On Test Generation for Microprocessors for Extended Class of Functional Faults

Adeboye Stephen Oyeniran, Raimund Ubar, Maksim Jenihhin, Jaan Raik

On Test Generation for Microprocessors for Extended Class of Functional Faults

Adeboye Stephen Oyeniran, Raimund Ubar, Maksim Jenihhin, and Jaan Raik

Centre for Dependable Computing Systems,
Department of Computer Systems,
Tallinn University of Technology.
Akadeemia 15a, 12618 Tallinn, Estonia
{adeboye.oyeniran,raimund.ubar,
maksim.jenihhin,
jaan.raik}@taltech.ee
http://www.ttu.ee/institutes/department-of-computer-systems/

Abstract. We propose a novel strategy of formalized synthesis of Software Based Self-Test (SBST) for testing microprocessors with RISC architecture to cover a large class of high-level functional faults. This is comparable to that used in memory testing which also covers a large class of structural faults such as stuck-at-faults(SAF), conditional SAF, multiple SAF and bridging faults. The approach is fully high-level, the model of the microprocessor is derived from the instruction set and architecture description, and no knowledge about gate-level implementation is needed. To keep the approach scalable, the microprocessor is partitioned into modules under test (MUT), and each MUT is in turn partitioned into data and control parts. For the data parts, pseudo-exhaustive tests are applied, while for the control parts, a novel generic functional control fault model was developed. A novel method for measuring high-level fault coverage for the control parts of MUTs is proposed. The measure can be interpreted as the quality of covering the high-level functional faults, which are difficult to enumerate. We apply High-Level Decision Diagrams for formalization and optimization of high-level test generation for control parts of modules and for trading off different test characteristics, such as test length, test generation time and fault coverage. The test is well-structured and can be easily unrolled online during test execution. Experimental results demonstrate high SAF coverage, achieved for a part of a RISC processor with known implementation, whereas the test was generated without knowledge of implementation details.

Keywords: microprocessor testing, high-level functional fault model, test generation, high-level fault coverage

1 Introduction

The growing density of integration in the semiconductor industry make today's chips more sensitive to faults while the mechanisms of the latter become more

complex. New types of defects need to be considered in test generation to achieve high test quality. Similar to memory testing, broader classes of faults dependent on the neighboring logic, should be used as test targets in case of general logic circuits. To make the tests less independent on particular implementation details, functional fault models and functional test approaches provide a good perspective to make the test development more efficient and to achieve higher test quality.

Software-Based Self-Test (SBST) [1–16] is an emerging paradigm in the test field. The major problem with SBST is usually the not sufficient test quality, measured by the single Stuck-at-Fault (SAF) coverage, let alone considering broader fault classes.

The quality of SBST is mainly affected by test data used in test programs. One of the ways to obtain test data is executing an Automated Test Pattern Generator (ATPG) [17, 18]. In [17] it was shown that the processor can be divided into Modules under Test (MUT) to ease the task of ATPG. The difficulties arise from the need of guiding ATPGs by functional constraints to produce functionally feasible test patterns. The method [18] requires enforcing constraints during ATPG test generation. The run-time for generating test using the complete set of SBST constraints is, however, high. An alternative is to use random test patterns for MUTs [3]. However, these approaches need the knowledge about implementation.

SBST can be structural and functional. Structural approaches [4–6] use information from lower level of design, whereas functional approaches use mainly information of instruction set architecture (ISA). Hybrid SBST was proposed for combining deterministic structural SBST with verification-based test [4, 5, 19, 20]. In addition to Hybrid SBST [7, 21], there are methods that achieve comparable results and improved scalability when generating SBST using only RTL [4–6]. The structural approach cannot be used when structural information about the processor is not available. In [7], for high-level generation of SBST, the implementation details are not required, however, the low-level fault cover is not sufficiently high.

One of the first ISA based methods, using pseudo-random test sequences was proposed in [22]. Another solution, FRITS (Functional Random Instruction Testing at Speed) [23], was based on test generation using random instruction sequences with pseudo-random data. Alternative cache-resident method for production testing [24, 25] using random generation mechanism proves that high cost functional testers can be replaced by low-cost SBST without significant loss in fault coverage. Another approach, based on evolutionary technique was proposed in [26]. Test is being composed of the most effective code snippets with good Stuck-at-fault (SAF) coverage, which were distinguished by constant re-evaluation. The method needs structural information. Later research has been concentrated on developing dedicated test approaches for specific processor parts like pipeline, branch prediction mechanism [11, 21], caches [22, 23].

The drawbacks of the known methods vary in the need of knowledge about implementation details, fault coverage is measured traditionally only with re-

spect to SAF, without considering broader fault classes, and no attempts have been made to evaluate the test coverage regarding multiple faults.

In this paper, to cope with the complexity of gate or RT level representations of microprocessors (MP), we consider the SBST generation with focus on modeling functional faults fully at the behavioral-level using only high-level information. We propose a deterministic high-level test generation method for SBST of processor cores, based on a novel implementation-independent high-level functional fault model. To compare the results with state-of-the-art, the quality of tests is measured by single SAF cover, however, at the same time, we target broader class of faults than single SAF, considering structural logic level faults such as conditional SAF [27, 28], bridging and multiple faults, as well as the functional fault classes used traditionally in memory testing. For formal high-level functional fault modeling and test generation we use the idea of representing the instruction set and architecture of the microprocessor in form of High-Level Decision-Diagrams (HLDD) [29, 30]. The HLDDs can be used as well for trading off different test characteristics such as test length, test generation time and fault coverage.

We generate tests separately for MUTs and in each MUT separately for its control and data parts[31]. The main contribution in the paper is related to test generation for the control parts, whereas for testing the data parts independently of the implementation details, we use the known pseudo-exhaustive test approach [32], not considered here in details.

The rest of the paper is organized as follows. In Section 2, we propose a novel concept of considering the control parts of the modules under test as generic abstract multiplexers, and in Section 3 we elaborate a concept of the novel high-level test generation for the control parts of processor modules. In Section 4, we develop a new general high-level functional fault model for control parts of modules. Section 5 describes a general scheme of test execution flow for the control parts of MUT. In Section 6, we compare the test flow with traditional memory March test, and introduce relationships between the proposed fault model with known functional and structural fault classes. In Section 7, we introduce High-Level Decision Diagrams for formalization of test generation and test optimization. Sections 8 is devoted to demonstrating of experimental results, and Section 9 completes the paper with conclusions.

2   High-Level Representation of Microprocessors

The main concept of the proposed method is based on partitioning the processor under test into functional entities – MUT, representing them as disjoint control and data parts. In this paper, we focus on the executing module and the pipeline forwarding unit as such entities, however showing that the approach is more general, and can be used always, when the MUT can be functionally represented as a set of well-defined functions.

In Fig.1, a part of the pipelined structure of the miniMIPS microprocessor [33] is depicted. In yellow colour, the executing unit is highlighted, whereas

the rest on the figure shows the main components of the pipeline architecture
– pipeline registers, hazard detection circuitry, and the forwarding unit shown
in grey colour. We consider the selected modules as consisting of disjoint control (decoder with MUX-s) and data parts, presented as hypothetical structures
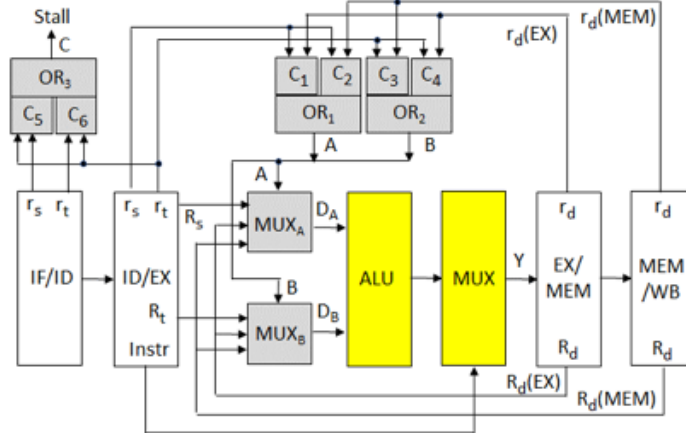without knowing their implementation details.

Fig. 1. A part of a RISC type microprocessor with executing unit in the pipeline and
data forwarding environment

The executing module in Fig.1 (shown in yellow) consists of the data part
concentrated into the ALU/MULT block, whereas the control functions are located in the decoder/multiplexer block MUX. The data part of the pipeline circuitry consists of the pipeline registers separating the different pipeline stages:
instruction fetch (IF), instruction decoding (ID), executing module (EX), memory access (MEM), and write back stage (WB). The control part of the pipeline
forwarding unit (shown in gray) consists of two multiplexer modules, MUX_A
and MUX_B, which are fed by 4 comparators C1-C4 for calculation the values of
control signals of multiplexers. The comparators C5 and C6 are used for hazard
detection in case of "load-use" situations in pipeline circuits [34].

Note, the high-level functionality of the ALU/MULT module (the set of executable functions) is derived from the instruction set of the microprocessor,
whereas the high-level functionality of the forwarding unit is derived from the
description of the architecture of the microprocessor – a set of executable functions, which will be selected by the multiplexers MUX_A and MUX_B. In this
paper, we concentrate on the ALU/MULT module.

We classify two types of high-level functional faults for the modules: control
faults (for control part), and data faults (for data part). We do not consider
data faults explicitly, rather we apply for data manipulation functions bit-wise
pseudo-exhausting tests, which guarantee high fault coverage of a broad class of
faults, whereas knowledge of implementation details is not needed.

For the high-level control faults, we introduce a novel functional fault model, as a general model, which covers a broad set of possible low-level structural faults, and also a set of traditional high-level functional fault models used in memory testing.
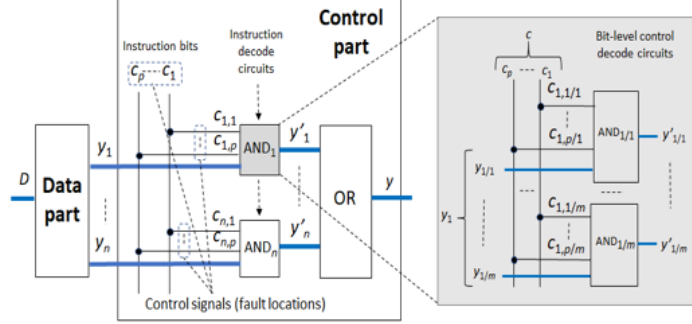

Fig. 2. Generic DNF based control structure of the executing unit

For developing the high-level functional control fault model, we introduce a generic representation of the control part in a form of high-level multiplexer. Consider the executing unit, shown in yellow in Fig.1, and in detailed view in Fig.2, where implementation details are abstract.

Assume, the data part in Fig.2 executes n different functions $y_i = f_i(D_i)$ controlled by a set $F = \{F_i\}$ of instructions (functions), where D is the set of given data operands to be manipulated with $f_i \in F$. The length of data word is m, and the number of control signals p must satisfy the constraint

$$\log_2 n \leq p \leq n \qquad (1)$$

and hence, depends on how the instructions are coded. However, the number of $1 - bit$ control signals p and the mapping of control vector signals to n instructions, where $n = |F|$, is considered as unknown. Let us keep for a while the coding of the control signals and the value of p in the model of MUTopen.

The control part consists of the multiplexer MUX, p control lines controlling the MUX, and an unknown circuit for mapping the instruction operation codes into the functional signals p. The high-level n AND blocks in MUX have each p control and a single m-bit data input, whereas the OR block in MUX has n data inputs from the outputs of AND blocks. Each AND block consists of m AND gates with p control inputs, and a single 1-bit data input. Hence, the described control module, represented in a form of a high-level multiplexer, consists of m different 1-bit logic level AND-OR multiplexers, used for decoding the instructions, and for extracting the results of executed instructions.

As it can be seen from Fig.2, the border between the control part and the data part is determined by the AND gates, where the 1-bit control signals and 1-bit data signals are joining. The number of the AND gates on this border is

equal to

$$n \times m \times p \tag{2}$$

By introducing the described hypothetical MUX-based executing module, we have functionally separated the data and control parts by the border of $n \times m \times p$ gates, and transformed the function of the control block from "active" controlling of the manipulations in the data part to "passive" selection of the results of data manipulations in the data part. In other words, we have neglected all possible optimizations, which may have been carried out during the design of the execution unit.

Let us introduce now the following abstraction, in accordance to Fig.2, as a set of m equivalent disjunctive normal forms (EDNF) representing an implementation independent design of the MUT at the expense of possible over-dimensioning the real logic design. The disjoint presentation of the control and data parts allows to create an implementation-independent high-level functional control fault model.

The justification of the proposed abstraction results from the fact that a test T_{EDNF} developed for detecting all non-redundant faults in the EDNF, will detect also all faults in the real optimized circuit of the executing unit [35]. On the other hand, if the implementation details of the real circuit equivalent to EDNF were known, then the test T_{RC} of the real circuit, in general case, may have shorter length than T_{EDNF}.

The second abstraction will concern the control signal decoding, which, in general, is highly depending on the details of implementation. To allow the test generation for the control part be implementation independent, and as simple as possible, we introduce the one-to-one coding between the control signals and instruction, so that to each functions $f_i \in F$ the control signal c_i will correspond, and $C = \{c_i\}$, where $|F| = |C|$, will represent the full set of control signals. In this case, each control signal $c_i \in C$ selects the related function $f_i \in F$. In other words, by this way, we have introduced a hypothetical and simple coding scheme of int ructions, where $p = n$, which represents, according to (1) the higher bound of the value p. On the other hand, it provides the minimal length $n \times m$ for the border between the control and data parts of the MUT, determined by (2).

This second abstraction allows to overcome the problem of illegal instruction codes and to make easier the identification of redundant faults in any of the further real implementations of the control part.

The justification of the proposed abstractions will be given in the next section, where we introduce the new control fault model.

3   Basic Concepts of Generating High-Level Tests

Consider a simplified MUT in Fig.3a, derived using the two described above abstractions, and consisting of the data and control parts. Fig.3a presents a $k - th$ bit slice of the m-bit control module, where m is the width of the data word carrying the value of f_i. The data manipulation block (e.g. ALU) executes

n different functions f_i, selected by the control codes c denoted by symbolic integers $c_i, i = 0, 1, \ldots n$.

Each bit-slice of the control part consists of MUX with n control lines (control inputs to each AND). The $1 - bit$ data lines from the data manipulation module (ALU) are the data inputs to each AND. The OR gate has n data inputs.

Consider testing of the MUX in Fig.3. Let us concentrate on testing the control code for the $k-th$ bit slice, which selects from ALU the result of the high-level defined operation $f_{i,k}$, producing the expected output value $y_k = f_{i,k}, (D)$. Consider the high-level symbolic (pseudo) control signals $c_{i,k}$, which may be applied ($c_{i,k} = 1$) or not applied ($c_{i,k} = 0$) as Boolean variables. From $c_{i,k} = 1$, it follows $c_{j,k} = 0$ for all $j \neq i$ due to the mutual exclusion of each other.
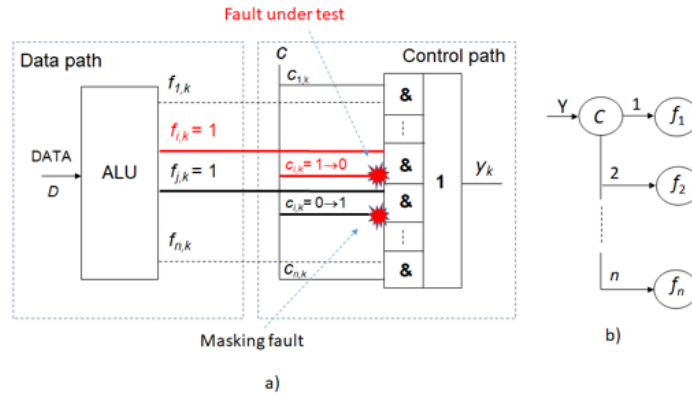

Fig. 3. A module consisting of data and control parts and its HLDD

The graph in Fig.3b represents a High-Level Decision Diagram (HLDD) which describes the mapping "if $C = i$ then $y = f_i$", meaning that if $c_i = 1$ then $y = f_i$. About the role of using the HLDD model for representing the MUT we discuss in Sections VII.

Definition 1. *Introduce a test T_i^* as a structure $T_i^* = (I_i, D_i)$ where I_i is an instruction, which performs the function $f_i \in F$ in the MUT represented by a set of functions F, and Di is a set of data operands used by the instruction Ii. The data Di may consist of one or more patterns $d \in D_i$. If D_i will consist of t patterns, the instruction Ii will be repeated in the test T_i^* for each pattern $d \in D_i$.*

Definition 2. *Let us introduce a notation $T_{i,k}^* = \{c_{i,k}, f_{i,k}\}$, where $c_{i,k} \in \{0,1\}$ and $f_{i,k} \in \{0,1\}$, for a single bit test, in accordance to Fig.3.*

Consider a test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 1\}$, which targets the detection of stuck-at-0 faults $c_{i,k} \equiv 0$ and $f_{i,k} \equiv 0$, in other words, the test is proving that the function $f_i \in F$ is controllable by $c_{i,k} = 1$, in the $k - th$ bit. However, this test is targeting the detection of single faults only, and the test may fail

in proving the controllability of $f_{i,k}$, if there exists any multiple fault of type $\{c_{i,k} \equiv 0, c_{j,k} \equiv 1\}$, $j \neq i$, because of mutual masking of these two faults. This masking situation is illustrated in Fig.3a.

Lemma 1. *There will be no masking of the fault $c_{i,k} \equiv 0$ by any other fault $c_{j,k} \equiv 1$, $j \neq i$, if the test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 1\}$ will be applied under the constraints $f_{j,k} = 0$, for all $j \neq i$.*

Proof. The expected value of y_k for the test $T_{i,k}^*$ in case of no faults will be $y_k = 1$. In case of a single fault $c_{i,k} \equiv 0$, the value of $f_{i,k} = 1$ will not propagate to the output y_k, due to $c_{i,k} \equiv 0$ and all $c_{j,k} = 0$, $j \neq i$, causing in such a way response $y_k = 0$, which means that the fault $c_{i,k} \equiv 0$ is detected. The response $y_k = 1$ would be the proof, that the function $f_{i,k}$ is controllable. However, this proof will be valid only for the case of assuming the single fault $c_{i,k} \equiv 0$. In case of any double fault $\{c_{i,k} \equiv 0, c_{j,k} \equiv 1\}$, $j \neq i$, instead of $f_{i,k} = 1$, which is blocked by $c_{i,k} \equiv 0$, the value of another function $f_{j,k} = 1$ is propagated by $c_{j,k} \equiv 1$ to the output y_k. Hence, as the response to the test, we will get the same value $y_k = 1$ as expected, which means that the fault $c_{i,k} \equiv 0$ is masked by $c_{j,k} \equiv 1$, and we still will not know if the function $f_{i,k}$ is controllable by the signal $c_{i,k}$ or not.

Let us introduce a notation for multiple faults of type $\{c_{i,k} \equiv 0, c_{j,k} \equiv 1\}$, where $j \neq i$, and $\{c_{j,k} \equiv 1\}$ represents any subset of faults $c_{j,k} \equiv 1$ for different combinations of $j \neq i$.

Lemma 2. *To detect any multiple fault of type $\{c_{i,k} \equiv 0, \{c_{j,k} \equiv 1\}\}$ in the MUT represented by a set of functions F, a test T_i^* must be generated, so that the constraint $f_{j,k} = 0$ were satisfied for each $f_j \in F$, $j \neq i$, by at least one pattern $d \in D_i$ in T_i^*.*

Proof. Since from $c_{i,k} = 1$ the value $c_{j,k} = 0$ follows for all $j \neq i$ due to the mutual exclusion of control signals, we have satisfied automatically the conditions of sensitizing the faults $c_{j,k} \equiv 1$ on the lines $j \in i$. On the other hand, the constraint $f_{j,k} = 1$ for all $j \neq i$ will serve as the condition of propagating the faults $c_{j,k} \equiv 1$, $j \neq i$, to the output y_k, to make all control faults $c_{j,k} \equiv 1$ detectable. .

There may be two border cases in generating the test T_i^* . First, a single data operand $d \in D_i$ may be generated, which allows detection of all possible multiple faults of type $\{c_{i,k} \equiv 0, \{c_{j,k} \equiv 1\}\}$ in the same time by the same data $d \in D_i$. In this case, the pattern d has to satisfy the constraints $f_{j,k} = 0$ simultaneously for all $j \neq i$, which may be a seldom case. Second, as a general case, a set of data D_i must be generated, so that each constraint $f_{j,k} = 0, j \neq i$, was satisfied at least by one pattern d in the set of data D_i.

From Lemma 1 the following corollary directly results.

Corollary 1. *The test T_i^* generated in accordance with conditions of Theorem 1 for all bits k, detects the control faults $c_{i,k} \equiv 0$, and the data faults $f_{i,k} \equiv 0$,*

which both belong to the fault class of SAF. The functional high-level meaning of the test T_i^* is that it proves that the function f_i, is controllable by the control signals $c_{i,k}$ in all bits k without masking due to possible additional faults $c_{j,k} \equiv 1$ on other control lines $j \neq i$.

The added value of test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 0\}$ that, a lot of data part faults, causing the change of the value of $f_{i,k}$, $1 \rightarrow 0$, will also be detected.

From Lemma 2 the following corollary directly results.

Corollary 2. *A test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 0\}$, which targets the detection of the data fault $f_{i,k} \equiv 1$, will detect simultaneously also all control faults $c_{j,k} = 1$, $j \neq i$, if the constraints $f_{j,k} = 1$ for all $j \neq i$, will be satisfied at least by one pattern $d \in D_i$ in T_i^*.*

The added value of the test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 0\}$, which has the goal of detecting the SAF $c_{j,k} \equiv 1$ on other control lines $j \neq i$, detects also a lot of data part SAF, which cause the change of the value of $f_{i,k}$, $0 \rightarrow 1$.

Table 1. SAF faults detection of 1-bit control signals

Test				Detected faults	Proof
$c_{i,k}$	$f_{i,k}$	$c_{j,k}$	$f_{j,k}$		
1	1	0	0	$ci, k \equiv 0$ (with no fault masking)	Corollary 1
	0		1	$c_{j,k} \equiv 1$ (of any multiplicity)	Corollary 2

From the previous discussion, it follows, that it would be very easy to test the control part of MUT if it would be implemented according to the proposed abstract model, represented by EDNF and using direct mapping $c_i \rightarrow f_i$. By the proposed two methods, the both types of SAF faults can be detected: $c_{i,k} \equiv 0$, using Corollary 1, and $c_{i,k} \equiv 1$, using Corollary 2. This result is illustrated also in Table 1.

In the following, in Sections IV – VI, we show that Corollary 2 can be extended for a very broad class of faults and used as the basis for developing an implementation-independent test generation method for the control part of MUT.

4   A New High-Level Functional Control Fault Model

From Lemmas 1-2 and Corollaries 1-2, a strategy of testing follows. When applying the test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 1\}$, it is recommended to generate data operands for applying the values $f_{j,k} = 0$ for as many $j \neq i$ as possible, to avoid mutual masking of $c_{i,k} \equiv 0$ by multiple faults $c_{j,k} \equiv 1$. On the other hand, when applying the test $T_{i,k}^* = \{c_{i,k} = 1, f_{i,k} = 0\}$, it is recommended to apply the values $f_{j,k} = 1$ for all $j \neq i$ to increase the efficiency of testing the faults $c_{j,k} \equiv 1$.

In functional testing, if two arguments or functions in the MUT model will due to physical defects interfere, then the resultant value of the interference can be calculated either by AND or OR function, depending on the technology.

Assume for the further discussion that we have OR-technology.

Definition 3. *Introduce a high-level functional control fault $f_{i,k} \rightarrow (f_{i,k}, f_{j,k})$, which means that instead of the function f_i, in the $k-th$ bit of the data word, both functions $f_{i,k}$ and $f_{j,k}$ will be selected and executed simultaneously. In case of the OR –technology, the result of activation of the function f_i in the presence of the fault $f_{i,k} \rightarrow (f_{i,k}, f_{j,k})$ in the $k-th$ bit will be $y_k = f_{i,k} \vee f_{j,k}$.*

Lemma 3. *To detect the fault $f_{i,k} \rightarrow (f_{i,k}, f_{j,k})$ in a MUT, represented as mapping $(c_i \in C) \rightarrow (f_i \in F)$, a test pattern $T_i^*(c_i = 1, d)$ must be applied with constraint $f_{i,k}(d) < f_{j,k}(d)$, where $d \in Di$.*

Proof. The proof results directly from Definition 3, because only if $f_{i,k}(d) = 0$, and $f_{j,k}(d) = 1$, the expected result $f_{i,k}(d) = 0$ and the faulty result $f_{i,k}(d) \vee f_{j,k}(d) = 1$ will be distinguishable.

Definition 4. *Introduce modifications of the high-level functional fault introduced in Definition 3, such as*

1. *$f_{i,k} \rightarrow f_{j,k}$, where instead of a function $f_{i,k}$ another function $f_{j,k}$, $j \neq i$, will be selected and executed, and*
2. *$f_{i,k} \rightarrow \{f_{j,k}\}$, where instead of a function $f_{i,k}$, a group of functions $f_{j,k}$ will be selected and executed.*

From Lemma 3 and Definitions 3 and 4, the following corollaries result:

Corollary 3. *To detect the fault $f_{i,k} \rightarrow (f_{j,k})$, a test pattern $T_i^*(c_i = 1, d)$ must be applied with constraint $f_{i,k}(d) < f_{j,k}(d)$, where $d \in D_i$.*

Corollary 4. *To detect the fault $f_{i,k} \rightarrow \{f_{j,k}\}$, a set of test patterns $T_i^*(c_i = 1, d)$, $d \in D_i$ must be applied, so that for each $f_{j,k} \rightarrow \{f_{j,k}\}$, a data pattern $d \in D_i$ exists, where $f_{i,k}(d) < f_{j,k}(d)$.*

Definition 5. *Let us call the set of all high-level functional control faults $CF = \{f_{i,k} \rightarrow (f_{i,k}, f_{j,k})\}$, for all pairs of $f_{i,k}, f_{j,k} \in F$, as functional control fault model of the MUT, represented as mapping $(c_i \in C) \rightarrow (f_i \in F)$. The size of the fault model is $|CF| = (n-1)^2 \times m$. Let us call the subset $CF(f_{i,k}) \subset CF$ as functional control fault model of the function $f_i \in F$.*

Theorem 1. *To detect all functional faults introduced in Definitions 3 and 4, for each function $f_i \in F$, a set of data operands must be generated, which satisfy the constraints:*

$$\forall k \in (1, m) \exists d \in D_i \left(f_{i/k}(d) < f_{j/k}(d) \right) \tag{3}$$

The proof of theorem results from Lemma 3 and Corollaries 4 and 5.

Definition 6. *Introduce a high-level control fault table* $R = \left\| r_{i,j/k} \right\|$ *as a 3-dimensional array for a given set of data patterns* D, *where the entries* $r_{i,j/k}$ *represent k-bit vectors and* $r_{i,j/k} = 1$, *if there exists a pattern* $d \in D_i$, *which satisfies the constraint* $f_{i,k}(d) < f_{j,k}(d)$, *otherwise* $r_{i,j/k} = 0$. *The size of the fault model* R *is equal to the size* $|CF| = (n-1)^2 \times m$.

The high-level control fault coverage is measured by the ratio of 1-s in the array to the size of R.

Calculation of the high-level fault coverage can be carried out by high-level fault simulation with the goal of checking if the constraints (3) are satisfied or not.

5 Test Structure and Test Execution

Denote by D_i the test data generated for detection the fault model $CF(f_i)$. Based on the data D_i, and according to Lemma 1, the following test structure results, as shown in Algorithm.1.

Algorithm 1: Test Execution Structure

---
1 **for** *all* $f_i \in F$ **do**
2   **for** *all* $d \in D_i$ **do**
3   apply test for $f_i(d)$

According to Algorithm 1, all tests for exercising the functions $f_i \in F$, are executed one by one, each of the tests in a loop using one by one the data $d \in D_i$ which satisfy the constraints (3). A test is a subroutine which initializes the data $d \in D_i$, executes an instruction (or a sequence of instructions) responsible for realizing the function $f_i(d)$, and performs the observation of the test result $y = f_i(d)$.

From the algorithm in Algorithm 1, the following structure of test execution can be derived: the full test T consists of a sequence of test modules $T_i, i = 1, 2, \ldots n$, where each i-th module consists of test patterns $T_{i,t}$, where each pattern $T_{i,t} \in T_{i,t}$ satisfies, a subset of constraints (3).

For each test pattern $T_{i,t} \in T_i$, including the data operand $d \in D_i$, and for each data bit k of the functions $f_i \in F$, the set $F = \{f_i\}$ can be partitioned for each k into two parts F_k^0 and F_k^1, so that

$$F_k^0 = \{f_{i,k} | f_{i,k}(d) = 0\}, and F_k^1 = \{f_{i,k} | f_{i,k}(d) = 1\}$$

.

Such a test pattern covers all the constraints $f_{i,k}(d) < f_{j,k}(d)$, according to (3), where $f_{i,k} \in F_k^0$ and $f_{j,k} \in F_k^1$.

Such a test execution according to Algorithm 1 is depicted in Fig.4. The unrolled test sequence consists of n test modules T_i, each of them consisting of a sequence of test patterns $T_{i,t}$ for testing a function $f_i \in F$. The behaviour of the MUT is highlighted for the $k-th$ bit of the test pattern $T_{i,t}$, showing the
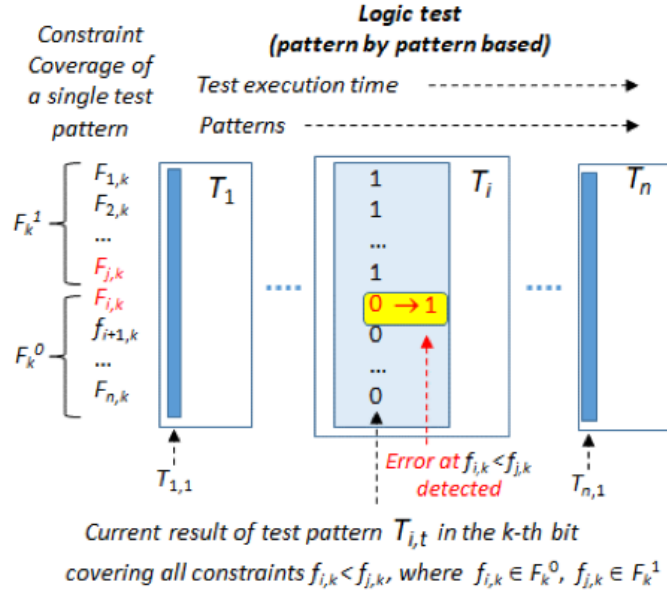
Fig. 4. Unrolled test execution evolving in time

subset of constraints satisfied by the pattern. As an example of error detection is shown, where the expected value of $f_{i,k}$ is changed from 0 to 1 due to a fault in the control part of MUT.

6   Extension of the Fault Class Beyond SAF

The ideas of the proposed fault model in Section IV, and the test concept in Section V are adopted from the known methods of memory testing, particularly from March test [36]. The motivation was driven by the purpose to extend the fault class, to be covered by test, to that of used in case of memories.

Let us consider an example of the March test depicted in Fig.5, and compare it with the test flow developed for a logic MUT shown in Fig.4. The analogy between the memory test and logic test is in similar handling of addressing the cells in the memory and controlling the functions of $f_i \in F$ in logic MUTs. In case of memories, testing of cells (data part) and the addressing logic (control part) can be easily joined in the same test, whereas in the proposed approach, testing of data part and control part proceeds separately.

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

In the proposed method, the constraints cannot be stored, rather they have to be produced "on-line" at each test pattern. In Fig.4, a test pattern $T_{i,t} \in T_i$

including test data $d \in Di$, is illustrated, showing the values it produces on-line for the $k - th$ bit of all functions $f_i \in F$, simultaneously. All functions for the bit k are partitioned by the data $d \in D_i$ into two groups F_k^0 and F_k^1, as explained in Section V. We see, that this particular test pattern with data d covers only a subset of constraints for $f_{i,k}(d) < f_{j,k}(d)$, where $f_{i,k} \in F_k^0$ and $f_{j,k} \in F_k^1$.
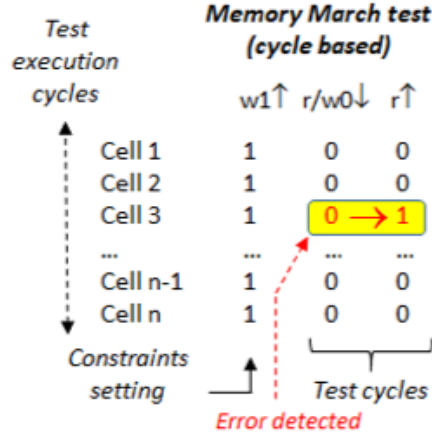

Fig. 5. Illustration of the March test for memories

In case of memory, in each step of the test cycle $(r/w0 \downarrow)$, when reading the Cell i, all constraints [Cell i] < [Cell j] are covered by a single run through all the cells. Here, [Cell i] means the value stored in the Cell i. In case of the proposed method of testing a logic MUT, the test for $f_i \in F$, has to be repeated with other data d till all the constraints (3) have been satisfied for all pairs of functions $\{f_{i,k}, f_{j,k}\}$.

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

Consider a MUT, represented by a set of mappings:

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

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

By test data generation, used in the March test for memories and in the proposed test method for logic MUTs, the coverage of the following functional fault classes by the proposed method results [36]:

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

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

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

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

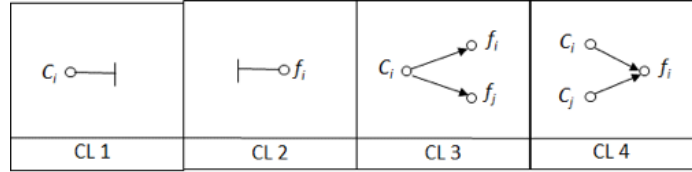The fault classes CL-1 – CL-4 are illustrated in Fig.7 [36]:

Fig. 6. Functional control fault classes CL 1 – CL 4

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

As shown in [37], address decoders built out of CMOS gates can exhibit CMOS stuck-open faults [CL-7]. The effect of such faults is that the combinational instruction decoder will behave as a sequential circuit for certain control signals. The consequence of such a fault is that another instruction will be decoded and executed. However, this fault can be also collapsed, because it will be covered by the faults of CL-4.

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

Regarding other general fault classes, such as conditional SAF (CL-9) [28], called also as functional faults [38], pattern faults [39], fault tuples [40] or cell-internal defects (CL-10) [41], will manifest themselves as a change of instruction code $c_i \rightarrow c_j$, and are covered by the fault class CL 4.

We have shown, that the structural fault classes CL-5 – CL-10 are collapsed by the implementation-independent high-level functional fault classes CL-1 – CL-4, which are used in memory testing and are covered by the March test [36]. On the other hand, in Section V, we have shown, that the test for microprocessor MUT, which satisfies the constraints (3), and is executed in accordance with the test flow in Algorithm.1, will cover the same fault classes CL-1 – CL-4 used in memory testing. Finally, from Theorem 1, it follows, that the fault classes CL-1 – CL-4 can be represented by a single fault class $CF = \{f_{i,k} \rightarrow (f_{i,k}, f_{j,k})\}$, as stated in Theorem 1.

The relationships between iterative fault collapsing are shown in Fig.7: first, collapsing of structural faults (CL-5 – Cl-9) by functional faults used in memory
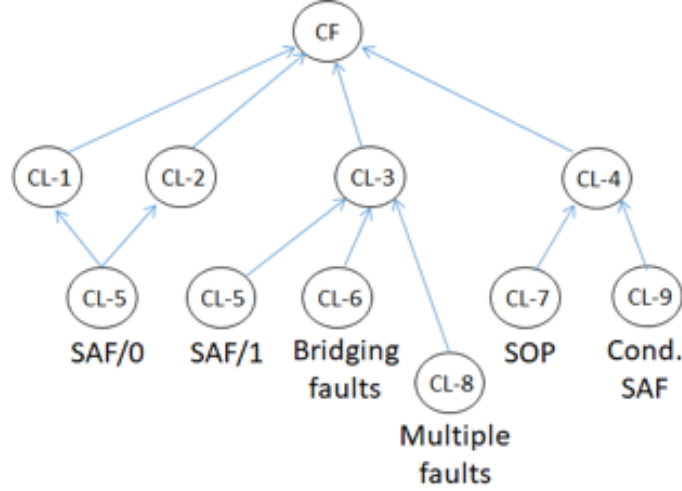
Fig. 7. Fault collapsing relationships

testing (CL1 – CL-4) [36], and thereafter, collapsing of the faults (CL1 – CL4) by the general high-level control fault CF, developed in the paper.

In this paper, we do not consider testing of the faults in data part, however, we propose to use here testing of all instructions separately by using pseudo-exhaustive test (PET) data operands [32]. It is well-known that PET provides also a good fault coverage for a broad fault class.

7   High-Level Decision Diagrams and Functional Test

The problem with the proposed functional fault model is a low scalability, because when the size n of the set of functions F is growing, the number of high-level faults $|CF| = (n-1)^2 \times m$ is growing very fast. This is actually the same problem as with memory testing: the broader class of faults is desired the longer test is needed.

From that a question follows, which is how to cope with the complexity explosion by looking for tradeoffs between some test characteristics like fault coverage, test length, test generation time etc. One possibilities is to partition the sets of F into smaller subsets, and consider high-level test generation for subsets of F separately.

High-level Decision Diagrams (HLDDs) [29, 30] can be used as a uniform approach for extracting and solving the constraints (3) in test generation for the modules of microprocessors.

Consider in Fig.8, a HLDD for a subset of 20 instruction of the MiniMIPS microprocessor [33] which represents a subset of function of the ALU. The single non-terminal decision node of the HLDD is labelled by the control variable c (denoted by the operation code of the instructions) having n control values
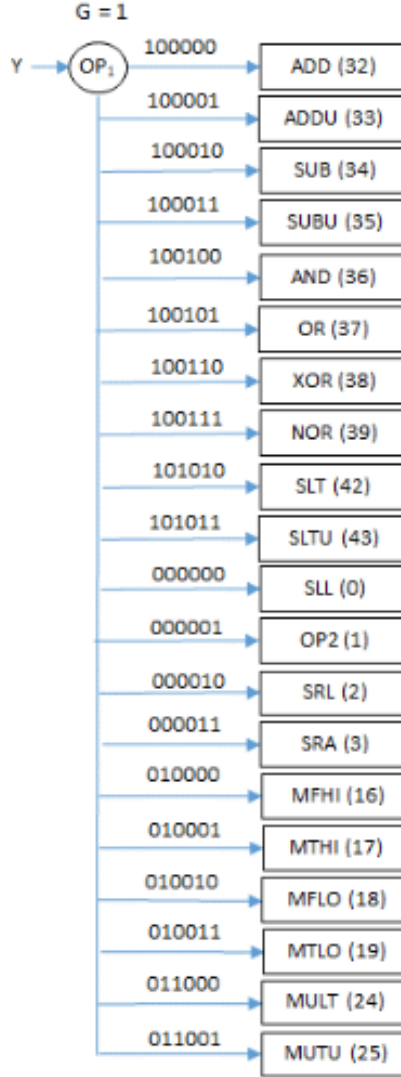
Fig. 8. HLDD with a single decision node for representing 20 MiniMIPS instructions

labelling the output edges of the node c. The terminal nodes are labelled by data manipulation functions fi to be used for creating the data constraints (3).

The HLDD in Fig.8 can be regarded as a MUX of the control part of the MUT, whereas terminal nodes describe the functions of the data part. Denote the HLDD as $G = 1$, meaning that the graph has 1 decision node. The size of the fault model for this subset of functions, $n = 20$ is $|CF| = (n - 1)^2 \times m = 11552$, assuming the data word length is $m = 32$.
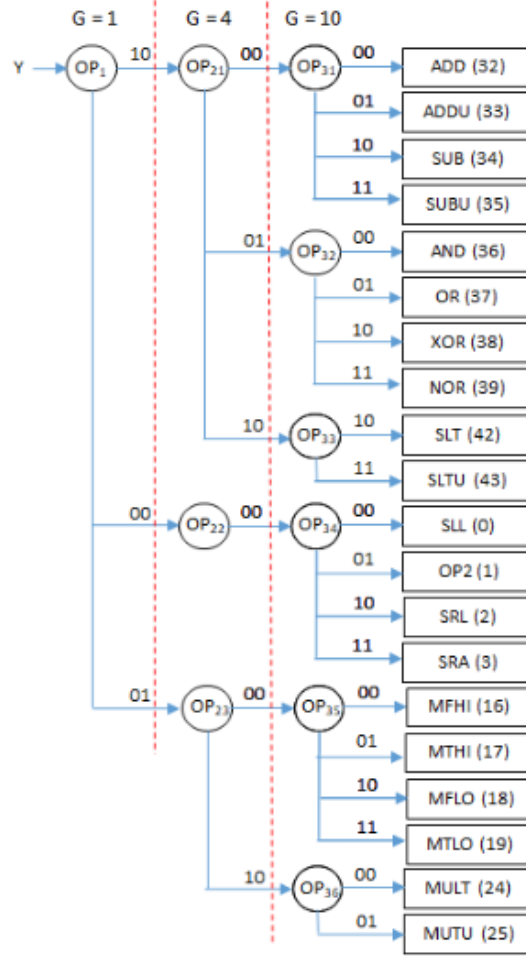
Fig. 9. HLDD for a subset of instructions of MiniMIPS

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

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

Consider now another version of the HLDD in Fig.9, which represents the same subset of 20 instructions of the MiniMIPS, but has now 10 decision nodes.

Each decision node represents a partitioning, and the number of edges of the decision node corresponds to the size of the related subset of functions F.

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

1. $G = 1$ as a HLDD with a single internal node and 20 terminal nodes with $N = 12160$ functional faults (Fig.8) ,
2. $G = 4$ as 4 subgraphs with decision nodes OP1, OP21, OP22 and OP23, and with 3, 10, 4 and 6 terminal nodes, respectively, resulting in $N = (6 + 100 + 12 + 30) \times 32 = 3904$ functional faults,
3. $G = 10$ i.e. the current HLDD in Fig.7 with 8 decision nodes, resulting in $N = (6 + (6 + 12 + 2) + (12 + 12 + 2) + (12 + 2)) \times 32 = 2112$ functional faults (Fig.9).

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

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

Table 2. Example of scalabilities for three versions of HLDDs

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

8   Experimental Results

We have carried out test generation experiment with two goals, first, to investigate the possible tradeoffs between the complexity of the high-level fault model, and the characteristics of generated tests such as test length, SAF coverage, and test generation time, and second, to compare the SAF coverage of gate-level, achieved by the proposed method with state-of-the-art methods. However, it should be mentioned, that in the latter case, the proposed method has additional advantage regarding the coverage of the extended functional fault class, that has not been taken into account in the state-of-the-art methods.

We carried out experiments on Intel Core i7 processor at 3.4GHz and 8GB of RAM. The target was to investigate the efficiency of the new high-level implementation independent SBST generation method for microprocessors by measuring both the high-level functional fault coverage, and the gate-level fault coverage (FC). As research objectives of the experiments, the executing and forwarding units of MiniMIPS [33] were chosen.

For investigating the possibilities of tradeoffs between the complexity of the high-level functional control fault model and the characteristics of generated tests the executing unit was used. It consists of adder and 2 multiplication modules MULT0 and MULT1. We targeted 28 instructions out of MiniMIPS 51, as the basis for the set of functions $F = \{f_i\}$ to be tested. The results are depicted in Table 4 and in Table 5. In Table 4 we show the SAF simulation results for only the control test, whereas in Table 5, we show the results of SAF simulation of the integrated control and data part tests. The latter experiment was needed to demonstrate the additional impact of the data part test to the control test, for the special case of SAF coverage.

Table 3. Test generation for different fault model complexities (only control part test is SAF simulated)

HLDD Nodes	Test Patterns	Functional Faults N	SAF FC%		Time (s)
			ALU	EX unit	
1	161	23328	99.07	98.06	131.0
3	146	12160	99.04	98.03	82.8
6	103	6400	98.79	97.78	45.8
12	83	4032	98.67	97.65	32.9

Table 4. Test generation for different fault model complexities (both control and data part tests are SAF simulated)

HLDD Nodes	Test Patterns	Functional Faults N	SAF FC%		Time (s)
			ALU	EX unit	
1	161	23328	99.32	98.33	131.0
3	146	12160	99.30	98.31	82.8
6	103	6400	99.11	98.13	45.8
12	83	4032	99.01	98.03	32.9

In Tables 3 and 4 we see, that the SAF coverage is very little depending on the size of the set of high-level functional faults used. We see also increase of the SAF coverage, if we simulate the full test including both control and data part tests. This is natural, because the control test is not targeting at all the data part. On the other hand, we see that the control test indirectly covers a huge amount of SAF faults in the data part (as added value of using the proposed high-level functional control fault model).

The reasons of not covering of all SAF in the gate-level simulated circuits may be twofold: (1) the faults are not detectable, or (2) the set of functions, used

as the target for test generation, may not cover the full circuit, which was selected for SAF simulation (the circuit may be responsible for other functions, not included into the set of functions, which was used for high-level test generation).

The second part of experiments consisted in comparing the results with state-of-the art methods, using for comparison only the gate-level SAF coverage, which however was not the target of the proposed method, which had the target to generate an implementation-independent test.

In Table 5, we compare our high-level approach with a commercial ATPG, where we showed that the latter had to use huge time when struggling with test generation for a sequential part of the circuit (8h), whereas the high-level approach for solving the combinational data constraints used less than a minute.

In Table 6, the fault coverage fault coverage and simulation times are given for the forwarding unit (FU), first, when applying only the ALU test, and then the dedicated test for only FU, and thereafter, combining both tests. The tests for FU were generated without knowing gate-level implementation detail, we relied only on general information of the MiniMIPS pipeline architecture, which includes the number of stages and forwarding paths.

In Table 7, we compare our results for 3 different MiniMIPS modules with 3 other test generators. Our approach is similar to [7] in the sense that the gate-level implementation details are not required, but it shows almost 5% improvement in FC compared to [7]. Although the method in [19] shows 1% improvement over the proposed method, it is based on requiring of structural information. Method in [18] requires enforcing set of constraints during ATPG test generation, requiring also gate-level information. Differently from state-of-the-art methods, where single SAF cover is the target, the proposed method targets extended class of faults including conditional and multiple SAF.

Table 5. Execute Unit Test

Method	Experiments		#Faults	FC(%)	Stored Patterns	Executed Patterns	ATPG Time
Proposed high-level method	High-level ATPG		756	100	166	4818	47s
	Gate-level Simulation	Adder	2516	99.92			
		MULT0	95188	99.52			
		MULT1	91810	99.16			
Commercial gate-level ATPG		Adder	2516	99.96	957	957	8h 27min
		MULT0	95188	97.40			
		MULT1	91810	97.71			

Table 6. Fault coverage of forwarding unit by different tests

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

Table 7. Comparison with other methods

Module/ unit	#faults	Gate-level implementation details are exploited		Gate-level implementation independent	
		ATIG[19]	SBST[18]	SBST[7]	Proposed
ALU	203576	98.67%	n.a	97.85%	99.06%
PPS_EX	211136	97.62%	96.20%	84.12%	98.37%
Forwarding Module	3738	99.00%	99.68%	93.64%	98.03%

9   Conclusions

In this paper, we propose a novel implementation independent SBST generation method for the modules of RISC type microprocessors, which produces high gate-level single fault coverage, comparable with the methods which use the knowledge of implementation details. However the main target of the paper is to propose a method which covers an extended class of structural faults including high-level functional faults used in memory testing.

The main idea of the method is to generate tests separately for modules under test (MUT) and in each MUT separately for its control and data parts. The main contribution in the paper is related to test generation for the control parts, whereas for testing the data parts independently of the implementation details, we use the well-known pseudo-exhaustive test approach, not considered here in detail.

A generic high-level functional fault model was developed, represented as a set of constraints to be satisfied by data operands, for the control parts of MUT. The fault model covers a broad set of low-level structural faults, and differently from state-of-the-art, a set of traditional functional fault models used in memory testing. We showed the possibility of focusing a large number of structural and functional fault classes into a single measurable high-level functional fault model.

Based on this representative fault model, a novel measure of high-level control fault coverage is proposed, and a method of evaluating the test quality using this measure, which can indirectly assess the capability of the test to cover a large class of faults beyond SAF.

The data constraints based fault model, and the introduced analogy of testing with March test flow for memories revealed the possibility of applying the proposed approach, not only for the combinational MUTs, but also for sequential ones. We introduced High-Level Decision Diagrams, as a means to be used for formalization of high-level test generation and optimization of test programs by trading off different test characteristics, such as the fault model complexity versus test length, test generation time and well measurable SAF coverage.

For comparison of our results with state-of-the-art we used the measure of SAF coverage. Experimental results demonstrate higher SAF coverage compared to other existing implementation-independent test generation methods for microprocessors. The added value of the proposed approach, compared with state-of-the-art, is the proof of covering extended fault class beyond SAF.

References

1. Gizopoulos, D., Paschalis, A., Zorian, Y.: Embedded Proecessor-Based Self-Test. Kluwer Acad. Publisher, (2004)
2. Gizopoulos, D: Advances in Electronic Testing. Springer, (2014)
3. Chen, L., Dey, S.: Software-based self-testing methodology for processor cores. IEEE.Trans. on CAD of IC and Systems. 20, 3, 369–380 (2001)
4. Kranitis, N., Gizopoulos, D., Xenoulis, G.: Software-based self-testing of embedded processors. IEEE.Trans. on Comp. 54, 4, 369–380 (2005)
5. Gurumurthy, S., Vasudevan, S., Abraham, J.A.: Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. ITC, (2006)
6. Chen, C., Wei, T., Gao, Lu and H.: Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 15, 5, 505–517 (2007)
7. Gizopoulos, D.et al.: Systematic Software-Based Self-Test for Pipelined Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 16, 11, 1441–1453 (2008)
8. Psarakis, M., Gizopoulos, D., Sanchez, E., Sonza Reorda M.: Microprocessor Software-Based Self-Testing. IEEE Design & Test of Computers. 27, 3, 4–19 (2010)
9. Di Carlo, S., Prinetto, P., Savino, A.: Software-Based Self-Test of Set-Associative Cache Memories. IEEE Transactions on Computers. 60, 7, 1030–1044 (2011)
10. Schölzel, M., Koal, T., Roder, S., Vierhaus, H.T.: Towards an automatic generation of diagnostic in-field SBST for processor components. In: 14th Latin American Test Workshop - LATW, Cordoba. 1–6 (2013)
11. Changdao, D.et al.: On the functional test of the BTB logic in pipelined and superscalar processors. In: 14th Latin American Test Workshop - LATW, Cordoba. 1–6 (2013)
12. Bernardi, P.et al.: On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors. In: 14th International Workshop on Microprocessor Test and Verification, Austin, TX. 52–57 (2013)
13. Bernardi, P.et al.: On the in-field functional testing of decode units in pipelined RISC processors. In: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam. 299–304 (2014)
14. Sanchez, E., Reorda, M.S.: On the Functional Test of Branch Prediction Units. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 23, 9, 1675–1688 (2015)
15. Bernardi, P., Cantoro, R., De Luca, S., Sánchez, E., Sansonetti, A.: Development Flow for On-Line Core Self-Test of Automotive Microcontrollers. IEEE Transactions on Computers. 65, 3, 744–754 (2016)
16. Riefert, A., Cantoro, R., Sauer, M., Sonza Reorda, M., Becker, B.: A Flexible Framework for the Automatic Generation of SBST Programs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 24, 10, 3055–3066 (2016)
17. Chen, L., Ravi, S., Raghunathan, A., Dey, S.: A scalable software-based self-test methodology for programmable processors. In: 40th annual Design Automation Conference (DAC). 548–553 (2003)

18. Riefert, A., Cantoro, R., Sauer, M., Sonza Reorda, M., Becker, B.: On the automatic generation of SBST test programs for in-field test. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). 1186–1191 (2015)

19. Zhang, Y., Li, H., Li, X.: Automatic Test Program Generation Using Executing-Trace-Based Constraint Extraction for Embedded Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 21, 7, 1220–1233 (2013)

20. Kranitis, N., Merentitis, A., Theodorou, G., Paschalis, A., Gizopoulos, D.: Hybrid-SBST Methodology for Efficient Testing of Processor Cores. IEEE Design & Test of Computers. 25, 1, 64–75 (2008)

21. Lu, T., Chen, C., Lee, K.: Effective Hybrid Test Program Development for Software-Based Self-Testing of Pipeline Processor Cores. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 19, 3, 516–520 (2011)

22. Shen, J., Abraham, J.A.: Native mode functional test generation for processors with applications to self test and design validation. In: International Test Conference 1998 (IEEE Cat. No.98CH36270)s. 990–999 (1998)

23. Parvathala, P., Maneparambil, K., Lindsay, W.: FRITS - a microprocessor functional BIST method. In: International Test Conference, Baltimore, MD, USA. 590–598 (2002)

24. Bayraktaroglu, I., Hunt, J., Watkins, D.: Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues. In: IEEE International Test Conference, Santa Clara, CA. 1–7 (2006)

25. Acle, J.P., Cantoro, R., Sanchez, E., Reorda,M.S.: On the functional test of the cache coherency logic inmulti-core systems. In: IEEE 6th Latin American Symposium on Circuits & Systems (LASCAS), Montevideo. 1–4 (2015)

26. Corno, F., Sanchez, E., Reorda, M.S., Squillero, G.: Automatic test program generation: a case study. IEEE Design & Test of Computers. 21, 2, 102–109 (2004)

27. Ubar, R.: Fault Diagnosis in Combinational Circuits with Boolean Differential Equations. Plenum Publishing Corporation, USA. 1693–1703 (1980)

28. Holst, S., Wunderlich, H.: Adaptive Debug and Diagnosis without Fault Dictionaries. 12th IEEE European Test Symposium (ETS'07), Freiburg. 7–12 (2007)

29. Karputkin, A., Ubar, R., Raik, J., Tombak, M.: Canonical representations of high-level decision diagrams. Estonian Journal of Engineering. 16, 1, 39–55 (2010)

30. Ubar, R., Tsertov, A., Jasnetski, A., Brik, M.: Software-based self-test generation for microprocessors with high-level decision diagrams. In: 5th Latin American Test Workshop - LATW, Fortaleza. 1–6 (2014)

31. Oyeniran, A. S., Ubar, R., Jenihhin, M., Raik. J.: Implementation-Independent Functional Test Generation for RISC Microprocessors, 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), Cuzco, Peru, 2019, pp. 82-87

32. Oyeniran, A.S., Azad, P.Z., Ubar, R.: Parallel Pseudo-Exhaustive Testing of Array Multipliers with Data-Controlled Segmentation. In: IEEE International Symposium on Circuits and Systems (ISCAS), Florence. 1–5 (2018)

33. MiniMIPS Instruction Set Architecture.: Opencore https://opencores.org/

34. Patterson, D., Hennessy, J.: Computer Organization and Design. Elsevier (2015)

35. Armstrong, D.B.: On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets. IEEE Transactions on Electronic Computers. EC-15,1 66–73 (1966)

36. van de Goor, A.J.: Semiconductor Memories: Theory and Practice. Wiley. pp.512 (1991)

37. Miczo, A.: Digital Logic Testing and Simulation. Wiley. pp.668 (2003)

38. Ubar, R.: Fault Diagnosis in Combinational Circuits by Solving Boolean Differential Equations. Automatics & Telemechanics, Moscow. 11 170–183 (1979)(In Russian)
39. Keller, K.B.: Hierarchical Pattern Faults for Describing Logic Circuit Failure Mechanisms. US Patent 5546408. Aug, 13, (1994)
40. Dwarakanath, K.N., Blanton, R.D.: Universal Fault Simulation using fault tuples. In: Proceedings 37th Design Automation Conference, Los Angeles, CA, USA. 786–789 (2000)
41. Happke, F. et al.: Cell-Aware Tests. IEEE Trans. on CAD of IC and systems. 33, 9 (2014)