TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
51/2018

Software-Based Self-Test for Microprocessors with High-Level Decision Diagrams

ARTJOM JASNETSKI

TTÜ PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems
This dissertation was accepted for the defence of the degree 10/07/2018

Supervisor: Prof. Raimund-Johannes Ubar
Department of Computer Systems
Tallinn University of Technology
Tallinn, Estonia

Co-supervisor: Dr. Anton Tsertov
Department of Computer Systems
Tallinn University of Technology
Tallinn, Estonia

Opponents: Prof. Heinrich-Theodor Vierhaus
Department of Computer Science
Brandenburg University of Technology
Cottbus, Germany

Prof. Anzhela Matrosova
Department of Programming
Tomsk State University
Tomsk, Russia

Defence of the thesis: 04/09/2018, 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 previously submitted for doctoral or equivalent academic degree.

ARTJOM JASNETSKI **ARTJOM JASNETSKI**

---
signature

European Union
European Social Fund
Investing in your future

ISSN 2585-6898 (publication)
ISBN 978-9949-83-312-2 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-313-9 (PDF)

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
51/2018

Mikroprotsessorite tarkvara-põhine enesetestimine kõrgtasandi otsustusdiagrammide põhjal

ARTJOM JASNETSKI

Contents

List of publications

The list of author's publications, on the basis of which the thesis has been prepared:

I Jasnetski, Artjom; Ubar, Raimund; Tsertov, Anton; Brik, Marina (2014). "Software-based self-test generation for microprocessors with high-level decision diagrams". Proceedings of the Estonian Academy of Sciences, 63 (1), 48-61.

II Jasnetski, Artjom; Raik, Jaan; Tsertov, Anton; Ubar, Raimund (2015). "New Fault Models and Self-Test Generation for Microprocessors using High-Level Decision Diagrams". IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS. Belgrade, Serbia, April 22-24, 2015: IEEE Computer Society Press, 251-254.

III Jasnetski, Artjom; Oyeniran, Adeboye Stephen; Tsertov, Anton; Schölzel, Mario; Ubar, Raimund (2016). "High-level modeling and testing of multiple control faults in digital systems". IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, 20-22 April 2016. IEEE, 1-6.

IV Jasnetski, Artjom; Ubar, Raimund; Tsertov, Anton (2017). "Automated Software-Based in-field Self-Test". International Journal of Microelectronics and Computer Science, 8 (2), 57-64.

V Ubar, Raimund; Jasnetski, Artjom; Tšertov, Anton; Oyeniran, Adeboye Stephen; (2018). "Software-Based Self-Test with Decision Diagrams for Microprocessors". 978-613-7-33947-3, Beau Bassin: LAP LAMBERT Academic Publishing, 171p.

Author's contribution to the publications

Contribution to the papers in this thesis are:

I The author participated in the decision-making process. The author planned the case study and contributed to the model creation. The author planned and prepared the evaluation environment. The author executed the necessary experiments. The author took part in the preparation of the paper for publication and presented it at a conference.

II The author contributed to the concept. The author developed test programs and ran experiments. The author wrote and prepared the paper for publication and presented it at a conference.

III The author developed the methodology through numerous discussions with the supervisors. The author carried out experiments. The author prepared the paper for publication and presented it at a conference.

IV The author developed the concept. The author implemented the concept in the software. The author implemented all of the necessary software components of the evaluation environment. The author planned and executed the necessary experiments. The author wrote the paper and presented it at a conference.

V The author wrote multiple chapters and sections of the book. The author prepared the book for publication.

Abbreviations

ALU	Arithmetic and Logic Unit
ATE	Automatic Test Equipment
ATPG	Automated Test Pattern Generator
BDD	Binary Decision Diagrams
BMC	Bounded Model Checker
CFFM	Control Functional Fault Model
CSAF	Conditional SAF
DD	Decision Diagram
DFFM	Data Functional Fault Model
DFT	Design For Testability
EDA	Electronic Design Automation
FCT	Full Conformity Test
FSM	Finite State Machine
HLDD	High-Level Decision Diagrams
ISA	Instruction Set Architecture
ISDL	Instruction Set Description Language
MUT	Module Under Test
NDA	Non-Disclosure Agreement
PCT	Partial Conformity Test
RTL	Register-Transfer level
SAF	Stuck-At Fault
SBST	Software-Based Self-Test
SCB	SAF, CSAF, Bridging
SSBDD	Structurally Synthesized BDD
TPG	Test Pattern Generation
TTPG	Targeted Test Pattern Generation
VLSI	Very Large Scale Integration

1 INTRODUCTION

The field of Software-Based Self-Test (SBST) has been a topic of extensive research in industry and academia for more than three decades. Despite this, an automated SBST generation is still lacking a suitable formalisation for modelling of microprocessors.

This thesis presents a methodology to formalise and automate SBST synthesis, leading to a reassessment of the microprocessor modelling process.

1.1 Motivation

Advances in modern technology in manufacturing and design of microprocessors are continuously increasing the difficulty of digital circuit testing. The manufacturing technology of integrated circuits is scaling, allowing the increase of transistor count per chip and increasing operation frequency. Such technology enables microprocessors to be built from billions of transistors and to operate at GHz frequencies. However, the manufacturing of chips has led to the emergence of different physical defects, which affect the parameters of the manufactured device. Therefore, advances in test methodology enable the production of integrated circuits of high quality without increasing the final cost. The varieties of different approaches to microprocessor testing reflect the continuous interest in this topic from academia and industry.

The development of methods for testing such complex digital circuits as microprocessors has been on-going for decades. Test generation time, consumed by sequential automated test pattern generator (ATPG) is, typically, beyond the constraints imposed by industry. The most common solution to testing VLSI designs is to apply design for testability (DFT) methods, such as insertion of scan-chains [1] [2]. Today, application of such a DFT technique is inevitable. However, scan-chain affects the design of a product and requires expensive test equipment.

During the last decade, the semiconductor industry has been challenged to launch new testing methods that can be incorporated into an established microprocessor test flow [3]. The primary demand is the manufacture of a high-quality product without increasing the cost of testing. A test method that raised product quality with only a minor cost increase was first proposed in 1980 [4], and is the SBST.

The main principle of SBST is to use the resources of the processor under test in order to test itself by executing programs. This approach does not require expensive external test equipment, and the test time depends on the performance of the processor and the size of the test program. The generation of test programs that allow high-quality fault coverage is the main research subject in the field of SBST.

The efficiency of test program generation (quality, time) is highly dependent on the abstraction level of representing the system and on the adequacy of fault models. Owing to the increasing complexity of digital systems like microprocessors, the gate-level approaches to test generation require more time in comparison to high-level approaches.

Due to the lack of efficient formal methods, self-test programs for microprocessors are generally written manually. High-level fault modelling approaches and formal test generation strategies have not been sufficiently investigated to support the automated synthesis of self-test programs and to provide fast methods of test quality evaluation.

Over the last years, academia has renewed its interest in SBST for in-field application on embedded devices. ISO 26262 [5] describes the demands for online periodic testing of processor cores in automotive devices. As a result, demand for SBST has increased

following the release of IEC 61508 [6] for industrial safety systems, ISO 26262 for automotive applications, and DO-0254 [7], not to mention the use of processor-centric systems in safety-critical applications.

The lack of access to structural information of commercial products due to NDA makes the functional SBST approach an exclusive solution for in-system or in-field testing. Concurrently, interest has arisen in the automation of the SBST approach, since the complexity of manual test program generation can be unacceptably high. Automated SBST [8] [9] [10] positively influences test development cost, which in turn affects the final price of a product.

1.2 Objectives

The previous section identified the importance of the development of formal methods of SBST generation, with the aim of automation, keeping in mind the constraints imposed by industry. To meet the demand, this research has the following objectives:

- The industry needs *efficient* (in terms of fault coverage) and *scalable* methods of SBST generation for microprocessors
- The industry needs a *formal solution* for *automated* SBST generation, or at least *assisted* SBST generation
- The industry needs a solution for SBST program generation which will satisfy demands of *in-field testing of microprocessors*

1.3 Problem formulation

To achieve the objectives formulated in the previous section, this thesis will solve the following problems:

- Efficient SBST programs
- Formalisation of SBST generation approach
- Automated generation of SBST programs
- SBST generation based only on information retrieved from documentation describing instruction set architecture

The goal is to improve the scalability of SBST generation by working with the highest possible level of abstraction – the instruction set description of a microprocessor. This also allows the widening of the scope of application of SBST to include the generation of in-field testing, where structural information of commercial products is kept under NDA.

A well-formalised approach to SBST synthesis is introduced, extending the high-level decision-diagrams methodology to include modelling microprocessors at the behavioural level. Another extension allows the modelling of behavioural level faults in microprocessors with HLDDs, introducing new high-level fault models. Both extensions expand the opportunities to automate the generation process of efficient SBST programs.

The goal of this thesis is to provide a concept of the platform for automated SBST program generation, which is based on the proposed formal methods for modelling of microprocessors.

1.4 Contribution

The main contributions of this thesis are listed below:

- A methodology for modelling microprocessors on the basis of its instruction set architecture
- Definition of new high-level classes of fault models for microprocessors, which are also mapped to corresponding low-level structural faults
- A formal method for generation of SBST on the basis of the HLDD model
- Framework for automated SBST synthesis

1.5 Thesis structure

The rest of this thesis is organised as follows.

Chapter 2 presents the background and overview of the microprocessor test, in particular the SBST methods. Different approaches to SBST generation are discussed and compared. This chapter presents background information on contemporary hardware modelling techniques, specifically modelling with high-level decision diagrams (HLDDs). The formal definition and basic principles of modelling with HLDDs are outlined.

Chapter 3 forms the core part of this thesis, presenting the method of building models for microprocessors from instruction set architecture description. The main properties of this modelling approach are discussed, with examples of the abstract microprocessor and the processor Parwan.

Chapter 4 gives an overview of existing fault modelling techniques for digital systems and introduces a novel HLDD-based fault model for microprocessors. Multiple high-level fault classes are proposed, dedicated to the control part and data path of the processor. Chapter 4 shows the mapping of existing high-level and low-level fault models for microprocessors in the proposed HLDD-based fault model. Several examples of HLDD-based fault model interpretations are outlined and compared with existing fault models.

Chapter 5 presents the methods for constructing SBST programs on the basis of the HLDD model. Two concepts are discussed: the conformity test, which targets the control part, and the scanning test for exercising the data path of the processor. The numerous advantages of the proposed HLDD-based test generation methods over traditional approaches are discussed. Experiments of quality and compactness evaluation on the manually synthesized SBST program conclude the chapter.

In Chapter 6, the implementation of the framework for automated SBST program generation is described, utilising the concepts described in Chapters 3 - 5. A bottom-up automation approach is presented, starting with the automation of microprocessor modelling, followed by automated test generation, and concluding with SBST program composition using the example of MiniMIPS processor.

Chapter 7 draws conclusions for the thesis and outlines the directions of the future work.

2 BACKGROUND

In this chapter, there is a discussion of the state-of-the-art microprocessor testing, starting with a general classification of test methods and venturing into the field of SBST approaches. This overview identifies the unsolved problems in the area of microprocessor testing and determines the boundaries where the method proposed in this thesis would best fit.

Since the biggest part of this thesis is dedicated to extending the area of application of HLDDs to microprocessor testing, an introductory description of this modelling approach is added.

2.1 State-of-the-art in microprocessor test

Different approaches in the field of microprocessor test can be distributed into three major groups: structural methods, functional methods and software-based self-test methods. The first approach - structural, is a widely-used solution for testing microprocessors. It is based on applying most common DFT technique - scan chain insertion [1] [2] into digital design. Scan chain structure provides sufficient test access to the resources of a processor core. However, adding scan chains affects the initial design of a product, and its parameters: performance, power consumption and chip area. Any change in design can be critical for such highly optimized devices like microprocessors. Still, applying DFT techniques is an inevitable part of wafer and package test within high-volume manufacturing flow. Test procedures that involve DFT structures require special external test equipment, which is limited in speed and affects the final quality of the test. Additionally, it is known, that stuck-at fault tests are more effective when applied at speed [11].

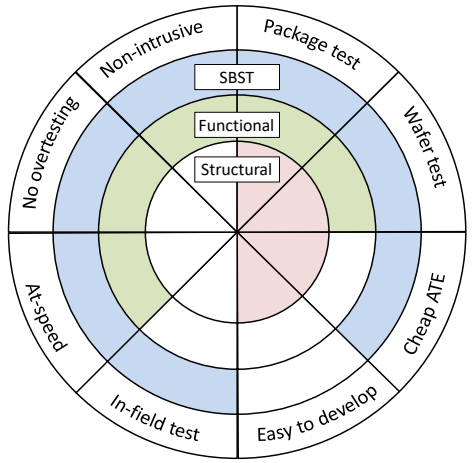


Figure 2-1 Features of microprocessor test methods

The second approach - functional, is capable of conducting tests at operational speed. Functional test is also employed in the final stage of chip manufacturing - speed binning. According to [12], the cost of functional automatic test equipment is about 3000 US dollars per pin for testing at speed of 1 MHz (the year 2000). Additional expenses in the amount of 0.5-1.2 millions of dollars are added by function generators for mixed-signal circuits. Due to the high cost of functional test equipment, the industry

raised the interest in structural scan-based test, which can negatively affect the yield due to over-testing.

Last but not least comes the method for testing microprocessors [4] that is called software-based self-test. The general idea of this method is to use the resources of a microprocessor to test itself by running specific test programs. This method was accepted by industry [13] and is complementing the other two test methods within the manufacturing process. Furthermore, interest in this method was raised in frames of in-field test. Currently, all of the manufactured microprocessors are going through all of these three test methods. The capabilities of the described methods are outlined in Figure 2-1. These three methods are complementing each other in order to increase the quality of the final product.

2.1.1 Software-Based Self-Test

Software-based self-test method was introduced in 1980 by S. M. Thatte and J. A. Abraham [4]. The approach of SBST was characterized as an attractive and promising functional test method, utilizing microprocessor organization and instruction set as parameters of test generation procedures. The main principle of this method is to execute a program on a microprocessor in order to test its own resources. Such approach does not require specific test hardware, and test sequences are executed at processor actual speed, allowing effective coverage of stuck-at faults [11]. The distinctive features of software-based self-test method are:

- **Nonintrusive**. SBST does not need additional ATE, which makes this approach more affordable, and can decrease the final price of a product. In addition, the characteristics (like power consumption, size or performance) of device are not affected by additional hardware on chip.
- **At-speed**. Tests are being run at actual processors speed, making Stuck-at fault tests more effective, and additionally cover delay defects.
- **Avoid overtesting**. Since SBST can use only instructions from defined set, there is no possibility to cover defects that cannot be activated during normal operation of the processor, thus lowering the over-testing effect.
- **In-field test**. Test programs can be reused in-field, or during product lifetime. Also, during return tests and diagnosis.

The general principle of SBST method emphasizes two major aspects of research in the area of SBST: test program generation and execution. Test execution is moderately trivial in comparison to test generation. First, in order to apply SBST, test program should be loaded into memory or cache [14], using external hardware. Then, execution of the test program should be initiated. The test program is generating responses that are stored back to memory. Finally, external hardware evaluates obtained responses and gives the diagnosis for the unit under test. Such test execution flow is used widely and has only minor differences between SBST approaches.

Currently, the second part - test generation, is the main subject of research in SBST field. This part is not trivial and must comply with high boundaries of quality requirements, imposed by industrial standards. The task of test SBST generation can be divided into four parts:

1. Development of code templates for test pattern delivery
2. Extraction of constraints imposed by instruction set architecture
3. Synthesis of test patterns for microprocessor
4. Conversion of test patterns into a test program

All these steps are fundamental for research in the area of SBST. Different research groups are investigating test generation in general, its automation or the quality of test programs. Nevertheless, there are plenty of disadvantages, which leave SBST as a complementary method for testing along with matured structural and functional test. According to Figure 2-1, SBST is more difficult to develop, in comparison to scan-based test. In addition, in comparison to SBST, there are industrial EDA tools available, which can generate structural tests that are capable of achieving high fault coverage. Functional test has also good fault coverage and covers the defects, which structural test did not.

SBST approaches can be divided into two major groups, which are structural and functional. These two groups are defined in this way: the functional group, containing methods that use instruction set architecture (ISA) information of the processor. The other group consists of structural approaches [15] [16] [17] [18] [19], based on generation using structural information (gate- or RTL-level description) of processor under test. These methods have different benefits and limitations because of their nature. Structural approaches benefit from information hidden in the depths of low-level design. Functional approaches are capable of test program generation without structural information, which is usually not available for commercial processors. The lack of such information can be the reason (not without exceptions) for less fault coverage in comparison to methods based on structural approaches.

2.1.2 Structural SBST

Due to its nature, a structural approach can be applied during the production of microprocessors, since structural information is usually available for a manufacturer. Structural SBST solutions can be divided into two major groups. These groups are - hierarchical, and RTL-based Structural SBST methods. Hierarchical approaches use the methodology of considering processor as modules. Only one module is considered at the same time, and stimuli are generated for it. After this, it is translated into stimuli for processor level. Then, these stimuli are being translated into instructions, and the test program is being composed.

First work using hierarchical structural SBST was proposed by Gurumurthy et. al [15]. In this case, ATPG tool is used to generate stimuli for activation of hard to detect faults in modules of a processor core. Then, generated stimuli were filtered with help of bounded model checker in order to match with instruction set of the processor under test. The next approach by Lingappan and Jha [16] is based on satisfiability-based ATPG. They proposed a framework, which evaluates the description of micro-architecture of the processor, by building models for each module of the processor under test. After that, test stimuli are being generated for each module, which are again filtered by satisfiability solver. Additional DFT changes are made to the system in order to apply generated tests.

In [15], Gurumurthy et al. describe the problem of hard to test faults, which cannot be covered by test programs generated randomly. They applied ATPG on each module of the processor, which has hard to test faults. Bounded model checker (BMC) was used to decide which instruction can activate inputs of the module with the precomputed stimuli.

Next hierarchical approach is based on learning [17] algorithms. The work is based on functional test generation approach (also called Targeted Test Pattern Generation - TTPG) where simulation results are used to guide the generation of additional tests. The proposed methodology for TTPG has two phases - simulation and generation. During simulation phase, the simulation I/O data is recorded for the modules under

learning. After data is collected, the specific learning method is used on each module to derive its learned model. Variety of the learning methods is presented in this paper [17]. In the TPG phase, the learned models replace actual modules before and after module under test (MUT). Then, structural ATPG is applied to produce the tests for detection of faults within the MUT. The inputs are then justified through the learned models to the processor's primary input boundaries and outputs propagated to output boundaries.

The second structural SBST method is RTL-level based. It uses information, obtained from both RTL and ISA descriptions. This information is used to generate instruction sequences for activation and propagation of the faults. For the first time, RTL SBST methodology was proposed in [18]. The development of the SBST is based only on the Instruction Set Architecture of the processor and its RTL-level description. The proposed SBST methodology consists of the three phases. During the first phase, the extraction of information from processors ISA for controlling and observing registers of the processor is made. During the second phase, the processor components are being categorized into classes with the same properties (functional, control, hidden components) and prioritized for test development. The last phase is focused on the development of deterministic SBST routines using compact loops of instructions.

Another interesting work is [19]. Different levels of processor description, starting from ISA description and going deeper to a gate-level netlist, are used in this approach. Each part of the processor is being threaded on the best matching level for pattern generation. For example, test for register bank is generated using RTL level description. Tests for ALU are generated using ATPG on a gate-level.

Despite the good results in terms of fault coverage, the efficiency and scalability of the presented methods is questionable, due to the tendency of increasing complexity and size of modern microprocessor designs.

2.1.3 Functional SBST

One of the first methods among functional SBST, proved its efficiency, is the method for SBST program generation using ISA description which was proposed by Shen and Abraham in [20]. They developed a framework called "Vertis", which generates test programs by manipulating with instruction set of the processor under test. For each instruction being tested, "Vertis" generates different test sequences. Test sequences can be generated pseudo-randomly, and use random data, or can be selected manually, which is not a trivial task. The framework can be used during different stages of production - verification, production test and post-manufacture test. Test program is verified experimentally on Intel 8085, covering satisfactory 90.2% of stuck-at faults, which was better fault coverage in comparison to ATPG tools. Significant drawback of this approach is test program size.

The next approach, by Parvathala, Maneparambil and Lindsay [14], is called "FRITS" (Functional Random Instruction Testing at Speed). In this approach test programs are generated from randomly selected instructions and pseudo-random data. Generally, this approach is based on test program generation with random instruction sequences using pseudo-random data. Also worth noting, that in this work cache-resident SBST mechanism is proposed for the first time. This method allows to run test programs directly from cache memory providing "isolated test" during wafer test. The main limitation of cache-resident mechanism is that "cache misses" nor "bus cycles" should not be produced. Test programs, generated by FRITS are verified on Intel Pentium and

Itanium processors, obtaining decent fault coverage results with 70% and 85% of stuck-at fault coverage respectively.

Bayraktaroglu, Hunt and Watkins propose the alternative cache-resident method for production testing [13]. These works both contribute to the usefulness of SBST approach in the production of industrial processors. Their approach is evaluated on Sun UltraSparc T1 microprocessor core. Test program is randomly generated, and the approach mostly concentrates on the development of the mechanism for cache-residency called "Load&Go", especially for the Sun processor family. Achieved fault coverage results are comparable to results obtained with commercial high-cost functional tester.

An alternative approach was proposed by Corno et al. [21]. This approach is based on so-called evolutionary algorithm. In the sense of microprocessor test, evolutionary means that each program is being re-evaluated and only the effective code is attached to it. In the process of test program generation, the feedback from test simulator is used. The algorithm was tested on Leon2 microprocessor and showed the superiority on purely random method in case of fault coverage, and test program length. This method uses the result of gate-level fault coverage as a feedback for evolutionary algorithm. However, it is impossible to apply this method for in-system test generation for commercial microprocessors due to lack of structural gate-level information.

Later research has shown the significance of holding in mind the complexity of processor architecture. The presence of pipeline is adding complexity to test program generation. Latest papers about SBST methodology are concentrating on the processors with pipeline, branch prediction [22] or caches [23]. Gizopoulos et al. in [24] are proposing a method to enhance SBST program quality by considering the properties of pipelined architecture and features of memory addressing of microprocessor under test. Their approach is using data about the architecture of the pipeline and the memory hierarchy to add program code lines in order to activate faults. The experimental results are promising, adding average improvement of 12% for miniMIPS and OpenRISC1200 processors.

Another approach was made by Bernardi et al. [25]. It is also concentrating on the testing of the pipeline, and proposing the strategy for improving test programs for better test coverage with pipelined processor miniMIPS. The proposed strategy is capable to cover faults in the pipeline logic, activated when data hazards or register forwarding problems occur. Their later research is widened with deeper analysis of decode stage of the pipeline in RISC processor [26].

Nevertheless, none of the reviewed methods is relying on formalized solution for modelling microprocessor functionality and faults. Such limitation leaves proposed approaches with problems of hard-to-test faults and fault masking at higher levels. Without theoretical basis for fault simulation and identification it is impossible to measure coverage of wide spectre of fault classes. Additionally, we consider well-formalized modelling of microprocessors as an essential element of automated SBST generation.

2.2 Formal models used in academia

The history [27] of using Binary Decision Diagrams (BDD) for representation and manipulation of Boolean functions is half-century-long. BDDs were first introduced for logic simulation in 1959 [28], and for logic level diagnostic modelling in [29] [30]. A new data structure - reduced ordered BDDs (ROBDDs) [31] was proposed by Bryant in 1986. BDDs became one of the most popular representations of Boolean functions [32] [33],

because of the simplicity of the graph manipulation and the model canonicity. Multiple types of BDDs have been proposed and investigated during decades, such as shared or multi-rooted BDDs [34], ternary decision diagrams (TDD) [32], multi-valued decision diagrams (MDD) [35], edge-valued BDDs (EVBDD) [34], functional decision diagrams (FDD) [36], zero-suppressed BDDS (ZBDD) [37], algebraic decision diagrams (ADD) [38], Kronecker FDDs [39], binary moment diagrams (BMD) [40], free BDDs [41], multiterminal BDDs (MTBDD) and hybrid BDDs [42], Fibonacci decision diagrams [43] etc.

Along with traditional (functional) use of BDDs, application of BDDs for modelling of the structural aspects of the circuit was proposed in [29] [44]. Pioneering alternative graphs (AG) were introduced as a special class of BDDs [29] synthesized directly from the gate-level description. Further, they were renamed to structurally synthesized BDD (SSBDD) [44] [45].

Although logic and RTL level modelling using BDDs is well developed, multi-level and hierarchical modelling is not covered with listed types of BDDs. In this thesis, we consider using high-level decision diagrams (HLDD) [44] [45], which can be used to model systems on different levels of abstraction, and because of their capability for uniform graph-based fault analysis and effect-cause or cause-effect diagnostic reasoning [45]. Additionally, HLDDs are satisfying the constraint of functional SBST, capable of synthesizing the model of the microprocessor from its instruction set architecture description.

Alternative solutions for ISA based modelling of microprocessors are available [46] [47], but their application for fault-modelling, diagnostics and testing are unknown in comparison to HLDDs [48] [49] [50] [51] [52].

2.2.1 Formal definition of high-level decision diagrams

High-level decision diagrams were proposed by Professor Raimund Ubar in 1983 [53]. Application area of HLDDs includes test generation and simulation due to its ability to efficiently and uniformly describe the structure, function and faults in digital circuits [51]. HLDD model can be efficiently used for simulation and fault modelling, capable of fast evaluation by graph traversal and easy identification of cause-effect relationships [54] [55].

A formal definition of high-level decision diagrams was given in [27]. Consider a digital subsystem $U = \{U_{Yout}, U_Q\}$, represented as a cycle-based finite state machine model described by the output vector function $Y_{OUT} = \lambda(X,Q)$, and state transfer (next state) vector function $Q^{t+1} = \delta(X,Q^t)$, where t denotes the number of the current cycle (e.g. clock, microinstruction or instruction cycle).

Definition 2-1. Consider a digital system represented as a universe of functional variables $U = \{U_D, U_C\}$ where U_D is a set of data variables, and U_C is a set of control variables.

A decision diagram G_Y (example in Figure 2-2)which represents a digital subsystem described as a vector function $Y = F(X)$, $Y \in U$, is defined as a non-cyclic directed graph $G_Y = (M, \Gamma, X)$ with a set of nodes M, a set of vector variables X , and a relation Γ in M. Denote the *root node* of G_Y as $m_0 \in M$. The set of nodes is partitioned into two subsets $M = M^N \cup M^T$ where M^N is a set of *non-terminal nodes*, and M^T is a set of *terminal nodes*. The nodes $m \in M^N$ are labelled by variables $x(m) \in X$, and the nodes $m \in M^T$ are labelled either by constants, variables or algebraic expressions (denoted by $f(m)$) of the variables $x \in X$. Concatenate the argument variables used in $f(m)$ as a vector $x(m)$. The mapping Γ describes the topology of the HLDD, how the nodes are connected by

edges where the subset of *successor nodes* of *m* is denoted by Γm, and the subset of *predecessor nodes* of *m* is denoted by $\Gamma^{-1}m$.

For each value *e* from a set V(*x*(*m*)), there exists a corresponding output edge (*m*, m^e) from the node *m* into the successor node $m^e \in \Gamma(m)$, $e \in$ V(*x*(*m*)).

$G_y = (M, \Gamma, X)$;

$M = M^N \cup M^T = \{m_0, m_1, m_2, m_3, m_4\}$;

$\Gamma = \{e_1, e_2, e_3, e_4, e_5\}$, $e_1 = (m_0, m_1)$, $e_2 = (m_0, m_3)$, $e_3 = (m_0, m_4)$, $e_4 = (m_1, m_2)$, $e_5 = (m_1, m_3)$;

$X(m_0) = X(m_4) = x_2$, $X(m_1) = x_3$, $X(m_2) = x_4$, $X(m_3) = x_1$;

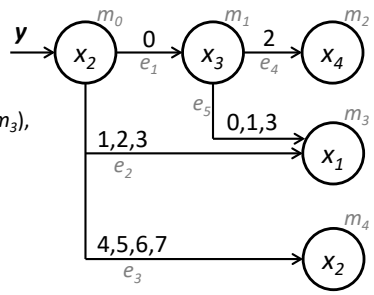


Figure 2-2 function y=f(x1,x2,x3,x4) represented with HLDD

The terminal nodes of the HLDDs, according to Definition 4-1, may be presented at a high functional (or behavioural) level, treating the related hardware modules as black boxes. If a more detailed presentation of the system is needed (for lower level fault simulation or fault diagnosis purposes), the functional expressions in the terminal nodes of HLDDs can be unfolded into lower level implementation descriptions, such as gate-level networks. This allows transforming high-level HLDD-based approach to a hierarchical multi-level approach, where the control functions will be modelled at the higher level using HLDDs, and the detailed data manipulation functions will be modelled at lower levels using SSBDDs.

2.2.2 Operations on HLDDs

In this section, we are outlining following operations on HLDDs: logic simulation, path activation and test generation. The complete list with description of operations on HLDDs is provided in [27].

Logic simulation. Logic simulation of applied vector X^t on graph G_y means traversing the nodes by X^t path $l(m_0, M^T)$ starting from root m_0 up to one of the terminal nodes M^T. The variable x_i of reached terminal mode determines the value of y for the given vector X^t. Example of logic simulation of the input pattern -025 (x_1, x_2, x_3, x_4) on HLDD is shown on Figure 2-3.By traversing of the path $l(m_0, m_2)$ through nodes m_0 and m_1 the output value of the circuit becomes $y = x_4 = 5$.

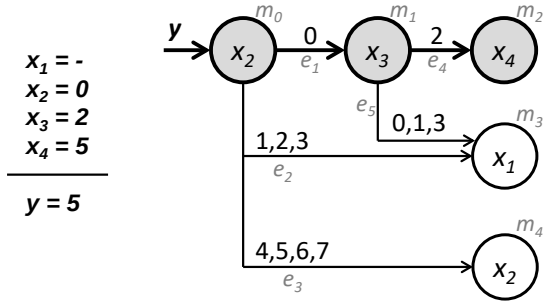


Figure 2-3 Logic simulation on HLDD

Path activation. Activation of the path between nodes m_i and m_j within HLDD requires to find a vector X^t which is capable to activate the path $l(m_i,m_j)$. Such path can be generated by finding the solution to equation $y = f(X)$.

Test generation. The task of test generation for a fault $x(m) \equiv e$ in a G_y, $e \in \{0,1\}$, is solved in G_y by activating the following paths:

$l_m = l(m_0, m)$, from the root node m_0 to the node m under test,
$l_1 = l(m^1, \#1)$, and $l_0 = l(m^0, \#0)$ from the node m to the related terminal nodes #1 and #0, respectively,

whereas the additional fault type constraint $x(m) = \neg e$ should be additionally satisfied. As the result of solving these tasks, a test vector X^t will be found, which detects the fault $x(m) \equiv e$.

2.2.3 Behavioural level synthesis of HLDDs from the procedural descriptions

Consider a procedure representing a behavioural level description of a digital system. It is possible to represent such procedure by a directed graph, such as data flow graph, and a path can be represented by a sequence of assignment statements and conditional expressions (i.e. by a sequence of assertions).

The full procedure of the HLDD synthesis from the behaviour level procedural description of a system consists of the following phases [51]:

State insertion into the procedural description. This action is performed in similarity to data-flow graphs, where behaviour of given automata is marked by states. The states, defined with q, are inserted so that during any state transfer, each data variable is calculated only once.

Creation of the FSM structural table. A table is constructed by tracing all the transfers in the data-flow graph from the previous step. Each row in the constructed table corresponds to a path between neighbouring states of the procedure.

Partitioning of the structural table into functional subtables. At this step, the set of all functional variables is extracted from the description of the system functionality of the FSM structural table. An example of the table with extracted behaviour of functional variable A is shown in Figure 2-4. The table consists of two parts: constraints (q, X_A, X_B, X_C), and assignment statements for variable A (right column). The constraints describe the needed conditions, which have to be satisfied for execution of the related assignment statements.

Generation of mixed predicate formulas for functional variables. Each table, extracted in the previous step can be represented by a mixed predicate formula

$$x = \vee C_i E_{i,S},$$

where x represents a functional variable, C_i is a logic condition (logic AND of all constraints), and $E_{i,S}$ is an algebraic expression of an assignment statement. Example of mixed predicate equation for variable A is shown on Figure 2-4.

Creation of HLDDs for the functional variables. This action is made by using Shannon factorization [32] [33]. The HLDD created by factorization of the mixed predicate formula for variable A is depicted in Figure 2-4. Variable A becomes the output of the graph. Constraint q becomes root node, with successor nodes representing constraints X_A, X_B, X_C. Assignment statements are represented with terminal nodes of the graph.

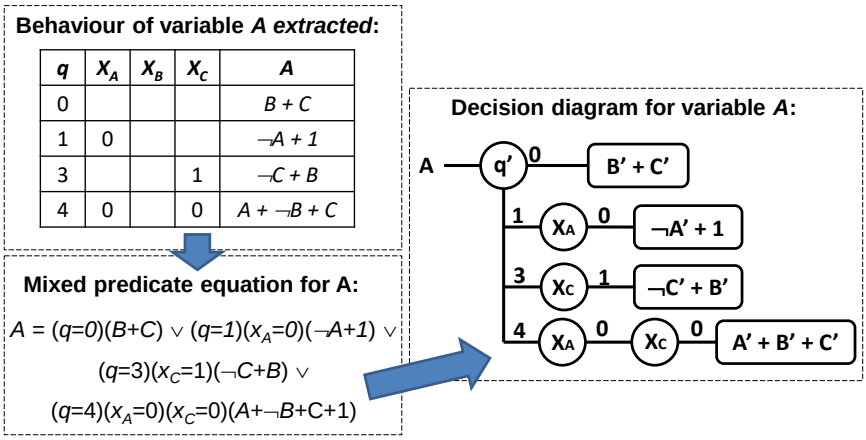


Figure 2-4 Synthesis of HLDD for functional variable A

2.2.4 Topology of HLDDs

Topologically, HLDD consists of a root, terminal and non-terminal nodes. The number of terminal nodes is not limited and is determined by the number of high-level operations supported by the digital circuit. Terminal nodes are labelled by high-level constants (vectors), bus or register variables, or by high-level algebraic operations. The non-terminal nodes of HLDDs represent the control variables. The number of output edges in HLDDs is not limited and is equal to the number of possible values of the control variable of the node. In other words, the non-terminal nodes in HLDDs model the control functions of the digital system, whereas the terminal nodes refer to the data manipulation functions. The Figure 2-5 depicts the described topology of HLDD model.

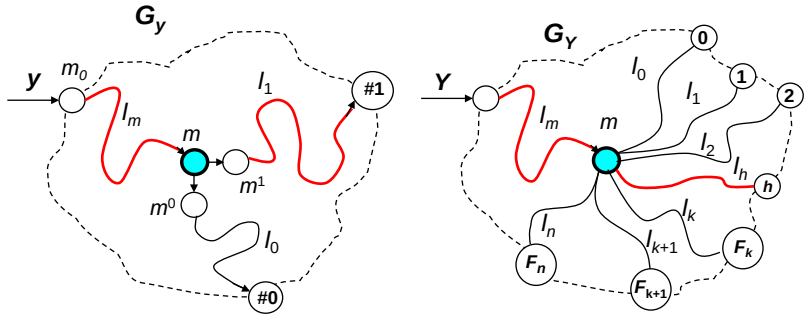


Figure 2-5 Topology comparison of SSBDD and HLDD

Testing of a digital system, represented by HLDDs means testing both types of nodes - non-terminal and terminal. When testing the non-terminal nodes of an HLDD, we are verifying the general control behaviour of the circuit, and when testing the terminal nodes, we are verifying the separate working modes of the circuit.

The procedures of test generation are different. To test a non-terminal node m, one has to activate a path $l_m = (m_0, m)$ from the root node m_0 to the node m, and from the node m, for each value $x(m) = h$, a path $l_h(m, m^h)$ to a terminal node h, so that the paths l_h were not overlapping. Additionally, the values of the data variables should be selected such that the values of operations at the terminal nodes h reached by paths l_h, were different. For testing each terminal node m, one has to activate a single path l_m to

this node *m*. Additional constraints need to be satisfied depending on the fault models adopted for testing the system, and which will be determined in terms of the HLDD model.

Described above forms the general problem of adopting HLDDs for SBST program generation for microprocessors. HLDD methodology should be sufficient to model microprocessor behaviour, described in instruction set architecture. Additionally, on the basis of such model, test programs with decent accuracy should be generated to target certain spectre of faults within microprocessor.

2.3 Summary

This chapter provides an overview of state-of-the-art methods of microprocessor testing, particularly the SBST approach. It describes the solutions for modelling hardware and discusses their limitations.

Specifically, introductory information on HLDDs is presented to provide a better understanding of Chapters 3 and 4, where the area of application of HLDD is extended to model microprocessors at the behavioural level and the faults within them.

Concluding this section, the main challenge of SBST generation for microprocessors on the basis of HLDD methodology is formulated.

3 SYNTHESIS OF BEHAVIORAL LEVEL MODEL OF MICROPROCESSOR WITH HLDDs

This chapter is based on publication I [56], where novel approach for high-level processor modelling using HLDDs was presented. This chapter discusses the extension of HLDDs, which allows the generation of a microprocessor model from instruction set description. The main contributions of this chapter are as follows:

1) *A formal method for modelling microprocessors* using instruction set description is elaborated
2) The applicability of the *approach to microprocessor modelling with HLDDs is evaluated* for an abstract processor and processor Parwan [57]
3) *The features and capabilities of HLDD models* for further use in testing purposes are *evaluated and discussed*

The outlined contributions are elaborated in detail in sections 3.1-3.5.

3.1 HLDD-based modelling for microprocessors

A digital design, like a microprocessor, can be represented with HLDDs at different levels of abstractions – structural, RTL or behavioural. In section 2.2.3, the behavioural level synthesis with HLDDs is discussed. In this work, we propose to move "one step higher" in abstraction to instruction set architecture description, which also represents the behaviour of the microprocessor. In this case, HLDDs are used to calculate the state of the system after execution of each instruction. Following this, we introduce the instruction-cycle based HLDDs, as a convolution in behavioural level modelling of microprocessors.

Table 3-1 Instruction set of a simple hypothetical microprocessor with ten instructions

I	Mnemonic	ISA level operation
1	MVI A,D	$A \leftarrow IN$
2	MOV R,A	$R \leftarrow A$
3	MOV M,R	$OUT \leftarrow R$
4	MOV M,A	$OUT \leftarrow A$
5	MOV R,M	$R \leftarrow IN$
6	MOV A,M	$A \leftarrow IN$
7	ADD R	$A \leftarrow A + R$
8	ORA A	$A \leftarrow A \vee R$
9	ANA R	$A \leftarrow A \wedge R$
10	CMA A,D	$A \leftarrow \neg A$

Consider a simplified hypothetical microprocessor with ten instructions as an example target for modelling. Instruction set of this processor is presented distributed by columns of Table 3-1: in the first column, *I* – is the high-level control variable whose integer values represent the operation codes; in the second column, the mnemonic of the instruction is provided to represent the behaviour hidden behind the instruction; in the third column the operations launched by instructions are described using the high-level data variables. Variable *R* denotes an internal general purpose register, variable *A*

represents accumulator register, variable *IN* denotes the input bus and variable *OUT* denotes the output bus.

For the synthesis of HLDDs, we use the method described in Chapter 2.2.3, omitting the first step in the model generation process. On the basis of the third column, we define the set of functional variables of the microprocessor – *FV* = {*A, R, OUT*}. These are the variables, which describe the state of the microprocessor, and which values are recalculated by execution of instructions. For each variable in *FV*, we synthesize an HLDD, depicted in Figure 3-1.

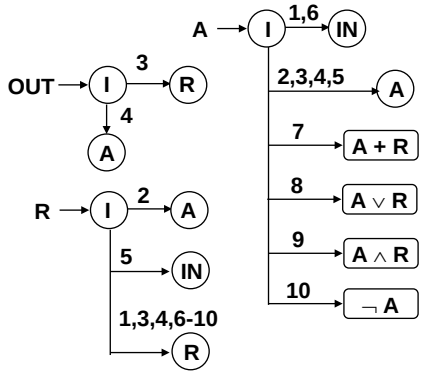


Figure 3-1 HLDDs for the microprocessor with instruction set in Table 3-1

The microprocessor is represented by three diagrams - G_{OUT}, G_R, and G_A. Diagram G_{OUT} represents the behaviour of output bus. The behaviour of internal general purpose register *R* is represented by G_R, and the behaviour of the accumulator *A* by graph G_A. Since there is only one constraint variable – *I*, it becomes a root node of the decision diagram with its values shown at edges. Variable *I* represents the instruction code, thus has the values from 1 to 10, corresponding to the instructions $I_1, I_2, \ldots, I_{10}$. The terminal nodes (successors of *I*) are labelled by the word variables *R* and *A,* representing the corresponding registers, along with data transfer buses (*IN*, *OUT*), or by expressions related to particular data manipulation operations of the microprocessor.

The HLDD model from this example was built based exclusively on the description of instruction set architecture, which is usually provided in the documentation for microprocessor. Despite that, the model can reveal specific, non-documented, information about functional variables, explaining how each variable will behave when different instructions are executed. In comparison to plain instruction-based information, the variable based information is more suitable for microprocessor test and fault diagnosis.

Additional value of modelling using HLDDs is the possibility to derive a high-level structure of the microprocessor from instruction set description. All HLDDs, representing hardware modules and united in the model of the microprocessor, are functionally interconnected by the functional variables used in the description of the instruction set. Hence, the network of connected HLDD-modules can be regarded as a high-level behavioural level structure of the microprocessor. Such a structure, derived from the instructions in Table 3-1, is presented in Figure 3-2.

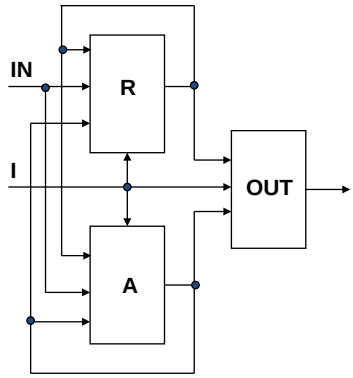


Figure 3-2 ISA-based high-level structure of the microprocessor described in Table 3-1

3.2 Instruction set as a basis for HLDD model generation

In order to demonstrate the feasibility of microprocessor modelling using HLDDs we propose to use more complex system where not only ALU, but also a register block, memory interface, and a control unit (program counter) are involved. For this purpose, we have chosen the microprocessor Parwan [57]. Figure 3-4 represents a high-level structure of chosen microprocessor, Figure 3-6 represents the HLDD model synthesized for its instructions set listed in Table 3-2.

Parwan is an 8-bit microprocessor described in VHDL, which has an 8-bit data bus and a 12-bit address bus for memory accesses. The instruction set of Parwan microprocessor counts 17 instructions in total: memory access, ALU operations, and branch instructions. It also supports direct and indirect addressing modes. Parwan processor includes the following datapath components: arithmetic logic unit (ALU), shifter unit (SHU), accumulator (AC), program counter (PC), status register (SR), memory address register (MAR), instruction register (IR) along with a control unit (CONTROL). It should be noted that the only data register, which is accessible, is the accumulator (AC).

Table 3-2 Instruction set of PARWAN microprocessor

Group	OP	D/I	P	Instruction mnemonic	Operation
A	0	0/1	Page #	LDA	*AC=M* *PC=PC+2* *$N,Z=f_{N,Z}(AC, M)$*
A	1	0/1	Page #	AND	*AC=AC&M* *PC=PC+2* *$N,Z=f_{N,Z}(AC, M)$*
A	2	0/1	Page #	ADD	*AC=AC+M* *PC=PC+2* *$N,Z,C,V=f_{N,Z,C,V}(AC, M)$*
A	3	0/1	Page #	SUB	*AC=AC-M* *PC=PC+2* *$N,Z,C,V=f_{N,Z,C,V}(AC, M)$*
A	4	0/1	Page #	JMP	*PC=A*

A	5	0/1	Page #	STA	*M=AC* *PC=PC+2*
A	6	-	----	JSR	*PC=A*
C	7	0	1	CLA	*AC=0* *PC=PC+1*
C	7	0	2	CMA	*AC=* $\neg$*AC* *PC=PC+1* $N = f_N(AC)$
C	7	0	4	CMC	*C=*$\neg$*C* *PC=PC+1*
C	7	0	8	ASL	*AC=2AC* *PC=PC+1* $N,Z,C,V=f_{N,Z,C,V}(AC)$
C	7	0	9	ASR	*AC=AC/2* *PC=PC+1* $N,Z=f_{N,Z}(AC)$
B	7	1	0	BRA_N	*PC=(N=1)? A : PC+2*
B	7	1	2	BRA_Z	*PC=(Z=1)? A : PC+2*
B	7	1	4	BRA_C	*PC=(C=1)? A : PC+2*
B	7	1	8	BRA_V	*PC=(V=1)? A : PC+2*

Details concerning the usage of different instructions are shown in Table 3-2. Instruction set of Parwan microprocessor is divided into three groups, depicted in Figure 3-3. Instruction word can be 1-byte (group *C*) or 2-byte (groups *A* and *B*) long, and represented with format *OP.I.P* or *OP.I.P.A* respectively. Instructions from group **A** are 2-byte long and support direct and indirect addressing by control field *I*. Field *OP* is used to select the desired operation. 12-bit long address consists of memory page number *P* and offset *A*. Group **B** consists of 2-byte long branch instructions, which can address memory only within single page using offset *A*. Instruction fields *OP*, *I* and *P* are controlling the selection of desired branch operation. Instructions of group **C** are not addressing memory, thus are 1-byte long, where fields *OP*, *I* and *P* are playing role of the operation code.

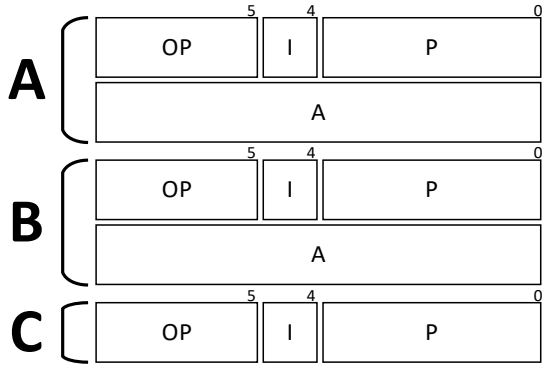


Figure 3-3 Instruction format groups of Parwan microprocessor

Let us partition Parwan into the three parts: control part, data part and memory. Control part consists of finite state machine (FSM) with state register and control logic

and register block R_{CONTR} = {*PC*, *MAR*}, where *PC* is the program counter, and *MAR* is the address register for addressing the data. Data part consists of register block R_{DATA} and ALU. The register block in the data part consists of a single general purpose data register R_{DATA} = {*AC*}. ALU is a combinational part of the microprocessor which covers all data manipulation circuits, decoders, multiplexers, demultiplexers etc.

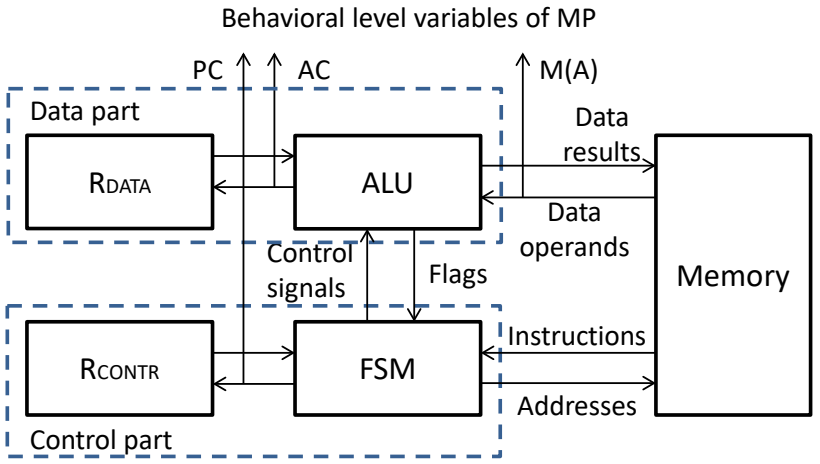


Figure 3-4 Behavioural level structure of Parwan microprocessor

For each variable of the Parwan microprocessor a mixed predicate formula can be extracted from instruction set description, as it was described in Chapter 2.2.3. A set of the following functions represent the functionality of Parwan microprocessor:

1) $AC = f_N(I, S(R)) = f_N(OP, I, P, S(R))$ where *R* is *AC*, $S(R) = \{AC, M\}$ is the set of data arguments for the function f_N;
2) $PC = f_{PC}(I, S(B), PC) = f_{PC}(OP, I, P, S(B), PC)$ where $S(B) = \{N, Z, C, V\}$ is a set of flag variables serving as the condition for branch operations;
3) $S(B) = \{N, Z, C, V\} = f_B(OP, I, P)$ where f_B is a function on operands to determine the flag condition.
4) $M = f_M(OP, P, S(M(A)))$ where $S(M) = \{AC, M\}$.

The functionality of microprocessor can now be represented by a set of behavioural level variables $Z = R_{DATA} \cup R_{CONTR} \cup M$ and by a set of functions $F = \{f_N, f_{PC}, f_B, f_M\}$. The behaviour of Parwan can be modelled by the functional basis *F* and monitored through the variables in *Z*. For modelling of *F* we will use the behavioural level HLDD model.

3.3 Generation of HLDDs for modules of the microprocessor

From the instruction set description, shown in Table 3-2, we can extract the following set of functional variables: 8-bit data vector variables *AC* – accumulator, *PC* – program counter, *M* – generic memory location, and 1-bit branch flag variables *N*, *Z*, *C*, V. Example of HLDD generation for functional variable *V*, which is an overflow flag variable, is shown on Figure 3-5.

OP	D/I	P	Instruction mnemonic	Operation
2	0/1	Page #	ADD	V=f_V(AC, M)
3	0/1	Page #	SUB	V=f_V(AC, M)
7	0	8	ASL	V=f_V(AC)

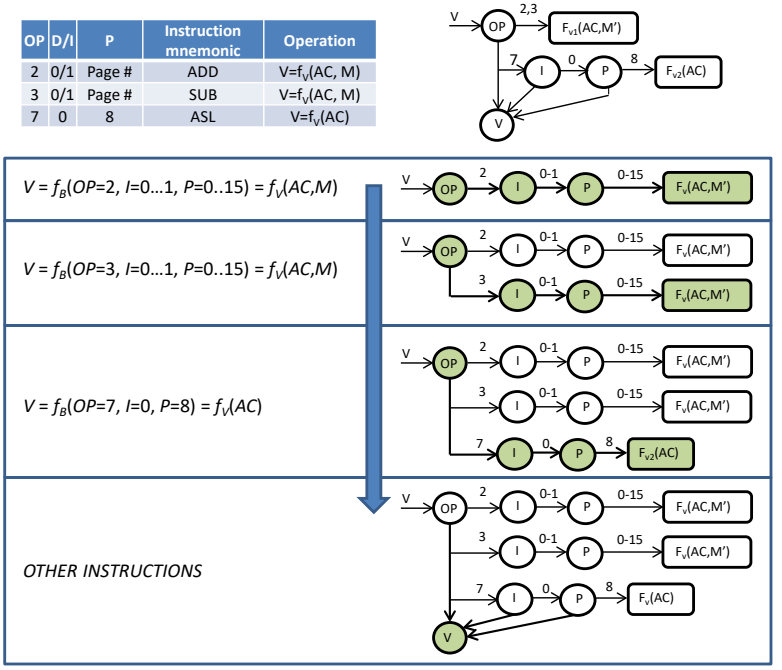


Figure 3-5 HLDD synthesis for functional variable V

First, the subset of instructions, affecting the behaviour of functional variable *V* (flag overflow) is selected, and collected into a smaller table (Figure 3-5). HLDD is being compiled by following this table row-by-row. Functional variable *V* becomes an output of the graph. Then, the graph is populated with green-coloured non-terminal nodes, representing control variables *OP*, *I* and *P* with corresponding values on edges. Function, representing the behaviour, becomes the terminal node of the graph. The expressions in the terminal nodes of HLDDs G_V for calculating the conditions of branch variable *V* are not specified in this model. Since table has only three rows, representing three instructions, the graph will be populated with three paths. One additional path is added to represent the behaviour of functional variable *V* during execution of other instructions – flag overflow holds its previous value.

3.4 Generation of HLDD model for microprocessor

For every functional variable of Parwan microprocessor, outlined in Table 3-2, HLDDs are generated, using the control variables *OP*, *I*, and *P* in non-terminal nodes for decision making. The HLDD model for Parwan microprocessor, consisting of a set of 12 HLDDs is depicted in Figure 3-6. Parwan has a 12-bit address bus, which is partitioned into sixteen pages of 256 bytes each. The four most significant bits of the address are for the page address and the remaining eight bits of the address are for the offset within the page. In accordance with this memory organization, the program counter variable *PC* is represented as a concatenation of two sub-variables *PC = PC_P.PC_A*, and the value of the next PC is composed by concatenation of the values of *PC_P* and *PC_A*, which are calculated by respective graphs "*Next memory page calculation*" and "*Next PC offset calculation*".

Instruction addressing mechanism is described with graphs $G_{OP.I.P}$ and G_A. Instructions of the Parwan microprocessor are encoded using up to two consecutive 8-bit long words (Figure 3-3). The first word consists of instruction fields *OP*, *I* and *P*, and is obligatory for every instruction. The second consecutive word holds the address *A* of specific location in memory, where data, required for this instruction is stored. This organization is modelled with two corresponding graphs $G_{OP.I.P}$ and G_A.*OP*, *I* and *P* fields will be fetched from memory at the address, stored in program counter *LOC(PC_A)*. Address field A will be fetched from address *LOC(PC_A)+1*, pointing to the second part of current instruction.

Fields of fetched instruction are affecting the result of ALU functionality, modelled with diagram G_{AC}. Accumulator register *AC* of Parwan is hardly tightened to ALU, keeping data for one of its input, and result of ALU operation after execution. The second operand for ALU functions (for example addition), is loaded from memory using address kept in *A*. Two modes of memory addressing are supported, selected by the value of field *I*, where 0 corresponds to direct and 1 to indirect addressing. Field *P* is used to address page in memory for functions with two operands (group A), and plays the role of the control variable in case of operations with *AC* register only. *OP* field of instruction word becomes root node of the G_{AC}, becoming the main control variable for selecting the operation of ALU. Terminal nodes of G_{AC} represent the functions of ALU, which behaviour is not modelled here. Functions with two input operands, like *AC+M'* are referring to data in memory. *M'* is used to address data in memory directly, and *M''* is used for indirect addressing mode.

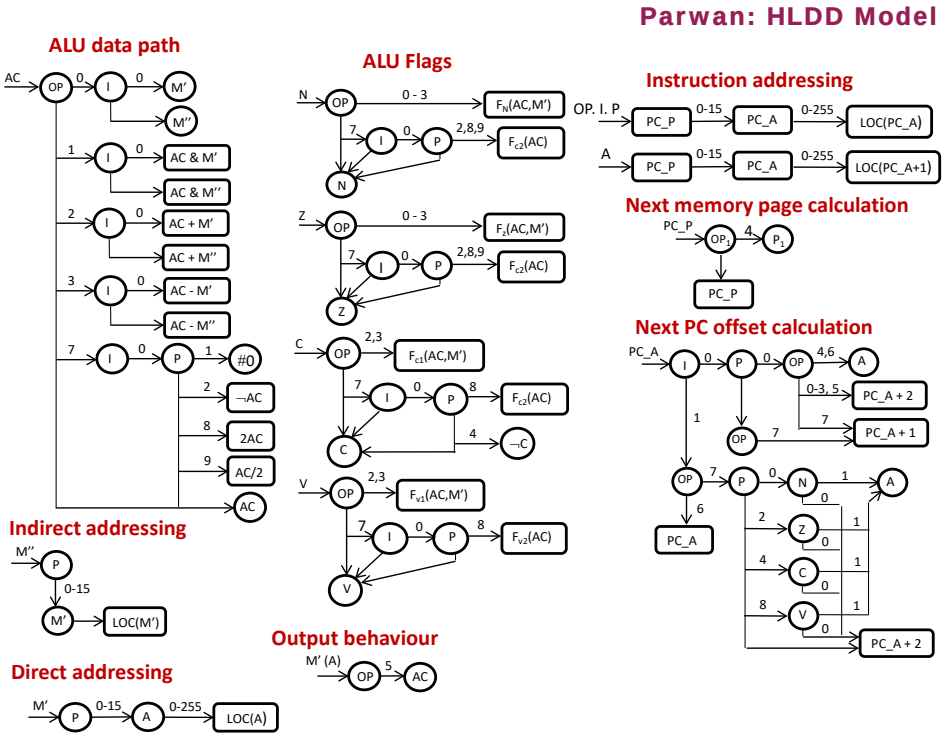


Figure 3-6 HLDD model for the microprocessor Parwan

I/O behaviour is represented by graphs $G_{M'}$, $G_{M''}$ and $G_{M(A)}$. Graphs $G_{M'}$ and $G_{M''}$ are representing the behaviour of loading data from memory to input of ALU, or into *AC* register. Direct addressing is represented with $G_{M'}$, where output *M'* holds the data, fetched from an address in memory selected by variable *P* (representing page number), and *A* (address within a page). $G_{M''}$ inherits the $G_{M'}$ pointing to location in memory, where the direct address to data is stored. Diagram $G_{M(A)}$ represents the behaviour of storing data in memory. The single path, controlled by *OP* variable in this graph is activated, when instruction "STA" (*OP=5*) is executed. In this case, the value of *AC* is being moved to a memory location defined by page *P* and offset *A*.

Flags, required for branch operations are modelled with graphs G_N, G_Z, G_C and G_V. The paths in these graphs are activated simultaneously, during execution of instructions. The decision, if the flag should be raised or not is represented with functions located in terminal nodes. Mostly, the flags depend on the data kept in *AC* register or loaded from memory location. Activation of different flags depends also on instructions, which are executed. For example, flag $N = G_N$ (negative number) can be raised if the sign of binary operand loaded to AC with instruction "LDA" (*OP*=0, *I=I*, *P=P*) is negative. However, during execution of the same instruction, flag V = G_V (overflow) is not affected at all.

Last but not least, graph G_{PC_A} for calculation of the next program counter offset is synthesized. Program counter can keep address within frames of one page only. When executing instructions of group **C** (*OP*=7), which are using only data in AC register, offset is incremented by 1 byte (*PC_A+1*), pointing to the next instruction in memory. Instructions of group **A**, using two operands, like addition (*OP*=2) or subtraction (*OP*=3), are incrementing program counter offset by two bytes (*PC_A+2*), jumping over 8-bit long instruction field *A*. Branch instructions (group **B**), depending on flags, which values are represented with non-terminal nodes *N*, *Z*, *C* and *V* in G_{PC_A} increment program counter by two, if branch is not needed. In the case when branch conditions are satisfied, program counter value is being overwritten by address data, kept in field *A* of a branch instruction (*PC_A = A*).

3.5 Simulation of instructions with HLDDs

It is possible to simulate instructions of the modeled microprocessor using the graph network, built in the previous section. Instruction simulation mechanism is similar to path activation, described in Chapter 2.2.2. Let's consider an example of simulation of the instruction *AND* = (*OP=1.I=0.P=0.*A=*8*) fetched from address 0 in memory. The following paths in Figure 3-7 have to be activated: $G_{OP.I.P}$: *L*(*PC_P=0*, *PC_A=0*, *0*); G_A: *L*(*PC_P=0*, *PC_A=0*, *0+1*); G_M: *L*(*P=0*, *A=8*, *LOC(8)*); G_{AC}: *L*(*I=0*, *P=0*, *OP=1*, *AC&M'*); G_N: *L*(*OP=1*, F_N(*AC*,*M*)); G_Z: *L*(*OP=1*, F_Z(*AC*,*M*)); G_{PC_A}: *L*(*I=0*, *P=0*, *OP=1*, *PC_A+2*) in the graphs $G_{OP.I.P}$, G_A, G_M, G_{AC}, G_N, and G_{PC_A} respectively. The activated paths are emphasized by bold edges and grey coloured nodes.

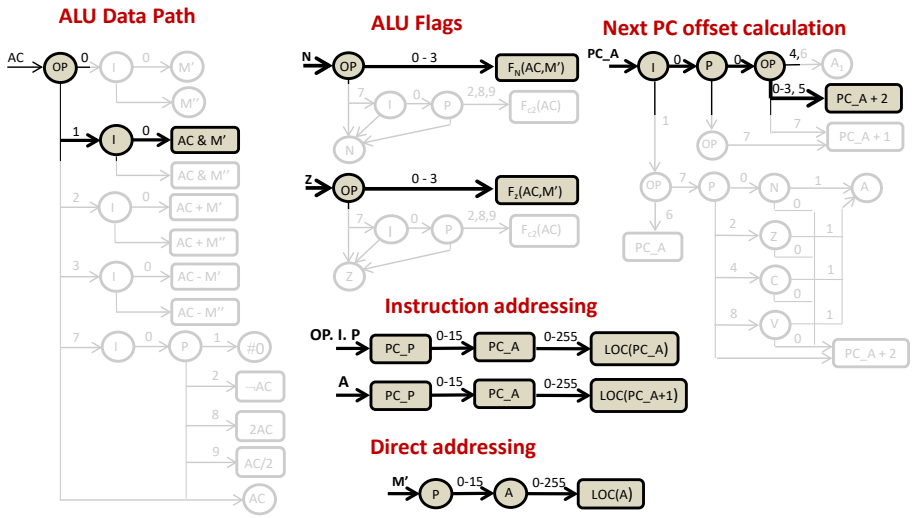


Figure 3-7 AND instruction simulation in PARWAN model

Each node of HLDD model can be considered as a hypothetical structural unit of the microprocessor exercised by a corresponding instruction. For example, the terminal nodes of the graph can be labelled by variables or arithmetic/logic expressions. Nodes, labelled by variables may represent registers or buses. Respectively, nodes, labelled by arithmetic or logic expressions represent the data manipulation sub-units in ALU. The nonterminal nodes of HLDDs are representing control-related units (*OP*, *I*, *P, N, Z* etc.) implemented as decoders, multiplexers or de-multiplexers. For example, the node *P* in G_{PC_A} represents a multiplexer, the nodes OP, and I in other graphs represent decoders.

The one-to-one mapping between the nodes in HLDDs and the matching high-level functionality opens the opportunity to use the HLDD nodes as a checklist for high-level test strategy planning and organization of test programs for microprocessors. For formalized test program generation, however, we need a suitable high-level (behavioural) fault model.

3.6 Summary

The key result of this chapter is the novel methodology for mapping the behavioural level instruction set into the separate control and data parts of the full functionality of the microprocessor, with the goal of improving the accuracy of its high-level modelling. The chapter introduced the HLDD model and discussed its extension to the modelling of microprocessors on the basis of the instruction set description.

First, the *methodology for analysing the instruction set of a microprocessor* under test is presented, followed by the *extraction of functional and control variables*. Every extracted function variable will become a graph variable, and each control variable will become a corresponding node in a graph.

Second, the *method for building graphs on the basis of data, extracted exclusively from instruction set architecture* description is proposed. Using this method, a microprocessor can be represented by the model consisting of a set of HLDDs, representing different functional units. Hence, the network of connected HLDD-modules can be regarded as a high-level behavioural level structure of the

microprocessor. Such a model can reveal specific, non-documented information about functional variables, explaining how each variable will behave when different instructions are executed. An HLDD model can be simulated, which is demonstrated using the example of the Parwan microprocessor model.

Third, the proposed modelling approach allows one-to-one mapping between the nodes in the HLDDs and the corresponding high-level functionality. The benefit of this is the *opportunity to use the HLDD nodes as a checklist for high-level test planning and organisation of test programs for microprocessors*. However, a suitable high-level fault model is required for this, and this will be presented in the next chapter.

Instead of the traditional microprocessor test concept, where the instructions as a whole are regarded as test objectives, a novel and more exact HLDD-driven test concept is introduced in this chapter, with the instructions split into more detailed subsets of test objectives.

4 HIGH-LEVEL FAULT MODELING FOR MICROPROCESSORS WITH HLDDs

In this chapter, a new model of high-level behavioural faults in microprocessors is developed, on the basis of HLDDs, presented in the previous chapter, providing better possibilities of formalising the test program synthesis procedure than the traditional high-level fault models. The material, presented in this chapter is based on publications II [58] and III [59].

The contributions of this chapter are summarised as follows:

1. *An overview of the fault models for microprocessors is given,* where high-level behavioural fault models are found to be more attractive than low-level fault models, in terms of efficiency/complexity ratio.

2. Three *novel classes of fault models for microprocessors*, represented by HLDDs, are proposed. These fault classes are considered compact and well-formalised super classes which cover a larger set of more detailed fault classes used traditionally in the testing of instructions. The new fault model is demonstrated using the HLDD model of Parwan microprocessor.

3. It is shown that the proposed new classes of HLDD-based *high-level fault models can be mapped onto and cover the lower level fault model subclasses*, particularly RTL-level and structural gate-level fault models, in order to guarantee the high quality of testing.

4.1 Fault modelling in digital systems

Fault modelling, being a central target, is an inseparable part of test generation and fault simulation. Despite the similarities, these tasks differ in the complexity. The complexity of fault simulation is linear, being insensitive to the size of fault lists to be simulated, is satisfied with existing low-level fault models. Test generation, in its turn, needs high-level fault modelling to cope with its high complexity.

Test generation task is always facing a trade-off between efficiency (cost of test generation) and quality (fault coverage) of outcome. Both criteria are highly depending on which fault models are used in test generation and in fault simulation for test quality assessment.

The stuck-at fault (SAF) model has been for a long time the prevalent technique to handle formally real physical defects in electronic systems. In today's systems, however, we have two difficulties when using this model: it is too complex for use in test generation because of the huge number of faults to be handled in systems, and it is inaccurate to represent real physical defects taking place in today's nanoelectronic circuits [27].

A conditional fault model has been proposed as an extension of the SAF model [60] [61]. It helps to increase the model accuracy of arbitrary physical defects in the modern complex digital systems, like microprocessors with nanometre technology. Applying of this model positively affects the size of the fault set and decreases the complexity of test generation. This model is also known as fault tuple model [62], pattern fault model [63], input pattern fault model [64], or functional fault model [65].

Similar models are gate-exhaustive fault model [66], and region-exhaustive fault model [67]. Many researchers have focused on developing new fault models for particular types of failure mechanisms like bridges [68] [69] [70] [71], transistor

stuck-opens [72] [73], failures due to delays [74] etc. For resistive shorts, opens and bridges a unified fault model as constrained multiple line SAF was proposed in [75]. All of them are developing the idea that a single fault can affect different combinations of fan-out branches.

To increase the speed of test generation and fault coverage evaluation, high-level fault models have been developed. High-level approaches for fault modelling in digital systems can be grouped into two different classes: (1) high level fault modelling for structural RTL descriptions [76] [77] [78], which is characterized with certain relationship between language constructs and the network structure; (2) behavioural level fault modelling [79] [80] [81] [82], which is oriented to analysis of only algorithmic descriptions. We consider as behavioural approaches also the high-level fault modelling of microprocessors which use only the information about instruction set architecture (the lists of instructions). Therefore, our solution to fault modelling, relying on information derived from instruction set description, does belong to the group of behavioural approaches.

High-level fault models for microprocessors have been usually derived from the high-level behavioural descriptions of instruction sets. State-of-the-art behavioural approaches such as [83] [77] [80] [81], distinguish following fault models *Fn*.

For faulty multiplexers, for a given source address any of the following fault models can be applicable:

F1: source is not selected;
F2: selected source is wrong;
F3: more than one source is selected and the multiplexer output is either a wired-AND or a wired-OR function of the sources, depending on the technology.

For faulty demultiplexers, for a given destination address any of the following fault models can be applicable:

F4: destination is not selected;
F5: instead of, or in addition to the selected correct destination, one or more other destinations are selected.

An instruction *I* of a microprocessor can be regarded as a sequence of microinstructions, where each microinstruction consists of a set of microorders which are executed in parallel. Microorders represent the elementary data-transfer and data manipulation operations. Addressing faults affecting the execution of an instruction may cause one or more of the following fault effects:

F6: one or more microorders not activated by the microinstructions of *I*;
F7: microorders are erroneously activated by the microinstructions of *I*;
F8: a different set of microinstructions is activated instead of, or in addition to, the microinstructions of *I*.

The data storage facility is usually implemented as a memory. Under a fault any of the following may happen to the memory cell array:

F9: one or more cells are stuck at 0 or 1;
F10: one or more cells fail to make a $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions;
F11: two or more pairs of cells are coupled; this means, a transition from *x* to *y* in one cell of the pair, say cell *i,* changes the state of the other cell, say *j*, from *x* to *y* or from *y* to *x*, where $x \{0,1\}$, and $y = \neg x$

F12: The data-transfer function implements all the data transfers along the buses between the registers and functional units of a microprocessor.

For buses under a fault:

F13: one or more lines can be stuck at 0 or 1;

F14: one or more lines may form a wired-OR or wired-AND function due to shorts or spurious coupling.

F15: data processing functional fault model; in the case of data processing functional units, no specific model has been proposed for microprocessors; it is assumed that a complete test set can be derived for the functional units of data processing by some other techniques.

The main disadvantage of the described classification approach concerns the formalism. All fault models presented above need dedicated specialized test generation procedures. Thus, automatization of test program generation, based on this high-level fault model, is a difficult task.

An ideal case would be to create a small and well-defined fault class with only a few high-level fault models and to build around it a well-standardized and uniform test algorithms. In this thesis, we have chosen high-level decision diagrams for modelling, since their high-level fault model is well suitable to support the development of a uniform and straightforward high-level test generation and fault simulation algorithm.

4.2 HLDD-based Functional Fault Models

Summarizing the presented overview of different approaches to high-level fault modelling in digital systems, let us map now the considered fault types and models into the following generalized HLDD-based fault model, using Definition 2-1 for HLDDs from Chapter 2.2.1.

Definition 4-1. Consider a digital system represented by an HLDD $G^Y = (M, \Gamma, X)$, where the set of nodes $M = M^N \cup M^T$ is partitioned into the subsets of non-terminal nodes M^N and terminal nodes M^T, and the set of variables $X = C \cup D$ is partitioned into the subsets of control variables C (e.g. instruction variables) and data variables D (operands).

Denote by T the test for the digital system represented as a set of test patterns $T = \{X^t\}$, where t is the number of a pattern, and each test pattern $X^t \in T$ can be represented as a concatenation $X^t = C^t.D^t$ of the control pattern C^t (instruction) and data pattern D^t (operand or group of operands).

Let us classify the HLDD-based faults into two general classes: *control faults*, which are related to the non-terminal nodes M^N, and *data faults*, which are related to the terminal nodes M^T.■

Definition 4-2. Introduce the term of *control functional fault model* (CFFM) *of a node* m in HLDD $G^Y = (M, \Gamma, X)$, $m \in M$, as a set of faults $R(m)$ partitioned into subsets $R(m,v) \subset R(m)$ of fault models where $v \in V(x(m))$.

A subset of faults $R(m,v) \subset R(m)$ is called *activated* by a test pattern X^t if X^t activates a path $l(m_0, m^{T,v})$ from the root node $m_0 \in M^N$ to a terminal node $m^{T,v} \in M^T$, so that $x(m) = v$, and $m \in l(m_0, m^{T,v})$. The expected response to the test pattern X^t is $Y = f(m^{T,v})$. If $Y \neq f(m^{T,v})$, there is a functional fault $r \in R(m,v)$ present.

The *control functional fault model for the HLDD* G^Y, is defined as a set $R = \{R(m) \mid m \in M\}$ of all FFM of the nodes in G^Y.■

Since activation of a path $l(m_0, m^{T,v})$ means launching a *working mode* $Y = f(m^{T,v})$ of the system, then testing a functional fault $r \in R(m,v)$ of a nonterminal node $m \in M^N$, means testing if the control signal $x(m) = v$ will not fail at launching this working mode. On the other hand, since this test is executed via data path of the system, then testing the functional fault $r \in R(m,v)$ means testing simultaneously also the terminal node $m^{T,v} \in M^T$, if the data path at this working mode $Y = f(m^{T,v})$, and at the given data specified by X^t, is working correctly.

The functional faults represented by models $R(m)$, and $R(m,v) \subset R(m)$, are called *control faults*. They are not specified here as lists of particular faults, rather we interpret them as some groups of faults. All manipulations with these faults are directed simultaneously to groups of faults, which as the result, reduces the complexity of solving test problems for complex systems, both test generation and fault simulation.

To *activate* the high-level faults $R(m)$, and $R(m,v)$ means activation of some subsets of low-level faults in particular *locations* (subcircuits) in the system. For mapping the high-level fault model $R(m)$ to lower-level structural faults in the *fault activated locations*, with the goal to assess the quality of tests, we will introduce later another functional fault class – *constrained functional fault model*.

Definition 4-3. Introduce the term of *data functional fault model* (DFFM) of the HLDD G^Y is a union of all functional fault models for the HLDD terminal nodes.

$$R_D = \bigcup_{m \in M^T} R(m)$$

The fault models $R(m) \subset R_D$ for terminal nodes $m \in M^T$ can be represented in two possible ways:

(1) as exhaustive (or pseudoexhaustive) fault model $R(m) = V(x(m))$ of the operational block, represented by the node expression $f(m)$, which leads to the exhaustive test of $f(m)$ (as a general case), or to pseudo-exhaustive test;

(2) as partial model $R(m) \subseteq V(x(m))$ (a special case); in this case, the problem of high-level fault modelling will be solved by an hierarchical multi-level approach, e.g. using any fault mapping method between levels.■

Each path of the HLDD designates the behaviour of the system in a specific working mode. The faults having effect on this behaviour are associated in some way with nodes along the path. From that, we can conclude that a control fault will always cause a corruption of the path, which can be modelled as incorrect leaving the path activated by the test. The data faults will corrupt the functions related to terminal nodes.
From above, the following corollaries about the sizes of the functional fault models defined for the HLDDs by Definitions 5-5 and 5-6 follow:

Corollary 4-1. The size $S(R_C)$ of the control functional fault model for the nonterminal part of the HLDD under test, covering the set of *control faults*, can be calculated as

$$S(R_C) = \sum_{m \in M^N} |V(x(m))| \qquad (4\text{-}1)$$

where M^N is the subset of nonterminal nodes in the HLDD.

Corollary 4-2. The higher bound of the size $S_{max}(R_D)$ of the data functional fault model for the terminal (data operation) part of the HLDD under test, covering the set of *data faults*, can be calculated as

$$S_{max}(R_D) = \sum_{m \in M^T} |V(x(m))| \quad (4\text{-}2)$$

where M^T is the subset of terminal nodes in the HLDD. The higher bound is reached only when the exhaustive test will be applied for the functions in terminal nodes. The real size of the functional test can be dramatically reduced when using hierarchical approach to involve also low-level fault modelling of the functional blocks which correspond to terminal nodes.

To make mapping of the high-level control functional fault model $R(m,v)$ to lower structural levels easier, we will introduce in the following subclasses of $R(m,v)$, which are more directly related to the structural aspects of systems under test. Here we will use also previous knowledge about high-level fault modelling in digital systems, discussed in Chapter 4.1.

Definition 4-4. A *control fault* $r(m, v) \in R(m,v)$ of the non-terminal node $m \in M^N$ may belong to the following three fault classes, $r(m, v) \in$ CL-1 $\cup$ CL-2 $\cup$ CL-3.

(1) ***CL-1***: *Missing edge*: $r(m, v \rightarrow \varnothing)$ – the output edge of the node m for $x(m) = v$, $v \in V(x(m))$, is broken, which means no change in the state Y of the system at the working mode $Y = f(m^{T,v})$ under test (it is similar to the logic level SAF $x/0$ for the line x);

(2) ***CL-2***: *Stuck edge:* $r(m, x(m) \equiv v)$ – the output edge of the node m for $x(m) = v$, $v \in V(x(m))$, is always activated (it is similar to the logic level stuck-at fault (SAF) $x/1$ for the line x);

(3) ***CL-3***: *Wrong activation of the edge*: $r(m, v \rightarrow V^*)$ where $V^* \subseteq V(x(m)$ – the fault causes wrong simultaneous activation of a subset of edges. ■

Note the fault class *CL-2* is a subclass of *CL-3*. We introduced it here for optimization of test generation and fault diagnosis purposes.

Table 4-1 Comparison of HLDD-based faults with high-level faults proposed in [83]

<table>
<tr><th>Microprocessor faults (Chapter 4.1, [83])</th><th colspan="2">HLDD faults</th></tr>
<tr><td>F1: No source is selected
F4: No destinations selected
F6: one or more micro-orders not activated;</td><td>CL-1</td><td rowspan="3">Non-terminal nodes</td></tr>
<tr><td>F2, F3, F5, F7, F8: Additional source is selected, stuck-at fault</td><td>CL-2</td></tr>
<tr><td>F2: A wrong source is selected
F3: More than one source is selected
F5, F7, F8: Instead of, or in addition to the selected destination, one or more other destinations are selected; micro-orders are erroneously activated</td><td>CL-3</td></tr>
<tr><td>F9-F14: Data storage, communication or manipulation faults</td><td colspan="2">Terminal nodes</td></tr>
</table>

The functional fault model defined above for HLDDs is related directly to the nodes of the HLDD and is an abstract one. It will have a semantic meaning only when the node has a particular physical interpretation. As an example, in Table 4-1, the mapping of different microprocessor fault classes, the 14 types of faults proposed in [83] and discussed in Chapter 4.1, is shown.

The fault classes *CL-1*, *CL-2* and *CL-3* (general for all non-terminal HLDD nodes) differ in the need of using different data constraints for propagating the fault effects to the observation points.

4.3 Interpretation of HLDD Based Fault Models for microprocessors

Fault classes, defined in previous chapters (Chapter 4.1 and 4.2) considered being used for microprocessors. In order to demonstrate this, we need to return to the model of PARWAN microprocessor, generated in Chapter 3.4. Its partial model is shown in Figure 4-1.

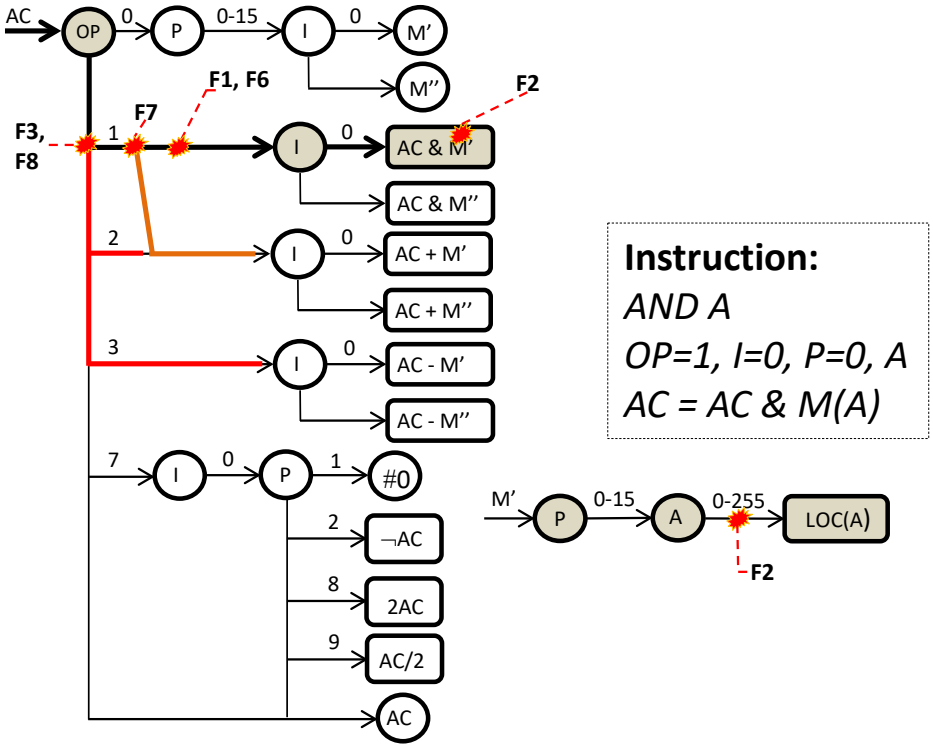


Figure 4-1 Demonstration of different faults in HLDD model of PARWAN

Two graphs G_{AC} and $G_{M'}$, representing the behaviour of accumulator and direct addressing, are derived from the model of PARWAN microprocessor (Chapter 3.4). Variable *AC* represents an accumulator register, *M* denotes the input bus, *OP*, *I* and *P* serve as instruction variables, and variable *A* represents the address in memory. The variables *OP, I, P* and *A* are labelling the internal decision nodes of the HLDDs with their values shown at edges. The terminal nodes are labelled by the variables *AC* and *M* representing the expressions related to particular data manipulation operations of the microprocessor.

Assume, the instruction *AND (OP=1, I=0, P=0, A)* is executed (high lightened on Figure 4-1) with expected result *AC=AC&M*. Table 4-2 illustrates how different high-level faults, defined in Chapter 4.1 and interpreted as HLDD faults, defined in Chapter 4.2.

Table 4-2 Interpretation of microprocessor faults in HLDD

Fault type		**Fault description**	**Interpretation of the fault in HLDD**
Ch.5.5	**HLDD**		
F1, F6	CL-1	No source selected	The output edge 1 of node *OP* is broken. The value of *AC* remains unchanged
F2, F7	CL-2, CL-3	Wrong source selected	Instead of the edge 1 of node *OP* another edge 2 is selected, and the variable *AC* will have the wrong value *AC=AC+M* instead of *AC=AC&M*. Value was read from wrong source address due to fault of output edge of node *A*.
F3, F8	CL-2, CL-3	More than one source selected	Instead of the edge 1 of node *OP* other edges 2 and 3 are selected, and the variable *AC* will have the wrong value *AC=(AC+M)* $\vee$ (AC-M) instead of expected *AC=AC&M* (the wrong value will be technology dependent)

The *addressing fault* of the node *A* in the graph G_M causes activation of the wrong edge instead of the planned edge. As the result, data from the wrong location in memory *LOC*(*A*) is addressed for using it in the operation of the terminal node *AC&M* of the graph G_{AC}. The *operation code fault* of the node *OP* in the ALU graph G_{AC} causes activation of the wrong edge 2 instead of the planned edge 1. As the result, wrong operation *OP = AC+M* is addressed instead of *AC&M* in the related terminal node of the graph G_{AC}. The next variation of *operation code fault* is causing to select two edges of node OP instead of one. The result of such failure will depend on the technology. Finally, the *addressing fault* of the node *OP* of graph G_{AC}, which leads to broken edge 1, will leave the value of *AC* unchanged.

Additionally to classes presented above, we present a novel *hard-to-test* fault class called "*unintended actions*". This fault model is presented in Figure 4-2 on example of abstract module of the processor with four instructions I_0 - I_3. The *n*-bit gate-level implementation of the related hardware consists of four *n*-bit registers - *A*, *B*, *C* and *D*, ALU, decoder and two multiplexers. Its ALU block can execute two operations - *AND* and *OR*. Figure 4-2 depicts also model of this microprocessor using two HLDDs G^C and G^D, for representing the input logic of the registers *C* and *D*.

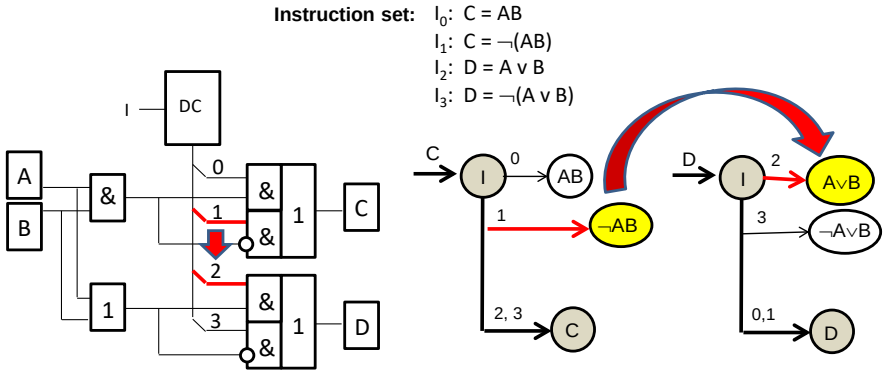


Figure 4-2 Illustration of the behaviour of a hard-to-test fault

Assume, there is a gate level OR-type short between the outputs 1 and 2 of the decoder, i.e. the instruction I_1 implies additional unintended action, related to the instruction I_2, which as the result changes also the content of the register *D*. According to the traditional approach, when testing the instruction I_1, we would read out and check only the content of register *C*, but we will not check the content of register *D*, because it is not involved in the execution of I_1, according to the manual of the microprocessor. In this way, the fault "*wrong change of D*" would escape. Such a fault can be considered as an "unintended action" added to I_1. It would be difficult to catch all similar erroneous "supplements" when testing only the intended, described in manuals, functionality of instructions, because the number of such cases may grow exponentially.

For this type of *high-level faults*, we can adopt a common term of "*hard to test faults*" from the field of gate-level testing, referring to the faults which can be detected by very rare patterns.

4.4 Mapping low-level control faults into HLDD-based functional fault model

In the following, we will analyze the capability of the high-level fault models to cover lower level logic faults in order to demonstrate the usefulness of HLDD modelling. Under logic faults, we imply the following classes of faults: stuck-at faults (SAF), conditional SAF (CSAF), and bridging faults. Let us call this joint fault class as SCB class (SAF, CSAF, bridging).

Consider the block level functional circuit $Y = F(X)$, representing portion of data path and control part of abstract microprocessor, which is also common for most microprocessors. It is illustrated in Figure 4-3 together with its HLDD. The control word *C* (decoder output vector) is a 3-bit Boolean vector variable $C = (c_2,c_1,c_0)$ with decimal values in $v \in V(C) = \{0,1,\ldots,7\}$, which activate the respective working modes $Y = f_v = f(m^{T,v})$. Denote the *k*-th bit of f_v as f_{vk}, $k = \{0,1,\ldots,7\}$. The data part of the unit consists of 8 sub-circuits for calculating f_v which will be selected by the multiplexer sub-circuit. The latter consists of 8 AND_v blocks which are controlled by the output signals $C = (c_2, c_1, c_0,)$ of the control block. Denote the control inputs of each AND_v block as vector variable $C_v = (c_{v2}, c_{v1}, c_{v0})$. Note, each AND_v block consists of 8 AND_{vk} gates for each data bit of the function $f_{v,k}$ and appropriate amount of inverter gates.

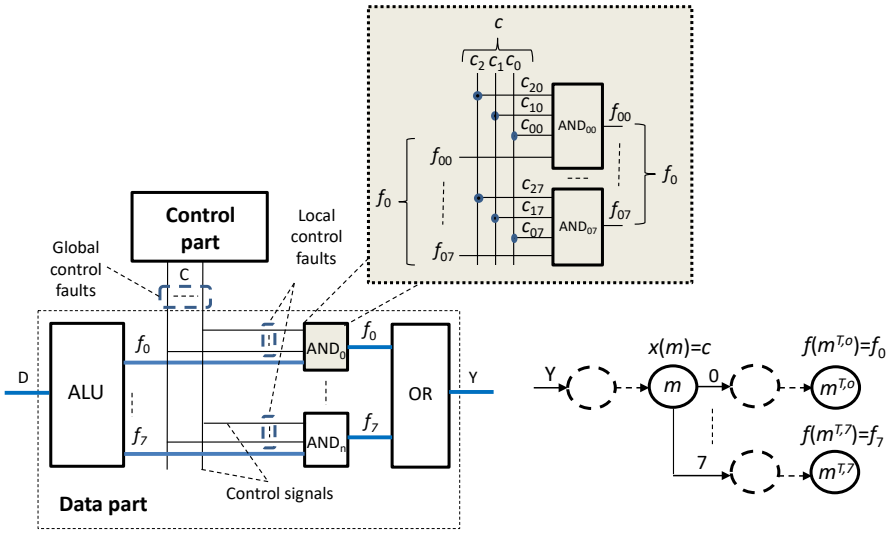


Figure 4-3 Digital system with its HLDD model

Table 4-3 shows the mapping of low-level single structural faults from the class SCB in the circuit of Figure 4-3 into the high-level functional control faults $r(m,v)$, $v \in V(C) = \{0,1,\ldots,7\}$, of HLDD in Figure 4-3. Let us call the faults on the output lines of the control unit as global faults (GF), and the faults on the fan-out branches of the control lines connected with the inputs of the AND gates as local faults (LF). In case of GF, the same fault has impact as a multiple fault on all AND blocks and all AND gates, whereas in case of LF, the fault may have impact either on a single AND block, but propagating to all 8 gates of this block, or only on the inputs of a single AND-gate only. In this example, we will consider LF only at the inputs of 8-bit AND-blocks, as it is shown in Figure 4-3.

Table 4-3 Mapping low level structural faults into high-level functional faults

Fault activation		**Covered structural faults**				
f_i	**Control word**	**Local SAF**	**Global SAF1**	**Global SAF0**	**OR bridge**	**AND bridge**
f_0	000	(0,1),(0,2),(0,4)	1,2,4	∅	∅	∅
f_1	001	(1,0),(1,3),(1,5)	3,5	0	3,5	0
f_2	010	(2,0),(2,3),(2,6)	3,6	0	3,6	0
f_3	011	(3,1),(3,2),(3,7)	7	1,2	7	1,2
f_4	100	(4,0),(4,5),(4,6)	5,6	0	5,6	0
f_5	101	(5,1),(5,4),(5,7)	7	1,4	7	1,4
f_6	110	(6,2),(6,4),(6,7)	7	2,4	7	2,4
f_7	111	(7,3),(7,5),(7,6)	∅	3,5,6	∅	∅

The rows of Table 4-3 correspond to the values v of the activated control faults $r(m,v)$ and to the expression $f_v = f(m^{T,v})$ of the related terminal node. The columns of the sub-table for "covered structural faults" correspond to the faults of SCB partitioned into 5 groups: local SAF, global SAF1, global SAF0, OR type of bridge, AND type of bridge. The entries of Table 4-3 show which high-level functional faults will be evoked

by the low-level structural faults for each activated working mode $Y = f_v = f(m^{T,v})$ of the sub-system. For example, to activate the high-level functional fault $r(m,0)$, the control pattern 000 (c_2, c_1, c_0) should be applied. By applying correct test data for this control pattern, all the low level structural faults depicted in the row f_0 will be covered by the high-level functional fault $r(m,0)$.

To explain the entries in Table 4-3 in more detail, consider the example of applying a control vector $C_3 = (c_2, c_1, c_0,) = 011$ as a test for activating the working mode f_3. The respective row in Table 4-3 is high-lighted in yellow. In column Local SAF we consider only these SAF which coincide with the needed bit values for activating f_3 i.e. $c_2 \equiv 0$, $c_1 \equiv 1$, and $c_0 \equiv 1$. The entry (3,1) means that in case of the local SAF $c_{11} \equiv 1$, the activation of f_3 by $v = 011$ will evoke the erroneous execution f_1 as well (this is the fault type of *several activated edges*), which causes erroneous output value $Y = f_3 \vee f_1$, instead of the expected correct value $Y = f_3$. For the local SAF faults $c_{20} \equiv 0$ and $c_{72} \equiv 1$, we get the erroneous behaviour $Y = f_3 \vee f_2$, noted as (3,2), and $Y = f_3 \vee f_7$, noted as (3,7), respectively.

The global SAF/1, $c_{22} \equiv 1$, will cause execution of f_7, instead of f_3. Similarly, for the case of global SAF/0, of the lines c_1 or c_0, either f_1 or f_2, respectively, will be erroneously executed, instead of expected f_3.

The global bridging faults will cause the following errors in executing of f_3: in case of OR bridge, f_7 will be executed, and in case of AND bridges, either f_1 or f_2 will be executed. The symbol $\varnothing$ in Table 4-3 has the meaning that at these low-level faults no operation is executed. Since the control word is exercised exhaustively, all of the conditional SAF will be detected as well, which corresponds to the *cell-aware testing* concept [84].

Let us compare the reduction in the fault model size for the low and high-level cases. The total number of 790 low-level faults consists of:

- (6 * 8 + 24) * 8 = 576 local SAF (8 AND blocks, each has 3 AND inputs (2 possible SAF each), and 12 inverters (2 possible SAF each), all multiplied by 8 because of the 8-bit data word;
- 3 * 2 = 6 global SAF (three outputs of the control circuit, 2 possible SAF each) ;
- (9 * 2) * 8 =144 bridging faults (9 bridge faults of each type, multiplied by 8 bits of data word);
- 8 * 8 = 64 CSAF (because of exhaustive testing of each of the gates, all high-level functional faults cover all CSAF as well).

The number of all high-level functional faults can be calculated using the formula:

$$S(R(M^N)) = \sum_{m \in M^N} [n_m(n_m + 1)] = S(R(m))$$

Since in this example $|M^N| = 1$, we will have

$$n_m(n_m + 1) = 8 * (8 + 1) = 72.$$

Hence, for this example, the compression of the number of faults when mapping them from low-level to high-level is 790 / 72 = 11 times.

Note, that the number of bit-level functional faults cannot be compared with the number of low-level structural faults in logic circuits, because in the latter case all faults must be processed separately, whereas in the high-level simulation the faults related to bits can be processed in parallel at the word level.

The idea of the presented mapping scheme is based on a hypothetical straightforward implementation where no optimization has been applied. For each behavioural level operation, a dedicated operational block is related, and for controlling the operations, a general multiplexer is introduced. For this hypothetical implementation, we have shown the exact one-to-one mapping between the high-level control faults and the related low-level faults.

Since the HLDD based high-level fault model is inducing the exhaustive exercising of the full behaviour (the set of all instructions), then for any optimization action regarding the implementation, the fault will become to some extent redundant, which will lead also to respective redundancy of the test. Hence, the low-level fault coverage cannot be hurt.

On the other hand, any available information about the real low-level implementation, i.e. about the implemented optimization steps, will give the opportunity to update also the high-level fault model, which will lead in its turn to optimization of the final test program.

4.5 Summary

In this chapter, it was shown that the HLDDs provide *better possibilities of formalising the modelling of high-level behavioural faults in microprocessors* compared to the state-of-the-art approaches.

Three novel high-level fault classes for microprocessors were proposed, which can be considered superclasses over the existing RTL-level fault models for microprocessors. On the other hand, the proposed transition in modelling to *HLDD-based higher levels of abstraction reduces the size of the fault model* by orders of magnitude, compared to the low-level abstractions.

The *proposed high-level fault model separately considers control faults and data manipulation faults,* which are related to internal and terminal nodes of HLDDs, respectively. The control faults are handled exclusively at the high-level, whereas the faults in data paths are processed hierarchically.

The proposed fault model guarantees a high accuracy of testing, which is demonstrated by mapping the new fault classes to lower level faults, and showing that the HLDD-based high-level fault classes fully cover the structural gate-level fault models.

In the first instance, *a novel formalised fault class called 'unintended operational action' was introduced* as a special case of developed fault classes. Using this fault class allows direct targeting of the so-called 'hard-to-detect faults', where traditional methods are not typically focused.

5 SOFTWARE-BASED SELF-TEST GENERATION FOR MICROPROCESSORS

In this chapter, a formalised method of SBST program synthesis for microprocessors is proposed, on the basis of HLDDs and the high-level behavioural fault models developed in the previous chapters. This chapter is based on publications I [56] and II [58].

The contributions of this chapter are summarised as follows:

First, two formal concepts for SBST generation are *proposed*: *conformity test* for the control part, and *scanning test* for the data path of the processor.

Second, *a general SBST program generation concept is described* and its compaction capabilities investigated. *The advantages of the proposed HLDD-based test generation methods* over traditional approaches are established by experimental research.

Experimental results are provided, representing proof of concept. Fault coverage and test overhead properties of a manually synthesized SBST program for the Parwan microprocessor are discussed.

5.1 Principles of software-based self-test generation with HLDD model

The test program synthesis using the HLDD model will cover two levels of the microprocessor: system level, and module level. Each HLDD describes the behaviour of a module, whereas the network of HLDDs represents the behaviour of the whole system. At the module level, the targets of test generation are the nodes of HLDDs, whereas at the system level the targets are the HLDDs themselves. At the system level, the locally generated HLDD (module) tests $T(m)$ will be embedded into the system level test program templates. In other words, the test stimuli for modules will be made controllable and the results of tests will be made observable at the system level.

The test programs are divided into two types: conformity test programs and scanning test programs.

Definition 5-1. *Conformity test* is a test for a non-terminal node of the HLDD, which has the goal to test the control part of the microprocessor. The conformity test will be generated according to the constraints set up for testing non-terminal nodes (Chapter 5.2).

Definition 5-2. *Scanning test* is a test for a terminal node of the HLDD, which has the goal to test the data path of the microprocessor. The scanning test will be generated according to the constraints set up for testing terminal nodes (Chapter 5.3).

5.2 Generation of Conformity Test for Control Part of Microprocessor

Consider an HLDD $G^Y = (M, \Gamma, X)$ with $Y = F(X)$, as a functional model of the instruction set of a given microprocessor, defined formally in Definition 2-1. Here $Y = F(X)$, where $X = C \cup D$, represents instruction format of the microprocessor, where Y denotes destination, C denotes op-code which may be partitioned into sub-fields $C_k \in C$ of the instruction format, and D denotes source which may as well be partitioned several sources $D_k \in D$. The source and destination data variables may refer directly to the registers or may refer to the addressable memory locations. Some examples of mapping between the instruction formats and the HLDD functional variables are depicted in Figure 5-1.

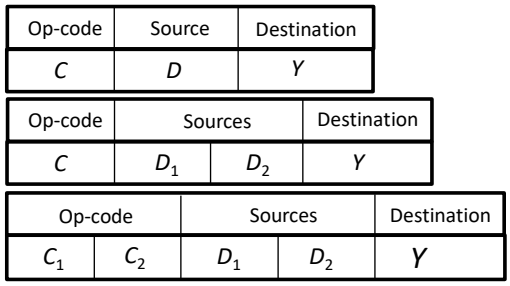


Figure 5-1 Mapping between the instruction formats and the vector functions Y=F(X)

The dependence relationships between variables are described by the network of HLDDs. Examples of HLDDs for different instruction formats are depicted in Figure 3-1 for a single op-code variable, and in Figure 3-6 where the instruction format includes two sources, and the opcode is split into two fields. Figure 3-6 demonstrates how the network of HLDDs reveals the dependence of functional variables for the microprocessor Parwan [57].

According to the concept of HLDD-based testing, the targets of the control tests are not the instructions as a whole, presented by the instruction format, which involves both control and data functions, but the parts of the instruction format. This means that if the opcode C is split into subfields $C_k \in C$, then the control tests will target all subfields C_k one by one. In relation to the hardware of the microprocessor, testing of C_k means to check if the control subfunctions decoded by C_k are correctly selected. In the HLDD to subfield C_k of the instruction format, a nonterminal node $m \in M^N \subset M$ labelled by the variable $x(m) = C_k$, corresponds. Hence, to test if all control subfunctions related to C_k, are correctly selected, the node m in the HLDD for all values $x(m) \in V(x(m))$ has to be tested. According to Definition 4-2, this corresponds to testing the *constrained control functional faults* of $R(m,v) \subset R(m)$, which leads to the following two-step test generation procedure:

Procedure 5-1. Generating a *test instruction* for testing a fault $r \in R(m,v)$

1) Finding a test pattern X^t which activates a path $l(m_0, m^{T,v})$ from the root node $m_0 \in M^N$ to a terminal node $m^{T,v} \in M^T$, so that $x(m) = v$, and $m \in l(m_0, m^{T,v})$; the pattern X^t corresponds to a full opcode C of instruction, which includes the needed value of C_k;
2) Completing the pattern X^t by generating the test data D, so that the constraints of Theorem 5-1 were satisfied.

Theorem 5-1. Any erroneous behavior in terms of the fault classes CL-1, CL-2 and CL-3 (see Chapter 4.3) of the nonterminal node m in HLDD $G_Y = (M, \Gamma, X)$, $m \in M^N \subset M$, and the functional fault model $\{R(m) \mid m \in M^N\}$, will be detected by the test $T = \{X^t\}$, which activates all functional faults $r(m,v) \in R(m)$, $v \in V(x(m))$ for all nonterminal nodes $m \in M^N$ to the respective terminal nodes $m^{T,v} \in M^T(m) \subseteq M^T$, under the following bit-wise constraints:

$$\forall m^T \in M^T(m): \exists X^t \rightarrow \forall k\, [f_k(m^T) \neq \Omega)], \tag{5-1}$$

$$\forall m^{T,i}, m^{T,j} \in M^T(m): \exists X^t \rightarrow \forall k\, [f_k(m^{T,i}) < f_k(m^{T,j})] \tag{5-2}$$

where {Ω = ZERO}, or {Ω = ONE} as the dual case, depending on the implementation technology and *k* denotes the number of data word bit. In case of *i* = *j*, the value of $f_k(m_j)$ in (5-2) refers to the previous state of the variable *Y*. The proof is given in [27] and also [59].

Since for satisfying the constraints of Theorem 5-1 more than one data may be needed, the result of Procedure 5-1 for testing the fault model $R(m,v) \subset R(m)$, in general case, will consist of a control pattern (instruction) $C(m,v)$, and a set of data patterns $D(m,v) = \{D(m,v,r)\}$. This means that in the final test, the instruction $C(m,v)$ will be repeated in the loop *r* times for all data patterns $\{D(m,v,r)\}$. Denote such an elementary test as a concatenation of the control vector and data vector as $T(m,v,r) = C(m,v).D(m,v,r)$ which has the meaning of fully specified instruction from the instruction set of the given microprocessor. Hence, the test which has the goal of testing the constrained control functional fault $R(m,v)$ can be presented as

$$T(m,v) = \{T(m,v,r)\} = \{C(m,v),\{D(m,v,r)\}\}$$

The procedure 5-1 has to be repeated in the loop for all values $v \in V(x(m))$. As the result a test

$$T(m) = \{T(m,v) \mid v \in V(x(m)\}$$

is constructed, which consists of repeating in the loop *v* sub tests $T(m,v)$.

If the HLDD model contains a single non-terminal node *m*, then the test $T(m)$ is the complete conformity test for the given microprocessor. This is the special case of the conformity test, where the *op-code as a whole* is the objective under test. Let us call this type of test as *full conformity test* (FCT).

If the HLDD model consists of more than one non-terminal nodes $|M^N| > 1$, then one higher level loop has to be created to generate the tests for all non-terminal nodes. Let us call this type of test as *partitioned conformity test* (PCT):

$$T(M^N) = \{T(m) \mid m \in M^N\}.$$

The full and partitioned conformity tests will differ in the test length and in the test quality. FCT can be interpreted as exhaustive test, whereas PCT can be interpreted as pseudo-exhaustive test.

The complexity of generating exhaustive FCT will grow exponentially regarding the number of bits in the full opcode. Let μ is the number of nodes in the HLDD, and each node variable has the same number φ of values. Assume that the constraints of Theorem 5-1 can be satisfied by data for a single instruction. Then the length of FCT will be $(\mu \cdot \varphi)^2$, whereas the length of PCT will be $\mu \cdot \varphi^2$, where $\mu \cdot \varphi^2 << (\mu \cdot \varphi)^2$.

The whole test with embedded loops, in general case, can be represented as the following conformity test Algorithm 5-1.

Algorithm 5-1: Conformity test for testing all non-terminal nodes $m \in M^N$ of HLDD

```
1 forall m ∈ M^N do                                   /* test T(M^N) for the fault model R */
2   forall v ∈ V(x(m)) do                             /* test T(m) for the fault model R(m) ⊂ R */
3     forall r do                                     /* test T(m,v) for the fault model R(m,v) ⊂ R(m) */
4       execute C(m,v).D(m,v,r);  /* test instruction T(m,v,r) for the fault
            r ∈ R(m,v) */
5     end
6   end
7 end
```

For implementing the test instructions as a sequence of a self-test program, proper templates should be created in assembly language of the given microprocessor. Therefore, two requirements should be followed – first, prior to each particular execution of the test instruction $T(m,v,r) = C(m,v).D(m,v,r)$ the data operands $D(m,v,r)$ have to be loaded into pre-specified registers. Secondly, the response of the test - the value of the graph functional variable *Y,* must be stored for further analysis.

In Figure 5-2, the instruction set of Parwan microprocessor and behavioural HLDD model of its ALU are depicted. Test for ALU module $T(M^N)$ can be generated using Algorithm 5-1, and represented with Parwan assembly language.

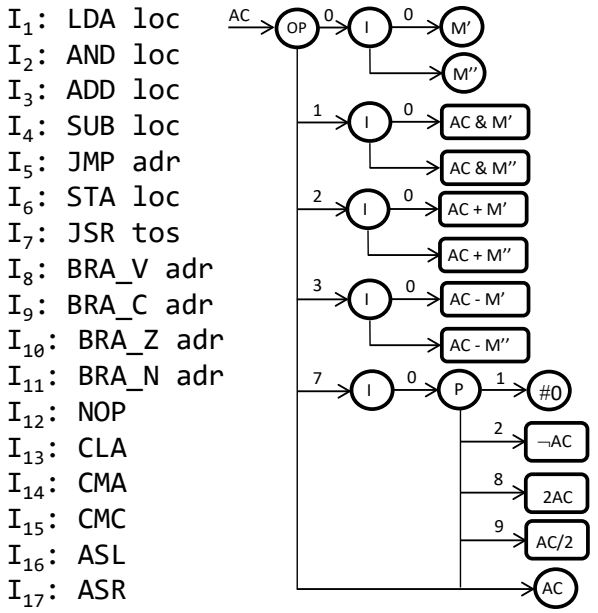


Figure 5-2 Instruction set of Parwan microprocessor and HLDD model of its ALU

An example of test template for testing control nodes of HLDD G_{AC} is presented in Figure 5-3. The template is used here to execute test $T(m)$ for the non-terminal control nodes *OP*, *I* and *P*, representing a subset of Parwan ISA. The test template is a simplified representation of loop, for all instructions I_1, I_2, …, I_{17} (line 1). Before executing test instruction I_v, $v \in \{1, …, 12\}$, internal accumulator register is initialized with data vector *D1* from memory (line 2). Additional data vector *D2* is fetched from source memory location if it's required by instruction (I_2, I_3, I_4) (line 3). Data vectors *D1* and *D2* are generated in a way to satisfy the constraints of Theorem 6-1. The result of instruction is stored to accumulator register and then to destination location in memory $loc(M_D)$ (line 4).

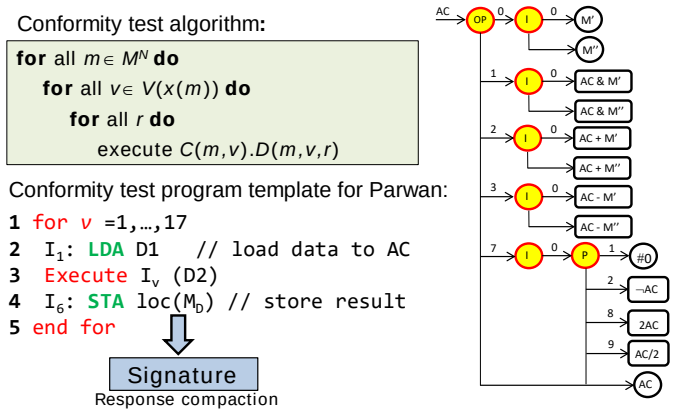


Figure 5-3 Test template for testing non-terminal nodes in the HLDD G_{AC}

5.3 Generation of Scanning Test for Data Part of Microprocessor

For testing data path of the microprocessor, represented as an HLDD model which consists of a network of HLDDs, we have to generate a test for all terminal nodes $m^T \in M^T \subset M$ in each HLDD $G^Y = \{G\}$, $Y \in U$, where the number of graphs is equal to $|U|$ (refer to Definition 2-1).

In Definition 4-3 we introduced for HLDDs $G^Y = \{G\}$ the *data functional fault model* as a union $R_D = \bigcup_{m \in M^T} R(m)$ of all functional fault models $R(m^T)$ of terminal nodes $m^T \in M^T \subset M$, which represent the working modes of the microprocessor $Y = f(m^T)$. Each functional fault $r \in R(m^T)$, similarly to the *conditional SAF* model developed for gate-level testing [61], has a meaning of a constraint (condition) for testing the function $Y = f(m^T)$.

Hence, to test the faults $r \in R(m^T)$ we have to execute in a microprocessor a set of instructions

$$T(m^T) = \{C(m^T).D(m^T, r)\},$$

where the value of $C(m^T)$ (instruction code) remains constant, but the data $D(m^T, r)$ will change and have the values from the set of constraints $R(m^T)$, i.e each constraint $D(m^T, r) \in R(m^T)$ is interpreted as a *data functional fault* $r \in R(m^T)$.

According to Definition 4-3, and discussion in Chapter 4.3, test generation for *data functional faults* of $r \in R(m^T)$ leads to the following two-step procedure:

Procedure 7-2. Generating a *test instruction* for testing a fault $r \in R(m^T)$

1) Finding a test pattern X^t which activates a path $l(m_0, m^T)$ from the root node $m_0 \in M^N$ to the related terminal node $m^T \in M^T$; the pattern X^t corresponds to a full opcode C of the instruction;
2) Completing the pattern X^t by generating a set of test data $R(m^T)$, according to Definition 4-3, either using a hierarchical two-level test pattern generation method to take into account the implementation details of the structure realizing the function $Y = f(m^T)$, or using an implementation free exhaustive or pseudo-exhaustive approach to exercise the function $Y = f(m^T)$.

We call testing of data manipulation functions related to the terminal nodes of HLDDs, as *scanning test*, because the idea of the test is to repeat the same instruction with data retrieved by scanning a given data array.

The full scanning test for the HLDD G^Y with a set of terminal nodes $M^T \subset M$, can be presented as

$$T(M^T) = \{T(m^T) \mid m^T \in M^T\}.$$

The whole test with two embedded loops, in general case, can be represented as the following scanning test Algorithm 5-2.

Algorithm 5-2: Scanning test for testing all terminal nodes $m \in M^T$ of HLDD

```
1 forall m ∈ M^T do                          /* test T(M^T) for the fault model R^T */
2   forall r do                              /* test T(m^T) for the fault model R(m) ⊂ R^T */
3     execute C(m^T).D(m^T,r);               /* test instruction T(m,r) for the fault
        r ∈ R(m^T) */
4   end
5 end
```

Since, before each particular execution of the test instruction $T(m,r) = C(m^T).D(m^T,r)$ the data operands $D(m^T,r) \in R(m^T)$ need to be loaded into pre-specified registers and the response of the test, the value of the graph functional variable Y must be stored for further analysis, then for implementing the test instructions as a sequence of a self-test program, proper templates should be created in assembly language of the given microprocessor.

An example of a test template for testing terminal node $f=AC+M'$ of HLDD G_{AC} is presented in Figure 5-4. The template is used here to execute the test $T(m^T)$ for the node labelled by the addition operation (AC + M') for a given set of operands (AC = D1(j) and M' = D2(j)). The test template is used in a loop for all test data. Before executing the test instruction I_3: ADD D2(j) = AC + D2 (line 3), the microprocessor state (the contents of register AC) is initialized by loading the data vectors D1(j) from memory (line 2).

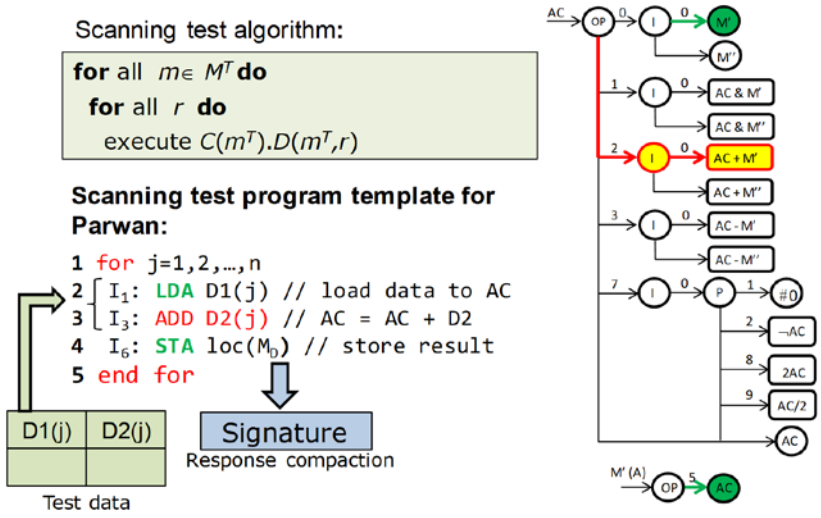


Figure 5-4 Test template for testing in the HLDD G_{AC} the node labelled by working mode (operation) AC+ M

Note, because of the well-defined structure of HLDDs where all instruction level activities of the microprocessor are well represented, the templates for test program

compilation can be synthesized straightforwardly. The information about which instruction is to be used for loading data into register AC can be found in the same HLDD G_{AC} (instruction I_1: $AC = M'$), where M' is source location in memory. In order to store the response to the test from register AC, we find the proper instruction in the graph $G_{M'(A)}$ (instruction I_6: $M'(A) = AC$, line 4).

5.4 Test program generation example

Consider again the example of Parwan microprocessor discussed in Chapter 3.4, where its instruction set (Table 3-2) and HLDD model (Figure 3-6) were introduced. Figure 5-5 demonstrates both conformity and scanning test generation for Parwan microprocessor.

In this example, ALU module of Parwan microprocessors is considered as unit under test. Its HLDD model G_{AC} is represented partially in Figure 5-5a,c. Additional graphs G_N, G_Z, $G_{M'}$ which are indirectly activated during test generation are shown in Figure 5-5b.

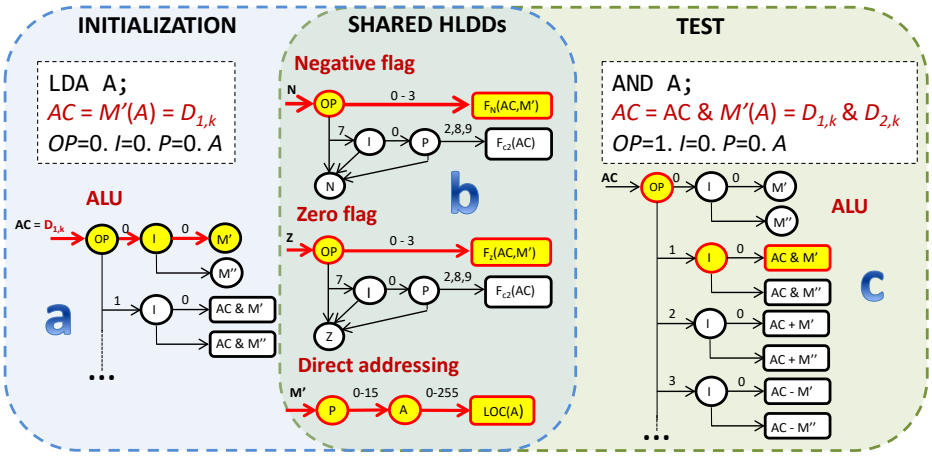


Figure 5-5 Test generation for Parwan microprocessor with shared HLDDs

Table 5-1 illustrates the conformity test for the nodes *OP* and *I* in the HLDD in Figure 5-5c. The test template consists of three instructions. The first instruction initializes the only data register of Parwan – accumulator AC. This procedure is illustrated by highlighted nodes and edges in the HLDD in Figure 5-5a. Nodes of graphs, indirectly activated by this instruction are highlighted in Figure 5-5b. The slot of the second instruction in the template is empty, and should be filled up and updated cycle by cycle with the next instruction under test in the loop during execution of the test program. The instructions to be tested are stored in the respective array in the memory. For this example, the instructions under test are depicted in Table 5-2. The highlighted column in Table 5-2 refers to the control variables under test.

Table 5-1 Conformity test template

No	Instruction mnemonic	Op-code			Data movement		Comments
		OP	I	P	Registers	Operation	
1	LDA A	0	0	P	AC = D_1	AC ← D_1	Initialization
2		Instruction is to be stored from Table 5-2					Instruction under test
3	STA A	5	0	P	AC	AC → M(A)	Storing response

Table 5-2 Instructions to be inserted into the conformity test program template

No	Mnemonic of the instruction	Op-code			Data movement		Result of operation
		OP	I	P	Registers	Flags	
1	LDA A	0	0	P	AC=AC	N,Z ← f(AC)	AC= D_n
2	AND A	1	0	P	AC=D_1	N,Z ← f(AC)	AC= AC&D_2
3	ADD A	2	0	P	AC=D_1	N,Z,C,V ← f(AC, M(A))	AC= AC+D_2
4	SUB A	3	0	P	AC=D_1	N,Z,C,V ← f(AC, M(A))	AC= AC-D_2
5	CLA	7	0	1	AC=D_1	-	AC=0
6	CMA	7	0	2	AC=D_1	N ← f(AC)	AC= ¬D_1
8	ASL	7	0	8	AC=D_1	N,Z,C,V ← f(AC)	AC=2*D_1
9	ASR	7	0	9	AC= D_1	N,Z ← f(AC)	AC= D_1//2

Figure 5-5c illustrates the target of the conformity test to exercise the correct decoding of the control variables *OP*{0,1,2,3,7} and *I*{0,1}. The operations to be executed during the instructions under test are shown in the high-lighted terminal nodes in Figure 5-5c, and the results of the operations are depicted in the last column of Table 5-2. For this test, the data operands D_1 and D_2 are used and stored in the array of data operands in the memory. The data operands should be generated in such a way that the constraints (5-1) and (5-2) in Theorem 5-1 were satisfied.

Table 5-3 Scanning test template to be repeated for the data operands in the memory

No	Mnemonic of the instruction	Op-code			Data movement		Comments
		OP	I	P	Registers	Operation	
1	LDA A	0	0	P	AC = D_1	AC ← M(A)	Initialization
2	ADD A	1	0	P	AC = D_1 & D_2	AC ← AC & M(A)	Instruction under test
3	STA A	5	0	P	AC	AC → M(A)	Storing response

Table 5-3 illustrates the scanning test for the terminal node *AC & M'* in the HLDD on Figure 5-5c. The row, representing instruction under test, is highlighted in Table 5-3. The test template consists of three instructions. First one has the role of initialization of the microprocessor, and similarly to the conformity test, its actions are illustrated by highlighted nodes and edges in the HLDD in Figure 5-5a and Figure 5-5b. However, the scanning test differs from the conformity test. During the initialization procedure, general purpose registers of the microprocessor are filled with prepared data, stored in the corresponding array of data operands in the memory.

Organization of the test programs for the microprocessors based on using the structural-behavioural information given in HLDDs allows compact presentation of the test program templates, arrays of instructions and arrays of data operands. A generalization of such a structure is depicted in Figure 5-6.

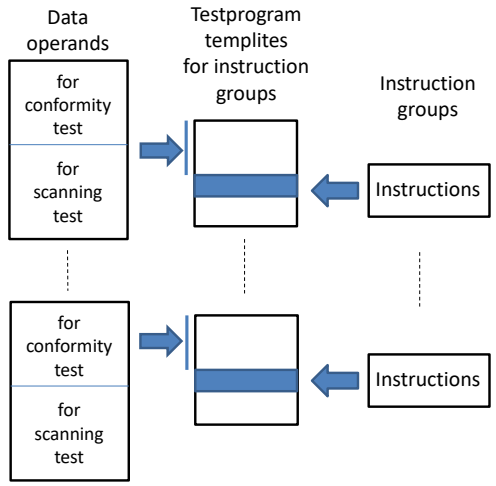


Figure 5-6 A generalized data structure for self-testing of microprocessors

The presented structure contains test data in a similar structure for both conformity and scanning tests. In the case of conformity test, the loop is organized over a subset of instructions whereas the data operands are for this loop the same. In general case, however, several data arrays may be needed to organize higher level loops. In the case of scanning test, the template is filled up by a single instruction whereas the loop is organized over an array of test data operands.

5.5 Discussion on the Properties of Conformity and Scanning tests

The main conception of test generation using HLDDs can be characterized by the following targets and improvements regarding the traditional microprocessor testing methods.

1) improved fault coverage regarding hard-to-test-faults with better diagnostic resolution;
2) reduced probability of fault masking;
3) compactness of the whole test program thanks to its cycle-based organization;

The main idea of the described HLDD-based approach is to test *the behaviour of functional variables* instead of *testing instructions*. With correct test data, test for all functional variables will stress outperform a simple instruction set test, avoiding fault escapes.

As an added value, the result of approach scanning and conformity test approaches, where "*smaller portions*" of the functionality of instructions are targeted in testing, the diagnostic resolution will be better.

Another added value of targeting by tests "smaller portions" of the functionality of microprocessor is the *reduced probability of fault masking*. Consider an example of memory-register-memory I/O operations shown in Figure 5-7, where data is loaded to internal registers, and stored back to memory, using instructions I_1*: LDA Reg Mem (Load data from memory to register)* and I_2*: STA Reg Mem (Store data from register to memory)*. Assume that there is a SAF on control line R_0. The test program with two consequent instruction pairs - *LDA* R_0, *STA* R_0 and *LDA* R_1, *STA* R_1 will pass the test,

despite addressing the incorrect behaviour of addressing in the register bank. As a result, SAF on R_0 control line will escape.

Multiplexor is faulty, causing only one register (R_0) being selected.

```
I1: LDA R0, M(0);
I2: STA R0, M(2);
I3: LDA R1, M(1);
I4: STA R1, M(3);
```

By executing such code, fault is successfully masked

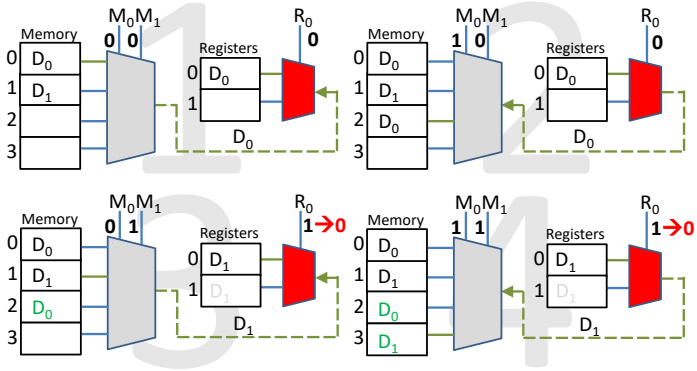


Figure 5-7 Example of fault masking during IO procedure

In order to reduce the probability of fault masking, we are testing the functional variables simultaneously, by initializing all of the available registers prior each test. Example of following technique is shown in Figure 5-8. In this example all registers in register bank are initialized with data during the test, adding observability to every incorrect behaviour, solving fault masking problem. In other words - we keep the initialization and observation sequences constant for the whole test of the variable under test. When recording the test results, we target always a single variable under test. In another case when trying to observe more than one variables, each observation action may cause changes in the state of the processor, which in its turn may activate other possible faults and cause fault masking. This approach is a good example of the trade-off between the test length and test accuracy. We use more processor cycles for constant initialization but, but on the other hand, we reduce the amount of test output data.

```
I1: LDA R0, M(0);
I2: LDA R1, M(1);
I3: STA R0, M(2);
I1: LDA R0, M(0);
I2: LDA R1, M(1);
I4: STA R1, M(3);
```

Instead of testing the instruction, we test the functional variables R_0 and R_1 simultaneously. Fault is successfully detected – masking avoided

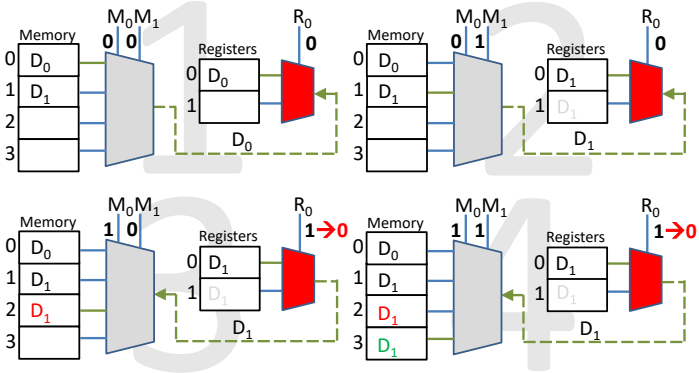


Figure 5-8 Example of fault masking avoidance technique

5.6 Experimental results

As a case study, we generated manually a self-test program for microprocessor Parwan modelled in Chapter 3.4, using fault models proposed in Chapter 4, and HLDD test generation algorithms described in Chapter 5. The obtained fault coverage for every module of MP is outlined in Figure 5-9. The whole test program was simulated by ModelSim to obtain local test data sequences for all modules, and these, in turn, were fault simulated at gate level to get SAF coverage. The comparison of fault coverages with method #1 [85] and method #2 [86] is depicted in Figure 5-9.

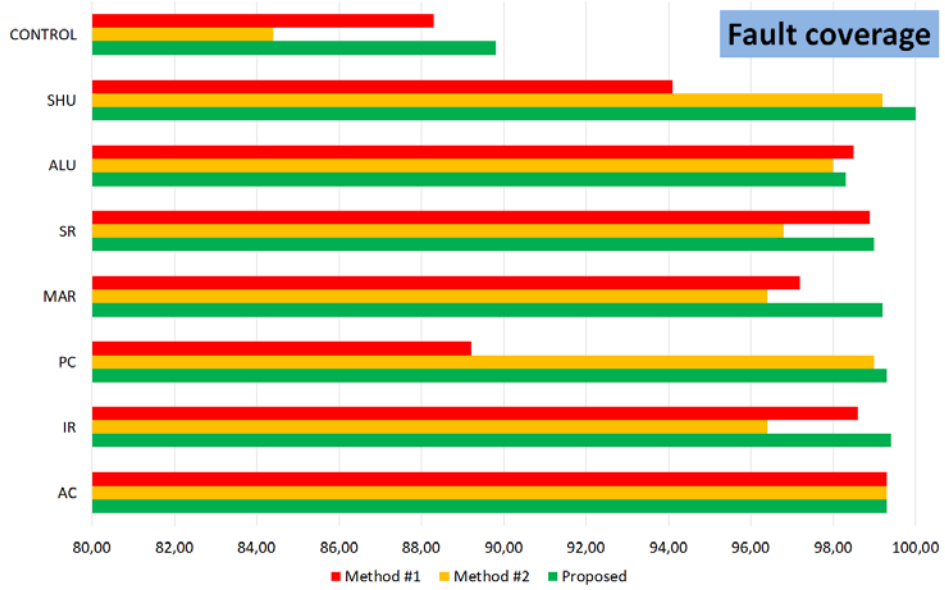


Figure 5-9 Comparison of different test coverages (PARWAN)

To sum up, for seven out of eight modules the proposed method shows advantage regarding test coverage over the previously published results for that processor. The positive impact of the novel high-level fault model can be seen in the higher fault coverage of the control part of MP. The comparison of volumes of test data is presented in Table 5-4. The proposed approach needs 75% fewer test data than in ATIG [86], but the generated program consists of 51% more instructions. However, the latter comparison is not completely fair, since there are single byte and double byte long instructions and such statistics is missing in [85] [86].

Table 5-4 Comparison of test lengths for testing PARWAN processor

Test overhead	**Method #1** [85]	**Method #2** [86]	**Proposed method**
Instruction #	575	189	260
Test data #	unknown	517	132

5.7 Summary

In this chapter, it was shown that the HLDD model provides the possibility of *formalising the SBST program generation process*, which will be the prerequisite and basis for automating this process to be discussed in the next chapter.

Two novel concepts for test generation were proposed, being conformity test and scanning test. Conformity test generation targets the control part of the microprocessor and is driven by non-terminal nodes of the HLDD. Scanning test, designed for the data path, is generated by activation of terminal nodes with predefined data sets.

The test data (operands) for scanning test may be generated in two ways: either applying hierarchical approach using gate-level ATPG-s, if the related implementation details of the data-path are available, or using heuristic functional data or pseudo-exhaustive test patterns, if the implementation details are not available.

Using both algorithms of conformity (Algorithm 5-1) and scanning test (Algorithm 5-2) generation together, it is possible to achieve compact presentation of the test program, which saves memory space, has high fault coverage and better diagnostic capabilities, and reduces the probability of fault masking.

The proposed *method was evaluated using the Parwan microprocessor*. A manually generated test program proved the consistency of the proposed method, by *demonstrating superior fault coverage and test length* over alternative methods.

6 SBST AUTOMATED GENERATION

This chapter presents a framework developed on the basis of the formal methods proposed in previous chapters. The purpose of this framework is to automatically generate SBST programs for microprocessors. The framework consists of three parts.

The first part is responsible for the *automatic synthesis of HLDDs from a description of instruction set architecture*. Requirements for generalisation of ISA description are outlined. Algorithms used for automation of the synthesis are proposed, with examples for MiniMIPS processor.

The second part automatically generates tests on the basis of the HLDD model of a microprocessor.

The third part *automatically converts tests into SBST programs* for microprocessors. Additionally, the equation for test length estimation is proposed. The fault coverage capabilities of the test program for MiniMIPS processor are evaluated.

This chapter is based on publications IV [87] and V [27], where the latest changes in framework were presented.

6.1 Introduction of SBST generation framework

Previously described concepts form the basis of the framework for automated test program generation for microprocessors. The general concept of the framework is shown in Figure 6-1. The framework consists of three modules: HLDD synthesizer, test vector generator, and SBST generator-synthesizer for converting test vectors into test-programs using beforehand prepared test code templates. The transition flow from instruction set to software-based self-test program is demonstrated on 32-bit RISC MiniMIPS microprocessor [88] with instruction set based on MIPS architecture [89].

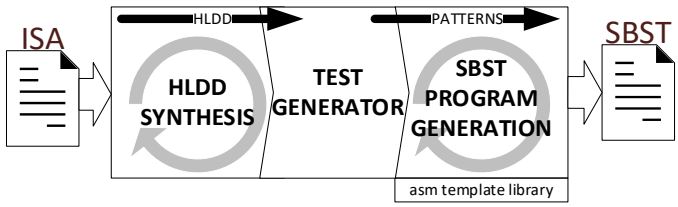


Figure 6-1 Software-Based Self-Test generation framework

6.2 Generalization of instruction set architecture

Instruction set architecture is an abstract representation of a processor, and its description is usually provided in architecture documentation. It usually includes the general description of the general-purpose registers, flags, list of instructions with their names, assembly language syntax, and binary representation. In other words - ISA description holds all the information about the processor necessary to write test programs. The description example of instruction *ADD* is taken from MiniMIPS processor manual [88], and is shown in Figure 6-2. The instruction code (Figure 6-2.A) is divided into fields of fixed widths (in bits) and labels. The mnemonic description (Figure 6-2.C) represents the function and data transition between general-purpose registers. Additionally, assembly syntax (Figure 6-2.B) is given for using addition operation in a program.

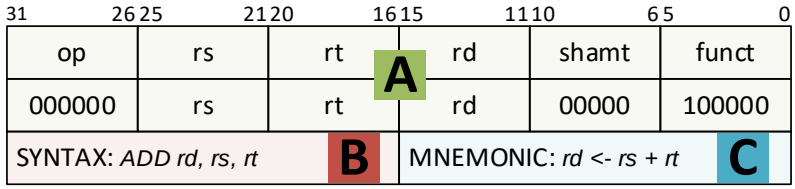


Figure 6-2 ADD instruction description from Minimips manual

The encoding of instruction ADD contains six fields: *op*, *rs*, *rt*, *rd*, *shamt*, *funct*. This specific set of instruction fields defines the format of the instruction. In MiniMIPS architecture, three formats of instructions, shown in Figure 6-3, are used. ADD instruction belongs to the *register* type. Its field *op* holds information about the instruction type. The fields *rs*, *rt* and *rd* are holding the index numbers of general-purpose registers (or system coprocessor registers). The next field – *shamt*, contains the number of shifts for operations with data shifting. The field *funct* distinguishes specific instruction in the *register* instruction domain. The field *imm* of instruction of type *immediate* holds immediate value. The last but not least, field *address* of *jump* instructions contains the memory address for jump operations.

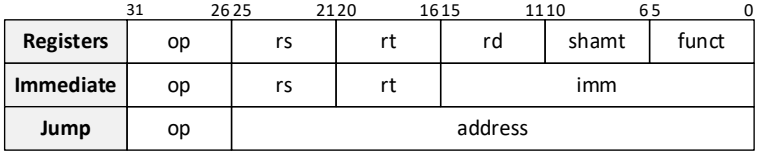


Figure 6-3 MiniMIPS instruction formats

The behavioural model of the processor can be built on the basis of information about instruction format and encoding. Additional information, obtained from ISA description, can help to append details to the model. For example, the information about changes in program counter can provide the basis for modelling the behaviour of program counter unit.

In order to process ISA automatically, it should be represented in a machine-readable way. We suggest bringing the ISA description to common ground manually, as it was shown in Chapter 3.2. As a replacement for functional tables, we outline the format – *ISDL (Instruction Set Description Language)* to generalize the description of miniMIPS ISA. *ISDL* is developed on the basis of the format previously proposed in [87]. It implies that each instruction is described using specific syntax, emphasizing its functionality, and extracting functional variables. For each type of instruction field, the specific syntax is envisaged in *ISDL* format. *ISDL* supports four types of instruction fields - operation code, register, data and constant. Their syntax is shown in Table 6-1, where placeholders are surrounded by *"< >"*.

Table 6-1 ISDL *syntax for instruction fields*

Field Type	***ISDL* Syntax**	**Description**
Operation code	op:<*name*>=<*width*>b<*value*>	
Register	<*direction*>:<*name*>=<*width*>b	<*direction*> can be *in* or *out*
Data	data:<*name*>=<*width*>b	Field for immediate value
Constant	con:<*name*>=<*width*>b<*value*>	

In addition to instruction word fields, the special syntax to describe instruction function, assembly and changes in the program counter are defined. Instruction mnemonic reflects its functionality and is important in test data generation. Assembly syntax will be used in test program generation process and will be described further. Function field should be highlighted with *{ }*, assembly code field with *[ ]*, and program counter field with *( )*. Function description is kept in a separate library, but a line in *ISDL* should have a link to it via function name. Additionally, this field can keep the information about flags, or data movement between registers.

An example of this description is shown in Figure 6-4, where *ADD* instruction has the link to the function in library defined with *{ADD(rs, rt)}*. Assembly field provided as *[add $rd, $rs, $rt]*, keeps the assembly code for test program generation stage. *rd*, *rs* and *rt* are the placeholders for general purpose register indexes. Program counter field is optional because this information can be hidden, and not included into documentation. The overall understanding of the architecture of the processor under test can help to isolate this information from the manual. In case of *ADD* instruction, branches do not happen, and program counter should increment in the way that next instruction is fetched. Thus, value of *PC* is incremented by four (bytes) i.e *(PC + 4),* in other words at the end of execution, *PC* should contain the address of the next instruction word.

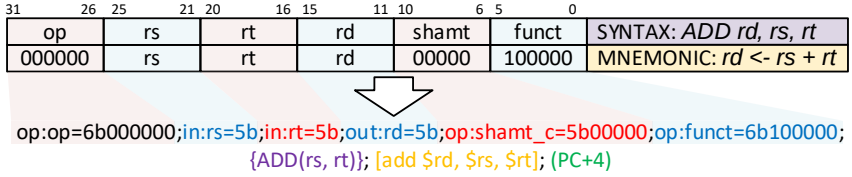


Figure 6-4 ADD instruction converted to ISDL

By using the proposed guidelines, it is possible to bring the instruction *ADD* (Figure 6-2) of miniMIPS to the following entry in *ISDL* format as shown in Figure 6-4. Instruction *ADD* belongs to the *registers* format of instruction according to Figure 6-3, and belongs to group of ALU-related instructions because of the value *000000* in its *op* field. The field *funct* with the value *100000* defines this instruction as *ADD* among other instructions with the same value in *op* field. Register related fields (*rs* - source register, *rt* - target register and *rd* - destination register) of the instruction code should hold the general purpose register numbers. The subset of mimiMIPS instructions brought in *ISDL* format is shown in Figure 6-5.

```
op:op=6b000000; con:rs_c=5b00000; in:rt=5b; out:rd=5b; data:shamt=5b;op:funct=6b000000; {SL(rt,shamt)}; [sll rd, rt, shamt]; (PC+4)
op:op=6b000000; con:rs_c=5b00000; in:rt=5b; out:rd=5b; data:shamt=5b;op:funct=6b000010; {SR(rt,shamt)}; [srl rd, rt, shamt]; (PC+4)
op:op=6b000000; con:rs_c=5b00000; in:rt=5b; out:rd=5b; data:shamt=5b;op:funct=6b000011; {SRA(rt,shamt)}; [sra rd, rt, shamt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b000100; {SL(rt,rs)}; [sll rd, rt, rs]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b000110; {SR(rt,rs)}; [srlv rd, rt, rs]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b000111; {SRA(rt,rs)}; [srav rd, rt, rs]; (PC+4)
op:op=6b000000; con:rs_c=5b00000; con:rt_c=5b00000; out:rd=5b; con:shamt_c=5b00000;op:funct=6b010000; {rd=regHI}; [mfhi rd]; (PC+4)
op:op=6b000000; in:rs=5b; con:rt_c=5b00000; con:rd_c=5b00000; con:shamt_c=5b00000;op:funct=6b010001; {regHI=rs}; [mthi rs]; (PC+4)
op:op=6b000000; con:rs_c=5b00000; con:rt_c=5b00000; out:rd=5b; con:shamt_c=5b00000;op:funct=6b010010; {rd=regLO}; [mflo rd]; (PC+4)
op:op=6b000000; in:rs=5b; con:rt_c=5b00000; con:rd_c=5b00000; con:shamt_c=5b00000;op:funct=6b010011; {regLO=rs}; [mtlo rs]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; con:rd_c=5b00000; con:shamt_c=5b00000;op:funct=6b011000; {regLO=[rt*rs]0,31; regHI=(rt*rs)31,63}; [mult rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; con:rd_c=5b00000; con:shamt_c=5b00000;op:funct=6b011001; {regLO=[rt*rs]0,31; regHI=(rt*rs)31,63}; [multu rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100000; {ADD(rs,rt)}; [add rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100001; {ADDU(rs,rt)}; [addu rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100010; {SUB(rs,rt)}; [sub rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100011; {SUBU(rs,rt)}; [subu rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100100; {AND(rs,rt)}; [and rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100101; {OR(rs,rt)}; [or rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100110; {XOR(rs,rt)}; [xor rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b100111; {NOR(rs,rt}; [nor rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b101010; {LESS(rs,rt)}; [slt rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; in:rt=5b; out:rd=5b; con:shamt_c=5b00000;op:funct=6b101011; {LESSU(rs,rt)}; [sltu rd, rs, rt]; (PC+4)
op:op=6b000000; in:rs=5b; con:rt_c=5b00000; con:rd_c=5b00000; con:shamt_c=5b00000; op:funct=6b001000; {}; [jr rs]; (rs)
op:op=6b000000; in:rs=5b; con:rt_c=5b00000; out:rd=5b; con:shamt_c=5b00000; op:funct=6b001001; {rd = PC+4, PC = rs}; [jalr rd, rs]; (rs)
op:op=6b010000; con:rs_c=5b00000; out:rt=5b; con:cs=5b00000; con:rd_c=5b00000; op:funct=6b000000; {rt = COP0}; [mfc0 rt, 0]; (PC+4)
op:op=6b010000; con:rs_c=5b00100; in:rt=5b; con:cs=5b00000; con:rd_c=5b00000; op:funct=6b000000; {regCOP0 = rt}; [mtc0 rt, 0]; (PC+4)
op:op=6b000001; in:rs=5b; con:rt_c=5b00000; data:offset=16b; {FLAG:LTZ(rs)}; [bltz rs, offset];(?(LTZ), PC+(offset<<2), PC+4)
op:op=6b000001; in:rs=5b; con:rt_c=5b00001; data:offset=16b; {FLAG:GEZ(rs)}; [bgez rs, offset];(?(GEZ), PC+(offset<<2), PC+4)
op:op=6b000100; in:rs=5b; in:rt=5b; data:offset=16b; {FLAG:EQ(rs,rt)}; [beq rs, rt, offset];(?(EQ), PC+(offset<<2), PC+4)
op:op=6b000101; in:rs=5b; in:rt=5b; data:offset=16b; {FLAG:NE(rs,rt)}; [bne rs, rt, offset];(?(NE), PC+(offset<<2), PC+4)
op:op=6b000110; in:rs=5b; con:rt_c=5b00000; data:offset=16b; {FLAG:LEZ(rs)}; [blez rs, offset];(?(LEZ), PC+(offset<<2), PC+4)
op:op=6b000111; in:rs=5b; con:rt_c=5b00000; data:offset=16b; {FLAG:GTZ(rs)}; [bgtz rs, offset];(?(GTZ), PC+(offset<<2), PC+4)
op:op=6b001000; in:rs=5b; out:rt=5b; data:immediate=16b; {ADD(rs,immediate)}; [addi rt, rs, immediate]; (PC+4)
op:op=6b001001; in:rs=5b; out:rt=5b; data:immediate=16b; {ADDU(rs,immediate)}; [addiu rt, rs, immediate]; (PC+4)
op:op=6b001010; in:rs=5b; out:rt=5b; data:immediate=16b; {LESS(rs,immediate)}; [slti rt, rs, immediate]; (PC+4)
op:op=6b001011; in:rs=5b; out:rt=5b; data:immediate=16b; {LESSU(rs,immediate)}; [sltiu rt, rs, immediate]; (PC+4)
op:op=6b001100; in:rs=5b; out:rt=5b; data:immediate=16b; {AND(rs,immediate)}; [andi rt, rs, immediate]; (PC+4)
op:op=6b001101; in:rs=5b; out:rt=5b; data:immediate=16b; {OR(rs,immediate)}; [ori rt, rs, immediate]; (PC+4)
op:op=6b001110; in:rs=5b; out:rt=5b; data:immediate=16b; {XOR(rs,immediate)}; [xori rt, rs, immediate]; (PC+4)
op:op=6b001111; con:rs_c=5b00000; out:rt=5b; data:immediate=16b; {rt = immdiate<<16 | rt}; [lui rt, immediate]; (PC+4)
op:op=6b100011; data:base=5b; out:rt=5b; data:offset=16b; {rt = memory[base+offset]}; [lw rt, offset(base)]; (PC+4)
op:op=6b101011; data:base=5b; in:rt=5b; data:offset=16b; {memory[base+offset] = rt}; [sw rt, offset(base)]; (PC+4)
op:op=6b110000; data:base=5b; con:cs_c=5b00000; data:offset=16b; {COP0 = memory[base+offset]}; [lwc0 rt, offset(base)]; (PC+4)
op:op=6b111000; data:base=5b; con:cs_c=5b00000; data:offset=16b; {memory[base+offset] = COP0}; [lwc0 rt, offset(base)]; (PC+4)
op:op=6b000010; data:instrindex=26b; {}; [j instrindex]; (instrindex)op:op=6b000011; data:instrindex=26b; {GPR31 = PC+4}; [j instrindex]; (instrindex)
```

Figure 6-5 Subset of miniMIPS instruction set in ISDL *format*

6.3 HLDD synthesis from *ISDL* description

The Correctly composed ISA description in *ISDL* format holds needed data to build HLDD diagram, representing the behaviour of the system (or it's part) under test. The solution for building HLDD graph model is based on the framework proposed in [50]. This framework provides the functionality to create, edit and import HLDD graphs. Figure 6-6 is a class diagram demonstrating the structure of behavioural model. *ModelingDomain* is the most top element in this metamodel that is used to collect *ModelingObjects*.

The domain (*ModelingDomain*) is typically a microprocessor whereas the objects (*ModelingObject*) are the units of a microprocessor.

Any *ModelingObject* may have a number of inputs that are implemented as *variables*.

Variable x is defined with the name and the width in bits. The modeling object is represented by the set of *GraphVariables*.

The possible values of the *GraphVariable* are modelled as terminal nodes *Termination* of the *graph* that are assigned to this *GraphVariable*.

Termination has a link to *Variable* that defines its value. The value of the *Termination* is defined by the object derived from the *Variable* class - *Input*, *GraphVariable*, *Function* and *Constant* objects.

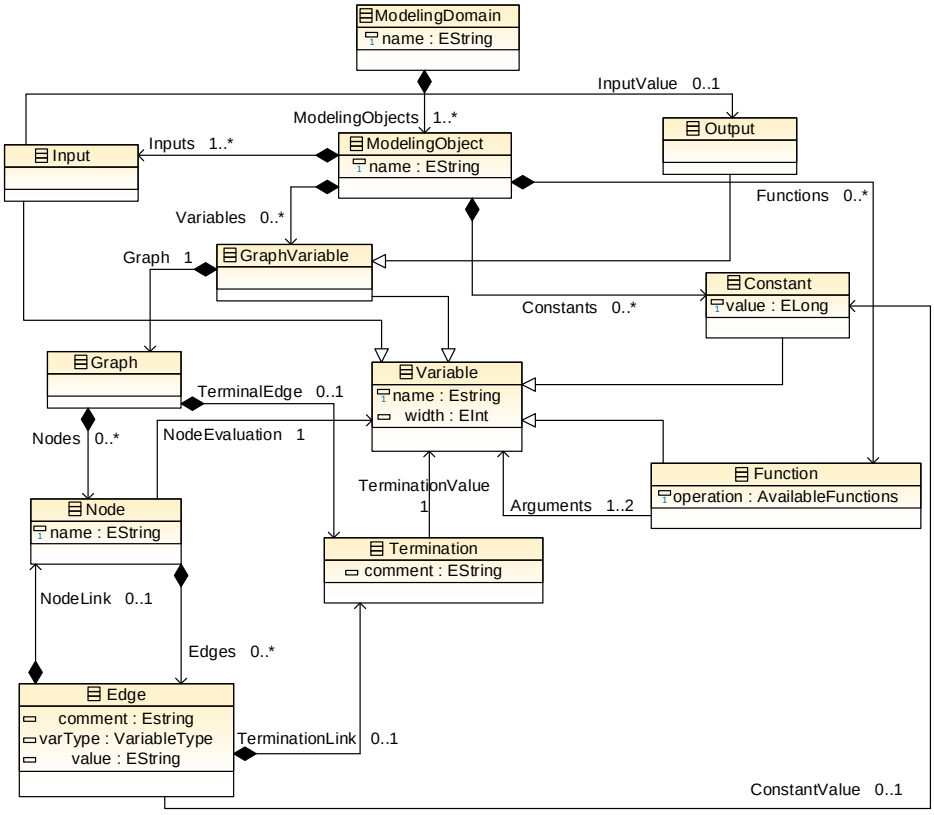


Figure 6-6 Metamodel of HLDD

Graph object has containment link to the nodes that belong to this graph.

Node has a link to the variable that contains the possible values of the node. Same nodes may be connected by more than one edge.

Edge may lead to the next non-terminal node (*NodeLink*) or to the terminal node (*TerminationLink*). The transition value of the edge may also be specified by the *ConstantValue* link to the predefined constant.

A *Function* is an object that defines the operations with variables. The *function* has a field for selecting an operation from a list of supported functions (*AvailableFunctions*). This list can be easily extended to support any operations (bitwise operations, logic operations, etc.). The arguments to the function are specified by the *Arguments* link that selects variables from the list of predefined variables.

Based on the described abstraction it is possible to process the instruction set data, given in *ISDL*, and to synthesize a graph mirroring the behaviour of the processor or its part. The meta-model shown in Figure 6-6 is general and can be applied to modelling microprocessors. In this case, the *ModelingDomain* represents the model of the processor, composed of different units (*ModelingObjects*) i.e ALU, PC, register bank etc. Each processor unit is represented with separate graph, which output or *GraphVariable* represents output register or flag. Instruction fields representing operational codes will become *Nodes* of the graph. The links - *Edges* between nodes are represented by the instruction field values. *Terminations* of the graph will be the functions following the execution of instructions.

The implementation of HLDD generation in frames of the proposed framework is represented with the set of algorithms – Algorithms 6-1, 6-2, 6-3, 6-4, 6-5, with links to the meta-model shown in Figure 6-6.

Algorithm 6-1: HLDD Synthesis - top level

```
1  PROCESSOR = new ModelingDomain();
2  FILE = open(isdl);
3  forall LINE in FILE do
4  |  IDATA = parse(LINE);
5  |  if IDATA has PC field then
6  |  |  if PC exists in PROCESSOR then
7  |  |  |  execute populateGraph(PC, pcsubset(IDATA));          /* Algorithm 6-3 */
8  |  |  else
9  |  |  |  Graph PC = new Graph(PC, pcsubset(IDATA));           /* Algorithm 6-2 */
10 |  |  end
11 |  end
12 |  if IDATA has no OUT field then
13 |  |  OUT_ = parseMnemonic(IDATA);
14 |  |  if OUT_ is an out register then
15 |  |  |  OUT = OUT_;
16 |  |  end
17 |  end
18 |  if OUT.type is general-purpose-register then
19 |  |  forall GPR in GPR-LIST do
20 |  |  |  if GPR exists in PROCESSOR then
21 |  |  |  |  execute populateGraph(GPR.idx, subset(IDATA));   /* Algorithm 6-3 */
22 |  |  |  else
23 |  |  |  |  Graph OUT = new Graph(GPR, subset(IDATA));       /* Algorithm 6-2 */
24 |  |  |  end
25 |  |  end
26 |  else
27 |  |  if OUT.name exists in PROCESSOR then
28 |  |  |  execute populateGraph(OUT.name, subset(IDATA));     /* Algorithm 6-3 */
29 |  |  else
30 |  |  |  Graph OUT = new Graph(OUT, subset(IDATA));          /* Algorithm 6-2 */
31 |  |  end
32 |  end
33 end
```

Algorithm 6-1 describes the top level of the HLDD synthesis framework. With the main cycle, the program is walking through the instruction list given in *ISDL* format, by reading it line by line. Each line is parsed in order to obtain key information about the instruction format and fields. First, the detection of information about PC is made. This information is optional, but in case such information exists, the PC graph is created (Algorithm 6-2) or populated (Algorithm 6-3) with new nodes. The next step is to find register information in instruction line, especially output register. In case output register does not have its separate field in the instruction word, the program continues to search for it in *function* field, highlighted by *{ }*. This field can hold the information about indirect activation of register. For example, instruction *MTHI* (move data to internal register *regHI*) has following description - *op:op=6b000000; in:rs=5b; con:rt_c=5b00000; con:rd_c=5b00000; con:shamt_c=5b00000; op:funct=6b010001;* ***{regHI=rs}****; [mthi rs]; (PC+4).* The register field with direction *"out"* does not exist within the list of instruction fields. However, function field ***{regHI=rs}*** holds information, that data from input register *rs* will move to *register regHI* during execution of *MTHI*

instruction. In this case, *regHI* becomes an *OUT* register. Last conditional operator in Algorithm 6-1 is checking if there were register fields with direction *out* within instruction field list. The new graph is created (Algorithm 6-2) or populated (Algorithm 6-3) if there is such register field. These operations are executed in cycle for each general purpose register of the processor. In order to use uniform function for creation or filling different types of graphs, the functions – *pcsubset(), subset()* and *brsubset()* are introduced. They are filtering the instruction data from *ISDL* line, needed to build a specific type of graph.

Algorithm 6-2: HLDD Synthesis - Graph constructor

```
1  GV = New GraphVariable(NAME);
2  PREV = GV;
3  if GV is general-purpose-register then
4  |  execute addNode(OUT);                          /* Algorithm 6-4 */
5  end
6  forall OP in PARAMETERS do
7  |  execute addNode(OP);                           /* Algorithm 6-4 */
8  end
9  forall CON in PARAMETERS do
10 |  execute addNode(CON);                          /* Algorithm 6-4 */
11 end
12 forall IN and OUT in PARAMETERS do
13 |  execute addNode(IN);                           /* Algorithm 6-4 */
14 end
15 forall DATA in PARAMETERS do
16 |  execute addNode(DATA);                         /* Algorithm 6-4 */
17 end
18 if FLAG is listed in PARAMETERS then
19 |  execute addNode(FLAG);                         /* Algorithm 6-4 */
20 else
21 |  execute addTermination(PARAMETERS);           /* Algorithm 6-5 */
22 end
```

Algorithm 6-2 describes the functionality of the constructor for *Graph* object. The constructor is called if the graph does not exist already in *ModelingDomain*. Otherwise, the existing graph is populated (Algorithm 6-3) with given instruction data. The first thing created by graph constructor is *GraphVariable*, which is an output of the graph. Then, instruction fields are attached to the graph as nodes (Algorithm 6-4) with edges in the following order: *op* (opcode field) > *con* (constant) > *in* (register) > *data* (immediate) > *flag* > *termination* (function or constant). Terminations are added (Algorithm 6-5) as leaves, and are holding constants or links to the functions in the library. *PREV* variable is a link between edges and nodes.

Algorithm 6-3 describes the graph population procedure. It is similar to graph construction (Algorithm 6-2) but has many conditional operators in order to check the existence of nodes prior adding them to the graph. Attachment on new edges, nodes and terminations is same as in Algorithm 6-2.

Algorithm 6-3: HLDD Synthesis - Graph filler

```
1  if GV is general-purpose-register then
2  |  PREV = getRootNode().EDGE(1);
3  else
4  |  PREV = GV;
5  end
6  forall OP in PARAMETERS do
7  |  if (PREV.next.name = OP.name) ∧ (PREV.value = OP.value) then
8  |  |  PREV = Node(OP).edge;
9  |  else
10 |  |  execute addNode(OP);                          /* Algorithm 6-4 */
11 |  end
12 end
13 forall CON in PARAMETERS do
14 |  if (PREV.next.name = CON.name) ∧ (PREV.value = CON.value) then
15 |  |  PREV = Node(CON).edge;
16 |  else
17 |  |  execute addNode(CON);                          /* Algorithm 6-4 */
18 |  end
19 end
20 forall IN in PARAMETERS do
21 |  if (PREV.next.name = IN.name) ∧ (PREV.value = IN.value) then
22 |  |  PREV = Node(IN).edge;
23 |  else
24 |  |  execute addNode(IN);                           /* Algorithm 6-4 */
25 |  end
26 end
27 forall DATA in PARAMETERS do
28 |  if (PREV.next.name = DATA.name) ∧ (PREV.value = DATA.value) then
29 |  |  PREV = Node(DATA).edge;
30 |  else
31 |  |  execute addNode(DATA);                         /* Algorithm 6-4 */
32 |  end
33 end
34 if FLAG is listed in PARAMETERS then
35 |  forall FLAG in PARAMETERS do
36 |  |  if (PREV.next.name = FLAG.name) ∧ (PREV.value = FLAG.value) then
37 |  |  |  PREV = Node(FLAG).edge;
38 |  |  else
39 |  |  |  execute addNode(FLAG);                     /* Algorithm 6-4 */
40 |  |  end
41 |  end
42 else
43 |  execute addTermination(PARAMETERS);              /* Algorithm 6-5 */
44 end
```

Algorithm 6-4 describes the procedure of new node addition to the graph. Depending on the type of node, two different paths exist for flag nodes and other nodes. In case of flag node, Boolean edges (1, 0) are being added to it with two different terminations. The terminations are taken from the *PC* field in instruction description in *ISDL* format – *(?(FLAG), A, B)*, where *FLAG* is a node, and *A* is termination with termination link *1(true)*, and *B* is with *0(false)*.

In addition, when flag node is added to the graph, the *ModelingDomain* is checked for the presence of flag-related graph. If there is no such graph, it's created with subset

of data, filtered with *brsubset()*. In case another non-flag node is added, its object is generated and the edge is linked to it.

Algorithm 6-4: HLDD Synthesis - node addition

```
1  if OUT then
2  |  NODE = New Node(OUT?idx);
3  |  PREV.next = NODE;
4  |  EDGE = newEdge(NODE, 0);
5  |  PREV = EDGE;
6  |  execute addTermination(GPR, idx);                    /* Algorithm 6-5 */
7  |  EDGE = newEdge(NODE, 1);
8  |  PREV = EDGE;
9  else
10 |  if FLAG then
11 |  |  NODE = New Node(name);
12 |  |  PREV.next = NODE;
13 |  |  EDGE = new Edge(NODE, 1);
14 |  |  PREV = EDGE;
15 |  |  execute addTermination(condition = 1);             /* Algorithm 6-5 */
16 |  |  EDGE = new Edge(NODE,0);
17 |  |  PREV = EDGE;
18 |  |  execute addTermination(condition = 0);             /* Algorithm 6-5 */
19 |  |  if NODE.name exists in PROCESSOR then
20 |  |  |  execute populateGraph(NODE.name,brsubset(PARAMETERS)
21 |  |  else
22 |  |  |  Graph NODE = New Graph(NODE.name, brsubset(PARAMETERS)
23 |  |  end
24 |  else
25 |  |  NODE = New Node(name);
26 |  |  EDGE = New Edge(NODE, value);
27 |  |  PREV.next = NODE;
28 |  |  PREV = EDGE;
29 |  end
30 end
```

Finally, the termination addition flow is described in Algorithm 6-5. It's a short procedure, which is linking existing edge to the created Termination object.

Algorithm 6-5: HLDD Synthesis - termination addition

```
1 TERMINATION = New Termination(PARAMETERS);
2 PREV.next = TERMINATION;
```

Let us have an example of synthesis of HLDD (based on the algorithms listed previously) for a single instruction *ADD* of miniMIPS processor, shown previously in Figure 8.2 and Figure 6-4. First, the information about program counter is checked in *ISDL* entry:

op:op=6b000000;in:rs=5b;in:rt=5b;out:rd=5b;con:shamt_c=5b00000;

op:funct=6b100000;{ADD(rs, rt)}; [add $rd, $rs, $rt]; (PC+4).

The presence of field *(PC+4)* indicates, that program counter is increased by 4 bytes after instruction execution. This means that graph for *PC* (Figure 8.7) can be built. PC will become a *GraphVariable*. Then operation code fields – *op* and *funct* are attached to it with corresponding node links – *000000* and *100000*, provided in the description. Next, constant *shamt_c* with edge *00000* as attached. Constant is followed by register placeholders – *rs*, *rt* and *rd*. For representative means, multiple edges for different

register indexes are grouped into one edge with a range from *0 to 31*. Finally, termination with formula for PC calculation is attached to the graph. The full model of miniMIPS PC unit is demonstrated in Figure 6-11.

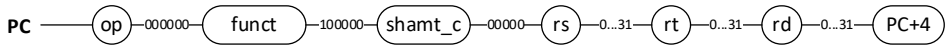


Figure 6-7 HLDD graph for PC on basis of ADD *instruction description in* ISDL *format*

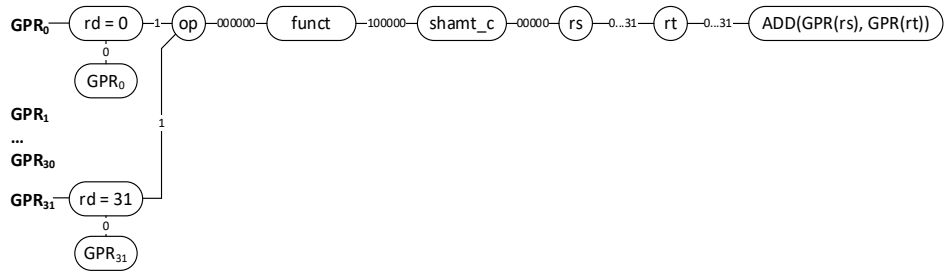


Figure 6-8 HLDD graph for GPR_i on basis of ADD instruction description

After actions with PC graph, *ADD* instruction description is analysed further to find if there is a register with direction *out*. In case of *ADD* instruction, such register is *rd*. This field of the instruction is keeping the index of register, in which the result of the addition will be stored. MiniMIPS has thirty two general purpose registers (GPR_0 – GPR_{31}), therefore the same amount of graphs will be synthesized. Using the same algorithm for graph synthesis, but with different subset of data, graph for each general purpose register GPR_i is built (Figure 6-8). For representative means, set of graphs G_{GPRi} is united into one graph with multiple graph variables (GPR_0 - GPR_{31}). Field *rd* becomes a root node, which is selecting the destination register for result of operation *ADD*. Fields *op*, *funct*, *shamt_c*, *rs* and *rt* become nodes of the graph with corresponding edges. The function (signed addition) is added to the terminal node of the graph. Function *ADD(GPR(rs), GPR(rt)),* should be also added to the library of functions, in order to use it further for test data generation. In general, the value of graph variable GPR_i should have a result of function in termination node, if the correct path is activated. Full representation for ALU unit of miniMIPS is shown in Figure 6-10.

Since register fields *rs*, *rd*, and *rt* are not representing data, but the indexes of registers in general-purpose register bank, special graphs are built. These graphs, shown in Figure 6-9, will explain the data movement between registers.

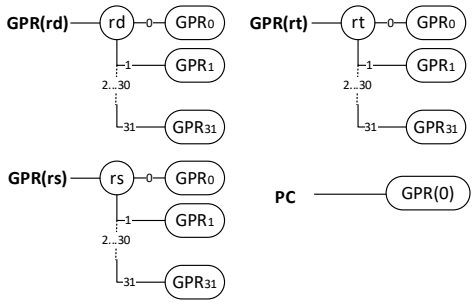


Figure 6-9 HLDD graphs for GPR registers

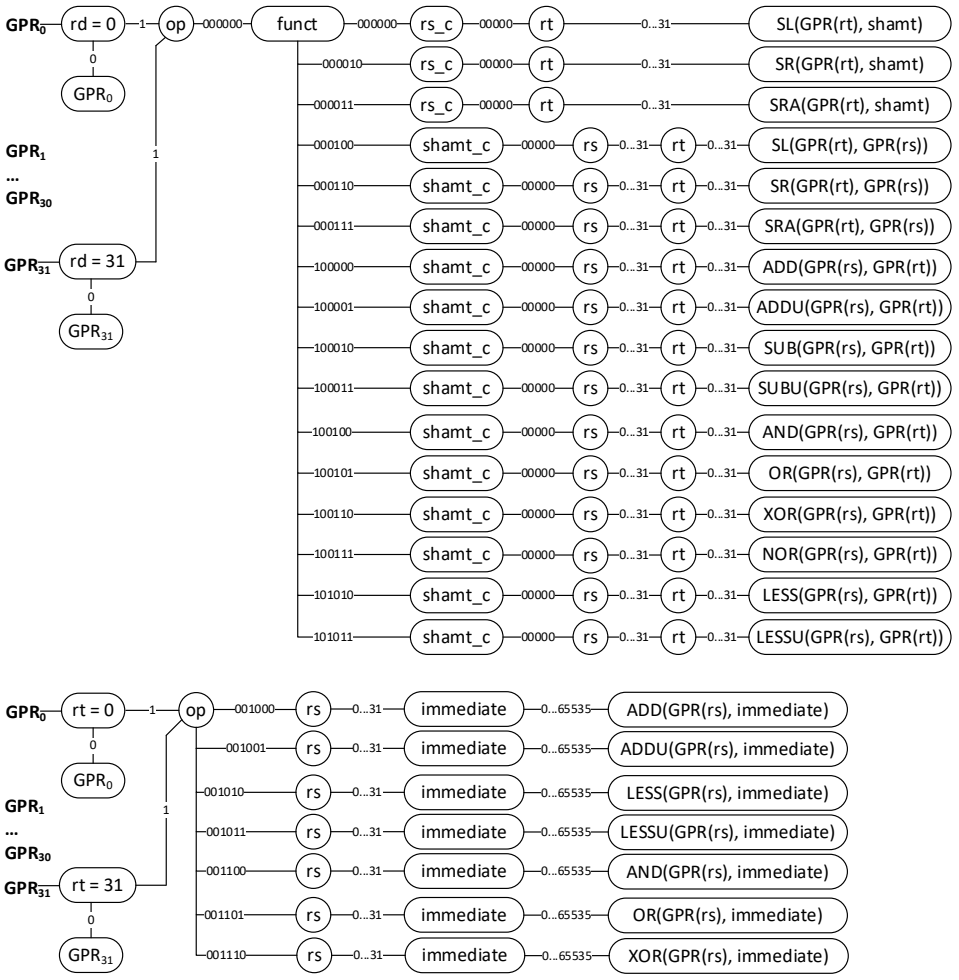


Figure 6-10 HLDD graphs for miniMIPS ALU

By executing the synthesis software, multiple graphs were built based on miniMIPS ISA description in *isdl* format. In Figure 6-10 the graph synthesized for ALU unit is shown. As it was described previously, this graph can be read from right to left, starting with termination, representing the function behind instruction code. The value calculated by function is stored to the general purpose register GPR_i, selected by *rd* or *rt*. The result of the function depends on the data, which is stored to *input* GPR*i* selected by instruction fields *rs* and *rt*. Moreover, the function itself is selected mainly by the values of *op* and *funct* codes and some constants – *rs_c*, *shamt_c*. The HLDD graphs representing different units of miniMIPS processor are shown in Figure 6-11, Figure 6-10, Figure 6-9, Figure 6-12 and Figure 6-13.

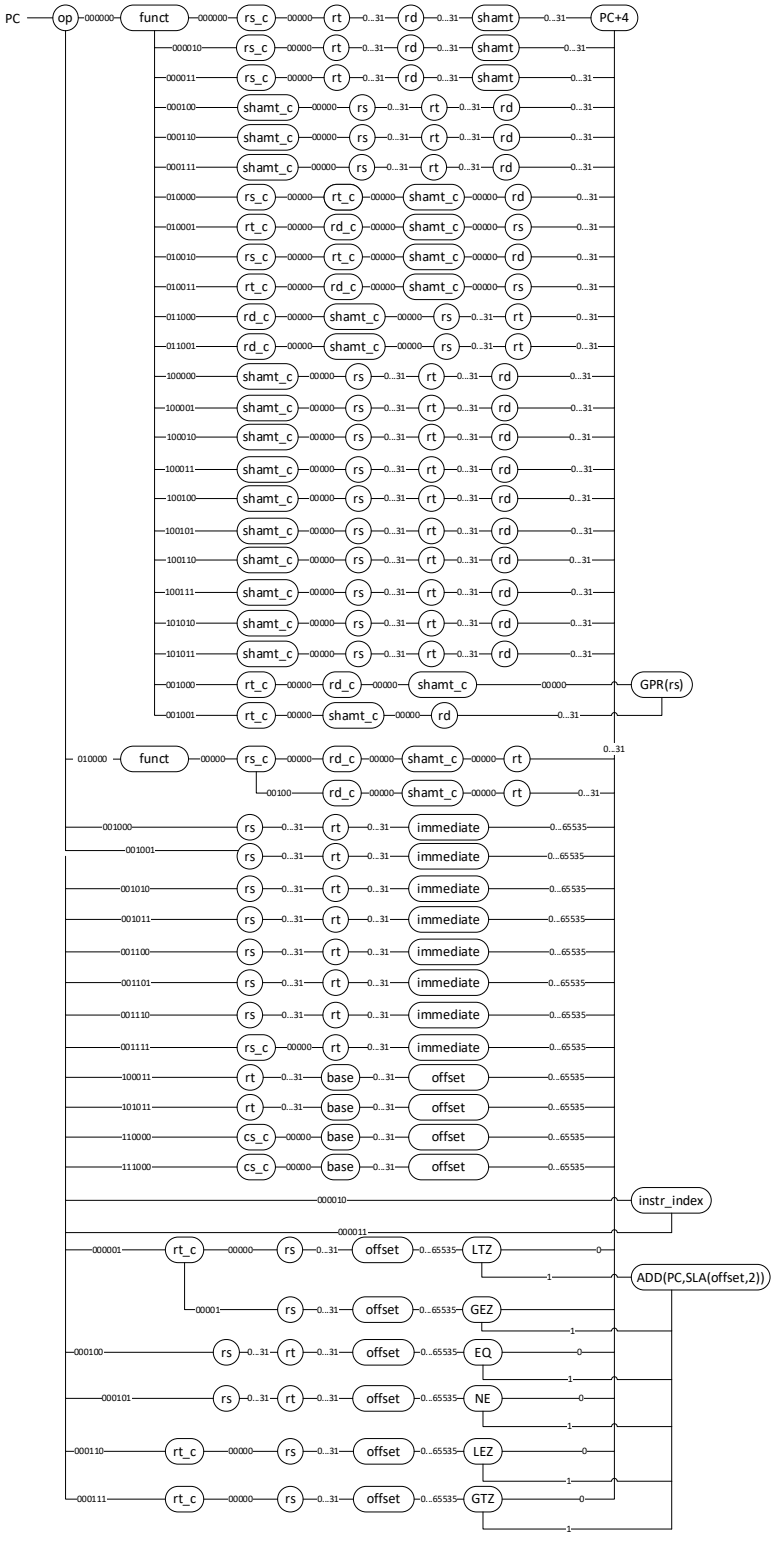


Figure 6-11 HLDD graph for miniMIPS program counter

Special treatment is needed for branching instructions. These instructions rely on the values in flag registers. In case flag registers are not defined in ISA manual, it is possible to define dummy flag variables in order to model branching behaviour. The nodes - *LTZ, GEZ, EQ, NE, LEZ, GTZ*, representing flags are added to the PC graph as shown in Figure 6-11. For each node separate graph is synthesized (Figure 6-12) based on the information given in *function {}* and *PC ()* fields of instruction description. For example, function *LTZ(GPR(rs))* is returning a Boolean value (1 or 0), if data in register *rs* is less than zero. Depending on that program counter value is chosen.

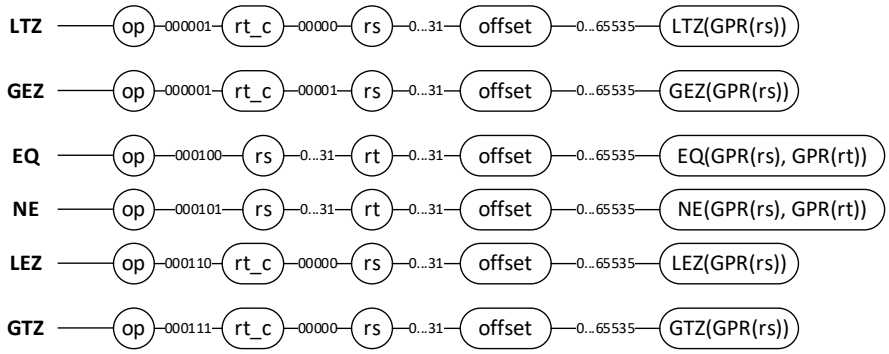


Figure 6-12 HLDD graphs for miniMIPS flags

Last but not least, the graph describing register-memory data movement is shown in Figure 6-13. This graph represents how data is being stored from GPR's and coprocessor registers to memory and loaded back to registers. Calculation of the address for loading and storing the data into memory is made using *ADD* function with *data* type fields *base* and *offset*.

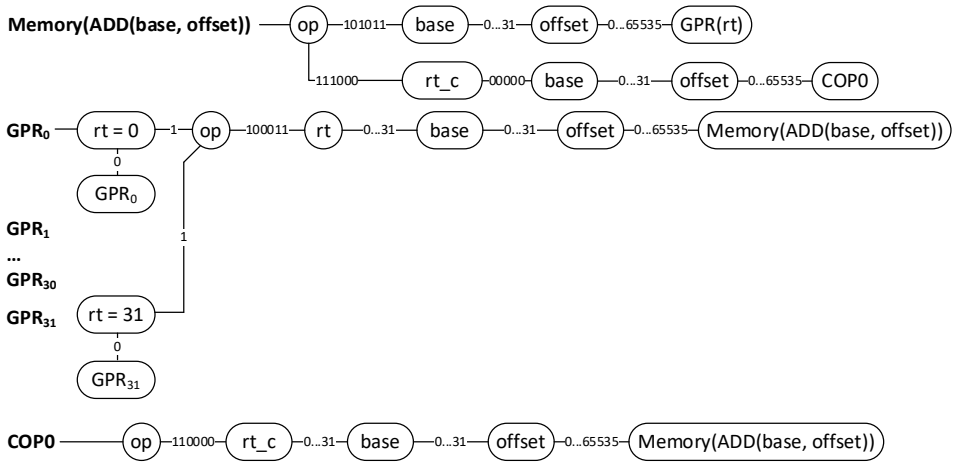


Figure 6-13 HLDD graphs for miniMIPS memory-register data movement

6.4 Test synthesis from HLDD

Once the HLDD graph model for the given processor is constructed, it can be used as a basis for test generation. The result of test generation is a set of test patterns for testing structural entities of the processor. The procedure of test generation mainly revolves around walking through the graph, activating its nodes and also generating specific test data patterns. In this thesis, the information regarding test data generation is omitted but can be found in [27].

An example of test generation is built on basis of miniMIPS *ADD* instruction, modelled in ALU HLDD. By walking through the graph, three lists (Figure 6-14) are being filled at the same time:

PATHLIST – holds the information about the path of nodes from *graphVariable* to termination node. The syntax is following – P#=$name_1$'$width_1$,…,$name_n$'$width_n$, where # is a placeholder for index, *name* is the name of node, and *width* is the numeric value corresponding to the node link (edge).

DATALIST – list of test data, which will be loaded into registers during test program execution. Syntax is following – D#:*binary_list*, where # is a placeholder for enumeration, and *binary_list* is a list of numeric values.

TESTLIST – list of tests, generated by walking through nodes. The syntax is following – P#:*test*:D#, where the # is a placeholder for enumeration, *P* is for addressing PATHLIST, *D* is for addressing *DATALIST*, and *test* is a binary representation of node link values.

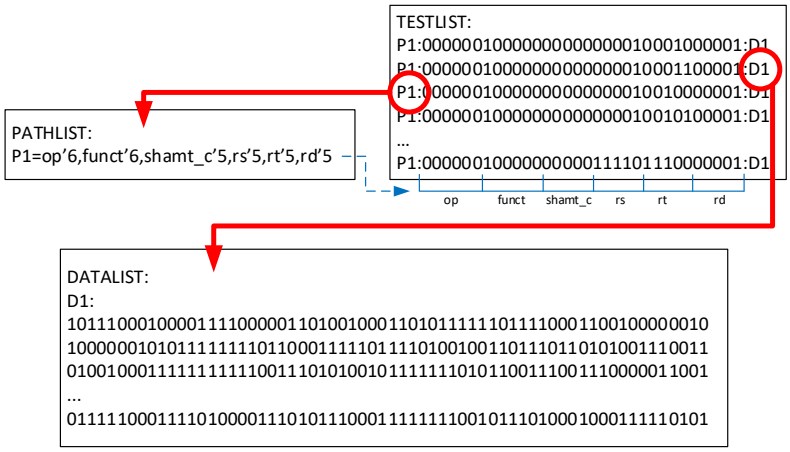


Figure 6-14 Example of test generation

ADD instruction path includes nodes – *op*, *funct*, *shamt_c*, *rs*, *rt*, *rd*. This path is added to *PATHLIST* as *P1*, and it will be valid for all instructions with the same set of nodes involved. Next, data for testing control part is generated and is aggregated into *DATALIST* subset – *D1*. It is generated by applying techniques explained in [27]. Finally, the test is generated to activate the path of nodes from graph variable to termination node. It is generated in pseudo-exhaustive manner and added to the *TESTLIST*, with addition of link to path description – *P1*, and test data set – *D1*. The test itself is represented in binary form, equivalent to instruction word, and is holding all information needed for further test program generation.

6.5 SBST program generation

The targets of test generation for a microprocessor using the HLDD model are not the instructions themselves, each of them taken as a whole as in traditional cases. Instead, the targets are small functional entities represented by the nodes of HLDDs. The terminal nodes represent selected data path functional entities (sub-circuits of ALU), and the nonterminal nodes represent the selected control functional entities related to the subfields of instruction words. Since the HLDD nodes as test targets represent smaller functional units than the instructions as a whole, it makes possible to use pseudo-exhaustive testing of the processor control part and to cope in this way better with the complexity of the test problem. Instead of full exhaustive testing of all operation codes, we test (pseudo)exhaustively its independent parts, guided by the HLDD internal nodes. For testing terminal nodes, we use test data generated for ALU at the gate level. From above, two approaches of testing, different for terminal and nonterminal nodes, result: conformity test for the control part (internal HLDD nodes), and scanning test for data path (terminal HLDD nodes).

The task of SBST generator is to decode patterns Figure 6-14, obtained from test generator into assembly instructions. This is done by using predefined templates stored in the assembly code library. As a result, the test program, composed from code templates is made. It can be edited further, in order to improve the fault coverage. Specific program code parts, important from the test coverage perspective, but hard to generate automatically, can be added manually.

The test program length in cycles (considering single instruction per cycle) can be calculated using following formula:

$$L_T = (I + S) \times (T_C + T_S) + V, \tag{6-1}$$

where *I* and *S* are representing the number of cycles for initialization and store procedures, depending on number of internal registers, needed to be loaded with data. T_C is the number of cycles required by conformity tests, depending on the amount of paths to be activated in HLDD model. T_S is the number of cycles for scanning test, heavily dependent on the amount of patterns for testing data path. Overhead *V*, needed to support functionality and compactness of the program, described in Chapter 5.4 is added.

The SBST generation process is shown in general in Figure 6-15. Generated SBST program can be logically divided in memory into two parts: *test program* and *test data*. *Test data* area is filled with data, given in *DATALIST* (Figure 6-15.A). The test *program* part is generated (Figure 6-15.B) based on test patterns obtained during test generation step. Generation of *test program* can be divided into three parts – initialization, test and store. The initialization part is loading test data into registers, and store part is saving obtained results back to memory. The *test* combines the instruction fields from the library into the full instruction code. In Figure 6-15.C, a subset of generated test program for testing control part is shown.

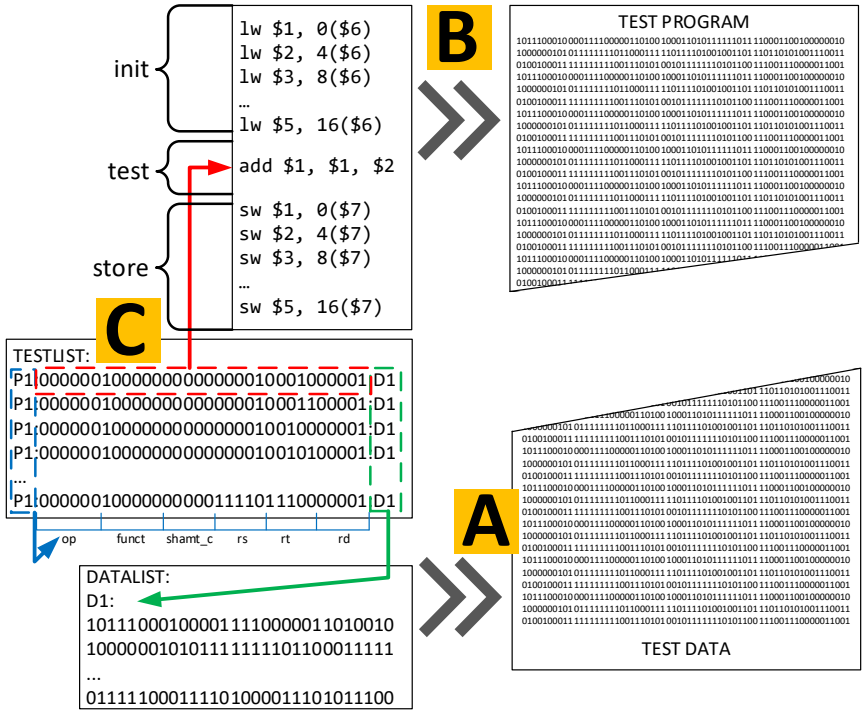


Figure 6-15 SBST program generation flow

The first part of the code represents initialization process. Every general-purpose register is loaded with data before testing each instruction so that to avoid fault masking [87]. Then, the test part is being generated. In section C of Figure 6-15, a test pattern string, retrieved from the *TESTLIST* is shown. Since the instruction fields are known (using the link to the *PATHLIST*), the test program generator can retrieve corresponding assembly instruction from the library. Indexes of register operands (*$*) are manipulated depending on the test pattern.

Test program generation is strongly affected by the modelling level made in previous steps. The more details can be extracted from instruction set architecture, the more detailed test program can be generated. The specific behaviour of the processor can be hidden or even made invisible from the ISA point of view. Hence, the simple list of instructions may be not sufficient to cover the realistic structural details.

The exact fault coverage can be calculated by gate-level fault simulation. The undetected gate-level faults may belong to the class of redundant faults. Otherwise, to detect these faults, low-level ATPGs can be used for generating additional test operands.

6.6 Environment for experiments and results

In order to obtain fault coverage metrics of generated SBST program, fault simulation tools are used. For our experiments, we are using test environment based on combination of Mentor Graphics ModelSim [90] simulator and Synopsys TetraMAX [91] fault simulator. The organization of our SBST evaluation framework is shown in Figure 6-16.

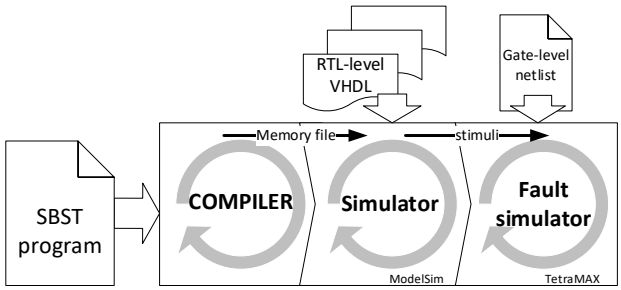


Figure 6-16 SBST program evaluation framework

A generated test program is compiled for the processor under test. In case of miniMIPS, memory file is generated as a result of compilation. This file is added to the processor RTL description. Using ModelSim software it's possible to simulate the behaviour of the processor during the execution of the test program. Command - *dumpports*, in ModelSim, allows storing the stimuli, obtained from inputs and outputs of the processor during simulation, in unified *vcd* format.

Stimuli data, obtained during simulation step is loaded into fault simulator as a list of test vectors. Gate-level netlist and technology library are loaded to TetraMAX fault simulator. Then, it is possible to allocate the module of the processor for fault simulation with patterns obtained during the previous step. Then, SAF are added to the model under test. Finally, fault simulation is running and fault coverage results are reported. Fault coverage results obtained from simulation with automatically generated SBST program for miniMIPS are shown in Table 6-2.

Table 6-2 miniMIPS fault coverage with generated SBST

Instance name	#faults	Fault coverage %
U1_pf (Fetch stage)	2182	59,01
U2_ei (PC)	1608	80,53
U3_di (Decode stage)	7472	78,10
U4_ex (Execute stage, ALU, MULT)	211136	96,42
U5_mem (Memory access stage)	2870	56,46
U6_renvoi (Bypass unit)	3738	78,18
U7_banc (Register bank)	43584	82,19
U8_syscop (Coprocessor)	6930	79,14
U9_bus_ctrl (Bus control)	2028	79,58
U10_predict (Branch prediction)	21286	53,06
Total	302986	89,46

The test data generation was targeting mostly the execute stage of MiniMIPS, which includes the biggest part of the processor core – ALU including two multiplication units. As a result, decent coverage of 96,42% of faults in U4_ex is achieved (Table 6-3), where 97,58% of faults in ALU were covered.

Table 6-3 Fault coverage of execute stage in details

Instance name	#faults	Fault coverage %
U4_ex (Execute stage)	211136	96,42
ALU	203576	97,58

The significant loss in fault coverage is due to the fact, that current model does not cover pipeline behaviour. This affects fault coverage in every stage of the pipeline, thereafter the fault coverage result for the pipeline-related control logic of execute stage (U4_ex) is only 86.32%.

This problem can be solved by populating test program with specific patterns of code, which will activate the faults in pipeline control and memory addressing unit [25]. Nevertheless, we find the obtained coverage as decent (89.46%), keeping in mind that it was obtained by automatically generated SBST program, i.e. effort was given only for composing the list of instructions in *isdl* format.

Table 6-4 Fault coverage results of different SBST methods (MiniMIPS ex & ALU)

Method	ATPG [92]	HLDD	#1 [9]	#2 [24]	#3 [92]
U4_ex	99,93	96,42	96,37	84,12	97,62
ALU	97,58	97,58	-	97,78	98,67

In table Table 6-4 the results of fault coverage for execute stage and ALU of MiniMIPS processor are shown.

The first method (#1) [9] is capable to automatically generate SBST programs. It is relying on using ATPG and SAT solver for pattern generation, which are generating test program by applying constraints during structural and functional analysis of the circuit under test. An additional hardware module is used for observing the inputs and outputs of processor during in-field application, in order to discover the incorrect behaviour. However, fault coverage result for *ex* stage including ALU is less compared to our HLDD-based approach. The difference, although is very small, and the result can be considered equal.

Next approach #2 [24] is similar to the method we have proposed in this thesis. SBST generation is based on instruction set model of the processor, additionally applying developed mechanisms in order to increase the fault coverage for pipeline control and memory addressing. Fault coverage result for execute stage of the pipeline is relatively low. The reason for that can be overall low attention to the control part of the processor under test. However, ALU coverage is superior to HLDD-based approach, outperforming it by 0,2%.

In the question of fault coverage percentage for execute stage of the pipeline and ALU, method named ATIG (#5) [92] has shown the best result. However, the fault coverage given in this work is computed by considering only structurally testable faults, i.e., structurally untestable faults are collapsed, making comparison unfair. This method is based on test generation using structural information (gate-level). Additionally, this method is relying on modification in RTL design in order to obtain the best observability of the system aiming to find best test patterns. However, fault coverage is measured on "clean" system.

We have compared fault coverage of execute stage and ALU only, since at the current stage we were targeting this part of the processor mostly, generating test data for it. All other modules of the processor were tested with the same data operands, delivering decent, but not superior or at least competitive coverage.

6.7 Summary

This chapter demonstrated that the previously proposed formal methods can be used to *automate the SBST generation process*. The implementation of these methods was conducted under the heading of the proposed framework. The structure and algorithms are presented in detail in order to leave the possibility for reproducing.

A formal methodology was automated for the synthesis of HLDD models for microprocessors on the basis of the instruction set description. A procedure of manual transformation of the instruction set architecture into machine-readable format was developed, which formed the basis of the automated synthesis of the graph model of microprocessors.

The second part of the framework used the *formal basis for the test generation process*. The implementation of these methods is described in detail, including the traversing through graphs to generate scanning and conformity tests.

The generated tests, obtained by traversing the model and applying the constraints developed in previous chapters, are *automatically converted into test programs* which can be loaded and executed on the processor under test.

As a proof-of-concept, an automated generation of test program targeting the ALU of MiniMIPS was demonstrated. The efficiency of generated test programs, in terms of test coverage, was evaluated separately for different modules of the microprocessor. The obtained fault coverage results for the execute stage and ALU module are *competitive or even superior to other state-of-the-art approaches.*

The main advantage of the newly developed methods is that the tests, generated for microprocessors on the basis of only high-level instruction set information, have the same quality as state-of-the-art methods which use additional information about the low-level implementation details.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The aim of this thesis is to propose a novel formalised methodology for modelling microprocessors on the basis of the instruction set description, with the goal of automated synthesis of SBST programs. The proposed modelling approach extends the theory of HLDDs to support behavioural modelling of the microprocessor, describing its high-level structure and components. The automation of the SBST program development is based on the topological analysis of HLDDs and solving the high-level data constraints deduced from the model of the microprocessor under test.

The main contributions of the presented work are summarised below.

- *The methodology for high-level modelling of microprocessors on the basis of its instruction set architectures*

The methodology is based on the theory of HLDDs to model microprocessors, based on the descriptions of their instruction set architectures. The use of high-level behavioural descriptions of microprocessors as input data for model synthesis makes this approach more scalable than other state-of-the-art approaches, which are based on lower-level descriptions. The proposed formal method introduces the important property of one-to-one mapping between the modelled processor and its corresponding high-level functionality. This allows the HLDD model to be used as a checklist for high-level test planning and organisation of test programs for microprocessors. Suggested techniques are applied to synthesise the model for Parwan and MiniMIPS processors. The model has compact representation, allowing more precise specification of the behaviour of the microprocessor, in more detail, than the traditional instruction set descriptions.

- *Definition of the new high-level classes of fault models for microprocessors, which are also mapped to related low-level structural faults*

The goal of this work is the development of a formal methodology for test generation. Well-defined formal test targets are thus required. After careful investigation of the properties of the chosen formal modelling method, a wide range of possibilities for fault modelling was discovered. HLDDs support multi-level fault modelling, allowing mapping of high-level functional faults to lower-level faults, guaranteeing the high accuracy of testing. Three novel high-level fault classes for microprocessors were proposed, considered superclasses over existing RTL-level fault models for microprocessors. The HLDD-based higher level of abstraction allows the reduction of the size of the fault model by orders of magnitude, compared to the low-level abstractions.

- *A formal method of generation of SBST on the basis of the HLDD model*

The proposed microprocessor models in the form of HLDD networks ensure well-defined structured information, which is more suitable for test generation purposes than traditional models in the form of instruction lists. In order to utilise these models in SBST generation, two novel concepts were proposed: conformity test and scanning test. Conformity test targets the control part of the microprocessor, while scanning test is designed for the data path. Due to the cyclic nature of both algorithms for test generation, it is possible to achieve compact test programs, thus saving memory space. In addition, the exhaustive and pseudo-exhaustive origin of the proposed methods offers high fault coverage and better diagnostic capabilities. The proposed regular

construction (*init-test-store*) of test templates reduces the probability of fault masking. On the other hand, exhaustive testing with repeated constant initialisation procedures has an impact on the number of processor cycles used for test program execution. However, test programs can be always optimised by consideration of the trade-off between accuracy and test length. The proposed SBST generation method was evaluated using Parwan and MiniMIPS microprocessors. Both manually and automatically generated test programs demonstrated their superiority, resulting in up to 10% higher fault coverage than alternative state-of-the-art methods, maintaining the small test program size.

- *Framework for automated SBST synthesis*

Finally, the automatization of the SBST program generation was introduced. Using the formal methods proposed in this work, the framework for automated SBST generation was developed. As the description of instruction sets in the documentation is not uniform, initial data should be extracted manually. Therefore, a methodology for data extraction and its composition into uniform machine-readable format is proposed. The extracted data is used in the developed framework to automatically synthesise the HLDD model of the given processor. Experiments with SBST generation were conducted for the MiniMIPS microprocessor, targeting its execute stage (containing the ALU module) of the pipeline. The obtained fault coverage results are competitive or even superior to those of other state-of-the-art approaches. Tests generated on the basis of only high-level instruction set information achieve the same quality as state-of-the-art methods which use additional information on the implementation details. Nonetheless, the proposed methodology must be extended to cope with the faults in pipeline logic and other traits in hardware implementation.

7.2 Future work

The main direction of the future work is test data generation. Test data generation on the basis of HLDD models is omitted from this work, as it is currently 'in-progress'. The test data used in the experiments were preliminary and manually generated using the ideas of pseudo-exhaustive testing. Properly generated test data may have a strong positive effect on the overall test quality and diagnostic properties of the proposed methodology. Thus, developments in this direction are of the highest priority.

Another functionality to be implemented in frames of SBST generation framework is the signature calculation module. The term signature is used for compressed test responses. Proper algorithms for signature calculation and decryption can improve the diagnostic qualities of the framework.

The work on extending the instruction description language and model synthesis should be continued. This may assist in the adoption of the proposed methodology for a wider spectrum of microprocessor architectures; for example, assisting in modelling complex pipeline behaviour, such as data hazards and stalls.

List of figures

List of tables

References

[1] E. B. Eichelberger and T. W. Williams, “A Logic Design Structure for LSI Testability,” in *Papers on Twenty-five Years of Electronic Design Automation*, New York, NY, USA, 1988.

[2] S. Funatsu, N. Wakatsuki and A. Yamada, “Designing digital circuits with easily testable considerations,” in *Semiconductor Test COnference*, Long Beach, 1978.

[3] 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, 2010.

[4] S. M. Thatte and J. A. Abraham, “Test Generation for Microprocessors,” in *IEEE T. Comput.*, 1980.

[5] *ISO Standard, Road vehicles - Functional safety, 26262,* 2011.

[6] *IEC Standard, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 61508,* 2010.

[7] *DO-254 Standard, Design Assurance Guidance for Airborne Electronic Hardware,* 2000.

[8] 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)*, 2014.

[9] A. Riefert, R. Cantoro, M. Sauer, M. S. 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, 10 2016.

[10] 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*, 2013.

[11] P. Nigh, W. Needham, K. Butler, P. Maxwell and R. Aitken, “An experimental study comparing the relative effectiveness of functional, scan, IDDq and delay-fault testing,” in *Proceedings. 15th IEEE VLSI Test Symposium (Cat. No.97TB100125)*, 1997.

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

[13] I. Bayraktaroglu, J. Hunt and D. Watkins, “Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues,” in *2006 IEEE International Test Conference*, 2006.

[14] P. Parvathala, K. Maneparambil and W. Lindsay, “FRITS - a microprocessor functional BIST method,” in *Proceedings. International Test Conference*, 2002.

[15] 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*, 2006.

[16] 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, 5 2007.

[17] C. H. P. Wen, L.-C. Wang and K.-T. Cheng, “Simulation-Based Functional Test Generation for Embedded Processors,” *IEEE Transactions on Computers,* vol. 55, pp. 1335-1343, 11 2006.

[18] 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, 4 2005.

[19] C. H. Chen, C. K. Wei, T. H. Lu and H. W. 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, 5 2007.

[20] J. 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)*, 1998.

[21] 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, 3 2004.

[22] 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, 9 2015.

[23] S. D. Carlo, P. Prinetto and A. Savino, “Software-Based Self-Test of Set-Associative Cache Memories,” *IEEE Transactions on Computers,* vol. 60, pp. 1030-1044, 7 2011.

[24] 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, 11 2008.

[25] 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*, 2013.

[26] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. Reorda, S. D. 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)*, 2014.

[27] R. Ubar, A. Jasnetki, A. Tsertov and A. S. Oyeniran, Software-Based Self-Test with Decision Diagrams for Microprocessors, LAP LAMBERT Academic Publishing, 2018.

[28] C. Lee, “Representation of Switching Circuits by Binary Decision Programs,” *The Bell System Technical Journal,* pp. 985-999, 1959.

[29] R. Ubar, “Test Generation for Digital Circuits with Alternative Graphs,” *Tallinn Technical University,* no. 409, pp. 75-81, 1976.

[30] S. Akers, “Functional Testing with Binary Decision Diagrams,” *Journal of Design Automation and Fault-Tolerant Computing ,* vol. II, pp. 311-331, 1978.

[31] R. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers,* Vols. C-35, no. 8, pp. 667-690, 1986.

[32] T. Sasao and M. Fujita, “Representations of Discrete Functions,” *Kluwer Academic Publishers,* 1996.

[33] R. Drechsler and B. Becker, Binary Decision Diagrams, Kluwer Academic Publishers, 1998.

[34] S. Minato and N. Ishiura, “Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation,” in *27th IEEE/ACM ICCAD*, 1990.

[35] A. Srinivasan, T. Kam, S. Malik and R. Bryant, “Algorithms for discrete function manipulation,” in *Informations Conference on CAD – ICCAD*, 1990.

[36] U. Kebschull, E. Schubert and W. Rosenstiel, “Multilevel logic synthesis based on functional decision diagrams,” in *IEEE EDAC*, 1992.

[37] S. Minato, “Zero-suppressed BDDs for set manipulation in combinational problems,” in *30th DAC*, 1995.

[38] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo and F. Somenzi, “Algebraic decision diagrams and their applications,” in *Internation Conference on Computer Aided Design*, 1993.

[39] R. Drechsler, A. Sarabi, M. Theobald, B. Becker and M. Perkowski, “Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams,” in *DAC*, 1994.

[40] R. Bryant and Y.-A. Chen, “Verification of arithmetic functions with binary moment diagrams,” in *32nd ACM/IEEE DAC*, 1995.

[41] J. Bern, C. Meinel and A. Slobodova, “Efficient OBDD-based manipulation in CAD beyond current limits,” in *32-nd DAC*, 1995.

[42] E. Clarke, N. Fujita and X. Zhao, “Multi-terminal binary decision diagrams and hybrid decision diagrams,” in *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996, pp. 93-108.

[43] R. Stanković, J. Astola, M. Stanković and K. Egiazarjan, “Circuit synthesis from Fibonacci decision diagrams,” *VLSI Design, Special Issue on Spectral Techniques and Decision Diagrams,* no. 14, pp. 23-34, 2002.

[44] R. Ubar, “Test Synthesis with Alternative Graphs,” *IEEE Design&Test of Computers,* pp. 48-57, 1996.

[45] R. Ubar, J. Raik, A. Jutman and M. Jenihhin, “Diagnostic Modeling of Digital Systems with Multi-Level DDs,” in *Design and Test Technology for Dependable SoC*, IGI Global, 2011, pp. 92-118.

[46] A. Fauth, M. Freericks and A. Knoll, “Generation of hardware machine models from instruction set descriptions,” in *Workshop on VLSI Signal Processing*, Veldhoven, 1993.

[47] V. Zivojnovic, S. Pees and H. Meyr, “LISA-machine description language and generic machine model for HW/SW co-design,” in *VLSI Signal Processing*, San Francisco, 1996.

[48] J. Raik, Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams, Tallinn: TUT Press, 2001.

[49] R. Ubar, J. Raik, A. Karputkin and M. Tombak, “Synthesis of High-Level Decision Diagrams for Functional Test Pattern Generation,” in *International Conference MIXDES*, Lodz, 2009.

[50] A. Tsertov, System Modeling for Processor-Centric Test Automation, Tallinn: TTU Press, 2012.

[51] A. Tsepurov, Hardware Modeling for Design Verification and Debug, Tallinn: TUT Press, 2013.

[52] M. Jenihhin, Simulation-Based Hardware Verification with High-Level Decision Diagrams, Tallinn: TUT Press, 2008.

[53] R. Ubar, "Test Generation for Digital Systems on the Vector Alternative Graph Model," in *13th Symposium on Fault Tolerant Computing*, Milan, 1983.

[54] R. Ubar, J. Raik and A. Morawiec, "Back-tracing and event-driven techniques in high-level simulation with decision diagrams," in *International Symposium on Circuits and Systems*, Geneva, 2000.

[55] R. Ubar, A. Morawiec and J. Raik, "Cycle-based Simulation with Decision Diagrams," in *Design, Automation and Test in Europe Conference and Exhibition*, Munich, 1999.

[56] A. Jasnetski, R. Ubar, A. Tsertov and M. Brik, "Software-based self-test generation for microprocessors with high-level decision diagrams," *Estonian Academy of Sciences,* vol. 1, no. 63, pp. 48-61, 2014.

[57] Z. Navabi, Analysis and Modeling of Digital Systems, McGraw-Hill, 1993.

[58] A. Jasnetski, J. Raik, A. Tsertov and R. Ubar, "New Fault Models and Self-Test Generation for Microprocessors using High-Level Decision Diagrams," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS*, Belgrade, 2015.

[59] A. Jasnetski, S. Oyeniran, A. Tsertov, M. Schölzel and R. Ubar, "High-level modeling and testing of multiple control faults in digital systems," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems - DDECS*, Kosice, 2016.

[60] U. Mahlstedt, J. Alt and I. Hollenbeck, "Deterministic Test Generation for Non-Classical Faults on Gate Level," in *ATS*, 1995.

[61] S. Holst and H.-J. Wunderlich, "Adaptive Debug and Diagnosis Without Fault Dictionaries," in *13th ETS*, 2008.

[62] K. Dwarakanath and R. Blanton, "Universal Fault Simulation using fault tuples," in *DAC*, 2000.

[63] K. Keller, "Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms". USA Patent 5546408, 13 August 1994.

[64] R. Blanton and J. Hayes, "On the Properties of the Input Pattern Fault Model," *ACM Transactions on Design Automation of Electronic Systems ,* vol. 8, no. 1, pp. 108-124, 2003.

[65] R. Ubar, "Detection of Suspected Faults in Combinational Circuits by Solving Boolean Differential Equations," *Automation and Remote Control,* vol. 40, no. 11, pp. 1693-1703, 1980.

[66] Y. Cho, S. Mitra and E. McCluskey, "Gate Exhaustive Testing," in *International Test Conference*, 2005.

[67] A. Jas, S. Natarajan and S. Patil, "The Region-Exhaustive Fault Model," in *16th Asian Test Symposium*, 2007.

[68] P. Maxwell and R. Aiken, "Biased Voting: A Method for Simulating CMOS Bridging Faults in the Presence of Variable Gate Logic Thresholds," in *ITC*, 1993.

[69] L. Zhuo, X. Lu, W. Qiu, W. Shi and D. Walker, “A Circuit Level Fault Model for Resistive Opens and Bridges,” in *VLSI Test Symposium*, Napa, 2003.

[70] P. Engelke, I. Polian, M. Renovell and B. Becker, “Simulating resistive bridging and stuck-at faults,” *IEEE Transactions on CAD of IC and Systems,* vol. 25, no. 10, pp. 2181-2192, 2006.

[71] A. Rousset, A. Bosio, P. Girard, C. Landrault, S. Pravossoudovitch and A. Virazel, “Fast Bridging Fault Diagnosis Using Logic Information,” in *16th ATS*, Beijing, 2007.

[72] S. Jain and V. Agrawal, “Modeling and Test Generation Algorithms for MOS Circuits,” *IEEE Transactions on Computers,* Vols. C-34, no. 5, pp. 426-433, 1985.

[73] H. Lee and D. Ha, “SOPRANO: An Efficent Automatic Test Pattern Generator for Stuck-Open Faults in CMOS Combinational Circuits,” in *DAC*, Orlando, 1990.

[74] A. Kristic and K. Cheng, Delay Fault Testing for VLSI Circuits, Dordrecht: Springer US, 1998.

[75] G. Chen, S. Reddy, I. Pomeranz, J. Rajski, P. Engelke and B. Becker, “A Unified Fault Model and Test Generation Procedure for Interconnect Opens and Bridges,” in *10th ETS*, Tallinn, 2005.

[76] M. Hansen and J. Hayes, “High-Level test generation Using Physically-Induced Faults,” in *VLSI Test Symposium*, 20-28, 1995.

[77] T. Lin and S. 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, 1985.

[78] G. Buonanno, F. Ferrandi and F. Fummi, “How an Evolving Model Improves the Behavioral Test Generation,” in *7th Great Lakes Symposium on VLSI*, 1997.

[79] A. Fin and F. Fummi, “A VHDL Error Simulator for Functional Test Generation,” in *Proceedings of the Design, Automation and Test Conference*, 2000.

[80] S. Ghosh and T. Chakraborty, “On Behavior Fault Modelling for Digital Designs,” in *Electronic testing: Theory and Applications*, Kluwer Academic Publishers, 1991, pp. 135-151.

[81] C. Cho and J. Armstrong, “A Behavioral Test Generation Algorithm,” in *International Test Conference*, 1994.

[82] R. Ramchandani and D. Thomas, “Behavioral Test Generation using Mixed Integer Non-linear Programming,” in *IEEE International Test Conference*, 1994.

[83] S. M. Thatte and J. A. Abraham, “Test Generation for Microprocessors,” *IEEE Transactions on Computers,* Vols. C-29, pp. 429-441, 6 1980.

[84] F. Happke and e. al., “Cell-Aware Test,” *IEEE Transactions on CAD of IC and Systems,* vol. 33, no. 9, 2014.

[85] L. 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, no. 3, pp. 369-380, 2001.

[86] Y. Zhang, H. Li and X. Li, “Software-Based Self-Testing of Processors Using Expanded Instructions,” in *19th IEEE Asian Test Symposium*, Shanghai, 2010.

[87] A. Jasnetski, R. Ubar and A. Tsertov, “Automated software-based self-test generation for microprocessors,” in *2017 MIXDES - 24th International Conference "Mixed Design of Integrated Circuits and Systems*, Bydgoszcz, 2017.

[88] OpenCores, “MiniMIPS ISA”.

[89] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[90] “ModelSim,” [Online]. Available: https://www.mentor.com/products/fv/modelsim/. [Accessed 18 12 2017].

[91] “TetraMAX ATPG,” [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/test-automation/tetramax-atpg.html. [Accessed 18 12 2017].

[92] Y. Zhang, H. Li and X.-W. 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, no. 7, pp. 1220-1233, 2013.

[93] J. Bertin, Graphics and Craphic Information Processing, Berlin: Walter de Gruiter, 1981.

[94] S. A. a. L. Shih, “Towards and Interactive Learning Approch in Cybersecurity Education.,” in *Proceedings of the 2015 Information Security Curruculum Development Conference*, New York, 2015.

Acknowledgements

I would like to thank all those who supported me during my PhD studies, and without whom this work would never have been completed.

I would like to express particular gratitude to my supervisors, Professor Raimund-Johannes Ubar and Dr. Anton Tšertov, for helping me to take my first steps in the engineering domain. They guided me through my PhD studies challenging me along the way. It has been a big pleasure to work alongside them.

I would also like to thank those in the Department of Computer Systems and my colleagues from Testonica Lab, all of whom contributed to my work through heated discussions and ideas.

Special thanks to Dr. Margus Kruus, the head of the Department of Computer Systems, for his support with many administrative issues.

I would also like to acknowledge several organisations that supported my PhD studies: the Tallinn University of Technology, the IT Academy of Estonia, the Information Technology Foundation (HITSA), the Estonian Association of Information Technology and Telecommunications (ITL) and the Estonian Ministry of Education and Research.

Abstract
Software-Based Self-Test for Microprocessors with High-Level Decision Diagrams

The field of Software-Based Self-Test (SBST) has been a topic of extensive research in industry and academia for more than three decades. Nevertheless, self-test programs for microprocessors are generally written manually, due to a lack of attention paid to efficient formal methods. High-level fault modelling and formal test generation strategies have not been studied sufficiently to support the automated synthesis of self-test programs and to provide methods for fast test quality evaluation. In addition, restrictions imposed by the NDA on commercial microprocessors, have made test program generation impossible for most state-of-the-art SBST methods.

This thesis contributes to closing these gaps by introducing a formal methodology for automated SBST program synthesis, based on instruction set description of microprocessors. High-level decision diagrams (HLDDs) were chosen to provide a formal ground for presented methodology.

The research presented in this thesis originated from a method of building HLDD models using data extracted exclusively from instruction set architecture description. This novel method models microprocessor as a set of interrelated HLDD graphs. The proposed modelling approach allows the reflection of high-level functionality of microprocessor with nodes in HLDDs. This provides an opportunity to use the nodes in HLDD graphs for the development of test strategies and the design of test programs for microprocessors.

In this work, it was established that in comparison to the state-of-the-art approaches, the HLDD-based model covers a wide spectrum of high-level behavioural faults in microprocessors. In addition, the transition from a lower to a higher level of abstraction reduces the size of HLDD-based fault models by orders of magnitude. Despite the compaction of the model, the newly proposed fault classes guarantee a high accuracy of testing, which was demonstrated by mapping the new fault classes onto lower level faults and showing that the HLDD-based high-level fault classes fully cover a broad class of structural gate-level fault models.

Two novel concepts for test generation are proposed in this thesis: conformity testing and scanning testing. The use of both algorithms of conformity and scanning test generation results in compact presentation of the test program, high fault coverage, increase in diagnostic capabilities, and reduction in the probability of fault masking.

The overall formalism of the presented methodology allows an automated model synthesis for the microprocessor, and self-test program generation. The implementation of these methods is conducted under the heading of the proposed automation oriented framework. Its structure and algorithms are discussed in detail, and evaluated on the examples of the Parwan and MiniMIPS processors. The obtained fault coverage results are competitive or even superior to those of other state-of-the-art approaches. The main advantage of the newly developed methods is in the capability of the tests generated for microprocessors on the basis of high-level instruction set information, which achieve the same quality as state-of-the-art methods do, which rely on additional information regarding low-level implementation details.

Lühikokkuvõte
Mikroprotsessorite tarkvara-põhine enesetestimine kõrgtasandi otsustusdiagrammide põhjal

Mikroprotsessorite tarkvara-põhise enesetestimise (SBST) valdkond on olnud ulatuslik teema tööstuses ja akadeemiliste ringkondades rohkem kui kolm aastakümmet. Tõhusate formaalsete meetodite puudumise tõttu programmeeritakse mikroprotsessorite enesekontrolli teste käsitsi. Rikete modelleerimise lähenemisviise kõrgematel abstraktsetel tasanditel ja formaalsete testimistegevuste strateegiaid ei ole piisavalt põhjalikult uuritud, et toetada mikroprotsessorite enesetestiprogrammide automaatset sünteesimist ja testimise kvaliteedi hindamise kiireid meetodeid. NDA poolt kaubanduslikele mikroprotsessorite kehtestatud lisapiirangute tõttu kirjeldatakse protsessorite funktsionaalsust üksnes käsustike arhitektuuri esitavate dokumentidega, avaldamata seejuures implementatsioonide detaile, muutes seetõttu kõrgekvaliteediliste testprogrammide automaatse genereerimise ja testide kvaliteedi hindamise võimatuks enamuste kaasaegsete SBST meetodite puhul.

Käesolev väitekiri on suunatud nimetatud lünkade likvideerimisele, pakkudes välja formaalse metoodika automatiseeritud enesetestiprogrammide sünteesiks mikroprotsessoritele, mis põhineb üksnes protsessorite käsustike kirjeldustel. Valitud metoodika formaalse aluse loomiseks valiti digitaalsüsteemide kõrgtasandi otsustusdiagrammid (HLDD).

Väitekirjas on välja töötatud meetod HLDD mudelite ehitamiseks protsessorite käsustike kirjelduste põhjal. Selle meetodi abil saab mikroprotsessorit kujutada mudeli abil, mis koosneb HLDD mudelite võrgust, kus üksikud HLDD-graafid kujutavad erinevaid protsessorite funktsionaalseid üksusi. Välja töötatud modelleerimisviis garanteerib üks-ühese vastavuse HLDD sõlmede ja mikroprotsessori funktsionaalsete alamskeemide vahel. See võimaldab kasutada HLDD sõlmede hulka kontrollnimekirjana protsessori testprogrammide planeerimiseks ja organiseerimiseks abstrahheerimise kõrgtasandil.

Käesolevas töös õnnestus kindlaks teha, et HLDD-mudel pakub paremaid võimalusi mikroprotsessori käitumishäirete modelleerimiseks kõrgtasandil, võrreldes nüüdisaegsete lähenemisviisidega. Traditsiooniliste meetodite puhul vaadeldakse protsessorite käske kui tervikuid, samal ajal töös välja arendatud uue lähenemisviisi puhul vaadeldakse käske kui funktsioonide komplekse, mis võimaldab detailsemat käsitlust ja seetõttu ka adekvaatsemate ja usaldusväärsemate tulemuste saamist.

Samal ajal võimaldab HLDD-põhine käsitlus vähendada rikete mudeli mahtu tervelt suurusjärgu võrra traditsioonilist loogikatasandit silmas pidades. Mudeli kokku surumisest hoolimata tagab uus rikete käsitlus samaväärse testimiskvaliteedi loogikatasandiga võrreldes, mis sai töös ka ära tõestatud uute rikete klasside kaardistamise teel loogikatasandile, näidates et HLDD-põhine ühtne rikete mudel katab täielikult loogikatasandi laia rikete mudelite klassi.

Töös on välja pakutud uue mikroprotsessorite mudeliga hästi kooskõlas olevad kaks uut kontseptsiooni testide genereerimiseks koos vastavate sünteesialgoritmidega - konformsustest ja skaneerimistest. Mõlema testi kooskasutusega on võimalik saavutada kogutesti suur kompaktsus, mis võimaldab vähendada testi salvestamiseks vajaliku mälu mahtu. Lisaks tagatakse väga hea rikete kate, parem diagnoosikvaliteet ja väheneb rikete maskeerimise tõenäosus ehk siis testimistulemuste usaldusväärsus.

Esitatud metoodika kõrge formaliseeritus võimaldab automatiseerida testitava mikroprotsessori kõrgtasandi mudeli sünteesi ja selle mudeli põhjal toimuvat testprogrammide genereerimist. Välja töötatud algoritmid on koondatud ühtsesse raamistikku. Selle struktuur ja kõik uued algoritmid on implementeeritud tarkvarana, mida on edukalt katsetatud kahe mikroprotsessori Parwan ja MiniMIPS testprogrammide sünteesi näitel.

Eksperimentaalse uurimistöö tulemused tõendavad, et uued meetodid, mis põhinevad ainult kõrgtasandi info (mikroprotsessorite käsustike) kasutamisel, on konkurentsivõimelised või isegi paremad, võrreldes olemasolevate meetoditega, mis lisaks käsustikule kasutavad ka lisainfot selle kohta, kuidas mikroprotsessorid on skeemiliselt implementeeritud.

Appendix A

Publication I

Jasnetski, Artjom; Ubar, Raimund; Tsertov, Anton; Brik, Marina (2014). "Software-based self-test generation for microprocessors with high-level decision diagrams." Proceedings of the Estonian Academy of Sciences, 63 (1), 48–61.

Proceedings of the Estonian Academy of Sciences,
2014, **63**, 1, 48–61
doi: 10.3176/proc.2014.1.08
Available online at www.eap.ee/proceedings

COMPUTER
ENGINEERING

Software-based self-test generation for microprocessors with high-level decision diagrams

Artjom Jasnetski, Raimund Ubar, Anton Tsertov*, and Marina Brik

Department of Computer Engineering, Tallinn University of Technology, Akadeemia tee 15a, 12618 Tallinn, Estonia

Received 3 February 2014, accepted 25 February 2014, available online 14 March 2014

Abstract. This paper presents a novel approach to automated behavioural level test program generation for microprocessors using the model of high-level decision diagrams (HLDD) for representing instruction sets. The methodology of using HLDDs for modelling of microprocessors, and a new HLDD-based fault model are developed. The procedures for automated test program generation are presented using a formal model of HLDDs. The feasibility and efficiency of the new methodology are demonstrated by carrying out experimental research on test generation for a 8-bit microprocessor. The results are promising, showing the advantages of the new method and demonstrating better quality of tests compared to previous results.

Key words: microprocessor, software-based self-test, test program generation, high-level decision diagrams.

1. INTRODUCTION

The modern technology advances are imposing new challenges on microprocessor testing. As the transistor size decreases, the number of transistors per chip and operating frequency are growing. Modern processor cores are built from billions of transistors and are capable to operate at gigahertz frequencies. Testing of such complex components has been a challenge for several decades. Sequential automated test pattern generator (ATPG) is, typically, inefficient in terms of test generation time for processor cores [1,2]. Historically, the most common way of solving testing problems for VLSI designs is to apply the design for testability (DFT), for example, the insert scan-chain [3]. However, scan-chains involve changes in the initial circuit design that affect performance, power consumption, and chip area. Despite that, today DFT techniques like scan-chains are an inevitable part of a processor testing plan that require an expensive external ATE.

During the last decade, the semiconductor industry was challenged to bring out new testing methods that can be incorporated in an established microprocessor test flow. Those methods are targeted high quality product development without excessive overhead in the test budget. Such a test method was first proposed in 1980 [4], called software-based self-test (SBST).

The main principle of SBST is to execute the test program on an embedded processor for the purpose of testing the processor itself and the surrounding resources. This approach eliminates the need of expensive external testing hardware. Hence, the test time is limited with the performance of the processor, as soon as the tests are executed at functional speed of the microprocessor. The interest for SBST was renewed during the past decade, because of growing cost of functional testers. The main subject in SBST methodology is a

* Corresponding author, anton.tsertov@ttu.ee

test program generation method, which must comply with the high-quality fault coverage standards imposed by the industry.

In general, the development of the SBST program consists of four steps:
1) creation and optimization of test pattern delivery templates in assembly language,
2) module-level instruction imposed (functional) constraint extraction,
3) test generation process for each module of the processor under test,
4) translation of test patterns to self-test programs.

The last step is basically a process of joining the test pattern with the test pattern delivery template.

Initial focus of the research was on the fault coverage of the tests. The fault coverage of the SBST test is primarily affected by the test patterns. One of the ways to obtain test patterns is to run ATPG. In [5] it was shown that the processor can be divided into modules under tests (MUTs) to ease the task of ATPG. The other way is to use random test patterns for MUTs [6]. 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. The latter requires a manual effort to collect the constraints that guide ATPG at gate level. Obviously, considering todays complexity of the gate level processor implementation it is not feasible to have manual operations at the gate level.

An automatic constraint extraction, based on the gate level simulation of generated tests to check their functional feasibility, was proposed in [7]. But the efficiency of the method on the industrial processors was known to be low. In [8] it is suggested to shift test pattern generation from the gate level to the RT level. This is achieved by the reuse of the verification test patterns. The drawback of this method is that high fault coverage for structural faults cannot be guaranteed by verification test patterns.

Considering the drawbacks of the previously mentioned methods we followed the idea to benefit from the test generation at gate level and to collect functional constraints at RT level description of the processor. One of the papers [9] shows the possibility to use the bounded model checker (BMC) to map the pre-generated test patterns into delivery templates program. Regardless of that, it is done at RT level, industrial processor designs cause time-out problems [10].

Another hybrid SBST method [11] was proposed to utilize the deterministic structural SBST methodologies (using RT-level test development and gate-level-constrained ATPG test development) combined with verification based self-test code development and directed random test pattern generator (RTPG). This method overcomes the drawbacks of [8] and [9].

In addition to hybrid SBST methods [11,12] that work on RT level and gate level, there are methods that achieve comparable results and improve scalability when generating SBST programs using only RT level description of the MUTs [10,13].

In this paper, the SBST program generation, using MUTs modelling, is considered at behavioural level, generally relying on the processor instruction set architecture (ISA). We propose a formal method to automate the test program generation for microprocessors using high-level decision diagrams (HLDD) [14,15] as a diagnostic model. The novelties of the approach are: reduced probability of fault masking, better diagnostic opportunities, and compactness of the whole test thanks to uniform organization of test routines. Experimental data for the Parwan microprocessor [16,17] show higher fault coverage in comparison to the state-of-the-art approaches.

The paper is organized as follows. Section 2 presents the mathematical basis that supports the HLDD theory. Section 3 is devoted to behavioural modelling of the microprocessors. The test generation details are outlined in Section 4. Experimental results are presented in the conclusive fifth section.

2. HIGH-LEVEL DECISION DIAGRAM AS A BEHAVIOURAL LEVEL MODEL OF A MICROPROCESSOR

Consider a digital system S as a network of components (or subsystems), where each component is represented by a function $z = f(z_1, z_2, \ldots, z_n) = f(Z)$, where Z is the set of variables (Boolean, Boolean vectors or integers), and $V(z_k)$ is the set of possible values for $z_k \in Z$, which are finite.

Definition 1. *A decision diagram (DD), which represents a digital function $z = F(Z)$, is a directed acyclic graph $G_z = (M, \Gamma, Z, F)$ with a set of nodes M and a mapping Γ from M to M. $\Gamma(m) \subset M$ denotes the set of all successors of the node $m \in M$, and $\Gamma^{-1}(m) \subset M$ denotes the set of all predecessors of m. M is partitioned into two subsets of nodes*: nonterminal M_N *and* terminal M_T *nodes. The graph has a root node m_0 with $\Gamma^{-1}(m) = \varnothing$. The nonterminal nodes $m \in M_N$ are labelled by variables $z(m) \in Z$, and they have at least two successors, $2 \le |\Gamma(m)| \le |V(z(m))|$, where $V(z(m))$ is the range of values of the node variable $z(m)$. The terminal nodes $m_k \in M_T$ are labelled by sub-functions $z(m_k) = f_k(Z_k)$, $f_k(Z_k) \in F$, which may be as well variables $z_k \in Z$ or constants.*

Definition 2. *For the assigned value of $z(m) = e$, $e \in V(z(m))$, we say that the edge from $m \in M$ to its successor $m^e \in \Gamma(m)$ is activated. Consider situation where to all variables $z \in Z$ is assigned a vector Z^t from the domains $V(Z)$. The activated by Z^t edges form an activated path $l(m_0, m_k) \subseteq M$ from the root node m_0 to one of the terminal nodes m_k, labelled by $f_k(Z_k)$.*

Definition 3. *We say that a decision diagram G_z represents a function $z = F(z_1, z_2, \ldots, z_n) = F(Z)$, iff for each value $V(Z) = V(z_1) \times V(z_2) \times \ldots \times V(z_n)$, a path in G_z is activated from the root node m_0 to a terminal node m_k, labelled by f_k, so that $z = f_k(Z_k)$ is valid.*

The traditional BDDs [18] represent a special case of DDs where for all $z \in Z$, $V(z) = \{0, 1\}$ and there are only two terminal nodes labelled by the Boolean constants 0 and 1. Depending on the class of the system (or its representation level), we may have various DDs, where nodes have different interpretations and relationships to the system structure.

In the following we will consider microprocessors (MP) presented on the behaviour level and described by instruction sets, which usually are described in manuals. Consider, as an example, a hypothetical simple microprocessor with its instruction set in Table 1 and a general behavioural level structure in Fig. 1.

Denote the instructions of the microprocessor as the values of a complex variable I, represented as concatenation of 5 instruction sub-variables $I = OP.B.A1.A2.A$. The variables OP and B denote two fields of the operation code, $A1$ and $A2$ are register addresses, and A is the memory address. Let $V(OP) = V(A1) = V(A2) = 0, 1, 2, 3$ and $V(B) = 0, 1$.

Let us divide MP into three parts: control part, data part, and memory. There are two register blocks, R_{DATA} and R_{CONTR}, in the MP: the register block in the data part consists of 4 general data registers $R_{DATA} = R1, R2, R3, R4$ and the control part includes 2 control registers $R_{CONTR} = PC, AR$ where PC is the program counter, and AR is the address register for addressing the data. ALU is a combinational part of the MP which covers all data manipulation circuits, decoders, multiplexers, demultiplexers, etc. Control part includes finite state machine (FSM) with state register and control logic.

Table 1. Instruction set of a hypothetical microprocessor with 10 instructions

OP	B	Mnemonic	Semantic	RT level operations
0	0	LDA A1, A	READ memory	R(A1) = M(A), PC = PC + 2
	1	STA A2, A	WRITE memory	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) = ¬ R(A2), PC = PC + 1
2	0	ADD A1,A2	Addition	R(A1) = R(A1) + R(A2), PC = PC + 1
	1	SUB A1,A2	Subtraction	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

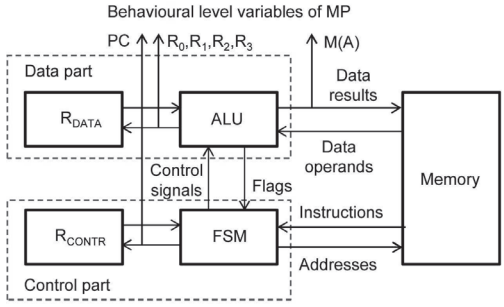


Fig. 1. Behavioural level structure of the microprocessor.

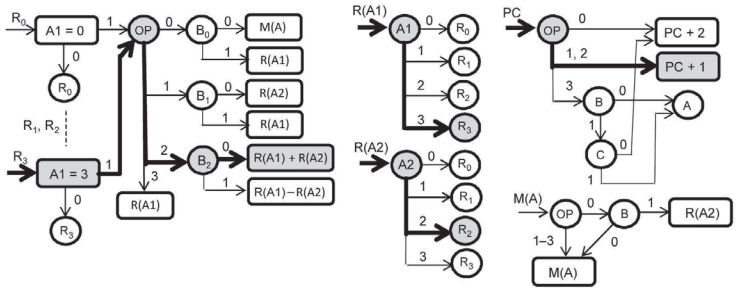


Fig. 2. HLDD model of the microprocessor.

Consider MP functionally as a set of the following behavioural level functions:

– $R_i = f_i(I, S(R_i)) = f_i(OP, B, S(R_i))$, where $R_i \in R_{DATA}$, $i = 0,1,2,3$, and $S(R_i) = \{R_{DATA}, M(A)\}$ is the set of data arguments for the functions f_i (a set of the source registers over all the instructions);

– $PC = f_{PC}(I, C, PC) = f_i(OP, B, PC)$, where C is the flag variable serving as the condition for the branch operation;

– $M(A) = f_M(I, S(M(A))) = f_i(OP, B, S(M(A)))$, where $S(M(A)) = \{R_{DATA}, M(A)\}$.

The functionality of MP can now be represented by a set of behavioural level variables $Z = R_{DATA} \cup R_{CONTR} \cup M(A)$ and by a set of functions $F = f_0, f_1, f_2, f_3, f_{PC}, f_M$. The behaviour of MP can be modelled by the functional basis F and monitored through the variables in Z. For modelling of F we will use the behavioural level HLDD model.

The HLDD model of the microprocessor, given by the instruction set in Table 1, is depicted in Fig. 2. It represents the set of 7 functions in F in the form of 7 HLDDs, respectively: G_{R_i}, $i = 0,1,2,3$; $G_{R(A2)}$, G_{PC}, and $G_{M(A)}$. The 4 graphs G_{Ri} are connected and share a similar sub-graph, which represents the logic of ALU. The graphs $G_{R(A1)}$ and $G_{R(A2)}$ are accessed when modelling the nodes $R(A1)$ and $R(A2)$ in the graph G_{R_i} or $G_{M(A)}$, respectively.

In the following we will call the nodes by the names of node variables or by the expressions in the nodes. To distinguish the nodes, which are labelled by the same variable in the given HLDD, we will use subscripts of this node variable. For example, in the graphs G_{R_i}, we have three different nodes labelled by the same variable B, and the subscript of B distinguishes the nodes.

Each instruction in Table 1 can be modelled by corresponding paths in the HLDD model. To simulate the instruction, its related path in HLDD is to be activated. For example, to simulate the instruction $I = (OP = 2.B = 0.A1 = 3.A2 = 2)$, the following paths in Fig. 2 have to be activated: $G_{R_3} : L(A1 = 3,$

$OP, B2, R(A1)+R(A2)$, $G_{R(A1)}: L(A1,R3)$, $G_{R(A2)}: L(A2,R2)$, $G_{PC}: L(OP,PC+1)$ in the graphs G_{R_2}, $G_{R(A1)}$, $G_{R(A2)}$, and G_{PC}, respectively. The activated paths are highlighted by bold edges and gray coloured nodes.

On the other hand, each HLDD node can be regarded as a hypothetical structural unit of the microprocessor, exercised by a corresponding instruction. For example, the terminal nodes, which are labelled by variables, may represent registers or buses, whereas other terminal nodes, which are labelled by arithmetic or logic expressions, represent the data manipulation sub-units in ALU. The nonterminal nodes of HLDDs are representing the units for interpretation of control information (OP, B, C, etc.) which may be decoders, multiplexers or de-multiplexers. For example, the node $A1 = 0$ in G_{R_0} represents a de-multiplexer, the node $A2$ in $G_{R(A2)}$ represents a multiplexer, the nodes OP and B in the graphs represent decoders.

Because of this 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 microprocessors. Since the proposed formalized test program generation is based on the behavioural model of the microprocessor, the behavioural fault model is required to automate test program generation and to evaluate the test quality. The challenge is to map the properties of the low-level fault model onto high-level description of the microprocessor.

3. BEHAVIOURAL LEVEL FAULT MODEL FOR MICROPROCESSORS

In the following we will develop a uniform fault model based on the HLDDs which targets the full functional testing of each node in the model. Each path in an HLDD describes the behaviour of the system in a specific mode of operation (working mode). The faults, which may have effect on the behaviour of this working mode, are associated with nodes along the path. A fault in each node may cause incorrect leaving the path activated by a test, which would mean a real activation of another path (in a wrong direction) in the HLDD terminating at a wrong terminal node.

From this point of view, the following abstract fault model for nonterminal nodes $m \in M_N$ with node variables $z(m)$ in HLDDs was defined [19].

Definition 4. *The HLDD based fault model for microprocessors includes three fault classes*:
$D1$: *The output edge of a node m for* $z(m) = v, v \in V(z(m))$ *is always activated*; *notation*: $z(m)/v$; (*it is similar to the logic level stuck-at fault* (*SAF*) $z/1$ *for the line* z);
$D2$: *The output edge for* $z(m) = v$ *is broken*; *notation*: $z(m)/\varnothing$; (*similar to SAF* $z/0$ *for the line* z);
$D3$: *Instead of the given edge for* $z(m) = v_i$, *another edge* v_j *or a set of edges* $V_j \in \varnothing$ *is activated*; *notation*: $z(m)/(v_i \to V_j)$.

The fault model, defined on HLDDs, is related to the nodes m of HLDDs, and is a very general one. In [19] it was shown that the fault model described in Definition 4 covers all the 14 different functional level fault classes for microprocessors, introduced in [4].

Let us extend now the fault model, described in Definition 4, by taking into account the following implementation related assumptions introduced in [4] that consider the technology depending details.

Definition 5. *If no register is accessed by the fault* $z(m)/\varnothing$ ($D2$) *then whenever a register* R_j *is to be retrieved, a ONE or ZERO* (*depending on the technology*), *are in fact retrieved. ONE denotes a binary vector* (111), *similarly ZERO stands for* (000).

Definition 6. *If a set R of wrong registers are accessed because of the fault* $z(m)/(v_i \to V_j)$ ($D3$) *then whenever the contents of a register set R is to be retrieved, the contents formed by the bit-wise OR or AND* (*depending on the technology*) *over the registers of the set R will be retrieved. Denote these results as OR*(R) *or AND*(R), *respectively.*

Definition 7. *Introduce a dummy vector* $\Omega \in \{ONE, ZERO\}$ *for general denoting the faulty retrieve specified by Definition* 5, *depending on the technology. Similarly, introduce a dummy operation* $\Psi(R) \in \{OR(R), AND(R)\}$ *for a general denoting of the fault specified by Definition* 6, *depending on the technology.*

Let us generalize now the fault model, introduced in Definition 4, by developing a new uniform HLDD based fault class which takes into account the dependence on the technology as well.

Definition 8. *Introduce a general fault model $D(m)$ for the nodes m of HLDD $G_z = (M,\Gamma,Z,F)$, as the set of the following constraints.*
(1) *Activation constraint. For all values $v \in V(z(m))$, non-overlapping paths must be activated through m, which terminate at non-coinciding terminal nodes $m_v \in M_T$.*
(2) *Propagation constraint. The following has to be satisfied by test data*:

$$\forall v \in V(z(m))[z(m_v) \neq \Omega], \tag{1}$$

$$\forall i,j \in V(z(m))[z(m_i) \neq z(m_j)], \tag{2}$$

$$\forall i,j \in V(z(m))[z(m_v) \not\subset z(m_j)]. \tag{3}$$

The requirement (1) results from Definition 5, and the requirement (2) results from Definition 6. The cases when the introduced requirements cannot be satisfied are classified as redundancies which need no test. Let us call the solution of the constraints in Definition 8 as a test set $T(z(m))$.

Lemma 1. *The test set $T(z(m))$ for a node m in a HLDD $Gz = (M,\Gamma,Z,F)$, which satisfies the activation constraints in Definition* 8, *has the following properties.*
(1) *Each test $t \in T(z(m))$ activates a path through the node m.*
(2) *For each value of $v \in V(z(m))$, there is a test $t_v \in T(z(m))$ with assignment $z(m) = v$.*
(3) *All $t_v \in T(z(m))$ activate paths which terminate at different terminal nodes $m_v \in M_T$.*

Proof. The listed properties result directly from the activation constraints of Definition 8 and can be easily proved by contradiction. □

Lemma 2. *The propagation requirement of Definition* 8 *is sufficient for testing the fault class D*3*, described in Definition* 4.

Proof. Let us have a test set $T(z(m))$ which has the properties of Lemma 1. Consider a HLDD G_R in Fig. 3 with the root node m_0, the node m under test, and a subset of terminal nodes $M_T(m) = m_1, m_{v^*}, m_k, m_n, \subseteq M_T$, reached from m by the paths activated with $T(z(m))$. The paths are shown by dotted edges, which means that they may pass through other nodes not shown in the picture. Assume that there is a fault $z(m)/(v^* \to V^*)$ of class D3, where $v^* \in V(z(m))$ and $V^* \subseteq V(z(m))$. It means that when activating the output edge v^* of the node m, then another set of edges V^* will be activated because of the fault. Both cases, according to D3, are

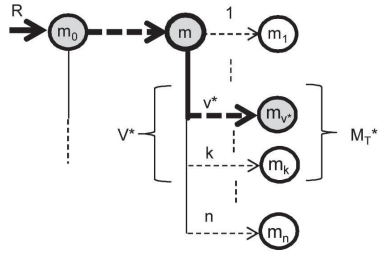


Fig. 3. Illustration of the conditions for testing the node m in the HLDD model G_R.

allowed: $v^* \in V^*$, or $v^* \notin V^*$. According to Lemma 1, when applying the test $t_{v^*} \in T(z(m))$ we expect the result $R = z(m_{v^*})$ in the fault free case. □

(1) Let us have the faulty case where $v^* \notin V^*$ and $V^* = \varnothing$. In this case, no source register will be retrieved, and according to Definitions 5 and 7, we will get $R = \Omega$. But, according to (1) in Definition 8, $z(m_{v^*}) \neq \Omega$, which means that the test $t_{v^*} \in T(z(m))$ detects the fault.

(2) Consider the faulty case when $v^* \notin V^*$ and $V^* = \varnothing$. Denote by $M_{T^*} \subseteq M_T(m)$ the subset of terminal nodes which will be reached when assigning the values $v \in V^*$ to $z(m)$. According to Definitions 6 and 7, the result of the test because of the fault will be $\Psi(z(m)|m \in M_{T^*})$.

From (2) and (3) in Definition 8, the following relationships follow, respectively,

$$\forall v \in V^*[z(m_v) \neq z(m_{v^*})], \tag{4}$$

$$\forall v \in V^*[z(m_v) \not\subset z(m_{v^*})]. \tag{5}$$

On the other hand, based on (4) and (5), it can be easily shown that

$$\Psi(z(m)|m \in M_{T^*}) \neq z(m_{v^*}). \tag{6}$$

From (6) it follows that the test $t_{v^*} \in T(z(m))$ detects the fault.

(3) Consider now the case when $v^* \in V^*$. There are two possibilities. First, the faults are coupled so that for at least two values $v_1, v_2 \in V^*$, we may get the similar result: $z(m_{v_1}) = z(m_{v_2})$. On the other hand, according to the condition (2), all the results of the tests in $T(z(m))$ must be different. Hence, from two similar results we can conclude that we have detected a fault.

Second, assume, that the condition (6) is not fulfilled, and the fault is not detected by the test $t_{v^*} \in T(z(m))$ because of fault masking. However, such a fault can be still detected when we include into $T(z(m))$ an addittional test t'_{v^*} by repeating t_{v^*}, but using different data, so that $z(m_{v^*})' \neq z(m_{v^*})$ would be satisfied. It is easy to show that $\Psi\{z(m)'\} \neq \Psi\{z(m)\}$, which means that the fault will be detected by the added new test.

Theorem 1. *The test set $T(z(m))$, generated for a node m in a HLDD $Gz = (M, \Gamma, Z, F)$ according to the fault model $D(m)$, covers all the fault classes D1, D2, and D3, described in Definition 4.*

Proof. The case D1: Let us prove by contradiction. Assume, there is a fault $z(m)/v \in D1$ in G_z, which is not detected by $T(z(m))$. According to Lemma 1, $T(z(m))$ always includes two tests $t_v, t_{v^*} \in T(z(m))$ with two assignments $z(m) = v$ and $z(m) = v^*$, respectively, where $v^* \neq v$. Hence, the activation constraint of Definition 8 is satisfied. On the other hand, according to Lemma 1, the tests $t_v, t_{v^*} \in T(z(m))$ activate two non-overlapping paths reaching different terminals $m_v, m_{v^*} \in M_T$, where $z(m_v) \in z(m_{v^*})$. Hence, the propagation requirement of Definition 8 is satisfied as well. From that it results that the initial assumption – that the fault $z(m)/v \in D1$ is not detected by $T(z(m))$ – must be false, and therefore the fault model $D(m)$ covers the fault class D1.

The case D2: For the fault class D2, the proof is similar to the case of D1.

The case D3: The proof results from Lemma 2. □

Corollary 1. *From above it follows that the test generation for a node m in HLDD consists of the following three steps*: (1) *activating a path from the root node m_0 to the node m under test*, (2) *activating the non-overlapping paths from m for all $v \in V(z(m))$ to the non-coinciding terminal nodes $m_v \in M_T$, and* (3) *generating the data operands to solve the constraints* (1)–(3) *in Definition 8.*

Corollary 2. *The test set $T(z(m))$, generated for a node m in a HLDD $G_z = (M, \Gamma, Z, F)$ according to Definition* 8, *tests the node exhaustively, and the lower bound of the test length is $|V(z(m))|$.*

Proof. The exhaustiveness of the test set $T(z(m))$ for testing the node m results from Lemma 1. From Property 2 in Lemma 1, also the lower bound $|V(z(m))|$ for the length of the test set $T(z(m))$ results. □

The lower bound will be exceeded if several reiterations of some tests $t \in T(z(m))$ with different data is needed to satisfy step by step the propagation requirements of Definition 8. This situation was discussed in the proof of Lemma 2.

Corollary 3. *When generating tests for the terminal nodes $m_v \in M_T$ of an HLDD, the step* 2 *of the procedure highlighted in Corollary* 1 *will collapse. Only activating a single path from the root node to the node $m_v \in M_T$ is needed.*

The test, generated for a terminal node $m_v \in M_T$, should be executed $|V(z(m_v))|$ times for all the values of $V(z(m_v))$. The tests for terminal nodes m_v are tests for sub-functions $z(m_v) = f_v(Z_v)$, which represent the data path of the system, and therefore, because of exploding size of the test set, cannot be tested exhaustively. Here, the hierarchical approach would be a better option, where the operands for testing the functions of terminal nodes are generated at lower hierarchical (e.g. gate) level. The number of test vectors generated at the lower level will determine the range of values $V(z(m_v))$ for terminal nodes, which will be used for test program synthesis at the higher behavioural level.

Corollary 4. *The test set $T(z(m))$ covers all the* 14 *fault classes introduced for microprocessors in* [4]. *The proof results from Theorem* 1 *and from the analysis carried out in* [19] *where it was shown that all the* 14 *fault classes introduced in* [4] *are covered by the fault classes $D1$, $D2$, $D3$.*

4. BEHAVIOURAL LEVEL TEST GENERATION FOR MICROPROCESSORS

The test program generation for a microprocessor using the HLDD model will proceed at two levels: system level, and module level. Each HLDD presents a module, whereas the whole set of HLDDs presents the system. So far we discussed the main principles of module level testing from a general point of view. At the module level, the targets of test generation are the nodes of HLDDs whereas at the system level the targets are the HLDDs themselves. At the system level, the problem of mapping of the HLDD tests on the system level will be solved; in other words, the test stimuli for the modules will be made controllable and the results of tests will be made observable. In this paper, the detailed discussion about how this mapping can be formalized is omitted.

Definition 9. *Let us call the test for a nonterminal node as* conformity test *for the microprocessor which has the goal to test the control part. The conformity test will be generated according to the procedure summarized in Corollary* 1. *On the other hand, let us call the test for a terminal node as* scanning test *for the microprocessor, which has the goal to test the data path. The scanning test will be generated according to the procedure summarized in Corollary* 3.

4.1. Generation of conformity tests

To generate a conformity test for the control function, represented as a nonterminal node m in the HLDD model, means to test the variable $z(m)$ exhaustively for all the values in $V(z(m))$. For that, we have to activate and exercise all the proper working modes, launched at least once by each value of $z(m)$. Before testing of each working mode, the needed state of the system should be initialized, so that every possible faulty change of $z(m)$ should produce a faulty next state, which would be different compared to the expected next state for the given working mode.

Algorithm 1. Conformity test generation for the control part (test for a nonterminal node m).

1. Generation of control data for the test. Activate a path from the root node m_0 to the node m under test, and for each value $v \in V(z(m))$, a path from the node m to a terminal node $m_v \in M_T$. The values v assigned to $z(m)$ will be cyclically varied during the test execution.
2. Generation of register data for the test. Find the proper initial states of MP for testing the node m. The initial states are determined by a set of contents of the register set, involved in testing of m. These contents

are generated by satisfying the constraints (1)–(3). Denote the set of initial states needed for testing m as $R(m) = (R(m,1), R(m,2), R(m,p))$ where $R(m,i)$ are different initial states. In the best case the constraints (1)–(3) can be satisfied by a single initial state $R(m,1)$. In general case the test set for the node m should be repeated for all the $p \geq 1$ initial states in $R(m)$.

From Algorithm 1 the following test execution program results.

Test program for the nonterminal node m in HLDD Gz:

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

Example 1. Consider the process of conformity test program generation according to Algorithm 1 for testing the node OP in HLDD G_{R_3} in Fig. 2. The goal of this test program is to test the functional behaviour of the control logic in decoding the field OP of the instruction code I.

For generating the control data for the test, we have to choose first the HLDD from 4 possibilities G_{R_i}, $i = 0,1,2,3$. Let us choose the option G_{R_3}, which means that the test result will be sent into the register R3. Now we have to activate first the path from the root node $A1 = 3$ to the node OP (shown by bold edges in G_{R3}). The path will be activated by assigning the value 3 to the variable A1.

Second, we activate the paths from the node OP for all values 0,1,2,3, to terminal nodes. The path for $OP = 2$ through the node B2 (by assigning $B = 0$) to the terminal node $R(A1) + R(A2)$ is shown in bold. Since the value of B is fixed to 0, the paths from OP to other terminal nodes M(A) and R(A2), for values $OP = 0$ and $OP = 1$, respectively, are as well determined. As the result, we have generated the control data for the test in a form of instruction code $I = (OP = VAR.B = 0.A1 = 3.A2 = 2)$. The value "$VAR$" means "the value under variation", i.e. the instruction I will be cyclically executed for all the values $VAR = 0,1,2,3$.

For generating the register data we have to solve the constraints (1)–(3) in Definition 8 for the functions of selected terminal nodes. For example, to satisfy the constraint (2), we have to solve the following inequality:

$$M(A) \neq R2 \neq (R2 + R3) \neq R3. \tag{7}$$

Assume, all the registers have 4-bit length. Then, a possible solution for satisfying the constraints (1)–(3) is: $M(A) = 0110$, $R2 = 0101$, $R3 = 0011$ whereas $R2 + R3 = 1000$. Let us store the test data in the memory at the following addresses: $M(0) = 0110$, $M(1) = 0101$, and $M(2) = 0011$. For the results we reserve the addresses starting from 10.

The test generation process has resulted now in the following test program (sequence of instructions) for testing the node OP in HLDD G_{R3} in Fig. 2.

```
FOR VAR=0,1,2,3
(1) LDA 2, 1   (Initialize R2 = M(1))
(2) LDA 3, 2   (Initialize R3 = M(2))
(3) Execute:  I = VAR.0.3.2 (Testing of
    instructions:  LDA, MOV, ADD, JMP)
(4) STA 3, 10+VAR (Write the content of R3
    into M(10+VAR))
END FOR
```

4.2. Generation of scanning tests

The scanning test program is synthesized hierarchically. The test program itself is generated at the high-level directly from the HLDD model, whereas the data for the test program is generated by a traditional gate-level ATPG using the low-level descriptions of the date path.

Algorithm 2. Scanning test generation for the data path (test for a terminal node m).
1. High-level test generation. Activate a path from the root node m_0 to the terminal node m.
2. Low-level test generation. Find the proper sets of data $R(m) = (R(m,1), R(m,2), \ldots R(m,p))$ for testing the functional expression $z(m)$ of the node m. Here, $R(m)$ is the set of registers (arguments) involved in $z(m)$, and p is the number of test vectors generated at low level.

From Algorithm 2 the following test execution program results:
Test program for the terminal node m in HLDD G_z

```
FOR t = 1,2,...,p
   Initialize the data registers R(m) with
     R(m,t)
   Execute the working mode under test
   READ the value of z.
END FOR
```

Example 2. Consider the process of scanning test program generation according to Algorithm 2 for testing the node $R(A1)+R(A2)$ in HLDD G_{R3} in Fig. 2. The goal of this test program is to test the functional behaviour of the adder in ALU of the microprocessor.

For generating the control data for test, we activate first the path from the root node $A1 = 3$ to the terminal node $R(A1)+R(A2)$ (shown by bold edges in G_{R3}) in a similar way as we did in Example 1. As the result, we have generated the control data for the test in a form of instruction code $I = (OP = 2.B = 0.A1 = 3.A2 = 2)$.

The data for the set of registers $R = R2, R3$ (operands for the addition operation) will be generated at the lower level to achieve the needed (100%) fault coverage. These data (operands) will be cyclically loaded into the registers R2 and R3, before the next execution of the addition operation. Assume that the number of operand pairs generated is 10. Let us store the contents of R2 starting from the memory address $A = 0$, the contents of R3 starting from $A = 10$, and the results starting from $A = 20$. Then the high-level generated test program for testing the node $R(A1)+R(A2)$ in G_{R3} will be as follows:

```
FOR t =0,1,2,...,9
  (1) LDA 2, A(0+t) (Initialize R2 = R2(t))
  (2) LDA 3, A(10+t) (Initialize R3 = R3(t))
  (3) ADD 3, 2 (Execute the instruction
      I = 2.0.3.2)
  (4) STA A(20+t),2 (Write the content of
      R3 into M(20+t))
END FOR
```

5. CASE STUDY AND EXPERIMENTAL DATA

As a case study we have chosen the 8-bit microprocessor Parwan [16,17]. It has instruction format $(OP.I.P.A)$, where OP is 3-bit opcode. If $OP = 7$, then 1-bit I and 4-bit P are used as extensions for opcode, otherwise, I defines addressing mode and P is used for page addressing. A is the 8-bit memory address (offset). The Parwan instruction set (the operation codes OP with extensions I and P) is explained in Table 2, and the HLDD model synthesized from the Parwan instruction set is presented in Fig. 4.

Table 2. Instruction set operation codes for Parwan

	OP			OP	I	P	
LDA	0	AC=M	CLA	7	0	1	AC=0
AND	1	AC=AC ∧ M	CMA	7	0	2	AC= ¬AC
ADD	2	AC=AC ∧ M	CMC	7	0	4	C= ¬C
SUB	3	AC=AC ∧ M	ASL	7	0	8	AC=2AC
JMP	4	PC=A	ASR	7	0	9	AC=AC/2
STA	5	M=AC	BRA_N	7	1	0	If negative
JSR	6	PC=A	BRA_Z	7	1	2	If zero
		Jump to	BRA_C	7	1	4	If carry
		subroutine	BRA_V	7	1	8	If overflow

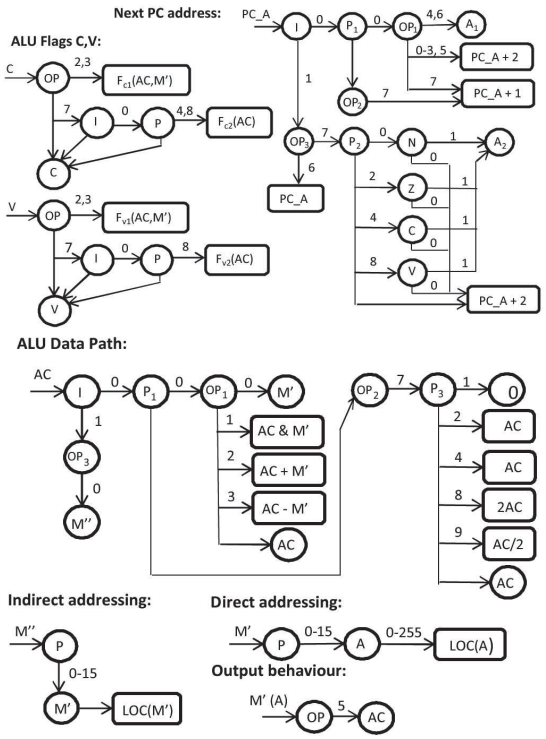


Fig. 4. HLDD model for microprocessor Parwan.

Based on the HLDD model we have created in a formal way the test program shown in Fig. 5. The test in Fig. 5 is based on three embedded cycles for joint testing of ALU and Flag instructions for all data operands. A test for single byte ALU instructions can be organized in a similar way; however, using only two cycles because of no need for testing second time the Flag logic. Using this type of joint test makes it easier to generate data. Generic operands can be generated on the low level with gate-level ATPG for the full combinational logic used for all instructions. Then, in the test execution phase the full test can be carried out cyclically over all generic operands. To use the test cycle like in Fig. 5 is a trade-off problem. Another option would be to flatten these embedded cycles and remove the not-needed repetitions.

```
FOR VAR1 = 0,1,2,3                  (For all double byte ALU instructions LDA, AND, ADD, SUB)
      FOR VAR2 = 0,2,4,... N        (For all data operands)
            FOR VAR3 = 0,2,4,8      (For all 4 branch operations)
k)                LDA, VAR2         (Data init.: AC is loaded with VAR2 for the current cycle)
k+2)              I=0,P=0,OP=VAR1(ALU instruction is tested for the current cycle VAR1)
k+3)              VAR2+1            (Data init: 2nd operand is loaded for the cycle VAR2)
k+4)              I=1,P=VAR3,OP=7(The test response is propagated)
k+5)              m                 (Branch instruction is tested; jump for fixing AC1 = AC)
k+6)              ADD CONST         (Another test response is created for Branch test: AC2=#)
k+8)              ADD, LOC(REF)     (Signature is calculated for ALU test: REF=REF+AC1)
k+10)             STA, LOC(REF)     (Signature is updated)
            END VAR3
      END VAR2
END VAR1
m)    ADD, LOC(REF)                 (Signature is calculated for Branch test: REF=REF+AC2)
m+2)  JMP, k+10
```

Fig. 5. Test program for Parwan.

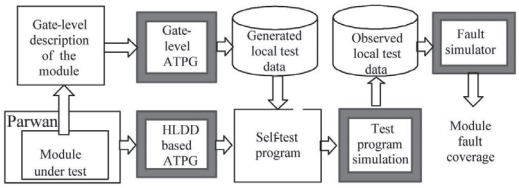


Fig. 6. Set-up of experiments.

Table 3. Experimental results for testing Parwan processor

Module	Gate level stuck-at faults			Fault coverage	
	Total	Tested	Untestable	Proposed, %	ATIG [20], %
AC	156	137	18	99.3	99.3
IR	228	161	66	99.4	96.4
PC	590	560	26	99.3	99.0
MAR	342	242	98	99.2	96.4
SR	130	99	30	99.0	96.8
ALU	962	939	7	98.3	98.0
SHU	310	310	0	100	99.2
Total	2718	2446	245	98.9	97.4

We carried out the experiments with Parwan microprocessor using test program in Fig. 5. The set-up for experiments is presented in Fig. 6 and the results are summarized in Table 3. In Fig. 6 it is shown that test patterns are automatically generated for MUTs at gate level. These test patterns are used as arguments in the test program that is generated from behavioural description (HLDD) of the processor. Then the test program with test patterns is supplied as memory file to ModelSim and simulated to obtain the data sequence for each MUT input signal. Then the test data for MUT inputs is simulated with Turbo Tester simulator at gate level to get the *stuck-at* fault coverage.

The comparison of the obtained fault coverage with the state-of-the-art method [20] is outlined in the leftmost two columns of Table 3. To sum up, for 6 out of 7 modules the proposed method shows advantage over the previously best published results for that processor. The test overhead data is presented in Table 4. The proposed approach needs 75% less test data than in ATIG [20], but the generated program consist of 76% more instructions. However, the latter comparison is not completely fair, since there are single byte and double byte long instructions and such statistics is missing in [20].

Table 4. Test overhead for testing Parwan processor

Test overhead	RSBST [20]	ATIG [20]	Proposed
Instructions	569	189	779
Data (Bytes)	1214	517	132

The fault coverage presented in Table 3 is calculated considering only testable faults. Same arithmetics is used in paper [20] that we use to evaluate the results. In Table 3 in the forth column is presented the number of faults that are proven to be untestable. According to the achieved fault coverage there are still few potentially testable faults that remained untested. At this moment authors consider testing of these faults as future work. The next step in proving the feasibility of the proposed approach is to apply the HLDD-based SBST solution in more complex microprocessors.

6. CONCLUSION

A formal test program generation method based on using high-level decision diagrams is proposed for microprocessors. The HLDD model is created from the instruction set, and it represents the high-level structure of the microprocessor. To take into account the low-level implementation details, the data operands to be used in the test program can be generated by gate-level ATPG. The novelty of the approach is cyclical organization of the test, which directly results from the model structure. The embedded test cycles are directed to exercising of high-level structural components with all instructions and over all data operands generated at the low (gate) level. Because of the cyclical organization, the test is very compact and uniform.

The advantage of such a test is the reduced probability of fault masking due to repeated use of the same initialization before each test step. Improved diagnostic resolution is another advantage of the test program, which directly results from the test structure and from the exact focus of each test step.

The disadvantage of the proposed approach is the test length overhead due to redundant repetition. On the other hand, the embedded test cycles can be easily unrolled, and the flattened test program can be optimized by removing the unnecessary repetitions.

ACKNOWLEDGEMENTS

The work has been supported in part by the EU FP7 STREP project BASTION, Estonian ICT project FUSETEST, by EU through the European Structural and Regional Development Funds, and by Estonian SF grants 8478 and 9429.

REFERENCES

1. Niermann, T. M. and Patel, J. H. HITEC: A test generation package for sequential circuits. In *Proc. European Confer. Design Automation*, 1991, 214–218.
2. Bencivenga, R., Chakraborty, T. J., and Davidson, S. The architecture of the gentest sequential test generator. In *Proc. Custom Integrated Circuits Conference*, 1991, 17.1.1–17.1.4.
3. Eichelberger, E. B. and Williams, T. W. A logic design structure for LSI testability. In *Proc. Design Automation Conference*. New Orleans, 1977, 462–468.
4. Thatte, S. M. and Abraham, J. A. Test Generation for Microprocessors. *IEEE T. Comput.*, 1980, **C-29**, 429–441.
5. Tupuri, R. S. and Abraham, J. A. A novel functional test generation method for processors using commercial ATPG. In *Proc. Internat. Test Confer.*, 1997, 743–752.
6. Chen, L. and Dey, S. Software-based self-testing methodology for processor cores. *IEEE T. Comput. Aid. D.*, 2001, **20**, 369–380.

7. Chen, L., Ravit, S., Raghunathant, A., and Dey, S. A scalable software-based self-test methodology for programmable processors. In *Proc. Design Automation Conference*. Anaheim, Ca, 2003, 548–553.
8. Kranitis, N., Paschalis, A., Gizopoulos, D., and Xenoulis, G. Software-based self-testing of embedded processors. *IEEE T. Comput.*, 2005, **54**, 461–475.
9. Gurumurthy, R. S., Vasudevan, S., and Abraham, J. A. Automated mapping of pre-computed module-level test sequences to processor instructions. In *Proc. Internat. Test Confer.*, 2005, 303–313.
10. Zhang, Y., Li, H., and Li, X. Automatic test program generation using executing-trace-based constraint extraction for embedded processors. *IEEE T. VLSI Syst.*, 2013, **21**, 1220–1233.
11. Kranitis, N., Merentitis, A., Theodorous, G., and Paschalis, A. Hybrid-SBST methodology for efficient testing of processor cores. *IEEE Des. Test Comput.*, 2008, **25**, 64–75.
12. Lu, T.-H., Chen, C.-H., and Lee, K.-J. Effective hybrid test program development for software-based self-testing of pipeline processor cores. *IEEE T. VLSI Syst.*, 2011, **19**, 516–520.
13. Wen, C. H.-P., Wang, Li-C., and Cheng, K.-T. Simulation-based functional test generation for embedded processors. *IEEE T. Comput.*, 2006, **55**, 1335–1343.
14. Ubar, R. Test synthesis with alternative graphs. *IEEE Des. Test Comput.*, 1996, 48–59.
15. Karputkin, A., Ubar, R., Raik, J., and Tombak, M. Canonical representations of high level decision diagrams. *Estonian J. Eng.*, 2010, **16**, 39–55.
16. Navabi, Z. *Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.
17. Testing the Parwan processor. http://mesdat.ucsd.edu/ lichen/260c/parwan/ (accessed 6.03.2014).
18. Lee, C. Y. Representation of switching circuits by binary decision programs. *AT&T Tech. J.*, 1959, 985–999.
19. Ubar, R., Raik, J., Jutman, A., Instenberg, M., and Wuttke, H.-D. Modeling microprocessor faults on high-level decision diagrams. In *Internat. Confer. Dependable Systems and Networks*. Anchorage, USA, 2008, c17–c22.
20. Zhang, Z., Li, H., and Li, X. Software-based self-testing of processors using expanded instructions. In *Proc. 19th IEEE Asian Test Symposium*, 2010, 415–420.

Kõrgtasemega otsustusdiagrammidel põhinev testprogrammide süntees mikroprotsessoritele

Artjom Jasnetski, Raimund Ubar, Anton Tsertov ja Marina Brik

On esitatud uudne lähenemisviis mikroprotsessorite testprogrammide formaalsele sünteesile, kasutades kõrgtaseme otsustusdiagrammide matemaatilist aparaati. On välja töötatud metodoloogia mikroprotsessorite diagnostiliseks modelleerimiseks käsusüsteemidega defineeritud käitumuslikul tasandil. On esitatud vastavad otsustusdiagrammidel põhinevad testide genereerimise algoritmid ja protseduurid. Uue metodoloogia rakendatavust ja efektiivsust on demonstreeritud eksperimentaaluuringutega konkreetse mikroprotsessori näitel. Saadud tulemused näitavad uue lähenemisviisi suuremat efektiivsust analoogsete eksperimentidega võrreldes.

Appendix B

Publication II

Jasnetski, Artjom; Raik, Jaan; Tsertov, Anton; Ubar, Raimund (2015). ”New Fault Models and Self-Test Generation for Microprocessors using High-Level Decision Diagrams”. IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS. Belgrade, Serbia, April 22-24, 2015.

New Fault Models and Self-Test Generation for Microprocessors using High-Level Decision Diagrams

Artjom Jasnetski
Testonica Lab OÜ
Tallinn, Estonia

Jaan Raik, Anton Tsertov, Raimund Ubar
Department of Computer Engineering
Tallinn University of Technology, Tallinn, Estonia

Abstract— **The paper presents a novel approach to high-level fault modeling and test generation for microprocessors using High-Level Decision Diagrams (HLDD). A general frame-work and novel techniques for automated software-based self-test program generation are discussed. On this basis new previously not published test quality improvement capabilities of the approach are high-lighted and explained. Based on the high level fault model defined for HLDDs a novel class of hard-to-test faults, called "*unintended actions*", is proposed. In addition, the mechanisms for reducing the risk of fault masking is explained. The experimental results show the superiority of the new method by achieving a higher quality of tests with shorter length compared to the previous results.**

Keywords— **microprocessor, software-based self-test (SBST), test program generation, high-level decision diagrams**

I. INTRODUCTION

Testing of complex systems like microprocessors has been a challenge for decades. Sequential ATPGs have been found inefficient in terms of test generation time and fault coverage [1,2]. Design-for-testability such as scan-path design allows to improve fault coverage, however, affecting negatively the performance, power consumption and chip area [3].

As the industry has shifted to sub-micron technologies, at-speed testing has become essential [4]. The traditional hardware based Built-In Self-Test (BIST) technology [5] in the case of microprocessors (MP) has proved to be less feasible than for Application Specific Integrated Circuits (ASICs) [6]. As an alternative for MPs, Software-Based Self-Test (SBST) has emerged [6-10]. The idea behind SBST is to execute a test program on an embedded processor for testing the processor itself and its surrounding resources. In [7] it was shown that a processor can be divided into Modules Under Test (MUT) to ease the task of ATPG. An alternative is to use random test patterns for modules [6].

An automated constraint extraction based on gate-level simulation of generated tests to check their functional feasibility was proposed in [8]. However, the efficiency of the method on industrial processors was shown to be low. In [9], shifting of SBST generation from gate-level to Register-Transfer Level (RTL) is suggested. This is achieved by reuse of verification test patterns. The drawback of this method is that high fault coverage for structural faults cannot be guaranteed. Another method that uses RT-Level description for program generation is based on Bounded Model Checker (BMC) [10]. In [11] it is shown that [10] is likely to cause time-out problems when applied to industrial designs.

The methods described in [12, 13] are based on structural SBST methodologies (RTL test development with gate-level constrained ATPG) and combined with verification based self-test. These methods overcome the drawbacks of [9] and [10] and are called as Hybrid SBST. An attempt to improve scalability of hybrid methods by substituting gate-level ATPG by simulation-based approach for test pattern generation was presented in [14].

Nevertheless, none of the state-of-the-art methods have so far tried to develop well formalized high-level (e.g. behavioral level) fault models for coping with hard-to-test faults and fault masking problems at higher levels with ultimate goal to improve test quality and to achieve compact test programs.

In this paper, we introduce a new concept for generating tests for microprocessors. While traditional approaches are based mainly on the idea of testing the instructions as functional entities, the proposed approach is based on partitioning the functionality of microprocessors into smaller entities along two dimensions. The first dimension covers the control activities and is represented by subfields of the instruction format, whereas the second dimension covers the set of all elementary data manipulation activities. Such a two-dimensional testing approach allows setting up of test targets which represent smaller slices of hardware. As the result, it becomes possible to test the target hardware slices nearly exhaustively, and because of shorter test sequences for each target, the approach will be less sensitive to fault masking due to over-writing of erroneous intermediate results during test.

For implementing the proposed conception, the High Level Decision Diagrams [16] (HLDD) derived from Instruction Set Architecture (ISA) descriptions proved to be very useful. The main difference of the HLDD-based test generation algorithms from the traditional methods of test generation is explained by the way how the test targets are chosen. The novel HLDD-based test generation approach is targeting structural entities of MP. The instruction list of MP is converted into a network of HLDDs where each HLDD represents a sub-circuit.

The nodes as test targets in the HLDD model represent smaller functional entities with fine grained (better defined) fault models compared to the traditional instruction based test approach. The new fault modeling idea facilitates achieving higher gate-level fault coverage, reduced risk of fault masking and compactness of the test program due to its cyclic organization. The paper is organized as follows: Section II presents the description of HLDDs. Section III presents the new high-level fault model for MP, and Section IV gives an overview of the algorithms for test generation. Experimental results are discussed in Section VI followed by Conclusions of the paper.

978-1-4799-6780-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DDECS.2015.56

II. Representing Microprocessors with HLDDs

Consider a hypothetical MP with the instruction set in Table 1. Denote the instructions of MP as values of a complex variable I represented as concatenation of 5 instruction sub-variables $I = OP.B.A1.A2.A$. OP and B denote two fields of the operation code, $A1$ and $A2$ are register addresses, and A is the memory address. Let $V(OP) = V(A1) = V(A2) = \{0,1,2,3\}$ and $V(B) = \{0,1\}$.

Consider the MP with the instruction set in Table 1 functionally as a set of the following functions:

$R_i = f_i\,(I, S(R_i)) = f_i\,(OP, B, S(R_i))$ where $R_i \in R_{DATA}$, $i = 0,1,2,3$, and $S(R_i) = \{R_{DATA}, M(A)\}$ is the set of data arguments for the functions f_i (a set of source registers);

$PC = f_{PC}\,(I, C, PC) = f_i\,(OP, B, PC)$ where C is the flag variable serving as the condition for the branch operation;

$M(A) = f_M(I, S(M(A))) = f_i(OP, B, S(M(A)))$ where $S(M(A)) = \{R_{DATA}, M(A)\}$.

The functionality of MP can now be represented by a set of behavioral level variables $Z = R_{DATA} \cup R_{CONTR} \cup M(A)$ and by a set of functions $F = \{f_0, f_1, f_2, f_3, f_{PC}, f_M\}$. The behavior of MP can be modeled by the functional basis F and monitored through the variables in Z. For modeling of F we will use the behavioral level HLDD model.

The HLDD model of MP given by the instruction set in Table 1 is depicted in Fig.1. It represents the set of 7 functions in F in the form of 7 HLDDs, respectively: G_{Ri}, $i = 0,1,2,3$; $G_{R(A2)}$, G_{PC}, and $G_{M(A)}$. The 4 graphs G_{Ri} are merged and share a similar sub-graph which represents the logic of ALU. $G_{R(A1)}$ and $G_{R(A2)}$ are accessed when modeling the nodes $R(A1)$ and $R(A2)$, respectively, in G_{Ri} or $G_{M(A)}$.

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 $PC = PC + 2$

In the following we will refer to nodes m by the node variables $z(m) \in Z$ or by the node expressions denoted as well by $z(m)$. To distinguish the nodes which are labeled by the same variable in the given HLDD, we will use subscripts at the node variable. For example, in the graphs G_{Ri}, we have three different nodes labeled by the same variable B, and the subscript at B distinguishes the nodes. There is one-to-one mapping between the values of $z(m)$ and the output edges of the node m. The edges of nodes are labelled by the corresponding values of node variables. Denote the range of values of $z(m)$ by $V(z(m))$.

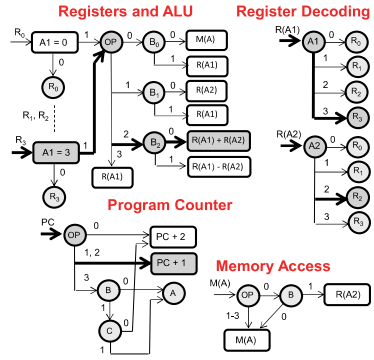


Fig. 1. HLDD model for the microprocessor in Table 1

Each instruction in Table 1 can be modeled by activated paths in the HLDD model. For example, when simulating the instruction I = (OP=2.B=0.A1=3.A2=2), the following paths in Fig.1 are activated: G_{R3}: $l(A1=3, OP, B_2, R(A1)+R(A2)$, $G_{R(A1)}$: $l(A1,R3)$, $G_{R(A2)}$: $l(A2,R2)$, G_{PC}: $l(OP, PC+1)$, respectively, in the graphs G_{R3}, $G_{R(A1)}$, $G_{R(A2)}$ and G_{PC}. The activated paths are highlighted by bold edges and gray colored nodes in Fig. 1.

Each HLDD node represents a functional unit of MP. The terminal nodes labeled by variables represent registers or buses, whereas other terminal nodes labeled by arithmetic or logic expressions represent the data manipulation units within ALU. Nonterminal nodes represent the parts of the control circuit (OP, B, C) like decoders, multiplexers, de-multiplexers. For example, the node $A1 = 0$ in G_{R0} represents de-multiplexer, $A2$ in $G_{R(A2)}$ represents multiplexer, the nodes OP and B represent decoders. Because of one-to-one mapping of HLDD nodes into high-level functional units, we can use the HLDD nodes as a checklist for high-level test planning and organization.

III. Uniform Fault Model for Microprocessors

In the following we develop a uniform fault model based on HLDDs which targets the functional testing of nodes in the model. Each path in HLDD describes the behavior of the system in a specific mode of operation (working mode). The faults which may affect on the behavior of this working mode can be associated with nodes along the path. A fault in each node may cause incorrect exit from the path activated by a test which would mean activation of another (wrong) path.

Definition 1. The HLDD based fault model for MPs includes three fault classes:

D1: The output edge for $z(m) = v$ is broken, $z(m)/\varnothing$; (similar to the logic level stuck-at fault (SAF) $z/0$ for the line z);

D2: The output edge of a node m for $z(m) = v$, $v \in V(z(m))$ is always activated, $z(m)/v$; (similar to SAF $z/1$);

D3: Instead of edge for $z(m) = v_i$, another edge v_j or a set of edges V_j is simultaneously activated, $z(m)/(v_i \rightarrow V_j)$.

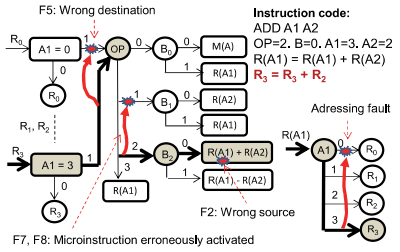


Fig. 2. Illustration of different faults in HLDDs

Consider in Fig. 2 how different fault models in [17,18] can be represented as the node faults on the HLDD model. Addressing fault is illustrated in the graph $G_{R(A1)}$. Instead of the edge 3 of the node *A*1, another edge 0 (or both edges simultaneously) are activated. This fault can propagate to other HLDDs of the model. For example, this fault can cause in ALU either "wrong source" or "wrong destination". A "control fault" is illustrated by the fault of node OP called „instead of edge 2, the edge 1 is activated".

The fault model in Definition 1 covers 14 microprocessor fault classes described in [17,18]. To take into account the implementation related assumptions considered in [17, 18], we introduce the following definitions for further specification of the fault model in Definition 1.

Definition 2. If no register is accessed because of D1 then whenever a register R_j is to be retrieved, a ONE or ZERO (depending on technology), is in fact retrieved. ONE denotes a binary vector (11…1), ZERO stands for (00…0).

Definition 3. If a set *R* of two or more wrong registers are simultaneously accessed because of D2 or D3 then whenever a content of a register set R is to be retrieved, the contents formed by bit-wise OR/AND (depending on technology) over the registers of R will be retrieved. Denote these results as Ψ(R) where Ψ may mean OR or AND, respectively.

Let us generalize now the fault model introduced in Definitions 1-3 by developing a new uniform HLDD based *conditional node fault model* which joins all the 3 fault classes, and takes into account the dependence of technology as well.

Definition 4. The HLDD based conditional node fault model consists of two types of constraints: (1) activation of the fault, and (2) propagation of the fault.

For non-terminal nodes $m \in M_N$, with $V(z(m))$:

(1) *Propagation constraints*. For all values $v \in V(z(m))$ of the node *m*, non-overlapping paths must be activated from the root node through *m* to non-coinciding terminal nodes $m_v \in M_{T,m}$, where $M_{T,m}$ is the subset of all terminal nodes $M_{T,m} \subseteq M_T$, reached from *m*.

(2) *Activation constraints:*

$$\forall v,j \in V(z(m)), i \neq j: [z(m_v) \neq (z(m_j) \neq \Psi_T(M_{T,m}) \neq 0 \neq 1] \quad (1)$$

where $\Psi_T(M_{T,m})$ is calculated by bit-wise OR/AND (depending on technology) over the functions $z(m_v)$ in terminal nodes m_v.

For terminal nodes $m \in M_T$, the propagation constraints are determined by the activated single path from root node to *m*, and the activation constraints are determined by test patterns sufficient for testing the function $z(m)$, denoted by $V(z(m))$.

Let us call the solution for propagation constraints in Def.4 as local test $T_{ST}(m)$ for testing the node *m*. The full test $T(m)$ consists of dynamic and static parts: $T(m) = T_{ST}(m).T_{VAR}(m)$ where the dynamic part $T_{VAR}(m)$ means the set of values $V(z(m))$.

For testing a node in the HLDD model, three actions are needed: (1) local fault activation by satisfying the activation constraints of Def.4, (2) propagation of the fault through HLDD by satisfying the propagation constraints of Def.4, and (3) system level fault propagation through the high-level components of the system, represented by HLDDs.

IV. Test Generation with HLDDs

The test program generation for MPs using HLDDs will proceed at two levels: system and sub-system levels. Each HLDD presents a sub-system, whereas the whole set of HLDDs presents the system. At sub-system level, the targets of test generation are the nodes of HLDDs whereas at the system level, the locally generated tests $T(m)$ will be embedded into the system level test program templates. Two types of test programs are generated: conformity and scanning tests.

Definition 5. *Conformity test* is a test for a non-terminal node with a goal to test a part of the control path of MP.

Definition 6. *Scanning test* is a test for a terminal node with the goal to test a part of the data path of MP.

5.1. Conformity tests. Generating a conformity test for $m \in M_N$ in HLDD, produces exhaustive test for $z(m)$. It consists of the two following steps:

1. Generation of a test $T(m)$ which satisfies the constraints of Def. 4 by activating a path from the root node to the node *m* under test, and for each value $v \in V(z(m))$, a path from *m* to a terminal node $m_v \in M_T$.

2. Justification of $T(m)$ by generating the needed register data $R(m)$ (initial state of the MP).

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

1. Generation of the static part of the test $T_{ST}(m)$ at the high HLDD level 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 generating the needed register data R(m) (initial state of the MP).

Two techniques guarantee reduced probability of fault masking: (1) initialization (for $R(m)$) and observation (for $z(m)$) sequences are kept constant for all cycles over $V(z(m))$; (2) only a single variable is observed when testing $z(m)$ to keep the test sequence as short as possible to avoid overwriting of erroneous signals and hence the fault masking during the test.

The novel idea of the exhausting and independent testing of very small parts of functionalities of instructions and using the fault models D2 and D3 allows covering a novel subclass of hard-to-test faults called "added unintended actions" which are usually neglected when testing MPs by available methods.

V. EXPERIMENTAL DATA

As a case study, we generated a self-test program using the proposed fault models and HLDD based technique described above for the microprocessor Parwan [19][20]. The obtained fault coverage for every module of MP is outlined in Table 2. The whole test program was simulated by ModelSim to obtain local test data sequences for all modules, and these, in turn, were fault simulated at gate level to get SAF coverage.

The comparison of fault coverages with methods [6, 21] is depicted in Table 2. To sum up, for 7 out of 8 modules the proposed method shows advantage regarding test coverage over the previously published results for that processor. The positive impact of the novel high-level fault model can be seen in the higher fault coverage of the control part of MP. The comparison of volumes of test data is presented in Table 3. The proposed approach needs 75% less test data than in ATIG [21], but the generated program consist of 51% more instructions. However, the latter comparison is not completely fair, since there are single byte and double byte long instructions and such statistics is missing in [6, 21].

TABLE II. COMPARISON OF DIFFERENT TEST COVERAGES (PARWAN)

Module	#Faults	Fault coverage %		
		Proposed method	[21]	[6]
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

TABLE III. TEST LENGTHS FOR TESTING PARWAN PROCESSOR

Test overhead	Li Chen [6]	ATIG [22]	Proposed method
Instruction #	575	189	**260**
Test data #	Unknown	517	**132**

We used for experiments the microprocessor PARWAN because of the possibility to compare the results with available published data. The complexity of the algorithms of high-level test generation does not depend on the length of the data word. The latter affects only on gate level test generation performance. But this was not the topic of the current paper. The complexity of the high-level test generation algorithms presented in the paper depends only on the lengths of the functionally independent instruction sub-fields.

CONCLUSIONS

A formal high-level test program generation method based on using HLDDs is proposed for microprocessors. The HLDD model is created from the instruction set, and it represents the high-level structure of MP. To take into account the low-level implementation details, the data operands used in the test program are generated by gate-level ATPG.

The novelty of the proposed approach is in targeting the functional variables as test objectives, represented by HLDDs, instead of testing the instructions as in the traditional cases. A novel HLDD-based fault model was introduced covering a subclass of hard-to-test faults called "added unintended actions" which are usually neglected when testing MPs. This novelty allows to prove easily the correctness of small "portions" of functionality with a side-effect of reducing the risk of fault masking and to improve the diagnostic resolution. Because of cyclical organization of test procedures, the whole test program will be compact and uniform.

Experimental results demonstrated higher fault coverage compared to the published results on the same MP. Additional advancements like testing a new type of hard-to-test faults and less risk for fault masking are hard to demonstrate by only measuring SAF coverage, but the more exact evaluation of this impact will be the future work.

REFERENCES

[1] T. M. Niermann and J. H. Patel, HITEC: A test generation package for sequential circuits. DAC, 1991, pp. 214 – 218.

[2] R. Bencivenga, T. J. Chakraborty, and S.Davidson. The architecture of the gentest sequential test generator. Custom IC Conf., 1991.

[3] E. B. Eichelberger and T.W.Williams, "A logic design structure for LSI testability," DAC, 1977, pp. 462 – 468.

[4] International technology roadmap for semiconductors ITRS.

[5] G.Hetherington, et. al. "Logic BIST for large industrial designs: real issues and case studies," ITC, 1999, pp. 358 – 367.

[6] L.Chen, 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.

[7] R.Tupuri and J.Abraham. A novel functional test generation method for processors using commercial ATPG, ITC, 1997, pp.743 – 752.

[8] L. Chen, S. Ravit, A. Raghunathant, and S. Dey. A scalable SW-based self-test methodology for processors. DAC, 2003, pp. 548 – 553.

[9] N. Kranitis et al. SW based self-testing of embedded processors. IEEE Trans. on Computers, vol. 54, no. 4, April 2005, pp. 461 – 475.

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

[11] Y.Zhang, H.Li,X.Li Automatic test program generation using executing-trace-based constraint extraction for embedded processors. IEEE Trans. on VLSI Systems, vol.21, no.7, July 2013, pp. 1220 – 1233.

[12] N. Kranitis et. al. Hybrid-sbst methodology for efficient testing of processor cores. Design &Test of Comp., vol. 25, no. 1, 2008, pp. 64-75.

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

[14] C.H.Wen, L.Wang, K.Cheng. Simulationbased functional test generation for embedded processors. IEEE Trans. on Comp., vol. 55, no. 11, Nov. 2006, pp. 1335 – 1343.

[15] R.Ubar, A.Tsertov, A.Jasnetski, M.Brik "Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams," 15th Latin American Test Workshop - LATW, 2014, pp. 1 – 6.

[16] A.Karputkin, R.Ubar, J.Raik, M.Tombak. Canonical representations of HLDDs. Estonian J. of Engineering, 2010, pp. 39 – 55.

[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] Z.Navabi, Analysis and Modeling of Digital Systems.McGraw-Hill,1993

[20] On-line: http://mesdat.ucsd.edu/ lichen/260c/parwan/

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

[22] A.Jasnetski, R.Ubar, A.Tsertov, M.Brik. "Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams (Ext.)," Proc. of the Estonian Academy of Sciences, 2014, 63-1, 48-61.

Appendix C

Publication III

Jasnetski, Artjom; Oyeniran, Adeboye Stephen; Tsertov, Anton; Schölzel, Mario; Ubar, Raimund (2016). ”High-level modeling and testing of multiple control faults in digital systems”. IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, 20-22 April 2016.

High-Level Modeling and Testing of Multiple Control Faults in Digital Systems

Artjom Jasnetski, Stephen Adeboye Oyeniran, Anton Tsertov, Mario Schölzel*, Raimund Ubar
Tallinn University of Technology, Estonia, *University of Potsdam, Germany

Abstract — **A new method for high level fault modeling to improve the test generation for the control parts of digital systems was proposed. We developed a new high-level functional fault model based on High-Level Decision Diagrams (HLDD). It allows uniform handling of possible defects in different control functions related to instruction decoding, data addressing, and data manipulation. It was shown how the proposed high-level fault model can be mapped on the low-level faults of a joint class of stuck-at faults (SAF), conditional SAF and bridging faults. We proposed uniform procedures for high-level fault activation as a graph traversing procedure on HLDDs related to selection of control signals, and for fault propagation as a task of solving data constraints without using implementation details. Experimental results demonstrated that combining both, high-level control fault reasoning and low-level test generation for data part of a system can help to achieve higher fault coverage and detection of redundant faults than using low-level test generation approach alone.**

Keywords: ***digital systems, control faults, multiple faults, high-level decision diagrams***

I. INTRODUCTION

The technology advancements impose new challenges to testing of systems-on-chip as device geometries shrink and the complexity of SOCs increases. Modern processor cores are built from billions of transistors and are capable of operating at gigahertz frequencies. Testing of such complex components has been a challenge for several decades. Application of sequential gate-level ATPG is inefficient in terms of test generation time and fault coverage. Design for testability, for example, scan-chains can improve the fault coverage, however, affecting negatively on performance, power consumption and the chip area.

For today's deep sub-micron technologies, at-speed testing has become essential for achieving high test quality. The gap between frequencies of external ATE and processor under test shows that external at-speed testing is practically infeasible. In addition, the ATE accuracy problems result in yield loss [1]. Traditional solution to cope with at-speed testing is Built-In Self-Test (BIST) [2]. 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 not as feasible as for memories or in application specific integrated circuits (ASICs) [3]. Furthermore, BIST results in over-testing as well as overstressing the circuit due to higher-than-normal switching activity during the test.

As an alternative to HW-based self-test such as BIST, software-based self-test (SBST) has emerged [3-7]. SBST is a non-intrusive test methodology that uses available processor resources. In SBST, the role of the ATE is to load the test program into memory and to read the final test results back after the execution of test program is finished.

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 [4] 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 [3]. 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 with 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 [5]. However, the efficiency of the method on the industrial processors was shown to be low. In [6], 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 [6-9]. In addition to Hybrid SBST [9, 10], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [5,8]. 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 [11], an approach to high-level fault modeling and SBST for microprocessors using High-Level Decision Diagrams (HLDD) was proposed, where a general framework and main algorithms for test program automated generation were outlined. On this basis in [12], a method of

generating high-level test groups was developed to avoid mutual masking of faults possibly appearing in different components of microprocessors.

In this paper, we advance further the results in [11,12], and focus mainly on modeling of faults in control units described by the Instruction Set Architectures (ISA). We extend the results of [11,12] by proving the consistency of the proposed high-level fault model and by showing that it covers a broad class of gate-level faults including SAF, conditional SAF and bridging faults with any multiplicity in the control part of a processor. The rest of the paper is organized as follows. Section 2 presents the method of modeling systems with HLDDs, and Section 3 presents the idea of HLDD based control fault modeling. In Section 4 we propose the data constraints based high-level fault model for the control part, Section 5 discusses experiments, and Section 6 concludes the paper.

II. MODELING DIGITAL SYSTEMS WITH HLDDS

We chose a VLIW processor for describing the modelling, because it's simple, but sufficient control structure ease the illustration of mapping between high- and low-level faults. For modeling the processors, we use HLDDs [13, 14] where the nodes represent functional blocks of the system and are considered as test targets. Consider a VLIW processor with 2 execution slots in Fig.3. Each slot has a fetch register that receives the current instruction from the memory via IN1, IN2. Instruction of slot *i* contains the control signals: d*i* - destination register, ctr*i*1 and ctr*i*2 - source registers, ctrAlu*i* - operation in ALU.

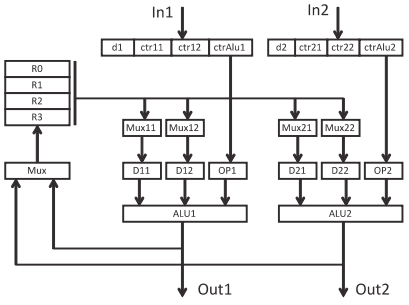


Figure 3. *Structural Representation of the VLIW processor.*

The components Mux*i*1 and Mux*i*2 represent the read ports of the register file for slot *i*. The processor has a register bank with 4 registers, and 2 write ports. Both write ports are represented by 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.

Definition 1. *Digital system model.* Let us have a digital sub-system, 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 HLDDs.

Definition 2. *HLDD for a sub-system* $\{y = f(C,D)\}$. HLDD is a directed, acyclic and connected graph $G(y,M,\Gamma)$ with a root y , and a set of nodes M where the *non-terminal* nodes $m \in M^N \subset M$ are labelled by control variables $x(m)$ of the set C, and the *terminal* nodes $m^T \in M^T = M - M^N$ are labeled by arithmetic or logic expressions $f(m^T)$ on the set of data variables D. For each non-terminal node $m \in M^N$, the graph determines a *mapping* $\Gamma(m)$: $V(x(m)) \rightarrow M(m)$, 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.

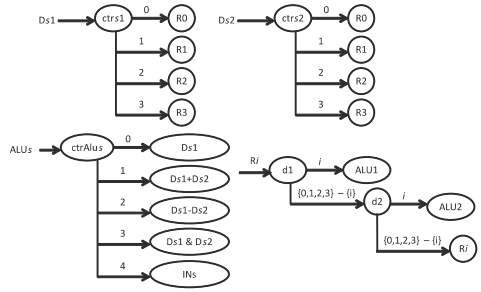


Figure 4. *HLDDs for the components of the VLIW processor*

Definition 3. *Activation of a path on the HLDD*. Denote by m^v the neighbor of the node m for the value $v \in V(x(m))$, according to mapping $V(x(m)) \rightarrow M(m)$. If the test pattern includes the assignment $x(m) = v$, we say that the *edge* from m to m^v *is activated* by the pattern. Each vector of the control variables of C, *activates a path* $l(m,m')$ if all the edges on this path are activated. A full activated path $l(m_0, m^T)$ from the root m_0 to a terminal $m^T \in M^T$ evokes a working mode of the system component $y = f(m^T)$ (e.g. a data transfer or a data manipulation) represented by the terminal node m^T.

Example 1. 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 and right read ports of the processor respectively (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.

III. HIGH-LEVEL FAULT MODELING WITH HLDDS

A. Classification of HLDD-based high-level faults

The HLDD model $G(y,M,\Gamma)$ for a given sub-system or system component $y = F(C,D)$ is well suitable for high-level

fault modeling and fault diagnosis in digital systems, in case when the diagnostic resolution is needed with accuracy of locating faulty blocks represented by the nodes of HLDDs.

The terminal nodes $m^T \in M^T$ of the HLDD represent the sub-functions carried out in the data part, classified as data transfer, storing and data manipulation, whereas nonterminal nodes represent the functions of the control part of a system. 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: (1) *control faults* (the faults related to the non-terminal nodes), and (2) *data faults* (the faults related to the terminal nodes of HLDDs).

In accordance to the VLIW processor model in Fig.4, the fault location targets will be the read port decoding blocks represented by the HLDD nodes ctrs1 and ctrs2, the write port decoding blocks modeled by the nodes d1 and d2, and the ALU control decoding block modeled by ctrALUs. 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 graph ALU1, and the faulty data registers modeled by the nodes R0, R1, R2, and R3 in the graphs Ds1 and Ds2

As the high-level fault model *for non-terminal nodes*, we will base on the exhaustive testing of these nodes. The fault model will include the set of all values of node variables (as the control part of test patterns), and the data constraints to be fulfilled during the exhaustive test of non-terminal nodes. The data constraints will guarantee that all the possible control faults (misbehavior of the non-terminal node) will be activated to produce errors in data part which will be propagated to the points of observation.

For the *terminal nodes* we use a mixed level fault model which combines high-level control fault activation with data constraints generated at low-level to take into account the implementation details of the data part.

B. High-level fault activation using the HLDD model

Test generation for a fault consists of two consistent data assignments where one of them is needed for *activation of the fault*, and the other is needed for *propagation of the fault* up to the point of test response observation.

Each path l in the HLDD from root up to a terminal node $m^T \in M^T$ describes the behavior of the system in a specific working mode related to the function $f(m^T)$. Assignment of control values which activate the path l, activate in the same time a set of faults related to the nodes $M(l)$ on the path l.

Definition 4. *Control fault* $r(m,v)$ of the non-terminal node $m \in M(l)$ at the value of the node variable $x(m) = v$ may cause on the activated path l one of the following fault types in the HLDD model.

(1) *Missing edge*: $(m,v \rightarrow \varnothing)$. The fault brokes the path l , which will mean no change in the state of the system;

(2) *Wrong activated edge*: $(m,v \rightarrow v^*)$. The fault causes incorrect leaving the node $m \in M(l)$ through another output edge of the node, related to the value $v^* \neq v$, which evokes erroneous activation of another path l^* from root up to another terminal node $m^{T^*} \in M^T$, instead of $m^T \in M^T$, and consequently, activates another working mode $f(m^{T^*})$ of the system;

(3) *Several activated edges*: $(m,v \rightarrow V^*(x(m))$ where $V^*(x(m) \subseteq V(x(m)$. The fault causes simultaneous activation of several output edges of the node m causing concurrent activation of several working modes.

Consider a sub-system $Y = F(C,D)$ which is represented by an HLDD $G(Y,M,\Gamma)$. Let us define the following high-level fault activation conditions resulting from Definition 4.

Definition 5. *Control fault activation.* We say that the control faults $r(m,v)$, related to the non-terminal node m and the value $v \in V(x(m))$, *are activated* if the following paths in the HLDD are activated:

(1) A path $l(m_0,m^{T,v})$ from the root node m_0 through m and its neighbor $m^{T,v}$ up to a terminal node $m^{T,v} \in M^T$.

(2) For each neighbor m^{v^*} of the node m, where $v^* \neq v$, a path $l(m^{v^*},m^{T,v^*})$ up to a terminal node m^{T,v^*}, so that any erroneous change of the value $v \in V(x(m))$ would cause activation of another path $l(m^{v^*},m^{T,v^*})$, non-overlapping with $l(m_0,m^{T,v})$, so that $m^{T,v^*} \neq m^{T,v}$.

Definition 6. *Data fault activation.* We say that all the data faults $r(m^T,v)$ related to the terminal node m^T *are concurrently activated* if a full path $l(m_0,m^T)$ from root to a terminal node $m^T \in M^T$, are activated.

IV. HIGH-LEVEL HLDD BASED FAULT PROPAGATION

For generating a test for a given control or data fault $r(m,v)$, the fault must be activated according to Definitions 5 and 6, and propagated to the observation point by solving the suitable constraints for test data values. For data faults related to the terminal nodes $m^T \in M^T$, these data constraints will be determined by the data operands generated for testing the function $f(m^T)$ at the low level by using any ATPG. Let us denote each k-th bit of the value of $f(m^T)$ as $f_k(m^T)$.

In the following we will formulate the data constraints to be satisfied for detecting the activated control faults $r(m,v)$.

Theorem 1. Any erroneous behavior according to the fault types of Def.4 of the sub-system $y=F(C,D)$, represented by a non-terminal node m in the related HLDD $G(y,M,\Gamma)$, will be detected by the test which activates all functional faults $r(m,v)$, $v \in V(x(m))$ under the following constraints:

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

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

where Ω = ZERO (or ONE), and * means $\vee$ (or $\wedge$), depending on technology used in the implementation [13].

Proof. Let us have the technology basis where Ω = ZERO, and * means $\vee$. (The basis where Ω = ONE, and * means $\wedge$ can be considered as a dual case). The constraint (1) results directly from the technology basis, and it covers the first type of faults – the *missing edge*: $(m,v \rightarrow \varnothing)$. For the constraint (2), the fault type of *several activated edges*

$(m,v \rightarrow \{v, v^*\}$ will cause a change of the function $f(m^{T,v})$ into another function $f(m^{T,v}) \vee f(m^{T,v^*})$. To detect the fault, we have to select a set of data words, so that the equation $f_k(m^{T,v}) \neq f_k(m^{T,v}) \vee f_k(m^{T,v^*})$ was satisfied at least once for each bit k in the selected set of data words. It is easy to see that this inequality can be substitute by two relations $f_k(m^{T,v}) < f_k(m^{T,v^*})$ and $f_k(m^{T,v}) > f_k(m^{T,v^*})$ which are simpler to satisfy with bit by bit constraints solving. By generalizing the latter two relations over the set of all pairs of functions $f(m^{T,v})$, $m^{T,v} \in M^T(m)$, we get the same requirement as constraint (2). The fault type of *wrong activated edge* can be considered as a special case of the several activated edges. □

Let us consider the following class of gate-level faults: *stuck-at faults* (SAF), *conditional SAF* (CSAF) and *bridging faults.* CSAF have been called in the past also as *functiona*l [15], *pattern* [16], or *fault tuple* faults [17], or *cell-internal defects* [18]. The latter has been introduced as the fault model for *cell-aware* ATPGs. Let us call this joint fault class as SCB class (**S**AF, **C**onditional SAF, **B**ridging faults).

Theorem 2. Any non-redundant multiple gate-level fault of SCB class in the sub-system $y = F(C,D)$, represented by a non-terminal node m in the related HLDD $G(y,M,\Gamma)$, will be detected by the test which activates all functional faults $r(m,v)$, $v \in V(x(m))$ under the constraints of Theorem 1.

Proof. Consider a block level functional unit $y = F(C,D)$ of a digital system in Fig.6, and a skeleton of its HLDD in Fig. 7. Let the control word C (decoder output vector) be a 3-bit Boolean vector variable $x(m) = (c_2,c_1,c_0,)$ with vector values $v \in V(x(m))$ which activate the respective working modes $y = f_v = f(m^{T,v})$. Denote the k-th bit of f_v as $f_{v,k}$. The data part of the unit consists of 8 sub-circuits for calculating f_v which will be selected by the multiplexer sub-circuit. The latter consists of 8 AND_v blocks which are controlled by the output signals $C = (c_2,c_1,c_0,)$ of the control block. Denote the control inputs of each AND_v block as vector variable $C_v = (c_{v,2}, c_{v,1}, c_{v,0})$. Note, each AND_v block consists of 8 $AND_{v,k}$ gates for each data bit of the function $f_{v,k}$.

Table 1 shows the mapping of low-level single structural faults from the class SCB in the circuit of Fig.6 into high-level functional control faults $r(m,v)$ of HLDD in Fig.7.

Let us call the faults on the control part output lines as global faults (GF), and the faults on the inputs of decoder AND gates as local faults (LF). In case of GF, the same fault has impact as a multiple fault on all AND blocks and AND gates, whereas in case of LF, a fault has impact on a single AND block only.

The rows of Table 1 correspond to the values v of the activated control faults $r(m,v)$ and to the expression $f_v = f(m^{T,v})$ of the related terminal node, as defined by Definition 5. The columns correspond to the faults of SCB partitioned into 5 groups: local SAF, global SAF1, global SAF0, OR type of bridge, AND type of bridge. The entries of Table 1 show which high-level functional faults will be evoked by the low-level structural faults for each activated working mode $y = f_v = f(m^{T,v})$ of the sub-system.

To explain the entries in Table 1 in more detail, consider the example of applying a control vector $C_3 = (c_2,c_1,c_0,) = 011$ as a test for activating the working mode f_3. In column Local SAF we consider only these SAF which coincide with the needed bit values for activating f_3 i.e. $c_2 \equiv 0$, $c_1 \equiv 1$, and $c_0 \equiv 1$. The entry (3, 1) means that in case of local SAF $c_{1,1} \equiv 1$, the activation of f_3 by $v = 011$, will evoke the erroneous execution f_1 as well (this is the fault type of *several activated edges*), which causes erroneous output value $y = f_3 \vee f_1$, instead of the expected correct value $y = f_3$. For the local SAF faults $c_{2,0} \equiv 0$ and $c_{7,2} \equiv 1$, we get the erroneous behaviors $y = f_3 \vee f_2$, noted as (3,2), and $y = f_3 \vee f_7$, noted as (3,7), respectively. For the remaining SAF of the considered local variables, $c_{1,1} \equiv 0$, $c_{2,2} \equiv 1$, and $c_{7,2} \equiv 0$, which are not reflected in Table 1, the operations f_1, f_2, and f_7, respectively, cannot be executed. The same is valid for local bridging faults (in case they don't have impact on the global control lines). The global SAF/1 will cause execution of f_7, instead of f_3. Similarly, for the case of global SAF/0, either f_1 or f_2 will be erroneously executed, instead of expected f_3. The global bridging faults cause instead of f_3 execution of f_7 in case of OR bridge, and either f_1 or f_2 in case of two possible AND bridges. The symbol ∅ has the meaning that at these low-level faults no operation is executed. Since the control word is exercised exhaustively, then all the conditional SAF will be detected as well which corresponds to *cell aware testing* concept as well.

From above, it results that for detecting all the low-level faults when applying the control vector 011, the following constraints must be satisfied: $f_3 \neq$ ZERO (for detecting the fault when f_3 will not be executed), $f_3 \neq f_3 \vee f_1, f_3 \neq f_3 \vee f_2, f_3 \neq f_3 \vee f_7$ (for detecting other local SAF), $f_3 \neq f_7$, (for detecting the global SAF/1 and OR bridge faults), $f_3 \neq f_1$, and $f_3 \neq f_2$ (for global SAF/0 and AND bridge faults).

Similarly, the constraints for all other 7 test vectors on the basis of Table 1 can be derived.

So far, only single low-level structural faults were considered. It is easy to see that the set of multiple faults from the class SCB cannot mask each other. The reason is that there is no fan-out re-convergence in the control logic of the subsystem. Hence, to guarantee the detection of any combination of multiple low-level faults, all the constraints derivable from Table 1 must be joined and taken into account when testing the control faults at the given node m. This requirement coincides with the constraints (2) of Theorem 1. The constraint (1) results from the need of detecting the case of "no operation". □

Let us compare the reduction in the fault model size for the low and high level cases. The total number of 736 low-level faults consists of: (6*8+24)*8=576 local SAF (3 AND inputs and 2 SAF, 8 AND blocks, 12 inverters and 2 SAF), all multiplied by 8 because of the 8 bit data word; 2*3 = 6 global SAF; (2*9)*8 =144 bridging faults. All faults cover all CSAF as well. The number of proposed high-level functional faults will be stated in the following Corollary.

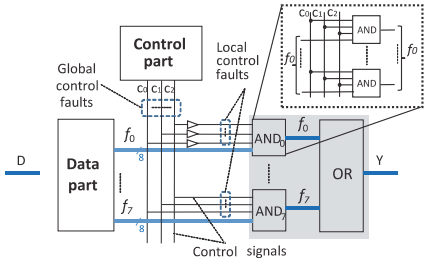


Figure 6. *Functional unit of the system*

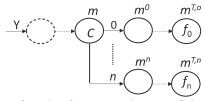


Figure *7. HLDD for the functional unit of the system in Fig. 6*

Tabel 1. Mapping low level structural faults into high-level functional faults

Fault activation		Covered structural faults				
f_i	Control word	Local SAF	Global SAF1	Global SAF0	OR bridge	AND bridge
f_0	000	(0,1),(0,2),(0,4)	1,2,4	∅	∅	∅
f_1	001	(1,0),(1,3),(1,5)	3,5	0	3,5	0
f_2	010	(2,0),(2,3),(2,6)	3,6	0	3,6	0
f_3	011	(3,1),(3,2),(3,7)	7	1,2	7	1,2
f_4	100	(4,0),(4,5),(4,6)	5,6	0	5,6	0
f_5	101	(5,1),(5,4),(5,7)	7	1,4	7	1,4
f_6	110	(6,2),(6,4),(6,7)	7	2,4	7	2,4
f_7	111	(7,3),(7,5),(7,6)	∅	3,5,6	∅	∅

Corollary 1. Consider a node m in the HLDD model with n output edges, i.e. $n = \left| V(x(m)) \right|$. The number of all possible low-level faults of the class SCB of any multiplicity in the control circuit represented by the node *m*, can be mapped by the proposed functional fault model into $n*(n-1) + n$ high-level functional faults.

Proof. Each fault of the proposed high-level functional fault model can be represented by a constraint formulated in Theorem 1. The number of constraints (1) in Theorem 1 is n, whereas the number of constraints (2) is $n*(n-1)$. □

Considering the example in Theorem 1, and Corollary, the number of 736 of single low-level faults can be covered by only 64 functional faults. According to Theorem 2 these 64 single faults can be mapped to any of the multiple fault combination of the 736 single structural faults.

Example 2. Consider the HLDD for ALU of the given slot *s* of the VLIW processor in Fig.4. The data D1, D2 and IN for testing the node ctrAlu and depicted in Table 2 are sufficient for satisfying the constraints (1) and (2) of Theorem 1 at least in one bit of the data words. The number of data bits needed for satisfying the constraints is only 5. Table 3 shows in which bit *k* of the data word f_i and f_j were distinguished $f_{i,k} < f_{j,k}$ (above the diagonal) and $f_{i,k} > f_{j,k}$ (below the diagonal). One of the constraints as a request of Theorem 1 $D1_k < D2_k$ (depicted as "None" in Table 2) cannot be satisfied, which refers to a redundant functional fault.

In this example, the control faults separately for all *n* bits of the data word where not taken into account. This allowed to reduce the number of functional faults i.e. the test generation complexity *n* times. To guarantee the detection of control faults separately for each data bit, we need only to modify the data patterns in Table 2 by shifting these *n* times, so that for each bit, the constraints were satisfied. □

Table 2. *Test data for testing ALU control*

	Data bits, *k*	4	3	2	1	0
Data words	D1	1	0	1	1	0
	D2	1	0	0	1	1
	IN1	-	0	1	0	1
f_0	D1	1	0	1	1	0
f_1	D1 + D2	0	1	0	0	1
f_2	D1 - D2	0	0	0	1	1
f_3	D1 & D2	1	0	0	1	0
f_4	IN	-	0	1	0	1

Table 3. *Highlighted bits where operations are distinguished*

Tested operations f_i	Distinguished operations f_j				
	D1	D1+D2	D1-D2	D1&D2	IN
D1		0	0	None	0
D1 + D2	1		1	1	2
D1 – D2	2	3		4	2
D1 & D2	2	0	0		0
IN	1	3	1	1	

The mapping of low level structural faults into high-level functional faults can help us in the following ways.

1) We can generate directly the test patterns for the control part of the sub-system by using the high-level fault model of lower complexity compared to with the low-level space of faults, similarly to Example 3.

2) We can use the ATPG for generating the test patterns at low level. However, the traditional ATPGs are using the assumption of a single fault case. Since the proposed fault model allows detection of multiple faults, we can re-simulate the test patterns generated by gate-level ATPG using the functional fault model.

3) In the latter case the following may happen. Some functional faults may be not covered by the test patterns generated with ATPG. In this case the missing patterns can be generated using the proposed functional fault model. On the other hand, it may happen that some functional faults may happen to be redundant. If this redundancy has not been detected by the ATPG, the fault coverage calculated by the ATPG tool can be updated (improved).

V. EXPERIMENTAL RESULTS

To compare the gate-level fault coverage obtained by the proposed high-level test generation method with available data in the literature we chose the processor Parwan [19]. The test data for the test program was generated by a gate-level ATPG. The whole test program was simulated by ModelSim to find the local test pattern sequences for all modules. These sequences were fault simulated at gate level to get SAF coverages to compare with the same coverages in [3,20]. From Table 4 we see that the proposed method outperforms in the achieved fault coverage the previously published results for that processor for all modules.

***Table 4**. Comparison of different test coverages for PARWAN MP*

Module	#Faults	Fault coverage %		
		Proposed method	[20]	[3]
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	956	**99.3**	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

The proposed functional fault model had the following contribution in this experiment which explains well the bridge between theoretical elaborations and the experiments.

After using the gate-level ATPG, we achieved for ALU the fault coverage 98.3 [11] which was less than in [3]. After that we carried out the check if all the constraints of Theorem 1 were satisfied to guarantee the full test of ALU control part. The number of 8-bit data pairs for testing all the modules of the processor, including the ALU, was 44. For all these patterns we calculated the coverage of the constraints between the operations of the subset of instructions {LOAD_*A*, *A*&*B*, *A*+*B*, *A*–*B*, NOT_*A*} where A is the data operand stored in the accumulator, and *B* is the second operand at the input of the ALU. As the result, we found that one constraint (one functional fault in all 8 bits) $A \neq A \,\&\, B$ at A = ZERO, i.e the constraint $A_k < B_k$ for all bits k = 0,1,…7 was not covered. It is easy to see that this functional fault is redundant.

Consider, as an example the formula $y = c_1c_2A \vee \neg c_1c_2AB$ where $c1$ and $c2$ are control signals for selecting between 2 operations $y = A$, and $y = A \,\&\, B$. It is easy to see that it is not possible to generate a test pattern for the fault $c_1 \equiv 1$.

In general case, such a redundant high-level functional fault may not have always a related structural redundant fault. This will happen when such a redundancy has been removed during the circuit synthesis process. In the current experiments with Parwan processor, the found functional redundancy motivated us to look deeper in the real circuit to find out the reasons why 100% fault coverage for ALU block was not achieved.

As the result we discovered 6 structural redundant faults which were not covered in our recent research [11]. We also detected other 6 SAF which were not functionally activated, and one undetected structural fault whose redundancy is still not proven.

In such a way, by detecting the redundancy of 8 high-level functional faults, it was possible to increase the ALU fault coverage, compared to the previous work.

VI. CONCLUSIONS

In this paper we developed a new method for high-level modeling and testing of multiple control faults in digital systems. The method is based on modeling of digital systems with HLDDs. We considered in the discussion the class of microprocessors, however, the model of HLDDs can well cover a broader class of digital systems.

The experimental results demonstrated that the new functional fault model defined on HLDDs will cover not only single faults but also multiple faults of a broad SCB class of low level faults and defects. We showed that using the proposed fault model allows detection of gate-level fault redundancies easier than it can be done by gate-level ATPGs, giving in such a way the possibility of creating a synergy from both low- and high-level test generation and fault simulation approaches.

In the future work we will investigate the efficiency of the proposed high-level fault modeling approach for the microprocessors with more complex control mechanisms. The exciting objectives for the future work will be using the HLDDs for high-level test generation for locating faulty blocks in systems as well.

ACKNOWLEDGEMENT

The work has been supported by IT Academy of Estonia, EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds.

REFERENCES

[1] International technology roadmap for semiconductors ITRS. 2001 Edition. Available: http://public.itrs.net/Files/2001ITRS/ Home.htm

[2] 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.

[3] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," in IEEE Trans. on CAD of integrated circuits and systems, vol. 20, no. 3, March 2001, pp. 369 - 380.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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, February 2008, pp. 64-75.

[10] 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.

[11] A.Jasnetski, J.Raik, A.Tsertov, R.Ubar. New Fault Models and Self-Test Generation for Microprocessors using High-Level Decision Diagrams. Proceedings of IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS. Belgrade, Serbia, April 22-24, 2015.

[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, December 14-16, 2015, Dead Sea, Jordan.

[13] R. Ubar. Test synthesis with alternative graphs. IEEE Design and Test of Computers, 1996, pp. 48 - 59.

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

[15] R.Ubar. Fault Diagnosis in Com. Circuits by Solving Bool. Diff. Equations. Automatics & Telemechanics, No.11, 1979, Moscow, pp.170-183 (in Russian).

[16] K.B.Keller. Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms. US Patent 5546408, Aug. 13, 1994.

[17] K.N.Dwarakanath, R.D.Blanton. Universal Fault Simulation using fault tuples. DAC, Los Angeles, June 2000, pp.786-789.

[18] F.Happke et al. Cell-Aware Test. IEEE Trans. on CAD of IC and systems, Vol. 33, No. 9, 2014.

[19] Z.Navabi, Analysis and Modeling of Digital Systems. McGraw-Hill, 1993.

[20] Y.Zhang, H. Li, and X. Li, "Software-based self-testing of processors using expanded instructions," in Proc. of 19th IEEE ATS, 2010, pp. 415 - 420.

Appendix D

Publication IV

Jasnetski, Artjom; Ubar, Raimund; Tsertov, Anton (2017). ”Automated Software-Based in-field Self-Test”. International Journal of Microelectronics and Computer Science, 8 (2), 57−64.

Automated Software-Based In-field Self-Test Program Synthesis

Artjom Jasnetski, Raimund Ubar, and Anton Tsertov

***Abstract*—This paper presents a methodology to automate functional Software-Based Self-Test program development. We rely on the previously published research on modeling processors using subclass of acyclic directed graphs called High-Level Decision Diagrams (HLDD). The HLDD model of the processor gets generated from its Instruction Set Architecture. The HLDD model is then used together with beforehand prepared assembly program templates in the generation of the complete self-test program. The research presented in this paper includes examples of test generation for the 32-bit SPARCv8 microprocessor Leon 3. The experimental results demonstrate that automatically generated SBST program obtains comparable to the state-of the art fault coverage data.**

***Index Terms*—microprocessor, software-based self-test (SBST), automatic test program generation, high-level decision diagrams (HLDD) synthesis.**

I. Introduction

ADVANCES of modern technology in manufacturing and design of microprocessors are continuously increasing the difficulty of digital circuit test [1]. Therefore, testing of constantly scaling complex digital systems like microprocessors, has been a challenge for decades. Software-based self-test method has emerged, and became a very promising competitor to the widely used, but slow, intruisive structural test [2] [3], and effective, but very expensive functional test [1]. The core idea of SBST approach is to use the resources of microprocessor to test itself, by running test programs. The nature of this method implies such features as nonintruisiveness, low cost and compatibility with at-speed and in-field testing [4]. This method was accepted by industry [5], and is complementing functional and structural methods within manufacturing process.

Furthermore, interest for this method was growing in frames of in-field test for processor-centric systems in safety-critical applications. Since, functional and structural test methods are not suitable for in-field test, SBST becomes very attractive solution [6] [7]. The academia was motivated to put more effort into studying SBST for in-field test, after publication of IEC 61508 for industrial safety systems, ISO 26262 for automotive applications, and release of DO0254.

Concurrently, big interest is gathered around the automation of SBST approach, since the complexity of manual test program generation can be inexcusably high. Automated SBST [7]–[9] can reduce the test development cost, and thereafter price of a product.

SBST approaches can be divided into two major groups - structural and functional. Structural approaches, such as [10]–[15], 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. Since in most cases structural information of commercial products is intellectual property held under NDA, the solution based on functional SBST is exclusive for in-system or in-filed test.

One of the first methods among functional SBST proved to be efficient, was proposed by Shen And Abraham [16]. Framework Vertis, capable of pseudo-random test sequence generation based on ISA information, has been proposed. Similar solution - FRITS (Functional Random Instruction Testing at Speed) [17] is based on test program generation on random instruction sequences with pseudo-random data. It suits well for wafer test, due to it's cache-resident nature. Alternative cache-resident method for production testing [5] using random generation mechanism proves, that high cost functional testers can be replaced by this SBST approach, without significant loss in fault coverage. Alternative approach, based on so-called evolutionary algorithm, was proposed by Corno et al. [18]. Test program is being composed of the most effective code snippets (in a question of SAF coverage), which were distinguished by constant reevaluation. Due to it's reevaluation-centric nature, this method is not capable of in-field test generation, due to lack of structural information. Later research concentrates on test approaches for specific processor parts like pipeline, branch prediction mechanism [19] or caches [20]. Gizopoulos et al. in [21] are proposing a method, which can enhance SBST program in order to bring more coverage to pipeline logic and also memory addressing (12% for miniMIPS and OpenRISC1200 processors). Further approach for testing the pipeline was made by Bernardi et al. [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 [6].

Nevertheless, none of the state-of-the-art methods have so far tried to develop well formalized high-level (e.g. behavioral level) fault models for coping with hard-to-test faults and fault masking problems at higher levels with ultimate goal to improve test quality and to achieve compact test programs. Additionally, coverage of wide specter of fault classes is left unmeasured, due to lack of methods for simulation and lack of theoretical basis for identification.

A. Jasnetski, R. Ubar and A. Tsertov are with Department of Computer Engineering, Tallinn University of Technology, Tallinn, Estonia (e-mails: ajasn@ati.ttu.ee, raiub@ati.ttu.ee, anton.tsertov@ttu.ee)

This work was jointly supported by IT Academy of Estonia, EU through European Regional Development Fund, H2020-ICT-2014-1 644905 project IMMORTAL as well as by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research.

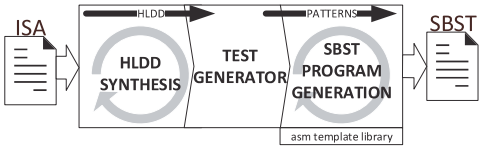


Fig. 1. General concept of SBST generation tool

Our previous work in SBST field is focused on adapting methodology of High-Level Decision Diagrams (HLDD) for modeling of microprocessors and faults [23]. In paper [23], we introduced a new concept for generating tests for microprocessors. The proposed approach considers program generation using MUTs modeling at behavioral level derived from ISA. HLDD model sets are synthesized from the ISA, hence, considered as a behavioral level model. The instruction list of MP is converted into a network of HLDD graphs where each graph represents a sub-circuit. Hierarchical approach is used for test generation: the control functions are tested exhaustively (by conformity test), but the data operands for testing the data path are generated by traditional gate-level ATPG (scanning test). Our latest work [24] adds new fault modeling idea for high-level faults, where experimental results obtained by formal generation of test program for the Parwan [25], [26] microprocessor are presented.

In this paper we develop our previous concepts into the tool, which automatically generates test programs for microprocessors. For that purpose we have proposed an algorithm of instruction set analysis. Generalizing the known method of BDD synthesis based on Shannon expansion of Boolean functions allows high level expansion of predicate expressions. The general concept of the tool is shown in Figure 1. The Tool consists of three modules: HLDD synthesizer, test vector generator, and SBST generator - synthesizer for converting test vectors into test-programs using beforehand prepared test code templates. The capabilities of the tool are demonstrated on two microprocessors - on the Parwan 8-bit microprocessor, and on Leon 3 32-bit microprocessor.

The paper is organized as follows: Section II presents the basis of HLDD synthesis procedure. Section III is demonstrating the HLDD synthesis functionality of proposed tool on example of Leon 3 microprocessor. Section IV is dedicated to fault modeling using HLDD diagrams. Section V gives an overview of test program generation functionality, and experimental results. The results, published in this paper are confirming the applicability of approaches presented in previous works [23] [24].

II. High-Level Decision Diagram Synthesis

In [27], High-Level Decision Diagrams (HLDD) were introduced, and a method was proposed for synthesis of HLDDs from Data Flow Diagrams (DFD). As the first step of synthesis, the DFD was transformed by symbolic execution into a Structural Table of Automaton (STA) [28]. In case of microprocessors when their behavior is given by the instruction set, such STA can be generated directly without symbolic execution of the model.

Consider an example of a fragment of a digital system is shown in Table I.

TABLE I
Description of an Automation

Instruction	Control constraints			Data assignment statements
I_k	x_1	x_2	x_3	
I_1	1	3	0	$y_1 = F_1(X)$
I_2	1	2		$y_1 = F_2(X)$, $y_2 = F_3(X)$
I_3	3		2	$y_1 = F_4(X)$
I_4		4	1	$y_2 = F_5(X)$
- - -				

Each row in Table I represents a state transfer in the automaton. In case of the microprocessor represented at high-level by its set of instructions, we can represent by each row the functionality of an instruction as follows.

The left-most cell in each row denotes the name (or the number) of the instruction I_k, the cells of the subtable Control constraints represent the code of the the related instruction word I_k split into the codes (x_1, x_2, x_3) of different subfields of the instruction format, and the cells of the subtable Data assignment statement represent the functional activities $y_k = F_k(X_k)$ of the related instruction where y_k denotes the output functional variable of the related functional block (e.g. the output register of ALU), and X_k represents the data variables involved as arguments in the data manipulation operation F_k.

For all the left-hand side variables y_k in Table I we create HLDDs which will describe the behavior of these variables during execution of the related instructions. We assume that the variables y_1 and y_2 represent the results of functions $F(X)$.

Consider the data in Table 1 as a set of tuples $N = \{N_k\}, N_k = (C_k, S_k)$ where C_i is a set of logical constraints given for the instruction I_k in the subtable control constraints, and S_k is a set of assignment statements. Each statement $s \in S_k$ (denoted by shortly by $y_{k,s} = F_{k,s}$) is an algebraic expression, which will be fulfilled if the set of constraints C_i is satisfied. By collecting all the statements s from N for a left-hand variable y we can represent the behavior of y as

$$y = \bigvee C_i F_{i,s}, \tag{1}$$

where the constraints C_i represent conjunctions of predicates weighted by the respective expression $F_{i,s}$.

As an example, according to the formula 1 the Table I can be represented now as the two predicate formulas:

$$y_1 = (x_1 = 1)(x_2 = 3)(x_3 = 0)F_1(X) \vee$$

$$(x_1 = 1)(x_2 = 2)F_2(X) \vee (x_1 = 3)(x_3 = 2)F_4(X)$$

$$y_2 = (x_1 = 1)(x_2 = 2)F_3(X) \vee (x_2 = 4)(x_3 = 1)F_5(X)$$

From these predicate formulas, the HLDDs for the variables y_1 and y_2 can be derived in a similar way as BDDs are derived by using Shannon factorization [29] for Boolean functions. The only difference is that instead of Boolean factorization we will use multi-valued factorization, depending on the possible number of values of the constraint variables x_k. The HLDDs created by factorization for y_1 and y_2 are depicted in Fig 2.

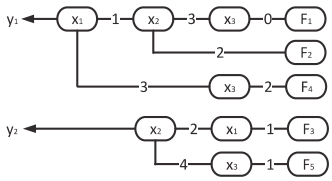


Fig. 2. HLDDs created by factorization of formulas for y_1 and y_2

Using the model of HLDDs we can simulate the instructions by tracing the graphs according to the values of the instruction variables (x_1, x_2, x_3). If a terminal node will be reached then the value of the graph variable y is updated by calculation of the value of the expression in the related terminal node. If no terminal node will be reached then the value of graph variable y will not be updated.

III. ISA Based HLDD Synthesizer

The methodology for generation of High-Level decision diagrams from instruction set architecture presented in previous section, is used by the proposed tool. The instruction set information is primary input data for HLDD synthesis. In order to process ISA automatically, it should be represented in a machine readable way. Usually the information about ISA is formatted and composed differently, making the universal parsing process nearly impossible.

First, we suggest to bring the ISA description to common ground. Thus, in this paper we outline two formats (CSV, XML) to generalize the description of Leon 3 ISA. In case of CSV format, each instruction field name, width and value must be provided using such syntax: $\%name\% = \%width\%'b\%value\%$. In case of XML format, each field must have own tag, where name, width and value are provided as follows: $< \%name\% = \%width\%'b\%value\% >< \%name\% >$. The proposed HLDD Synthesizer tool is capable to read both formats: CSV table and XML.

As a case study, microprocessor Leon 3 was used. This processor was chosen to represent the complexity of modern processor cores, which is suitable to show the scalability of proposed approach. Table II shows the complexity difference between two processors used in case study of this paper. An integer unit module of Leon 3, as a core component, is used as an unit under test.

TABLE II
PARWAN and Leon 3 IU comparison

	PARWAN	LEON 3 Integer Unit
Bit depth	8	32
Instructions	16	46
Architecture	Custom	SPARCv8
Gates	1480	15161

Leon 3 is a microprocessor with SPARCv8 architecture [30]. SPARCv8 instructions can be divided into four groups: memory, control, ALU and miscellaneous. Each instruction is aligned to a specific format. SPARCv8 architecture has 6 different formats for instruction set listed in Table III.

TABLE III
SPARCv8 instruction formats

1	op	disp30						
2	op	rd		op	imm22			
3	op	a	cond	op2	disp22			
4	op	rd		op3	rs1	i=0	asi	rs2
5	op	rd		op3	rs1	i=1	imm13	
6	op	rd		op3	rs1	opf		rs2

The fields from instruction format table can be divided in two types - operational fields (*op, op2, op3, i, a, asi, cond, opf*) and register related fields (*rs1, rs2, rd, imm*). This information must be handled differently, in order to build a proper HLDD diagram. Operational code fields must be marked differently from register fields.

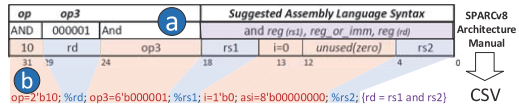


Fig. 3. CSV and XML representation of AND instruction

An example of *AND* instruction of SPARCv8 architecture is provided in Figure 3. This figure represents the process of modifying the instruction description from architecture reference manual into machine readable format. In the given example *AND* instruction description (Figure 3.a) is cut from SPARCv8 architecture manual. By it's format *AND* instruction belongs to the ALU instruction group. Additional information about instruction fields is also taken from manual - instruction word consists of four operational fields - *op, op3, i* and *asi*, and three register fields - *rd, rs1* and *rs2* (Table III, line 4). Based on this information, an entry containing instruction field names and their length, can be added to the CSV file (Figure 3.b). *op, op3, i* and *asi* fields have constant values, but *rd, rs1* and *rs2* fields are dynamic, since holding information about register index. Dynamic nature of register index is represented with symbol %. Complementary information should be provided as parameters separately, in order to sort dynamic fields by their specification - if it is a source register, destination register or immediate value.

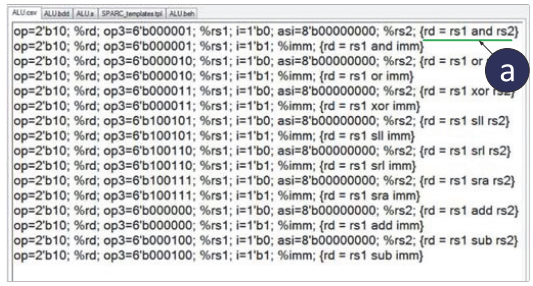


Fig. 4. Part of SPARCv8 ALU instructions

In case of *AND* instruction, *rd* is a destination register index and *rs1, rs2* are source register indexes. Additionally, CSV entry should contain the description of operation (Figure 4.a),

for the *AND* operation in hand is *rd = rs1 and rs2*. Complementary, but important information about microprocessor architecture should be also added as separate parameters. Such is the data about register amount and their width, needed at the stages of HLDD generation and test synthesis.

Correctly composed CSV with complementary files is holding needed data to build HLDD diagram, representing the behavior of the system (or it's part) under test. As an example, a small subset of SPARCv8 instruction set, representing ALU instructions, is shown in Figure 4. The HLDD graph, synthesized from SPARCv8 ALU-type instruction list is shown in Figure 5. In case of *AND* instruction, the destination register, represented by instruction field *rd*, becomes the output of the graph. Then, path of consequent nodes from output to the leaf of graph is build from operational fields of *AND* instruction. Functional description of *AND* instruction becomes the leaf. Such way of modeling allows to store the behavior of the system as follows: the result of operation *rs1 and rs2* will be stored to *rd*, if operational instructions fields *op, op3, i* and *asi* are holding specific values. As a consequence, each node represents an element of the control part, and leafs represent data path of the modeled system.

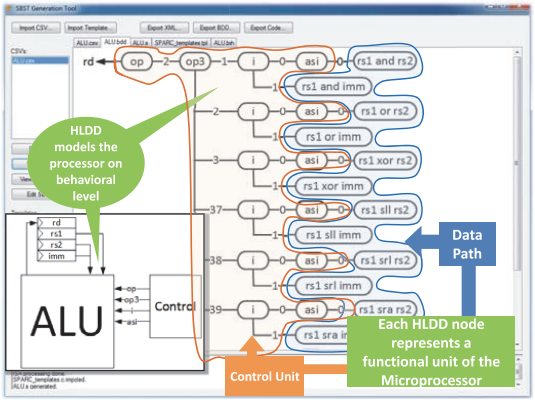


Fig. 5. HLDD graph synthesised from SPARCv8 instructions

IV. ISA Based High-Level Fault Modeling

In the ISA based HLDD model, the nodes of the decision diagram (DD) are classified into two groups: internal nodes, and terminal nodes. Internal nodes represent the control functions of the system, and terminal nodes represent the data path functions of the system.

The instruction words of microprocessors are usually split into several fields. This corresponds to partitioning of the instruction variable into concatenation of the field variables e.g. as *I = OP.A1.A2*, where *I* is the instruction variable, *OP* is the operation code, whereas *A1*, and *A2* denote register addresses of the first and second operands, respectively. In the HLDD model, to each of these field variables related internal nodes correspond. On the other hand, each of these nodes represent sub-circuits which are responsible for addressing the operands and controlling the operation related to the value of *OP*. The nodes *OP*, *A1*, and *A2* represent a path in the HLDD, which will be activated if the instruction *I* is called. The path terminates in the terminal node of the HLDD labelled by a functional expression to be processed in the data path of the system if the instruction *I* is called.

Each path in a DD describes the behavior of the system in a specific mode of operation. The faults having effect on this behavior are associated with nodes along the activated path. In case of a control fault, the path activated by instruction *I*, will be corrupted, and the effect of the fault will cause incorrect leaving the path in the faulty node. In this case, a wrong terminal node will be reached instead of the terminal node which should have been reached at instruction *I*. In case of the data fault, the functional expression in the terminal node of the activated path will be corrupted.

As the *fault universe* to represent all possible control malfunctions in a system, we assume any corruption in the behavior of non-terminal nodes in HLDDs, expressed in the following ways [31]:

1) output edge of a node is broken;
2) output edge of a node is constantly activated;
3) instead of the activated edge, another edge or a set of edges are simultaneously erroneously activated.

The practical meaning of this fault universe stands in application of the idea to test exhaustively the behavior of each non-terminal node at all possible values of the node variable.

The fault universe of control faults can be expanded with the universe of data path faults related to terminal nodes of HLDDs to get the full fault universe of the system under test. To do this, in each terminal node, and for each expected value v of the functional expression of the node, we introduce a dummy output edge into a new dummy terminal node labelled with the value v as a constant. In such an extended HLDD, all the node related faults can be activated in a uniform way.

Several optimizations of the described exhaustive fault universe can be undertaken, based on either transforming exhaustive test into pseudo-exhaustive one, or transforming the functional test into the structural one.

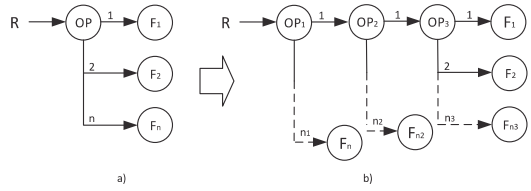


Fig. 6. Transformation of HLDDs to reduce the complexity of model

In Figure 6, it is shown how the exhaustive test concept can be substituted by the pseudo-exhaustive approach to reduce the control fault universe of the system. Two possible HLDDs are presented for representing the behavior of the same subsystem with output register *R*. The subsystem is controlled by a set of *n* instructions. The instruction variable *I* can be represented as a concatenation of the field variables in two ways: *I = OP.A1.A2*, or *I = OP1.OP2.OP3.A1.A2*, resulting in two different HLDDs.

Instead of testing exhaustively the node *OP* in the HLDD (Figure 6a) for all *n* values of the instruction subfield variable *OP*, we may test separately and exhaustively the component variables *OP1*, *OP2*, and *OP3* in the functionally equivalent HLDD (Figure 6b), which corresponds to traditional pseudo-exhaustive test [31]. The complexity of the test generation task, and the length of the resulting test can be using the HLDD in Figure 6b considerably reduced due to the reduction of the number of output edges of internal nodes in the graph, compared to the HLDD in Figure 6a:

$$n_1 + n_2 + n_3 << n = n_1 \times n_2 \times n_3.$$

On the other hand, the exhaustive test of the functions *Fk* in terminal nodes, can be replaced either by also pseudo-exhaustive tests [32], or by structurally generated test patterns using any traditional low-level ATPG (if the related implementation details of the system are given). In the latter case, a combined use of hierarchical test generation may be used, exploiting both, ISA-based HLDDs, and structurally synthesized BDDs [33].

To organize test generation for each HLDD node, using HLDD-based fault universe, two steps are to be processed: activation of the node under test, and sensitizing the faults of the activated node.

To activate internal node *m* under test needs assignment of proper values to the node variables in the HLDD, so that the following paths were activated: (2) a path from the root node to *m*, and (3) a subset of non-overlapping paths from all output edges of the node *m* to a subset of terminal nodes $M^T(m)$. To activate a terminal node *m* under test needs to activate only a single path from the root node to *m*. Sensitizing the faults of the activated internal node m needs to solve the following equations as constraints when testing the node *m* [2]:

$$\forall m^T \in M^T(m) : [f(m^T) \neq \Omega], \tag{2}$$

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

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [34], [35]. Here, ZERO denotes a binary vector (000), and, similarly, ONE stands for (111). The index *k* refers to the bit number of the data words. Satisfaction of the constraints (2) and (3) guarantees that the expected and erroneous test responses will be distinguished at any corruption of the activated HLDD node *m* under test due to a fault in the sub-system, represented by the node.

The described high-level fault model defined for HLDDs, together with the node activation concept, can be regarded as a generalization of the classical gate-level stuck-at fault (SAF) model for high-level representations of digital systems. Both represent a *node based fault model* in decision diagrams [33]. The only differences are in the number of output edges of the nodes, and in the number of terminal nodes in the decision diagrams. In both cases, the node variables are tested exhaustively: two test patterns are needed for the Boolean variables labelling the nodes in BDDs whereas the number of patterns needed for HLDD nodes is equal to the number of output edges of the node.

The described concept of satisfaction of the constraints (2) and (3) is similar to the extended conditional SAF model [36]–[38] developed for Boolean level test generation of physical defects inside complex gates in digital circuits. In the latter case, additional conditions map the impact of defects into SAF at related BDD nodes, whereas in case of the ISA based HLDD model, the constraints (2) and (3) specify the reasons of the corruptions in behaviors of nodes under test.

V. SBST Generation with HLDD Model

The targets of test generation for a microprocessor using the HLDD model are not the instructions each of them taken as a whole as in traditional cases. Instead of that, the targets are smaller functional entities represented by the nodes of HLDDs. The terminal nodes represent selected data path functional entities (sub-circuits of ALU), and the nonterminal nodes represent the selected control functional entities related to the subfields of instruction words. Since the HLDD nodes as test targets represent smaller functional units than the instructions as a whole, it makes possible to use pseudo-exhaustive testing of the processor control part and to cope in this way better with the complexity of the test problem. Instead of full exhaustive testing of all operation codes we test (pseudo)exhaustively its independent parts, guided by the HLDD internal nodes. For testing terminal nodes we use test data generated for ALU at the gate level.

From above, two approaches of testing, different for terminal and nonterminal nodes, result: conformity test for the control part (internal HLDD nodes), and scanning test for data path (terminal HLDD nodes) [39].

A. Conformity tests

The test program is synthesized on the high-level directly from the HLDD model, and the data for the test program are generated to satisfy the constraints (2) and (3).

Algorithm 1. Conformity test for the control part (test for a nonterminal node m).

1) Control data (instruction code) generation: activate in the HLDD a path lm from the root node to the node m under test, and for each output k of the node m a path l_k to a terminal node m_k^T with operation $f(m_k^T)$. The value of $z(m)$, which represents a sub-field of the instruction code, will be cyclically varied during the pseudo-exhaustive test execution.
2) Data path initialization: find the proper sets of data values $D(m)$ which satisfy the constraints (2) and (3).
3) Test implementation: the generated instruction should be repeated for all the values $z(m)$ of the node m under test, updated dynamically by these values, and using always the same data operands in $D(m)$.

B. Scanning tests

The test program is synthesized on the high-level directly from the HLDD model, and the data for the test program are

generated by a traditional gate-level ATPG using the given descriptions of the data blocks.

Algorithm 2. Scanning test program generation for testing the data path (terminal node m) for operation $f(m)$.

1) High-level test generation: activate in the HLDD a path l_m from the root to the terminal node m.
2) Low-level test generation: find the proper sets of data values $D(m)$ for the arguments in $f(m)$.
3) Test implementation: the generated instruction should be repeated for all the values of $D(m)$, i.e. for all of the arguments of $f(m)$.

C. SBST generation with a tool

The proposed tool, is utilizing these ideas in the test program generation process. The result of test generation, is a test pattern, which holds encoded information about instruction and operands (Figure 7(A)). Since, there is normally no framework available to handle test program for microprocessor in machine code, the task of SBST generator (see Figure 1) is to decode patterns (Figure 7(B)) obtained from test generator into assembly instructions. This is done by using predefined templates stored in the assembly code library. As a result, the test program, compiled from code templates is made. It can be edited further, in order to improve the fault coverage, or add code parts, which can not be generated automatically.

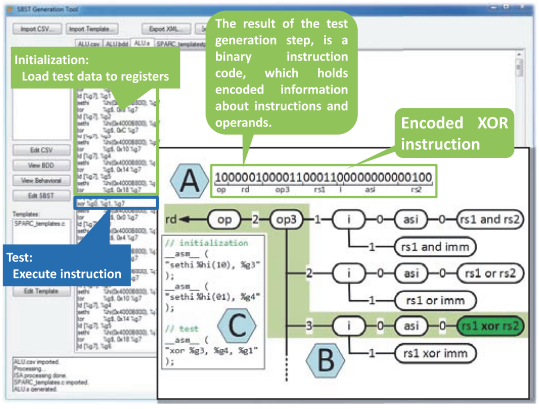


Fig. 7. Example of test program generation

The test program generation process is shown in general in Figure 7. The process can be divided into two parts - initialization and test. The initialization part is loading test data into registers, and the test part is combining the instruction fields from the library into the full instruction code. In Figure 7(C), a subset of generated test program is shown. The first part of code represents an initialization process. The registers are filled with data (partially shown). Every register is loaded with data before testing each instruction, so that to avoid fault masking [24]. Then, the test part is being generated. In the section A of Figure 7, a test pattern string, retrieved from the test generator is shown. Since the instruction fields are known (Table III), the test generator can walk trough HLDD nodes, and construct the corresponding instruction. A path, highlighted in green leads to the instruction XOR (Figure 7(B)). Test generator is looking for XOR assembly code template in premade SPARCv8 library, and modifies it to read data from registers, specified by the test generator (rd, $rs1$ and $rs2$ fields) (Figure 7(A, C)).

Test program generation is strongly affected by the modeling level made in previous steps. The more details can be extracted from instruction set architecture, the better test program can be generated. Specific behavior of the processor can be hidden or even invisible from the ISA point of view, and simple list of instructions not enough to cover the realistic structural results.

The exact fault coverage can be calculated by gate-level fault simulation. The not detected gate-level faults may belong to the class of redundant faults. Otherwise, to detect these faults, low-level ATPGs can be used for generating additional test operands.

The main contribution of the proposed method of SBST generation is to substitute existing labour extensive low-level test generation methods with fast high-level test generation, accompanied with as well fast low-level fault simulation to obtain the exact evaluation of the test quality.

VI. Experimental Results

In the experiments, a test program was generated automatically for the Integer Unit (IU) of Leon 3. The Fault simulation of the generated test program was made using TetraMAX [40] software. The fault simulation framework is described in details in our previous work [24]. The results are shown in Table IV. "HLDD test program" shows a result of fault simulation with automatically generated test program. "HLDD test program random" is automatically generated, but test operands are selected randomly. "Leon3 startup test" is a test program supplied with processor description files [41], which tests memory and peripherals on startup. "TetraMAX ATPG" represent a local fault coverage of patterns generated by a sequential ATPG tool. However not all generated test vectors are functionally correct (they cannot be reproduced during normal CPU operation), hence the coverage is overestimated. The row "HLDDtp + Leon 3 st" represents the fault coverage result for both test programs. According to calculated fault coverage, we can assume, that those programs can complement each other and cover additional faults in components of Integer Unit. From the point of view of test engineer, our tool is a good opportunity to improve fault coverage (about 5%), with minimal effort.

The low fault coverage is explained by the fact that not all instructions using the Integer Unit were taken into account for building the HLDD model. Extension of the model for the full instruction list needs further research.

However, to demonstrate the feasibility and efficiency of the approach for the case when the HLDD model covers full set of microprocessor instructions, the experimental research was carried out for the PARWAN microprocessor. The results in comparison with previous research on PARWAN [42], [43] are depicted in Table V, which demonstrates the superiority of our results. The PARWAN was used because of the availability of published results for that microprocessor.

TABLE IV
LEON 3 INTEGER UNIT FAULT SIMULATION RESULTS

Leon 3 Integer Unit	Faults total/testable	FC, %	Fault simulation, in minutes
HLDD test program	42780 / 38847	43,84	34
HLDD test program random		41,40	78
Leon 3 startup test		40,93	22
HLDD tp + Leon 3 st		45,26	78
Tetra max ATPG		72,89	2496 *

** time used for ATPG and fault simulation together*

TABLE V
PARWAN FAULT SIMULATION RESULTS

Module	#Faults	Fault coverage %		
		Proposed method	[43]	[42]
AC	156	**99.3**	99.30	99.30
IR	228	**99.4**	96.40	98.60
PC	590	**99.3**	99.00	89.20
MAR	342	**99.2**	96.40	97.20
SR	130	**99.0**	96.80	98.90
ALU	956	**99.3**	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 this experiment with Leon processor we concentrated on test generation for the data path of the IU, which is related to fetch, decode and memory stages of the pipeline [30], other stages are tested indirectly. Moreover, since we concentrated our efforts only for IU-related instruction groups (ALU and memory), then a lot of control part functionality was not covered by HLDD test, like state, flags, traps, FPU and Coprocessor instructions and controling rotation of register windows (exclusive for SPARC architecture). This explains the low fault coverage. The further work will be to extend the not yet covered hardware part responsible for the unused instruction groups. Still, the composed program was able to discover a considerable amount of faults, which weren't covered by default Leon 3 test program. Test data, which was selected by test generator gives better fault coverage, than random data. The fault simulation time is increased in case of random data, because the amount of data is multiple times more than in case of deterministic test data generation.

VII. CONCLUSION

In this paper we developed first, a novel algorithm and a tool for formal synthesis of the HLDD model for a given set of the instructions of the microprocessor under test, and second, a tool for automated software-based self-test program generation for microprocessors based on the HLDD model.

The novelty of described tool is an automation of test program generation. The capabilities of the tools are demonstrated on the 8-bit microprocessor PARWAN, and on the 32-bit Leon 3 SPARCv8 microprocessor. In combination with the fault simulation tools, the described in the paper test generation tool promises to be a helpful instrument for test engineers. The positive fault coverage results, obtained during evaluation of the test program with TetraMAX simulator, are confirming the feasibility of the proposed approach.

However, because of the unique features of the investigated Leon microprocessor architecture, a fully-automatic approach for its full instruction set is not available at the moment. Still, test engineer can modify generated test program in order to increase fault coverage.

A field of the future work exists. A backwards analysis of assembly program, can show, which paths are covered on the HLDD model of the microprocessor, and make an approximate fault coverage estimation of the test program. This will decrease test program development time, since fault simulation, especially sequential, takes lots of time and computational resources.

REFERENCES

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

[2] E. B. Eichelberger and T. W. Williams, "A logic design structure for lsi testability," in *Papers on Twenty-five Years of Electronic Design Automation*, ser. 25 years of DAC. New York, NY, USA: ACM, 1988, pp. 358–364. [Online]. Available: http://doi.acm.org/10.1145/62882.62924

[3] E. B. Eichelberger and T.W.Williams, "A logic design structure for LSI testability," in *Proc. of the DAC*, 1977, pp. 462 – 468.

[4] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, May 2010.

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

[6] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. Reorda, S. D. Luca, R. Meregalli, and A. Sansonetti, "On the in-field functional testing of decode units in pipelined risc processors," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 299–304.

[7] A. Riefert, R. Cantoro, M. Sauer, M. S. 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, no. 10, pp. 3055–3066, Oct 2016.

[8] 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)*, March 2014, pp. 1–6.

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

[10] 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*, Oct 2006, pp. 1–9.

[11] 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, no. 5, pp. 518–530, May 2007.

[12] 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*, Oct 2006, pp. 1–9.

[13] C. H. P. Wen, L.-C. Wang, and K.-T. Cheng, "Simulation-based functional test generation for embedded processors," *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1335–1343, Nov 2006.

[14] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, April 2005.

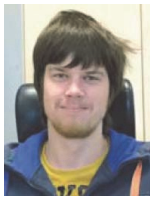
[15] C. H. Chen, C. K. Wei, T. H. Lu, and H. W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 5, pp. 505–517, May 2007.

[16] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, Oct 1998, pp. 990–999.

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

[18] 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, pp. 102–109, Mar 2004.

[19] 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, no. 9, pp. 1675–1688, Sept 2015.
[20] S. D. Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, July 2011.
[21] 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, no. 11, pp. 1441–1453, Nov 2008.
[22] 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*, Dec 2013, pp. 52–57.
[23] R. Ubar, A. Tsertov, A. Jasnetski, and M. Brik, "Software-based self-test generation for microprocessors with high-level decision diagrams," in *Proc. of the Latin-American Test Workshop*, 2014, pp. 1–6.
[24] A. Jasnetski, J. Raik, A. Tsertov, and R. Ubar, "New fault models and self-test generation for microprocessors using high-level decision diagrams," in *Proc. of the International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2015, pp. 251–254.
[25] Z.Navabi, *Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.
[26] [Online]. Available: http://mesdat.ucsd.edu/ lichen/260c/parwan/
[27] R.Ubar, J.Raik, A.Karputkin, and M.Tombak, "Synthesis of high-level decision diagrams for functional test pattern generation," in *Proc. of 16th Int. Conference MIXDES*, Lodz, Poland, Jun. 2009, pp. 519–524.
[28] S. Baranov, *Logic and System Design of Digital Systems*. Tallinn, Estonia, year =: TUT Press.
[29] B. R. Drechsler, *Binary Decision Diagrams*. Boston, MA, USA: Kluwer Academic Publishers, Boston, 1998.
[30] S. international Inc. The sparc architecture manual, version 8. [Online]. Available: http://www.gaisler.com/doc/sparcv8.pdf
[31] E. J. McCluskey, "Verification testinga pseudoexhaustive test technique," vol. Vol. C-33, pp. 541 – 546, 07 1984.
[32] 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)*, June 2017, pp. 1–6.
[33] R. Ubar, "Test synthesis with alternative graphs," in *IEEE Design and Test of Computers*, 1996, pp. 48 – 59.
[34] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," vol. C-29, no. 6, June 1980, pp. 429–441.
[35] D. Brahme and J. A. Abraham, "Functional testing of microprocessors," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 475–485, June 1984.
[36] R. D. Blanton and J. P. Hayes, "Properties of the input pattern fault model," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, Oct 1997, pp. 372–380.
[37] S. Holst and H. J. Wunderlich, "Adaptive debug and diagnosis without fault dictionaries," in *2008 13th European Test Symposium*, May 2008, pp. 199–204.
[38] 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, Sept 2014.
[39] R.Ubar, J.Raik, and H.-T.Vierhaus, *Design and Test Technology for Dependable Systems-on-Chip*. IGI Global, 2011.
[40] [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx
[41] [Online]. Available: http://www.gaisler.com/products/grlib/grlib-gpl-1.4.1-b4156.tar.gz
[42] 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, pp. 369 – 380.
[43] Y. Zhang, H. Li, and X. Li, "Software-based self-testing of processors using expanded instructions," in *Proc. of 19th IEEE Asian Test Symposium*, 2010, pp. 415 – 420.



Artjom Jasnetski received his M.Sc.degree in computer engineering from Tallinn University of Technology, Estonia in 2013 and currently he is Ph.D. student at Tallinn University of Technology. His research interests include such topics as microprocessor test, digital system modelling, ISP and HW driver implementation.

Raimund Ubar is a member of IEEE, professor at Tallinn University of Technology, and Head of the Centre for Integrated Electronic Systems and Biomedical Engineering in Estonia. He received PhD degree in 1971 from the Bauman Technical University in Moscow, and DSc degree in 1987 from the Latvian Academy of Sciences. His scientific interests include computer science, design for testability and diagnostics of technical systems. Raimund is a member of European Test Technology Technical Committee, a member of Estonian Academy of Sciences, and Golden Core member of IEEE Computer Society. He was awarded from the Estonian Government by White Cross Orden of III Class, and by Meritorious Service Award of the IEEE Computer Society.

Anton Tsertov received his M.Sc. and Ph.D. degrees in computer engineering from Tallinn University of Technology, Estonia in 2007 and 2012 respectively and currently holds the position of researcher in Tallinn University of Technology. His research interests include such topics as system and board level test, high-level system modelling, microprocessor functional and structural test.

Curriculum vitae

Personal data

Name:	Artjom Jasnetski
Date of birth:	20.05.1988
Place of birth:	Narva, Estonia
Citizenship:	Estonian

Contact data

Address:	ICT-511, Akadeemia tee 15A, Tallinn 12618
E-mail:	artjom.jasnetski@ttu.ee

Education

2013 – 2018:	Tallinn University of Technology – PhD
2010 – 2013:	Tallinn University of Technology – MSC
2007 – 2010:	Tallinn University of Technology – BSC

Professional employment

2011 – …:	Testonica Lab OÜ, test engineer

Elulookirjeldus

Isikuandmed

Nimi:	Artjom Jasnetski
Sünniaeg:	20.05.1988
Sünnikoht:	Narva, Eesti
Kodakondsus:	Eesti

Kontaktandmed

Aadress:	ICT-511, Akadeemia tee 15A, Tallinn 12618
E-post:	artjom.jasnetski@ttu.ee

Hariduskäik

2013 – 2018:	Tallinna Tehnikaülikool – doktorikraad
2010 – 2013:	Tallinna Tehnikaülikool – tehnikateaduse magister
2007 – 2010:	Tallinna Tehnikaülikool – tehnikateaduse bakalaureus

Teenistuskäik

2011 – ...:	Testonica Lab OÜ, arendusinsener