easily segmented into cells with three inputs (partial dependence circuit) and can be exhaustively tested with eight patterns in parallel. The process of producing test patterns for a ripple-carry adder is demonstrated in Table 6.

The system of multiplying two 8-bit patterns can be shown in Fig.26 as the traditional "pen and paper" method. Here, A-pattern $(a7;a6;a5;a4;a3;a2;a1;a0)$ represents the multiplicand multiplied by the B-pattern (11111111). By multiplying the multiplicand with each bit of the multiplier B, 7 partial products are generated and added. Such a multiplier, represented as a matrix of $(n-1)^2$ full adders and n half adders can be regarded as a 2-dimensional ILA which is very difficult to test by organizing PET for all 1-bit cells in parallel.

We propose a method for transforming this 2-dimensional n-bit array multiplier ILA into a set of $(n-1)$ 1-dimensional n-cell ILAs ($n1$ 1-bit full-adder and 1-bit half-adder), which can be pseudo-exhaustively tested almost as easily as ripple-carry adders. We

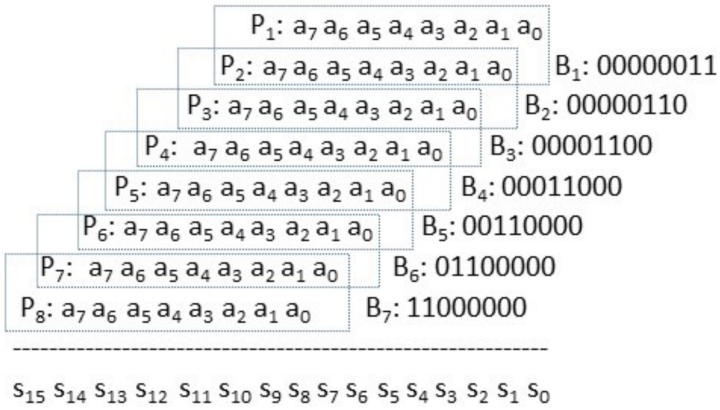


Figure 26 – Process of multiplying

introduce a method of data-controlled circuit segmentation for such a transformation, where multiplier segments are selected by multiplier operands (B-patterns) so that each B-pattern selects a related single 1-dimensional ILA of 1-bit adders. In total $(n-1)$B-patterns with a pair of 1-s and all other bits 0 are needed for n-bit multipliers, as shown in Fig. 26.

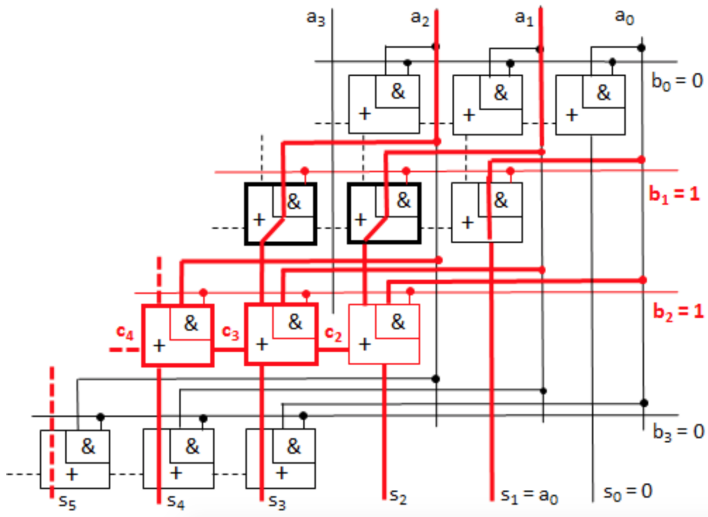


Figure 27 – Segmentation of multiplier

In Fig.27, we illustrated how the data-controlled circuit segmentation is performed to select the desired 1-dimensional array of 1-bit adders for testing by selecting the B-operands. The B pattern 0110 in this instance selects the 3rd column of adders to be evaluated (b2 $=1$), the value of b1 $=1$ selects the first operand, and the value of b2 $=1$ selects the second operand in this 1-bit adder row. The bold red lines illustrate the links involved in transmitting stimuli from primary inputs to the inputs of the adder being evaluated and propagating response signals to primary outputs.

The PET generation of A-patterns is illustrated in Table 8 as a process of assigning consistent values to the tuples of signals (c_i, a_{i+1}, a_i), where $i = 0, 1, \ldots, n-1$, and n

Table 8 – Pseudo-exhaustive test data generation for ripple-carry multiplier array

N	7			6			5			4			3			2			1			0	
	c7	a8	a7	c6	a7	a6	c5	a6	a5	c4	a5	a4	c3	a4	a3	c2	a3	a2	c1	a2	a1	a1	a0
1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1
2	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0	0	1
3	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
6	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0
7	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1	0
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1
10	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0
11	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0
12	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1
13	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	0	0	1
14	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1
15	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1

is the data word length. Each row in Table 8 represents a test pattern (two operands) for testing the 1-bit adder row selected by a B-pattern. The values of (a_i, a_{i+1}) are chosen so that in each column all the 8 combinations of bits were present.

5.3 Combining pseudo-exhaustive test with deterministic test data

In Table 9, we compared various changes to the proposed pseudo-exhaustive test (PET) for the regular array multiplier and the multiplier of a MIPS-like microprocessor. The column labelled (n) indicates the different word length we are considering. We measure the SAF coverage (FC%) and test length (# pat) using only 11 PET patterns for the A-operand. PET* is an extended version of PET where additional $3*(n-2)$ patterns were added, which is generated by shifting "111" and "101" in B-patterns for the selected 3 A-patterns. A good quality of PET* is that it preserves the regularity property of test data, making it suitable for built-in self-test (BIST). (PET+DET) in Table 9 shows a hybrid test using PET with additional ATPG generated deterministic test patterns. This guarantees 100% SAF coverage without any change to the multiplier and therefore is well usable for SBST.

Table 9 – Comparison of different PET versions

Standard array multiplier								
n	Regular solution:For BIST and SBST						Only for SBST (with additional stored patterns)	
	PET		PET*		PET + DFT		PET + DET	
	FC%	# pat: $11\times(n-1)$	FC%	# pat: $11\times(n-1)+3\times(n-2)$	FC%	# pat	FC%	# pat
8	98.03	77	100	95	100	77	100	77+7
16	98.89	165	99.92	207	100	165	99.98	165+13
32	99.49	341	99.95	431	100	341	99.98	341+17
Multiplier of MiniMIPS microprocessor and Booth Multiplier								
n	Regular solution: For BIST and SBST				Only for SBST(with additional stored patterns)			
	miniMIPS		Booth		PET + DET(miniMIPS)		PET + DET(Booth)	
	FC%(PET)	FC%(PET*)	FC%(PET)	FC%(PET*)	FC%	# pat	FC%	# pat
8	98.03	99.82	98.02	99.13	100	77+15	100	77+13
16	95.83	99.56	98.05	98.15	100	165+44	100	165+8
32	93.71	98.06	96.0	97.35	99.98	341+77	99.98	341+75

5.4 Summary

In this chapter:

1. A general pseudo-exhaustive method for generating test data for the data parts of microprocessor modules was presented. This guarantees high coverage of a large class of faults including stuck-at-faults(SAF), shorts, conditional SAF and multiple SAF.
2. A novel repetition-based pseudo-exhaustive test generation method was developed for different classes of multipliers as a case study.
3. Experimental results demonstrate that for standard array multipliers, the proposed data controlled segmentation method achieves 100% fault coverage for a broad class of faults. For other classes of multipliers (MiniMIPS, Booth multiplier), SAF coverage is less. This can be seen as a limitation of the method.
4. Also, from the experimental results, it was shown that combining the proposed pseudo-exhaustive method with deterministic patterns for the MiniMIPS and Booth multipliers, it was possible to achieve 100% SAF coverage.

6 Software-based test program generation for microprocessors

In this chapter, the process of generating software-based self-test is discussed. For such programs, no implementation details are needed. We relied on the HLDD for modelling the microprocessors using the instruction set description given in the manual of the processor. The task of test program generation is divided into several parts. The first part includes the generation of data for testing the control parts of the processor modules, while the pseudo-exhaustive data generation method mentioned in previous chapters has been adopted for the data part. The second part involves the automatic synthesis of the test program from the manual parameter file constructed from the HLDD.

6.1 Environment for the SBST synthesis (The flow of tasks)

The SBST framework designed in this thesis is divided into three parts, as shown in Fig.28. The first part is fully automated and involves quality high-level test data generation. The second part uses the data generated by the first step, and a test template to generate test programs. This step has been semi-automated in python scripts. The last step involves fault grading based on dumps collected during execution of the test program generated at the second phase of the workflow. The fault grading is performed using Tetramax developed by Synopsys. The inputs for the designed framework are the MUTs of processors and the sets of test templates created using the information of instruction set architectures (ISA) and other architectural information about microprocessors like pipeline, forwarding, prediction units, etc. The high-level (HL) test data generator for the control part of MUT, uses a novel constraint-based functional HL control fault model and generates tests to verify that all functions of MUT are selected correctly. This has been described as the conformity test.

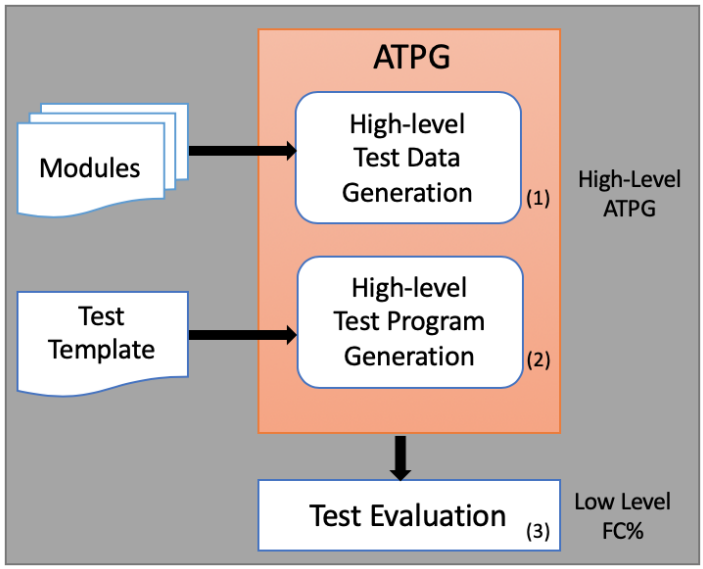


Figure 28 – High-level Test Generation Big Picture

Since the efficiency of a SBST is largely dependent on the quality of the test data

used. We prove by experiments that these data generated covers 100% high-level faults and also 100% low-level faults if all redundant faults have been removed.

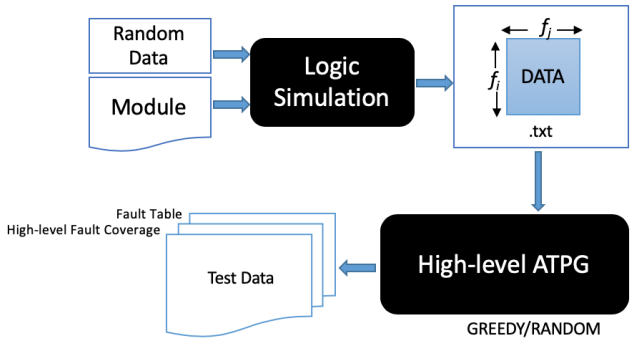


Figure 29 – High-level Test Data Generation

Fig.29 describes the HL ATPG tool within the SBST framework. The tool aims at generating quality test data for testing the control part of the processor from a random set of data by simulating the functions of MUT. The Questasim simulator generates for each data, a set of functional outputs $y_i = f_i(d_i)$ for every instruction Ii: $f_i \in F$. The functional output of the logic simulator is passed as input to the HL ATPG tool that we have developed.

Let's take an example of the ALU module of the MIPS-like processor for explaining the test data generation process. Table 10 shows a typical subset of instructions of the ALU of the processor. The operations are represented by operation codes and the related formulas f_i for calculation of the output values of the ALU, which can be obtained from the logic simulation of the module. The high-level structure of an ALU is depicted in Fig.30. The control variable c can have values from the domain $\{0,1,2,\ldots,n\}$. We denote by I_i the instruction (with opcode c_i) which performs the function f_i in the ALU.

The Questasim simulator generates for each data, a set of functional outputs $y_i = f_i(d_i)$ for every instruction Ii: $f_i \in F$. The functional output of the logic simulator is passed as input to the HL ATPG tool that we have developed.

Table 10 – Instruction subset for MIPS-like ALU

Mnemonic	f_i	Mnemonic	f_i
ADD	f_0	SLT	f_8
ADDU	f_1	SLTU	f_9
SUB	f_2	BEQ	f_{10}
SUBU	f_3	BNE	f_{11}
AND	f_4	BLTZ	f_{12}
OR	f_5	BGTZ	f_{13}
XOR	f_6	BLEZ	f_{14}
NOR	f_7	BGEZ	f_{15}

Two High-level test data generation methods were used. The first method is purely random, where a huge search space of random data is used, and each data is iterated through while checking its contribution to the fault coverage of the test for the control path. Data which contributes to the coverage of the test is added to the list of needed patterns until 100% fault coverage is reached or until redundant faults present are detected. The second method is called a greedy-random method. Here, the entire

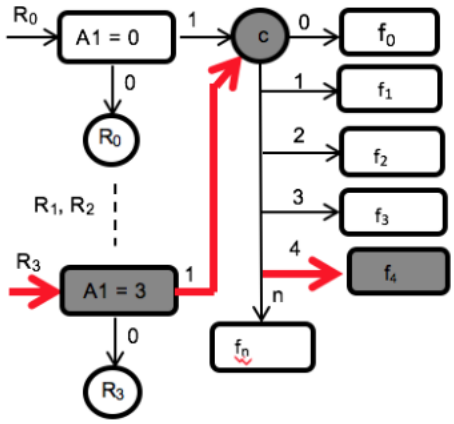


Figure 30 – HLDD for ALU in Table.10

search space is searched through, and only patterns with the best contribution to the fault coverage until 100% coverage is achieved are selected.

Fig.31 presents the full HL SBST program generation tool within the SBST framework. Here, the high-level test data, which includes data generated from the high-level ATPG for testing the control part, pseudo-exhaustive data for testing the data part and manually generated test templates, serve as inputs. Based on these inputs, the HL SBST program generation autogenerates the test program for testing the processor.

Figure 31 – High-level Test Generation

Fig. 32, shows a flow of the test infrastructure, where all the tools used are highlighted as a cooperative system. The flow begins with simulating the modules with random data. It helps to easily generate control test data using the high-level ATPG with fewer efforts. From the flow, one can proceed to use the test program generating tool to produce software-based self-test applications for the given processor.

The flow of test generation in the context of the SBST framework developed through this research can be divided into the following as represented by the numbering in Fig.32:

1. Data generation
2. Manual preparation
3. Program generation
4. Program compilation
5. Test execution

6. Fault grading/simulation

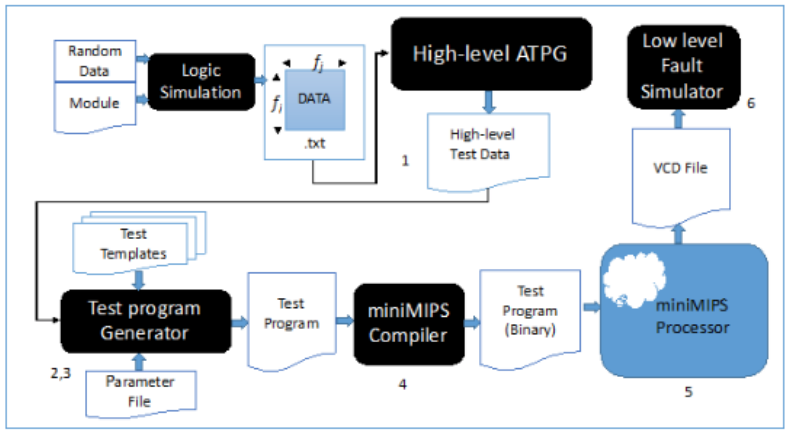


Figure 32 – Test Framework

The test program and test data are loaded into the processor's memory, as depicted in Fig.33. The test response is stored as a VCD file and used for fault grading analysis in the third part of the synthesis.

6.2 Test templates and the concepts of conformity and scanning tests

Following the methods described to generate test data for processor testing, the test patterns can be divided into two parts: **conformity test patterns** targeting control faults and **scanning test patterns** targeting data faults of the unit. The test generated for the control part using the conformity test patterns is called conformity test, while scanning test is the test generated for the data part using the scanning test patterns. In general, the conformity test by definition is a test for a non-terminal node to test a part of the control part of the microprocessor while the scanning test is is a test for a terminal node to test a part of the data path of the microprocessor.

Conformity tests. Generating a conformity test for $m \in M_N$ in HLDD produces exhaustive test for the variable x(m) which labels the node m. It consists of the two following steps:

1. Generation of a test $T(m)$, which satisfies the constraints of Definition 3.2, by activating a path from the root node to the node m under test, and for each value $v \in V_{(x(m))}$, a path from m to a terminal node $m_v \in M_T$. Here $V_{(x(m))}$ denotes the full set of values of the node variable $x(m)$.

2. Justification of $T(m)$ by initialization of registers involved in the test with data satisfying the constraints (4,5).

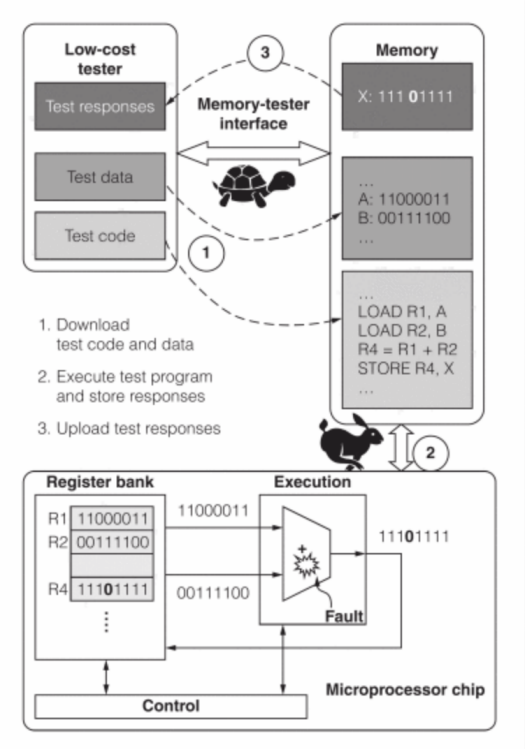


Figure 33 – SBST Test program flow [30]

Algorithm 5: Conformity Test Algorithm

```
for all v ∈ V(x(m)) do
    for t = 1,2,...,p do
        Initialize the data registers R(m) with contents R(m,t);
        Execute the working mode under test;
        READ the value of x.
```

Scanning tests. The scanning test T(m) consisting of the static part $T_{ST}(m)$ and dynamic part $T_{VAR}(m)$ is synthesized hierarchically:

1. Generation of the static part of the test $T_{ST}(m)$ on the HLDDs by activating a path from the root node to the terminal node m, and generation of the dynamic part $T_{VAR}(m)$ at gate level by any ATPG.
2. Justification of $T(m)$ by initialization of registers involved in the test with data in $T_{VAR}(m)$.

Algorithm 6: Scanning Test Algorithm

```
for t = 1,2,...,p do
    Initialize the data registers R(m) with contents R(m,t);
    Execute the working mode under test;
    READ the value of x.
```

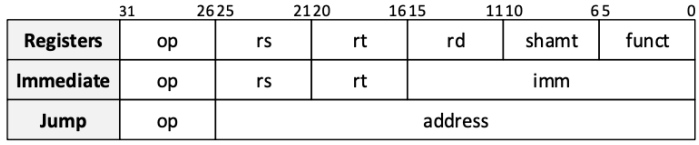


Figure 34 – MIPS-like processor instruction format

The concept of creating test-templates for microprocessors based on HLDD was introduced. The idea of templates is known as instruction-grouping. This happens during the process of transversing through the nodes of the HLDD. For the templates, gaining some understanding of the instruction formats of the processor would be useful. Instructions are first grouped based on information from the manual, according to the instruction types. For the MIPS-like processor used as one of the case studies for this research, instructions are divided into 3 formats: R-type, I-type and J-type instruction format. Fig.34 shows a pictorial representation of the instruction types of the processor.

For the R-type instruction format, The OP and the funct can be represented as OP1 and OP2, respectively. These are the decoders for selecting the instructions to be executed by this group of instructions. The rs, rt and the rd can also be expressed as decoders A1, A2 and R respectively, for selecting the source and destination registers. The HLDDs for these register decoders is illustrated in Fig. 5, while Figure 6 illustrates OP1 and OP2. The instruction and register decoders represent the control parts which can be tested by the conformity test approach as described in algorithm 5.

Figure 35 – Generalization of instruction format of MIPS-like processors

The R-type instruction format can be used as a baseline for creating a generalized representation of all the instructions in the microprocessor, as shown in Fig. 35. This generalization translates into the different conformity test variations forming five different templates based on Algorithm 7.

Algorithm 7: Test program template Algorithm

```
for all instructions I_i, i = 1,2,...,n do
    for all data operands (d_{i,j,1}, d_{i,j,2}), j = 1,2,... n_i do
        Read d_{i,j,1};
        Read d_{i,j,2};
        Execute the instruction I_i, = (opcode_i, d_{i,j,1}, d_{i,j,2});
        Write the test result in signature analyzer
```

Some examples of template structures as described by Algorithm 7 is depicted in Fig. 36. In this example, four different templates were shown starting with the generalised case of R-type instructions.

Figure 37 shows a mapping between the HLDD constructed from ISA and the instruction type of the processor. As previously mentioned, the R-type instructions represent a generalized case. Other instruction formats can also be represented with similar mappings. From this mapping, it is easy to understand that the instructions terminating from OP2 belong to the same group, while instructions terminating from OP1 (usually the immediate instruction types) also belong to the same group.

Operation_**instruction_method** load pattern **Instruction** rd, O1, O2 sw rd, offset(M) Jal increment	Operation_**instruction_method** load pattern **Instruction** rd, O1, SA sw rd, offset(M) Jal increment
Operation_**instruction_method** load pattern **Instruction** rd, O1, I sw rd, offset(M) Jal increment	Operation_**instruction_method** load pattern **Instruction** rd, O1, O2 MFLO rd sw rd, offset(M) MFHI rd Jal increment

Figure 36 – Test program templates

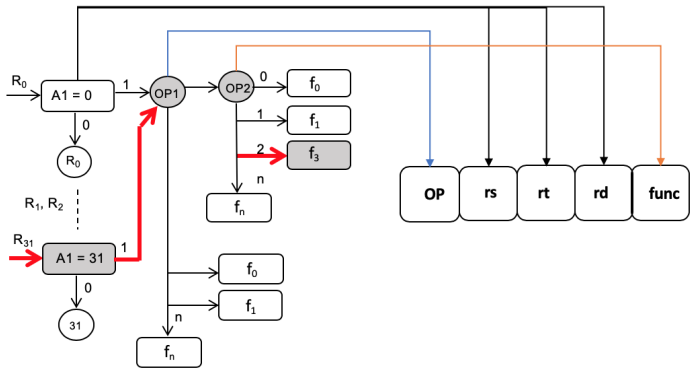


Figure 37 – Mapping between HLDD and Instruction Type

In Fig.38, a general concept of template variation for testing the different control nodes is shown. Based on the targeted control node also know as the non-terminal nodes from the HLDD of the microprocessor, test templates are constructed.

6.3 Organization of the full test program

The test is organized according to the structure shown in Fig.39, which essentially consists of test templates, loop execution instruction and test data. The complete test is divided into sections where each section's purpose is to test a sub-set of Functions.

The complete test has three embedded loops structure. The first and outer loop consists of using the same template to execute the test sections. The number of loops is equal to the number of different test templates describing a subroutine with a uniform structure. These templates have three parts which are: initialization of the processor, execution of the instruction under test which targets a function $f_i \in F$, and propagating the response to the node of observation as shown in Fig 15.

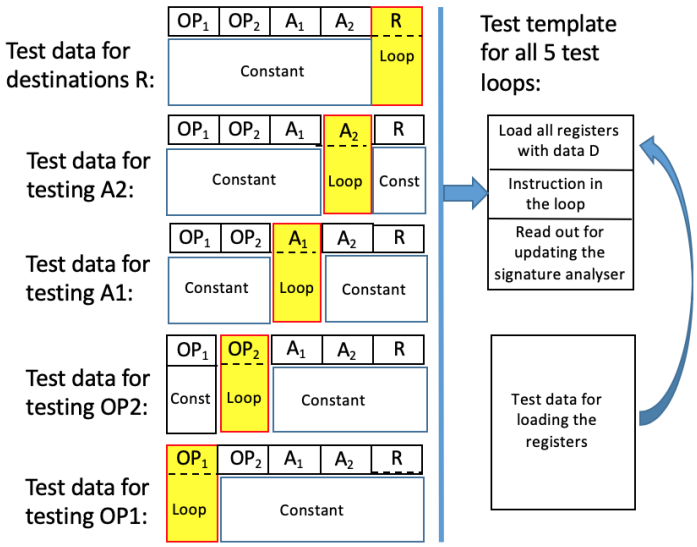


Figure 38 – Conformity Template Variations

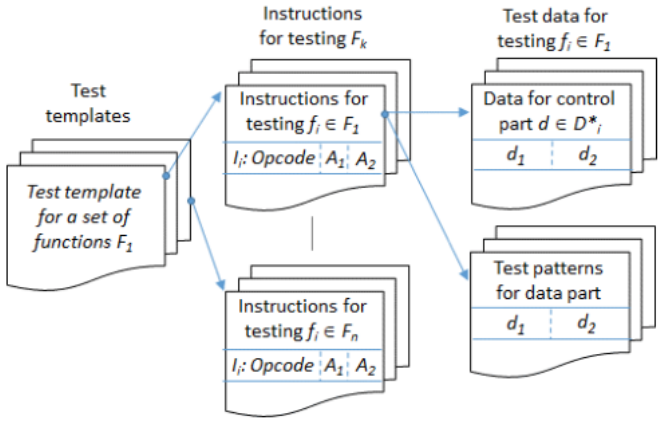


Figure 39 – Architecture of the test program

The second and middle loop consists of repeating the selected test template for all functions $f_i \in F$ over a list of related instructions I_i.

Two consecutive inner loops will be performed for each $Ii \in F$ instruction under test: for testing the control part and testing the data part. The number of test data $d = (d_1; d_2) \in D_i^*$ for testing the control part is calculated by the method described earlier for test data generation, while the number of test patterns for testing the data part is determined by the length of the pseudo-exhaustive test sequence derived by the methods discussed in [117], [118].

6.4 Multiple fault detection in microprocessors

Most of the fault simulation tools and tests targets only stuck-at faults [119]. In this research, we investigated the possibility of targeting multiple faults using the concept of test groups for HLDD. The idea of test groups is to prove the correctness of a part

of the system's functionality rather than keeping track of fault coverage during test generation as in the traditional case. The method can be regarded as a generalization of the logic level test-pair approach for identifying fault-free wires in gate-level networks to a high-level identification of fault-free functional blocks.

We consider a VLIW processor with two execution slots [120] as shown as a high-level structural layout in Fig.40. Each slot has a fetch register that receives the current instruction from the program memory via IN buses. Instruction of the slot i contains the following control signals: d_i (destination register), $ctri_1$ and $ctri_2$ (source registers), $ctrAlu_i$ (operation to be performed by ALU).

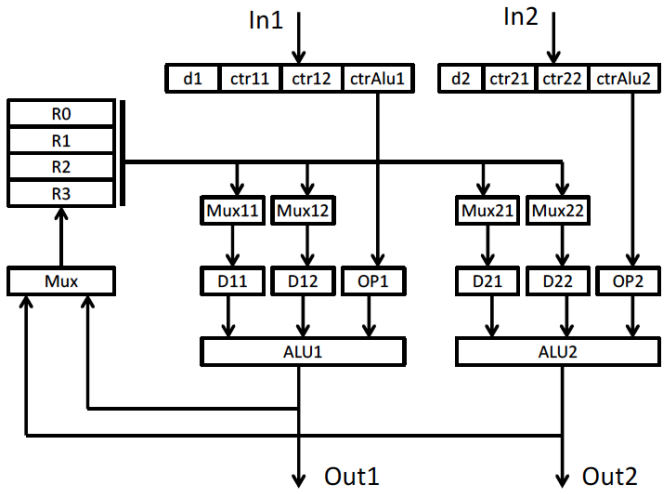


Figure 40 – Structural Representation of the VLIW processor

The components $Muxi1$ and $Muxi2$ represent the read ports of the register file for slot i. The processor has a simple register bank with four registers, and two write ports. Both write ports are represented by the component Mux, whereby each write port i is controlled by the bit field d_i. $Di1$ and OP_i represent the pipeline register between decode and execute-stage. The write-back stage of the processor neglected for simplicity.

As shown in Fig.41, we can describe the components of a slot of the VLIW processor by HLDDs. The graph $D1$ describes the behaviour of the processor's left read port, and a register is selected depending on the value of the $ctrs1$ control variable. The read value from the selected register is assigned to the variable $D1$ as one of the arguments for functions of the ALU. For the second argument $D2$, a similar graph as $D1$ is needed. The behavior of the ALU is modeled by the graph Alu. The graph $R0$ represents one of the registers and the $DMUX$-component for writing into registers the output value of ALU. The value will be written into $R0$ if $d1 = 0$, otherwise, for any other value of d1, the content of $R0$ will be held.

The graphs in Fig.41 represent the behavior of the VLIW processor components during the cycle of the instruction, and in this sense can be named as single-cycle HLDDs. However, we can represent the behavior of the processor with a single multi-cycle HLDD, as in Fig.42 by joining the graphs in Fig.41, and including the full block of registers and the control circuitry in the model.

Such a joint HLDD allows better modeling of the multi-cycle test sequences consisting of test stimuli and test response observation cycles controlled by respective instructions.

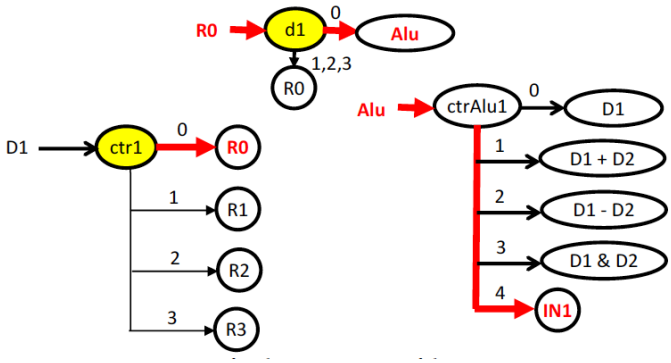


Figure 41 – HLDDs for the components of the VLIW processor

In the joint HLDD, the register variables R_i represent both the current state (as part of the stimuli, for cycle $\tau - 1$) and the next state (as a response to the stimuli, for cycle τ) of the processor. Formally, the register nodes R_i embody in the joint HLDD model two roles: they are interpreted as roots of HLDDs (to model the destinations of Write (WR) operations for cycle $\tau - 1$), and as terminal nodes of HLDDs (to model the sources for Read (RD) operations for cycle τ).

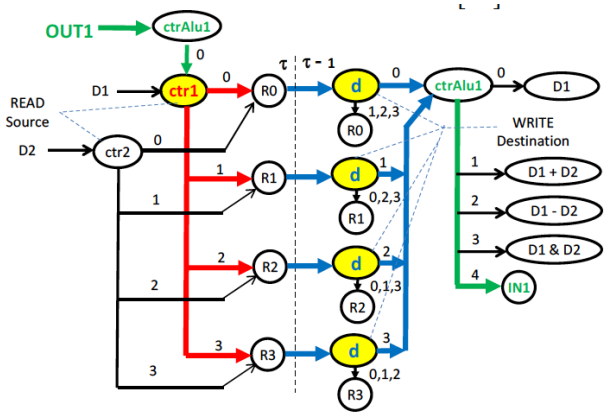


Figure 42 – Joint cycle-based HLDD model for slot 1 of the VLIW processor

For testing multiple faults in logic-level circuits, the concept of test pair was introduced. The goal of the low-level test pair as exhaustive test of a node m in the Boolean case of SSBDDs, where $|V(x(m))| = 2$, $V(x(m)) = \{0,1\}$, was to prove that the wire in the circuit represented by the node m is fault-free.

In [121, 122], the idea of test pairs was extended to the concept of test groups to improve the robustness of test with regards to multiple faults. Similarly, the high-level test of the node m in the HLDD can be regarded as an exhaustive test of the sub-circuit (instead of a wire) represented by the node m. The number of test patterns in the exhaustive test in the high-level case will be $|V(x(m))| \geq 2$.

To examine the high-level multiple fault-masking phenomenon in digital systems using HLDD-based topological view on the mutual interaction of faults, we denote an intra-node fault related solely to the node m by $r(m)$ and an inter-node fault related to nodes mi and mj by $r(mi \rightarrow mj)$.

In Fig.43 we present a sub-graph of the HLDD presented in Fig.42. Assume the

register variables $R0$ and $R1$ are initialized, we can construct a test sequence of two instructions; T1: $WR(R0)$, $RD(R0)$ (write $R0$ and read $R0$, respectively), which assigns the values $ctr1 = 0$, $d = 0$, $ctrAlu1 = 4$, and activates in the HLDD the (red) paths $l1,0(R0, IN1)$ and $l2,0(ctr1, R0)$ in the 1st and 2nd instruction cycles respectively, resulting in the data transfer $D1 \leftarrow IN1$. The test T1 is able to detect directly the intra-node control faults $r(ctr1)$, $r(d0)$, $r(ctrAlu1)$, assuming the data satisfy the constraints (5).

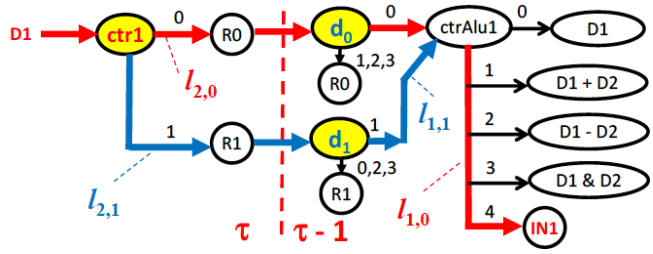


Figure 43 – Sub-graph of Joint cycle-based HLDD model for slot 1 of the VLIW processor

On the other hand, the same test $T1$ may activate other inter-node faults which will not propagate to the output $D1$, but may affect the register $R1$ and remain latent. For example, the instruction $WR(R0)$ may activate an inter-node fault $r(d0 \rightarrow d1)$ and the instruction $RD(R0)$ may cause another inter-node fault $r(ctr1 \rightarrow d1)$, both resulting in the unintended overwriting of the content of register $R1$. For detecting these faults we need the second test $T2 : RD(R1)$ which assigns $ctr1 = 1$ and activates the path $l2,1(ctr1, R1)$.

The target of the test pair $(T1, T2)$ described above was to prove the correct behavior of the node $ctr1$ at values $ctr1 = 0$ and $ctr1 = 1$, to allow any multiple fault activated by $(T1, T2)$. In a similar way, we have to construct the test pairs for testing ctr1 at other values $v \in V(ctr1)$ to fully prove the correct behavior of $ctr1$.

Definition 6.1. *Let us call the instruction sequence $T(m, v)$, which consists of initialization, execution of the operation evoked by $x(m) = v, v \in V(x(m))$ and observation steps, as a partial test for the node m at the value $x(m) = v$.*

Example 6.1.1. *The test sequence $T1 = WR(R0)$, $RD(R0)$ for the HLDD in Fig.43 is equivalent to both cases: $T(ctr1, 0)$ and $T(d1, 0)$. For $T(ctr1, 0)$, $WR(R0)$ has the role of initialization, and $RD(R0)$ has the role of both execution and observation. For $T(d1, 0)$, $WR(R0)$ has the role of initialization and execution while $RD(R0)$ has the role of observation.*
A test sequence $T = WR(R1), WR(R2), OP(R0 = R1 + R2), RD(R0)$ represents a test $T(ctrAlu1, 1)$.

Definition 6.2. *Let us call the instruction sequence $T(m, v_i, v_j) = (T(m, vi), T(m, v_j))$, $i \neq j$, where ($T(m, v_i)$ and $T(m, v_j)$) differ only in the value of $x(m) = v$, $v \in V(x(m))$, as a partial test pair for the node m at the values $x(m) = v_i$ and $x(m) = v_j$.*

Example 6.2.1. *A test $T = WR(R0)$, $RD(R0)$, $RD(R1)$ for HLDD in Fig.43 can be regarded as the partial test pair $T(ctr1, 0, 1)$, whereas the test $T = WR(R0)$, $WR(R1)$, $RD(R0)$ can be regarded as a partial test pair $T(d1, 1, 0)$.*

Lemma 6.1. *A passed test pair $T(m, v_i, v_j)$ is a proof that the control signal $x(m) = v_i$ or (v_j) is acting fault free, and that it does not produce any change of the system state influencing on the operation controlled by $x(m) = v_j$ (or v_i).*

Proof. The first statement of Lemma results from Definition 6.1 and the second statement from Definition 6.2. □

Definition 6.3. *Let us call the collection of all test pairs* $T(m,v_i,v_j)$ *for all pairs of* $v_i,v_j \in V(x(m))$, $v_i \neq v_j$ *as a test group* $T(m)$ *for the node* m.

Theorem 6.1. *The test group T(m) is sufficient to prove that the sub-circuit represented by the node m is fault free.*

Proof. The proof follows from Definition 6, and from applying the 1st statement of Lemma 6.1 for each value of $v \in V(x(m))$, and the 2nd statement for each pair of values $v_i,v_j \in V(x(m))$, $v_i \neq v_j$. If there will be a fault related to any other node involved in this test group, the fault will always be detected in accordance with the test pair concept, stating that the second test pattern of a test pair will detect the fault which has eventually masked the fault during the first test pattern [121, 122]. □

Example 6.1.1. *Let us generate the full test for all of the control nodes in the HLDD in Fig.42. First, the test group T(ctr1) (and similarly for ctr2) will be as follows:*

1. $WR\uparrow$ *(initialization of all registers* R_i*)*
2. *For all* R_i*:*$WR(R_i)$, $RD(R_i)$, *{For all* R_j, $(j \in i)$*: RD* (R_j)*}*

The length of the test is $L(ctr1) = n+3n(n-1) = 3n^2-2n$.

The test groups for all d_i *can be joined in a single* $T(d)$. *Since ctr1 is proved already as fault free by* $T(ctr1)$, *the test group can be shortened as follows:*

1. $WR\uparrow$
2. *For all* $R_i : WR(R_i)$, $RD\uparrow$.

Here $RD\uparrow$ *denotes reading of the contents of all registers. The length of the test is* $L(d) = n+n(n+1) = n^2+2n$, *less than* $3n^2-2n$. *The full test length for testing the nodes related to the access of the registers is* $L(ctr1,d) = 4n^2$.

When generating the test group for ALU control, represented by the node $ctrAlu1$ *in Fig.42. we can take into account that the nodes* $ctr1$, $ctr2$ *and* d *are tested and hence, proved as fault free. This knowledge allows us to simplify the test group of Theorem 6.1, as follows:*

1. $WR\uparrow$
2. *For all* $v \in V(ctrAlu1) : WR\uparrow_v, ALU_v, RD(R_v)$.

Here ALU_v *means execution of the ALU operation under control* $ctrAlu1 = v$, *and the result is stored in any of the available registers* R_v. $WR\uparrow_v$, *means initialization of the subset of registers to store the arguments of the operation* ALU_v. *The length of the test depends on the number of arguments of each operation controlled by* $ctrAlu1$.

The described concept of multiple fault testing can be applied to the test generation methods described in this thesis.

6.5 Introducing the result of the thesis into engineering education

As part of this study, we propose an extension of the current TEAM [123] teaching environment and new research scenarios as an extension to the previous laboratory works based on manual test program generation. In the proposed research scenarios, the focus is on high-level generated test data reasoning such as fault coverage (as compared to low-level fault coverage), identification of fault redundancy (as opposed to low-level redundancy), and investigation of pseudo-exhaustive test capabilities for complex operations.

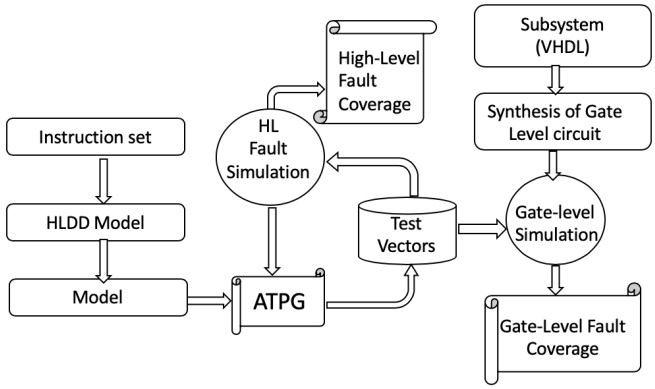


Figure 44 – Research environment for teaching high-level test

Fig.44 depicts the proposed research environment as a task flow diagram and consists of the following:

1. ATPG for conformity test high-level test data generation, which satisfy constraints (4 and 5),
2. Fault simulator for evaluating the conformity test high-level fault coverage, and revealing the redundant fault candidates (for creative analysis and proof of the redundancy of faults)
3. Low-level fault simulator (for low-level fault coverage evaluation, and for analysis of mapping between high- and low-level faults).

In this environment, we use the following tools: Mentor Graphic Modelsim, Python compiler, logic and fault simulator of Turbo Tester [124]. This implementation currently runs on Linux and Windows operating system.

The laboratory workflow diagram is illustrated in Fig.45 with the list of experiments to be performed by students in a holistic view. The purpose of the laboratory work is not only to teach students about high-level test generation and diagnosis, but also to provide them with practical training and experience. Students will have experience in synthesizing test programs for processor sub-circuit control parts to evaluate test quality. Students will also learn how to prove the redundancy of faults at high level. As shown in Fig.45, the laboratory work is divided into sub-tasks to train students on high-level test generation. Students do not have to struggle with huge amounts of details in manual or language-based information.

The main purpose of this task is to give the students a basic understanding of the distinctions and peculiarities of high-level and low-level test synthesis and analysis, and

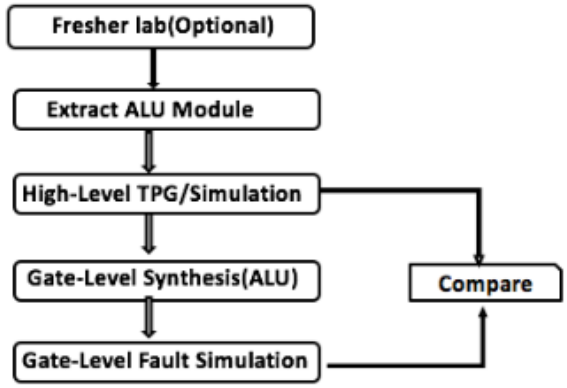


Figure 45 – High-Level Test Pattern Generation Lab Scenario

to help them realize that high-level and hierarchical modelling methods are the only way to cope with the complexities of today's and tomorrow's digital systems in synthesizing test programs and tools.

6.6 Summary

In this chapter, the following contributions have been highlighted:

- A commercial/in-house tool-based SBST synthesis environment for carrying out the experiments with all methods and algorithms developed in this thesis was presented.
- The SBST concept developed in this thesis represents a novel method of test compaction which is to be unrolled during execution of the test.
- The SBST concept is based on the novel architecture of the compacted test which is represented as a structure composed of the sets of test templates, instructions and test data.
- A method of constructing the test program for microprocessors for detecting multiple faults and avoiding fault masking, based on using the HLDD model was proposed for the first time.
- A novel educational tool environment was developed based on the results of this thesis. The efficiency of such environment was investigated at Tallinn University of Technology in teaching of masters students by involving them in the laboratory research as part of the Design and Test course.

7 Experimental Results

To show the efficiency of our method, we experimented with two processors; VLIW and a MIPS-like processor. The objectives of the experiments were ALU unit of VLIW [125] and different units of MIPS-like [111], like execute, forwarding and branch control sub-circuits. We carried out experiments on an Intel Core i7 processor at 3.4GHz and 8GB of RAM.

For the VLIW processor, we investigated the speed and quality of random test data search to test the execute unit's control part. We investigated two search algorithms which are pure-random and optimized greedy-random. In Table 11, we show the result of the experiment. Here it is shown that both approaches achieve 100% fault coverage at high-level and 99.34% fault coverage at low-level faults. However, we have proven that any low-level faults not detected by a set of data that achieved 100% fault coverage at high-level must be redundant. From the table, we also showed that the greedy method guarantees fewer test data at a trade-off of generation time. The choice of which method to choose is left to the users.

Table 11 – Comparison of control test algorithms

Method	Test-length, #instructions	Fault Coverage		Time (s)
		High-level faults	SAF	
RANDOM	204	100%	99.34%	2.00
GREEDY	139	100%	99.34%	7.85

The relationships between the fault coverage and test length is illustrated by the curves in Fig.46, and the dependence of test generation time on the size of search space (number of random test candidates) is illustrated in Fig.47.

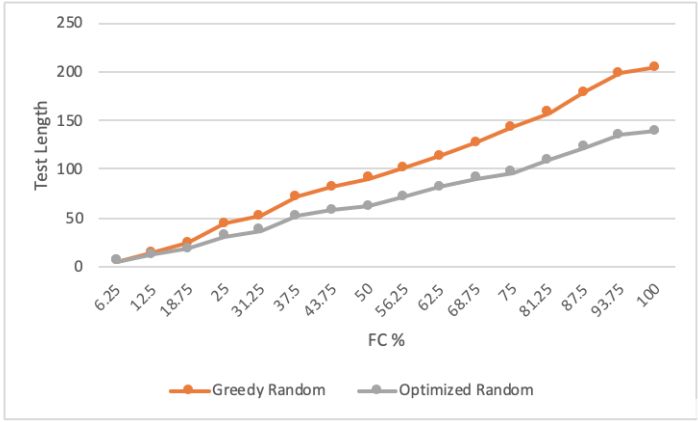


Figure 46 – Dependence of the high-level control fault coverage on test length

For the MIPS-like processor, we carried out more tests which include the forwarding unit, register decoders and execute unit. Fig.48 shows a distribution of all the faults in the processor.

The execute unit takes 70% of the total number of faults in the processor. It consists of the adder, two multiplication modules(MULT0 and MULT1) and interconnections. Our designed test program based on the method described in this thesis was applied to target the faults in the execute unit. The high-level test was simulated by a commercial

Figure 47 – MIPS-like processor Fault Distribution

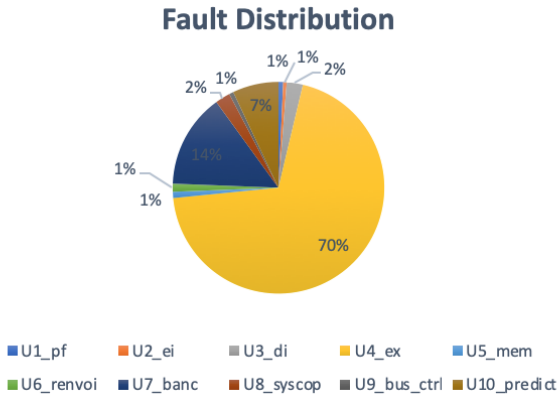


Figure 48 – MIPS-like processor Fault Distribution

tool to grade the gate-level SAF coverage. To evaluate the efficiency of the high-level ATPG, we used commercial gate-level ATPG tool for comparison.

Table 12 – Execute Unit Test

Method	Experiments		#Faults	FC(%)	Stored Patterns	Executed Patterns	ATPG Time
Proposed high-level method	High-level ATPG		756	100	166	4818	$47s$
	Gate-level Simulation	Adder	2516	99.92			
		MULT0	95188	99.52			
		MULT1	91810	99.16			
Commercial gate-level ATPG		Adder	2516	99.96	957	957	$1h34min$
		MULT0	95188	97.40			
		MULT1	91810	97.71			

Table 12 shows the result of this evaluation, where it could be easily seen that the gate-level SAF coverage, achieved by the proposed ATPG for the whole module under test, is better than that achieved by the commercial tool. Also, the time cost of the proposed method is less than that of the commercial tool.

In Table .13, we show the fault coverage and simulation time of test for the forwarding unit (FU). The second column shows the result of fault simulation for test generated

only for the ALU. It became obvious that there is a need for specific tests for the forwarding unit. We further generated a dedicated test only for the FU, the result of this is presented in the third column, which shows to cover 8% more faults, compared to the ALU test. We combined both the MUX-based ALU control test, with the dedicated FU test. The coverage was recorded in the fourth column, giving an improvement of 8.32% compared to only the ALU test.

Table 13 – Fault coverage of forwarding unit by different tests

Module/Unit	ALU Test(%)	Forwarding Test	Combined (%)	Improvement (%)
Forwarding Unit	89.71	97.84	98.03	8.32
Time(s)	808	48	460	

The tests for the FU were generated, without knowing gate-level implementation detail, we only relied on the general information of the MIPS-like pipeline architecture, which includes the number of stages and forwarding paths.

Table 14 – Targeted modules comparison with other methods

Module/ unit	#faults	Gate-level implementation details are exploited		Gate-level implementation independent	
		ATIG [98]	SBST [126]	SBST [37]	Proposed
ALU	203576	98.67%	n.a	97.85%	99.06%
PPS_EX	21136	97.62%	96.20%	84.12%	98.37%
Forward	3738	99.00%	99.68%	93.64%	98.03%
Register Banc	43584	99.90%	100%	99.98%	99.99%
Syscop	6930	93.60%	98.04%	87.90%	87.65%

In Table 14, the results of this research are compared with three state-of-the-art approaches or methods for three different MIPS-like processor modules: ALU, PPS EX(Execute Unit), and Forwarding Unit. In the sense that gate-level implementation specifics are not needed, the approach presented in this thesis is similar to [37], but it shows nearly 5% improvement in fault coverage compared to [37] While the method in [98] shows an improvement of 1% over the proposed method, details of implementation are needed. The method in [126] requires that constraints be applied during the generation of ATPG tests, requiring gate-level information as well. Here, the result of our method shown for the forwarding unit and the system co-processor doesn't consider the untestable faults in the modules. In Table 15, we show the results of these units when the proven untestable faults have been removed from the fault list.

Table 15 shows the result of the proposed method for the full processor. We have partitioned the test into several categories. The first test in column 2 targets the register decoder. We measured the impact of this test in relations to other modules in the microprocessor. As you could see, the fault coverage of the execute unit is ridiculously low as expected. In column 3, we experimented with the MUX-based approach for the ALU and measured the impact on other modules in a similar way as the register decoder test. In column 4, we combined the ALU MUX-based test with the register decoder test. The impact on the fault coverage is readily seen in execute module and the register decoder modules. The MUX-based approach, as mentioned earlier, was not enough to cover the faults in the forwarding unit. To take care of this limitation, we therefore produced a dedicated test of the forwarding unit combined with the ALU

Table 15 – Fault coverage of whole processor by different tests

Module/ Unit	Register Test %	ALU Test %	ALU+Registe Test %	ALU+Register+ FU Test %	ALU+Register+ FW+Syscop %
U1_pf	65.38	69.19	69.33	69.42	69.6
U2_ei	78.67	85.072	85.07	85.01	85.07
U3_di	73.27	85.70	85.93	86.04	86.52
U4_ex	5.29	98.32	98.35	98.34	98.35
U5_mem	58.46	75.05	76.26	76.84	76.84
U6_renvoi	65.56	86.03	89.20	97.89	98.03
U7_banc	98.16	37.51	99.34	99.85	99.85
U8_syscop	47.56	62.71	64.35	64.34	87.65
U9_bus_ctrl	84.21	80.61	86.19	86.58	86.58
U10_predict	37.55	58.54	58.87	58.99	58.93
Total	26.09	84.96	94.06	94.26	94.70

and the register test as shown in column 5. This shows almost 10% improvement in fault coverage for this unit. In column 6, an attempt was made to generate test for the system co-processor in addition to the test in column 5 which further improves the coverage for the forwarding unit.

Table 16 – Fault coverage of whole processor in comparison with other methods

	Module/Unit	ATIG(%) [98]	SBST(%) [126]	SBST(%) [37]	Proposed Method(%)
Without Prediction	U1_pf	98.32	91.97	86.32	70.00
	U2_ei	99.71	96.82	90.86	85.50
	U3_di	95.28	92.45	90.24	89.70
	U4_ex	97.62	96.20	97.85	98.68
	U5_mem	83.41	71.29	81.87	90.63
	U6_renvoi	99.00	99.68	93.64	98.50
	U7_banc	99.90	100	99.98	99.99
	U8_syscop	93.60	98.04	87.90	93.53
	U9_bus_ctrl	92.62	92.20	93.95	89.78
	Total	97.52	97.46	95.08	98.03
With Prediction	U10_predict	96.01	99.34	–	59.19
	Total	97.31	97.46	95.08	95.30

In Table 16, we compared our approach with state-of-the-arts methods in terms of test for the full processor. Using the information about untestable faults in [126], we removed the identified and proven untestable faults from the fault list. The result shows that the approach presented in this thesis covers more faults in the processor with no knowledge of the implementation details. However, there are still some faults undetected in the modules. For example, the fault coverage for the forwarding unit with respect to the testable faults in the module stood at about 98.50%. The reason for the undetected 1.50% faults is due to signals such as the interrupt signal and exceptions that can not be activated using a functional method. [97] identified 39 of the total fault list for the forwarding unit to belong to this category of undetectable faults.

The approach presented in this research work doesn't target the branch prediction unit since faults in the unit do not lead to any functional incorrectness, but performance

cost, which is usually two or more clock cycles, depending on the architecture [127]. Therefore, branch prediction units are hard to test by functional methods without having dedicated observation points and the same applies to our approach. The approach used for testing the branch prediction unit in [126] is based on using the algorithm mostly used in memory testing. We can easily adapt this to our test. To have a fair comparison, we have decided to exclude the faults related to the branch prediction unit since also the result in [37] is based on implementation of miniMIPS without this unit.

8 Conclusions

In this thesis, a new high-level and implementation-independent test program generation method for modules of RISC processors is proposed with improved quality compared to the known methods. The higher quality is achieved through expansion of the fault class covered.

The proposed approach is based on the introduction of a new high-level functional control fault model for testing the control parts of the processor functional modules in combination with the implementation-independent and fault model-free testing of the data parts of the Module Under Test (MUT) with the pseudo-exhaustive test patterns. The high quality of the test programs was theoretically proven, and the proof was confirmed by experimental results, where the quality of the experiments was measured in relation to the SAF class and the TDF class. It has been shown that the proposed high-level control fault model covers a broader class of structural faults, including conditional SAF, shorts, multiple SAFs and TDFs without the need to specifically list these faults.

A correlation has been shown between the sequential March test used for memory testing and the combinational test for decoding circuits in processor logic modules. This allowed proving that the developed test generation method can detect a larger high-level functional fault class similar to faults detected with the March test in addressing logic. The data constraint-based fault model and the introduced analogy of testing with March test flow for memories revealed the possibility of applying the proposed approach, not only for the combinational MUTs but also for sequential ones.

A high-level fault coverage metric and a high-level fault simulation tool have been developed to support the proposed test generation methodology. This has made it possible to develop a novel mixed-level method for the identification of high-level functional fault redundancy and low-level structural fault redundancy.

A further added benefit of the proposed method was discovered during experimental work in the generation of test data for RISC processors. Test programs created specifically for testing only control parts of the modules achieve very high fault coverage for the data part as well. This is due to the influence of the novel data constraints introduced for the selection of data operands.

SAF coverage metric was used to compare the findings of this research with state-of-the-art methods. Experimental results indicate higher SAF coverage compared to other current implementation-independent test generation methods for microprocessors. This shows that the proposed method is an important step forward compared to the state-of-the-art methods for providing accurate and safe information on the quality of the test programs produced.

The miniMIPS and VLIW microprocessors were used as a case study for demonstrating the effectiveness of the proposed High-Level Implementation-Free Functional SBST method. Pure-random and greedy algorithms were implemented for generating high-level test data. In the VLIW microprocessor, we were able to cover all the high-level faults (100% fault coverage) of the ALU module, while at low-level, 99.34% of SAF fault was detected. However, any low-level faults not detected by a set of data that achieved 100% fault coverage at high-level must be redundant. The two algorithms present a trade-off choice to users in the sense that the greedy method guarantees fewer test data at a trade-off of generation time.

The test data generation algorithms were tested against the miniMIPS microprocessor in comparison with commercial ATPG. Test data which detects 100% of high-level faults of the ALU, detects 99.92%, 99.52% and 99.16% SAF of Adder, MULT0 and MULT1

sub-modules of the miniMIPS ALU respectively. The advantage of this in comparison with commercial ATPG is not only in higher SAF coverages but also in the time cost of test data generation. Due to redundancy proofs, the low-level faults not detected are also redundant for this microprocessor.

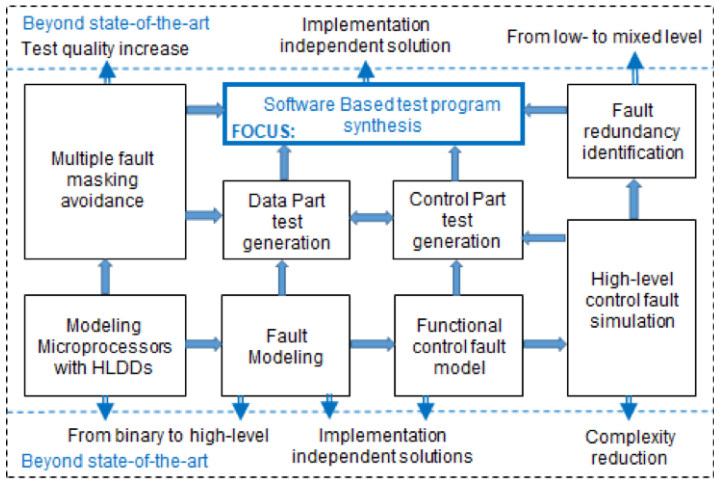


Figure 49 – The big picture

The results of this research, in comparison with state-of-the-art approaches for miniMIPS microprocessor, shows that while implementation-details are not known, high fault coverage of SAF faults can be achieved. When compared with [37] where gate-level implementation specifics are not needed, the result of this research shows about 4% coverage improvement when the prediction unit is not considered. While the results of [98] and [126] show about 2% improvement over the method proposed in the thesis for the full processor, gate-level information is needed.

Fig. 49 gives an overview and summary of the contributions of this thesis beyond the state-of-the-art. Each of the aspects of the contributions is explained in detail in each of the chapters of the thesis. This highlights the main issues addressed in this thesis, ranging from high-level fault modelling to high-level test data generation, fault redundancy identification and high-level fault simulation.

List of Figures

1 Network of computing nodes for the instruction set in Table 1 24
2 HLDDs for the processor described in Table 1 25
3 MIPS-like processor data-path and pipeline 26
4 A part of a RISC type microprocessor with executing unit in the pipeline and data forwarding environment 27
5 HLDD for Register Decoders of a MIPS-like Microprocessor 27
6 HLDD for MIPS-like Microprocessor 28
7 Illustration of different corruptions of the HLDD by faults in MP 29
8 Control Path TTs, HLDD for a microprocessor with 8 instructions and 3 op-code fields 31
9 TT split by selection of x_2 as the root variable in the HLDD 32
10 TT split by selection of x_3 as the root variable in the HLDD 32
11 Minimized HLDD model for the MIPS-like microprocessor 33
12 Not-minimized HLDD model for the MIPS-like microprocessor 33
13 HLDD with a single decision node for representing 20 MiniMIPS instructions 34
14 HLDD for a subset of instructions of MiniMIPS 35
15 Test Execution setup 38
16 Illustration of the proposed test concept 39
17 Generic DNF based control structure of the executing unit 40
18 Unrolled test execution evolving in time 45
19 Illustration of the March test for memories 46
20 Functional control fault classes CL 1 – CL 4 47
21 Fault collapsing relationships 48
22 Transition delay fault testing in modules under test 48
23 Example of redundancy proofs with 1-bit truth table 50
24 Example of testing the pipeline forwarding unit 51
25 Examples of testing the pipeline forwarding unit 51
26 Process of multiplying 55
27 Segmentation of multiplier 55
28 High-level Test Generation Big Picture 58
29 High-level Test Data Generation 59
30 HLDD for ALU in Table.10 60
31 High-level Test Generation 60
32 Test Framework 61
33 SBST Test program flow [30] 62
34 MIPS-like processor instruction format 63
35 Generalization of instruction format of MIPS-like processors 63
36 Test program templates 64
37 Mapping between HLDD and Instruction Type 64
38 Conformity Template Variations 65
39 Architecture of the test program 65
40 Structural Representation of the VLIW processor 66
41 HLDDs for the components of the VLIW processor 67
42 Joint cycle-based HLDD model for slot 1 of the VLIW processor 67
43 Sub-graph of Joint cycle-based HLDD model for slot 1 of the VLIW processor 68
44 Research environment for teaching high-level test 70

45 High-Level Test Pattern Generation Lab Scenario 71
46 Dependence of the high-level control fault coverage on test length 72
47 MIPS-like processor Fault Distribution 73
48 MIPS-like processor Fault Distribution 73
49 The big picture 78

List of Tables

1 Instruction set of a microprocessor 24
2 Example of scalabilities for three versions of HLDDs 36
3 Example of a high-level control test 42
4 Example of a High-Level Fault Table 44
5 Test generation for pipeline forwarding unit 51
6 Pseudo-exhaustive test data for addition operation 53
7 Pseudo-exhaustive test data for subtraction operation 54
8 Pseudo-exhaustive test data generation for ripple-carry multiplier array .. 56
9 Comparison of different PET versions 56
10 Instruction subset for MIPS-like ALU 59
11 Comparison of control test algorithms 72
12 Execute Unit Test 73
13 Fault coverage of forwarding unit by different tests 74
14 Targeted modules comparison with other methods 74
15 Fault coverage of whole processor by different tests 75
16 Fault coverage of whole processor in comparison with other methods 75

References

[1] R. Ubar, S. A. Oyeniran, M. Scholzel, and H. T. Vierhaus, "Multiple fault testing in systems-on-chip with high-level decision diagrams," *2015 10th International Design Test Symposium (IDT)*, pp. 66–71, Dec 2015.

[2] A. S. Oyeniran, U. E. Odozi, and R. Ubar, "A new measure for calculating multiple fault coverage of microprocessor self-test," in *2016 15th Biennial Baltic Electronics Conference (BEC)*, pp. 75–78, Oct 2016.

[3] A. S. Oyeniran, A. Jasnetski, A. Tsertov, and R. Ubar, "High-level test data generation for software-based self-test in microprocessors," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, June 2017.

[4] A. S. Oyeniran, R. Ubar, S. P. Azad, and J. Raik, "High-level test generation for processing elements in many-core systems," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, July 2017.

[5] A. S. Oyeniran and R. Ubar, "High-level functional test generation for microprocessor modules," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pp. 356–361, June 2019.

[6] A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "High-level combined deterministic and pseudo-exhuastive test generation for risc processors," in *2019 IEEE European Test Symposium (ETS)*, pp. 1–6, May 2019.

[7] A. S. Oyeniran, S. P. Azad, and R. Ubar, "Parallel pseudo-exhaustive testing of array multipliers with data-controlled segmentation," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018.

[8] A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "Mixed-level identification of fault redundancy in microprocessors," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, March 2019.

[9] A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors," *Journal of Electronic Testing*, vol. 36, no. 1, pp. 87–103, 2020.

[10] A. S. Oyeniran, S. P. Azad, and R. Ubar, "Combined pseudo-exhaustive and deterministic testing of array multipliers," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, 2018.

[11] A. Jasnetski, S. A. Oyeniran, A. Tsertov, M. Schölzel, and R. Ubar, "High-level modeling and testing of multiple control faults in digital systems," in *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–6, 2016.

[12] S. Payandeh Azad, A. S. Oyeniran, and R. Ubar, "Replication-based deterministic testing of 2-dimensional arrays with highly interrelated cells," in *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 21–26, 2018.

[13] A. S. Oyeniran, R. Ubar, and M. Kruus, "Teaching digital system test," in *2017 27th EAEEIE Annual Conference (EAEEIE)*, pp. 1–6, 2017.

[14] L. Jürimägi, R. Ubar, M. Jenihhin, J. Raik, S. Devadze, and A. S. Oyeniran, "Application specific true critical paths identification in sequential circuits," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 299–304, 2019.

[15] A. S. Oyeniran and R. Ubar, "High-level functional test generation for microprocessor modules," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pp. 356–361, 2019.

[16] S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur, J. Raik, G. Jervan, R. Ubar, and T. Hollstein, "From online fault detection to fault management in network-on-chips: A ground-up approach," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 48–53, 2017.

[17] R. Ubar, A. S. Oyeniran, and O. Medaiyese, "Minimization of the high-level fault model for microprocessor control parts," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, pp. 1–4, 2018.

[18] A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "Implementation-independent functional test generation for risc microprocessors," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 82–87, 2019.

[19] R. Ubar, L. Jürimägi, E. Orasson, G. Josifovska, and S. A. Oyeniran, "Double phase fault collapsing with linear complexity in digital circuits," in *2015 Euromicro Conference on Digital System Design*, pp. 700–705, 2015.

[20] R. Ubar and S. A. Oyeniran, "Multiple control fault testing in digital systems with high-level decision diagrams," in *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, 2016.

[21] L.-T. Wang, C.-W. Wu, and X. Wen, "Vlsi test principles and architectures. design for testability," *Elsevier*, 2006.

[22] P. Georgiou, X. Kavousianos, R. Cantoro, and M. S. Reorda, "Fault-independent test-generation for software-based self-testing," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pp. 79–84, July 2018.

[23] R. Ubar, S. Kostin, and J. Raik, "How to prove that a circuit is fault-free?," in *2012 15th Euromicro Conference on Digital System Design*, pp. 427–430, Sep. 2012.

[24] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic bist for large industrial designs: real issues and case studies," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pp. 358–367, Sep. 1999.

[25] Li Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 369–380, March 2001.

[26] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial atpg," in *Proceedings International Test Conference 1997*, pp. 743–752, Nov 1997.

[27] Li Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pp. 548–553, June 2003.

[28] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, pp. 461–475, April 2005.

[29] S. Guramurthy, S. Vasudevan, and J. A. Abraham, "Automated mapping of pre-computed module-level test sequences to processor instructions," in *IEEE International Conference on Test, 2005.*, pp. 10 pp.–303, Nov 2005.

[30] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, pp. 4–19, May 2010.

[31] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. 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)*, pp. 299–304, Oct 2014.

[32] 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)*, pp. 1462–1467, March 2013.

[33] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. S. Reorda, and B. Becker, "An effective approach to automatic functional processor test generation for small-delay faults," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[34] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 3055–3066, Oct 2016.

[35] M. Schölzel, T. Koal, S. Röder, and H. T. Vierhaus, "Towards an automatic generation of diagnostic in-field sbst for processor components," in *2013 14th Latin American Test Workshop - LATW*, pp. 1–6, April 2013.

[36] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Design Test of Computers*, vol. 21, pp. 102–109, March 2004.

[37] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 1441–1453, Nov 2008.

[38] E. Sanchez and M. S. Reorda, "On the functional test of branch prediction units," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 1675–1688, Sep. 2015.

[39] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 393–398, July 2006.

[40] Zhuo Li, Xiang Lu, Wangqi Qiu, Weiping Shi, and D. M. H. Walker, "A circuit level fault model for resistive opens and bridges," in *Proceedings. 21st VLSI Test Symposium, 2003.*, pp. 379–384, May 2003.

[41] H. K. Lee and D. S. Ha, "Soprano: an efficient automatic test pattern generator for stuck-open faults in cmos combinational circuits," in *27th ACM/IEEE Design Automation Conference*, pp. 660–666, June 1990.

[42] A. Kristic and K. Cheng, *Delay Fault Testing for VLSI Circuits*. Dordrecht, The Netherlands, Kluwer Academic Publishers, 1998.

[43] J. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop*, vol. 10, pp. 278–291, July 1966.

[44] R. D. S. Blanton and J. P. Hayes, "On the properties of the input pattern fault model," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, pp. 108–124, Jan. 2003.

[45] K. Keller, "Hierarchical pattern faults for describing logic circuit failure mechanisms," in *US Patent 5546408*, August 1994.

[46] R. Ubar, *Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations*, vol. 40. Plenum Publishing Corporation, USA, Nov. 1980.

[47] J. Raik, R. Ubar, J. Sudbrock, W. Kuzmicz, and W. Pleskacz, "Dot: new deterministic defect-oriented atpg tool," in *European Test Symposium (ETS'05)*, pp. 96–101, May 2005.

[48] U. Mahlstedt, J. Alt, and I. Hollenbeck, "Deterministic test generation for non-classical faults on the gate level," in *Proceedings of the Fourth Asian Test Symposium*, pp. 244–251, Nov 1995.

[49] S. Holst and H. Wunderlich, "Adaptive debug and diagnosis without fault dictionaries," in *12th IEEE European Test Symposium (ETS'07)*, pp. 7–12, May 2007.

[50] Kyoung Youn Cho, S. Mitra, and E. J. McCluskey, "Gate exhaustive testing," in *IEEE International Conference on Test, 2005.*, pp. 7 pp.–777, Nov 2005.

[51] A. Jas, S. Natarajan, and S. Patil, "The region-exhaustive fault model," in *16th Asian Test Symposium (ATS 2007)*, pp. 13–18, Oct 2007.

[52] K. N. Dwarakanath and R. D. Blanton, "Universal fault simulation using fault tuples," in *Proceedings 37th Design Automation Conference*, pp. 786–789, June 2000.

[53] M. Psarakis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Sequential fault modeling and test pattern generation for cmos iterative logic arrays," *IEEE Transactions on Computers*, vol. 49, pp. 1083–1099, Oct 2000.

[54] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast, "Cell-aware test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, pp. 1396–1409, Sep. 2014.

[55] L. Shen and S. Su, "A functional testing method for microprocessors," *IEEE Transactions on Computers*, vol. 37, no. 10, pp. 1288–1293, 1988.

[56] M. Hansen and J. Hayes, "High-level test generation using physically-induced faults," *VLSI Test Symposium*, pp. 20–28, 1995.

[57] L. Vandeventer and J. Santucci, "Algorithms for behavioral test pattern generation from vhdl circuit descriptions containing loop language constructs," *Conference on European Design Automation*, 1994.

[58] J. Santucci, A. Courbis, and N. Giambiasi, "Behavioral testing of digital circuits," *Journal of Microelectronics Systems Integration*, vol. 1, no. 1, pp. 55–77, 1993.

[59] M. O'Neil, D. Jani, C. Cho, and J. Armstrong, "Btg: A behavioral test generator," *9th International Symposium On CHDLs*, 1989.

[60] C. Cho and J. Armstrong, "A behavioral test generation algorithm," *International Test Conference*, 1994.

[61] V. Kumar, M. Jeelani, A. Mulai, and A. Shandilia, "Employing functional analysis to study fault models in vhdl," *International Journal of Scientific Engineering and Technology*, vol. 1, no. 5, pp. 207–208, 2012.

[62] Y. Joannon, V. Beroulle, C. Robach, S. Tedjini, and J.-L. Carbonero, "Choice of a high-level fault model for the optimization of validation test set reused for manufactoring test," *Hindawi VLSI Design*, 2008.

[63] A. K. Gupta and J. R. Armstrong, "Functional fault modeling and simulation for vlsi devices," in *22nd ACM/IEEE Design Automation Conference*, pp. 720–726, June 1985.

[64] S. Thatte and J. Abraham, "Test generation for microprocessors," *IEEE Transactionson Computers*, pp. 429–441, 1980.

[65] D. Brahme and J. Abraham, "Functional testing of microprocessors," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 475–485, 1984.

[66] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "An rt-level fault model with high gate level correlation," in *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No.PR00786)*, pp. 3–8, Nov 2000.

[67] V. A. P. Thaker and M. Zaghloul, "Rt level modeling and test evaluation techniques for vlsi circuits," in *ITC*, 2000.

[68] A. Fin and F. Fummi, "A vhdl error simulator for functional test generation," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 390–395, March 2000.

[69] P. C. Ward and J. R. Armstrong, "Behavioral fault simulation in vhdl," in *27th ACM/IEEE Design Automation Conference*, pp. 587–593, June 1990.

[70] S. Ghosh and T. J. Chakraborty, "On behavior fault modeling for digital designs," *Journal of Electronic Testing*, vol. 2, pp. 135–151, Jun 1991.

[71] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[72] Tonysheng Lin and S. Y. H. Su, "The s-algorithm: A promising solution for systematic functional test generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, pp. 250–263, July 1985.

[73] G. Buonanno, F. Ferrandi, L. Ferrandi, F. Fummi, and D. Sciuto, "How an "evolving" fault model improves the behavioral test generation," in *Proceedings Great Lakes Symposium on VLSI*, pp. 124–129, March 1997.

[74] R. Ramchandani and D. Thomas, "Behavioral test generation using mixed integer non-linear programming," *IEEE International Test Conference*, 1994.

[75] F. Ferrandi, G. Ferrara, D. Sciuto, A. Fin, and F. Fummi, "Functional test generation for behaviorally sequential models," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 403–410, March 2001.

[76] Levendel and Menon, "Test generation algorithms for computer hardware description languages," *IEEE Transactions on Computers*, vol. C-31, pp. 577–588, July 1982.

[77] A. Jasnetski, *Software-Based Self-Test for Microprocessors with High-Level Decision Diagrams*. PhD thesis, Tallinn University of Technology, 2018.

[78] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 393–398, July 2006.

[79] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. S. 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)*, pp. 1–6, May 2012.

[80] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Proceedings. International Test Conference*, pp. 590–598, Oct 2002.

[81] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, pp. 34–40, April 1999.

[82] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante, "On the test of microprocessor ip cores," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 209–213, March 2001.

[83] C. H. . Wen, L. . Wang, Kwang-Ting Cheng, Kai Yang, Wei-Ting Liu, and Ji-Jan Chen, "On a software-based self-test methodology and its application," in *23rd IEEE VLSI Test Symposium (VTS'05)*, pp. 107–113, May 2005.

[84] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *2006 IEEE International Test Conference*, pp. 1–9, Oct 2006.

[85] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1203–1215, Nov 2006.

[86] M. Hatzimihail, G. Xenoulis, M. Psarakis, D. Gizopoulos, and A. Paschalis, "Software-based self-test for pipelined processors: a case study," in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, pp. 535–543, Oct 2005.

[87] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 518–530, May 2007.

[88] C. Chen, C. Wei, T. Lu, and H. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 505–517, May 2007.

[89] C. H. . Wen, L. C. Wang, and Kwang-Ting Cheng, "Simulation-based functional test generation for embedded processors," in *Tenth IEEE International High-Level Design Validation and Test Workshop, 2005.*, pp. 3–10, Nov 2005.

[90] 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)*, pp. 990–999, Oct 1998.

[91] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache resident functional microprocessor testing: Avoiding high speed io issues," in *2006 IEEE International Test Conference*, pp. 1–7, Oct 2006.

[92] 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*, pp. 1–8, Oct 2014.

[93] A. Krstic, Wei-Cheng Lai, Kwang-Ting Cheng, L. Chen, and S. Dey, "Embedded software-based self-test for programmable core-based designs," *IEEE Design Test of Computers*, vol. 19, pp. 18–27, July 2002.

[94] D. Changdao, M. Graziano, E. Sanchez, M. Sonza Reorda, M. Zamboni, and N. Zhifan, "On the functional test of the btb logic in pipelined and superscalar processors," in *2013 14th Latin American Test Workshop - LATW*, pp. 1–6, April 2013.

[95] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, pp. 1030–1044, July 2011.

[96] J. P. Acle, R. Cantoro, E. Sanchez, and M. S. Reorda, "On the functional test of the cache coherency logic in multi-core systems," in *2015 IEEE 6th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, Feb 2015.

[97] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. S. 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*, pp. 52–57, Dec 2013.

[98] Y. Zhang, H. Li, and X. Li, "Automatic test program generation using executing-trace-based constraint extraction for embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 1220–1233, July 2013.

[99] J. Hudec, "An efficient adaptive method of software-based self test generation for risc processors," in *2015 4th Eastern European Regional Conference on the Engineering of Computer Based Systems*, pp. 119–121, 2015.

[100] V. M. Suryasarman, S. Biswas, and A. Sahu, "Automation of Test Program Synthesis for Processor Post-silicon Validation," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 34, no. 1, pp. 83–103, 2018.

[101] J. Zhengfei, Z. Ying, and C. Xin, "A novel on-line test scheme for avionics controller based on sbst," in *2014 International Test Conference, London, U.K*, pp. 241–246, July 2018.

[102] N. Hage, R. Gulve, M. Fujita, and V. Singh, "On testing of superscalar processors in functional mode for delay faults," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, pp. 397–402, 2017.

[103] C. Chen and J. Huang, "Reinforcement-learning-based test program generation for software-based self-test," in *2019 IEEE 28th Asian Test Symposium (ATS)*, pp. 73–735, 2019.

[104] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug 1986.

[105] R. Ubar, J. Raik, and A. Morawiec, "Back-tracing and event-driven techniques in high-level simulation with decision diagrams," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, pp. 208–211 vol.1, May 2000.

[106] R. Ubar, J. Raik, A. Jutman, M. Jenihhin, M. Brik, M. Instenberg, and H. Wuttke, "Diagnostic modeling of microprocessors with high-level decision diagrams," in *2008 11th International Biennial Baltic Electronics Conference*, pp. 147–150, Oct 2008.

[107] M. Abramovici, M. Breuer, and A. Friedman, "Digital systems testing & testable designs," *Computer Science Press*, 1995.

[108] A. Jasnetski, *Software-Based Self-Test for Microprocessors with High-Level Decision Diagrams*. PhD thesis, Tallinn University of Technology, 2018.

[109] A. Jasnetski, S. A. Oyeniran, A. Tsertov, M. Schölzel, and R. Ubar, "High-level modeling and testing of multiple control faults in digital systems," in *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 1–6, April 2016.

[110] A. van de Goor, *Semiconductor Memories: Theory and Practice*. Wiley, 1991.

[111] MiniMIPS, "https://opencores.org/projects/minimips," tech. rep.

[112] D. B. Armstrong, "On finding a nearly minimal set of fault detection tests for combinational logic nets," *IEEE Transactions on Electronic Computers*, vol. EC-15, pp. 66–73, Feb 1966.

[113] A. Miczo, *Digital Logic Testing and Simulation*. Wiley, 2003.

[114] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast, "Cell-aware test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1396–1409, 2014.

[115] R. Ubar, "Fault diagnosis in combinational circuits by solving boolean differential equations," in *Automatics & Telemechanics*, pp. 170–183, 1979.

[116] M. D. Pulukuri and C. E. Stroud, "Built-in self-test of digital signal processors in virtex-4 fpgas," in *2009 41st Southeastern Symposium on System Theory*, pp. 34–38, March 2009.

[117] A. S. Oyeniran, R. Ubar, S. P. Azad, and J. Raik, "High-level test generation for processing elements in many-core systems," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, July 2017.

[118] A. S. Oyeniran, S. P. Azad, and R. Ubar, "Parallel pseudo-exhaustive testing of array multipliers with data-controlled segmentation," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018.

[119] Yong Chang Kim, V. D. Agrawal, and K. K. Saluja, "Multiple faults: modeling, simulation and test," in *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pp. 592–597, Jan 2002.

[120] R. Ubar, S. A. Oyeniran, M. Scholzel, and H. T. Vierhaus, "Multiple fault testing in systems-on-chip with high-level decision diagrams," in *2015 10th International Design Test Symposium (IDT)*, pp. 66–71, Dec 2015.

[121] R. Ubar, S. Kostin, and J. Raik, "About robustness of test patterns regarding multiple faults," in *2012 13th Latin American Test Workshop (LATW)*, pp. 1–6, April 2012.

[122] R. Ubar, S. Kostin, and J. Raik, "Multiple stuck-at-fault detection theorem," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 236–241, April 2012.

[123] A. Jasnetski, R. Ubar, A. Tsertov, and H. Kruus, "Laboratory framework team for investigating the dependability issues of microprocessor systems," in *10th European Workshop on Microelectronics Education (EWME)*, pp. 80–83, May 2014.

[124] T. Tester, "http://www.pld.ttu.ee/testing/labs/ttfiles/turbotech. rep., Tallinn University of Technology.

[125] M. Scholzel, *Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis*. PhD thesis, Brandenburg University of Technology Cottbus-Seftenberg, 2015.

[126] A. Riefert, R. Cantoro, M. Sauer, M. S. 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)*, pp. 1186–1191, March 2015.

[127] P. Bernardi, L. Ciganda, M. Grosso, E. Sanchez, and M. Sonza Reorda, "A sbst strategy to test microprocessors' branch target buffer," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 306–311, April 2012.

Acknowledgements

I would like to express my heartfelt appreciation to my supervisor, Professor Raimund Ubar for his support, mentoring, and several meetings and discussions throughout my PhD studies. It has been a real honor working with you. I would also like to thank Professor Jaan Raik and Professor Maksim Jenihhin for their collaboration and support.

Special thanks to the Head of the Department of Computer Systems, Dr. Margus Kruus for the support throughout my studies. I would also like to thank my colleagues at the Department of Computer Systems and my friends both at Grace Chapel Tallinn and around the world.

Likewise, I would like to acknowledge the organizations that have supported my PhD studies: Tallinn University of Technology, Estonian IT Academy program, EU's H2020 RIA IMMORTAL, Estonian Center of Excellence in IT (EXCITE) and Estonian Research Council.

I would like to finally thank my family, my parents, my siblings and my wife, Onoitem. Thank you for your endless support.

Abstract
High-Level Implementation-Independent Software-Based Self-Test for RISC Type Microprocessors

Advances in technology make it possible for microprocessors to be built from billions of transistors and operate at ever-increasing operational frequencies. However, the probability of different physical defects is also increasing, and the growing complexity of systems makes testing problems difficult to solve. Processor at-speed testing is a problem for external tester technologies, as they cannot reduce the growing gap between processor frequencies and test frequencies. Built-in Self-Test, on the other hand, imposes area overhead, performance degradation and excess power dissipation on systems such as microprocessors. To address these issues, Software-Based Self-Test (SBST) was proposed.

Software-Based Self-Test (SBST) is an emerging paradigm in the test domain which relies on the exploitation of existing available resources resident in the system. The essence of this approach implies features such as non-intrusiveness, low cost and reliability with speed, and in-field testing. The quality of SBST is mainly evaluated by single stuck-at fault (SAF) coverage, and no measures exist for evaluating the test coverage regarding other fault classes. Another problem which has not been sufficiently investigated is test program generation, where no information about the details of implementation is given.

To address these drawbacks, a novel high-level approach was adopted in this thesis for implementation-independent SBST generation for a broader class of faults other than only the single SAF model. This approach targets processors with RISC architecture.

A novel high-level model of the microprocessor is derived from the instruction set and from the architectural information of features introduced for increasing performance, like pipelining, forwarding, hazard handling and prediction.

The main concept of the approach is in partitioning the processor into functionally well-defined modules under test (MUT), and each MUT into two formal disjoint parts which are the control and data parts. As a theoretical basis for developing a new approach, the model of High-Level Decision Diagrams (HLDD) was chosen and adapted for modeling of microprocessors at the architectural level. Firstly, HLDDs allow for a straightforward partitioning of the modules under test (MUT) of the microprocessor into two disjoint parts (control and data parts). Secondly, it allows for a flexible partitioning of the set of functions that are mapped to the MUT, into disjointed groups. Based on such partitioning, two separate and independent methods for test generation were developed for testing the control and data parts.

To test the control part of a MUT, a novel high-level control fault model was developed and defined as a set of data constraints derived from the HLDD model. The control part of MUT is presented as a set of functions, which can be partitioned into subsets to keep the test generation complexity scalable. For partitioning, HLDDs are used. The data constraints are to be satisfied by test data generation, for which two algorithms were developed. For testing the data part of a MUT, structured pseudo-exhaustive test data is generated.

A method was developed for simulating the high-level control faults and for evaluating the high-level control fault coverage. It was proven that a test which guarantees 100% coverage of non-redundant high-level faults, will also guarantee 100% non-redundant SAF coverage, while all gate-level SAF not covered by the test are identified as redundant. Hence, the proposed high-level fault model allows the reduction of time for redundancy

proof by replacing the time-consuming proof of SAF redundancy with very fast gate-level SAF simulation.

It was shown theoretically that the developed high-level test program generation approach covers a very large class of high-level functional faults in a similar way to those used in memory testing. Consequently, a large class of structural faults such as conditional SAF, multiple SAF and bridging SAF are also covered.

The feasibility of the approach and high efficiency of the generated test programs was demonstrated for the modules of the miniMIPS RISC processor, such as execute module, forwarding module and register decoders. It was also shown by experiments that the novel implementation-independent test generation approach enables the fast generation of manufacturing tests with very high single SAF coverage in the case where the implementation details are given for test evaluation purposes. In the experimental part of the thesis, the quality of test with regards to SAF coverage was compared with state-of-the-art methods for the modules of the processor used as a case study.

Kokkuvõte
Mikroprotsessorite tarkvarapõhine implementatsioonist mittesõltuv funktsionaalne enesekontroll

Tehnoloogia areng on teinud võimalikuks valmistada mikroprotsessoreid, mis koosnevad miljarditest transistoridest ja mis töötavad üha kõrgematel sagedustel. Koos sellega kasvab ka erinevate füüsikaliste defektide tõenäosus ja süsteemide suurenev keerukus muudab testimisprobleemide lahendamise üha raskemaks. Protsessorite dünaamika testimisest on saanud tõsine väljakutse protsessorite ja väliste testseadmete vahelise kasvava erinevuse tõttu töösagedustes. Teiselt poolt, spetsiaalsete sisse-ehitatud testriistvara kasutamine põhjustab aga suuremat riistvara kulu, töökiiruse langust ja võimsustarbe kasvu. Loetletud probleemidest üle saamiseks on tarkvarapõhine isetestimine (TPI) üha suuremat populaarsust võitmas.

TPI kujutab endast arenevat paradigmat testimise valdkonnas, mille iseärasuseks on süsteemi enda sisemiste ressursside ära kasutamine. TPI põhineb spetsiaalsetel tarkavaprogrammidel, mis on väljatöötatud protsessorite isetestimiseks. Nende olemuseks on madal hind ja kõrge töökindlus. TPI kvaliteet sõltub aga oluliselt testandmete valikust.

TPI süntees protsessoritele digitaalsüsteemides on teadusvaldkond, milles on toimunud intensiivne uurimistöö juba aastaid. Ometi on selles valdkonnas vähem uuritud alasid, nagu näiteks probleem, kuidas sünteesida testimistarkvara tingimustes, kus puudub detailne informatsioon testitava objekti ehk siis protsessori elektroonilise struktuuri kohta, kus aga samal ajal on vaja tagada kõrget testimise kvaliteeti – kõrget rikete katet, vähendades samal ajal kulutusi testimistarkvara väljatöötamisel.

Nimetatud probleemi lahendamiseks on välja töötatud uudne süsteemi kõrgtasandil läbiviidav ja süsteemi detailsest realisatsioonist mittesõltuv funktsionaalne TPI meetod RISC-arhitektuuriga protsessoritele.

Uus meetod võimaldab genereerida tööstuslikke teste, mis tagavad kõrget konstantrikete katet. Võtmekontseptsiooniks uue meetodi puhul on uudse kõrgtasandi rikkemudeli väljatöötamine ja testandmete genereerimine testitavate moodulite juht- ja andmeosadele eraldi. Protsessori mudeli loomisel on lähteandmeteks üksnes kõrgtasandi info – käsusüsteem ja arhitektuuri kirjeldus.

Uue meetodi teoreetiliseks baasiks on kõrgtasandi otsustusdiagrammide mudel, mis võimaldab eraldi käsitleda nii konkreetse mooduli juht- ja andmeosa, kui ka optimeerida juhtosa tükeldamist. Juhtosa testimiseks genereeritakse deterministlikud testandmed, mis rahuldaksid uuest rikkemudelist tulenevaid konformsuse tingimusi, andmeosa testid aga genereeritakse struktureeritud pseudo-ammendavaid operande kasutades iga käsu jaoks eraldi.

Testide kvaliteedi mõõtmiseks on välja töötatud kõrgtasandi funktsionaalsete rikete simulaator. Töö üheks oluliseks tulemuseks on tõestus, et mitteliiaste kõrgtasandi rikete 100%-line kate tagab ka 100%-lise madala taseme struktuursete rikete katte, kusjuures lisatulemuseks on võimalus identifitseerida madala taseme simulatsioonil katmata jäävaid rikkeid kui liiaseid ehk siis mitteolulisi.

Käesolevas töös väljatöötatud meetodid on kasutatavad ka nn. mittefunktsionaalsete moodulite testimiseks, nagu näiteks ennetusskeemid. Niisuguste moodulite kohta on töös samuti läbiviidud vastav eksperimentaalne uurimistöö.

Uus meetod võimaldab genereerida testprogramme, mis katavad ühelt poolt väga laia funktsionaalsete rikete klassi, mida kasutatakse ka mäluseadmete testimisel, aga samuti ka väga laia struktuursete rikete klassi, nagu üksikud ja kordsed konstantrikked, tingimuslikud konstantrikked ja lühisrikked.

Eksperimentaalsete katsete abil uuriti uute meetodite tõhusust ja kvaliteeti. Eksperimentide abil demonstreeriti meetodite suuremat efektiivsust konkreetse RISC-tüüpi mikroprotsessori moodulite puhul, võrreldes seniste meetoditega. Uute meetodite abil genereeritud testide poolt saavutatud konstantrikete katteid võrreldi seniste meetodite abil saadud rikete katteid ning demonstreeriti märgatavat paremust. Veel olulisemaks tulemuseks on aga see, et uued meetodid katavad palju laiemat rikete klassi, kui seni kasutatavad meetodid. Sellesse laiendatud rikete klassi kuuluvad tingimuslikud konstantrikked, lühisrikked aga eelkõige funktsionaalsed skeemide realisatsioonist sõltumatud kõrgtaseme rikked.

Appendix 1

I

R. Ubar, S. A. Oyeniran, M. Scholzel, and H. T. Vierhaus, "Multiple fault testing in systems-on-chip with high-level decision diagrams," *2015 10th International Design Test Symposium (IDT)*, pp. 66–71, Dec 2015

Multiple Fault Testing in Systems-on-Chip with High-Level Decision Diagrams

Raimund Ubar, Stephen Adeboye Oyeniran
Tallinn University of Technology
Computer Engineering Department
Estonia

Mario Schölzel[1], *Heinrich T. Vierhaus*[2]
[1]*University of Potsdam,*
[2]*Technical University of Brandenburg,*
Germany

***Abstract*—A new method of high level test generation based on the concept of test groups to prove the correctness of a part of system functionality is proposed. High-level faults of any multiplicity are assumed to be present in the system, however, there will be no need to enumerate them. Unlike the known approaches, we do not target the faults as test objectives. The goal of using the test groups is to extend step by step the fault-free core of the system by exploiting the knowledge about already successfully tested parts of the system. In case when the proof fails, fault diagnosis will follow. To cope with the complexity of multiple fault masking mechanisms, high-level decision diagrams (HLDD) are used. The proposed method can be regarded as a generalization of the logic level test pair approach for identifying fault-free wires in gate-level networks. Preliminary experimental results, and a discussion of the complexity of the method is presented.**

Keywords—digital systems, multiple faults, fault masking, high-level decision diagrams

I. INTRODUCTION

The technology advancements impose new challenges to testing systems-on-chip as device geometries shrink and the complexity of SOCs increase. Traditional test approaches are based on a single fault assumption, but assuming only single fault cases cannot be any more valid for today's nanoscale circuits, because the effect of fault masking due to multiple faults remains in this case neglected. A complete test for single faults, in general, may be either incomplete for detecting multiple faults due to possible fault masking, or may result in wrong fault diagnosis.

Some results have been achieved in test generation for multiple stuck-at-faults (SAF) at gate-level [1-6]. The main idea of these methods has been to use test pairs to identify fault-free lines in circuits, instead of finding separate test patterns for detecting each SAF separately. The advantage of these methods is that there is no need for creating of single or multiple fault lists, but the complexity of test generation is still close to that of single SAF test generation. The method [2] is based on 16-valued simulation, whereas in [5] an ATPG algorithm based on 7-valued calculus was developed. In [6], a method was proposed based on test pair analysis of the given test, and constructing additional pairs for undetected faults. The paper [4] presents a two phase method where first, the test pairs are found to detect the target SAF independently of other faults, and thereafter, a sophisticated branch and bound procedure is used to complete the test set generation for the faults undetected during the first phase.

In [7, 8] it was shown that test pairs not always can avoid fault masking. In [9, 10], a generalization of the test pair conception was proposed by introducing a new test structure as a test group. The necessary and sufficient conditions were formulated for test groups capable to detect any non-redundant multiple fault in a combinational circuit. The meaning of a test group is to identify a fault-free sub-circuit instead of proving the correctness of a single wire only as in case of test pairs. The conception of test groups was developed using topological analysis of paths in Structurally Synthesized BDDs (SSBDD) [11-13].

In this paper we generalize the logic level test group approach for identifying fault-free sub-circuits in digital systems (systems-on-chip) represented at higher register-transfer levels (RTL) or functional levels using High-Level Decision Diagrams (HLDD). The faults of any multiplicity are assumed to be present in the system, and there will be no need to enumerate the faults. The goal of using test groups is to extend step by step the fault-free core of the system by exploiting the knowledge about already successfully tested parts of the system. In case when the test group will fail, fault diagnosis will follow in the part of the system targeted by failing test group.

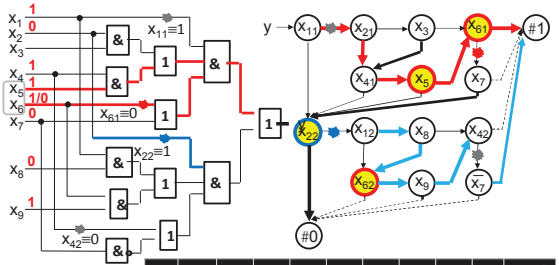
The rest of the paper is organized as follows. Section 2 introduces the concept of topological view on testing of logic level circuits with SSBDDs and explains the meaning of test groups. Section 3 presents the method of modeling digital systems with HLDDs on the example of a VLIW processor, and introduces two classes of high-level faults. In Section 4 we develop a new method of test generation using high-level test groups for testing multiple faults. In Section 5 we give estimations for test length produced by the new method, and compare it with a straightforward method. Section 5 concludes the paper.

II. TOPOLOGICAL VIEW ON THE PROBLEM OF FAULT MASKING IN LOGIC LEVEL CIRCUITS

In the following we give a short explanation on the method [10] of proving the correctness of a gate-level sub-circuit (core)

978-1-4673-9994-4/15$31.00 ©2015 IEEE

by applying *test pairs*, or in general case, by *test groups* (a compressed collection of test pairs)



T_i	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	Y/Y^F
T_1	1	0	0	1	1	1	0	0	1	1/**1**
T_2	1	0	0	1	1	0	0	0	1	0/**0**
T_3	1	0	0	1	0	1	0	0	1	0/**1**

Figure 1. *Illustration of test generation for multiple faults*

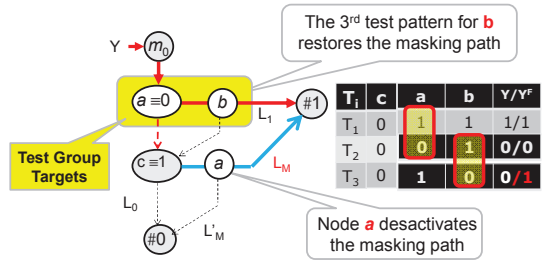
TABLE I. Test Patterns for Selected Faults in Fig.1

t	Test patterns T_t									Test faults	Mask faults
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9		
1	0	0	-	1	1	1	0	1	0	$x_{11}\equiv 1$	$x_{61}\equiv 0$
2	1	0	0	1	1	1	0	0	1	$x_{61}\equiv 0$	$x_{22}\equiv 1$
3	0	0	1	1	0	1	1	-	1	$x_{22}\equiv 1$	$x_{42}\equiv 0$
4	0	1	0	1	1	1	1	-	1	$x_{42}\equiv 0$	$x_{11}\equiv 1$

Example 1. Consider a circuit in Fig.1 which contains 4 SAF: $x_{11}\equiv 1$, $x_{22}\equiv 1$, $x_{42}\equiv 0$, and $x_{61}\equiv 0$. The second subscript at variables denotes the fan-out branch (1 – upper branch, 2 – lower branch). Table I contains 4 patterns for testing these faults (column 11) as single cases, which all will pass if all they are present because of circular masking (by respective faults in column 12). In Fig.1 the pattern T_1 is for detecting $x_{61}\equiv 0$, but it will pass because of masking fault $x_{22}\equiv 1$. The test pair (T_1, T_2) is for proving the correctness of signal path (x_{61}, y) in the circuit. But, this example is exactly the case where the test pair "does not work". Only the group of all 3 patterns is able to detect the fault $x_{61}\equiv 0$ (or, if all 3 patterns will pass, then to prove the correctness of both signal paths (x_{61}, y), and (x_5, y), shown with bold red lines in the circuit).

The fault masking mechanism in Example 1 and in Fig.1 is illustrated topologically in the SSBDD which models the circuit. Each node in SSBDD represents a signal path in the circuit. The paths in SSBDD activated by T1 are shown by red and blue edges (we exit the node x to the right if $x = 1$, and downwards if $x = 0$). The red path terminating in #1 determines the expected value $y = 1$. The fault $x_{61} \equiv 0$ will change the direction of the activated path, however, the masking fault $x_{22} \equiv 1$ will switch to the blue path which terminates again in #1, determining again the value $y = 1$. Hence, $x_{61} \equiv 0$ remains undetected.

Fig.2 illustrates how to cope with fault masking in combinational circuits in presence of multiple faults. In Fig.2, a skeleton of SSBDD is presented with a root node m_0. To test $a \equiv 0$ in SSBDD (similarly as SAF-0 of the node x_{61} in SSBDD in Fig.1), using test pairs, two patterns T_1 and T_2 are generated. T_1 activates a red path L_1 from root to terminal #1 by assigning a = 1 (For $x_6 = 1$ in Fig.1) with expected response Y=1, and a path L_0 from a (x_{61}) to #0. In case of the fault, the path L_0 will be active, and the result Y=0 for T_1 would indicate the presence of fault.



T_i	c	a	b	Y/Y^F
T_1	0	1	1	1/1
T_2	0	0	1	0/0
T_3	0	1	0	0/1

Figure 2. *Detection of the masking fault by test pairs*

In case of a second fault $c \equiv 1$ (similarly as SAF ≡1 of the node x_{22} in SSBDD in Fig.1), the fault $a\equiv 0$ ($x_{61}\equiv 0$) will not be detected by T_1, because $c \equiv 1$ ($x_{22}\equiv 1$) will evoke a masking blue path L_M, which produces the expected result 1 of the fault-free case. To detect the combination of two faults, a second pattern T_2 will be applied by changing in T_1 the value of a (x_6) to 0, and keeping the values of all other variables unchanged. The expected response of the fault-free circuit to T_2 should be 0. But, thanks to unchanged values of all other variables in T_2, the masking path L_M will remain activated, and the response 1 to T_2 will indicate the detection of the masking fault $c \equiv 1$ ($x_{22}\equiv 1$).

The goal of using the test pair (T_1, T_2) is to prove the correctness of the node a under test, if both of the patterns will pass. If there will be the fault $a \equiv 0$, but T_1 will still pass because of the masking fault $c \equiv 1$, then the role of T_2 will be to detect the masking fault.

Test pairs are not always sufficient for proving the correctness of the target variable. For example, if the masking path L_M in Fig.2 involves the same variable a under test (the node x_{62} in Fig.1), the change of the value of a (x_6) in T_2 may discontinue the masking path L_M, which would mean that the masking fault $c \equiv 1$ ($x_{22}\equiv 1$) will remain undetected by T_2. To cope with this disadvantage of test pairs, in [10] the conception of *test groups* was developed.

The main idea of *test groups* is to merge several test pairs for jointly targeting several nodes on the initial path L_1 under test (in Fig2 the node b, and in Fig1 the node x_5), so that at least one of these test pairs would keep the masking path constantly activated, to detect the masking fault. The second test pair (T_1, T_3) targets the node b, and in Fig.1 the node x_5.

III. Modeling Digital Systems with High-Level Decision Diagrams

A. The HLDD-Model for the VLIW Processor

Consider a structural view on the VLIW processor as shown in Fig. 3. It has two execution slots. Each slot has a fetch register that receives the current instruction from the program memory via IN1 (resp. IN2). Instruction of slot i contains the following

control signals: d*i* - destination register, ctr*i*1 and ctr*i*2 - source registers, ctrAlu*i* - operation to be performed by ALU.

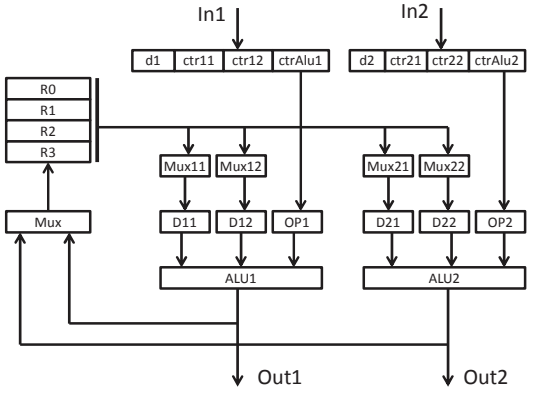


Figure 3. *Strucural Representation of the VLIW processor.*

The components Mux*i*1 and Mux*i*2 represent the read ports of the register file for slot *i*. The processor has a simple register bank with four registers, and two write ports. Both write ports are represented by the component Mux, whereby each write port *i* is controlled by the bit field d*i*. D*i*1, D*i*2, and OP*i* represent the pipeline register between decode- and execute-stage. The write-back stage of the processor has been neglected.

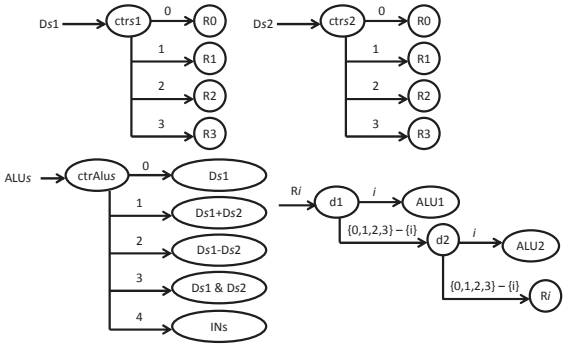


Figure 4. *HLDDs for the components of the VLIW processor*

Let us have a digital system, particularly a VLIW, represented by a set of high level functions $Y = F(C,D)$ determined by the *Instruction Set Architecture* (ISA) of the system, and characterized by a set of *control* variables C, and a set of *data* variables D. Such functions can be represented by *High Level Decision Dia*grams (HLDD). Each HLDD is a directed, acyclic and connected graph G with a root Y and a set of nodes M with *non-terminal* nodes $m \in M^N \subset M$ labelled by control variables $x(m)$ of a set C, and *terminal* nodes $m^T \in M^T = M - M^N$ labeled by operations $f(m^T)$ on a set of data variables D. For each non-terminal node $m \in M^N$, the graph determines a *mapping* $V(x(m)) \rightarrow M(m)$ where $V(x(m))$ is the set of possible values of the variable $x(m)$, and $M(m) \subseteq M$ is the set of successors of the node m.

Denote by m^v the neighbor of the node m for the value $v \in V(x(m))$, according to the mapping $V(x(m)) \rightarrow M(m)$. If a test pattern includes the assignment $x(m) = v$, we say that the *edge* from m to m^v *is activated* by the pattern. We say that each vector of the control variables of C, according to this mapping, *activates a path* in G from the root Y to a terminal node $m^T \in M^T$ and evokes a working mode of a system component $f(m^T)$, e.g. a data transfer or a data manipulation.

Example 2. As an example, the components of the slots $s \in \{1,2\}$ of the VLIW processor in Fig.3 can be represented by HLDDs in Fig.4. The graphs Ds1 and Ds2 represent the left (respectively right) read port of the processor (components Mux11, Mux12, Mux21, Mux22). Depending on the value of the control variables *ctrs*1 and *ctrs*2, a register is selected. The read value from the selected register is assigned to the variables *Ds*1 and *Ds*2. The behavior of ALUs is modeled by the graph ALUs in Fig.4. Finally, the graph *Ri* represents the Mux-component for writing into registers the value of ALUs. The value of *Ri* (register *i*) is determined as follows. If the destination register for the result in slot 1 is *i* (i.e., d1 = *i*), then the value of variable ALU1 is selected. Otherwise, it is checked if the destination register of slot 2 is *i* (d2 = *i*). If this is the case, then the result of variable ALU2 is selected. Otherwise, the value of register *Ri* is hold.

The set of single-cycle HLDDs in Fig.4, can be joined (e.g. for the case of slot s=1) into a single multi-cycle graph in Fig.5. Such a joint HLDD allows better modeling of the multi-cycle test sequences consisting of test stimuli and test response observation cycles controlled by respective instructions. In the joint HLDD, the register nodes R_i represent simultaneously the current state (as part of the stimuli, for cycle τ-1) and the next state (as response to the stimuli, for cycle τ) of the processor. Formally, the register nodes R_i embody in the joint HLDD model two roles: they are interpreted as roots of HLDDs (to model the destinations of Write (WR) operations for cycle τ-1), and as terminal nodes of HLDDs (to model the sources for Read (RD) operations for cycle τ).

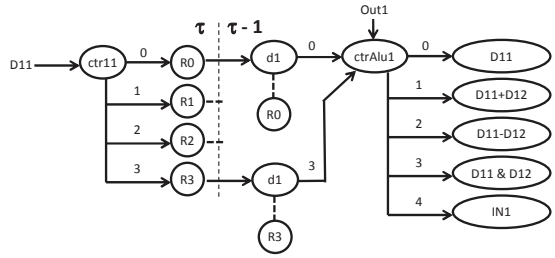


Figure 5. *Joint cycle-based HLDD model for slot 1 of the VLIW processor*

The HLDD model in Fig.5 shows formally how the control signals described by the instruction fields can be used to control the data flow during the instruction cycle from inputs to outputs. The inputs are: IN1, R0, R1, R2, R3, and the outputs are: R0, R1, R2, R3, OUT1. Thereby only IN1 and OUT1 are regarded as test data source and sink, respectively. By giving control signals particular values (derived from

instructions), the data flow, associated with instructions, can be formally specified for test generation purposes on the HLDD.

B. High-Level Fault Modeling with HLDDs

The HLDD model is well suitable for high-level fault diagnosis, in case when the diagnostic resolution is needed with accuracy of locating faulty components represented by the nodes of HLDDs.

In accordance to the VLIW processor's model in Fig.5, the fault location targets will be the read port decoding block modeled by ctr11, write port decoding block modeled by d1, and ALU control decoding block modeled by ctrALU1. These diagnostic targets belong to the control part of the processor. The diagnostic target of the data part will be to locate the faulty ALU operations modeled by the terminal nodes in the sub-graph with the root ctrALU1, and the faulty data registers modeled by the nodes {R0, R1, R2, R3}.

In the following we introduce two types of faults: (1) *node related faults*, and (2) *inter-node faults*.

As the *node related fault* type, we refer to the fault model developed for the HLDDs in [14], which targets exhausting testing of each node in the model. Each path in an HLDD describes the behavior of the system in a specific mode of operation (working mode). The faults which may have effect on the behavior of this working mode can be associated with nodes along the path. A fault in each node may cause a break or incorrect leaving the path activated by a test which would lead to an erroneous activation of another path or several paths simultaneously, terminating in wrong terminal nodes. Exhaustive testing of a node m means the full check if the mapping $V(x(m)) \rightarrow M(m)$ is correctly implemented.

The class of *inter-node faults* emerges from the case when several nodes are labeled by the same high-level variable. In Fig.5, the nodes labelled by *di* represent a de-multiplexer used in write port decoding. The faults in de-multiplexer may cause wrong register accesses and writing data into wrong registers.

IV. Test Generation with HLDDs

A. Fault Activation Constraints

To test the correctness of mapping $V(x(m)) \rightarrow M(m)$ in a graph G, we need to synthesize a test pattern which satisfy two types of constraints: (1) control constraints to activate the desired working modes of the system, and (2) data constraints, for testing that the selected working modes were correctly selected.

Constraints for control variables. To test in the graph G a non-terminal node m, the control variables must be assigned to values which activate the following paths: (1) from root to node m, and (2) from all neighbors of m non-overlapping paths to a subset of nodes of M^T. Denote this subset as $M^*(m) \subseteq M^T$. To test a terminal node m, a single path from root to m has only to be activated.

Example 3. In HLDD *Ds*1 in Fig.4, all these constraints, for testing the node *ctrs*1, are "automatically" satisfied. On the other hand, for testing the node $d1$ in the HLDD *Ri* in Fig.4, there are two possibilities: (1) if we will select $M^*(d1) = \{ALU1, ALU2\}$ then we need to assign $d1 = d2 = i$, and (2) if we will select $M^*(d1) = \{ALU1, R_j\}$ then we need to assign for $d2$ an arbitrary value from $\{0,1,2,3\} - \{i\}$.

To test the node *ctrs*1 in HLDD *Ds*1 in Fig.4, we need to read in turn the contents of all 4 registers shown in the terminal nodes by applying the values $ctrs1 \in \{0,1,2,3\}$. To detect the erroneous behavior of the node *ctrs*1, the contents of registers must be selected in such a way that each possible fault should evoke an erroneous reading.

Consider the following consequences of the fault model described above to the behavior of a node m.

If no register is accessed by the fault then whenever a register R_j is to be retrieved, a ZERO (or ONE, depending on the technology), will be retrieved. Here, ZERO denotes a binary vector (00…0), similarly ONE stands for (11…1). If a subset R' of wrong registers is accessed because of the fault, then the contents formed by bit-wise OR (or AND, depending on the technology) over the registers of R' will be retrieved. The described consideration has been used as well in the fault model developed for microprocessors in [14]. From above, the following formal constraints for data variables can be derived.

Constraints for data variables. From the considerations above, analogically to [3], to test a non-terminal node m, the following constraints must be satisfied for test data:

$$\forall m^T \in M^*(m): [f(m^T) \neq \Omega)], \quad (1)$$

$$\forall m_j, m_j \in M^*(m): [(f(m_i) \vee f(m_j)) \neq f(m_i)], \quad (2)$$

where Ω = ZERO (or ONE, depending on the technology used in the implementation). As constraints for test data for testing a terminal node m with label function $f(m)$, a set of test patterns can be used generated by any ATPG for the logic-level circuit which implements the function $f(m)$. In this way, the terminal nodes in HLDD are tested hierarchically by combining high-level control constraints with low-level data constraints.

Example 4. For testing the node *ctrs*1 in the HLDD *Ds*1 in Fig.4, we may initialize the registers by data: $R_0 = 0001$, $R_1 = 0010$, $R_2 = 0100$, $R_3 = 1000$. Such a solution satisfies the data constraints (1) – (2), and when using these data, any possible multiple fault in the decoding block *ctrs*1 can be detected. To test the control variable ctrAlus in Fig.4, the values of data variables *Ds*1 and *Ds*2 should be selected in a similar way, so that the constraints (1) – (2) were satisfied.

B. Testing by Component Level Test Groups

From the example above, it follows that the test of the node *ctrs*1 needs 8 instructions, first, loading all 4 registers with patterns $T = \{0001,0010,0100,1000\}$, respectively, and then reading out the contents of the registers. If the test passes, assuming that the write operations are fault-free, the Mux *s*1 of the read port *Ds*1 can be stated as fault-free. In case of any possible multiple fault in this sub-circuit, at least once during the 4 read operations, a data word will be read out which does not belong to T, and the sub-circuit should be considered as faulty.

Let us call such an exhaustive test of a node in HLDD as a *component level test group*. The correct result of the test

guarantees that in the component or sub-circuit, represented by the node under test, any multiple fault in the control part is missing, under the presumption that the other nodes involved in the activated HLDD path are fault-free. This statement results from the fault model for multiplexers as a bit-wise OR of data operations (see: data constraints) to be propagated through the multiplexer. The faults of this type can produce only increasing 1-s (if not the case Ω = ZERO) in the propagated data word of the tested operation and hence, the multiple faults in this sub-circuit can never mutually mask each other. The component level test group (exhaustive test for a node in HLDD) can be regarded as a generalization of the logic level test-pair (exhaustive test for an SSBDD node) for proving the correctness of the respective wire or signal path through the gate-level circuit.

As we saw in Section 2, there may still exist multiple faults which cannot be detected by test pairs because of fault masking. Similarly, the described component level test group may not detect a class of high-level multiple faults which present a combination of node related and inter-node faults.

C. Testing by System Level Test Groups

Let us generalize the SSBDD based multiple fault test group idea [10] for being used as well in HLDDs. Consider test generation for data transfer nodes *ctr*1 and *d*1*j* on the HLDD in Fig.6, extracted as a subgraph from HLDD in Fig.5 by fixing *ctrAlu*1=4 for Write and *ctrAlu*1=0 for Read cycle. The extracted HLDD in Fig.6 joins two consecutive test write and read cycles, and allows better following the cause-effect relationships of the test along the HLDD paths. When testing *ctr*1, or *d*1*j*, any possible masking fault related to the nodes *ctrAlu*1 in Fig.6 will be detected, because the masking paths evoked by these faults will be kept stable for the whole test, and hence, according to the test pair conception, will be detected. So, the problem of inter-node fault masking will arise here only in relation to the two nodes *ctr*1 and *d*1*j*,

The control variable *d*1 is split into 4 node variables *d*1*j* which represent a de-multiplexer, and illustrate well the case of inter-node faults. Because of multiple faults in this de-multiplexer, all 4 nodes *d*1*j* may cause erroneous WR. For example, when testing *d*10 by assigning *d*1=0 to access R0 for writing, any other *d*1*j*, j≠0, may be as well activated because of an inter-node fault, which will cause wrong writing to *Rj* and data overwrite. This fault may be later masked due to overwrite of the same register in turn by other data used for test stimuli. In Fig.6, possible masking paths are highlighted with blue color. As in logic level case [10], each possible masking path (shown in blue) must be kept stable (here, the erroneously accessed register must not be overwritten before the read operation to detect the masking fault).

To avoid masking of possible WR faults, after activating each *d*1*j* node, any possible wrong activation of another *d*1*k*, $k \neq j$, must be immediately checked. In other words, for each register, after WR, the contents of other registers must be checked before any new WR. Let us call this method, as *straightforward high-level test pair* approach where write and read cycles have to be intermittent. Such an approach, however, will produce explosion of the test length.

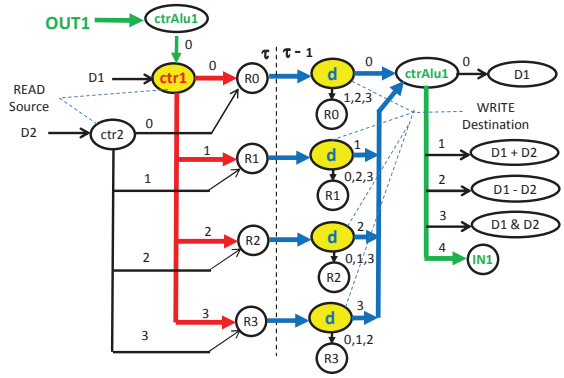


Figure 6. *Joint cycle-based HLDD model for slot 1 of the VLIW processor*

To cope with the masking effect of inter-node multiple faults, we generalize now the SSBDD based test group approach [10] to a new HLDD based high-level integrated test group conception which allows generating test programs immune to mutual multiple fault masking.

Denote by Wr↑ and Rd↑ the write and read sequences, respectively, where the arrow shows the order of addresses. Let us load the 4 registers with set of patterns T={0001,0010,0100,1000}, derived earlier. Any multiple fault at ctr1 can be detected by any of 4 component level test groups: T_{00}=(Wr↑,Rd↑), T_{01}=(Wr↑,Rd↓), T_{10}=(Wr↓,Rd↑), T_{11}=(Wr↓,Rd↓), according to discussion in sections 4-A,B, however, under presumption that Wr↓ is correct.

To cope with the mutual masking of inter-node multiple faults related to the nodes *d*1*j* (WR), the test group $T_{00,10}$ = {(Wr↑,Rd↑),(Wr↓,Rd↑)} may be used. It can be easily shown that the inter-node faults which will be masked in Wr↑ due to overwriting, and not detected by T_{00}, will be still detected by T_{10}, because the order of accessing the registers in Wr↓ will be opposite to Wr↑. To cope with possible mutual masking of inter-node faults related to *ctr*1, the test group $T_{01,00}$ = {(Wr↑,Rd↓),(Wr↑,Rd↑)} will be sufficient, because now the order in Rd↑ will be opposite to Rd↓. To summarize, the system level test group $T_{00,10,01}$ merges 2 component level test groups and will detect all multiple (node related and inter-node related) faults with respect to the nodes d1j and ctr1.

The method proposed is valid for testing of any non-terminal node in HLDD which represent a control variable. The complexity of the method depends on the number of values of control variable under test. The main burden of test generation for terminal nodes falls on gate-level ATPG to solve the data constraints as discussed in Section IV-A.

V. Complexity of the Method and Experimental Results

Let us compare the test lengths of two approaches: (1) the *straightforward high-level test pair* approach derived from the

logic level test pair method [10], and (2) the proposed *high-level test group* approach. Consider for comparison the discussed problem of testing the nodes *ctr*1 and *d*1*j*in Fig.6.

Let n be the number of registers R_j. The straightforward approach to testing the nodes *ctr*1 and *d*1*j* will involve the following three parts: (1) initialization of registers (n write cycles), (2) high-level test group for testing the node *ctr*1 (n read instructions), and (3) n test pairs for each node *d*1*j*: WR for R_j with immediate RD for another R_k, $k \neq j$). Hence, the length of the straightforward multiple fault test will be $2n + 2n^2$. When applying the new proposed *system level test group* approach, we have to construct a test sequence: $T_{00,10,01}$ = {(WR↑,Rd↑), (WR↓,Rd↑), (WR↑,Rd↓)}, which has the length 6n. Hence, the length of the test, developed by the proposed method will be $(2n+2n^2)/6n = n/3$ times shorter than the test length of the straightforward approach.

Table 2. *Comparison of different test generation methods*

Module	#Faults	Fault coverage %		
		Proposed method	[17]	[16]
AC	156	99.3	99.3	99.3
IR	228	99.4	96.4	98.60
PC	590	99.3	99.0	89.20
MAR	342	99.2	96.40	97.20
SR	130	99.0	96.80	98.90
ALU	556	98.30	98.00	98.50
SHU	310	100	99.20	94.10
Control	648	89.8	84.40	88.30
Total	2960	98.04	96.19	95.51

In Table 2, the experimental results, compared with [16, 17], and carried out for microprocessor PARWAN [14], demonstrate the gate-level SAF coverage achieved by HLDD node exhaustive testing. The new proposed method, which targets additionally inter-node multiple faults, will improve in turn the quality of test, as it was shown by reasoning in Section 4-C. To evaluate this additional impact experimentally will be the objective of further research. Note, the complexity of the proposed high-level test generation method does not depend on the length of the processor's data word. The latter will influence only on the performance of the gate-level ATPG when solving the data constraints for testing HLDD terminal nodes.

VI. CONCLUSIONS

In this paper we generalized the logic level test group approach for identifying fault-free sub-circuits for using it for digital systems (systems-on-chip) represented at higher register-transfer or instruction set based functional levels by exploiting High-Level Decision Diagrams. The faults of any multiplicity are assumed to be present in the core under test, and there will be no need to enumerate the multiple faults. Differently from known methods, we do not target the faults themselves as test objectives. Instead, the goal is to verify the correctness of a selected core in the system. It was shown that the test length produced by the proposed method can be estimated as n/3 times shorter than a straightforward approach which was proposed as the first step in this attempt to cope with multiple fault mutual masking.

The method opens a new scheme to fault diagnosis in the presence of multiple faults. The knowledge about identified correct parts of the circuit allows extending step by step the core of the circuit proved as correct. In case when the proof fails, fault diagnosis will follow, but the knowledge about already proved correct functions of the system may considerably simplify the fault location process.

Acknowledgement: The work has been supported by EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds. We thank Artjom Jasnetski and Anton Tsertov for carrying out experiments with PARWAN microprocessor.

VII. REFERENCES

[1] A.G. Birger, E. T.Gurvitch, S.Kuznetsov, "Testing of Multiple Faults in Comb Circuits", Avtomatika i Telemehanika, No8, 1975, 113-120.

[2] H.Cox,J.Rajski. A Method of Fault Analysis for Test Generation and Diagnosis, IEEE Trans. on CAD,v.7,No.7,1988,pp.813-833.

[3] S.Kajihara, T.Sumioka, K.Kinoshita, "Test Generation for Multiple Faults Based on Parallel Vector Pair Analysis", ICCAD'93, 436-439.

[4] A.Agrawal, A.Saldanha, L.Lavagno, "Compact and Complete Test Set Generation for Multiple Stuck-at Faults", ICCAD, 1996, 212-219.

[5] H.Takahashi et al. Test generation for Combinational Circuits with Multiple Faults. Int.Symp.on Fault-Tolerant Systems. 1991.

[6] S.Kajihara, R.Nishigaya, T.Sumioka, K.Kinoshita. Efficient Techniques for Multiple Fault Test Generation. 3rd ATS, 1994.

[7] I.V.Kogan, "Testing of Missing of Faults on the Node of Comb. Circuit", Avtomatika i Vychislitelnaja Tehnika, Automation and Computer Engineering, No 2, 1976, pp. 31-37 (in Russian).

[8] R.Ubar, "Complete Test Pattern Generation for Combinational Networks", Proc. Estonian Academy of Sciences, Physics and Mathematics, No 4, 1982, pp. 418-427 (in Russian).

[9] R.Ubar, S.Kostin, J.Raik, "About Robustness of Test Patterns Regarding Multiple Faults", 13th IEEE LATW, 2012, pp. 86-91

[10] R.Ubar, S. Kostin, J. Raik: "Multiple Stuck-at-Fault Detection Theorem", 15th IEEE DDECS, 2012, pp. 236-241.

[11] R.Ubar, "Test Synthesis with Alternative Graphs". IEEE Design&Test of Comp., 1996, Spring, pp. 48-57.

[12] R.Ubar. Overview of Low & HLDDs for Diagnostic Modeling. Facta Univers,Nis,Ser.Elec.En.vol.24,no.3,2011,pp.303-324.

[13] RUbar, J.Raik, H.Vierhaus (Eds). Design and Test Technology for Dependable Systems-on-chip,.), 2011, pp.92-118.

[14] A.Jasnetski, J.Raik, A.Tsertov, R.Ubar. New Fault Models and Self-Test Generation for Microprocessors using HLDDs. IEEE DDECS. Belgrade, Serbia, April 22-24, 2015.

[15] S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors, IEEE Trans. on Computers, Vol. C-29, No. 6, pp.429-441, 1980.

[16] L.Chen, S.Dey. SW-based self-test methodology for processor cores. IEEE Trans. on CAD of IC & systems, vol.20,no.3, 2001.

[17] Y. Zhang, H. Li, and X. Li, "Software-based self-testing of processors using expanded instructions. ATS,2010,pp.415-420.

Appendix 2

II

A. S. Oyeniran, U. E. Odozi, and R. Ubar, "A new measure for calculating multiple fault coverage of microprocessor self-test," in *2016 15th Biennial Baltic Electronics Conference (BEC)*, pp. 75–78, Oct 2016

A New Measure for Calculating Multiple Fault Coverage of Microprocessor Self-Test

Adeboye Stephen Oyeniran, Uzochukwu Eddie Odozi, Raimund Ubar
Tallinn University of Technology, Estonia

Abstract—**A new measure is proposed for evaluating multiple fault coverage of test sequences for microprocessor circuits. The class of faults under consideration includes gate-level Stuck-at-Faults (SAF), conditional SAF, and bridging faults of any multiplicity in control paths of microprocessors (MP). A new high-level functional control fault model for MP is introduced, and it is shown that 100% coverage of the high-level functional faults will be equivalent to 100% coverage of the low-level structural faults from the mentioned class of faults of any multiplicity. A simple method was developed for test data generation for high-level control faults, and a fault simulation method was developed for calculating the high-level fault coverage. Several high-level methods of test generation for MP were investigated, and the quality of the related tests were compared using the proposed fault measure.**

Keywords: ***Microprocessors, control fault models, test program generation, high-level fault simulation, software-based self-test***

I. INTRODUCTION

For the last decade, there has been an extensive research on software-based self-test (SBST) of embedded processors [1-5]. The quality of SBST is mainly affected by test data used in test programs. One of the ways to obtain test data is executing an Automated Test Pattern Generator (ATPG). In [1, 2] it was shown that processor can be divided into Modules under Test (MUT) to ease the task of ATPG. On the other hand, the difficulties of the method arise from the need of guiding ATPGs by functional constraints to produce functionally feasible test patterns. An alternative way is to use random test patterns for MUTs [3]. In [4], shifting of SBST generation from gate- to Register-Transfer Level (RTL) was suggested. The drawback of this method is that high fault coverage of structural faults cannot be guaranteed. Hybrid SBST methods were proposed for combining deterministic structural SBST with verification-based self-test codes [4-7]. In addition to Hybrid SBST [7, 8], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [6, 9].

The drawbacks of the known methods are: the fault coverage is traditionally measured only regarding SAF, no broader fault classes have been considered, and no attempts have been made to evaluate the test quality regarding covering of multiple faults.

To cope with the complexity of the gate or RT level representations of microprocessors (MP), we consider the problem of SBST program generation with focus on modeling functional faults at the behavioral-level using Instruction Set Architecture (ISA). At the same time, for the purpose of evaluating the quality of tests generated at high-level, we target a broader class of faults than SAF. We consider the conditional SAF and bridging faults as well.

Our previous work has been targeting SBST field with methodology of using High-Level Decision Diagrams (HLDD) for modeling of MP and faults at the behavior kevel [10-12]. For test program generation hierarchical approach was used where the control functions of MP were tested exhaustively (by conformity test), but the data operands for testing the data path were generated by gate-level ATPG (scanning test). The paper [12] was devoted to generation of test groups for detecting of multiple faults, and avoiding fault masking. However, in [12] only the multiple faults in the READ/WRITE logic were considered.

In [11] it was shown that the high-level functional fault model for the control part of MP, based on HLDDs, can be mapped well on the related low-level faults of a joint class of stuck-at faults (SAF), conditional SAF and bridging faults. However, no high-level algorithms have been proposed so far for generating data operands in a formal way for testing the control part of MP. No methods have been proposed for high-level fault simulation as well.

In this paper, we extend the results of [11,12], concentrating on the operational subsystem of MP on the example of ALU. For testing the control part of ALU, we propose a method for high-level synthesis of data operands. We also propose a high-level fault simulation method to evaluate high-level control fault coverage. We show that this fault coverage can be used as a measure of test quality regarding a broad class of multiple faults in MP.

In Section 2, the method of modeling MP with HLDDs is presented. A new method for high-level fault modeling is proposed in Section 2. Section 4 presents a method for high-level test data generation and an algorithm for calculating high-level functional fault coverage. Section 5 presents experimental data, and Section 6 concludes the paper.

II. MODELING MICROPROCESSORS WITH HLDDS

In the following we consider microprocessors (MP) presented on the behavior level and described by instructions given in manuals. An example of a subset of MP instructions is depicted in Table 1.

978-1-5090-1393-7/16/$31.00 ©2016 IEEE

Table 1. *Instruction set of a microprocessor*

OP	B	Mnemonic	Semantics and RT level operations
0	0	LDA A1, A	READ: $R(A1) = M(A)$, $PC = PC + 2$
	1	STA A2, A	WRITE: $M(A) = R(A2)$, $PC = PC + 2$
1	0	MOV A1,A2	TRANSFER: $R(A1) = R(A2)$, $PC = PC + 1$
	1	CMA A1,A2	COMPLEMENT: $R(A1) = \neg R(A2)$, $PC = PC + 1$
2	0	ADD A1,A2	ADD: $R(A1) = R(A1) + R(A2)$, $PC = PC + 1$
	1	SUB A1,A2	SUBTRACT: $R(A1) = R(A1) - R(A2)$, $PC = PC + 1$
3	0	JMP A	JUMP: $PC = A$
	1	BRA A	Conditional jump (Branch instruction): IF C=1, THEN $PC = A$, ELSE $PC = PC + 2$

For behavioral modeling of the set of instructions in MP we use the behavioral level HLDD model in Fig.1 derived directly from the instruction set as explained in [10]. Since the HLDD model is synthesized from the instruction set, it becomes a behavioral level model for MP.

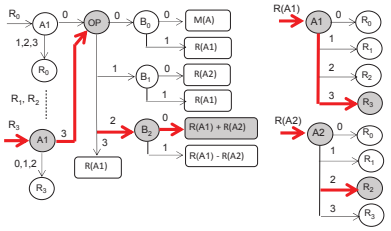


Fig. 1. *HLDDs for the processor given by instructions in Table 1.*

The instruction list of MP is converted into a network of HLDDs where each HLDD represents a functional unit or a subsystem of MP. Each graph has an entry variable (called as graph variable) which represents the output of the unit. The value of this variable can be calculated by tracing the graph according to the values of the node variables. In each node, a decision is made about the direction of tracing according to the value of the node variable. The node variables represent the functional variables used in the descriptions of instructions in Table 1 (OP, B – opcode variables, A – address variables, R – register variables etc.). The value of the graph variable will be equal to the value of the expression (or variable) in the terminal node reached by tracing the graph. As an example, the red arrows in Fig. 1 show the track (activated path) of tracing the HLDDs for the instruction ADD A1 A2 (OP=2, B=0, A1=3, A2=2).

Each HLDD node on the activated path represents a functional unit of MP activated by the given instruction. The terminal nodes labeled by variables may represent either registers or buses, whereas the nodes labeled by arithmetic or logic expressions represent data manipulation units within ALU. The nonterminal nodes of HLDDs represent the units processing the control information (*OP*, B_i, *A*1, *A*2), which may be decoders, multiplexers or de-multiplexers (the subscript at *B* in the nodes of HLDD are introduced to distinguish the nodes which are labeled by the same variable). For example, the node *A*1 in the HLDD G_{R0} represents de-multiplexer, the node *A*2 in $G_{R(A2)}$ represents multiplexer, and the nodes *OP* and *B* in the graphs represent decoders.

Because of the one-to-one mapping between the nodes in HLDDs and the corresponding high-level functional units, we can use the HLDD nodes as a checklist for high-level test planning and organization of test programs for MP.

In this paper we concentrate only on testing of the control part of ALU using the same HLDD based approach for behavioral modeling of ALU at the given instruction list of MP. Consider a typical set of instructions in Table 2. The operations are represented by operation codes and the related formulas f_i for calculation of the output values of ALU. The behavior level structure of the ALU and its High-Level Decision Diagram (HLDD) model are depicted in Fig2. The HLDD has a single internal node labelled by the control variable *c*. The terminal nodes are labeled by the operation formulas f_i of ALU. The control variable *c* can have values from the domain {0,1, …, 15}. Let us introduce the edge variables c_i, so that $c_i = 1$ if $c = i$.

Table 2. *Instruction subset for an ALU of a microprocessor*

Mnemonic	f_i	*opcode*	Mnemonic	f_i	*opcode*
MOV	f_0	0000	SHL	f_8	1000
ADD	f_1	0001	SHR	f_9	1001
SUB	f_2	0010	ASR	f_{10}	1010
CMP	f_3	0011	INC	f_{11}	1011
AND	f_4	0100	DEC	f_{12}	1100
OR	f_5	0101	RLC	f_{13}	1101
XOR	f_6	0110	RRC	f_{14}	1110
NOT	f_7	0111	NOP	f_{15}	1111

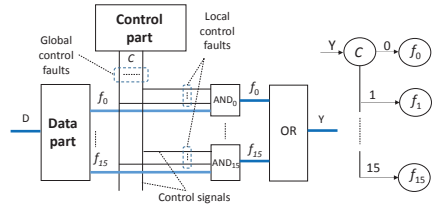


Fig.2. *Behavior level structure of ALU and its HLDD*

III. HIGH-LEVEL FAULT MODELING IN MP

The HLDD model is well suitable for high-level fault modeling in digital systems in case when the fault location accuracy is determined with granularity of the given high-level description. Then, the faults are located in terms of faulty blocks (as black boxes) shown by the HLDD nodes.

The terminal nodes of HLDDs represent the sub-functions carried out in data paths, whereas nonterminal nodes represent the functions of the control part of MP. According to such a mapping of the system functions into the HLDD model, we can classify two types of HLDD-based high-level fault models: *control faults* (the faults related to non-terminal nodes), and *data faults* (the faults related to terminal nodes).

Definition 1. As the fault universe related to non-terminal nodes we allow any corruption in the behavioral of the nodes defined as follows [10]: (1) the output edge of the

node is broken; (2) the output edge of a node is always activated; (3) instead of the activated edge, another edge or a set of edges are simultaneously erroneously activated. The faults of terminal nodes may be interpreted as a special case of the fault model of non-terminal nodes.

From Definition 1, the high-level fault model *for non-terminal nodes* results as the need for exhaustive testing of the nodes. That means the fault model of the non-terminal node must include the set of all values of a node variable to be exercised, accompanied with constraints for data operands to be fulfilled during the exhaustive test of the node. As the fault model for *terminal nodes* we use a set of data patterns needed for testing the related function of the data path.

Denote by m the internal node under test in the HLDD, and by $M^T(m)$ the set of terminal nodes which can be reached from the node m. Let $f(m^T)$ denote the expression labeling the terminal node $m^T \in M^T(m)$. In [11] the following constraints (as part of the fault model) were introduced for testing the control part of MP :

$$\forall m^T \in M^T(m): [f(m^T) \neq \Omega)], \quad (1)$$

$$\forall m_i, m_j \in M^T(m),\ _i \neq _j : \forall k\ [f_k(m_i) < (f_k(m_i) * f_k(m_j))] \quad (2)$$

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [13]. Here, ZERO denotes a binary vector (00…0), and, similarly, ONE stands for (11…1). The index k refers to the bit number of the data words.

Let us focus here only on the case: Ω = ZERO, * stands for OR. Since $f_k(m_i) < (f_k(m_i) * f_k(m_j))$ is valid always if $f_k(m_i) < f_k(m_j)$ is valid, and since in the bit-based analysis the constraint (1) results directly from (2), we can simplify the constraints (1) and (2) as follows:

$$\forall m_i, m_j \in M^T(m),\ _i \neq _j : \forall k\ [f_k(m_i) < f_k(m_j)]. \quad (3)$$

Let us focus now on testing of the single non-terminal node c of the HLDD in Fig.2. As mentioned, it represents the control path of ALU and has to be tested by exercising of all the 16 values of c at the constraints defined by (3).

Let us investigate in more details the meaning of the constraint (3). Denote $f_k(m_i)$ for brevity as f_{ki}. Consider the ALU in Fig.3 divided into control and data paths. Let us concentrate on testing the ALU control at $c_i = 1$. This means that we assign the control value $c = i$, and activate the operation f_i: $y = f_i$. This is the test of existence of the operation – testing that the edge c_i is not broken. Assume now that there is a control fault $c_i \equiv 0$ present (the edge is broken). The fault $c_i \equiv 0$ can be masked if there is another control fault $c_j \equiv 1$, and the data operands are selected so that $f_{jk} = 1$. If the constraint (3) for the bit k is satisfied, i.e. if $f_{jk} < f_{ik}$, such a masking is not possible, and the fault $c_i \equiv 0$ will be detected. Another meaning of the costraint (3) is to detect the faults described in Definition 1 by cases (2) and (3). For example, the fault $c_j \equiv 1$ can be tested by applying the control value $c = i$ under constraints $f_{ik} = 0$, and $f_{jk} = 1$.

In [11] it was shown that the test sequences which satisfy the constraints (1) and (2) will cover a broad class of single faults including Stuck-at Faults (SAF), conditional SAF and bridging faults. It is easy to see that the same is valid also for the constraints of (3) which are easier to use.

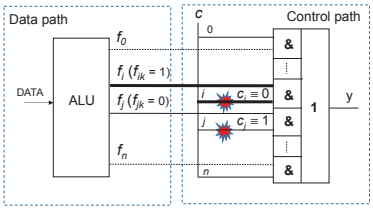


Fig.3. *Behavior level structure of ALU and its HLDD*

The discussion showed that the faults $c_i \equiv 0$ in Fig.3 cannot be masked by any single or multiple faults of type $c_j \equiv 1$, if the constraints (3) are satisfied. Hence the proposed fault model is valid also for testing multiple control faults of any multiplicity whereas only single high-level functional faults need to be counted and targeted.

From above, it follows that the set of all constraints in (3) can be represented as the universe of high-level faults, and the coverage of this universe by generated test can be used as the measure of the quality of the test.

IV. Test Data Generation and Fault Simulation

Currently there are no tools available for generating data operands for control test which satisfy the constraint (3). In [11], a method and algorithm were proposed for simplified *deterministic* generation of data operands, where it was sufficient to satisfy the constraint (3) at least for a single bit. In this paper, we propose a pseudoexhaustive control part test method which targets the constraints (3) for all bits.

The idea of the method is to exercise each ALU bit for each operation pseudoexhaustively. The number of test patterns depends on the number of data operands involved in the operation. For testing 1-bit unary operations (MOV, NOT, SHIFT) only 2 data operands {0, 1} are needed, for testing 1-bit binary operations (AND, OR , XOR) 4 data operands {00, 01, 10, 11} are needed, for testing 1-bit trenary operations (ADD, SUB with carry bits) already 8 data operands or more are needed etc. For n-bit operations (INC, DEC, CMP). The number of actual data operands is n times bigger where n is the length of data words. The reason is that all the needed data combinations must be applied in each bit. As an example, for testing 1-bit binary operations including 1-bit unary operations, the following data operands with word length 8 bit in Table 3 are needed.

Table 3. *Data operands for testing 1-bit binary operations*

	t	$D_1(t)$	$D_2(t)$
To be used by all instructions	0	11001100	10101010
	1	10011001	01010101
	2	00110011	10101010
	3	01100110	01010101

The calculation of the coverage of the proposed high-level functional fault model for the given test sequence leads to the following fault simulation algorithm. Denote by D =

$\{D_t\}$ the set of all data used for testing where D_t is the subset of data operands to be initialized for each test step (instruction to be tested). For example, in Table 3, $D_t = (D_1(t), D_2(t))$, 4 subsets (pairs of data operands) are depicted. Introduce for each t a simulation matrix $S_t = \left| s^k_{ij} \right|$ where $s^k_{ij} = 1$ if $f_{ik} < f_{jk}$, and $s^k_{ij} = 0$, otherwise. Let n be the number of operations, m – the number of data subsets, and w – the length of the data word. $F = n^2 * w$ is the number of all high-level functional faults represented by the bits in S_t where 1 means that the fault is covered, and 0 means – not covered. Let $F(S_t)$ be the number of 1-s in S_t, and $F(S) = \sum_t(S_t)$, t = 1,2, ... m, – the logic sum of all S_t. Then the high-level control fault coverage for the test program T can be calculated as $FC(T) = F(S)*100\%/F$.

The calculated fault coverage $FC(T)$ can be served as the proposed measure for gate-level fault coverage of the test program T for a broad class of faults of any multiplicity, including SAF, conditional SAF and bridging faults.

V. Experimental results

We carried out experiments with ALU (16 operations) of a single slot of VLIW processor [14]. We extracted from the full VHDL design the ALU part, and synthesized from it the ALU gate-level network for evaluating gate-level (GL) SAF coverage of the tests generated at high level (HL). The simulation of tests was performed using ModelSim to obtain the functional data D_t used for HL fault simulation.

Table 4. *Experimental data*

No	Simulation experiment	# Test data	Test length # instr	Fault coverage for	
				High level	Gate level
M1	Gate-level deterministic test	68	68	57.2%	100%
M2	Simplified determ. contr. test [11]	3	48	56.0%	85.8%
M3	Pseudo-exhaustive data part test [11]	77	95	90.3%	99.1%
M4	Pseudo-exhaustive control part test	4	64	85.0%	99.9%
M5	Combined test (M3, M4)	81	159	97.7%	100%

In Table 4, 5 methods for test generation are compared to show the differences between HL and GL fault coverages. The tests were fault simulated at both levels. **M1** forms the GL basis for comparison. We generated with a GL ATPG a test with 100% single SAF coverage. Note, because of very **low HL fault** coverage this test gives **no information about** the quality of **multiple fault detection**. Then, we implemented two HL test generation methods (M2, M3) proposed in [11], and generated a simplified deterministic test for the control part of MP, and a pseudo-exhaustive test for the data part of MP. For both tests, HL fault coverage was calculated by the method described in Section IV. For **M2**, the length of test [11] was 48 instructions: all 16 operations were carried out with all 3 pairs of data operands. The HL control fault coverage was low, because of the control function was tested not in all bits. For **M3**, the data path test achieved a good HL fault coverage because it targeted each ALU operation separately, using dedicated pseudo-exhaustive tests for all bits. For **M4**, the proposed method targeted the control of all bits of data words, which resulted in considerable increase of fault coverage, compared with M2. For **M5**, the combination of two methods (M3, M4) allows considering the specifics of each operation separately. As the result, the best fault coverages were achieved. We have got the same 100% GL fault coverage as for M1, but in addition **very high HL fault coverage**. As a side effect, it was possible to prove the redundancy of 287 HL faults among all 1920 faults. For the 2.3% faults not covered, additional dedicated patterns can be manually generated to cover 100% of HL faults that will guarantee **detecting of all multiple low-level faults**.

VI. Conclusions

In this paper, the first time a measure of testing quality regarding a broad class of multiple gate-level faults in control circuits of MP is proposed. The proposed measure is based on the novel high-level functional fault model. It was shown that the 100% high-level fault coverage is equivalent to 100% fault coverage of multiple gate-level faults of a broad class including SAF, conditional SAF and bridging faults. The measure is qualitative and is not based on counting of single or multiple faults of the broad low-level fault class specified in the paper.

For investigating the feasibility and efficiency of the measure, we proposed a pseudo-exhaustive method of test data generation, and developed a high-level fault simulation algorithm. Both high- and gate-level structural fault simulation methods demonstrated excellent match in reaching 100% fault coverage at both levels.

Acknowledgement: The work has been supported by EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds. We thank Mario Schölzel from U Potsdam for providing us with VHDL description of the VLIW processor for carrying out the experiments.

References

[1] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," in Proc. of ITC, 1997, pp. 743 - 752.

[2] L.Chen, et al. A scalable software based self-test methodology for programmable processors," in Proc. of DAC, 2003, pp. 548 - 553.

[3] L. Chen and S. Dey. SW-based self-test methodology for processor cores, IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, March 2001, pp. 369 - 380.

[4] N.Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software based self-testing of embedded processors," in IEEE Trans. on Comp., vol.54, no.4, 2005.

[5] R. S. Gurumurthy, S. Vasudevan, J.A. Abraham. "Automated mapping of pre-computed module-level test sequences to processor instructions," ITC, 2005.

[6] Y.Zhang, H.Li, and X.Li. Automatic test program generation using executing-trace-based constraint extraction for embedded processors," in IEEE TransactionsVery Large Scale Integration (VLSI) Systems, vol.21, no.7, 2013.

[7] N. Kranitis, et al "Hybrid-sbst methodology for efficient testing of processor cores," in IEEE Design and Test of Computers, vol. 25, no. 1, 2008, pp. 64-75.

[8] C.-H. C. Tai-Hua Lu and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," IEEE Trans. On VLSI Systems, vol. 19, no. 3, March 2011, pp. 516 - 520.

[9] C. H.-P. Wen, et al. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., Vol.55, No. 11, 2006.

[10] A.Jasnetski et al. SW-based Self-Test Generation for Microprocessors with HLDDs. Proc. of the Estonian Academy of Sciences, 2014, 63, 1, 48-61.

[11] A.Jasnetski, S. Adeboye Oyeniran, A.Tsertov, M.Schölzel, R.Ubar. High-Level Modeling and Testing of Multiple Control Faults in Digital Systems. Proc. of DDECS. Košice, Slovakia, April 20-22, 2016, 6p.

[12] R.Ubar, M. Schölzel, S.A. Oyeniran, H.T. Vierhaus. Multiple Fault Testing in Systems-on-Chip with High-Level Decision Diagrams. 10th IEEE International Design & Test Symposium IDT'15, Dead Sea, Jordan,.December 14-16, 2015.

[13] S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors, IEEE Trans. On Computers, C-29, No.6, pp.429-441, June 1980.

[14] M.Schölzel. Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis. Brandenburg University of Technology Cottbus-Seftenberg, 2015.

Appendix 3

III
A. S. Oyeniran, A. Jasnetski, A. Tsertov, and R. Ubar, "High-level test data generation for software-based self-test in microprocessors," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, June 2017

High-Level Test Data Generation for Software-Based Self-Test in Microprocessors

Adeboye Stephen Oyeniran, Artjom Jasnetski, Anton Tsertov, Raimund Ubar
Tallinn University of Technology
Tallinn, Estonia

Abstract— **A new high-level fault model and test generation method for software-based self-test in microprocessors (MP) is proposed and investigated. The model is derived directly from the instruction set of the given MP. A deterministic high-level method and algorithm for test data generation based on this fault model are proposed for the control part of MP. For the data path of MP, pseudo-exhaustive test generation method is proposed, which provides high gate-level SAF coverage. The methods proposed are fully high-level approaches, and no gate-level ATPG is needed. The capability of the new approach to achieving high gate-level fault coverage is demonstrated by low-level SAF simulation.**

Keywords- Microprocessors, fault models, test generation, fault simulation, software-based self-test

I. INTRODUCTION

For today's deep sub-micron technologies, at-speed testing has become essential for achieving high test quality. The traditional solution to cope with at-speed testing is Built-In Self-Test (BIST) [1]. In BIST the tasks of test pattern generation and response evaluation are moved from external ATE to processor embedded logic. This facilitates achieving high-level test quality (including testing of dynamic defects and delay faults), it leads as well to test cost reduction. However, the BIST related testing approaches for microprocessors are found not as feasible as for memories or in application specific integrated circuits (ASICs) [2]. Furthermore, BIST results often in over-testing as well as over-stressing the circuit due to higher than normal switching activity during the test.

As an alternative to hardware-based self-test such as BIST, software-based self-test (SBST) has emerged [2-6]. SBST is a non-intrusive test methodology that is based on using the available processor resources.

For the last decade, there has been an extensive research on SBST of embedded processors. The quality of SBST is primarily affected by test patterns. One of the ways to obtain test patterns is executing an Automated Test Pattern Generator (ATPG). In [3] it was shown that processor can be divided into Modules under Test (MUT) to ease the task of ATPG. An alternative way is to use random test patterns for MUTs [2]. Although the gate-level fault coverage for MUT is acceptable in deterministic and random test pattern generation, some of the generated patterns are typically functionally infeasible when considering the processor as a whole. Thus, ATPG has to be guided by functional constraints to produce functionally feasible test patterns.

An automatic constraint extraction based on gate-level simulation of tests to check their functional feasibility was proposed in [4]. However, the efficiency of the method on the industrial processors was shown to be low. In [5], shifting of SBST generation from gate-level to Register-Transfer Level (RTL) was suggested. The drawback of this method is that high fault coverage of structural faults cannot be guaranteed. Hybrid SBST methods were proposed for combining deterministic structural SBST with verification-based self-test codes [5-8]. In addition to Hybrid SBST [8, 9], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [7, 10]. However, the tendency of embedding more components into a single package is making the efficiency and scalability of the state-of-the-art SBST methods presented above questionable.

In this paper, to cope with the complexity of the gate- or RT level representations of microprocessors, we consider the problem of SBST program generation with focus on modeling functional faults at the behavioral-level using Instruction Set Architecture (ISA). Our previous work was targeting SBST field with a methodology of using High-Level Decision Diagrams (HLDD) for modeling of microprocessors and faults [11,12]. In [13], a new HLDD-based concept for formal test generation for microprocessors was introduced.

In [14], a novel class of hard-to-test faults was introduced, called "unintended actions". It was shown how the self-test programs developed using HLDDs allow detecting the faults from this class. The papers [15, 16] were devoted to the generation of test groups for detecting multiple faults, and for avoiding the mutual fault masking of multiple faults. In [16] it was shown that the test programs generated for the control part of MP using HLDDs can be mapped well on the related low-level faults of a joint class of stuck-at faults (SAF), conditional SAF and bridging faults. However, no algorithms have been proposed so far for generating the test data at high-level in a formal way for testing the control and data parts of MP. Traditionally the test data are generated at the low level with gate-level ATPGs.

In this paper, a method is proposed for fully high-level test program generation for both control and data parts of MP. The test operands for control part are generated using high-level

978-1-5090-6742-8/17/$31.00 ©2017 IEEE

control fault model, and the data path is tested by pseudo-exhaustive operands. The experimental results demonstrate high low-level fault coverage for test programs generated at high-level.

The rest of the paper is organized as follows. In Section 2 the method of modeling MP with HLDDs is presented, and in Section 3, the idea of high-level fault modeling and test program generation using HLDDs is discussed. Section 4 presents the algorithm for high-level deterministic data generation based on the HLDD-based fault model for testing of the control part, and Section 5 presents the ideas for pseudo-exhaustive test pattern generation for testing of the data path. Section 6 presents experimental data, and Section 7 concludes the paper.

II. Modeling Microprocessors with HLDDs

In the following, we consider microprocessors (MP) presented on the behavior level and described by instructions given in manuals. Let us have an example of a subset of instructions in Table 1.

Table 1. *Instruction set of a microprocessor*

OP	B	Mnemonic	Semantics and RT level operations
0	0	LDA A1, A	READ: $R(A1) = M(A)$, $PC = PC + 2$
	1	STA A2, A	WRITE: $M(A) = R(A2)$, $PC = PC + 2$
1	0	MOV A1,A2	TRANSFER: $R(A1) = R(A2)$, $PC = PC + 1$
	1	CMA A1,A2	COMPLEMENT: $R(A1) = \neg R(A2)$, $PC = PC + 1$
2	0	ADD A1,A2	ADD: $R(A1) = R(A1) + R(A2)$, $PC = PC + 1$
	1	SUB A1,A2	SUBTRACT: $R(A1) = R(A1) - R(A2)$, $PC = PC + 1$
3	0	JMP A	JUMP: $PC = A$
	1	BRA A	Conditional jump (Branch instruction): IF C=1, THEN $PC = A$, ELSE $PC = PC + 2$

Denote the instructions as the values of a complex variable I which may be represented as a concatenation of sub-variables I = OP.B.A1.A2 or (I = OP.B.A1.A). The functionality of MP can be described based on the set of control and data variables. The control variables OP and B represent two fields of the opcode. The addresses A1, A2 and A are interpreted as well as control variables describing access to internal registers or memory locations represented by data variables of the set RDATA = {R0, R1, R2, R3, M}.

Consider in the following only the ALU part of MP, which includes data manipulation circuits represented by behavioral level functions: Ri = fRi (OP, B, A1, R(A1), R(A2), M(A)) where $R_i \in R_{DATA}$, i = 0,1,2,3.

For modeling of the set of described functions in MP, we use the HLDD model depicted in Fig.1. Each graph has an entry variable (called as graph variable), the value of which can be calculated by tracing the graph according to the values of the node variables. The value of the graph variable will be equal to the value of the expression in the terminal node reached by tracing the graph.

The HLDD model in Fig.1 represents 6 functions of the MP in the form of 6 HLDDs: GRi for fRi, respectively, where i = 0,1,2,3, and GR(A1), GR(A2) – for addressing the registers. The 4 graphs GRi are merged and share a similar sub-graph with the root node OP, which represents the logic of ALU. The graphs GR(A1) and GR(A2) are accessed when processing the nodes R(A1) and R(A2), respectively, in graphs GRi.

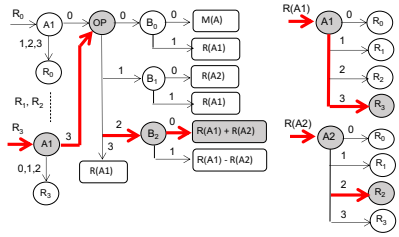


Figure 1. *HLDDs for the processor given by instructions in Table 1.*

We will call in the further text the nodes in graphs by the names of node variables, or by the expressions labeling the terminal nodes. To distinguish the nodes which are labeled by the same variable in the given HLDD, we use subscripts at the node variable. For example, in G_{Ri}, we have three different nodes labeled by the same variable B, and the subscript at B will distinguish the nodes.

Each instruction in Table 1 can be modeled by the related path in the HLDD model. When simulating an instruction, its related path in the HLDD is activated. For example, when simulating the instruction I = OP=2.B=0.A1=3.A2=2, the following paths l in Fig.1 are activated: $l(A1, OP, B_2, R(A1)+R(A2))$ in G_{R3}, $l(A1,R3)$ in $G_{R(A1)}$, and $l(A2,R2)$ in $G_{R(A2)}$. The paths are highlighted by bold edges and grey colored nodes in Fig.1.

Each HLDD node can be regarded as a functional unit of MP activated by corresponding instruction. For example, the terminal nodes labeled by variables may represent either registers or buses, whereas the nodes labeled by arithmetic or logic expressions represent data manipulation units within the ALU. The nonterminal nodes of HLDDs represent the units processing the control information (OP, B_i, $A1$, $A2$), which may be decoders, multiplexers or demultiplexers. For example, the node $A1$ in G_{R0} represents de-multiplexer, the node $A2$ in $G_{R(A2)}$ represents multiplexer, and the nodes OP and B in the graphs represent decoders.

Because of the one-to-one mapping between the nodes in HLDDs and the high-level functional units, we can use the HLDD nodes as a checklist for high-level test planning and organization of test programs for microprocessors. For formalized test program generation, however, we need a suitable high-level (behavioral) fault model.

III. Fault Modeling and Test Program Generation for MP with HLDD

The HLDD model is well suitable for high-level fault modeling and fault diagnosis in digital systems in a case where the

diagnostic resolution is needed with high-level accuracy of locating only faulty blocks (as black boxes) represented by the nodes of HLDDs.

The terminal nodes of HLDDs represent the sub-functions carried out in data paths, whereas nonterminal nodes represent the functions of the control part of MP. According to such a mapping of the system functions into the HLDD model, we can classify two types of HLDD-based high-level fault models: *control faults* (related to non-terminal nodes), and *data faults* (related to terminal nodes).

The high-level fault model *for non-terminal nodes* is based on the exhaustive testing of the nodes. As the fault model for *terminal nodes*, we use a pseudo-exhaustive set of data patterns needed for testing the related function.

Each path in an HLDD describes the behavior of the system in a specific mode of operation (working mode of MP). The faults which may affect the particular working mode can be associated with nodes along the related path.

Definition. A non-terminal node related fault in the HLDD may cause the following corruptions of the model:

1) the output edge of the node is broken;
2) the output edge of a node is always activated;
3) instead of the activated edge, a combination of other edges is erroneously activated.

The faults related to terminal nodes are covered by pseudo-exhaustive test patterns.

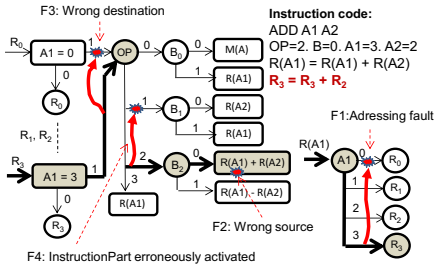


Figure 2. *Illustration of different corruptions of the HLDD by faults in MP*

Example 1. Consider in Fig. 2 how different fault models described in Definition may be represented in a uniform way as the node faults on the HLDD model. An *addressing fault* F1 is illustrated in the graph $G_{R(A1)}$: instead of the edge 3 of the node *A*1, another edge 0 (or concurrently both edges) are activated. This fault can propagate to other HLDDs of the model. For example, it can cause in the ALU graph either the fault of a *wrong source* (F2) or a fault of a *wrong destination* (F3). The fault type F4 – *an instruction part erroneously activated* – is illustrated by the fault of the node OP as "instead of the edge 2 the edge 1 is activated". All these faults belong to the third class of the HLDD fault model defined by Definition.

Denote by *m* the internal node under test in the HLDD and by $M^T(m)$ the set of all terminal nodes which can be reached from the node *m* at the current instruction under test starting from all erroneously activated output edges of the node *m*. Let $f(m^T)$ denote the expression labeling the terminal node $m^T \in M^T(m)$. In [16] the following constraints (as part of the fault model) were introduced for testing the control part of MP:

$$\forall m^T \in M^T(m): [f(m^T) \neq \Omega)], \quad (1)$$

$$\forall m_i, m_j \in M^T(m),\ i \neq j: \forall k\ [f_k(m_i) < (f_k(m_i) * f_k(m_j))] \quad (2)$$

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [17,18]. Here, ZERO denotes a binary vector (00…0), and, similarly, ONE stands for (11…1). The index *k* refers to the bit number of the data words.

For the test program generation with a HLDD G_Y using the constraints (1) and (2) for testing the control part (non-terminal nodes in HLDD), and using the local test patterns generated for testing the data path (terminal nodes in HLDD), we use the following procedures.

Conformity tests. Generating a conformity test $T(m)$ for a non-terminal node *m*, labeled by variable $z(m)$, produces an exhaustive test of $z(m)$. $T(m)$ is a cycle of testing the behavior of the control function related to the node *m*. Each step of the cycle consists in:

1. Initialization of all registers involved in operations at all terminal nodes $m^T \in M^T(m)$ with values satisfying the constraints (1) and (2);
2. Applying the instruction that: assigns to $z(m)$ the next (in turn) value, activates in HLDD a path to the node *m*, and the paths from *m* to $m^T \in M^T(m)$;
3. Observation of the value of the HLDD G_Y.

Scanning tests. Generating a scanning test $T(m^T)$ for a terminal node $m^T \in M^T(m)$, labeled by the expression $f(m^T)$, produces a local test for $f(m^T)$. $T(m^T)$ is a cycle which consists in:

1. Initialization of all registers involved in $f(m^T)$ at the node $m^T \in M^T(m)$;
2. Applying the instruction that activates in HLDD a path to the node m^T;
3. Observation of the value of the HLDD G_Y.

In the next sections, we present the methods and algorithms for generating test data to be used in the described conformity and scanning test programs.

IV. Generation of Operands for Conformity Test

As it was stated, the test data (operands) used in the conformity test program for testing the control part in MP should satisfy the constraints (1) and (2).

For solving this task, we have developed the following deterministic algorithm presented in Fig.3. The idea of the algorithm is to generate the bits of the data words (operands) D_1 and D_2, starting from the least significant bit, and proceeding then bit by bit towards the most significant bit, so that the constraints (2) were solved for all pairs of the functions ($f_k(m_i)$

and $f_k(m_j)$. We consider here only binary operations with one or two data operands.

ALGORITHM 1: Test data generation for the control part

Input: HLDD model for MP; node m under test
Output: Set of test data pairs to be used in the conformity test of MP
Notations:
D_1, D_2 – the pair of test data;
w – the length of the data words D_1 and D_2;
n – the number of terminal nodes $m^T \in M^T(m)$,
(the number of operations controlled by $z(m)$);
k – the current bit number under determination;
I, J – the global number of bit assigned to the 1st bit of the next (in turn) data words to be generated.

$k = 0$, $I = 0$, $J = 0$;
$D_1 = (d_{1,w}, d_{1,w-1}, \ldots, d_{1,1}, d_{1,0}) = (00\ldots0)$;
$D_2 = (d_{2,w}, d_{2,w-1}, \ldots, d_{2,1}, d_{2,0}) = (00\ldots0)$;
FOR $j \leftarrow I$ to n
  FOR $i \leftarrow J$, $i \neq j$ to n
    IF $f_i(D_1,D_2) < f_i(D_1,D_2) \vee f_j(D_1,D_2)$
    at least in one bit
    THEN $c_{ij} = k$;
    ELSE $k = k + 1$;
      IF $k = w + 1$
        THEN $I = i$, $J = j$;
        GO TO END
      END IF
    Find the proper values
    $(\boldsymbol{d_{1,k}}, \boldsymbol{d_{2,k}}) \in \{(0,0), (0,1), (1,0), (1,1)\}$,
    for the data words
    $D_1 = (d_{1,w}, d_{1,w-1}, \ldots, \boldsymbol{d_{1,k}}, \ldots, d_{1,1}, d_{1,0})$,
    $D_2 = (d_{2,w}, d_{2,w-1}, \ldots, \boldsymbol{d_{2,k}}, \ldots, d_{2,1}, d_{2,0})$,
    so that
    $f_i(D_1,D_2) < f_i(D_1,D_2) \vee f_j(D_1,D_2)$,
    at least in one bit;
    IF no solution found
    THEN $c_{ij} = \varnothing$ **ELSE** $c_{ij} = k$;
    END IF
  END FOR i
END FOR j
END

Figure 3. *Algorithm for test data generation*

Example 2. As an example, let us have the following set of 16 operations of ALU depicted in the left most column of Table 2. The operations correspond to part of the instruction set used in the VLIW processor developed at TU Brandenburg, Germany [19]. The HLDD of the ALU for this subset of instructions, as shown in Fig.4, will contain a single nonterminal node m with 16 output edges entering directly into 16 terminal nodes labeled by functions f_i, i = 1,2,…16 (related to the 16 instructions in Table 2).

The first pair of data words D_1, D_2 with length of 8 bits (the arguments of f_i) for testing the node m (labelled by the control variable C) is depicted in the rows 2 and 3 of Table 2. This pair of data was generated by the first run of Algorithm 1. The remaining rows in Table 2 numbered from 1 to 16 present the values of 16 functions f_i, respectively, calculated at D_1 and D_2. Table 4 shows in which bit k of the data words the constraint $f_{i,k} < f_{j,k}$ is satisfied the first time. The symbol ∅ in highlighted cells means that the constraint (2) for this pair of functions can never be satisfied, and the related control fault should be considered as redundant. Other highlighted cells with global bit numbers bigger than 7 refer to the fact that the 1st pair of data operands was not able to satisfy the related constraints.

Mnemonic	f_i	Mnemonic	f_i
MOV	F_1	SHL	F_9
ADD	F_2	SHR	F_{10}
SUB	F_3	ASR	F_{11}
CMP	F_4	INC	F_{12}
AND	F_5	DEC	F_{13}
OR	F_6	RLC	F_{14}
XOR	F_7	RRC	F_{15}
NOT	f_8	NOP	f_{16}

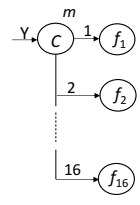


Figure 4. *HLDD for the ALU with control logic of VLIW MP*

To finish the procedure and to satisfy the remaining constraints, two more runs of Algorithm 1 were needed to generate two more pairs of test data. The additional data pairs are presented in Table 3. The remaining constraints are now satisfied in the bits with global numbers 8, 10, and 17. It was not possible to solve the remaining constraint for INC and DEC operations in the 2nd data pair by any of 4 combinations of the bits 11 for D_1 and D_2 because of the influence of already fixed values in previous bits. Hence, the 3rd pair of data words was needed.

Table 2. *The 1st pair of test data*

Data	Bit	7	6	5	4	3	2	1	0
D1		0	0	1	0	1	1	0	0
D2		0	0	1	0	0	1	1	0
MOV, D2	1	0	0	1	0	0	1	1	0
ADD D1, D2	2	0	1	0	1	0	0	1	0
SUB D1, D2	3	1	0	0	0	0	1	1	0
CMP, D1	4	0	0	1	0	1	1	0	0
AND D1, D2	5	0	0	1	0	0	1	0	0
OR D1, D2	6	0	0	1	0	1	1	1	0
XOR D1, D2	7	0	0	0	0	1	0	1	0
NOT, D2	8	1	1	0	1	1	0	0	1
SHL, D1	9	0	1	0	1	1	0	0	1
SHR, D1	10	0	0	0	1	0	1	1	0
ASR, D1	11	0	0	0	1	0	1	1	0
INC, D1	12	0	0	1	0	1	1	1	1
DEC, D1	13	0	0	1	0	1	0	1	1
RLC, D1	14	0	0	1	0	1	1	0	0
RRC, D1	15	0	0	1	0	1	1	0	0
NOP	16	0	0	0	0	0	0	0	0

Table 3*. The 2nd and 3rd pairs of test data*

Data	Bit	…	18	17	16	…	11	10	9	8
D1			1	0	0			1	0	1
D2				1				0	0	1
MOV, D2	1							0	0	1
ADD D1, D2	2									
SUB D1, D2	3									
CMP, D1	4							1	0	1
AND D1, D2	5							0	0	1
OR D1, D2	6							1	0	1
XOR D1, D2	7			1				1	0	0
NOT, D2	8								1	0
SHL, D1	9						1	0	1	1
SHR, D1	10									
ASR, D1	11									
INC, D1	12		1	0	1			1	1	0
DEC, D1	13		0	1	1			0	0	0
RLC, D1	14							1	0	1
RRC, D1	15							1	0	1
NOP	16									

To test the control part, according to Section 3, all the ALU operations have to be carried out for all 3 pairs of data words depicted in Tables 2 and 3. The "unoccupied" free bits in the 2nd and 3rd pairs of data can be filled up randomly.

Table 4 shows the high-level fault coverage where the non-empty cells show that the constraint $f_{i,k} < f_{j,k}$ has been satisfied at least once.

Table 4. *The bits where two operations are distinguished*

Operations f_i		Operations f_j															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MOV, D2	1		4	7	∅	∅	3	3	0	0	1	1	1	1	3	3	∅
ADD	2	5		2	2	2	2	3	0	0	2	2	0	0	2	2	∅
SUB	3	5	4		3	5	3	3	0	0	4	4	0	0	3	3	∅
CMP, D1	4	1	1	1		∅	1	1	0	0	1	1	0	0	∅	∅	∅
AND	5	1	1	1	3		1	1	0	0	1	1	0	0	3	2	∅
OR	6	∅	4	7	∅	∅		∅	0	0	4	4	0	0	∅	∅	∅
XOR	7	2	4	2	2	2	2		0	0	2	2	0	0	2	2	∅
NOT, D2	8	1	1	1	2	2	1	1		8	1	1	1	1	2	2	∅
SHL, D1	9	1	1	1	2	2	1	1	7		1	1	1	1	2	2	∅
SHR, D1	10	5	6	7	3	5	3	3	0	0		∅	0	0	3	3	∅
ASR, D1	11	5	6	7	3	5	3	3	0	0	∅		0	0	3	3	∅
INC, D1	12	8	4	7	8	8	8	17	4	4	4	4		17	8	8	∅
DEC, D1	13	2	4	2	2	2	2	10	4	4	2	2	2		2	2	∅
RLC, D1	14	1	1	1	∅	∅	1	1	0	0	1	1	1	1		1	∅
RRC, D1	15	1	4	7	∅	∅	3	3	0	0	4	4	0	0	3		∅
NOP	16	1	0	0	1	1	0	0	2	0	2	0	0	0	2	1	

V. GENERATION OF OPERANDS FOR SCANNING TEST

For generating test data for the scanning test program to test the data path of MP represented by terminal nodes of the MP (see Fig.4), we use the pseudo-exhaustive test patterns instead of exploiting traditional gate-level ATPG.

Table 5. *Generation of pseudo-exhaustive data for adder*

No	…	4-bit	3-bit	2-bit	1-bit	0-bit
		$a_4\ b_4\ c_4$	$a_3\ b_3\ c_3$	$a_2\ b_2\ c_2$	$a_1\ b_1\ c_1$	$a_0\ b_0\ c_0$
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	0 1 0	0 1 0	0 1 0	0 1 0	0 0 1
3	…	1 0 0	1 0 0	1 0 0	1 0 0	0 1 0
4	…	1 1 0	0 0 1	1 1 0	0 0 1	0 1 1
5	…	0 0 1	1 1 0	0 0 1	1 1 0	1 0 0
6	…	0 1 1	0 1 1	0 1 1	0 1 1	1 0 1
7	…	1 0 1	1 0 1	1 0 1	1 0 1	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

Table 6 *Generation of pseudo-exhaustive data for subtractor*

No	…	4-bit	3-bit	2-bit	1-bit	0-bit
		$a_4\ b_4\ c_4$	$a_3\ b_3\ c_3$	$a_2\ b_2\ c_2$	$a_1\ b_1\ c_1$	$a_0\ b_0\ c_0$
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	1 1 0	0 1 1	1 1 0	0 1 1	0 0 1
3	…	0 0 1	1 0 0	0 0 1	1 0 0	0 1 0
4	…	1 0 0	1 1 0	1 0 0	1 1 0	0 1 1
5	…	0 1 1	0 0 1	0 1 1	0 0 1	1 0 0
6	…	1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
7	…	0 1 0	0 1 0	0 1 0	0 1 0	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

All bits of all ALU operations fi, i = 1,2,…,16, were tested exhaustively. For ADD and SUB operations, 8 data pairs are needed to cover all combinations of 3 inputs (two operands and carry bit) of each bit of the adder (subtractor). As an example, the procedure of test generation for ADD and SUB, bit by bit, is illustrated in Tables 5 and 6. We start to generate patterns from the least significant bits, calculate the carry ci for the next bit and fit in the next bit the values of operand bits ai and bi to the given value of ci so that all pseudo-exhaustive combinations for this bit section were achieved. The columns "2-bit" and "1-bit" can be copy-pasted for the next two-bit-pairs to right.

The patterns in Tables 5 and 6 are valid as pseudo-exhaustive tests for ADD and SUB operations, however, only in case of ripple carry. Since in our case we had a carry-ahead circuit, an additional pair of patterns was needed.

For logic operations, we need 4 exhaustive patterns {(0,0), (0,1), (1,0), (1,1)} per bit, for other unary operations two patterns are sufficient.

VI. EXPERIMENTAL RESULTS

The motivations of the high-level test generation are threefold: possibility to generate test without knowing the details of implementation, speed-up the procedure thanks to the lower complexity of the high-level model, and the possibility to cover multiple gate-level faults with higher confidence [16]. We carried out experiments with two MP: ALU sub-circuit of the VLIW processor [19], and Integer Unit of Leon 3 [20]

In the case of VLIW processor, we concentrated on testing of the ALU together with its control sub-circuit, which provided an easy comparison between high- and low-level fault coverages. We extracted from the VHDL design of the VLIW processor [19] the ALU and synthesized its gate-level network to evaluate the gate-level stuck-at-fault (SAF) coverage of the tests generated at the behavior level by the methods developed in the paper. The functional simulation of the ALU was done using ModelSim to obtain the functional data used for generating test data by Algorithm 1.

Table 7. *Experimental data of the VLIW processor*

No	Simulation experiment	# test data	Test length # instr	Fault coverage for	
				High level	Gate level
M1	Gate-level deterministic test	68	68	57.2%	100%
M2	Det. conform. test (1 bit)	3	48	56.0%	85.8%
M3	Det. conform. test (all bits)	24	384	94.4%	100%
M4	Ps.-exhaustive scanning test	34	54	74.4%	99.1%
M5	Combined test (M2, M4)	57	103	81.0%	100%
M6	Combined test (M3, M4)	101	479	96,9%	100%

In Table 7, six methods for test generation are compared to show the differences between High-Level (HL) and Gate-Level (GL) fault coverages. The test experiment M1 with 100% single SAF coverage forms the GL basis for comparison. Note, because of very low HL fault coverage this test gives no information about the quality of multiple fault detection. We implemented two HL conformity test generation methods (M2, M3): M2 corresponds to Algorithm 1 which targets HL fault coverage at least in a single bit of data word, whereas M3 tries to target the fault coverage in all bits by extending the test M2 by shifting test data 8 times by one bit. M3 provides high HL and GL fault coverages, however, at the high cost of test length. M4 targets the faults only in the data path and covers only partially the faults in the control part. Combining now the HL tests for both, control and data path, it was possible to increase the HL fault coverage by tests M5 and M6 keeping 100% GL fault coverage.

Table 8. *Leon Integer Unit Fault Simulation Results*

	Leon 3 Integer Unit	Faults total /testable	FC %	Simul time min.
1	Proposed HLDD method	42780/ 38847	44.9	37
2	Leon 3 startup test		40.9	22
3	HLDD + Leon 3		45.3	78
4	Load/Store cycle		35.5	27
5	Tetra max ATPG		72.9	2496*
*Time used for ATPG and fault simulation together				

In the experiments with Leon 3, the fault simulation of the test program generated with HLDDs was performed with TetraMAX [21] software. The fault simulation framework is described in details in [14, 22]. In the test experiment, only the part of Integer Unit was involved.

The results are shown in Table 8. The 1st row shows the fault coverage achieved by the proposed method. "Leon3 startup test" is a test program supplied with processor description files [23], which tests memory and peripherals on startup. "LOAD/STORE cycle" is a program which is loading and storing random data to memory million times. "TetraMAX ATPG" represents a local fault coverage of patterns generated by a sequential ATPG tool.

The last case shows that, despite the better GL fault coverage, the attempt of using gate level ATPG is not scalable regarding testing time. Moreover, in this case, not all generated test patterns are functionally correct (i.e. they cannot be reproduced during normal CPU operation). Hence, the real fault coverage in this case is overestimated.

The row "HLDD + Leon 3" represents fault coverage for the joint test program combining the tests in rows 1 and 2, producing the best test quality. The low fault coverage, in fact, is misleading, because not all instructions using the Integer Unit were taken into account for building the HLDD model. In other words, the faults in a major part of the simulated Integer Unit were not targeted at all by the proposed method. Extension of the model for the full instruction list needs further experimental setup.

VII. CONCLUSIONS

In this paper, the first time to our knowledge, a method is proposed for fully high-level test program generation for both, control and data parts of MP, with high low-level fault coverage and without the need to know implementation details of MP. The known methods of test program generation for data manipulation units of MP use gate-level ATPGs.

We proposed a novel high-level deterministic test generation algorithm which showed the capability of achieving 100% gate-level and close to 100% high-level fault coverage in the main core of MP. For testing the data path, we proposed, instead of using traditional gate-level ATPG, the pseudo-exhaustive test generation approach which, in combination with the high-level deterministic control path test, achieved 100% SAF fault coverage. The important effect of the proposed Algorithm 1 is high multiple gate-level fault coverage for a broad class of SAF, conditional SAF and bridging faults, which results from the satisfaction of the constraints (2) [16] by Algorithm 1. This is a significant contribution to the test community since common test methods are targeting only single faults and this is the reason why the short 100% GL test M1 may not be preferred to the longer tests M5 or M6.

The future work will be on improving the methods M3 and M6 in order to reduce the number of test data and to still achieve high HL fault coverage together with high multiple GL fault coverage.

Acknowledgement: The work has been supported by EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds. We thank also prof. Mario Schölzel from the University of Potsdam for providing us with the VHDL description of the VLIW processor for carrying out the experiments.

REFERENCES

[1] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for large industrial designs: real issues and case studies," in Proc. of the International Test Conference, 1999, pp. 358 - 367.

[2] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," in IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, March 2001.

[3] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," in Proc. of ITC, 1997, pp. 743 - 752.

[4] L.Chen, et al. A scalable software based self-test methodology for programmable processors," in Proc. of DAC, 2003, pp. 548 - 553.

[5] N.Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," in IEEE Trans. on Comp., vol.54, no.4, 2005.

[6] R. S. Gurumurthy, S. Vasudevan, J.A. Abraham. "Automated mapping of pre-computed module-level test sequences to processor instructions," ITC, 2005.

[7] Y.Zhang, H.Li, and X.Li. Automatic test program generation using executing-trace-based constraint extraction for embedded processors," in IEEE TransactionsVery Large Scale Integration (VLSI) Systems, vol.21, no.7, 2013.

[8] N. Kranitis, A. Merentitis, G. Theodorou, and A. Paschalis, "Hybrid-sbst methodology for efficient testing of processor cores," in IEEE Design and Test of Computers, vol. 25, no. 1, Feb 2008, pp. 64-75.

[9] C.-H. C. Tai-Hua Lu and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," IEEE Trans. On VLSI Systems, vol. 19, no. 3, March 2011, pp. 516 - 520.

[10] C. H.-P. Wen, et al. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., Vol.55, No. 11, 2006.

[11] R. Ubar, "Test synthesis with alternative graphs," in IEEE Design and Test of Computers, 1996, pp. 48 - 59.

[12] A. Karputkin, R. Ubar, J. Raik, and M. Tombak, \Canonical representations of high-level decision diagram

[13] R.Ubar, A.Tsertov, A.Jasnetski, M. Brik. Software-based selftest generation for microprocessors with high-level decision diagrams. Proc. of LATW 2014.

[14] A.Jasnetski, J.Raik, A.Tsertov, R.Ubar. New Fault Models and Self-Test Generation for Microprocessors using HLDDs. Proc. of DDECS, 2015.

[15] R.Ubar, M. Schölzel, S.A. Oyeniran, H.T. Vierhaus. Multiple Fault Testing in Systems-on-Chip with High-Level Decision Diagrams. 10th IEEE International Design & Test Symposium IDT'15, Dead Sea, Jordan,.December 14-16, 2015.

[16] A.Jasnetski, S. Adeboye Oyeniran, A.Tsertov, M.Schölzel, R.Ubar. High-Level Modeling and Testing of Multiple Control Faults in Digital Systems. Proc. of DDECS, April 20-22, 2016.

[17] S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors, IEEE Trans. On Computers, C-29, No.6, pp.429-441, June 1980.

[18] D.Brahme, J.A.Abraham. Functional Testing of Micro-processors. IEEE Trans. on Comp, C-33,No.6,pp.475-485, 1984.

[19] M.Schölzel. Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis. Brandenburg University of Technology Cottbus-Seftenberg, 2015.

[20] The sparc architecture manual: http://www.gaisler.com/doc/sparcv8.pdf

[21] [Online]. Available:http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx

[22] A. Jasnetski, R. Ubar, A. Tsertov, and H. Kruus. Laboratory framework team for investigating the dependability issues of microprocessor systems. Microelectronics Education 10th European Workshop, May 2014, pp. 80–83.

[23] [Online]. Available: http://www.gaisler.com/products/grlib/grlib-gpl-1.4.1-b4156.tar.gz

Appendix 4

IV
A. S. Oyeniran, R. Ubar, S. P. Azad, and J. Raik, "High-level test generation for processing elements in many-core systems," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, July 2017

High-Level Test Generation for Processing Elements in Many-Core Systems

Adeboye Stephen Oyeniran, Raimund Ubar, Siavoosh Payandeh Azad, Jaan Raik
Tallinn University of Technology, Estonia

Abstract—**The advent of many-core system-on-chips (SoC) will involve new scalable hardware/software mechanisms that can efficiently utilize the abundance of interconnected processing elements found in these SoCs. These trends will have a great impact on the strategies for testing the systems and improving their reliability by exploiting system's re-configurability to achieve graceful degradation of system's performance. We propose a strategy of Software-Based Self-Test (SBST) to be used for testing of processing elements in many-core systems with the goal to increase fault coverage and structuring the test routines in a way which makes test-data delivery in many-core systems more efficient. A new high-level fault model is introduced, which covers a broad class of gate-level Stuck-at-Faults (SAF), conditional SAF, and bridging faults of any multiplicity in processor control paths. Two algorithms for high-level simulation-based test generation for the control path and a bit-wise pseudo-exhaustive test approach for data path are proposed. No implementation details are needed for test data generation. A novel method for proving the redundancy of high-level functional faults is presented, which allows for precise evaluation of fault coverage.**

Keywords: ***processor core testing, high-level control faults, test generation, high-level fault coverage, fault redundancy***

I. INTRODUCTION

Technology scaling trends and the advent of many-core chips suggest that communication, not computation, will dominate delay, area and power budgets in future computing systems. The shift to communication-centric focus on many-core architectures will involve new scalable hardware/software mechanisms that can efficiently utilize the abundance of interconnected processing elements found in these new architectures. These trends will have a great impact on the strategies for testing the systems and improving their reliability by exploiting system's re-configurability as a mechanism for providing graceful degradation of system's performance. In modern many-core system-on-chips, obtaining an overall overview of the system's health is becoming a more pressing issue for resource management and application re-mapping.

Maintaining a global view of the system health requires a testing and monitoring strategy that covers system's processing elements, memory and interconnection infrastructure along with a mechanism for diagnostic information propagation to system's kernel. Using the diagnostic information obtained from different sources, system's kernel can perform more optimal reconfiguration and application mapping decisions. This in turn, will positively contribute to maintaining a graceful degradation of system performance during its lifetime.

There are many challenges for the architecture design of many-core processors, one of which has the most serious concern is manufacturing yield because an IC's profitability depends heavily on it [1]. With the ever-increasing circuit density, obtaining high fabrication yield solely through improving the manufacturing process is increasingly difficult and will become unaffordable in the near future. A more practical solution is to provide defect tolerance capabilities on-chip by incorporating redundant circuits. Previous attempts in this domain mainly focused on introducing microarchitecture-level redundancy [2, 3]. This is appropriate for multicore processors with a restricted number of cores in order to keep the hardware overhead small. The situation is different in the case of many-core processors when the number of cores increases to a point when a single core becomes inexpensive compared to the entire processor. In this case, it is not necessary to tolerate defective cores at the microarchitecture-level, and it will be more appropriate to employ core-level redundancy to reduce the complexity associated with microarchitecture-level redundancy.

More pronounced aging effects (wear-out), process variability, more frequent early-life failures, and incomplete testing or verification due to time-to-market pressure in new fabrication technologies impose also reliability challenges on forthcoming systems [4].

A promising solution to these reliability challenges is concurrent on-line self-test of computing cores and self-reconfiguration of system on chips [5-7]. Core self-test can be performed concurrently with applications executing normally. Concurrent on-line test (COLT) exploits the massive structural redundancy of multi- and many-core architectures by shutting down some subset of cores within the SoC for testing while the remaining cores run user applications as normal. This allows the system to achieve its reliability requirements and maintain an extremely high level of availability with minimum application intrusion. A prerequisite of that is the availability of core self-test routines which should provide high fault coverage at minimum testing time cost.

978-1-5386-3344-1/17/$31.00 ©2017 IEEE

Software-Based Self-Testing (SBST) [8-10] is an emerging new paradigm in the testing domain, which relies on the exploitation of existing available resources resident in the system. The SBST approach is based on software programs that are designed to test the functionality of the processor cores. The major cost of SBST is the time overhead incurred by the execution of the test routines. The hardware overhead is either non-existent, or negligible, and no Instruction Set Architecture (ISA) extensions are required. An important aspect of on-line core testing in SoCs is the scheduling strategy of the test process. One approach is to periodically initiate testing on all system cores simultaneously [11, 12] causing the entire system to be offline, thereby interrupting the execution of application programs. Another approach is to initiate testing on individual cores that have been observed to be idle for some time [13]. In this case, the testing process is minimally intrusive, but the time required to complete the test for all cores is longer. Testing may also be selective, targeting cores that have experienced prolonged stressing due to high utilization.

Recent research results in periodic organization of online testing of many-core processors are presented in [4] to facilitate autonomous detection and omission of faulty cores which makes graceful degradation of the many-core architecture possible. In [14], test-data delivery optimization algorithms are developed for SoC designs with hundreds of cores, where a network-on-chip (NoC) is used as the interconnection fabric.

In this paper, we focus on the quality of the core self-test in terms of increasing the fault coverage, minimizing test length and producing well-structured test routines to ease the diagnosis and test-data delivery around the SoC. The memory and interconnect testing remain beyond the scope of the paper. Also, we will not target the problem of organization and scheduling of test sessions in SoC as a whole.

The rest of the paper is organized as follows. In section 2, we present the state-of-the-art of core testing. In section 3, a method of high-level fault modeling is proposed, and in section 4, the problem of mapping high-level faults into the gate-level faults is investigated. Section 5 presents two new methods for test generation of the control parts of MP cores, and in section 6, a method for proving the redundancy of high-level functional faults is proposed. Section 7 discusses a method for high-level testing of the data path. In section 8, we present the structure of the test routines delivered to the cores of SoC, section 9 presents experimental data, and section 10 concludes the paper

II. STATE OF THE ART

For the last decade, there has been an extensive research on SBST of processors [8,15-18]. The quality of SBST is mainly affected by test data used in test programs. One of the ways to obtain test data is executing an Automated Test Pattern Generator (ATPG). In [16] it was shown that the processor can be divided into Modules under Test (MUT) to ease the task of ATPG. On the other hand, the difficulties of the method arise from the need of guiding ATPGs by functional constraints to produce functionally feasible test patterns. An alternative way is to use random test patterns for MUTs [17]. In [18], shifting of SBST generation from gate-level to Register-Transfer Level (RTL) was suggested. The drawback of this method is that it has not been shown how to achieve high fault coverage of low-level structural faults. Hybrid SBST methods were proposed for combining deterministic structural SBST with verification-based self-test codes [8, 18-20]. In addition to Hybrid SBST [20, 21], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [19, 22].

The drawbacks of the known methods are: the fault coverage is traditionally measured only with respect to SAF, no broader fault classes have been considered, and no attempts have been made to evaluate the test quality regarding covering of multiple faults with avoidance of fault masking.

To cope with the complexity of gate or RT level representations of microprocessors (MP), we consider the problem of SBST program generation with focus on modeling functional faults at the behavioral-level using Instruction Set Architecture (ISA). At the same time, for the purpose of evaluating the quality of tests generated at high-level, we target a broader class of faults than SAF. We consider the conditional SAF and bridging faults as well.

Our previous work has been targeting SBST field with the methodology of using High-Level Decision Diagrams (HLDD) for diagnostic modeling of MP at the behavior level [23-25]. For test program generation, a hierarchical approach was used where the control functions of MP were tested exhaustively (by conformity test), but the data operands for testing the data path were generated by gate-level ATPG (scanning test). The paper [25] was devoted to the generation of test groups for detecting multiple faults and avoiding fault masking. However, in [25] only the multiple faults in the READ/WRITE logic were considered.

In [24] it was shown that the high-level functional fault model for the control part of MP, based on HLDDs, can be mapped well on the related low-level faults of a joint class of the stuck-at faults (SAF), conditional SAF and bridging faults. Moreover, it was proven that the conformity test with 100% high-level fault coverage would be able to detect also any multiple gate-level fault from the mentioned joint single fault class.

In [26], a method for high-level functional fault simulation for microprocessors was proposed. Based on this method it became possible to measure the quality of test programs also with respect to high-level fault coverage. It was shown that the simplified deterministic high-level test generation method developed in [24] for testing the control faults was not able to achieve 100% high-level fault coverage. We have established two reasons for that: (1) the redundant

high-level faults were not identified, and (2) the proposed deterministic method was not able to handle so-called "hard-to-test high-level faults".

In this paper, we extend the previous results [24, 26] by proposing a new test generation method for testing control faults in processor cores of digital systems, which is able to better handle "hard-to-test high-level faults". We also propose a method for proving possible redundancy of not detected high-level faults.

III. HIGH-LEVEL FAULT MODEL FOR PROCESSOR CORES

In this paper, we focus on testing of the ALU module of processor cores. Consider a typical set of instructions in Table 1. The operations are represented by operation codes and the related formulas f_i for calculation of the output values of the ALU. The high-level structure of an ALU is depicted in Fig 1. The control variable c can have values from the domain {0,1, …, 15}. Denote by I_i the instruction (with opcode c_i) which performs the function f_i in ALU.

Table 1. *Instruction subset for an ALU of a microprocessor*

Mnemonic	f_i	*Opcode c*	Mnemonic	f_i	*opcode*
MOV	f_0	0000	SHL	f_8	1000
ADD	f_1	0001	SHR	f_9	1001
SUB	f_2	0010	ASR	f_{10}	1010
CMP	f_3	0011	INC	f_{11}	1011
AND	f_4	0100	DEC	f_{12}	1100
OR	f_5	0101	RLC	f_{13}	1101
XOR	f_6	0110	RRC	f_{14}	1110
NOT	f_7	0111	NOP	f_{15}	1111

Represent the instruction set in Table 1 by the high-level structural circuit and High-Level Decision Diagram (HLDD) [23-25] in fig 1. The HLDD has a single decision node labeled by the control variable c and 16 terminal nodes labeled by the functions *fi* implemented in the ALU data path and selected by instruction *fi* respectively. The node c represents the whole control part of the ALU. In general case, if the system is described by more control variables (representing control fields of the instruction word, register addresses, flags, conditions etc.), the internal structure of the HLDD will be more complex as well [25].

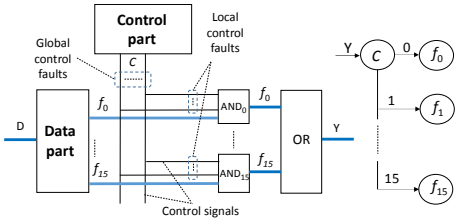


Fig.1. *Behavior level structure of ALU and its HLDD*

The value of the graph variable Y is calculated by traversing the HLDD from the root node to the terminal nodes. In the current example, the value of c decides the direction of traversing. If $c = i$, we will have $Y = f_i$

According to such a mapping of the system functions into the HLDD model, we can classify two types of HLDD-based high-level functional fault models: *control faults* (the faults related to the non-terminal nodes), and *data faults* (the faults related to the terminal nodes).

Definition 1. As the fault universe related to non-terminal nodes we allow any corruption in the behavior of the nodes be defined as follows [23]:

(1) the output edge of the node is broken (the control signal SAF $c_i \equiv 0$, or at the bit level SAF $c_{i,k} \equiv 0$ for controlling function in the bit k);

(2) the output edge of a node is always activated (SAF $c_i \equiv 1$, or SAF $c_{i,k} \equiv 1$);

(3) instead of the activated edge, another edge or a set of edges are at the same time erroneously activated

The last most general fault can be notated as a conditional SAF ($c_j \equiv 1/c_i$), or as ($c_{j,k} \equiv 1/c_i$), This fault type may be caused by bridging fault, other line coupling faults, or it may be explained by more complex physical defects of the control line c_j (or $c_{j,k}$). □

Denote by m the internal node under test in the HLDD, and by $M^T(m)$ the set of terminal nodes which can be reached from the node m. Let $f(m^T)$ denote the expression labeling the terminal node $m^T \in M^T(m)$. In [24] the following constraints (as part of the fault model) were introduced for testing the control part of MP:

$$\forall m^T \in M^T(m): [f(m^T) \neq \Omega)], \quad (1)$$

$$\forall m_i, m_j \in M^T(m),\ _i \neq _j : \forall k\ [f_k(m_i) < (f_k(m_i) * f_k(m_j))] \quad (2)$$

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [27]. Here, ZERO denotes a binary vector (00…0), and, similarly, ONE stands for (11…1). The index k refers to the bit number of the data words.

Let us focus here only on the case: Ω = ZERO, * stands for OR. Since $f_k(m_i) < (f_k(m_i) * f_k(m_j))$ is valid always if $f_k(m_i) < f_k(m_j)$ is valid, and since in the bit-based analysis the constraint (1) can be satisfied indirectly from (2), we can simplify the constraints (1) and (2) as follows:

$$\forall m_i, m_j \in M^T(m),\ _i \neq _j : \forall k\ [f_k(m_i) < f_k(m_j)] \quad (3)$$

From Definition 1, the high-level fault model *for non-terminal nodes* results, which leads in a natural way to exhaustive testing of control modes of MP. In other words, the fault model of a non-terminal node leads to exercising of the set of all possible values of the node variable, accompanied with the constraints (3) for data operands to be satisfied during the exhaustive test of the node.

The number of functional faults for the non-terminal node m can be calculated as $N(m) = n_m(n_m - 1)k$ where n_m – is the number of output edges of the node m, and k – is the length of the data word. In our example, we have $N(c) = 16*15*k = 240*k$.

Definition 2. As the high-level fault universe for terminal nodes f_j., we use the functional fault model defined as the sets of test data $D(f_j)$ needed for testing the functions f_j □

The size of the data path high-level fault model is measured as the number of faults $N(f) = \Sigma_j |D(f_j)|$ where $|D(f_j)|$

is the number of test data for testing the sub-circuit of ALU, responsible for the ALU operation f_j.

From the one-to-one correspondence between the functional faults in the Data Path and the tests targeting these faults, the following results.

Corollary 1. The length of the full ALU data path test $T(f)$, in terms of the number of instructions involved as test objectives, is equal to the number of functional faults in the ALU data path $T(f) = N(f)$ □

Proof. The proof results from the principle of how we test the data path.

Note, the number of high-level functional faults in the data path is not predefined before test generation, rather it is the byproduct of the test generation procedure. The quality of the ALU data path test is measured by low-level simulation to calculate the low-level fault coverage.

The situation will be different for the ALU control part where the high-level functional fault coverage may have even broader meaning and importance than the traditional SAF coverage. As we will see in the next Section, by high-level fault coverage it becomes possible to evaluate the low-level test quality for a broader class of faults than SAF, including multiple faults.

IV. Relationships between High- and Low-Level Fault Models

Consider now the following statements about the quality of test programs synthesized according to the fault models described in the previous Section for control and data parts of the ALU. For simplicity, consider in the following discussion the simplified case of HLDD in Fig.1. Denote $f_k(m_i)$ in (3) for brevity as $f_{i,k}$.

Introduce the following notations. Let F – be the set of functions at the terminal nodes reachable from the node c under test, $d(f_i)$ – a group of data operands as arguments for the function f_j, and $D(f_i)$ – a set of data groups $d(f_i)$. Denote by $T(c_i)$ a test which consists in repeating the instruction I_i for all groups of data operands in $D(f_i)$.

Theorem 1. The control test $T(c_i)$ will detect all gate level conditional SAF faults $\boldsymbol{c_{j,k} \equiv 1/c_i}$ for all $j \neq i$, and SAF $\boldsymbol{f_{i,k} \equiv 1}$, iff for each pair (i,j) and each bit k, there is at least one $d(f_i) \in D(f_i)$ which satisfies the set of constraints.

$$\forall f_j \in F,\ _{i \neq j}: \forall k\ (f_{i,k} < f_{j,k}) \qquad (4)$$

Proof. Consider the ALU in Fig.2 divided into control and data paths. Let us focus on testing of the ALU control part, using the instruction I_i which produces the opcode $c = i$, and assigns the values $c_i = 1$ ($c_{i,k} = 1$ for all k) for all related control signals, and activates the operation f_i: $Y = f_i$. This is the test for checking if the output edge $c = i$ of the node c is not broken in the HLDD in Fig. 1 (no fault of type $c_{i,k} \equiv 0$), according to the fault model in Definition 1. The edge is working correctly if the non-zero correct value of f_i propagates from the output of data path to the output Y of the control path (see Fig.2). However, according to the fault model in Definition 1, there may be present another conditional SAF fault $c_{j,k} \equiv 1/c_i$ on other edges of the HLDD which may mask the faults $c_{i,k} \equiv 0$.

Assume now that there is a fault $c_{j,k} \equiv 1/c_i$, and the data operands are selected so that $f_{j,k} = 1$. If the constraint (4) for the bit k is satisfied, i.e. if $f_{i,k} < f_{j,k}$, the fault $c_{j,k} \equiv 1/c_i$, will be detected. Moreover, from the constraints $f_{k,\,i} < f_{k,j}$, it results that the only way to satisfy this constraint is to assign $f_{i,k} = 0$, and $f_{j,k} = 1$, which means that the faults $\boldsymbol{f_{i,k} \equiv 1}$ will be as well detected. □

To satisfy the constraint (4), a set $D(c_i)$ of several data operands may be needed. Hence, the test $T(c_i)$ for the instruction f_i will consist of repeating the instruction $|D(c_i)|$ times with all data operands in $D(c_i)$.

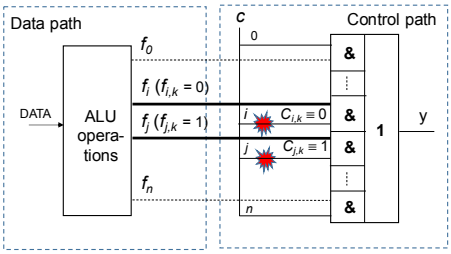


Fig.2. *Behavior level structure of ALU and its HLDD*

Definition 3. Denote the full ALU control test as $T(c)$ which consists of all tests $T(c_i)$ for all functions $f_i \in F$. The length of the test $T(c)$ in terms of the number of instructions to be tested is $N(c) = \Sigma_i |D(c_i)|$

Corollary 2. From Theorem 1, it results that by synthesizing the test $T(c)$ all conditional SAF faults $c_{i,k} \equiv 1/c_j$ for all $j \neq i$, and all SAF $f_{i,k} \equiv 1$, in the control part, will be detected. □

From Theorem 1, it results that the condition (4) is not sufficient for testing the SAF faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$ that is needed for testing the correctness of the output edge $c = i$ of the node c in the HLDD in Fig. 1, which was the basis of synthesizing the test data for $T(c_i)$. The same is valid in accordance to Corollary 2 for all other $j \neq i$.

However, this deficiency of the test $T(c_i)$ is not critical, as it can be concluded from the following statement.

Corollary 3. SAF faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$ in the control part of ALU will be detected by the ALU data path test $T(f_i)$ for the instruction I_i as a byproduct.

Proof. The proof is straightforward. The test $T(f_i)$ consists in repeating the same instruction f_i with different data from $D(f_i)$. But any data in $D(f_i)$ which produces the values $c_{i,k} = 1$ creates the situation where both values, $f_{i,k} = 1$ and $c_{i,k} = 1$ are supporting mutually the propagation of faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$, to the output Y. □

Definition 4. Define the full ALU data path test $T(f)$ as the set $\{T(f_i)\}$ of all tests $T(f_i)$ for the ALU functions $f_i \in F$, respectively, and define the full ALU test as a sum $T_{ALU} = T(c) + T(f)$.

From Corollary 1 and Definition 3 the following results:

Corollary 4. The length of the full ALU test T_{ALU} is $N_{ALU} = N(c) + N(f)$.

In [24] it was shown that the test sequences which satisfy the constraints (1) and (2) will cover a broad class of single faults including Stuck-at Faults (SAF), conditional SAF and bridging faults. It is easy to see that the same is valid also for the constraints (4) used in Theorem 1, which are easier to simulate.

Note that to create final test program, each instruction $I_i \in T(c) \cup T(f)$ with related data operands $op \in D(f_i) \cup D(f_i)$ implies a sequence of instructions which consists of the initialization phase of loading the data operands into the proper registers, testing phase of applying the instruction I_i, and final phase of storing the test result.

V. Generating Data for Control Path Test

From above, it follows that the fault model defined by the set of constraints in (4) can be interpreted as the definition of the universe of high-level faults. A direct impact of this interpretation is the possibility of evaluating the high-level functional fault coverage as the percentage of constraints in (4) for the given test. The fact that the faults $f_{i,k} \equiv 0$ and $c_{i,k} \equiv 0$ in the control path will not be taken in the fault universe of the control faults can be overseen because, according to Corollary 3, these faults will be covered anyway as the byproduct by the data path test $T(f)$.

In the following, we present an algorithm for generating the set T(c) of test data for testing the control path.

Algorithm 1: RANDOM test data generation for ALU

Input: Instruction set of the processor
Output: Sets of test operands OP_i for each instruction, and fault table D
Notations: n – number of instructions (functions F_i), op – test operand, OP – current set of selected random test operands, $f_j(op)$ – the result of the instruction I_j for the operand(s) op, D – fault table, D_{ij} – w-bit entry in D (w – length of the data Word).

1 Initialize $OP = \emptyset$
2 Generate a set of R random operands
3 **for** $i = 1, \ldots, n$
  ***generation of operands for instruction I_i
4 Initialize $OP_i = \emptyset$,
5 **for** $j = 1, \ldots, n$ ($j \neq i$)
    ***operands for solving constraints $f_{i,k} < f_{j,k}$
6 Initialize $D_{ij} = 0$
7 **for** all $op \in R$ **while** $D_{ij} \neq 0$
      ***adding new operands for covering D_{ij}
8 $D_{ij}(op) = f_j(op) \wedge (f_i(op) \oplus f_j(op))$
      *** calculating fault coverage for op
9 **if** $(D_{ij}(op) \vee D_{ij}) \oplus D_{ij} \neq 0$ **then**
        *** check for the coverage increment
10 **begin**
11 $D_{ij} = D_{ij} \vee D_{ij}(op)$
        *** update of the coverage vector
12 Include op into OP_i
        *** new operand is selected
13 **end**
14 **endfor** op
15 **endfor** j
16 **endfor** i

The result of Algorithm 1 will be a set of operands OP_i for each instruction I_j, and the fault table $D = \| D^k_{ij} \|$ where $D^k_{ij} = 1$ means that the functional fault described by the constraint $f_{i,k} < f_{j,k}$ is covered at least by one operand $op \in OP_i$, otherwise $D^k_{ij} = 0$. The percentage of 1s in D is the high-level functional fault coverage of the test for control path.

The Algorithm 1 is called RANDOM since in each step of 7, the first random operand $op \in R$ will be chosen which produces an increase in the fault coverage, no matter how big it is. To reduce the test length for testing the control path we implemented another algorithm called GREEDY.

Algorithm 2: GREEDY test data generation for ALU

GREEDY algorithm differs from RANDOM in running the step 7 before selecting the operand the whole search space R of random operands to calculate the fault coverage increase for all $op \in R$, and only then selects the best one which produces the maximum increase in fault coverage. Then the next operand is selected in a similar way. The step 7 ends when the goal of $D^k_{ij} = 1$ is reached, or no more operands can be selected to satisfy all constraints $f_{i,k} < f_{j,k}$ □

The constraints $f_{i,k} < f_{j,k}$ may not be solved for two reasons: either the related functional fault is redundant, or the search space R is not big enough.

The proof of redundancies of high-level faults introduced in this paper is easy compared to the proof of fault redundancies at gate-level.

VI. Redundancy Proof of High-Level Faults

Consider Table 2, which illustrates the high-level fault coverage D, which was generated by the algorithms in the previous Section for a subset of instructions in Table 1.

In Table 2, the 0s may refer to possible redundancies for the functional faults related to the constraints $f_{i,k} < f_{j,k}$ where i and j correspond to the rows and columns, respectively. All 0s for a D_{ij} refer to the high probability of redundancy of the related fault, i.e. that the constraint $f_i < f_j$ is not possible to satisfy. In most cases of ALU operations, it is very easy to demonstrate this type of redundancy. For example, if i refers to AND operation and j refers to OR, it is straightforward that $(a \vee b) < (a \wedge b)$ can never happen. In Table 2, all 0-s refer to the redundant high-level faults.

Table 2. *Example of a High-Level Fault table*

	f_1 (MOV)	f_2 (ADD)	f_3 (SUB)	f_4 (CMP)	f_5 (AND)
f_1 (MOV)		111111	111111	111111	000000
f_2 (ADD)	11111		111110	111111	111111
f_3 (SUB)	11111	111110		111111	111111
f_4 (CMP)	11111	111111	111111		000000
f_5 (AND)	11111	111111	111111	111111	

In cases when there is 0 in D_{ij} which refers to the case of no solution for $f_{i,k} < f_{j,k}$, only in a single bit k, or few bits, we can suggest for the proof a method which can be called as "*partial truth table method*". The idea of the method stands

in showing the equivalence of partial truth tables (or impossibility to solve the constraint) for the functions involved, where as few as possible responsible bits should be selected for the need of proof. Let us consider few examples for the redundancy proof possibilities for the set of ALU functions in Fig. 1.

In Table 3, examples are shown for 4 partial truth tables for the functions SUB, ADD, OR, AND and for the bit k, where the columns 00, 01, 10, 11 represent the values of the data variables in the bit k. For SUB and ADD, the equivalence of the behavior in the given bit is demonstrated, and in the case of OR and AND, the missing of solution for (4) is highlighted.

Table 3. *Examples of redundancy proofs with 1-bit truth tables*

No	$f_{i,k} < f_{j,k}$	D_{ij}	$f_{i,k}/f_{j,k}$	00	01	10	11
1	SUB < ADD	1…110	SUB	0	1	1	0
			ADD	0	1	1	0
2	OR < ADD	1…110	OR	0	1	1	1
			ADD	0	0	0	0

It is easy to show the equivalence of operations ASR and SHR for all bits, except the most significant bit (MSB). Hence, for all the bits k except for MSB, we can prove that the value $D^k_{ij} = 0$ refers to the redundant faults.

In some cases of proof the partial truth method will not work, because the results of operations may depend substantially on all bits of the word like for increment and decrement. When this happens, specific corner cases should be found for the proof. For example, to proof the equivalence of increment and decrement operations in the least significant bit, the operand 1…110 should be used, where both instructions INC and DEC produce the same result "all 1s".

VII. Generating Operands for Data Path Test

For testing the data path, we can use the pseudo-exhaustive testing approach instead of exploiting traditional gate-level ATPG [28]. Using pseudo-exhaustive data makes the test generation procedure not depending on the implementation details of the processor cores under test.

The method of pseudo-exhaustive test generation lays on the idea of testing the operations in all bits independently of other bits.

Since the logic operations are substantially independent in all bits, we can apply true exhaustive approach for testing, using only 4 exhaustive patterns {(0,0), (0,1), (1,0), (1,1)} per bit. For unary operations like shifts or moves, only two patterns are sufficient. Examples of the pseudo-exhaustive test data for addition and subtraction operations are shown in Tables 4 and 5. Here the cases of ripple carry for addition and ripple borrow for subtraction are used. In general case, e.g. at carry-ahead-addition, the method of pseudo-exhaustive approach will be more complex, and more test data will be needed.

In Tables 4 and 5, all bits of all ALU operations f_i, i = 0,1,2,…., are tested exhaustively. For ADD and SUB operations, 8 data pairs are needed to cover all combinations of 3 inputs (two operands and carry/borrow bit) of each bit of the adder (sub-tractor). We start to generate patterns from the least significant bits, calculate the carry c_i for the next bit and fit for the next bit the values of operand bits a_i and b_i with the calculated carry c_i, so that all exhaustive combinations for this bit section were achieved. In such a way created patterns for the 2nd bit and 1st bit can be "copy-pasted" for the next two-bit sections to right.

Table 4. *Pseudo-exhaustive test data for addition operation*

No	…	4-bit $a_4\ b_4$ c_4	3-bit $a_3\ b_3$ c_3	2-bit $a_2\ b_2$ c_2	1-bit $a_1\ b_1$ c_1	0-bit $a_0\ b_0$ c_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	0 1 0	0 1 0	0 1 0	0 1 0	0 0 1
3	…	1 0 0	1 0 0	1 0 0	1 0 0	0 1 0
4	…	1 1 0	0 0 1	1 1 0	0 0 1	0 1 1
5	…	0 0 1	1 1 0	0 0 1	1 1 0	1 0 0
6	…	0 1 1	0 1 1	0 1 1	0 1 1	1 0 1
7	…	1 0 1	1 0 1	1 0 1	1 0 1	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

Table 5 *Pseudo-exhaustive test data for subtraction operation*

No	…	4-bit $a_4\ b_4$ c_4	3-bit $a_3\ b_3$ c_3	2-bit $a_2\ b_2$ c_2	1-bit $a_1\ b_1$ c_1	0-bit $a_0\ b_0$ c_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	1 1 0	0 1 1	1 1 0	0 1 1	0 0 1
3	…	0 0 1	1 0 0	0 0 1	1 0 0	0 1 0
4	…	1 0 0	1 1 0	1 0 0	1 1 0	0 1 1
5	…	0 1 1	0 0 1	0 1 1	0 0 1	1 0 0
6	…	1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
7	…	0 1 0	0 1 0	0 1 0	0 1 0	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

VIII. Composition of Test Routines for Cores

In accordance to the described method of generating test data for the testing the processor, we can divide the test patterns into two parts: (1) *conformity test patterns* which target the control faults, and (2) *scanning test patterns* which target the data faults of the core. Each test pattern $T_{i,j} = (I_i, d_{i,j,1}, d_{i,j,2})$ consists of an instruction pattern I_i, = ($opcode_i$, $A_{i,j,1}$, $A_{i,j,2}$) and two data operands d_{j1}, d_{j2} with addresses $A_{i,j,1}$, $A_{i,j,2}$, respectively. Denote for each instruction I_i the numbers of data operand pairs by c_i and s_i, for the conformity and scanning test parts, respectively. Since each instruction under test should be executed for all related conformity and scanning test patterns, we can represent the all test information as a set of $n + 1$ arrays (n is the number of instructions under test): the array I of n instruction patterns, and n data arrays with $n_i = c_i + s_i$, data patterns (operand pairs) each (see Fig.3).

From above, a test program structure results, which will consist of n loops where n is the number of tested instructions. The body of the i-th loop will consist of (1) the initialization sequence of instructions for loading the data d_{j1}, d_{j2} into the registers involved in the instruction I_i, (2) executing the instruction I_i, under test, and (3) observation sequence.

The whole test can be compressed by representing it as a program template consisting of two embedded loops, and n + 1 data arrays as program parameters: an array of n instructions, and n data arrays with $n_i = c_i + s_i$, data patterns (operand pairs) each (see Fig.4).

The first loop consists of test program cyclic accesses using indirect addressing mode to the array of instructions under test. In the body of the second internal loop of the test program, the data operands will be cyclically prepared for use by the current instruction under test. Observation and analyzing of test results can be implemented by software signature analyzer.

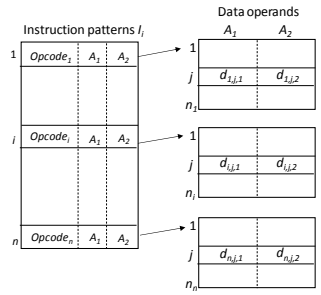


Fig.3. *Structure of the test information for self-test of cores*

For all instructions I_i, $i = 1,2, \ldots n$
  For all data operands $(d_{i,j,1}, d_{i,j,2}), j = 1,2, \ldots n_i$
    Read $d_{i,j,1}$
    Read $d_{i,j,2}$
    Execute the instruction I_i, = ($opcode_i$, $d_{i,j,1}$, $d_{i,j,2}$)
    Write the test result in signature analyzer
  End for data
End for instructions

Fig.4. *Test program template for self-test of cores*

The originality of the test strategy stands in on-line test generation method based on modifying on the fly the stored test program template which uses hierarchically organized test data.

The presented test program with embedded two loops and the hierarchical structure of the test information in form of instruction and data operand arrays allow splitting the test information to be used in the template in Fig. 4 in arbitrary ways into different segments (by specifying different values of n). This provides high flexibility for delivering test data for cores under test in the system and gives in such a way better possibilities for optimization of scheduling strategies of the test process at given constraints of currently running applications.

IX. EXPERIMENTAL RESULTS

The motivations of high-level test generation are threefold: generating test programs without knowing the details of circuits' implementations, speeding up the generation procedure, and increasing the quality of tests due to the possibility of covering multiple low-level faults without their counting. We carried out experiments with ALU sub-circuit of the VLIW processor [29].

The experimental results for the two proposed algorithms RANDOM and GREEDY for generating control part test, are depicted in Table 6, and Table 7. The results are compared with the reference method in [26]. The results in Table 6 correspond to the search space of 1500 random patterns. The not-100% SAF coverage is explained by the not tested faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$, which were not the target of the control test (Corollary 3), and which will be covered by the data path test.

Table 6. *Comparison of the control test algorithms*

Method	Test length, # instructions	Fault coverage		Time, s
		HL-FF	SAF	
RANDOM	204	100 %	99.34 %	2.00
GREEDY	139	100 %	99.34%	7.85
Method [26]	48	56.0 %	85.8 %	Manually generated
Gate-Level Deterministic	68	57.2%	100%	

Table 7. *Numbers of test operands for testing of the control path*

Op-code	# patterns			Op-code	# patterns		
	RANDOM	GREEDY	PS. D		RANDOM	GREEDY	PS. D
MOV	5	5	2	SHL	11	10	2
ADD	9	7	9	SHR	12	9	2
SUB	10	6	9	ASR	14	9	2
CMP	20	13	2	INC	16	6	3
AND	8	6	4	DEC	15	13	3
OR	20	15	4	RLC	20	13	2
XOR	9	6	4	RRC	20	13	2
NOT	9	4	2	NOP	6	4	2
	Total				204	139	54

The relationships between the fault coverage and test length are illustrated by the curves in Fig.5, and the dependence of test generation time on the size of search space (number of random test candidates) is illustrated in Fig.6.

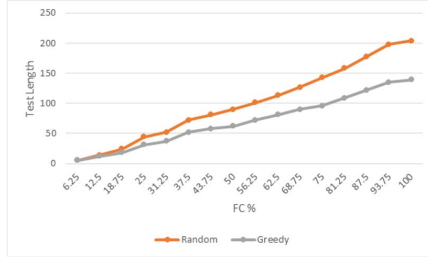


Fig.5. Dependence of the high-level control fault coverage on test length

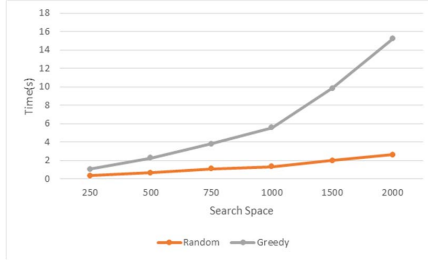


Fig. 6. *Test generation time and the size of search space*

The data path test with pseudo-exhaustive patterns had a length of 95 patterns and produced 99.1% gate-level fault coverage. Both control and data path tests together had a total length of 234 instructions and achieved 100% fault coverage for both high-level and gate-level faults. As a comparison, the referenced method [26] was able to achieve only 56.0 % high-level fault coverage. We also compared with gate-level deterministic approach which gives 100% single SAF coverage. However, because of very low high-level fault coverage this test gives no information about the quality of multiple fault detection.

X. CONCLUSIONS

In this paper, we propose a novel test program generation method which produces high fault coverage. The originality of the proposed method stands in on-line test generation based on modifying on the fly the stored test program template which uses hierarchically organized test data.

The well-structured test program provides high freedom of splitting it into segments, and therefore also high flexibility for delivering test data for cores under test in them system. This is a good basis for optimization of scheduling strategies of organizing test processes at given constraints of currently running applications.

High test quality is achieved by using a new functional model for high-level control faults in processor cores. The method does not need information about implementation details of the cores and uses only the instruction set as input data. We proposed a novel method to detect and prove the redundancy of high-level functional faults, which allows evaluation of the high-level fault coverage more precisely thanks to the possibility of removing redundant faults from the fault list. As the result, it was possible to prove the 100% high-level fault coverage for the control path test.

In [24] and [26], it was shown for the case of testing the ALU instructions that the bigger is the high-level fault coverage of the ALU control test, the bigger will be the confidence of covering multiple low-level faults in the control part of the processor. In accordance with that statement, we can now claim that the 100% high-level control fault coverage, achieved by the proposed test generation method, is equivalent to 100% coverage of low-level multiple faults from the broad class of SAF, conditional SAF, and bridging faults, with avoidance of mutual fault masking in ALU control.

The future work will be in extending the proposed method for the broader instruction sets, and for covering the faults in pipeline stages and addressing logic of processors.

Acknowledgment: The work has been supported by EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds. We thank Mario Schölzel from U Potsdam for providing us with VHDL description of the VLIW processor for carrying out the experiments.

REFERENCES

1. L. Zhang, Y. Han, Q. Xu, X. Li. Defect Tolerance in Homogeneous Manycore Processors Using Core-Level Redundancy with Unified Topology. Design, Automation and Test in Europe, 2008. DATE '08.
2. S. Premkishore, S. W. Keckler, C. R. Moore, D. Burger. Exploiting microarchitectural redundancy for defect tolerance. Proc. ICCD, pp. 481–488, 2003.
3. E. Schuchman, T. N. Vijaykumar. Rescue: a microarchitecture for testability and defect tolerance. Proc. ISCA, pp. 160–171, 2005.
4. A. Kamran, Z. Navabi. Self-Healing Many-Core Architecture: Analysis and Evaluation. VLSI Design Volume 2016, Article ID 9767139, http://dx.doi.org/10.1155/2016/9767139
5. M. A. Skitsas, C. A. Nicopoulos, M. K. Michael. Exploration of System Availability During Software-Based Self-Testing in Many-core Systems under Test Latency Constraints. 2014 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014.
6. J. D. Lee, R. N. Mahapatra, P. S. Bhojwani. A Distributed Concurrent On-Line Test Scheduling Protocol for Many-Core NoC-Based Systems. IEEE Int. Conf. on Computer Design – ICCD, 2009.
7. Y. Li, S. Mitra. VAST: Virtualization-Assisted Concurrent Autonomous Self-Test. Proc. International Test Conference (ITC), 2008, pp. 1-10.
8. S. Gurumurthy, S. Vasudevan, and J. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. ITC '06. Oct 2006, pp. 1–9.
9. D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi. IEEE Trans. on Systematic software-based self-test for pipelined processors. Vol. 16, no. 11, pp. 1441 –1453, Nov. 2008.
10. M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Reorda. Microprocessor software-based self-testing. IEEE Design Test of Computers. Vol. 27, no. 3, pp. 4–19, May 2010.
11. A. Apostolakis, D. Gizopoulos, M. Psarakis, A. Paschalis. Software-based self-testing of symmetric shared-memory multiprocessors. IEEE Trans. on Computers, vol. 58, no. 12, pp. 1682 –1694, Dec. 2009.
12. N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, A. Gonzalez. Self-test optimization in multithreaded multicore architectures. ITC 2010, Nov. 2010, pp. 1 –10.
13. Y. Li, O. Mutlu, S. Mitra. Operating system scheduling for efficient online self-test in robust system. Proc. 2009 Int. Conf. on Computer-Aided Design, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 201–208.
14. M. Agrawal, M. Richter, K. Chakrabarty. Test-Delivery Optimization in Manycore SOCs. IEEE Trans. on computer-aided design of integrated circuits and systems, vol. 33, no. 7, 2014.
15. D.Gizopulos, A.Paschalis, Y.Zorian. Embedded Processor-Based Self-Test. Kluwer Acad. Publishers, 2004, 216 p.
16. L.Chen, et al. A scalable software based self-test methodology for programmable processors," in Proc. of DAC, 2003, pp. 548 - 553.
17. L. Chen and S. Dey. SW-based self-test methodology for processor cores, IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, March 2001, pp. 369 - 380.
18. N.Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software based self-testing of embedded processors," in IEEE Trans. on Comp., vol.54, no.4, 2005.
19. Y.Zhang, H.Li, and X.Li. Automatic test program generation using executing-trace-based constraint extraction for embedded processors," in IEEE Trans. on VLSI Systems, vol.21, no.7, 2013.
20. N. Kranitis, et al "Hybrid-sbst methodology for efficient testing of processor cores," in IEEE Design and Test of Computers, vol. 25, no. 1, 2008, pp. 64-75.
21. C.-H. C. Tai-Hua Lu and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," IEEE Trans. on VLSI Systems, vol. 19, no. 3, March 2011, pp. 516 - 520.
22. C. H.-P. Wen, et al. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., Vol.55, No. 11, 2006.
23. A.Jasnetski et al. SW-based Self-Test Generation for Microprocessors with HLDDs. Proc. of the Estonian Academy of Sciences, 2014, 63, 1, 48-61.
24. A.Jasnetski et. al. High-Level Modeling and Testing of Multiple Control Faults in Digital Systems. Proc. of DDECS. Košice, Slovakia, April 20-22, 2016, 6p.
25. R.Ubar, M. Schölzel, S.A. Oyeniran, H.T. Vierhaus. Multiple Fault Testing in Systems-on-Chip with High-Level Decision Diagrams. 10th IEEE International Design & Test Symposium IDT'15, Dead Sea, Jordan,.December 14-16, 2015.
26. A.S.Oyeniran et al. A New Measure for Calculating Multiple Fault Coverage of Microprocessor Self-Test. BEC, Tallinn, Oct 3-5, 2016.
27. S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors, IEEE Trans. On Computers, C-29, No.6, pp.429-441, June 1980.
28. M.L.Bushnell, V.D.Agrawal. Essentials of Electronic testing. Kluwer Acad. Publishers, 2013.
29. M.Schölzel. Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis. Brandenburg University of Technology Cottbus-Seftenberg, 2015.

Appendix 5

V

A. S. Oyeniran and R. Ubar, "High-level functional test generation for microprocessor modules," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pp. 356–361, June 2019

Proceedings of the 26th International Conference *"Mixed Design of Integrated Circuits and Systems"*
June 27-29, 2019, Rzeszów, Poland

High-Level Functional Test Generation for Microprocessor Modules

Adeboye Stephen Oyeniran, Raimund Ubar
Tallinn University of Technology
Tallinn, Estonia

***Abstract*—A new high-level implementation-independent and automated test program generation method for RISC processors is proposed. For testing the control parts of the processor modules, a novel high-level control fault model is proposed. Using this model, a set of deterministic test data operands are generated for each instruction of the instruction group using the module under test. For generating tests for the data path of the module under test, pseudo-exhaustive patterns are used. The set of test data is generated in two parts. The first part is based on the deterministic test data generated for the control test, and the second part is formed by the sets of pseudo-exhaustive test data operands, generated for each instruction of the group separately. We investigated the feasibility of the approach and demonstrated high gate-level stuck-at-fault coverage of the generated test programs for testing the execute module of the MiniMIPS RISC processor.**

***Keywords*—RISC processor testing, high-level control faults, high-level test data generation**

I. Introduction

Software-Based Self-Testing (SBST) [1-3] is an emerging paradigm in the testing domain, which relies on the exploitation of existing available resources resident in the system. The SBST approach is based on software programs that are designed to test the functionality of the processor cores. The major problem with SBST is usually the not sufficient fault coverage achieved. The hardware overhead is either non-existent, or negligible, and no Instruction Set Architecture (ISA) extensions are required.

The quality of SBST is mainly affected by test data used in test programs. One of the ways to obtain test data is executing an Automated Test Pattern Generator (ATPG). In [4] it was shown that the processor can be divided into Modules under Test (MUT) to ease the task of ATPG. On the other hand, the difficulties of the method arise from the need of guiding ATPGs by functional constraints to produce functionally feasible test patterns. An alternative way is to use random test patterns for MUTs [5].

SBST approaches can be structural and functional. Structural approaches [6-9], are based on test generation using information from lower level of design (gate-level or RTL-level description) of processors, whereas, functional approaches use mainly instruction set architecture (ISA) information. In [8], shifting of SBST generation from gate-level to Register-Transfer Level (RTL) was suggested. The drawback of this method is that it has not been shown how to achieve high fault coverage of low-level structural faults.

Hybrid SBST methods were proposed for combining deterministic structural SBST with verification-based self-test codes [3, 8, 10, 11]. In addition to Hybrid SBST [12, 13], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [7, 10].

The structural approaches cannot be used when the structural information about the processors to be tested is not available.

One of the first ISA based methods, using pseudo-random test sequences was proposed in [14]. Another solution, FRITS (Functional Random Instruction Testing at Speed) [15], was based on test program generation on random instruction sequences with pseudo-random data. It suits well for wafer test, due to its cache-resident nature. Alternative cache-resident method for production testing [16] using random generation mechanism proves that high cost functional testers can be replaced by the low-cost SBST without significant loss in fault coverage. Another approach, based on evolutionary technique was proposed in [17]. Test program is being composed of the most effective code snippets (in a question of SAF coverage), which were distinguished by constant re-evaluation. The method, however, is based on structural information.

Later research concentrates on test approaches for specific processor parts like pipeline, branch prediction mechanism [18-19] or caches [20-21]. In [22], a method is proposed, which can enhance SBST program in order to bring more coverage to pipeline logic and also memory addressing. Another approach for testing the pipeline was made in [23]. The proposed strategy involves the activation of faults related to the data hazards and register forwarding logic in processor core, and later research concentrates on decode stage of the pipeline [24].

The drawbacks of the known methods are: the fault coverage is traditionally measured only with respect to SAF, no broader fault classes have been considered, and no attempts have been made to evaluate the test quality regarding covering of multiple faults with avoidance of fault masking.

To cope with the complexity of gate or RT level representations of microprocessors (MP), we consider the problem of SBST program generation with focus on modeling functional faults fully at the behavioral-level using only Instruction Set Architecture (ISA). At the same time, for the purpose of evaluating the quality of tests generated at high-level, we target a broader class of faults than SAF, e.g. considering also conditional SAF and bridging faults as well.

In this paper, we propose a novel deterministic high-level test generation method for SBST of processor cores which is based on a novel implementation-free high-level functional fault model. The idea is to represent the information given in

the instruction set in the form of High-Level Decision-Diagrams (HLDD) [25-26]. HLDDs are used as a convenient and strict mathematical structure for representing all control functions of the given processor core using the same formalism, and to develop uniform high-level fault model for organizing efficient test programs. Moreover, this formalism allows the first time automation of the high-level test program synthesis for microprocessor cores. The experimental results of test generation for the RISC type microprocessor MiniMIPS [27] demonstrate high gate-level Stuck-at-Fault (SAF) coverage despite of not taking into account in test generation the details of actual implementation of circuits under test.

The rest of the paper is organized as follows. In Section 2, we present the main conception of modeling microprocessors with HLDDs. In Section 3, we develop a method for high-level fault modeling, and in Section 4, we propose a method for generating test data guided by the high-level fault model. In Section 5, we present the general structure and composition of the full test program synthesized using the proposed new test method. Section 6 presents experimental data, and Section 7 concludes the paper

II. Modeling of Microprocessors with HLDDs

The microprocessor as a digital system can be regarded, in a general case, as a network of nodes, where each node represents a module, which executes either data manipulation, data transfer, or data storage function. Each node has data and control inputs, as arguments for the executed function. The output of a node serves as input for other nodes. In the following, to represent the functions of nodes in the system network, we use high-level decision diagrams (HLDD) [25,26].

Consider, as an example in Table I the instruction set of a hypothetical microprocessor where each of 8 instructions is represented by a complex instruction variable I as a concatenation of 5 sub-variables $I = OP.B.A1.A2.A$. Here, OP and B denote two fields of the operation code, $A1$ and $A2$ are register addresses, and A is the memory address. The column of semantics in Table I lists all the functions executed in the system network, which is presented in Fig. 1, and consists of the following nodes (modules): I – control module, M – memory, R – register block, ALU – execution module, and PC – program counter. The modules are controlled by the control signals (shown with red color), decoded from the instruction word. The data manipulation functions executed in the modules, are represented by 5 HLDDs in Fig. 2.

TABLE I.
Instruction set of a microprocessor

OP	B	Mnemonic	Semantics and RT level operations
0	0	LDA A1, A	READ: $R(A1) = M(A)$, $PC = PC + 2$
	1	STA A2, A	WRITE: $M(A) = R(A2)$, $PC = PC + 2$
1	0	MOV A1,A2	TRANSFER: $R(A1) = R(A2)$, $PC = PC + 1$
	1	CMA A1,A2	COMPLEMENT: $R(A1) = \neg R(A2)$, $PC = PC + 1$
2	0	ADD A1,A2	ADD: $R(A1) = R(A1) + R(A2)$, $PC = PC + 1$
	1	SUB A1,A2	SUBTRACT: $R(A1) = R(A1) - R(A2)$, $PC = PC + 1$
3	0	JMP A	JUMP: $PC = A$
	1	BRA A	Conditional jump (Branch instruction): IF C=1, THEN $PC = A$, ELSE $PC = PC + 2$

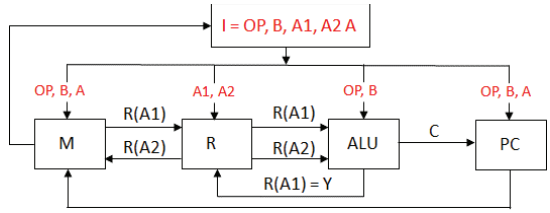


Figure 1. Network of computing nodes for the instruction set in Table I.

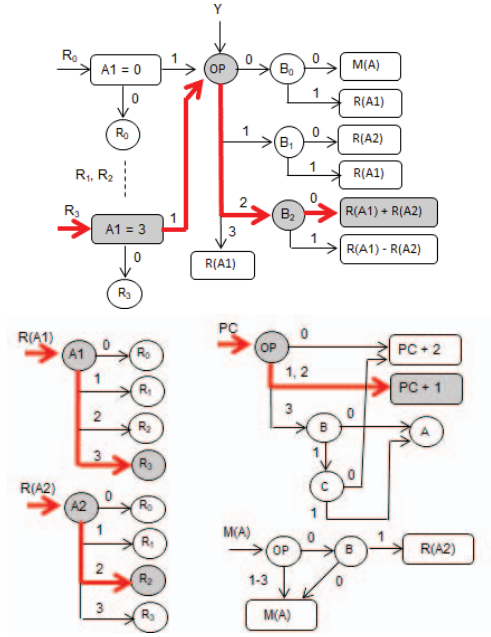


Figure 2. HLDDs for the processor described in Table I.

The HLDDs represent a data structure used for simulation and test generation purposes. Each module contains of the control and data part. The internal nodes of HLDDs represent the control part, whereas the terminal nodes represent the data part. The HLDDs have entry edges, which are labeled by the functional variable (called HLDD variable), whose value can be calculated using this graph. Simulator traverses the graph, according to the control values decoded from instruction word, and the HLDD variable will be assigned to the value calculated by the function in the terminal node where the traversing procedure is ended. The red edges in the graphs illustrate the traversed paths for the instruction: OP=2, B=0, $A1$=3, $A2$=2. The following values are calculated for this instruction on the HLDD model: $R(A1) = R_3$, $R(A2) = R_2$, $Y = R(A1) + R(A2)$ and $R_3 = R(A1) = Y$. The content of the program counter will be as well updated: $PC = PC + 1$.

Note, the upper HLDD in Fig. 2 has 4 entry edges where the graphs with entries R_0, R_1, R_2, R_3, are sharing the same subgraph Y which represents the functions of the ALU module.

As an example, the HLDD in Fig. , represents the execution module of the microprocessor MiniMIPS, which was used in this paper for carrying out test generation experiments. According to Fig. 3, the result of each operation, calculated using the graph, is stored in the general-purpose register R_0 decoded by A_0. The red colored lines show the path activated by the MiniMIPS instruction "and $1, $2, $3", which calculates the value $1 = $2 & $3, using the notation on the HLDD: R0 = $1, rs = $2, rt = $3.

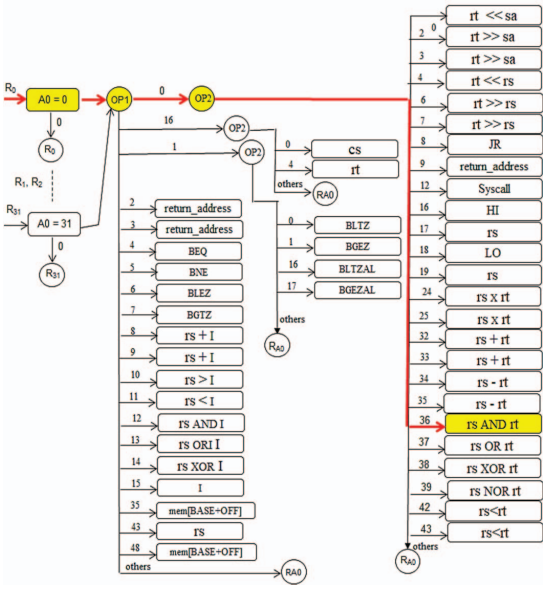


Figure 3. HLDD for the execution module of miniMIPS microprocessor

III. The Concept of High-Level Fault Modeling in Modules of Processor Cores

Each module of the microprocessor network consists of the data path and control path. Consider such a module in Fig. 4 represented by a single k-th bit slice of the full m-bit circuit, where m is the width of the data word, and the corresponding HLDD model, which describes the high-level behavior of the circuit. The data manipulation block (ALU in this case) executes $n + 1$ different functions f_i, selected by the control codes c denoted by integers 0, 1, … n+1. Denote the full set of functions by F.

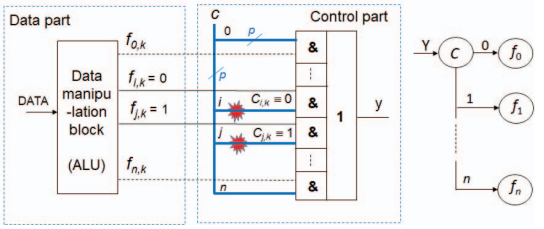


Figure 4. A module consisting of data and control parts and its HLDD

Each bit-slice of the control part consists of the multiplexer MUX with p control lines as control inputs to each AND gate, and a single 1-*bit* data line from the data manipulation block as the data input to each AND gate, whereas the OR gate has n+1 data inputs connected to the outputs of AND gates.

Having described the modules of processor cores as a composition of control and data part, we can better understand now the meaning of HLDDs described in the previous Section: the internal nodes of HLDD represent the MUXes of the control part, and all the functions labelling the terminal nodes, are describing the functionality of the data part.

For example, the HLDD in Fig.3, models two MUXes for decoding the two control fields *op*1 and *op*2 in the instruction format of the microprocessor MiniMIPS.

According to such a mapping of the system functions into the HLDD model, we can classify two types of HLDD-based high-level functional fault models: *control faults* (the faults related to the non-terminal nodes), and *data faults* (the faults related to the terminal nodes).

Denote by y_i the data word considered as the result of execution of the function f_i with data operand(s) d_i as $y_i = f_i(d_i)$.

Definition 1. Introduce for the function (instruction) $f_i \in F$, the following *high-level control fault model* $M(f_i)$ as a set of data operands $M(f_i) = \{D_i\}$, which satisfy the following constraints at least once for each bit k of y_i:

$$\forall f_i \in F:\ \forall k \in (1,m):\ \{\exists d_i \in M(f_i)\ (y_{i/k} \neq 0)\}, \qquad (1)$$

$$\forall f_j \in F, j \neq i:\ \forall k \in (1,m)\{\exists d_i \in M(f_i)\ (y_{i/k} < y_{j/k})\} \qquad (2)$$

Depending on the technology, implemented in the microprocessor, the constant 0 in formula (1) can be changed into 1, and instead of the relation " < " in formula (2), there can be " > ".

Note, if the HLDD has more than one internal nodes, then for each non-terminal node m the test is generated separately, and a set of functions $F(m)$ for that node m under test is set up, so that to each output of the node m, a single terminal node m^T (having a path from m to m^T) with related function f_j is mapped, so that $f_j \in F(m)$.

When examining the roles of the constraints (1) and (2) we can state (using the notations in Fig. 4) the following:

(1) if the constraints (1) for $y_{i/k}$ are satisfied, then the faults $f_{j,k} \equiv 0$ and $c_{j,k} \equiv 0$ are detected (see Fig. 4);
(2) if the constraints $y_{i/k} < y_{j/k}$ (2) are satisfied, then the faults $f_{i,k} \equiv 1$ and $c_{j,k} \equiv 1$ are detected (see Fig. 4).

The main idea of testing the high level control faults is in converting the task for testing the DNF of the control function described as a MUX. It can be easily shown that satisfication of constraints (1) and (2) guarantees testing of all SAF in the MUX. The circuit in Fig. 4 can be regarded as representation of the equivalent disjunctive normal form EDNF. In [28] it was shown that the test, which detects all SAF in the circuit described as EDNF, will detect also all faults in any optimized circuit which is equivalent to the original EDNF.

This fact allows us to claim that the control test satisfying the constraints (1) and (2) is an implementation-free test.

Definition 2. As the high-level fault universe for terminal nodes f_j., we use the functional fault model defined as the sets of test data $D(f_j)$ needed for testing the functions f_j.

Corollary 1. The control test, which satisfies the constraints (1) and (2), detects all multiple SAF, conditional SAF and bridging faults in the control part of the module under test.

Proof. The proof is straightforward, and is based on the fact that first, each AND channel of the MUX of the control part is a tree-like circuit, and second, this tree-like circuit is tested exhaustively.

For generating the test data $D(f_j)$ for the data part, we apply pseudo-exhaustive testing approach which is also independent on the implementation details.

Note, when testing the function $f_i \in F$ in the data path then also the faults $f_{i,k} \equiv 0, f_{i,k} \equiv 1$ and $c_{i,k} \equiv 0$ in the control path are tested. From that it results, that for evaluating the quality of the control test, only the coverage of constraints (2) should be calculated.

IV. Test Data Generation for Testing the Control Parts of Processor Modules

From above, it follows that the fault model defined by the set of constraints in (1) and (2) can be interpreted as the definition of the universe of high-level faults. A direct impact of this interpretation is the possibility of evaluating the high-level functional fault coverage as the percentage of constraints in (2) for the given test. The fact that the faults $f_{i,k} \equiv 0$ and $c_{i,k} \equiv 0$ in the control path will not be taken in the fault universe of the control faults can be overseen because, according to Corollary 3, these faults will be covered anyway as the byproduct by the data path test $T(f)$.

In the following, we present an algorithm for generating the set T(c) of test data for testing the control path.

Algorithm 1. Test generation for control part of the module

Input: Instruction set of the processor
Output: Sets of test operands OP_i for each instruction, and fault table D
Notations: n – number of instructions (functions F_j), op – test operand, OP – current set of selected random test operands, $f_j(op)$ – the result of the instruction I_j for the operand(s) op, D – fault table, D_{ij} – w-bit entry in D (w – length of the data Word).

1 Initialize $OP = \varnothing$
2 Generate a set of R random operands
3 **for** $i = 1, \ldots, n$
  ***generation of operands for instruction I_i
4 Initialize $OP_i = \varnothing$,
5 **for** $j = 1, \ldots, n$ ($j \neq i$)
    ***operands for solving constraints $f_{i,k} < f_{j,k}$
6 Initialize $D_{ij} = 0$
7 **for** all $op \in R$ **while** $D_{ij} \neq 0$
      ***adding new operands for covering D_{ij}
8 $D_{ij}(op) = f_i(op) \wedge (f_i(op) \oplus f_j(op))$ ***
      calculating fault coverage for op
9 **if** $(D_{ij}(op) \vee D_{ij}) \oplus D_{ij} \neq 0$ **then**
        *** check for the coverage increment
10 **begin**
11 $D_{ij} = D_{ij} \vee D_{ij}(op)$
        *** update of the coverage vector
12 Include op into OP_i
        *** new operand is selected
13 **end**
14 **endfor** op
15 **endfor** j
16 **endfor** i

The result of Algorithm 1 will be a set of operands OP_i for each instruction I_j, and the fault table $D = \| D^k_{ij} \|$ where $D^k_{ij} = 1$ means that the functional fault described by the constraint $f_{i,k} < f_{j,k}$ is covered at least by one operand $op \in OP_i$, otherwise $D^k_{ij} = 0$. The percentage of 1s in D is the high-level functional fault coverage of the test for control path.

The test data according to Algorithm 1 are generated randomly: in each step 7, the first randomly produced operand $op \in R$ will be chosen which produces an increase in the fault coverage.

The constraints $f_{i,k} < f_{j,k}$ may not be solved for two reasons: either the related functional fault is redundant, or the search space R is not big enough.

In case of testing the addressing functions for accessing registers (or memory locations) the test data to satisfy the constraints (2), due to the independence of the bits of data words, can be found straightforwardly. It is sufficient to generate test data in such a way that for each pair of addresses (i,j) we will have the test data, so that $R_{i,k} < R_{j,k}$ were satisfied for each data bit, as shown in Table II.

TABLE II.
Data for Register Decoder Test

Register	Data 1	Data 2	Data 3	Data 4
R0	0000	0000	1111	1111
R1	0000	1111	1111	0000
R2	1111	0000	0000	1111
R3	1111	1111	0000	0000

The test generation problems discussed (the ALU control, and register addressing) belong to the combinational problem where a single test instruction is needed after the initialization phase is finished. In general case, the test may need several instructions. This case is illustrated in Fig.5, where a fragment of the execution stage of the pipeline is shown.

The execution stage is placed between pipeline registers ID/EX and EX/MEM. Assume, there are two instructions in a sequence: the first is writing the value f_j calculated in ALU into the register $R(b)$ with adress b, and the next instruction must use this value. Instead of stalling the pipeline till f_j will be written in to the register block to make it accessible for the second instruction, f_j can be forwarded immediately in the next clock from EX/MEM to ALU via MUX controlled by the signal B ($B = 1$, if $a = b$). To test the forwarding control unit, we can use the same data constraints (1) and (2), developed in Section IV with the difference that the values of f_i and f_j are stored not in the register block, rather in the different pipeline registers. Hence, the test program should consist now two subsequent instructions instead of the single instruction as discussed earlier.

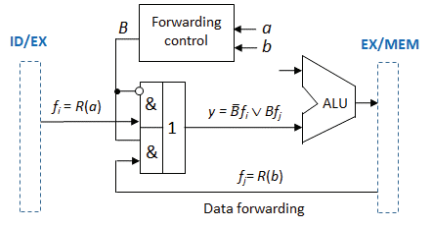


Figure 5. Data forwarding in the execution stage of a pipeline circuit

Note, since the pipeline architecture is a part of the processor core, which is not described in the instruction list, the HLDD model should be generated using additional information about the high-level architecture of the processor. Such an additional knowledge, does not need to include information about the low level imlementation details.

V. Composition of Test Routines for Cores

In accordance to the described method of generating test data for testing the processor cores, we can divide the test patterns into two parts: (1) *conformity test* for detecting the control faults, and (2) *scanning test* for testing the data faults.

Let us design the test program and the test data structure using the following abstract concept: each test pattern $T_{i,j} = (I_i, d_{i,j,1}, d_{i,j,2})$ will consist of an instruction pattern I_i, = ($opcode_i$, $A_{i,j,1}$, $A_{i,j,2}$) and two data operands d_{j1}, d_{j2} with addresses $A_{i,j,1}$, $A_{i,j,2}$, respectively. Denote for each I_i the numbers of data pairs by c_i and s_i, for the conformity and scanning tests, respectively.

Since each instruction is executed for all conformity and scanning test patterns, we can represent the test information as a set of $n + 1$ arrays (n is the number of instructions under test): the array I of n instruction patterns, and n data arrays with $n_i = c_i + s_i$, data patterns (operand pairs) as shown in tab. 6). From above, a test program structure results, which will consist of two embedded loops. The first external loop consists of the cyclic access to the instructions under test. In the body of the second internal loop, the data operands are initialized, instruction under test is executed, and observation procedure is carried out.

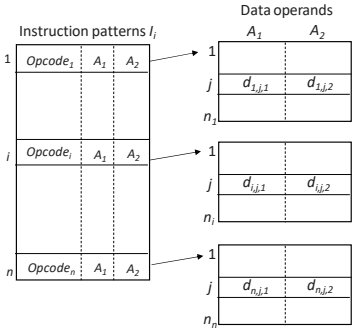


Figure 6. Structure of the test information for self-test of cores

VI. Experimental Results

The experiments targeted investigations of the feasibility and efficiency of the developed high-level test program generation method. The objective of test experiments was the execute stage of MiniMIPS processor [27], particularly the ALU module, and two multiplication modules MULT0 and MULT1. The test program generation included automatic synthesis of test templates from manual parameter file, automated high-level test data generation to satisfy the constraints (1) and (2). The test generation was carried out at the functional high-level without knowing the details of implementation. However, to compare the results with state-of-the-art commercial gate-level ATPG, the test quality evaluation for our test program had to be carried out by fault simulation at gate-level. For that purpose, we synthesized with Synopsys synthesis tool a gate-level implementation of the execute stage of MiniMIPS processor, and calculated with commercial fault simulation tool the gate-level SAF coverages for our high-level generated test program.

The experimental research targeted 25 instructions $I_i \in I$ out of MiniMIPS 51 instructions, as the basis of the set of functions $F = \{f_i\}$ investigated in the paper. The results are shown in Table III. The high-level fault model for the control part of tested modules consisted 756 high-level faults (data constraints solved) which all were covered. The high-level test includes 166 patterns which is less than test length of 957 patterns generated with commercial ATPG. This is due to the compact form of presentation. However, the number of clock cycles of the high-level test exceeds about 5 times that of the ATPG test. The test quality of the proposed test in terms of SAF coverage is better, and the high-level test generation time 47s is dramatically less compared with the time cost of using commercial gate-level ATPG.

TABLE III.
Experimental data

Method	Experiments		Faults	FC%	Stored pat	Exec. pat	ATPG time
New high-level method	High-level ATPG		756	100	166	4818	47 s
	Gate-level simu-lation	Adder	2516	99.92			
		MULT0	95188	99.52			
		MULT1	91810	99.16			
		ALU	18954	99.06			
Commercial gate-level ATPG		ALU	18954	97.73	957	957	8h 27 min
		Adder	2516	99.96			
		MULT0	95188	97.40			
		MULT1	91810	97.71			

In Table IV, we compare our approach with other state-of-the-art SBST methods published in [22] and [29] for the ALU and for the EX stage of the pipeline in MiniMIPS. We see the advantages of the proposed method regarding higher SAF fault coverage.

TABLE IV.
Fault coverage on miniMIPS by different SBSST approaches (EX and ALU)

Module/Unit	Systematic SBST [22]	ATIG [29]	Proposed method
ALU	97.85	98.67	99.06
PPS EX	84.12	97.62	97.96

VII. Conclusions

In this paper, we proposed a novel test program generation method for microprocessor cores, which produces high fault coverage. High test quality was achieved due to using a new functional model for high-level control faults in processor modules. The method does not need information about implementation details and uses only the instruction set as input data.

Experimental results for selected modules of the execute module of MiniMIPS processor show that the high-level generated test program has several advantages compared to the

commercial ATPG tool: (1) the test generating is easier due to the reduced complexity of the model used (756 high-level faults vs. 189514 gate-level faults); (2) gate-level fault coverage is higher (99.06% vs. 97.73%), and (3) time cost for test generation is several orders of magnitude less (47s vs. 8h 27 min).

Additional added value of the proposed method is in proven 100% fault coverage for the control parts of modules regarding the broader class of faults: single SAF, multiple SAF, conditional SAF and bridging faults. Traditional fault simulation tools do not provide information about the coverage of multiple SAF and conditional SAF.

Experiments were carried out co far only for the execute module of the processor, however the applicability of the method was shown also for the register and memory addressing modules and for the pipe-line circuitry.

The theoretical basis of the work is the High-Level Decision-Diagram model, which allows straightforward automation of the test program generation procedure

The future work will be in extending the proposed method for the broader instruction sets, and for experimental research of applying the HLDD-based method for pipeline stages and addressing logic of processors.

ACKNOWLEDGMENT

The work has been supported by EU's H2020 projects TUTORIAL and RESCUE, Estonian research grant IUT 19-1, and funded by Excellence Centre EXCITE in Estonia. We thank you also our former MSc student Olusiji Oloruntobi Medaiyese for carrying out a part of experiments.

REFERENCES

[1] L.Lingappan, N. K. Jha. Satisfiability-based automatic test program generation and design for testability for microprocessors. IEEE Trans. on VLSI Systems, vol.15, no.5, pp. 518–530, 2007.

[2] C.H.Wen, L.-C.Wang, K.-T.Cheng. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., vol.55, no.11, 2006.

[3] S.Gurumurthy, S.Vasudevan, J.A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. IEEE International Test Conference, 2006.

[4] L.Chen, et al. A scalable software based self-test methodology for programmable processors," in Proc. of DAC, 2003, pp. 548 - 553.

[5] L. Chen and S. Dey. SW-based self-test methodology for processor cores, IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, March 2001, pp. 369 - 380.

[6] L.Lingappan, N. K. Jha. Satisfiability-based automatic test program generation and design for testability for microprocessors. IEEE Trans. on VLSI Systems, vol.15, no.5, pp. 518–530, 2007.

[7] C.H.Wen, L.-C.Wang, K.-T.Cheng. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., vol.55, no.11, 2006.

[8] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software based self-testing of embedded processors. IEEE Trans. on Comp., vol.54, no.4, 2005.

[9] C.H.Chen, C.K.Wei, T.H.Lu, H.W.Gao. Software-based self testing with multiple-level abstractions for soft processor cores. IEEE Trans on VLSI Systems, vol.15, no.5, pp. 505–517, 2007.

[10] Y.Zhang, H.Li, and X.Li. Automatic test program generation using executing-trace-based constraint extraction for embedded processors," in IEEE Trans. on VLSI Systems, vol.21, no.7, 2013.

[11] N. Kranitis, et al "Hybrid-sbst methodology for efficient testing of processor cores," in IEEE Design and Test of Computers, vol. 25, no. 1, 2008, pp. 64-75.

[12] C.-H. C. Tai-Hua Lu and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," IEEE Trans. on VLSI Systems, vol. 19, no. 3, March 2011, pp. 516 - 520.

[13] C. H.-P. Wen, et al. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., Vol.55, No. 11, 2006.

[14] J.Shen, J.A.Abraham. Native mode functional test generation for processors with applications to self test and design validation. Int. Test Conference, 1998..

[15] P.Parvathala, K.Maneparambil, W.Lindsay. Frits - a microprocessor functional bist method. International Test Conference, 2002, pp. 590–598.

[16] I.Bayraktaroglu, J.Hunt, D. Watkins. Cache resident functional microprocessor testing: Avoiding high speed io issues. IEEE Int. Test Conference, 2006.

[17] F.Corno, E.Sanchez, M.S.Reorda, G.Squillero. Automatic test program generation: a case study. IEEE Design Test of Computers, vol.21, no.2, 2004.

[18] D.Changdao, M.Graziano, E.Sanchez, M.Sonza Reorda, M. Zamboni, N.Zhifan. On the functional test of the BTB logic in pipelined and superscalar processors. LATW, 2013.

[19] E. Sanchez and M. S. Reorda. On the functional test of branch prediction units. IEEE Trans. on VLSI Systems, vol.23, no.9, 2015, pp. 1675–1688.

[20] S. D. Carlo, P. Prinetto, and A. Savino. Software-based self-test of setassociative cache memories. IEEE Trans. on Computers, vol.60, no.7, 2011, pp. 1030–1044.

[21] J.Perez Acle, R.Cantoro, E.Sanchez, M.Sonza Reorda. On the functional test of the cache coherency logic in multi-core systems. LATS, 2015.

[22] D.Gizopoulos, M.Psarakis, M.Hatzimihail, M.Maniatakos, A.Paschalis, S. Ravi, A.Raghunathan. Systematic software-based self-test for pipelined processors. IEEE Trans. on VLSI Systems, vol.16, no.11, 2008, pp.1441–1453.

[23] P.Bernardi, R.Cantoro, L.Ciganda, B.Du, E.Sanchez, M.S.Reorda, M.Grosso, O.Ballan On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors. 14th Int. Workshop on Microprocessor Test and Verification, Dec 2013, pp. 52–57.

[24] P.Bernardi, R.Cantoro, L.Ciganda, E.Sanchez, M.S.Reorda, S.D.Luca, R.Meregalli, A.Sansonetti. On the in-field functional testing of decode units in pipelined risc processors. IEEE Int Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems. 2014, pp. 299–304.

[25] A.Karputkin, R.Ubar, J.Raik, M.Tombak. Canonical Representations of High Level Decision Diagrams. Estonian Journal of Engineering, Vol. 16, Issue 1, 2010, pp.39-55.

[26] R.Ubar, A.Tsertov, A.Jasnetski, M.Brik. Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams. IEEE LATW-2014. Fortaleza, Brazil, March 12-15, 2014.

[27] OpenCores, "MiniMIPS ISA"

[28] D.B.Armstrong. On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets. IEEE Trans. on Electronic Computers, v.EC-15, no.1,1966 pp.66-73.

[29] Ying Zhang, Huawei Li, Xiaowei Li. Automatic Test Program Generation Using Execution-Trace Based Constraint Extraction for Embedded Processors. IEEE Transactions on Very Large Scale Integration(VLSI) Systems, vol.21, no. 7, pp.1220-1233, 2013.

Appendix 6

VI

A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "High-level combined deterministic and pseudo-exhuastive test generation for risc processors," in *2019 IEEE European Test Symposium (ETS)*, pp. 1–6, May 2019

2019 24th IEEE European Test Symposium (ETS)

High-Level Combined Deterministic and Pseudo-exhuastive Test Generation for RISC Processors

Adeboye Stephen Oyeniran
Department of Computer Systems
Tallinn University of Technology
Estonia
adeboye.oyeniran@taltech.ee

Raimund Ubar
Department of Computer Systems
Tallinn University of Technology
Estonia
raiub@pld.ttu.ee

Maksim Jenihhin
Department of Computer Systems
Tallinn University of Technology
Estonia
maksim@pld.ttu.ee

Cemil Cem Gürsoy
Department of Computer Systems
Tallinn University of Technology
Estonia
cemil@ati.ttu.ee

Jaan Raik
Department of Computer Systems
Tallinn University of Technology
Estonia
jaan@pld.ttu.ee

***Abstract*—Recent safety standards set stringent requirements for the target fault coverage in embedded microprocessors, with the objective to guarantee robustness and functional safety of the critical electronic systems. This motivates the need for improving the quality of test generation for microprocessors. A new high-level implementation-independent test generation method for RISC processors is proposed. The set of instructions of the processor is partitioned into groups. For each group, a dedicated test template is created, to be used for generating two test programs, for testing the control and the data paths respectively. For testing the control part, a novel high-level control fault model is proposed. Using this model, a set of deterministic test data operands are generated for each instruction of the given group. The advantage of the high-level fault model is that it covers larger than SAF fault class including multiple fault coverage in the control part. For generating the data path test, pseudo-exhaustive data operands are used. We investigated the feasibility of the approach and demonstrated high efficiency of the generated test programs for testing the execute module of the miniMIPS RISC processor.**

Keywords— RISC processors, high-level fault model, high-level test generation, deterministic and pseudo-exhaustive tests, control and data path tests

I. INTRODUCTION

Despite the fact that test generation for embedded processor cores of digital systems is a problem intensively investigated during decades in the test community, there is still a need for improvements in fault coverage and speed of test program generation in cases where no information about the details of implementation is given.

For the last decade, there has been an extensive research on Software-Based Self-Test (SBST) of processors [1-12]. The general idea of SBST is to use the resources of processors to test themselves, by running specific test programs. The nature of this method implies such features as non-intrusiveness, low cost and compatibility with at-speed and in-field testing [4-5]. SBST method is well accepted in industry. The interest in this method is growing in frames of in-field test for processor-centric systems in safety-critical applications [5-6]. Recent application domain standards, e.g. ISO26262, IEC61508, DO0254 set very stringent requirements for the target fault coverage in embedded microprocessor circuits, with the objective of guaranteeing robustness and functional safety of the critical electronic systems. Hence, more effort is being put into SBST for in-field test to satisfy these requirements. It is interesting to note at this point that one of the benefits of automated SBST is in reduction in test development cost [6-7].

SBST approaches can be structural and functional. Structural approaches [8-12], are based on test generation using information from lower level of design (gate-level or RTL-level description) of processors, whereas, functional approaches use mainly instruction set architecture (ISA) information. The structural approaches cannot be used when the structural information about the processors to be tested is not available. One of the first ISA based methods, using pseudo-random test sequences was proposed in [13]. Another solution, FRITS (Functional Random Instruction Testing at Speed) [14], was based on test program generation on random instruction sequences with pseudo-random data. It suits well for wafer test due to its cache-resident nature. Alternative cache-resident method for production testing [15] using random generation mechanism proves that high cost functional testers can be replaced by the low-cost SBST without significant loss in fault coverage. Another approach, based on evolutionary technique was proposed in [16]. Test program is being composed of the most effective code snippets (in a question of SAF coverage), which were distinguished by constant re-evaluation. The method, however, is based on structural information.

Later research concentrates on test approaches for specific processor parts like pipeline, branch prediction mechanism [17-18] or caches [19-20]. In [21], a method is proposed, which can enhance SBST program in order to bring more coverage to pipeline logic and also memory addressing. Another approach for testing the pipeline was made in [22]. The proposed strategy involves the activation of faults related to the data hazards and register forwarding logic in processor core, and later research concentrates on decode stage of the pipeline [5]. A variation of on-line SBST with the objective of enhancing lifetime reliability was proposed in [31].

In this paper, we propose a novel deterministic high-level test generation method for SBST of embedded processors which is based on a novel implementation-free high-level functional fault model. The advantage of the model is higher fault class than the well measurable standard single SAF, covering as well bridging and multiple SAF faults in the

978-1-7281-1173-5/19/$31.00 ©2019 IEEE

control part. The determinism of the fault model stands in a novel proposed set of data constraints to be satisfied by generating data operands to be used with instructions under test. For testing the data-path, pseudo-exhaustive data operands are used. Experimental result shows that the data constraints proposed for the control test contributes also noticeably to reaching high SAF coverage for the data-path test.

The rest of the paper is organized as follows. In section 2, we present a novel high-level control fault model for microprocessors, and in section 3, we investigate the problem of mapping the high-level fault model to low gate-level faults. In section 4, we present a fault simulation algorithm, and discuss the problems of high-level fault coverage measurement. Section 5 is devoted to the overall composition of test programs. In section 6, we present experimental data, and section 7 concludes paper.

II. HIGH-LEVEL CONTROL FAULT MODEL FOR PROCESSORS

The purpose of this research is to propose a novel method for testing RISC microprocessors in a functional way and without resorting to the knowledge of implementation details.

The main concept of the proposed method is based on partitioning the set of instructions of the processor under test into groups which can be tested by test templates which includes initialization, instruction under test, and observation of the results, in a similar way as in [5]. In this paper, we focus on testing of the executing units in pipelined RISC processors consisting of a control part and data path as shown in Fig.1. The method can be generalized also for testing other specific parts of microprocessors, such as other pipeline stages, register decoding, flag testing, branch prediction mechanism etc.

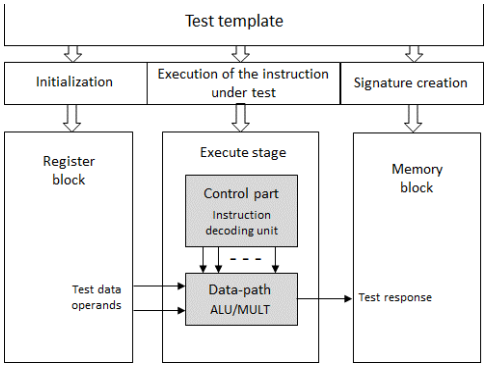


Fig.1. *Test execution set up*

The gray part of Fig.1 presents the test target which is the goal of this research. In Fig.2, we represent the execute unit in an implementation-free generic way as an equivalent circuit where the control part is highlighted as AND-OR multiplexer for decoding the instructions and extracting the results of the executed instructions. The circuit in Fig.2 represents equivalent disjunctive normal form (EDNF) related to the execute unit. The independence from implementation details results from the fact that a test developed for detecting all non-redundant faults in the EDNF, will also detect all faults in the original circuit [27]. Moreover, the exhaustiveness of the control signals together with the functional data constraints as the basis of the proposed method will target larger fault class than traditionally measured single SAF coverage contributes.

Assume, the ALU executes n different functions $y = f_i(d)$ by a set $F = \{f_i\}$ of instructions, where d represents data operand(s) for f_i, where the length of the data word (operand) is m, and ALU is controlled by p control signals. In Fig.1, the control part consists of the multiplexer MUX and p control lines (originating in the opcode field of the instruction register) as control inputs to MUX. The n AND blocks (consisting of m AND gates) in the control part of the execute unit have each p control and a single m-bit data input, whereas the OR block has n data word inputs from the outputs of AND blocks. Each AND block consists of m AND gates with p control inputs, and a single bit data input.

Let us classify two types of high-level functional fault models for the ALU: *control faults* (the faults related to the control part of the ALU), and *data faults* (the faults related to the data part of the ALU). For the control faults, we will introduce a novel high-level functional control fault model as follows.

Denote by y_i the data word considered as the result of execution of the function f_i with data operand(s) d_i as $y_i = f_i(d_i)$.

Definition 1. Introduce for the function (instruction) $f_i \in F$, the following *high-level control fault model* $M(f_i)$ as a set of data operands $M(f_i) = \{D_i\}$, which satisfy the following constraints at least once for each bit k of y_i:

$$\forall k \in (1,m): \{\exists d_i \in M(f_i)\ (y_{i/k} \neq 0)\}, \quad (1)$$

$$\forall f_j \in F, j \neq i: \forall k \in (1,m)\{\exists d_i \in M(f_i)\ (y_{i/k} < y_{j/k})\} \quad (2)$$

Depending on the technology, implemented in the microprocessor, the constant 0 in formula (1) can be changed into 1, and instead of the relation " < " in formula (2), there can be " > ".

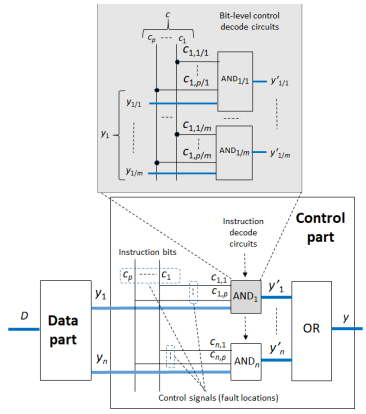


Fig.2. *Generic DNF based control structure of ALU*

The constraint (1) is needed for testing that the function f_i can be executed and the result "$y_i = 1$" can be produced in each bit of the data word to detect the faults SAF/0 on all inputs of AND-gates. The constraint (2) is needed for testing that the result "$y_i = 0$" can be produced in each bit of the data word to decect two types of faults: SAF/1 on all inputs of the AND-gates related to the function f_i, and all functional faults of overwriting the value "$y_i = 0$" in each bit due to the control faults of other functions $f_j, j \neq i$.

The proposed fault model can be regarded as a generalization of the *conditional SAF model* or *input pattern fault model* (similar to ones considered in [23-26]). In case of conditional SAF, we are testing SAF on the gate-level lines at some constrained signals on other lines, whereas in case of the proposed high-level fault model of Definition 1, we are testing the instructions of microprocessors at a set of constraints for data (operands).

There are two novelties of this approach. First, due to using the EDNF based (not optimized) control unit model, the generated test may be over dimensioned. Second, the functional constraints (1) and (2) tend to produce more test patterns than it is needed for only single SAF detection. However, both aspects work in favour of larger fault class coverage, including multiple faults also, as already mentioned.

The size (complexity) of the proposed high-level control fault model can be represented by the number of data constraints to be satisfied, that is $C = n(n\text{-}1)mp$

III. Mapping of High-Level Faults to Gate-Level Faults

Introduce the following notations of the input information for solving the problem.

Definition 2. Let D^*_i be the set of data operands which satisfy the constraints of the fault model $M(f_i\}$, T^*_i is the test for the instruction f_i, which uses the data operands $d \in D^*_i$, and $T^* = \{T^*_i\}$ is the full test, generated for all high-level control faults for the set of instructions $F = \{f_i\}$.

Theorem 1. The test $T^* = \{T^*_i\}$, which covers all non-redundant high-level faults of the fault model $M(f_i)$, covers also all gate-level non-redundant SAF in the control part of the microprocessor, which controls the set of functions F.

Proof. The proof can be done in 2 steps. Firstly, consider the equivalent circuit of ALU control part presented in Fig.2, and described as the following DNF

$$y = c_{1,1}c_{1,2}\dots c_{1,p}y_1 \vee c_{2,1}c_{2,2}\dots c_{2,p}y_2 \vee \dots \vee c_{n,1}c_{n,2}\dots c_{n,p}y_n \quad (3)$$

for each bit of the data word in the output of OR block. We can easily show that from generation of data which satisfy the constraints (1) and (2) for all functions $f_i \in F$, it follows that in the DNF all SAF faults will be detected. In this DNF the variables $c_{i,j}$ for selecting the data results y_i, $i = 1, \dots n$, represent the global control signals c_j, $j = 1,\dots p$, being either inverted or not, and covering in general case exhaustively all the 2^p combinations. Secondly, assume that the control circuit is optimized and is represented as a multi-level combinational circuit instead of the two-level DNF. In this case, we can represent the circuit as an equivalent disjunctive normal form in a similar way as DNF (3). As already mentioned, if there is a test set which detects all non-redundant faults in the EDNF, this test will detect also all faults in the original possibly optimized multi-level circuit [27]. ■

Corollary 1. If a high-level test is generated, so that the the constraints (1) and (2) are fully satisfied, but if there are some SAF in the related EDNF, which remain not detected by the high-level test, the not detected SAF are redundant.

Corollary 2. If there are some cases in the constraints (2), which cannot be satisfied by selecting data operands, these cases refer to the high-level redundancies in the model $M(f_i)$.

Corollary 3. If the high-level redundancies can be removed from $M(f_i)$, and the high-level test is generated, the not detected SAF are redundant.

Example 1. Consider a simplified ALU unit which implemets the set of three functions f_1, f_2, f_3, activated by a set of control signals $\bar{c_2}c_1, c_2\bar{c_1}, c_2c_1$ respectively. The ALU can be represented by the DNF:

$$y = \bar{c_2}c_1y_1 \vee c_2\bar{c_1}y_2 \vee c_2c_1y_3. \quad (4)$$

The test $T^* = \{T^*_1, T^*_2, T^*_3\}$ generated for the control part of ALU that satisfies the constraints (2) is depicted in Table 1.

Table 1. *Example of a high-level control test*

T^*_i	Test					Fault table						Constraints satisfied
	c_2	c_1	y_1	y_2	y_3	$\bar{c_2}c_1y_1$			$c_2\bar{c_1}y_2$			$c_2c_1y_3$			
T^*_1	0	1	0	1	1	1	1	0	0	0	1	**0**	1	1	$y_1 < y_2$, $y_1 < y_3$
T^*_2	1	0	1	0	1	0	0	1	1	1	0	1	**0**	1	$y_2 < y_1$, $y_2 < y_3$
T^*_3	1	1	1	1	0	**0**	1	1	1	**0**	1	1	1	0	$y_3 < y_1$, $y_3 < y_2$

The table contains the test patterns in column 2, the fault table in columns 3-5, and the constraints satisfied by generating data for the control test patterns in column 6. The detected gate-level faults in the fault table are highlighted by red colour: 0 means the value of a signal which activates the fault SAF/1. For example, in case of the fault $c_2 \equiv 1$ in column 5, the value of the output signal $y = y_1 = 0$ will change from 0 to $y = y_1 \vee y_3 = 1$. For detecting the faults SAF/0, 3 more test patterns are needed (not shown in the table).

We see in the fault table that the faults $c_1 \equiv 1$ in column 3 and $c_2 \equiv 1$ in column 4 are not detected. Based on Corollary 1, these faults are redundant. By minimizing the function (4), we get a new formula

$$y = \bar{c_2}y_1 \vee c_2(\bar{c_1}y_2 \vee c_1y_3).$$

where the redundancies are removed, and all SAF/1 are detectable by the test T^*.

The case of high-level redundancies is discussed in the following Sections.

Note, Theorem 1 and Corollaries 1-3 were formulated, considering the single SAF model. In fact, the power of the proposed high-level control fault model stretches far beyond the fault class of single SAF, as it will be shown in the following corollaries.

Corollary 4. The test $T^* = \{T^*_i\}$, covers all gate-level multiple SAF and bridging faults between control lines in the control part of the microprocessor, which controls the set of functions $F = \{f_i\}$.

Proof. From (2) it follows that for each function $f_i \in F$, $\forall k$: $(y_{i/k} < y_{j/k})$ for all $j \neq i$ must hold. This means that not only SAF/1 in a single control signal of a single function $f_j \in F$, $j \neq i$, can be detected (by overwriting $y_{i/k} = 0$ with $y_{j/k} = 1$), where the control words for f_i and f_j differ in a single bit, rather such overwriting of signals $y_{i/k} = 0$ with 1 can happen, and hence, can be detected, due to multiple changes $0 \rightarrow 1$ for $f_j \in F$, $j \neq i$, leading to detecting multiple faults. On the other hand, from the constraints (1-2), and from the exhaustiveness of testing all the control functions function $f_j \in F$, $j \neq i$, it follows that non-redundant bridging faults between the control lines can be also detected by T^*. ■

In case, when the target would be to detect only single SAF, then the fault model defined by the constraints (1) and (2) is over-dimensioned. For the case of full single SAF coverage, it would be sufficient to loosen the constraint (2) to

$$\forall f_j \in F, (HD(f_i,f_i)=1), j \neq i : \forall k \in (1,m)\{\exists d_i \in M(f_i)\ (y_{i/k} < y_{j/k})\},$$

where $HD(f_j,f_i)=1$ is the constraint that the Hamming distance between the control codes for f_j and f_i must be 1. This simplication is similar to the approach used in [5]

The size of the reduced high-level control fault model applied only to the code-neighboring functions f_j, f_i with $HD(f_j,f_i)=1$, is equal to $C_{\text{red}} = nmp < C = n(n\text{-}1)p$.

IV. High-Level Fault Coverage

To measure the fault coverage for the fault model $M(f_i)$, $f_i \in F$, proposed in Definition 1, by the given test T^*_i and the set of operands D^*_i , we introduce the high-level fault table as a matrix $E = || e_{i,j} ||$ with n columns and n rows, where n – is the number of functions in F. Each entry $e_{i,j}$ in E is a m-bit vector $e_{i,j} = (e_{i,j/1}, e_{i,j/2}, \ldots, e_{i,j/m},)$, where m is the number of bits in the data-words $y_i = f_i(d_i)$, $d_i \in D^*_i$. We denote by $e_{i,j/k} = 1$, if the constraint $y_{i/k} < y_{j/k}$ for the bit k in the set of constraints (2) is satisfied by the set of data operands in $D^*_i = \{d_i\}$, and $e_{i,j/k} = 0$ if not.

Table 2. *Example of a High-Level Fault Table*

	f_1 - MOV	f_2 - ADD	f_3 - SUB	f_4 - CMP	f_5 - AND
f_1 - MOV		111111	111111	111111	**000000**
f_2 - ADD	11111		11111**0**	111111	111111
f_3 - SUB	11111	11111**0**		111111	111111
f_4 - CMP	11111	111111	111111		**000000**
f_5 - AND	11111	111111	111111	111111	

An example of the matrix $E = || e_{i,j} ||$ for a test T^* for a set of functions $F = \{ f_i \}$ executed by the set of instructions $I = \{$MOV, ADD, SUB, CMP, AND$\}$, is presented in Table 2. Each i-th row in the table represents the high-level control fault coverage of testing the function $f_i \in F$, (and the respective instruction $I_i \in I$.

The fault table $E = || e_{i,j} ||$ is the result of high-level fault simulation for the given set of operands D^*_i , to be used by the high-level test T^*_i . In this paper we have implemented the following high-level control fault simulation algorithm.

Algorithm 1.

(1) **for** all row instructions f_i, $i = 1,\ldots,n$
(2) **for** all data operands $d_{i,j.1}, d_{i,j.2}, j = 1,\ldots,n_i$
(3) **for** all column instructions f_h, $h = 1,\ldots,n$
(4) calculate the value y_h
(5) check the relation $y_i < y_h$, $h \neq i$
(5) update the vector $e_{i,h} \in E$
(6) **end for** column instructions
(7) **end for** data operands
(8) **end for** row instructions

Based on Algorithm 1, we implemented a simulation based high-level test generation method on the basis of random search for test data to satisfy the constraints (2).

In Table 2, 0s refer either to not detected high-level control faults or to the possible high-level redundancies of the faults related to the constraints $y_{i/k} < y_{j/k}$, where i and j correspond to the rows and columns, respectively, and k refers to the bit number. All 0s in e_{ij} refer to high probability of the redundancy of the high-level fault model.

In most cases of ALU operations (like for e_{15} and e_{45} in Table 2), it is very easy to identify this type of redundancy. For example, if $y_i = f_i(a, b)$ refers to the AND operation and $y_j = f_j(a, b)$ refers to OR, it is straightforward that the constraint $y_i < y_j$, i.e. $(a \vee b) < (a \wedge b)$ cannot be satisfied by any values for a and b.

In cases when there is an entry $e_{i,j/k} = 1$ in a single bit k of the vector e_{ij} (like for e_{23} and e_{32} in Table 2), or in only few bits of the vector e_{ij}, we can suggest for the redundancy proof a method called "*partial truth table method*". The idea of the method stands in showing the equivalence of partial truth tables (or to prove the impossibility of solving the related constraints) for the functions involved in the constraint relation, so that as few as possible responsible bits should be selected for the need of the proof.

Table 3. *Examples of high-level fault redundancy proofs*

#	$y_{i/k} < y_{j/k}$	e_{ij}	$y_{i/k} < y_{j/k}$	00	01	10	11
1	SUB < ADD	1…11**0**	SUB	0	1	1	0
			ADD	0	1	1	0
2	OR < ADD	1…11**0**	OR	0	1	1	1
			ADD	0	1	1	0
3	OR < AND	**0**…**000**	OR	0	1	1	1
			AND	0	0	0	1
4	OR < XOR	**0**…**000**	OR	0	1	1	1
			XOR	0	1	1	0

In Table 3, examples for 1-bit partial truth tables for the functions SUB, ADD, OR, AND, and XOR, for selected bits k (shown with red color) are shown. The pairs 00, 01, 10, 11 in the title row represent the values of the data variables $d_{i/k}$ (as arguments for $y_{i/k}$) in bit k. The 1-bit values in the columns show the results of the related operations for the k-th bit. For the constraints SUB<ADD, and OR<ADD, the equivalence of the behavior in the least significant bit is demonstrated, which contradicts to the constraint (2). For the cases OR<AND, and OR<XOR, the missing of a solution for (2) is also shown for all possible input data combinations, and for all bits k. In some specific corner cases, the proof of redundancy may be more difficult.

The proof of high-level fault redundancy was not the target of the paper, and it needs special investigations. The quality of tests derived by the proposed method, SAF coverage was measured. The knowledge about redundancy of high-level faults is important when using of Corollary 3 for identification of redundant SAF by only applying fault simulation.

V. High-Level Test Program Composition

The full test T for testing the set of functions $F = \{f_i\}$ can be represented as a set of subtests $T_i(f_i)$:

$$T = \{T_i(f_i)\} = \{(I_i, D_i) \mid i : f_i \in F\}$$

where I_i denotes the instruction which executes the function $f_i \in F$, and D_i denotes the set of data patterns (operands), each of them has to be used by the instruction I_i. The data patterns $d_{i,j} \in D_i$ may represent either single operands or concatenation of two operands $(d_{i,j,1}.d_{i,j,2})$ stored in the memory. For each group of similar instructions, there is a template – a subroutine, repeated in a loop for all instructions I_i , where $i : f_i \in F$, and each instruction I_i is executed in a nested loop for all data operands in D_i, which are loaded by the initialization part of the template.

The architecture of test program is shown in Fig.3. The test tempates are created on the basis of Algorithm 2.

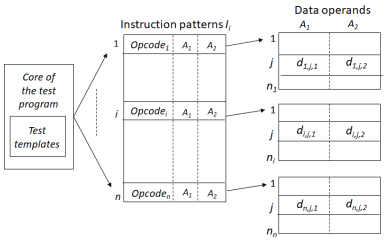


Fig.3. *Architecture of the test program*

Algorithm 2.

(1) **for** all instructions $I_i \in I$, $i: f_i \in F$
(2) **for** all data operands $d_i \in D_i$
(3) read d_i
(5) execute the instruction I_i
(6) store the test result $y_i = f_i(d_i)$
(7) **end for** data
(8) **end for** instructions

Each subtest $T_i(f_i) \in T$ for testing $f_i \in F$ is partioned into two parts: test for the control part, and test for the data path. These two parts differ in how the data sets D_i are generated.

For testing the control part, we use the data operands $D_i = D^*_i$, which are generated to satisfy the constraints of the fault model $M(f_i)$ according to Definition 1. For testing the data-path, for each instruction, dedicated data operands are to be generated. Denote these sets of operands as $D_i = D^{**}_i$.

Generation of the data operands to build the sets D^{**}_i was not the objective of this paper. In the experimental research, to achieve the complete test results, we exploited for creating the data sets D^{**}_i the parallel pseudoexhaustive test (PET) data operands, generated for selected data bits separately, and replicated then for other bits, using the methods presented in [28] for ALU, and in [29] for multiplication.

In this paper, we propose a new alternative approach for data-path testing, which directly results from the data operands generated for testing the control part – to execute each instruction using all data operands generated according to Definition 1 for all functions of the group D^*, so that

$$D_i = D^* = \cup_i D^*_i \mid i: f_i \in F.$$

In this data set, the data operands for testing the control and data paths are joined. This approach happened to be unexpectedly very efficient regarding the achieved SAF coverage, and at the same time, without adding cost for storing the test data in the memory.

Comparison of different approaches is presented in the Section for experiments.

VI. Experimental Results

We carried out experiments, consisting in high-level test data generation for the control and data parts of the execute stage of MiniMIPS processor [30], consisting of ALU and two multiplication modules MULT0 and MULT1.

The test program generation included automatic synthesis of test templates from manual parameter file, automated high-level test data (operands) generation to satisfy the constraints (1-2) and based on the fault simulation according to Procedure 1, and manual removal of the high-level fault redundancies to prove the 100% high-level test coverage.

To compare the quality of our high-level generated test program with commercial gate-level ATPG, we synthesized with Synopsys synthesis tool a gate-level implementation of the execute stage of MiniMIPS processor, and calculated with commercial fault simulation tool the gate-level SAF coverages for our high-level generated test program using two options of data sets described in Section V. The experimental research targeted 25 instructions $I_i \in I$ out of MiniMIPS 51 instructions, as the basis of the set of functions $F = \{f_i\}$ investigated in the paper.

Experimental results are shown in Table 4.

Table 4*. Experimental data*

Quality measures	Parts of the execute module	# Faults	Comparison of methods		
			Proposed ATPG		Gate level ATPG
			Only control data	Control + PET data	
Fault coverage %	Execute Stage	203576	98.70	99.02	97.73
	ALU	2516	99.92	99.92	99.96
	MULT0	95188	99.09	99.52	97.40
	MULT1	91810	99.05	99.16	97.71
# Stored test patterns			166	166	957
# Executed test patterns			4150	4818	957
Test generation time			47s	Manually added PET data	8h 27m

We investigated two versions of test data generation. In the first version "only control data" we used the full data set D^* generated automatically using the constraints (1-2). In the second version "control + PET data", we added to the data set D^* additional manually generated pseudo-exhaustive test patterns, using the results in [29]. Both high-level tests were simulated by commercial tool to grade the gate-level SAF coverage. In both cases, the proposed method of high-level test generation, where the knowledge of implementation details was not needed, produced high gate-level SAF coverage for both, control and data parts of the execute module in MiniMIPS.

To evaluate the efficiency of the high-level ATPG, we used commercial gate-level ATPG for comparison. The time cost for high-level automated test generation is about two orders of magnitude less than the time cost of the commercial ATPG. The gate-level SAF coverages, achieved by the proposed method for the whole module under test, and also for the separate submodules ALU, MULT0 and MULT1 are significantly better than that of achieved by the commercial ATPG tool.

The proposed method has also advantage compared to the commercial gate-level ATPG in the number of test patterns to be stored in the memory. The test is stored in the compact form, unrolling only during the test execution.

VII. Conclusions

In this paper, we proposed a new high-level test program generation method for execute modules of RISC microprocessors, which achieves gate-level SAF coverage significantly higher than a commercial gate-level ATPG. Furthermore, the speed of test generation exceeds the speed of the commercial ATPG more than two orders of magnitude.

The proposed method is based on a new high-level control fault model for microprocessors, which consists of a set of data constraints to be satisfied in test generation. The new test generation method uses as input information only the description of the instruction set, which is available in the

manuals, and no knowledge of implementation details is needed.

The test is able to achieve very high coverage of non-redundant single SAF, as demonstrated by experiments.

Additional contribution of the paper, which shows advantage over state-of-the-art methods, is the coverage of a larger class of faults than only single SAF, including bridging faults and multiple SAF in the control parts under test. Hence, the proposed method for testing the control circuit faults is more powerful than the traditional gate-level ATPGs, which target only the single SAF fault class. However, this claim is based only on theoretical considerations. The related experimental research should be the future work.

The method was extended also to testing the faults in the data path of the execute modules of microprocessors. A metric of high-level fault coverage and a method for high-level fault simulation were developed. Additionally, a manual method for proof of high-level fault redundancies was also developed.

The future work will target optimization of test data operands, and the extensions of the proposed method for other modules of microprocessors not targeted in this paper.

ACKNOWLEDGMENT

The work has been supported in part by project H2020 MSCA ITN RESCUE (EU Horizon 2020, Grant 722325), Estonian research grant IUT 19-1 and Excellence Centre EXCITE in Estonia.

REFERENCES

[1] L.Chen, S.Dey. Software-based self-testing methodology for processor cores. IEEE Trans. on CAD of IC and systems, vol.20, no.3, 2001, pp. 369 - 380.

[2] N.Kranitis, A.Paschalis, D.Gizopoulos, G.Xenoulis. Software based self-testing of embedded processors. IEEE Trans. on Comp., vol.54, no.4, 2005.

[3] P.Bernardi, R.Cantoro, S.De Luca, E.Sanchez, A.Sansonetti. Development Flow for On-Line Core Self-Test of Automotive Microcontrollers. IEEE Trans. on Comp., v.65, no.3, 2016, pp-744-754.

[4] M.Psarakis, D.Gizopoulos, E.Sanchez, M.S.Reorda Microprocessor software-based self-testing. IEEE Design Test of Computers, v.27, no.3, 2010.

[5] P.Bernardi, R.Cantoro, L.Ciganda, E.Sanchez, M.S.Reorda, S.D.Luca, R.Meregalli, A.Sansonetti. On the in-field functional testing of decode units in pipelined risc processors. IEEE Int Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems. 2014, pp. 299–304.

[6] A.Riefert, R.Cantoro, M.Sauer, M.S.Reorda, B.Becker. A flexible framework for the automatic generation of SBST programs. IEEE Trans on VLSI Systems, vol.24, no.10, 2016, pp. 3055–3066.

[7] M.Schölzel, T.Koal, S.Rieder, H.T.Vierhaus. Towards an automatic generation of diagnostic in-field sbst for processor components. LATW, 2013.

[8] S.Gurumurthy, S.Vasudevan, J.A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. IEEE International Test Conference, 2006.

[9] L.Lingappan, N. K. Jha. Satisfiability-based automatic test program generation and design for testability for microprocessors. IEEE Trans. on VLSI Systems, vol.15, no.5, pp. 518–530, 2007.

[10] C.H.Wen, L.-C.Wang, K.-T.Cheng. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., vol.55, no.11, 2006.

[11] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software based self-testing of embedded processors. IEEE Trans. on Comp., vol.54, no.4, 2005.

[12] C.H.Chen, C.K.Wei, T.H.Lu, H.W.Gao. Software-based self testing with multiple-level abstractions for soft processor cores. IEEE Trans on VLSI Systems, vol.15, no.5, pp. 505–517, 2007.

[13] J.Shen, J.A.Abraham. Native mode functional test generation for processors with applications to self test and design validation. Int. Test Conference, 1998..

[14] P.Parvathala, K.Maneparambil, W.Lindsay. Frits - a microprocessor functional bist method. International Test Conference, 2002, pp. 590–598.

[15] I.Bayraktaroglu, J.Hunt, D. Watkins. Cache resident functional microprocessor testing: Avoiding high speed io issues. IEEE Int. Test Conference, 2006.

[16] F.Corno, E.Sanchez, M.S.Reorda, G.Squillero. Automatic test program generation: a case study. IEEE Design Test of Computers, vol.21, no.2, 2004.

[17] D.Changdao, M.Graziano,E.Sanchez, M.Sonza Reorda, M. Zamboni, N. Zhifan. On the functional test of the BTB logic in pipelined and superscalar processors. LATW, 2013.

[18] E. Sanchez and M. S. Reorda. On the functional test of branch prediction units. IEEE Trans. on VLSI Systems, vol.23, no.9, 2015, pp. 1675–1688.

[19] S. D. Carlo, P. Prinetto, and A. Savino. Software-based self-test of setassociative cache memories. IEEE Trans. on Computers, vol.60, no.7, 2011, pp. 1030–1044.

[20] J.Perez Acle, R.Cantoro, E.Sanchez, M.Sonza Reorda. On the functional test of the cache coherency logic in multi-core systems. LATS, 2015.

[21] D.Gizopoulos, M.Psarakis, M.Hatzimihail, M.Maniatakos, A.Paschalis, S. Ravi, A.Raghunathan. Systematic software-based self-test for pipelined processors. IEEE Trans. on VLSI Systems, vol.16, no.11, 2008, pp.1441–1453.

[22] P.Bernardi, R.Cantoro, L.Ciganda, B.Du, E.Sanchez, M.S.Reorda, M.Grosso, O.Ballan On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors. 14th Int. Workshop on Microprocessor Test and Verification, Dec 2013, pp. 52–57.

[23] K.B.Keller. Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms. US Patent 5546408, Aug. 13, 1994.

[24] R.Ubar. Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations. Automation and Remote Control, Vol.40, No 11, part 2, Nov. 1980, Plenum Publishing Corporation, USA, pp. 1693-1703.

[25] R.D.Blanton, J.P.Hayes. On the Properties of the Input Pattern Fault Model. ACM Trans. Des. Automat. Electron. Syst., Vol. 8, No. 1, pp. 108-124, Jan. 2003.

[26] S.Holst, H.-J.Wunderlich. Adaptive Debug and Diagnosis Without Fault Dictionaries. Proc. of 13th ETS, Verbania, Italy, May 2008, pp.199-204.

[27] D.B.Armstrong. On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets. IEEE Trans. on Electronic Computers, v.EC-15, no.1,1966 pp.66-73.

[28] A.S.Oyeniran, A.Jasnetski, A.Tsertov, R.Ubar. High-Level Test Data Generation for Software Based Self-Test in Microprocessors. 6th Mediterranean Conference on Embedded Computing (MECO 2017), 2017.

[29] A.S.Oyeniran, S.P.Azad, R.Ubar. Parallel Pseudo-Exhaustive Testing of Array Multipliers with Data-Controlled Segmentation. Int. Symp. on Circuits and Systems (ISCAS), 2018.

[30] OpenCores, “MiniMIPS ISA”.

[31] F. Pellerey et al., "Rejuvenation of NBTI-Impacted Processors Using Evolutionary Generation of Assembler Programs," 2016 IEEE 25th Asian Test Symposium (ATS), Hiroshima, 2016, pp. 304-309

Appendix 7

VII
A. S. Oyeniran, S. P. Azad, and R. Ubar, "Parallel pseudo-exhaustive testing of array multipliers with data-controlled segmentation," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018

Parallel Pseudo-Exhaustive Testing of Array Multipliers with Data-Controlled Segmentation

Adeboye Stephen Oyeniran, Siavoosh Payandeh Azad, Raimund Ubar
School of Computer Systems
Tallinn University of Technology, Estonia

***Abstract*—This paper presents a new method for pseudo-exhaustive testing of standard array multipliers using a novel approach of data-controlled segmentation of the circuit. The method covers both combinational and sequential fault classes. Differently from previous papers, the proposed separate cell-testing approach targets multiple faults in different cells and avoids fault masking. The method is also applicable to other multiplier architectures like Booth and MiniMIPS with high stuck-at fault (SAF) coverage. The regular structure of the test allows efficient implementation of the method as both software based self-test (SBST) and hardware-based BIST.**

I. Introduction

Multiplication is one of the most traditional operations used in scientific calculations. Multipliers are embedded in data-path architectures of traditional microprocessors, DSPs or SOCs. Many current FPGAs incorporate embedded cores such as DSPs and microprocessors in addition to logic blocks [1]. In microprocessors, Software-Based Self-Test (SBST) approaches are widely used [2]. However, the test programs need to be equipped with efficient data operands. In cases where the multipliers have limited I/O access, effective Built-In Self-Test (BIST) solutions are needed [3].

In BIST, Linear Feedback Shift Registers (LFSR) based pseudo-random testing is the most popular strategy. However, in the case of large circuits like multipliers, the testing time tends to increase rapidly. Another prospective BIST solution is *Pseudo-Exhaustive Test*(PET) technique [4] which has shown to be more effective than pseudo-random test.

In this paper, we develop a new method for applying the pseudo-exhaustive test concept to test generation of multipliers. In Section II we give an overview of pseudo-exhaustive test followed by the description of the state-of-the-art of multiplier testing in Section III. In Section IV we present our new method of PET for multipliers. In Section V experimental results are presented. Section VI concludes the paper.

II. Pseudo-exhaustive test Concept

The pseudo-exhaustive testing [4] is applicable if each primary output of the circuit depends only on a small subset of primary inputs. If any circuit output depends on all of the circuit inputs, the circuit is called *Full Dependence Circuit (FD circuit)* [4]. PET for testing such circuits was described in [5] using *segments partitioning* technique. Segmentation can be performed by inserting additional hardware (multiplexers) [6] or algorithmically by generating proper control patterns [5]. Using exhaustive test for segments, all combinational faults can be detected with the exception of some bridging faults between different segments.

Extensions of PET for sequential circuits are presented in [7, 8]. In [9], a method is proposed to generate PET patterns at a more general implementation-free functional level.

In [10], a BIST methodology is presented for PET of unilateral, 1-dimensional *Iterative Logic Arrays* (ILA)s. In [11], a CAD tool which derives specifications for PET hardware for semi-regular combinational circuits is described.

Several *Design for Testability* (DFT) techniques for constructing testable bit-sliced ALUs easily were developed, and the notations of *C-testability*, *I-testability*, and *CI-testability* of the arrays were introduced in [12].

III. Overview of Multiplier Testing Methods

Multiplication in data-path architectures of microprocessors is usually performed by optimized array multipliers of various architectures. An overview and comparison of different types of multipliers is provided in [13].

For *standard array multipliers* to alleviate the test problem, several approaches of DFT (by adding extra hardware) were proposed [14–17]. In [14] an $n \times n$ *carry-save array* multiplier needs 7 additional inputs. Only *single cell combinational* (SCC) *fault model* is assumed. In [15], a design for an $n \times n$ *carry-propagate array multiplier* is given which needs $(n-1)$ extra EX-OR gates, and 5 extra inputs. In [16], a parallel $n \times n$ multiplier design is presented which is linearly testable with $3n + 60$ patterns, uses n extra OR gates and n^2 additional transmission gates, but only 1 extra controllable input. In [17], other DFT solutions for $n \times n$ *carry-save array multiplier* with 1 extra input were presented.

All these approaches need to change the design, which as a rule introduces performance issues. For SBST in general purpose microprocessors, the added extra control inputs make these approaches not applicable. Moreover, these methods are not developed for detecting delay and stuck-open(SOP) faults.

DFT approaches for modified Booth multipliers and Booth Wallace tree architectures have been reported in [18–21]. The 4×4 test algorithm in [19] and 5×3 algorithm in [20] both use an 8-bit counter to generate 256 test patterns and achieve 99% single SAF and SCC coverage. An algebraic method is presented in [26] for detecting only single SAF. The described above methods are not targeting multiple faults, and are not usable for detecting sequential faults.

The BIST design for testing sequential types of faults in multipliers was investigated in [22–25]. We adopt from these methods the idea of testing the cells with single input change (SIC) pairs, but propose a novel approach of direct

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

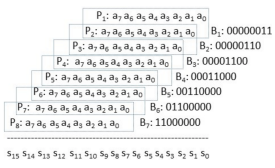


Fig. 1: Process of multiplying

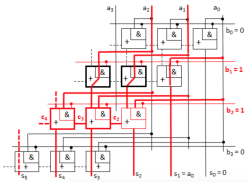


Fig. 2: Segmentation of multiplier

TABLE I: Pseudo-exhaustive test data generation

N	7			6			5			4			3			2			1			0	
	a7	a6	a5	a5	a4	a3	c3	a6	a5	c4	a5	a4	c3	a4	a3	c2	a3	a2	c1	a2	a1	c0	a1	a0
1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1	1
2	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	1	0	1	1
3	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1
7	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	1
10	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0
11	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0	1
12	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1
14	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1
15	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1

and independent access to cells to cover CMOS stuck-open and delay faults by considerably shorter test sequences.

An instruction level SBST approach for testing multipliers is described in [27]. This approach is not applicable for BIST due to the need of storing long test sequence.

In this paper, we propose a novel test generation approach which is suitable for both BIST and SBST related applications. The novelty is in a regular structure of the test data which consists of two sets of patterns: multiplier operands and multiplicands, both easily generated on-line algorithmically using BIST. The proposed novel PET method allows detection of multiple combinational cell faults (avoiding fault masking) and also detection of sequential faults.

IV. Pseudo-Exhaustive Multiplier Test with Data-Controlled Circuit Partitioning

Consider a ripple-carry adder as a 1-dimensional ILA. It belongs to the class of FD circuits. However, it can be easily segmented into cells with 3 inputs to be exhaustively tested.

The process of multiplying of two 8-bit patterns is represented as a *"paper and pencil"* method in Fig.1. Here A-pattern $(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ represents the multiplicand multiplied by the B-pattern (11111111). By multiplying the multiplicand with each bit of the multiplier B, 7 partial products are generated and added. Such a multiplier, represented as a matrix of $(n-1)^2$ full adders and n half adders, can be regarded as a 2-dimensional ILA which is very difficult to test by organizing PET in parallel for all 1-bit cells.

In the following, we propose a method to transform this 2-dimensional ILA of n-bit array multiplier into a set of $(n-1)$ 1-dimensional ILAs of n cells ($n1$ one-bit full- and 1 one-bit half-adders), which can be tested pseudo-exhaustively nearly as easily as ripple-carry adders. For such a transformation we introduce a concept of data-controlled segmentation of the circuit, where the segments of the multiplier will be selected by multiplier operands (B-patterns), so that each B-pattern selects a related single 1-dimensional ILA of 1-bit adders. For n-bit multiplier we need in total $(n-1)$ B-patterns with a pair of 1-s and all other bits 0, as shown in Fig.1.

In Fig.2, it is illustrated how by choosing the B-operands, the data controlled segmentation of the circuit is performed to select the desired 1-dimensional array of 1-bit adders for testing. In this example, the B pattern 0110 selects the 3^{rd} row of adders ($b_2 = 1$) to be tested. The value of $b_1 = 1$ selects the first operand, and the value of $b_2 = 1$ selects the second operand to be added in this row of 1-bit adders. The connections, involved in sending stimuli signals from primary inputs to the inputs of the adders under test, and propagating response signals to primary outputs are highlighted by bold red lines.

The PET generation of A-patterns is illustrated in Table I as a process of assigning consistent values to the triples of signals (c_i, a_{i+1}, a_i), where $i = 0, 1, ..., n-1$, and n is the data word length. Each row in Table I represents a test pattern (two operands) for testing the 1-bit adder row selected by a B-pattern. The values of (a_i, a_{i+1}) are chosen so that in each column all the 8 combinations of bits were present. To solve the inconsistencies caused by using similar variables a_i in the neighbor columns, and by the dependencies of carries c_{i+1}, on the values of variables (c_i, a_{i+1}, a_i) in the previous columns, 11 A-patterns were needed. To minimize the BIST hardware, the goal was set to create A-patterns with as less unique columns as possible in Table I. As a result of this, a PET was created where the unique columns (2,3,4,5) can be replicated for all higher significant 4-bit groups of the operands, independent of their lengths. This replication possibility makes the created PET very regular, and allows to generate a very simple BIST.

However, it was not possible to create the patterns 100 and 110 for the triple (c_1, a_2, a_1) in column 1, due to unsolvable inconsistencies in columns 0 and 1 (shown by green bits in the rows 2 and 10). To guarantee the highest quality of PET, and enabling still all the needed 8-patterns for testing the 1-bit adder related to column 1, additional input with $n-1$ OR gates can be inserted into multiplier to be controlled by BIST.

The first 11 A-patterns in Table I multiplied by B-patterns with cyclically shifted 11s (see Fig.1) forms a PET which guarantees coverage of combinational cell faults like SAF, conditional SAF and bridging faults in cells.

Differently from the previous methods, the described approach of testing each cell separately allows avoiding mutual masking of multiple faults in different cells. The exception may be two faults in neighbor cells in the same adder row. Assume, there are two faults f_i and f_{i+1} in the cells $C_{i,k}$ and $C_{i+1,k}$, respectively, where k is the number of the row of 1-bit adders in the multiplier array selected by the B-operand. The fault in f_i will be detected on the output s_i of the cell $C_{i,k}$, and propagate up to the primary output out_i of the circuit. The possible faults in the cells $C_{i,m}$, $m > k$, on the propagation path cannot mask the fault f_i, since all the 1-bit adders on this path between the cell $C_{i,k}$ and the primary output out_i will not be activated by the B-operand, and play only the role of passing the faulty signals at the constant working mode. Any possible multiple fault on this path will stuck the path output into constant, and hence, will be detected. If the the fault f_i will propagate via carry c_i to the neighbor cell $C_{i+1,k}$, the

two faults f_i and f_{i+1} may mutually mask each other in the cell $C_{i+1,k}$. In case if f_i will be not detected on the output s_i, both faults may remain undetected. Two faults f_i and f_{i+2} in not neighboring cells $C_{i,k}$ and $C_{i+2,k}$, respectively, cannot mask each other, because the fault f_i will cause an error either at s_i or s_{i+1} or both, due to the property of PET, and hence, detected at the primary outputs. A multiple not redundant fault in the same cell cannot be mutually masked due to the property of PET.

Detection of sequential faults like Transition Delay Faults(TDF) and SOP requires application of test vector pairs to the circuit under test. Thanks to the proposed data-controlled method which enables separate and independent testing of all 1-bit adders of the multiplier. It became possible too apply the same idea proposed in [22]... here which is applying single input change(SIC) pairs to every adder cell.

The proposed approach of separate testing of 1-bit adders allows extension the PET to cover the sequential faults. The property of covering sequential faults is achieved by proper reordering the rows in Table I, so that for all 1-bit adders at all inputs, both transitions $0 \rightarrow 1$ and $1 \rightarrow 0$ were present with keeping Hamming distance 1 between the paired patterns. To achieve this property for the PET, 4 more patterns were needed to add into Table I. The single input transitions in all columns of Table I are highlighted by red numbers. The green cell in 13^{th} row refers to the added input to be controlled by BIST.

Fig. 3 shows the implementation of the proposed method as BIST. The circuit consists of a Finite State Machine(FSM) with 11 states (according to 11 A-patterns) which controls the test process. The extended version of FSM for testing sequential faults consists of 15 states. The A-patterns are loaded directly from the FSM, replicating the 4-bit groups (a_2,a_3, a_4,..., a_5)for all of the next higher 4-bit groups, while a barrel shifter which is shifting predefined values loads the B-patterns. An "operation mode" signal can switch the multiplier from normal mode to test mode by utilizing two multiplexers.

TABLE II: Comparison of different PET versions

Standard array multiplier								
n	Regular solution:For BIST and SBST						Only for SBST (with additional stored patterns)	
	PET		PET*		PET + DFT		PET + DET	
	FC%	# pat: $11 \times (n-1)$	FC%	# pat: $11 \times (n-1) + 3 \times (n-2)$	FC%	# pat	FC%	# pat
8	98.03	77	100	95	100	77	100	77+7
16	98.89	165	99.92	207	100	165	99.98	165+13
32	99.49	341	99.95	431	100	341	99.98	341+17
Multiplier of MiniMIPS microprocessor and Booth Multiplier								
n	Regular solution: For BIST and SBST				Only for SBST(with additional stored patterns)			
	miniMIPS		Booth		PET + DET(miniMIPS)		PET + DET(Booth)	
	FC%(PET)	FC%(PET*)	FC%(PET)	FC%(PET*)	FC%	# pat	FC%	# pat
8	98.03	99.82	98.02	99.13	100	77+15	100	77+13
16	95.83	99.56	98.05	98.15	100	165+44	100	165+8
32	93.71	98.06	96.0	97.35	99.98	341+77	99.98	341+75

V. Experimental Results

In Table II we compare different modifications of the proposed PET for the standard array multiplier and for the multiplier of the MiniMIPS microprocessor, with different word lengths n, for SAF coverage (FC%) and test lengths (# pat), using only 11 PET patterns for the A-operand. PET* is an extended version of PET where additional $3 \times (n-2)$ patterns were added: shifted "111" in B-patterns with 2 A-patterns, and shifted "101" in B-patterns with 1 A-pattern. PET* preserves the regularity property of test data, and hence, is well suitable for BIST. Added 1 extra input and $(n-1)$ OR gates into the multiplier (PET + DFT) allows to achieve 100% SAF coverage. A hybrid test using PET with additional ATPG-generated deterministic test patterns (PET + DET) guarantees as well 100% SAF coverage, but without any change in the multiplier, and therefore is well usable for SBST. In Table III, we compare the 4×4 method from [20], a modified PET method, where for generation of the B-operands we exploit the idea from [20] to replicate in 4-bit groups of B all 16 combinations (referred as "$R2^4(B)$)", and finally, the same with replication of 3-bit groups "$R2^3(B)$)". These tests provide a good single SAF coverage, however, the PET properties of A-operands may suffer, because instead of single adders, two or more adders are now involved in operations. To compare the proposed PET with other state-of-the-art, the methods in [13-21] are targeting only combinational faults, and in most of them, DFT modifications of the circuit are needed, which makes them difficult to use for SBST in standard microprocessors. Differently from state-of-the-art, the proposed approach is targeting the multiple faults in different cells by avoiding mutual fault masking. In Table IV a comparison is provided between the BIST [23] and the proposed method, where both are targeting sequential fault (SOP and delay) coverage based on applying test pattern pairs. The lengths of the test sequences and the coverages of single input changes (SIC) in test pairs (as the quality of sequential fault testing) are provided.

TABLE III: Comparison of PET for 2 different multipliers

n	4 x 4 [20]			Variations of the proposed PET method					
				PET 11 (A), $R2^4$ (B)			PET 11 (A), $R2^3$ (B)		
	FC%		# pat	FC%		# pat	FC%		# pat
	A-M	M-M		A-M	M-M		A-M	M-M	
8	99.94	99.83	16×16 = 256	99.89	99.82	11×16 = 176	99.89	99.82	11×8 = 88
16	99.98	99.97		99.97	99.95		99.61	99.10	
32	99.99	99.97		99.99	99.88		99.25	96.98	

TABLE IV: Comparison for sequential fault coverage

No	Method		Test Length	SIC coverage	
1	BIST [23] Robust implementation		$512(3n_x + n_y 2))$	98%	
2	BIST [23] Low-Cost implementation		4096 for any length	Not available	
3	Proposed PET extended for sequential fault test	Without any DFT	$15 \times (n_y - 1)$	$n_x = 8$	97.8%
				$n_x = 16$	98.9%
		With DFT(1 extra input, (n-1) OR gates)		$n_x = 32$	99.4%
4			$15 \times (n_y - 1)$	100%	

The resulting BIST hardware was synthesized using AMS 0.18 μm CMOS technology library [28] for different bit-widths. Fig.4 depicts the normalized area overhead of the proposed BIST for different data widths. Fig.5 compares the area overhead of the proposed BIST with the traditional LFSR-based BIST. The experiments show that for larger bit-widths, the area overhead of the test circuitry becomes negligible (Fig.4), however, the proposed BIST is growing much slower in area compared to the LFSR-based BIST (Fig.5). Fig.6 depicts the consumption of total power (internal, switching and leakage) of the test program for LFSR-based and proposed BIST. The proposed BIST consumes much less power than the LFSR based solution, which makes it a desirable solution.

Algorithms 1 and 2 describe the unfolded SBST (where all test patterns are stored in the memory and are loaded them one by one) and proposed SBST (which stores the patterns for operand A but calculates operand Bs values on the fly by shifting it to the right). Fig. 7 compares the memory

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

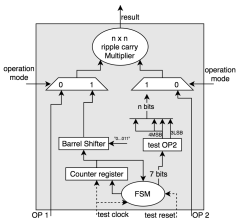


Fig. 3: BIST architecture for the multiplier

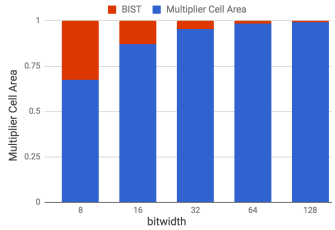


Fig. 4: Overhead of proposed BIST for different data widths

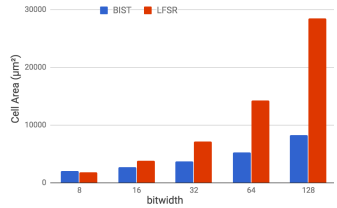


Fig. 5: Area overhead for proposed BIST and LFSR-based BIST

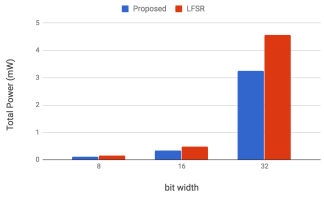


Fig. 6: Power consumption of proposed BIST and LFSR-based BIST

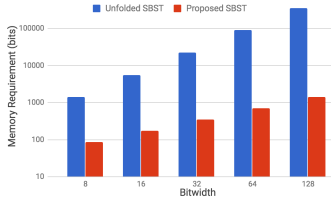


Fig. 7: Comparison of memory requirements for two SBST methods

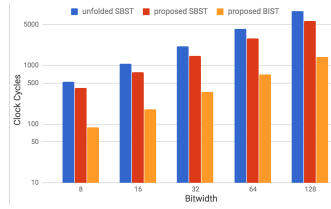


Fig. 8: Performance comparison of SBST and BIST methods

initialize the pointer to beginning of test data;
while $(i < 11 \times n \times 2)$ **do**
  load reg_1 from memory;
  i++;
  load reg_2 from memory;
  i++;
  multiply reg_1, reg_2;
end

Algorithm 1: unfolded SBST

initialize the pointer to beginning of test data for operand A;
while $i < 11$ **do**
  load reg_1 from memory;
  j = 0;
  while $j < n$ **do**
    reg_2 <= calculate operator B for cycle n;
    multiply reg_1, reg_2;
    j ++;
  end
  i ++;
end

Algorithm 2: proposed SBST

TABLE V: Experimental Data for BIST and SBST

comparison of different approaches for 16 bit multiplier				
Parameters	BIST		SBST	
	proposed	LFSR	SBST	proposed
memory	0	0	5632	176
time (clk)	177	203	1060	764
area overhead	14.87%	21.08%	0	0
Power (mW)	0.3426	0.4749	Depends on the processor	

requirement for the above mentioned approaches, and Fig. 8 compares the clock cycles for performing the test program using above mentioned SBST algorithms against the proposed BIST for a 4-stage pipelined processor for different bit-width for the multiplier which show the effectiveness of the proposed approach in contrast to unfolded SBIST. Table V provides an overview of parameters of the above mentioned methods for testing a 16-bit multiplier.

VI. Conclusions

We presented a novel approach for pseudo-exhaustive testing of standard array multipliers, which is able to cover a broader class of faults, compared to state-of-the art array multipliers, including both combinational and sequential faults. The proposed method provides a high stuck-at fault coverage also for other types of multipliers. The main idea of the approach is in the novel data-controlled partitioning method which enables separate access to all 1-bit adder cells, and as the result high quality of testing with well structured test data and less test lengths compared to state-of-the-art methods. The proposed method targets both combinational faults (multiple SAF in cells, conditional SAF, shorts), and sequential faults (CMOS stuck-opens and delay faults). Differently from the previous methods, the proposed PET approach targets also multiple faults in different cells and avoids mutual fault masking due to separate application of PET patterns to cells. Several modifications of PET approaches are proposed to improve its quality. The proposed test set has a regular structure, which makes it applicable for both, HW-based logic BIST and SW-based self-test to be used in standard processors and DSPs. Experimental research results showed that the proposed novel PET based BIST solution outperforms the parameters like area overhead and power consumption in traditional LFSR based BIST. The regular structure of PET data allows more efficient solutions also for SBST.

Experimental results demonstrate that for standard array multipliers the proposed data controlled segmentation method achieves 100% fault coverage for a broad class of faults. For other classes of multipliers (MiniMIPS, Booth multiplier), SAF coverage is less (see Table II), which can be seen as a limitation of the method. However, combining the proposed PET with deterministic patterns for the mentioned multipliers, it was possible to achieve 100% SAF coverage.

Acknowledgments The work has been supported by EU's H2020 RIA IMMORTAL, EU's Twinning Action TUTORIAL, Estonian institutional research grant IUT 19-1, Estonian IT Academy program and funded by Excellence Centre in IT in Estonia (EXCITE) project.

REFERENCES

[1] Pulukuri, M., Stroud, C. BIST of Digital Signal Processors in Virtex-4 FPGAs. IEEE Southeastern Symp. on System Theory, 2009.

[2] Chen, L., and Dey, S. SW-based self-testing methodology for processor cores. IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, 2001, pp.369-380

[3] D. Gizopoulos, A. Paschalis, Y. Zorian, An Effective BIST Scheme for Parallel Multipliers, IEEE Trans. on Comp., vol. 48, no. 9, 1999, pp.936-950.

[4] McCluskey, E.J. Verification Testing - A Pseudo-Exhaustive Test Technique. IEEE Trans. on Comp., Vol. 33, No. 6, 1984, pp.541-546.

[5] Udell,J.G., McCluskey, E.J. Pseudo-exhaustive test and segmentation: formal definitions and extended fault coverage results. FTCS, 1989, pp. 292-298.

[6] Udell, J.G. Reconfigurable Hardware for Pseudo-Exhaustive Test. International Test Conference, 1988, pp.522-530.

[7] Wunderlich, H-J., Hellebrand, S. Generating pattern sequences for the pseudo-exhaustive test of MOS-circuits. FTCS, 1988, pp.36-41.

[8] Wunderlich, H-J., Hellebrand, S. The Pseudoexhaustive Test of Sequential Circuits. IEEE Trans. on CAD of IC and Systems, vol.11, no.1, 1992, pp.26-33.

[9] Tang, R., Si, P.F., Huang, W.K., Lombardi, F. Testing IP Cores with Pseudo-Exhaustive Test Sets. 4th Int. Conf. on ASIC, 2001, pp.740-743.

[10] Su, C.C., Lemke, J.K., Chen, M., Kime. C.R. A BIST Methodology for Iterative Logic Arrays. IEEE Int. Symp. Circuits and Systems, 1992, Vol.1, pp.411-414.

[11] Su, C.C., Kime. C.R. Computer-Aided Design of Pseudo-exhaustive BIST for Semiregular Circuits. Proc. Int. Test Conf. 1990, pp. 680-689.

[12] Sridhar, T., Hayes, J.P. Design of Easily Testable Bit-Slice Systems. IEEE Trans. on Computers, vol. C-30, no. 11, 1981, pp. 842-854.

[13] Pulukuri, M.D., Starr, G.J., Stroud, C.E. On Built-In Self-Test for Multipliers. Proc. IEEE SoutheastCon 2010, pp. 25-28.

[14] Shen, J.P., Ferguson, F.J. The Design of Easily Testable VLSI Array Multipliers. IEEE Trans. on Comp., vol.33, No.6, 1984, pp.554-560.

[15] Chatterjee, A., Abraham, J.A. Test Generation for Arithmetic Units by Graph Labeling. Proc. Int. Symp. FTC, 1987, pp.284-289.

[16] Hong, S.J. An Easily Testable Parallel Multiplier. FTCS, 1988, pp. 214-219.

[17] Takach, A.R., Jha, N.K. Easiliy Testable Gate-Level and DCVS Multipliers. IEEE Trans. on CAD, vol.10, NO.7, 1991, PP.932-942.

[18] Stans, R. The Testability of a Modified Booth Multiplier. Proc. 1st European Test Conference, 1989, pp.286-293.

[19] Gizopoulos, D., Paschalis, A., Zorian, Y. An Effective BIST Scheme for Booth Multipliers. Int. Test Conference, 1995, pp.824-832.

[20] Gizopoulos, D., Paschalis, A., Zorian, Y. An Effective BIST Scheme for Parallel Multipliers. IEEE Trans. on Comp., vol.48, no.9, 1999.

[21] Booth, A.D. A Signed Binary Multiplication Technique. A. J. Mech. Appl. Math. 4, 1951, pp. 260-264.

[22] Psarakis, M., Gizopoulos, D., Paschalis, A., Zorian, Y. Sequential Fault Modeling and Test Pattern Generation for CMOS Iterative Logic Arrays. IEEE Trans. on Comp., vol.49, no.10, 2000.

[23] Psarakis, M., Gizopoulos, D., Paschalis. Built-In Sequential Fault Self-Testing of Array Multipliers. IEEE Trans. CAD of IC and Syst., vol.24, no.3, 2005.

[24] Smith, G.L. Model for Delay faults Based upon Paths. IEEE Int. Test Conference, 1985, pp.342-349.

[25] Liang, H.-C., Huang, P.-H. Testing TDF in Modified Booth Multipliers by Using C-Testable and SIC patterns. IEEE Reg. 10 TENCON Conf., 2007.

[26] Rahaman, H., Mathew, J., Pradhan, D.K., Jabir, A.M. Derivation of Reduced Test Vectors for Bit-Parallel Multipliers Over GF(2m). IEEE Trans. on Computers, vol. 57, no. 9, 2008, 1289-1294.

[27] Lin, M., Yan, G. Instruction level test for parallel multipliers. 15th IEEE International Conference on Electronics, Circuits and Systems ICECS, 2008.

[28] http://ams.com/eng/Products/Full-Service-Foundry/ Process-Technology/ CMOS/0.18-m-CMOS-process/

978-1-5386-4881-0/18/$31.00 ©2018 IEEE

Appendix 8

VIII
A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy, and J. Raik, "Mixed-level identification of fault redundancy in microprocessors," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, March 2019

Mixed-level identification of fault redundancy in microprocessors

Adeboye Stephen Oyeniran, Raimund Ubar, Maksim Jenihhin, Cemil Cem Gürsoy, Jaan Raik
Tallinn University of Technology, Estonia
adeboye.oyeniran@ttu.ee, raiub@pld.ttu.ee, maksim@pld.ttu.ee, cem@ati.ttu.ee, jaan@pld.ttu.ee

Abstract— **A new high-level implementation independent functional fault model for control faults in microprocessors is introduced. The fault model is based on the instruction set, and is specified as a set of data constraints to be satisfied by test data generation. We show that the high-level test, which satisfies these data constraints, will be sufficient to guarantee the detection of all non-redundant low level faults. The paper proposes a simple and fast simulation based method of generating test data, which satisfy the constraints prescribed by the proposed fault model, and a method of evaluating the high-level control fault coverage for the proposed fault model and for the given test. A method is presented for identification of the high-level redundant faults, and it is shown that a test, which provides 100% coverage of non-redundant high-level faults, will also guarantee 100% non-redundant SAF coverage, whereas all gate-level SAF not covered by the test are identified as redundant. Experimental results of test generation for the execution part of a microprocessor support the results presented in the paper.**

Keywords: ***processor core testing, high-level control fault model, high-level fault simulation, fault coverage, fault redundancy***

I. INTRODUCTION

Technology scaling in today's deep-submicron processes produce new failure mechanisms in electronic devices, which has forced researchers to develop more advanced fault models compared to the traditional *stuck-at fault* (SAF) model [1], and to investigate the possibilities of reasoning the faulty behavior of systems without using any particular fault models [2, 3].

Fault models for digital circuits have been developed for different types of failure mechanisms like signal line *bridges* [4], transistor *stuck-opens* [5] or failures due to increasing circuit *delays* [6]. Another trend has emerged to develop general fault modeling mechanisms and corresponding test tools that can effectively analyze arbitrary fault types. The oldest example is the *D-calculus* [7]. A generalization of this approach has been found in the *input pattern fault model* [8], and in the *pattern fault model* [9], which can represent any arbitrary change in the logic function of a circuit block, where a block is defined to be any combinational sub-circuit described at any level of the design hierarchy.

A similar pattern related fault modeling approach called *functional fault model* was proposed earlier in [10] for the module level fault diagnosis in combinational circuits. The *functional* (or *pattern*) fault model allows an arbitrary set of signal lines to be grouped into activation conditions for a single fault site, allowing a variety of *physical defect* types to be modeled. Based on the *functional fault model* a deterministic *defect-oriented* test pattern generator DOT was developed in [11] which allowed proof of the logic redundancy of not detected physical defects.

In [12], a similar model called *conditional faults* was proposed for test generation purposes, and in [13] for diagnosis purposes. A conditional fault allows additional signal line objectives to be combined with the detection requirements of a particular fault. For complete exercising blocks in combinational circuits on the gate level, a similar pattern oriented *gate-exhaustive fault model* was proposed in [14], which was extended to target bigger regions (collections of gates) by *region-exhaustive fault* model in [15].

The described functional, conditional and pattern fault models offer high flexibility in defect modeling beyond single SAF model. Further advancements of the low-level fault modeling have been achieved by introducing the fault *tuple fault* model [16], *realistic sequential cell fault* model [17], or *cell-internal defect* model [18], where the last two cases provide general capability to handle sequential misbehavior of circuits.

The conditional SAF model (and other listed models) [8-18] support *hierarchical test approach*, where the test pattern (or sequence), which activates a low-level fault (e.g. physical defect) at the lower level can be considered as the high-level *condition* (or *constraint*) for the functional fault defined at the higher level.

To increase the speed of test generation and fault coverage evaluation, high-level (functional or behavioral) fault models have been developed. Such a model can be considered as "good", if the tests generated using this model provide a high coverage of SAF or physical defects.

In the design hierarchy, higher-level descriptions have fewer implementation details, but more explicit functional information than lower level descriptions. High-level fault models depend on which level the tests are generated.

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

Usually, the methods of high-level test generation are divided into structural RTL based methods [19-20], or behavioral test generation methods [21-22]. A high-level fault model can be explicit or implicit [23-24]. An explicit model identifies each fault individually, and every fault in this model will be a target for test generation. Implicit models are based on the assumption that all gate-level faults may not be represented at the RT level, and this motivated to develop dedicated RTL fault models with dependence on implementation details.

High-level fault models are used widely in the field of Software-Based Self-Test [25-30]. These approaches can be divided into two major groups - structural and functional. Structural approaches, such as [25-26], are based on test generation using information from lower level of design (gate- or RTL-level description) of processor under test. Functional, in its turn, is using instruction set architecture (ISA) information of the processor under test [27-30].

The main and general problem of high-level faults is the difficulty of proving that the model covers all low-level detectable (non-redundant) faults. In existence of such a high-level proof, it would be possible to identify the redundancy of gate-level faults exclusively by only gate-level fault simulation, which has cheaper cost than low-level fault redundancy proof by conventional gate-level ATPG-s.

In this paper, we make such attempt for a restricted class of circuits with well-defined functionality. Particularly, we target ALU control circuits. We propose a high-level data constraint based functional control fault model, and we prove that the test producing 100% high-level fault coverage will also guarantee 100% low-level detectable SAF coverage, and that all not detected SAF, identified by low-level fault simulation, are redundant.

The rest of the paper is organized as follows. In section 2, we present a novel control fault model for microprocessors, and in Section 3 we investigate the problem of mapping these high-level faults to low-level. Section 4 discusses high-level fault coverage measurement. In Section 5, we investigate the problem of high-level fault redundancies, and in Section 6 low-level fault redundancies. Section 7 presents experimental data, and Section 8 concludes the paper.

II. High-level control fault model for processors

In this paper, we focus on testing of the ALU control, as a part of all control circuits in microprocessor cores.

Assume, the ALU executes n different functions $y = f_i(D_i)$ by a set $F = \{f_i\}$ of instructions, where D_i is the set of data operands for f_i, the length of the data word is m, and ALU is controlled by p control signals. Consider a general ALU model partitioned into the control and data parts as shown in Fig.1. The control part consists of the multiplexer MUX and p control lines as control inputs to MUX. The n AND blocks in MUX have each p control and a single m-bit data input, whereas the OR block in MUX has n data word inputs from the outputs of AND blocks. Each AND block consists of m AND gates with p control inputs, and a single bit data input.

Let us classify two types of high-level functional fault models for ALU: *control faults* (the faults related to the control part of ALU), and *data faults* (the faults related to the data part of ALU). In the following, we will consider the control fault testing.

Definition 2. Introduce for the function (instruction) $f_i \in F$, the following *high-level control fault model* $CFM(f_i) = \{Ex(f_i), C(y_i, F)\}$, where $C(y_i, F)$ is a set of the constraints to be satisfied for each bit k of y_i:

$$\forall k: (y_{i/k} \neq 0), \quad (1)$$

$$\forall f_j \in F, j \neq i: \{\forall k: (y_{i/k} < y_{j/k})\} \quad (2)$$

Depending on the technology, implemented in the microprocessor, the constant 0 in formula (1) can be changed into 1, and instead of the relation " < " in formula (2), there can be " > ".

Definition 1. Let us introduce *control fault universe* as a set of any multiple SAF and bridging faults on the control lines of the control part (shown as control fault locations in Fig.1).

Introduce the following notations: $Ex(f_i)$ – execution of the instruction f_i, y_i – the data word considered as the result of the introduction f_i at the data operands D_i.

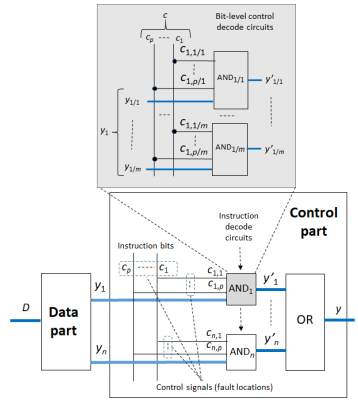


Fig.1. *Generic DNF based control structure of ALU*

The proposed fault model can be regarded as a generalization of the *conditional SAF model* (or similar ones considered in [8-18]). In case of conditional SAF, we are testing SAF on the gate-level lines at some constrained signals on other lines, whereas in case of the high-level fault model of Definition 2, we are testing the instructions of microprocessors at a set of constraints for data (operands).

Let us compare the complexities of the proposed high-level control fault model and the traditional SAF model of the control part architecture in Fig.1. The complexity of SAF model can be represented by the size of the model, i.e. by the number of control lines in the circuit multiplied by two for both SAF types: $C(\text{SAF}) = 2nmp$. On the other hand, the complexity of the proposed high-level control fault model (CFM) can be represented by the number of data constraints to be satisfied, that is $C(\text{CFM}) = n(n\text{-}1)mp$.

The time costs TC of test generation for both cases of the fault model can be estimated roughly by multiplying the size of the model with average test generation time t per fault as

$$TC(\text{SAF}) = 2nmp * t_{SAF} \quad (3)$$

$$TC(\text{CFM}) = n(n\text{-}1)mp * t_{FFM} \quad (4)$$

Despite that the size C(SAF) is linear with the circuit size, the gate-level test generation time with ATPGs is not scalable. On the other hand, despite the quadratic size of the high-level control fault model C(CFM), the test generation time for solving the data constraints (2) is linear, and very efficiently executable by random search (see experimental data).

Note, to simplify the model proposed in Definition 2 with the goal to reduce its size, the set of instructions F can be partitioned into subsets of F, similarly as proposed in [29], and for each subset, dedicated high-level fault model according to Definition 2 can be derived.

The practical reason for such partitioning of F may result from the instruction coding scheme. For example, if different fields of the instruction format have separate decoding circuit, then for each field, a separate set of instructions F for the proposed in Definition 2 fault model can be assigned.

III. High-level control fault mapping to gate-level faults

Let us consider in the following, how the gate-level fault redundancies in the control part of ALU can be identified by mixed level fault reasoning. To reduce the complexity of the problem, we propose, instead of exploiting slow conventional gate-level ATPGs for SAF redundancy proof, to use the combination of faster high-level test generation, and faster than ATPG low level fault simulation to achieve the same result – identification of the redundant low-level faults.

Introduce the following notations of the input information for solving the problem.

Definition 3. The set $CFM = \{CFM(f_i)\}$ for all $f_i \in F$ represents the full high-level control fault model for the given set of functions $F = \{f_i\}$ of the microprocessor.

Definition 4. Let D^*_i – be the set of data operands which satisfy the constraints of the fault model $CFM(f_i\}$, T^*_i – the test, which uses the data operands D^*_i, and $T^* = \{T^*_i\}$ – the full test, generated for the control fault model CFM.

Theorem 1. The test $T^* = \{T^*_i\}$, which covers all non-redundant high-level faults of the model $CFM = \{CFM(f_i)\}$, covers also all gate-level testable SAF in the control part of the microprocessor, which controls the set of functions F.

Proof. Consider the generic ALU control part presented in Fig.1 and described as the following DNF:

$$y = c_{1,1}c_{1,2}\dots c_{1,p}y_1 \vee c_{2,1}c_{2,2}\dots c_{2,p}y_2 \vee \dots \vee c_{n,1}c_{n,2}\dots c_{n,p}y_n \quad (5)$$

In this DNF the variables $c_{i,j}$ for selecting the data results y_i, $i = 1, \dots n$, represent the global control signals c_j, $j = 1,\dots p$, being either inverted or not, and covering in general case exhaustively all the 2^p combinations. In DNF, according to Definition 4, and due to satisfied constraints (2) of the fault model in Definition 2, at least once the value of y_i for each $i = 1, \dots n$, will be $y_i = 1$. On the other hand, due to the exhaustiveness of all 2^p combinations of control signals, for each term of DNF with $y_i = 1$, there will be a combination of control signals $c_{i,1}c_{i,2}\dots c_{i,p}$ consisting of a single 0, e.g. $c_{i,p} = 0$, with all others control signals $c_{i,r} = 1, r \neq p$. This is the case, where in the term $c_{i,1}c_{i,2}\dots c_{i,p}y_i$, the SAF $c_{i,p} \equiv 1$ is activated. For propagating the fault $c_{i,p} \equiv 1$ to the output y, all other terms in DNF must have at least one 0 assigned to the variables of the term. This is guaranteed, because due to the constraints (2), which demands that in the term where all $c_{j,k} = 1$, the value of y_j must be 0, and in other terms there must be at least one variable assigned by 0. Hence, all SAF faults of type $c_{i,p} \equiv 1$ in all variables $c_{i,p}$ can be tested by T^*.

The faults $c_{i,p} \equiv 0$ will be tested by patterns in T^* where the constraint (1) is satisfied. ■

Corollary 1. Any gate-level SAF in the control part related to $F = \{f_i\}$, not detectable by the test $T^* = \{T^*_i\}$ which covers all not redundant high-level control faults of the model $CFM = \{CFM(f_i)\}$, is redundant.

Proof. In Theorem 1, exhaustiveness of using all the combinations of the local control signals $c_{i,1}c_{i,2}\dots c_{i,p}$ was assumed. If not all combinations are used in the instruction set of the microprocessor, which is the typical practical case, then, not all patterns can be generated for activating all SAF of type $c_{i,p} \equiv 1$. Usually these cases are used for optimization of the gate-level structure of the control part of ALU. If however the optimization process has not removed all hardware redundancy, then as the result, the control part may consequently contain also redundant faults. These redundant faults can be identified by simple and fast gate-level fault simulation of the high-level generated test T^*. ■

Example 1. Consider a simplified ALU unit with the set of three functions f_1, f_2, f_3, activated by a set of control signals $\bar{c_2}c_1, c_2\bar{c_1}, c_2c_1$ respectively. The ALU can be represented by the DNF:

$$y = \bar{c_2}c_1y_1 \vee c_2\bar{c_1}y_2 \vee c_2c_1y_3.$$

The test $T^* = \{T^*_1, T^*_2, T^*_3\}$ generated for the control part of ALU that satisfies the constraints (2) is depicted in Table 1.

Table 1. *Example of a high-level control test*

T^*_i	Test	Fault table			Constraints satisfied
	$c_2\ c_1$ $y_1\ y_2\ y_3$	$\bar{c_2}\ c_1\ y_1$	$c_2\ \bar{c_1}\ y_2$	$c_2\ c_1\ y_3$	
1	2	3	4	5	6
T^*_1	0 1 0 1 1	1 1 0	0 0 1	**0** 1 1	$y_1 < y_2$, $y_1 < y_3$
T^*_2	1 0 1 0 1	0 0 1	1 1 0	1 **0** 1	$y_2 < y_1$, $y_2 < y_3$
T^*_3	1 1 1 1 0	**0** 1 1	1 **0** 1	1 1 0	$y_3 < y_1$, $y_3 < y_2$

The table contains the test patterns in column 2, the fault table in columns 3-5, and the constraints satisfied by generating data for the control test patterns in column 6. The detected gate-level faults in the fault table are highlighted by red colour: 0 means the value of a signal which activates the fault SAF/1. For example, in case of the fault $c_2 \equiv 1$ in column 5, the value of the output signal $y = y_1 = 0$ will change from 0 to $y = y_1 \vee y_3 = 1$. For detecting the faults SAF/0, more 3 test patterns are needed (not shown in the table). We see in the fault table that the faults $c_1 \equiv 1$ in column 3 and $c_2 \equiv 1$ in column 4 are not detected, because of the control code $c_2c_1 = 00$ is illegal (not usable in this ALU). According to

Corollary 1, these gate-level faults are redundant (in case if the control circuit is implemented as DNF). As the example shows, the redundancy of the gate-level faults can be derived by simple low-level SAF simulation. ■

Note, Theorem 1 and Corollary 1 were formulated and the proofs were given, considering so far only the single SAF model. In fact, the power of the proposed high-level control fault model stretches far beyond the fault class of single SAF, as it will be shown in the following corollaries.

Corollary 2. The test $T^* = \{T^*_i\}$, covers all gate-level multiple SAF and bridging faults between control lines in the control part of the microprocessor, which controls the set of functions $F = \{f_i\}$.

Proof. From (2) it follows that for each function $F = \{f_i\}$, $\forall k$: $(y_{i/k} < y_{j/k})$ for all $j \neq i$ must hold. This means that not only SAF/1 in a single control signal of a single function $f_j \in F$, $j \neq i$, can be detected (by overwriting $y_{i/k} = 0$ with $y_{j/k} = 1$), where the control words for f_i and f_j differ in a single bit, rather such overwriting of signals $y_{i/k} = 0$ with 1 can happen, and hence, can be detected, due to multiple changes $0 \rightarrow 1$ for $f_j \in F$, $j \neq i$, leading to detecting multiple faults. This explanation can be derived also from reasoning of DNF (5).

On the other hand, from the constraints (1-2), and from the exhaustiveness of testing all the control functions function $f_j \in F$, $j \neq i$, it follows that non-redundant bridging faults between the control lines can also be detected by T^*. ■

In case, when the target would be to detect only single SAF, then the fault model defined by the constraints (1) and (2) is over-dimensioned. For the case of full single SAF coverage, it would be sufficient to loosen the constraint (2) to

$$\forall f_j \in F, j \neq i, (HD(f_j, f_i) = 1) : \{\forall k: (y_{i/k} < y_{j/k})\} \qquad (6)$$

where $HD(f_j, f_i) = 1$ is the constraint that the Hamming distance between the control codes for f_j and f_i is 1. This simplication is similar to the approach used in [29]

Corollary 3. The size of the proposed high-level control fault model applied only to the code-neighboring functions f_j, f_i with $HD(f_j, f_i) = 1$, is equal to $C(\text{CFM}, HD=1) = nmp$.

Proof. The proof is straightforward, since for each $f_j \in F$, instead of $m(n\text{-}1)$, only mp comparisons are needed. ■

The size of the updated with (6) high-level control model is 2 times smaller than for the SAF model C(SAF). Regarding the test cost, since $t_{CFM} << t_{SAF}$, we get

$$TC(\text{CFM}) = nmp\ t_{CFM} << TC(\text{SAF}) = 2nmp\ t_{SAF}. \quad (7)$$

IV. HIGH-LEVEL FAULT COVERAGE MEASUREMENT

From above, it follows that the high-level control fault model CFM defined by the set of constraints (2) can be interpreted as the definition of the universe of high-level control faults. A direct impact of this interpretation is the possibility of evaluating the high-level fault coverage as the percentage of satisfied constraints (2) by the given test. The measuring of the coverage of constraints (1) is not needed, because they will be satisfied anyway as the byproduct of the data path test.

The size of the proposed high-level functional fault model results from the fault table for representing the coverage of satisfied constraints (2).

Let us introduce the high-level fault table as a matrix $D = ||D_{i,j}||$ with n columns and n rows, where n – is the number of functions in F. Each entry $D_{i,j}$ in D is a m-bit vector $D_{i,j} = (D_{i,j/1}, D_{i,j/2}, \ldots, D_{i,j/m},)$, where m is the number of bits in the data-word. $D_{i,j/k} = 1$, if the constraint $y_{i/k} < y_{j/k}$ for the bit k is satisfied, and $D_{i,j/k} = 0$ if not.

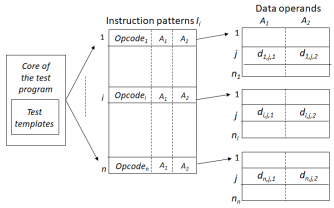


Fig.2. *Architecture of the test program*

Consider a simplified architecture of a test program for testing the control part of ALU as shown in Fig.2. The test $T^* = \{T^*_1, \ldots, T^*_n\}$ for ALU with n functions of the set $F = \{f_1, \ldots, f_n\}$ consists of a core of the test program, array of test patterns (instructions) and array of test data operands. The core consists of a small set of test templates for initializing registers, executing test patterns and processing test results. The test patterns are instructions, and to each instruction, a set of data operands is assigned, to be exercised cyclically. Each test pattern with related operands forms a test $T^*_i \in T^*$. The task of the core is execution of the full test T^*.

For high-level fault simulation, there is no need to simulate the full test program illustrated in Fig.2. Instead of that, only the array of data operands should be processed according to the following procedure.

Procedure 1.
1) **for** $i = 1,\ldots,n$
2) **for** all data operands $d_{i,j,1}$, $d_{i,j,2}$, $j = 1,\ldots,n_i$
3) **for** all instructions f_h, $h = 1,\ldots,n$
4) calculate the value y_h
5) check the relation $y_i < y_h$, $h \neq i$
5) update the vector $D_{i,h} \in D$
6) **end for**

For high-level test generation we developed a simulation based random search for test data to satisfy the constraints (2), where for constraint checking we used Procedure 1.

V. IDENTIFICATION OF HIGH-LEVEL CONTROL FAULT REDUNDANCIES

Consider Table 2, which illustrates a fragment of the high-level fault coverage matrix D, for a test T^* generated for the MiniMIPS processor [24]. In this fragment 8-bit data-words, and 5 functions OUI, ADD, SUB, SLT, AND of the MiniMIPS microprocessor are considered.

In Table 2, the 0s refer to the possible high-level redundancies of the control faults related to the constraints $y_{i/k} < y_{j/k}$, where i and j correspond to the rows and columns, respectively. All 0s in D_{ij} refer to the high probability of

redundancy of the full set of high-level faults for all bits, which means that the constraints $y_{i/k} < y_{j/k}$, for all k cannot be satisfied. In most cases of ALU operations, it is very easy to identify this type of redundancy. For example, if $y_i = f_i(a, b)$ refers to AND operation and $y_j = f_j(a, b)$ refers to OR, it is straightforward that the constraint $y_i < y_j$, i.e. $(a \vee b) < (a \wedge b)$ cannot be satisfied.

Table 2. *Example of a High-Level Fault Table*

	f_1 (OUI)	f_2 (ADD)	f_3 (SUB)	f_4 (SLT)	f_5 (AND)
f_1 (OUI)		111111	111111	111111	**000000**
f_2 (ADD)	11111		11111**0**	111111	111111
f_3 (SUB)	11111	11111**0**		111111	111111
f_4 (SLT)	11111	111111	111111		**000000**
f_5 (AND)	11111	111111	111111	111111	

In Table 2, the 0s refer to the possible high-level redundancies of the control faults related to the constraints $y_{i/k} < y_{j/k}$, where i and j correspond to the rows and columns, respectively. All 0s in D_{ij} refer to the high probability of redundancy of the full set of high-level faults for all bits, which means that the constraints $y_{i/k} < y_{j/k}$, for all k cannot be satisfied. In most cases of ALU operations, it is very easy to identify this type of redundancy. For example, if $y_i = f_i(a, b)$ refers to AND operation and $y_j = f_j(a, b)$ refers to OR, it is straightforward that the constraint $y_i < y_j$, i.e. $(a \vee b) < (a \wedge b)$ cannot be satisfied.

In cases when there is an entry $D_{i,j/k} = 1$ in a single bit k of the vector D_{ij}, or in only few bits of it, we can suggest for the proof a method called "*partial truth table method*". The idea of the method stands in showing the equivalence of partial truth tables (or to prove the impossibility of solving the related constraints) for the functions involved in the constraint relation, so that as few as possible responsible bits should be selected for the need of the proof.

In Table 3, examples are shown for 1-bit partial truth tables for the functions SUB, ADD, OR, AND, for bit k. The pairs 00, 01, 10, 11 represent the values of the data variables (as arguments) in bit k, and the 1-bit values in the columns show the results of the related operations for this k-th bit. For SUB and ADD, the equivalence of the behavior in the given bit is demonstrated, which contradicts to the constraint (2), and in the case of OR and AND, the missing of a solution for (2) is also shown for all possible input data combinations.

It is easy also to show for example, the equivalence of operations ASR and SHR for MiniMIPS for all bits, except the most significant bit MSB. Hence, for all bits except for MSB, the entry $d_{i,j/k} = 0$ refers to the redundant control fault.

In some cases, the partial truth table method will not work, because the results of operations may substantially depend on all bits of the word like for increment or decrement operations. When this happens, specific corner cases should be found for the proof of redundancy. For example, to prove the equivalence of increment and decrement operations in the least significant bit, the operand 1…110 should be used, where both instructions INC and DEC produce the same result "all 1s".

Table 3. *Examples of redundancy proofs with 1-bit truth tables*

#	$y_{i/k} < y_{j/k}$	D_{ij}	$y_{i/k} < y_{j/k}$	00	01	10	11
1	SUB < ADD	1…11**0**	SUB	0	1	1	0
			ADD	0	1	1	0
2	OR < ADD	1…11**0**	OR	0	1	1	1
			ADD	0	0	0	0

VI. MIXED-LEVEL IDENTIFICATION OF FAULT REDUNDANCIES

Let us now draft the general procedure of the mixed-level identification of gate-level single SAF, where the test is generated at the high-level using the proposed high-level control fault model, and the redundancy of the low-level SAF is identified by low-level fault simulation of the test, generated at the high-level.

Procedure 2.

1) Generation of the high-level test T^* for the given set of functions (instructions), with finding the data which satisfy the constraints (2) (see Section II).

2) Generation of the high-level fault coverage table D by high-level fault simulation of the test T^* (see Section IV). The steps 1 and 2 can be carried out jointly (see Section IV).

3) High-level fault redundancy identification. For all not covered high-level faults (all 0s in the fault table D), the redundancy of the high-level control faults is identified (see Section V).

4) If the high-level redundancy cannot be proven for some of high-level fault

s, the test T^* must be extended to satisfy the constraints (2), and to achieve 100% high-level fault coverage. This is the prerequisite (Theorem 1) for the next step of redundant SAF identification.

5) Gate-level fault simulation of the test T^*. The not detected SAF are identified as redundant low level faults in the control circuit of ALU (Corollary 1).

VII. EXPERIMENTAL RESULTS

We carried out experiments which consists of high-level test data generation for the control part of Execute stage of MiniMIPS processor [31], Fig.3. The test program generation included manual synthesis of test templates, high-level generation of test data (operands) to satisfy constraints (1-2), test program synthesis and high-level fault simulation. For high-level test generation and fault simulation we used home-made tools, whereas for gate-level operations we used commercial tool. Experimental results are shown in Table 4.

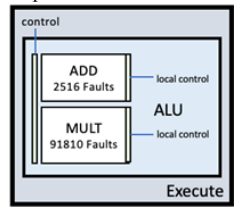


Fig.3. Simplified structure of the Execute stage of MiniMIPS

***Table 4**. Experimental data*

Approach	Experiments		Faults	FC%	# Pat	ATPG time
Proposed high-level approach	High-level ATPG		756	100	196	47 s
	Gate-level simulation	ALU	2516	99.92		
		MULT	91810	99.09		
Commercial gate-level ATPG		ALU	2516	99.96	169	1h 34 min
		MULT	91810	98.63		

The experiments targeted ALU and MULT modules in the Execute stage of MiniMIPS. We generated a test with 100% coverage of high-level control faults. The operands generated according to (1-2), produced high gate-level SAF coverage for both, control and data parts of the Execute module.

The high-level test was simulated by commercial tool to grade the gate-level SAF coverage. To evaluate the efficiency of the high-level ATPG, we used for comparison also commercial gate-level ATPG. The time cost for high-level ATPG is about two orders of magnitude less than that of the commercial ATPG. The gate-level SAF coverage, achieved by the proposed ATPG for the whole module under test, is better than that achieved by the commercial tool.

The main goal of the experiments was to demonstrate the possibility of identification by high-level test generation the gate-level SAF redundancies. We demonstrated it on the basis of ALU test. The SAF coverage 99.92, achieved by 100% high-level fault coverage test, means that 2 faults remained in ALU not detected, and are qualified, according to Corollary 1, as redundant. Since by fault simulation of the test for ALU (without its local control part) we found 100% SAF coverage, we can conclude that the 2 faults belong to the ALU control part. On the other hand, since low-level ATPG found 1 undetected fault in the ALU joint data/control circuit, we can conclude that this redundant fault belongs to the ALU local control part, and the second redundant fault belongs to the ALU global control part (see Fig.3).

In the MULT block, fault coverage 99.09 refers to 835 not covered faults, which should be qualified according to Corollary 1 as redundant. By gate-level ATPG we found that from the 1256 not covered by ATPG faults, 865 were ATPG untestable, 105 were classified as redundant, and 286 remained not detected. From the latter it follows, that the 444 faults (the difference 1256 – 835), not covered by gate-level ATPG, however, were covered by the high-level ATPG. These faults should belong to the class of gate-level ATPG untestable faults.

VIII. CONCLUSIONS

In this paper, we propose a novel high-level fault model which was experimented for test generation for ALU control parts in processors. The model consists of a set of data constraints to be satisfied by data operands that is to be used in the test. The constraints are derived from instruction set, in which case, no implementation details are needed. The test is able to detect all non-redundant single and multiple SAF, and bridging faults in the control circuit under test. Hence, the proposed method is more powerful than the traditional ATPGs, which target only single SAF. A metric and a method for high-level fault simulation with a method for identification of high-level fault redundancies were developed. We demonstrated the feasibility of the proposed method to identifying low level redundant SAF by combining high-level ATPG and low level SAF simulation. The test program generated explicitly for testing only the control part achieves as well a very high fault coverage for data part. This is due to the power of constraints (1-2) to be used for selecting data operands.

The future work will be to extend the proposed method for broader instruction sets of processors. Several optimization techniques are also possible.

Acknowledgment: The work has been supported by EU's H2020 project RESCUE, Estonian research grant IUT 19-1, and funded by Excellence Centre EXCITE in Estonia.

REFERENCES

[1] L.-T.Wang, Ch.-W.Wu, X.Wen. VLSI Test Principles and Architectures. Design for Testability. Elsewier, 2006.
[2] P. Georgiou1, X. Kavousianos1, R.Cantoro, M. Sonza Reorda. Fault-Independent Test-Generation for Software-Based Self-Test. IOLTS, Costa Brava, Spain 2018.
[3] R.Ubar, S.Kostin, J.Raik. How to Prove that a Circuit is Fault-Free? Proc. EUROMICRO, Cesme, Turkey, Sept. 5-8, 2012´.
[4] L.Zhuo et.al. A Circuit Level Fault Model for Resistive Opens and Bridges. Proc. VLSI Test Symp., Napa, CA, Apr./May 2003, pp. 379-384
[5] H.K.Lee, D.S.Ha. SOPRANO: An Efficent Automatic Test Pattern Generator for Stuck-Open Faults in CMOS Combinational Circuits. DAC, 1990.
[6] A.Kristic, K.T.Cheng. Delay Fault Testing for VLSI Circuits. Dordrecht, The Netherlands, Kluwer Academic Publishers, Oct. 1998.
[7] J.P.Roth. Diagnosis of Automata Failures: A Calculus and a method. IBM J. Res. Develop., Vol. 10, No. 4, pp. 278-291, July 1966.
[8] R.D.Blanton, J.P.Hayes. On the Properties of the Input Pattern Fault Model. ACM Trans. Des. Automat. Electron. Syst., Vol. 8, No. 1, pp. 108-124, Jan. 2003.
[9] K.B.Keller. Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms. US Patent 5546408, Aug. 13, 1994.
[10] R.Ubar. Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations. Automation and Remote Control, Vol.40, No 11, part 2, Nov. 1980, Plenum Publishing Corporation, USA, pp. 1693-1703.
[11] J.Raik, R.Ubar, J.Sudbrock, W.Kuzmicz, W.Pleskacz. DOT: New Deterministic Defect-Oriented ATPG Tool. IEEE ETC, 2005.
[12] U.Mahlstedt, J.Alt, I.Hollenbeck. Deterministic Test Generation for Non-Classical Faults on the Gate Level. 4th ATS., Bangalore, Nov. 1995.
[13] S.Holst, H.-J.Wunderlich. Adaptive Debug and Diagnosis Without Fault Dictionaries. Proc. of 13th ETS, Verbania, Italy, May 2008, pp.199-204.
[14] Y.Cho, S.Mitra, E.J.McCluskey. Gate Exhaustive Testing. ITC 2005.
[15] A.Jas, S.Natarajan, S.Patil. The Region-Exhaustive Fault Model. 16th Asian Test Symposium. Beijing, China, Oct. 2007, pp. 13-18.
[16] K.N.Dwarakanath, R.D.Blanton. Universal Fault Simulation using fault tuples. DAC, Los Angeles, June 2000, pp.786-789.
[17] Psarakis, M., Gizopoulos, D., Paschalis, A., Zorian, Y. Sequential Fault Modeling and TPG for CMOS It. Logic Arrays. IEEE Trans. on Comp, vol.49, no.10, 2000.
[18] F.Happke et al. Cell-Aware Test. IEEE Trans. on CAD of IC, vol.33, no. 9, 2014.
[19] L. Shen and S. Su, "A Functional Testing Method for Microprocessors," IEEE Transactions on Computers, vol. 37, no. 10, pp. 1288-1293, 1988.
[20] F. Corno, G. Cumani, M. Sonsa Reorda, G. Squillero. An RT-Level Fault Model with Gate Level Correlation. Int. High Level Design Validation Workshop, 2000.
[21] C.Cho, J.Armstrong. A Behavioral Test Generation Algorithm. ITC, 1994.
[22] V. Kumar et al. Employing Functional Analysis to Study Fault Models in VHDL.Int. J. of Scientific Engineering and Technology, vol.1, no.5, 2012.
[23] M. Bushnell and V. Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Springer, 2013.
[24] P. Thaker, V. Agrawal and M. Zaghloul, "RT Level Modeling and Test Evaluation Techniques for VLSI Circuits," in ITC, 2000
[25] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. SBST of embedded processors. IEEE Transactions on Computers, vol. 54, no. 4, 2005.
[26] C.H.P.Wen, L.C.Wang, K.-T. Cheng Simulation-based functional test generation for embedded processors. IEEE Trans. on Computers, vol. 55, no. 11, 2006.
[27] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero. Automatic test program generation: a case study. IEEE Design Test of Computers, vol. 21, no. 2, 2004.
[28] D. Gizopoulos, et al. Systematic software-based self-test for pipelined processors IEEE Transactions on VLSI Systems, vol. 16, no. 11, 2008.
[29] P. Bernardi, et al. On the in-field functional testing of decode units in pipelined RISC processors. Int. Symp. on DFT in VLSI and Nanotechnol. Systems, 2014.
[30] E. Sanchez and M. S. Reorda, "On the functional test of branch prediction units," IEEE Trans. on VLSI Systems, vol. 23, no. 9, 2015.
[31] OpenCores, "MiniMIPS ISA".

Appendix 9

IX
A. S. Oyeniran, R. Ubar, M. Jenihhin, and J. Raik, "High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors," *Journal of Electronic Testing*, vol. 36, no. 1, pp. 87–103, 2020

Noname manuscript No.
(will be inserted by the editor)

High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors

Adeboye Stephen Oyeniran · Raimund Ubar · Maksim Jenihhin · Jaan Raik

Received: date / Accepted: date

Abstract The paper proposes a novel high-level approach for implementation-independent generation of functional software-based self test programs for processors with RISC architectures. The approach enables fast generation of manufacturing tests with high stuck-at fault coverage. The main concept of the method is based on separate test generation for the control and data parts of the high-level functional units. For the control part, a novel high-level control fault model is introduced whereas for the data part, pseudo-exhaustive test approaches can be applied to keep the independence from the implementation details. For the control parts, a novel high-level fault simulation method is proposed for evaluating the high-level fault coverage. The approach can be used for easy identification of redundant gate-level faults in the control part. The redundant faults can be identified by simple gate-level fault simulation of the generated high-level test when implementation is available. Experimental results of test generation for different units of a RISC processor support the solutions presented in the paper.

Keywords Processor testing · high-level control fault model · functional test generation · fault simulation · high-level fault coverage · low-level fault redundancy

1 Introduction

Technology scaling in today's deep-submicron processes produces new failure mechanisms in electronic devices. This has forced researchers both, to develop

A. Stephen Oyeniran
Akadeemia tee 15A, 12618 Tallinn
E-mail: adeboye.oyeniran@taltech.ee

Raimund Ubar
Akadeemia tee 15A, 12618 Tallinn
E-mail: raiub@pld.ttu.ee

more advanced fault models compared to the traditional stuck-at fault (SAF) model [1], and to investigate the possibilities of reasoning the faulty behavior of systems without using any particular fault models [2, 3]. The importance of the latter approach is ever increasing, due to the growing number of low-level redundant faults in today's complex systems. On the other hand, the implementation details of the processor circuits are not always available, when testing is needed.

Fault models for digital circuits have been developed for different types of failure mechanisms such as signal line bridges [4], transistor stuck-opens [5] or failures due to increasing circuit delays [6]. Another trend has emerged to develop general fault modeling mechanisms and corresponding test tools that can effectively analyze arbitrary fault types. The oldest example is the D-calculus [7]. A generalization of this approach has been found in the input pattern fault model [8], and in the pattern fault model [9]. These can represent any arbitrary change in the logic function of a circuit block, where a block is defined to be any combinational sub-circuit described at any level of the design abstraction.

A similar pattern related fault modeling approach called functional fault model was proposed earlier in [10] for the module level fault diagnosis in combinational circuits. The functional (*or pattern*) fault model allows an arbitrary set of signal lines to be grouped into activation conditions for a single fault site, allowing a variety of physical defect types to be modeled. Based on the functional fault model a deterministic defect-oriented test pattern generator DOT was developed in [11] which allowed proof of the logic redundancy of not detected physical defects.

In [12], a similar model called *conditional faults* was proposed for test generation purposes, and in [13] for diagnosis purposes. A conditional fault allows additional signal line objectives to be combined with the detection requirements of a particular fault. To completely exercise gate-level blocks in combinational circuits, a similar pattern oriented gate-exhaustive fault model was proposed in [14], which was extended to target bigger regions (collections of gates) by *region-exhaustive* fault model in [15].

The described functional, conditional and pattern fault models offer high flexibility in defect modeling beyond single SAF model. Further advancements of the low-level fault modeling have been achieved by introducing the fault tuple fault model [16], realistic sequential cell fault model [17], or cell-internal defect model [18], where the last two cases provide general capability of handling sequential misbehavior of circuits.

The conditional SAF model (and also other listed models) [8–18] support hierarchical test approach, where the test pattern (or sequence), which activates a low level fault (e.g. physical defect) at the lower level, can be considered as the high-level condition (or constraint) for the functional fault defined at the higher level.

To increase the speed of test generation and fault coverage evaluation, high-level (*functional or behavioral*) fault models have been developed. Such a model can be considered efficient, if the tests generated using this model

provide a high coverage of SAF or physical defects. However, a formal and uniform high-level fault model for testing of processors is still missing.

In the design abstraction, higher-level descriptions have fewer implementation details, but more explicit functional information than lower level descriptions. High-level fault models are categorized by the level at which the tests are generated. Usually, the methods for high-level test generation are divided into structural Register-Transfer Level(RTL) based methods [19, 20], or behavioral test generation methods [21, 22]. A high-level fault model can be explicit or implicit [23, 24]. An explicit model identifies each fault individually, and every fault in this model will be a target for test generation. Implicit models are based on the assumption that all gate-level faults may not be represented at the RT level, and this motivated to develop dedicated RTL fault models with dependence on implementation details.

High-level fault models are used widely in the field of Software-Based Self-Test [25–30]. These approaches can be divided into two major groups - structural and functional. Structural approaches, such as [25, 26], are based on test generation using information from lower level of design (gate-level or RTL-level description) of processor under test. Functional approaches on the other hand use instruction set architecture (ISA) information of the processor under test [27–30].

The main and general problem of high-level faults is the difficulty of proving that the model covers all low-level detectable (non-redundant) faults. In existence of such a high-level proof, it would be possible to identify the redundancy of gate-level faults exclusively by only gate-level fault simulation, which has a cheaper cost than low-level fault redundancy proof by conventional gate-level ATPG-s.

We have made such an attempt for the class of circuits with well-defined high-level functionality in [31], and the results are extended in the present paper. We propose a novel method for implementation-independent test generation for modules of RISC type microprocessors. The method differs from known methods by giving the tests capability of detecting larger class of faults than the traditional single stuck-at faults (SAF). The new method provides tests which are capable of detecting single and multiple SAFs, conditional SAFs, shorts, unknown cell defects, and functional decoder faults (similar to that of address decoder faults considered in memory test). Delay faults and the faults which convert combinational circuits into sequential ones are not considered in this paper.

The implementation independence of tests and the capability of detecting a larger class of faults are achieved by separate testing of control and data parts of the module, by applying pseudo-exhaustive test patterns to the data part, and by using the novel data constraints based functional control fault model, which allows to prove the correctness of the control part. The test generation details for the data part are not discussed, but in the experimental research we relied on our previous research results [32, 33].

We demonstrate the universality of the method by generating tests for different processor modules, such as ALU, Execute module, Register bank,

Forwarding unit in pipelines, and Coprocessor unit. Based on the proposed high-level control fault model, we propose also a method of mixed-level identification of redundant low-level structural faults, which is faster than the traditional ATPG-based methods for redundancy proof.

The rest of the paper is organized as follows. In section 2, we describe the main concept of the implementation-independent high-level functional testing, and in Section 3, we present a method of high-level modeling of modules under test. Section 4 is devoted to development of a novel high-level control fault model as the basis of the proposed approach, while Section 5 deals with High-level Implementation-independent Control Test. In sections 6 and 7 we describe a novel mixed level method of identification of redundant faults for the high and low level control fault classes, respectively. Section 8 presents a case study for demonstrating how the method can be used in other types of modules, in particular, for Pipeline Forwarding Unit, and Section 9 presents the overall automated test program generation scheme and shows the test organization architecture. Section 10 discusses experimental results. Section 11 concludes the paper. Appendix A gives a theoretical foundation of the proposed high-level control fault model and Appendix B gives a theoretical foundations for identification of not-detectable low-level faults in the control circuits.

2 The proposed Implementation-independent Concept of High-level Testing of RISC processors

The objective of this research is to propose a novel method for high-level testing of the modules of RISC microprocessors in a functional way and without resorting to the knowledge of low-level implementation details. The main concept of the proposed method is based on the abstract presentation of the module under test (MUT) as a model which consists of two parts: the control part and the data part. For both, we generate separate high-level implementation-independent test programs.

The main novelty of the approach is to give the generated tests the property of detecting larger class of structural faults in the implementation, compared with the known approaches, such as not redundant single and multiple stuck-at faults, shorts, unknown cell defects and functional decoder faults. The faults, which convert combinational circuits into sequential ones, and delay faults are not considered. The detection of that larger fault class is achieved by the following strategies. The data part is tested by pseudo-exhaustive test pattern sequences, which according to definition, are capable of detecting all not redundant structural faults in sub-circuits tested exhaustively, independently of implementation, assuming the fault is not converting the circuit into a sequential one.

For the control part, which is the main objective of the paper, we have developed a novel data constraints based functional control fault model, which will serve as a proof that all functions will be selected correctly in case of the a successful result of the control test.

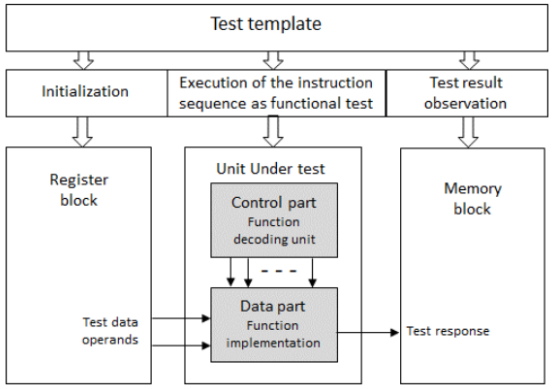


Fig. 1: Overview of the proposed high-level test concept of a processor module

For each high-level module under test we define the set of logic functions of the processor as the target of testing. Such a set of functions is defined on the basis of the instruction set, or by examining the architecture of the processor. For testing the parts of the processor, not directly described by instruction set, such as pipeline architecture or forwarding circuitry, we have to define the set of functions on the basis of the description of the processor architecture given usually in manuals.

The full set of functions under test is partitioned into subsets or groups of functions, for which the results of executed functions can be observed in comparable way at the same node of the high-level structure. For example, one group of functions may consist of logic and arithmetic operations derived from the instruction set and another group of functions may be defined for the forwarding functionality of pipeline, derived from the architectural description of the processor. In both cases, the observable node will be the output of ALU unit. Another group of functions may be the branch functionality, which differs from the previously mentioned groups in having the observable node in the ALU 1-bit flag, which determines the branch direction of the instructions flow.

For all the functions of a particular group, the same test template (sequence of instructions) will be constructed, which consists of the data initialization sequence applying the pattern which executes the particular function under test, and the observation sequence. The latter is needed for observation of the results of the test at the same node in the circuit, determined for the whole group of functions under test.

A big picture of the processor's high-level test organization for a particular module is depicted in Fig.1. The module will have a single or several test templates for compiling assembler test routines, to be executed in embedded loops using the subset of instructions and related data sets.

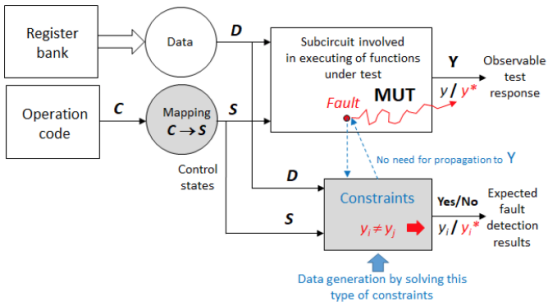


Fig. 2: Illustration of the proposed test concept

The proposed new concept of high-level implementation-independent functional control test approach is illustrated in Fig.2. Consider MUT as a functional unit as a set F of functions $y = f_i\,(s, d)$, $f_i \in F$, represented at high-level in a vector form $Y = F\,(S, D)$.$S = \{s_i\}$ is the set of control states, $s_i \rightarrow f_i$, activating the related functions f_i, $D = \{d\}$ is the set of data operands generated for testing the functions f_i, and Y is the set of output responses y_i of MUT at given values of $s_i \in S$ and $d \in D$. Since the low-level structure of the MUT is unknown, the decoding details of the operation code C into the control signals is also unknown. For that reason, we ignore these mapping details and introduce direct one-to-one mapping between the control states $s_i \in S$ and the functions $f_i \in S$. This mapping is sufficient for distinguishing the behaviors of all functions, independently of how the control signals are coded by the operation code in the instruction format. The possible faults in the mapping $C \rightarrow S$, will be indirectly taken into account when generating tests for the set of functions $Y = F\,(S, D)$. It will be shown in Section 7.

Since the internal logic structure of the MUT is unknown, traditional methods of path activation from the fault site up to an observable node cannot be used. For example, a fault at some location inside the circuit of MUT cannot be propagated to the observable point to produce there a faulty value y^* instead of the expected value $y \neq y^*$, $y \in Y$.

We introduce a novel concept of detecting the faults by replacing the idea of traditional fault propagation through the structure of the circuit with comparing the expected values y with possible erroneous values y^* produced by faults inside the MUT directly at the observable point Y, without the need of fault propagation along the structure of the circuit. In such a way, we map the structural problem of fault propagation into the functional problem of comparing the expected responses of the test with faulty responses.

We reformulate the target of testing the correctness of the control part of MUT as the target to distinguish each function from all others. This target can be achieved by generating a proper set of data $D = \{d\}$, which will produce

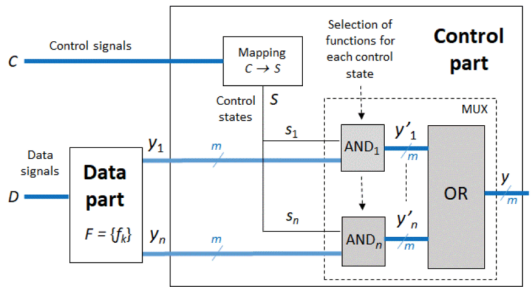


Fig. 3: Implementation independent ALU model

for all functions $f_i \in F$ the corresponding output values $Y = \{y_i\}$, so that they would differ from each other at least once in each bit.

Fig.2 illustrates how the upper black-box model of MUT is replaced with lower constraints based functional model for the control part of ALU. Here $y_i \neq y_j$ means that if there are data generated in D, which satisfy this constraint for expected results of functions f_i and f_j, then these functions are distinguishable, and opposite, if this constraint at least in one bit is not satisfied, some faults can remain not detectable by the test.

In the next Sections, we first, present the high level model of MUT, and thereafter, we introduce a novel high-level functional control fault model, which is purely data related and has no relations to the internal structure of MUT.

3 High-level modeling of modules under test

Consider ALU of a processor as a module under test (MUT) of a digital system, composed of data and control parts as shown in Fig.3. We present it in implementation-independent generic way as an equivalent circuit, where the control part is highlighted as AND-OR multiplexer, which extracts the results $y_i \in Y$ of the functions $f_i \in F$. The functions f_i are executed by instructions, which produce the respective control state signals $s_i \in S$.

Let us ignore possible optimizations of the MUT circuit, and expand it in the presented way to separate the data and control parts. This is similar to the procedure of expanding a given optimized Boolean expression by opening the parentheses and producing DNF with different appearances of the same variable. Hence, the circuit in Fig.3 can be presented as DNF:

$$y = (s_1 \& y_1) \vee (s_2 \& y_2) \vee \ldots \vee (s_n \& y_n) \tag{1}$$

where for Boolean control variables we have constraints $\forall s_i, s_j \in S$: $s_i \& s_j = 0$, and $y_i = f_i(d)$ are the Boolean functions executed in the data part, which are represented as well as separate DNFs.

We call the formula (1), after having substituted the variables y_i with respective Boolean expressions and opened all parentheses, as equivalent disjunctive normal form (EDNF), similarly to [37], to stress that to each appearance of the literal in EDNF, a different path in the original circuit corresponds. As shown in [37], a test for a literal appearance sensitizes the path in the original circuit associated with that literal appearance. It was also shown that if the generated test will detect all irredundant faults in the expanded EDNF, it will detect also all irredundant faults in the original optimized version of the original circuit.

From that it follows that the test generated for all irredundant faults in the EDNF representing the expanded model of MUT will test also all irredundant faults in the original optimized version of the implemented circuit of MUT.

Consider the following example. Let the formula (2) represent an EDNF for a MUT represented as the model in Fig.3, which executes 4 functions $y_1 = d_1 d_2$, $y_2 = d_1$, $y_3 = d_2$, $y4 = \overline{d_1} d2$, where $d_1, d_2 \in D$ represent data operands, under the control of variables s_1, s_2, s_3, s_4, respectively, and let the formula(3) represent the optimized implementation of the same MUT:

$$y = s_1 d_{11} d_{21} \vee s_2 d_{12} \vee s_3 d_{22} \vee s_4 \overline{d_{13}} d_{23} \tag{2}$$

$$y = d_1 \left(s_1 d_2 \vee s_2\right) \vee d_2 \left(s_3 \vee s_4 \bar{d}_1\right) \tag{3}$$

The test for all SAF of the literal appearances in EDNF (2) will have the length of 10 patterns, and it will detect all SAF of the variables in the optimized formula (3). The disadvantage of the approach is that the optimized implementation(3) can be tested by a shorter test consisting of 8 patterns. However, the test for (2) is more general, because it is implementation independent, and as it will be seen, it will cover using proposed method of test generation, larger fault class than SAF. On the other hand, if application will be given, the general test for the expanded model can be always minimized for the given class of faults.

We introduced the example to explain the idea of making test generation implementation independent. In fact, the proposed method does not need to descend to the EDNF level. We will stay at the higher level described by the generic formula (1), generate a separate test for the control variables $s_i \in S$ of the control part (to test the instruction decoder and the multiplexer in Fig.3), and then apply the pseudo-exhaustive test sequences one-by-one for the functions $y_i = f_i(d)$ of the data part. The values (test responses) of y_i will propagate to the output Y correctly, in condition if the test for the control part has passed.

4 High-level Implementation-independent Control Fault Model

To test the control part we propose a new high-level functional control fault model. We will show also, that if the tests, generated for the high-level function $Y = F(S, D)$ of the MUT, using the proposed control fault model, will produce

correct expected responses for the given implementation $Y^* = F^*(C, D)$, then the faults in the control part of the implementation are missing.

Denote by y_i the data word considered as the result of execution of the function f_i of data operand(s) $d \in D_i$ as $y_i = f_i(d)$. Let the number of bits in the data words be p, and let $y_{i/k}(d)$ denote the value of the kth bit of the data word calculated as $y_i = f_i(d)$. Let $D_i \subseteq D$ be the subset of test data operands generated for executing the function f_i.

Definition 1 Introduce for the function $f_i \in F$, the following high-level control fault model $M(f_i)$ as a set of data constraints:

$$\forall k \in (1, m) \exists d \in D_i \left(y_{i/k}(d) \neq 0\right) \tag{4}$$

$$\forall f_{ij \neq 1} \in F \forall k \in (1, m) \exists d \in D_i \left(y_{i/k}(d) < y_{j/k}(d)\right) \tag{5}$$

to be satisfied by a set of data operands D_i, at least by one pattern $\{D_i\}$ for each bit k of the data word y_i. The full high-level control fault model $M(F)$ for a set of functions F, can be considered as a joint set of constraints (4) and (5) in a form

$$M(F) = \bigcup_i M(f_i), i : f_i \in F$$

to be satisfied by a set of data operands D, at least by one data pattern $d \in D$ for each bit k of the data word y_i, whereas

$$D = \bigcup_i D_i \quad i : f_i \in F$$

Note; depending on the technology implemented in the microprocessor, the constant 0 in formula (4) can be changed into 1, and instead of the relation "$<$" in formula (5), there can be "$>$" in case of the true value "1" is represented by low voltage, and "0" by high voltage.

Definition 2 Let us introduce the list of high-level functional control faults as the list of all constraints (5):

$$L = \left\{r_{i,j/k} : \left(y_{i/k} < y_{j/k}\right)\right\}$$

We say that $r_{i,j/k} = 1$, if the constraint $(y_{i/k} < y_{j/k})$ is satisfied at least once, by one of the test data $d \in D$. This is equivalent of saying, that the fault $r_{i,j/k}$ is covered by the test generated for the set of functions F.

Definition 3 Let us introduce the matrix $R = \| r_{i,j/k} \|$ to be called *high-level functional control fault table*. The rows and columns of this matrix will represent the functions f_i and f_j, respectively, whereas the entries represent p-bit Boolean vectors. The procedure, which calculates the values of $r_{i,j/k}$ for the given test set is called high-level functional control fault simulation.

We have developed an automated test data generation tool as implementation of the method [31] for solving the constraints in Definition 1. The high-level fault simulator for producing the fault table $R = \| r_{i,j/k} \|$, and for calculating the high-level functional fault coverage of generated test, implements the procedure that was presented in [43] and a theoretical foundation of the proposed high-level control fault model was presented in Appendix A.

5 High-level Implementation-independent Control Test

Definition 4 Let us introduce a control test $T(f_i)$ as a sequence of the repeatedly executed subroutine for testing the function $f_i \in F$ in a loop for all test data $di \in D_i$ generated using the fault model $M(f_i)$. The subroutine is constructed using a template, dedicated for the set of functions F, which includes initialization of test data $d_i \in D_i$, execution of the instruction under test, and observation sequence. Denote the full control test for F as $T(F)$, which represents the sequence of $T(f_i)$.

The high-level functional control fault test, which satisfies the constraints (4, 5) will cover the following fault classes (for each control word bit):

1. For MUX (Fig.3): SAF/0 due to constraints (4), and SAF/1 (due to constraints (5) on the inputs and related paths to outputs; conditional SAF, shorts, and multiple SAF on the inputs of AND-gates (due to applying exhaustive patterns)
2. Functional faults in the instruction decoder: no function accessed, multiple functions simultaneously accessed – similar to address decoder faults of memory test (due to constraints (5)) [35]
3. Functional microprocessor faults: instead of function f_i, another function f_j accessed, or multiple functions simultaneously accessed (due to constraints (5)) [36].

Note, the proposed idea of data constraints based functional testing of decoders (4, 5) is close to that used in memory testing. In Fig.4, the proposed method is compared with the fragment of memory test. In case of memory the initialization of constraints (writing 1s ($w1\uparrow$) into cells) can be done once for all cells in a single cycle. Then, having these constraints stored, the following test cycle ($r/w0\downarrow$) and observation cycle ($r1$) can be carried out. In the proposed method, the constraints cannot be stored, rather they have to be produced "on-line" at each test pattern. In Fig.4, a test pattern is illustrated by showing the values it produces for functions. All functions are partitioned into two groups Y^1, with values $y_j = 1$, and Y^0, with values $y_j = 0$. The selected function f_i under test produces value $y_i = 0$. We see that the constraint (5) are satisfied only for the functions related to the group Y^1. For other functions, the test for f_i has to be repeated with other data until the constraint (5) will be satisfied for all bits of the data word. In the examples on Fig.4, only 1-bit cells of all memory locations and 1-bit data words for all functions f_i , $f_j \in F$ are considered.

This comparison of the data constraints based test $T(F)$ with March test reveals the possibility of applying the proposed approach to other modules, for which the data path state initialization task for executing the functions $f_i \in F$ and sensitization of faults needs to load with data $d \in D$ more than two registers, as in case of ALU. An example is the pipeline forwarding unit to be discussed in Section 8. Another example is the register bank, where our method can be directly compared with March test, however, with the difference

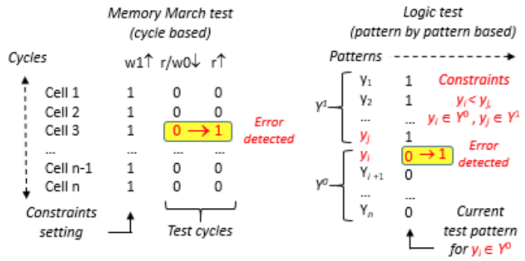


Fig. 4: Comparison of March test and the proposed data constraints

in the number of destinations to be observed (in case of the register bank test, a single register was used).

6 Identification of Redundant High-level Control Faults

Consider as an example Table 1, which illustrates a high-level control fault table $R = \|r_{i,j/k}\|$ for a test $T(F)$, generated for 5 functions $f_1\,(OR)$, $f_2\,(ADD)$, $f_3\,(SUB)$, $f_4\,(XOR)$, and $f_5\,(AND)$. Assume the length of the data words is $k = 6$ bit. The entries 0 (highlighted in red) in Table 1 refer to faults $r_{i,j/k} \in R$, not detectable by the test $T\,(F)$, as the constraints $y_{ik} < y_{jk}$ (5) have not been satisfied by generating test data for T. For these faults, either the test $T\,(F)$ has to be improved, or it should be proven that the not detected faults are functionally redundant.

In the following, we propose a method of identification of the high-level control fault redundancy, which in most cases is not difficult to perform due to the quite understandable high-level functional fault universe.

Example 1 For example, in most cases of ALU operations, it is very easy to identify this type of redundancy. For example, if a 1-bit function $y_i = f_i\,(d_1, d_2)$ refers to AND operation and $y_j = f_j\,(d_1, d_2)$ refers to OR, it is straightforward that the constraint $y_i < y_j$, i.e. $(d_1 \vee d_2) < (d_1 \wedge d_2)$ cannot be satisfied as it was shown in Fig.13.

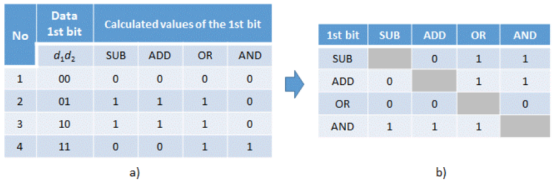
Table 1: Example of a High-Level Fault Table

	f_1(OUI)	f_1(ADD)	f_1(SUB)	f_1(SLT)	f_1(AND)
f_1(OUI)		111111	111111	111111	000000
f_1(ADD)	111111		111110	111111	111111
f_1(SUB)	111111	111110		111111	111111
f_1(SLT)	111111	111111	111111		000000
f_1(AND)	111111	111111	111111	111111	

In cases where there is an entry $r_{ijk} = 0$ in a single bit k of the vector r_{ij}, or only in few bits of it, we can suggest for the proof a method which can

be called as "partial truth table method". The idea of the method stands in showing the equivalence of partial truth tables (or to prove the impossibility of solving the related constraints) for the functions involved in the constraint relation, so that as few as possible responsible bits should be selected for the need of the proof.

In Fig.5, examples are shown for 1-bit partial truth tables for the functions SUB, ADD, OR, AND, for the least significant bit. The pairs 00, 01, 10, 11 in Fig.5a represent the values of the data variables (as arguments) for the bit under analysis, and the 1-bit values in the columns show the results of the related. From comparison of the columns 3-6 in the table of Fig.5a, straightforwardly the entries into the fault table in Fig.5b result.



No	Data 1st bit	Calculated values of the 1st bit			
	d_1d_2	SUB	ADD	OR	AND
1	00	0	0	0	0
2	01	1	1	1	0
3	10	1	1	1	0
4	11	0	0	1	1

a)

1st bit	SUB	ADD	OR	AND
SUB		0	1	1
ADD	0		1	1
OR	0	0		0
AND	1	1	1	

b)

Fig. 5: Example of redundancy proofs with 1-bit truth table

In some cases the partial truth table method will not work, because the results of operations may substantially depend on all bits of the word like for increment (INC) or decrement (DEC) operations. When this happens, specific corner cases should be analyzed to prove the redundancy. For example, to prove the equivalence of increment and decrement operations in the least significant bit, the operand $1 \ldots 110$ has to be used, where both instructions INC and DEC produce the same result "all 1s", which proves the functional redundancy of both functions in this bit.

7 Identification of Redundant Low-level Faults in the Control Part of UUT

Let us consider in the following, how the gate-level fault redundancies in the control part of ALU can be identified by mixed level fault reasoning. To reduce the complexity of the problem, we propose, instead of exploiting slow conventional gate-level ATPGs for SAF redundancy proof, to use the combination of faster high-level test generation, and faster than ATPG low level fault simulation to achieve the same result – identification of the redundant low-level faults.

In Appendix B, we gave a theoretical foundations for identification of not-detectable low-level faults in the control circuits. A test that covers all non-redundant high-level faults was shown to also cover all gate-level testable SAF in the microprocessor's control part.

We will illustrate with example 2, the identification of redundant low-level faults in the control part of a simplified ALU.

Example 2 Consider a simplified ALU unit with the set of three functions f_1, f_2, f_3, activated by a set of control signals $\overline{c_2}c_1, c_2\overline{c_1}, c_2c_1$ respectively. The ALU can be represented by the EDNF:

$$y = \overline{c_2}c_1y_1 \vee c_2\overline{c_1}y_2 \vee c_2c_1y_3$$

The test data $D = \{d_1, d_2, d_3\}$ generated for the control part of ALU that satisfies the constraints (2) is depicted in Table 2. In the column 6, it is also shown, taking into account the values of y_i in column 2, that all 1-bit constraints (5) needed for high-level test of the MUT, are satisfied.

Table 2: Example of a high-level control test

D_i^*	Test		Fault Table			Constraints Satisfied
	c_2c_1	$y_1y_2y_3$	$\overline{c}_2c_1y_1$	$c_2\overline{c_1}y_2$	$c_2c_1y_3$	
1	2		3	4	5	6
d_1	0 1	0 1 1	1 1 0	0 0 1	0 1 1	$y_1 < y_2, y_1 < y_3$
d_2	1 0	1 0 1	0 0 1	1 1 0	1 0 1	$y_2 < y_1, y_2 < y_3$
d_3	1 1	1 1 0	0 1 1	1 0 1	1 1 0	$y_3 < y_1, y_3 < y_2$

The table contains the test patterns in column 2, the fault table in columns 3-5, and the high-level constraints (5) satisfied by generating test data in column 6. The detected gate-level faults in the fault table are highlighted by red colour: 0 means the value of a signal which activates the fault SAF/1. For example, in case of the fault $c_2 \equiv 1$ in column 5, the value of the output signal $y = y1 = 0$ (selected by the control signals $\overline{c}_2c_1 = 11$) will change from 0 to $y = y_1 \vee y_3 = 1$ (both functions f_1 and f_3, are simultaneously selected). For detecting the faults SAF/0, more 3 test patterns are needed (not shown in the table). We see in the fault table that the faults $c_1 \equiv 1$ in column 3 and $c_2 \equiv 1$ in column 4 are not detected, because of the control code $c_2c_1 = 00$ is illegal (not usable in this MUT). According to Corollary 1, these gate-level faults are redundant (in case if the control circuit is implemented as DNF). As the example shows, the redundancy of the gate-level faults can be derived by simple low-level SAF simulation.

Note, Theorem 2 and Corollary 4 in appendix 2 were formulated and the proofs were given, considering only the single SAF model. In fact, the power of the proposed high-level control fault model stretches far beyond the fault class of single SAF, as it was shown in Section 4.

In the following, the general procedure of the mixed level identification of redundant gate-level SAF is shown, where the test is generated at the high-level using the proposed high-level control fault model, and the low-level redundancy

Procedure 1

1. Generation of the high-level test $T(F)$ for the given set F of functions (instructions), with finding the data which satisfy the constraints (4,5).

2. Generation of the high-level fault coverage table $R = \|r_{ijk}\|$ by high-level fault simulation of the test $T(F)$ [32].
3. High-level fault redundancy identification. For all not covered high-level faults, the redundancy of the high-level control faults is identified.
4. If the high-level redundancy cannot be proven for some of high-level faults, the test $T(F)$ must be extended to satisfy the constraints (4,5), and to achieve 100% high-level fault coverage. This is the prerequisite for the next step of redundant SAF identification.
5. Gate-level fault simulation of the test $T(F)$. The not detected SAF are identified as redundant low level faults in the control circuit of the MUT.

8 Case Study of the Control Part Test for Forwarding Unit in MiniMIPS RISC Processor

The proposed high-level functional control part testing approach was discussed on the example of the group of functions related to the execute unit and ALU example. In this case, the set of function F was derived directly from the subset of instructions of the processor. Consider now a case, where the set of functions /(F/) to be tested is representing not the main functional properties of the processor described by the instruction set, but other non-functional properties like performance, for which for example the forwarding unit is responsible.

Consider the forwarding unit as the high-level circuit in Fig.6, which consists of the pipeline registers ID/EX, EX/MEM and MEM/WB [38], forwarding control unit, multiplexer for selecting the data from pipeline registers, and ALU with observable output node. The role of the control unit is to compare the addresses r_s used in the current instruction with addresses $r_d(EX)$, $r_d(MEM)$ stored in the respective pipeline registers, and to produce the control signals c_1, c_2 and $c*$ for selecting the data D_1, D_2, and D_3, respectively, from different pipeline registers. The data D_i represent the values of the functions $f_i \in F$, where F represents the functionality of the forwarding unit.

The test patterns, generated for testing the forwarding function in Fig.6 are depicted in Table 3. The test consists of 3 groups of patterns (in each 2 patterns, framed in bold): forward from EX/MEM (c_1, D_1), MEM/WB (c_2, D_2), and no forward (c_*, D_3). Each of the 6 test patterns in Table 4 are applied in a cycle for as many test data needed to solve the constraints (5).

For solving the constraints (5) like $r_d(EX) < r_d(MEM)$, the data for the register numbers to produce or not to produce hazards, is generated using pseudo-exhaustive test data for comparators. To organize parallel testing (solving the constraints (5)) for all the bits of the data words, the patterns "all 1s" and "all 0s" are used. Differently, from testing the execute unit (in the previous sections), where the constraints were solved by data of the same time-frame (for all of instructions), in case of forwarding unit the constraints

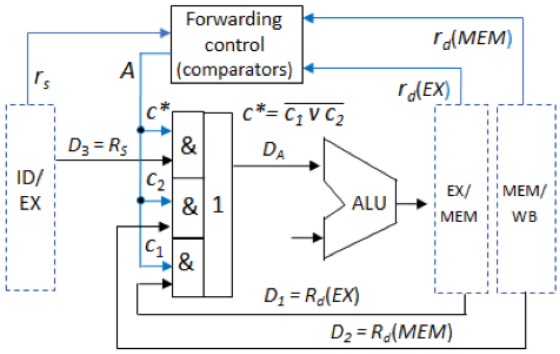


Fig. 6: Example of testing the pipeline forwarding unit

Table 3: Test generation for pipeline forwarding unit

Var	The values of var(hazard detection condition)	Test Pattern					
		1	2	3	4	5	6
c_1	$rd(EX) = rs$	1	1	0	0	0	0
$f_1 = D_1$	$R_d(EX)$	0	1	1	0	1	0
c_2	$rd(MEM) = r_s$	0	0	1	1	0	0
$f_2 = D_2$	$R_d(MEM)$	1	0	0	1	1	0
c^*	$\overline{c_1 \vee c_2} = 1$	0	0	0	0	1	1
$f_3 = D_3$	R_s	1	0	1	0	0	1

are solved by data from different time frames, and locating in different pipeline registers. For this purposes, relevant test templates were produced.

As an example, Fig.7 demonstrates two test programs generated for testing the hazard detection and data forwarding functions. In case of hazards in addresses of registers ($r_a = c_b$), the data from the register r_b is forwarded from EX/MEM stage (for Fig.7a), and from MEM/WB (for Fig.7b). The cases correspond to test patterns 1-2, and 3-4, respectively, in Table 3. For the 1st patterns the values "all 0s" are forwarded, and for the 2nd patterns "all 1s" are forwarded. Two tests are needed for satisfying mutually the constraints (5): $rd(EX) < rd(MEM)$ and $rd(MEM) < rd(EX)$.

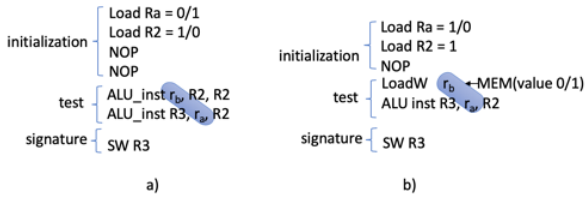


Fig. 7: Examples of testing the pipeline forwarding unit

The red entries in Table 3 correspond to the signals where the SAF faults are sensitized and can be detected. The test patterns 1,3 and 5 are created for testing the control faults where the constraint (5) is satisfied. These patterns also test the SAF/1 faults in the data part. The test patterns 2,4,6, satisfying the constraint (4), are testing the SAF/0 faults in both of the control and data parts.

9 Generation and Organization of the Complete Test Program

Our test program generation is divided into two parts. The first part involves automated high-level test data generation. While the second part uses the data generated by the first step together with manually created test template to automatically generate the test programs. The flow of both parts is described in Fig.8 and 9. The high-level ATPG works according to the two algorithms (random and greedy) developed in [31]

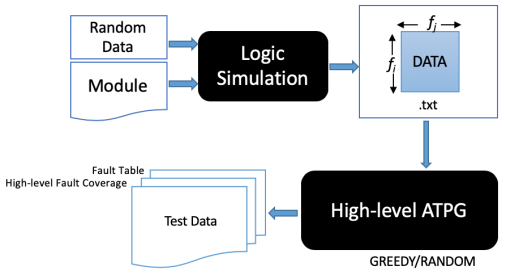


Fig. 8: High-level Test Data Generation

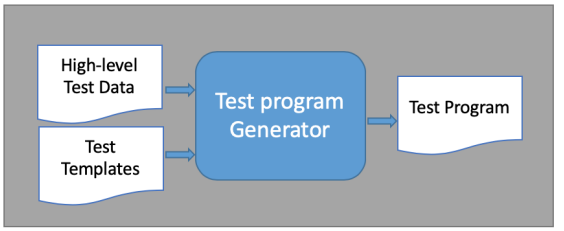


Fig. 9: High-level Test Generation

In accordance with the described method of generating test data for testing processor modules, we organize the test according to the architecture shown in

Fig.10. The full test is divided into sections, where the target of each section is to test a subset of functions F, and for testing each function $f_i \in F$, the same test template is used.

The full test has a structure of three embedded loops. The first outer loop consists of execution of the test sections using the same template. The number of loops is equal to the number of different test templates, which represent a subroutine with a uniform structure consisting of three parts: initialization of the processor, execution of the instruction under test which targets a function $f_i \in F$, and propagating the response to the node of observation. The second middle loop consists of repeating the selected test template for all functions $f_i \in F$ over a list of related instructions I_i to be inserted into the current template. Each instruction pattern $I_i = (\text{opcode}_i, A_1, A_2)$ refers to the data operands d_1, d_2 according to addresses A_1, A_2, respectively (the number of data operands is optional and is determined by the test template).

For each instruction $I_i \in F$ under test, two consecutive inner loops will be carried out: for testing the control part and for testing the data part. The number of test data $d = (d_1, d_2) \in D_i^*$ for testing the control part is found by the method described in this paper, whereas the number of test patterns for testing the data part is determined by the length of the pseudo-random test sequence, derived, for example by the methods considered in [31, 32].

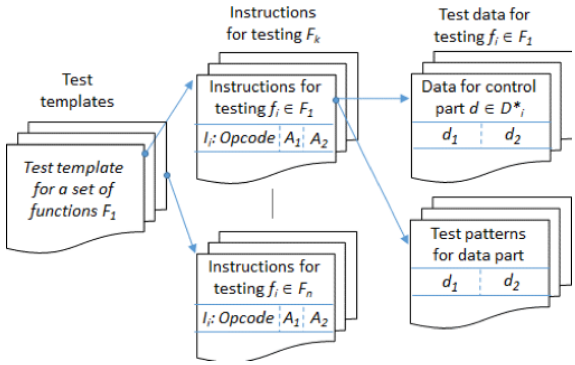


Fig. 10: Architecture of the test program

The originality of the proposed test strategy stands in on-line test generation based on unrolling on the fly the stored in compact way of all needed test information in the form of the sets of test program templates, test instructions and related test data lists.

10 Experimental Results

We carried out experiments on Intel Core i7 processor at 3.4GHz and 8GB of RAM. The target was to investigate the efficiency of the new high-level implementation-independent SBST generation method for microprocessors by measuring gate-level SAF coverage (FC) for comparison purposes with other methods. The objectives of experiments were ALU unit of VLIW [37] and different units of MiniMIPS [38], like execute, forwarding and branch control sub-circuits.

Using the subset of 16 VLIW instructions [37], we investigated the speed of random search of test data for testing the control part of the execute unit. We investigated two search algorithms pure random and optimized greedy random approach. The results are depicted in Table 4.

Table 4: Comparison of control test algorithms

Method	Test-length, #instructions	Fault Coverage		Time (s)
		High-level faults	SAF	
RANDOM	204	100%	99.34%	2.00
GREEDY	139	100%	99.34%	7.85

The execute unit of MiniMIPS [38] consists of Adder and 2 multiplication modules MULT0 and MULT1 (Table 5). We targeted 25 instructions out of MiniMIPS 51 instructions, as the basis of the set of functions $F = \{f_i\}$ to be tested. The high-level test was simulated by commercial tool to grade the gate-level SAF coverage. To evaluate the efficiency of the high-level ATPG, we also used commercial gate-level ATPG for comparison. The time cost for high-level ATPG is about two orders of magnitude less than that of the commercial ATPG. The gate-level SAF coverage achieved by the proposed ATPG for the whole module under test, is better than that achieved by the commercial tool.

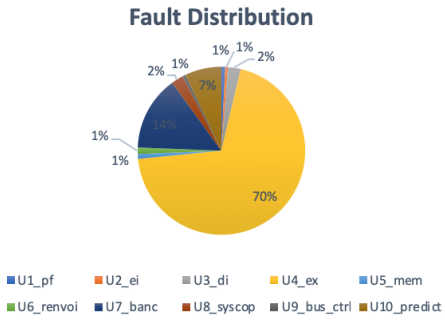


Fig. 11: miniMIPS processor Fault Distribution

Fig. 11 shows the distribution of faults in the microprocessor according to modules. For example, the execute unit takes 70% of the total number of faults in the processor. It consists of the adder, two multiplication modules(MULT0 and MULT1) and interconnections. Note that the modules correspond here to column 2 in Table 8

Table 5: Execute Unit Test

Method	Experiments		#Faults	FC(%)	Stored Patterns	Executed Patterns	ATPG Time
Proposed high-level method	High-level ATPG		756	100	166	4818	47s
	Gate-level Simulation	Adder	2516	99.92			
		MULT0	95188	99.52			
		MULT1	91810	99.16			
Commercial gate-level ATPG		Adder	2516	99.96	957	957	8h 27min
		MULT0	95188	97.40			
		MULT1	91810	97.71			

In Table 6, FC and simulation times are given for the forwarding unit (FU). First, when applying only the ALU test, then the dedicated test for only FU, and thereafter, combining both tests. The tests for FU were generated without knowing gate-level implementation detail. We relied only on general information of the MiniMIPS pipeline architecture, which includes the number of stages and forwarding paths.

Table 6: Fault coverage of forwarding unit by different tests

Module/Unit	ALU Test(%)	Forwarding Test(%)	Combined(%)	Improvement(%)
Forwarding Unit	89.71	97.84	98.03	8.32
Time(s)	808	48	460	

In Table 7, we compare our results for 3 different MiniMIPS modules (ALU, PPS EX(Execute Unit), and Forwarding Unit) with 3 other test generators. Our approach is similar to [28] in sense that gate-level implementation details are not required, but it shows almost 5% improvement in FC compared to [28].

Table 7: Targeted modules comparison with other methods

Module/ unit	#faults	Gate-level implementation details are exploited		Gate-level implementation independent	
		ATIG [39]	SBST [40]	SBST [28]	Proposed
ALU	203576	98.67%	n.a	97.85%	99.06%
PPS_EX	211136	97.62%	96.20%	84.12%	98.37%
Forwarding Module	3738	99.00%	99.68%	93.64%	98.03%
Register Banc	43584	99.90%	100%	99.98%	99.99%
Syscop	6930	93.60%	98.04%	87.90%	87.65%

Although the method in [39] shows 1% improvement over the proposed method, it requires implementation details. Method in [40] requires enforcing set of constraints during ATPG test generation, requiring also gate-level information. Here, the result of our method shown for the forwarding unit and the system co-processor doesn't consider the untestable faults in the modules. In

Table 8, we show the results of these units when the proven untestable faults have been removed from the fault list.

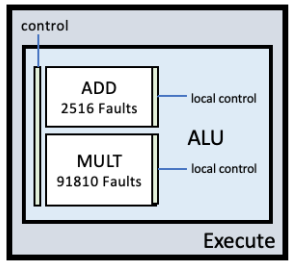


Fig. 12: Simplified structure of the Execute Unit of MiniMIPS RISC Processor

One of the goals of the experiment in Table 5 was to demonstrate the feasibility of identification of the gate-level SAF redundancies by high-level test generation. We demonstrated it on the basis of ALU test for MiniMIPS processor. The SAF coverage 99.92%, achieved by 100% high-level fault coverage test means that 2 faults in ALU remained not detected, and are qualified according to Corollary 4 as redundant. Thus, establishing 100% fault efficiency. Since by fault simulation of the test for ALU (without its local control part) we found 100% SAF coverage, we can conclude that the 2 faults belong to the ALU control part. On the other hand, since low-level ATPG found 1 undetected fault in the ALU joint data/control circuit, we can conclude that this redundant fault belongs to the ALU local control part, and the second redundant fault belongs to the ALU global control part (see Fig.12).

In the MULT block, fault coverage of 99.09% refers to 835 not covered faults, which should be qualified according to Corollary 4 as redundant. By gate-level ATPG we found that from the 1256 not covered by ATPG faults, 865 were ATPG untestable, 105 were classified as redundant, and 286 remained not detected. From the latter it follows, that the 444 faults (the difference 1256 – 835), not covered by gate-level ATPG, however, were covered by the high-level ATPG. These faults should belong to the class of gate-level ATPG untestable faults.

It is important to mention that as opposed to the compared state-of-the-art methods, where only single SAF coverage has been the measurable target, the proposed method covers extended classes of low-level faults, including conditional and multiple SAF. This was proved theoretically with regards to the control faults in this paper. For the faults in the data parts, the same broader fault coverage results from the nature of exhaustiveness of test patterns.

In Table 8, we compared our approach with state-of-the-arts methods in terms of test for the full processor. Using the information about untestable faults in [40], we removed the identified and proven untestable faults from the fault list. The result shows that our approach covers more faults in the

Table 8: Fault coverage of whole processor by different tests

	Module/Unit	ATIG(%) [39]	SBST(%) [40]	SBST(%) [28]	Proposed Method(%)
Without Prediction	U1_pf	98.32	91.97	86.32	70.00
	U2_ei	99.71	96.82	90.86	85.50
	U3_di	95.28	92.45	90.24	89.70
	U4_ex	97.62	96.20	97.85	98.68
	U5_mem	83.41	71.29	81.87	90.63
	U6_renvoi	99.00	99.68	93.64	98.50
	U7_banc	99.90	100	99.98	99.99
	U8_syscop	93.60	98.04	87.90	93.53
	U9_bus_ctrl	92.62	92.20	93.95	89.78
	Total	97.52	97.46	95.08	98.03
With Prediction	U10_predict	–	–	–	59.19
	Total	97.31	97.46	95.08	95.30

processor with no knowledge of the implementation details. However, we still have some faults undetected in the modules. For example, the fault coverage for the forwarding unit(U6_renvoi) with respect to the testable faults in the module stood at about 98.50%. The reason for the undetected 1.50% faults is due to signals such as the interrupt signal and exceptions that can not be activated using the particular functional method, where only forwarding functions were taken into account. [42] identified 39 of the total fault list for the forwarding unit to belong to this category of undetectable faults.

Our approach doesn't target the branch prediction unit, since faults in the unit do not lead to any functional incorrectness but performance cost which is usually two or more clock cycles depending on the architecture [41]. Therefore, branch prediction units are hard to test by functional methods without having dedicated observation points and the same applies to our approach. The approach used for testing the branch prediction unit in [40] is based on using the algorithm mostly used in memory testing. We can easily adopt this to our test as it was explained in Section 4. To have a fair comparison, we have decided to exclude in Table 8, the faults related to the branch prediction unit since the result in [28] is also based on implementation of miniMIPS without this unit.

11 Conclusions

A new high-level and implementation-independent test program generation method for modules of RISC processors is proposed with improved quality compared to the known methods. The higher quality is achieved through widening the fault class covered.

The proposed method is based on introducing a novel high-level functional control fault model for testing the control parts of the functional modules of processors, and combining it with implementation-independent and fault model free testing of data parts of the modules under test (MUT) with pseudo-exhaustive test patterns. The high quality of test programs was proven theoretically, and the proof was supported by experimental results where the

quality of tests was measured regarding the SAF class. It was proven that the proposed high-level control fault model covers as added value on one hand a larger class of structural faults, including conditional SAF, shorts, multiple SAF without the need of listing these faults explicitly, and on the other hand also functional faults of decoding circuits. A similarity was shown between the sequential March test used for memory testing, and combinational test for decoding circuits in logic modules of processors.

A high-level fault coverage metric and a method for high-level fault simulation were developed to support the proposed test generation approach, which made it possible to develop a novel mixed-level method for identification of high-level functional redundancy of faults, and low-level structural redundancy of gate-level faults.

The novelties of the proposed approach are based on two new ideas. We introduced a method for modeling of MUT by symbolic EDNF, to be able to partition the circuit into the control and data parts without knowing its implementation details. We also translated the classical low-level fault propagation task into the functional problem of solving high-level data constraints, to achieve independence of implementation details for high-level test generation.

There are two side-effects of the discussed novelties of the paper as follows. First, due to EDNF based control model, the generated test may be over dimensioned. Second, the novel idea of data constraints based test generation may tend to produce more test patterns than it would be needed for only single SAF detection. However, both aspects, which can be considered as a negative factors regarding the class of SAF, on the other hand, will help to cover larger fault classes, including multiple faults. Therefore, the proposed method is more powerful than traditional ATPGs, which target single SAF.

Another added value of the proposed approach was discovered during experimental research in test data generation for RISC processors. The test programs generated explicitly for testing only the control parts of modules achieve as well very high fault coverage for data part. This is due to the power of novel data constraints proposed for selecting data operands.

To sum up, the proposed method is an important step ahead compared to state-of-the-art for providing trustworthy and safe information about the quality of test programs generated. The future work will be in extending the proposed method for using it for other types of more complex processors, including investigations on different test optimization ideas.

Acknowledgements The work has been supported by EU's H2020 project RESCUE, Estonian research grant IUT 19-1, and funded by Excellence Centre EXCITE in Estonia.

References

1. L.-T.Wang, Ch.-W.Wu, X.Wen. VLSI Test Principles and Architectures. Design for Testability. Elsewier, 2006, 777 p.

2. P. Georgiou, X. Kavousianos, R. Cantoro and M. S. Reorda, "Fault-Independent Test-Generation for Software-Based Self-Testing," Proc. 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, 2018, pp. 79-84.
3. R. Ubar, S. Kostin and J. Raik, "How to Prove that a Circuit is Fault-Free?," Proc. 2012 15th Euromicro Conference on Digital System Design, Izmir, 2012, pp. 427-430.
4. Zhuo Li, Xiang Lu, Wangqi Qiu, Weiping Shi and D. M. H. Walker, "A circuit level fault model for resistive opens and bridges," Proc. 21st VLSI Test Symposium, 2003., Napa, CA, USA, 2003, pp. 379-384.
5. H. K. Lee and D. S. Ha, "SOPRANO: an efficient automatic test pattern generator for stuck-open faults in CMOS combinational circuits," Proc. 27th ACM/IEEE Design Automation Conference, Orlando, FL, USA, 1990, pp. 660-666.
6. A.Kristic, K.T.Cheng. Delay Fault Testing for VLSI Circuits. Dordrecht, The Netherlands, Kluwer Academic Publishers, Oct. 1998.
7. J.P.Roth. Diagnosis of Automata Failures: A Calculus and a method. IBM J. Res. Develop., Vol. 10, No. 4, pp. 278-291, July 1966.
8. R.D.Blanton, J.P.Hayes. On the Properties of the Input Pattern Fault Model. ACM Trans. Des. Automat. Electron. Syst., Vol. 8, No. 1, pp. 108-124, Jan. 2003.
9. K.B.Keller. Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms. US Patent 5546408, Aug. 13, 1994.
10. R.Ubar. Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations. Automation and Remote Control, Vol.40, No 11, part 2, Nov. 1980, Plenum Publishing Corporation, USA, pp. 1693-1703.
11. J. Raik, R. Ubar, J. Sudbrock, W. Kuzmicz and W. Pleskacz, "DOT: new deterministic defect-oriented ATPG tool," Proc. European Test Symposium (ETS'05), Tallinn, Estonia, 2005, pp. 96-101.
12. U. Mahlstedt, J. Alt and I. Hollenbeck, "Deterministic test generation for non-classical faults on the gate level," Proc. Fourth Asian Test Symposium, Bangalore, India, 1995, pp. 244-251.
13. S. Holst and H. Wunderlich, "Adaptive Debug and Diagnosis without Fault Dictionaries," Proc. 12th IEEE European Test Symposium (ETS'07), Freiburg, 2007, pp. 7-12.
14. Kyoung Youn Cho, S. Mitra and E. J. McCluskey, "Gate exhaustive testing," Proc. IEEE International Conference on Test, 2005., Austin, TX, 2005, pp. 7 pp.-777.
15. A. Jas, S. Natarajan and S. Patil, "The Region-Exhaustive Fault Model," Proc. 16th Asian Test Symposium (ATS 2007), Beijing, 2007, pp. 13-18.
16. K. N. Dwarakanath and R. D. Blanton, "Universal fault simulation using fault tuples," Proc. 37th Design Automation Conference, Los Angeles, CA, USA, 2000, pp. 786-789.
17. M. Psarakis, D. Gizopoulos, A. Paschalis and Y. Zorian, "Sequential fault modeling and test pattern generation for CMOS iterative logic arrays," in IEEE Transactions on Computers, vol. 49, no. 10, pp. 1083-1099, Oct. 2000.
18. F. Hapke et al., "Cell-Aware Test," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 9, pp. 1396-1409, Sept. 2014.
19. L. Shen and S. Su, "A Functional Testing Method for Microprocessors," in IEEE Transactions on Computers, vol. 37, no. 10, pp. 1288-1293, 1988.
20. F. Corno, G. Cumani, M. Sonza Reorda and G. Squillero, "An RT-level fault model with high gate level correlation," Proc. IEEE International High-Level Design Validation and Test Workshop (Cat. No.PR00786), Berkeley, CA, USA, 2000, pp. 3-8.
21. Chang Hyun Cho and J. R. Armstrong, "B-algorithm: a behavioral test generation algorithm," Proc., International Test Conference, Washington, DC, USA, 1994, pp. 968-979.
22. V. Kumar et al. Employing Functional Analysis to Study Fault Models in VHDL. Int. J. of Scientific Engineering and Technology, vol.1, no.5, 2012, pp.2017-208.
23. M. Bushnell and V. Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Springer, 2013, 690 p.
24. P. A. Thaker, V. D. Agrawal and M. E. Zaghloul, "Register-transfer level fault modeling and test evaluation techniques for VLSI circuits," Proc. International Test Conference 2000 (IEEE Cat. No.00CH37159), Atlantic City, NJ, USA, 2000, pp. 940-949.

25. N. Kranitis, A. Paschalis, D. Gizopoulos and G. Xenoulis, "Software-based self-testing of embedded processors," in IEEE Transactions on Computers, vol. 54, no. 4, pp. 461-475, April 2005.
26. C. H. -. Wen, L. C. Wang and Kwang-Ting Cheng, "Simulation-based functional test generation for embedded processors," Proc. Tenth IEEE International High-Level Design Validation and Test Workshop, 2005., Napa Valley, CA, 2005, pp. 3-10.
27. F. Corno, E. Sanchez, M. S. Reorda and G. Squillero, "Automatic test program generation: a case study," in IEEE Design & Test of Computers, vol. 21, no. 2, pp. 102-109, March-April 2004.
28. D. Gizopoulos et al., "Systematic Software-Based Self-Test for Pipelined Processors," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 11, pp. 1441-1453, Nov. 2008.
29. P. Bernardi et al., "On the in-field functional testing of decode units in pipelined RISC processors," Proc. 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam, 2014, pp. 299-304.
30. E. Sanchez and M. S. Reorda, "On the Functional Test of Branch Prediction Units," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 9, pp. 1675-1688, Sept. 2015.
31. A. S. Oyeniran, R. Ubar, S. P. Azad and J. Raik, "High-level test generation for processing elements in many-core systems," Proc. 2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), Madrid, 2017, pp. 1-8.
32. A. S. Oyeniran, S. P. Azad and R. Ubar, "Parallel Pseudo-Exhaustive Testing of Array Multipliers with Data-Controlled Segmentation," Proc. 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, 2018, pp. 1-5.
33. A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy and J. Raik, "Mixed-level identification of fault redundancy in microprocessors," Proc. 2019 IEEE Latin American Test Symposium (LATS), Santiago, Chile, 2019, pp. 1-6
34. D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," in IEEE Transactions on Electronic Computers, vol. EC-15, no. 1, pp. 66-73, Feb. 1966.
35. A. J. van de Goor, "Testing Semiconductor memories. Theory and practice," Wiley, 1991, 512p
36. S. M. Thatte, J. A. Abraham, "Test Generation for Microprocessors," in IEEE Transactions on Computers, vol. C-29, no. 6, pp. 429-441, June 1980.
37. M.Schölzel. Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis. Brandenburg University of Technology Cottbus-Seftenberg, 2015.
38. OpenCores, "MiniMIPS ISA".
39. 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, vol. 21, no. 7, pp. 1220-1233, July 2013.
40. A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda and B. Becker, "On the automatic generation of SBST test programs for in-field test," Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, 2015, pp. 1186-1191.
41. P. Bernardi, L. Ciganda, M. Grosso, E. Sanchez, M. Sonza Reorda, "A SBST strategy to test microprocessors' Branch Target Buffer," Proc. 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS),2012,pp.306-311
42. P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. S. Reorda, M. Grosso, O. Ballan, "On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors," Proc. 14th International Workshop on Microprocessor Test and Verification,2013,pp.52-57
43. A. S. Oyeniran, R. Ubar, M. Jenihhin, C. C. Gürsoy and J. Raik, "High-Level Combined Deterministic and Pseudo-exhuastive Test Generation for RISC Processors," 2019 IEEE European Test Symposium (ETS), Baden-Baden, Germany, 2019, pp. 1-6.

Appendix A
Theoretical Foundation of The Proposed High-level Control Fault Model

Consider a set $F = \{f_i\}$ of functions, selected as a group of functions to be tested using the fault model of constraints (4,5).

Let us define for each function a set of patterns

$$D^0\left\{f_{i/k}\right\} = \left\{d | f_{i/k}(d) = 0\right\}$$

for which the function f_i produces the value $y_{i/k} = 0$ in the bit k. Let us call the set of patterns $D^0\left(f_{i/k}\right)$ as the 0-domain of the function f_i for the bit k.

Theorem 1 *If for two functions $f_{i/k}$ and $f_{j/k}$ the following $D^0\left\{f_{i/k}\right\} - D^0\left\{f_{j/k}\right\} = \varnothing$ is valid, then the fault $r_{i,j/k}$ is functionally redundant.*

Proof From $D^0\left\{f_{i/k}\right\} - D^0\left\{f_{j/k}\right\} = \varnothing$ it follows that $D^0\left\{f_{i/k}\right\} \subset D^0\left\{f_{j/k}\right\}$. This means that for each data pattern $d \in D^0\left(f_{i/k}\right)$, we have $y_{i/k}(d) = 0$, but also $y_{j/k}(d) = 0$. Hence, there exists no data which can satisfy the constraint (5), and the functional fault $r_{i,j/k}$ according to Definition 2 is not testable, and is functionally redundant.

From Theorem 1, the following corollaries straightforwardly result.

Corollary 1 *If $D^0\left\{f_{i/k}\right\} - D^0\left\{f_{j/k}\right\} = \varnothing$then the difference $D^0\left\{f_{i/k}\right\} = D^0\left\{f_{j/k}\right\}$ is the set of data patterns which can be used for testing the fault $r_{i,j/k}$.*

Corollary 2 *If for all bits k, $D^0\left(f_{i/k}\right) - \bigcup_{j,j\neq i} D^0\left(f_{j/k}\right) \neq \varnothing$ is valid, then the function $f_{i/k}$ has unique patterns, which produce $y_{i/k} = 0$ for only the function $f_{i/k}$ and for no one else $f_{j/k}$, $f_j \in F$. Each of these patterns serves as test data for detecting the faults $r_{i,j/k}$ for all $f_j \in F$, $j \neq i$.*

Corollary 3 *If for all bits k $D^0\left\{f_{i/k}\right\} - D^0\left\{f_{j/k}\right\} = \varnothing$ is valid, then the function f_i is fully distinguishable from the function f_j in accordance with the constraints (5), and the functional faults $r_{i,j/k}$ are testable for all bits of k. If all functions $f_j \in F$ are mutually distinguishable then the set all high-level control faults in $M(F)$ are testable.*

In example 3, we will demonstrate the testability of selected ALU functions in a microprocessor using the proposed high-level control fault model.

Example 3 Consider, as an example, a set of three 1-bit functions $f_1(OR)$, $f_2(AND)$, and $f_3(XOR)$. The truth table of these functions f_i and the Venn diagrams of the sets $D^0\{f_i\}$ are illustrated in Fig.13. According to Venn diagrams, we can notice that there exist the following domains $D^0(f_i)$: $D^0(OR) = \{00\}$, $D^0(AND) = \{00, 01, 10\}$, $D^0(XOR) = \{00, 11\}$.

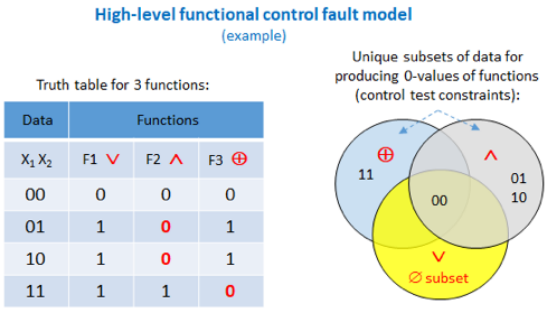


Fig. 13: Three functions selected for testing as MUT

Few examples which illustrate the testability of functions OR, AND, XOR according to Corollaries 1-3, are depicted in Table 1.

Table 9: Examples of relations between 0-domains

No	Relations for $D^0(f_i)$	Patterns	Comments
1	$D^0(AND) - D^0(XOR)$	$\{01, 10\}$	AND is distinguishable from XOR
2	$D^0(XOR) - D^0(AND)$	$\{11\}$	XOR is distinguishable from AND
3	$D^0(OR) - D^0(AND)$	$\varnothing$	OR is not distinguishable
4	$D^0(OR) - D^0(XOR)$	$\varnothing$	
5	$D^0(AND) - \left(D^0(OR) \cup D^0(XOR)\right)$	$\{01, 10\}$	Unique patterns for AND
6	$D^0(OR) - \left(D^0(AND) \cup D^0(XOR)\right)$	$\varnothing$	No unique patterns for OR

Appendix B
Theoretical Foundations for Identification of Not-detectable Low-level Faults in the Control Circuits

In the following we provide the main statements and proofs, which justify the identification of the not-detectable (functionally redundant) low-level faults in the real implementation of the given control circuit by its low-level fault simulation using the test T(F) constructed in Section 5.

Theorem 2 *The test $T(F)$, which covers all non-redundant high-level faults of the fault model $M\{F\}$, covers also all gate-level testable SAF in the control part of the microprocessor, which controls the set of functions F.*

Proof Consider the control part of a MUT presented at high-level in Fig.3, and described in the real implemented version expanded as the following EDNF as explained in [34], and also in Section 3::

$$y = c_{1,1}c_{1,2}\ldots c_{1,p}y_1 \vee c_{2,1}c_{2,2}\ldots c_{2,p}y_2 \vee \ldots \vee c_{n,1}c_{n,2}\ldots c_{n,p}y_n \qquad (6)$$

Here, the high-level control states $s_i \in S$ used in the formula (1) are mapped into the control signals $c_{i,1}c_{i,2}\ldots c_{i,p}$ of the decoder of the operation code embedded in the instruction of the processor. In [34] and Section 3, it was shown that if all non-redundant appearances of the literals in the EDNF will be tested, then all the non-redundant faults in the original circuit implementation will also be tested. Let us show now, that the high-level test generated for the high-level expression (1) will also test all non-redundant faults in the expression (6).

In the EDNF (6), the variables $c_{i,j}$ for selecting the data results $y_i, i = 1,\ldots n$, represent the global control signals $c_{i,j} = 1,\ldots p$, which may be either inverted or not, and which cover, in general case, exhaustively all 2^p combinations. In the EDNF, due to satisfied constraints (5) of the fault model in Definition 1, when testing the function y_i , at least once the value of y_j for each $i \neq j$ will be $y_j = 1$. On the other hand, due to the exhaustiveness of all 2^p combinations of control signals, for each term of EDNF with $y_i = 1$, there will be a combination of control signals $c_{i,1}c_{i,2}\ldots c_{i,p}$ consisting of a single 0, e.g. $c_{i,p} = 0$, with all others $c_{i,r} = 1, r \neq p$. This is the case, where in the term $c_{i,1}c_{i,2}...c_{i,p}y_i$, the SAF $c_{i,p} \equiv 1$ is tested. For propagating the fault $c_{i,p} \equiv 1$ to the output y, all other terms in DNF must have at least one 0 assigned to the variables of the term. This is guaranteed, because due to the constraints (5), which demand that in the term where all $c_{j,k} = 1$, the value of y_j must be 0, and in other terms there must be at least one variable assigned by 0. Hence, all SAF faults of type $c_{i,p} \equiv 1$ in all variables $c_{i,p}$ can be tested by $T(F)$. The faults $c_{i,p} \equiv 0$ are tested by test data used in $T(F)$ where the constraint (1) is satisfied. ■

Corollary 4 *Any gate-level SAF in the control part related to $F = \{f_i\}$, not detectable by the test $T(F)$ which covers all not redundant high-level control faults of the model $M\{F\}$, is redundant.*

Proof In Theorem 2, exhaustiveness of using all the combinations of the local control signals $c_{i,1}c_{i,2}\ldots c_{i,p}$ was assumed. If not all combinations are used in the instruction set of the microprocessor, which is the typical practical case, then, not all patterns can be generated for activating all SAF of type $c_{i,p} \equiv 1$. Usually these cases are used for optimization of the gate-level structure of the control part of ALU. If however the optimization process has not removed all hardware redundancy, then as the result, the control part may consequently contain also redundant faults. These redundant faults can be identified by simple and fast gate-level fault simulation of the high-level generated test T.

Curriculum Vitae

1. Personal data

Name	Adeboye Stephen Oyeniran
Date and place of birth	30 April 1983 Lagos, Nigeria
Nationality	Nigerian

2. Contact information

Address	Tallinn University of Technology, School of Information Technologies, Department of Computer Systems, Ehitajate tee 5, 19086 Tallinn, Estonia
Phone	+372 58337792
E-mail	adeboye.oyeniran@taltech.ee, aysteph3@yahoo.com

3. Education

2015–2020	Tallinn University of Technology, School of Information Technologies, Computer and Systems Engineering, PhD studies
2013–2015	Tallinn University of Technology, Faculty of Information Technologies, Computer and Systems Engineering, MSc *cum laude*
2003–2008	University of Agriculture, Abeokuta, Faculty of Natural Science, Computer Science, BSc

4. Language competence

Yoruba	native
English	fluent

5. Professional employment

2018– . . .	Tallinn University of Technology, Early Stage Researcher
2014–2016	Arvato Systems, Technical Specialist
2009–2013	Firstlogic IT Solutions, IT Facilitator/Software Developer

6. Computer skills

- Operating systems: Unix, Windows
- Document preparation: Ms-Word, LaTeX
- Programming languages: VHDL, Java, Python, R, SQL, C

7. Honours and awards

- 2015, **Merit Award**, Graduating Cum Laude from Faculty of Information Technology.
- 2016, **Best Paper Award**, IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), Cluj-Napoca, Romania.

8. Defended theses

- 2015, Double phase fault collapsing with linear complexity in digital systems, MSc, supervisor Prof. Raimund Ubar, Tallinn University of Technology, Institute of Information Technologies

Elulookirjeldus

1. Isikuandmed

Nimi	Adeboye Stephen Oyeniran
Sünniaeg ja -koht	30.04.1983, Lagos, Nigeeria
Kodakondsus	Nigeeria

2. Kontaktandmed

Aadress	Tallinna Tehnikaülikool, arvutisüsteemide Instituut, Ehitajate tee 5, 19086 Tallinn, Estonia
Telefon	+372 58337792
E-post	adeboye.oyeniran@taltech.ee, aysteph3@yahoo.com

3. Haridus

2015–2020	Tallinna Tehnikaülikool, infotechnoloogia teaduskond, arvutisüsteemid, doktoriõpe
2013–2015	Tallinna Tehnikaülikool, infotechnoloogia teaduskond, arvutisüsteemid, MSc *cum laude*

4. Keelteoskus

yoruba keel	emakeel
inglise keel	kõrgtase

5. Teenistuskäik

2018– . . .	Tallinna Tehnikaülikool, Varajaste teadur
2014– 2016	Arvato Systems, Tehniline spetsialist
2009–2013	Firstlogic IT Solutions, IT Juhendaja/Tarkvara arendaja

6. Arvutioskus

- Operatsioonisüsteemid: Unix, Windows
- Kontoritarkvara: Ms-Word LaTeX
- Programmeerimiskeeled: VHDL, Java, Python, R, SQL, C

7. Kaitstud lõputööd

- 2015, Kahefaasiline lineaarse keerukusega lagoritm rikete kollapseerimiseks digitaalskeemides, MSc, juhendaja Prof. Raimund Ubar, Tallinna Tehnikaülikool, arvutisüsteemide Instituut