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, $2 \leq |\Gamma(m)| \leq |V(z(m))|$, 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

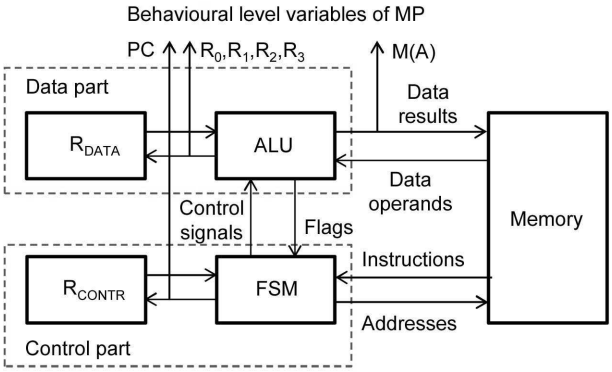

Fig. 1. Behavioural level structure of the microprocessor.

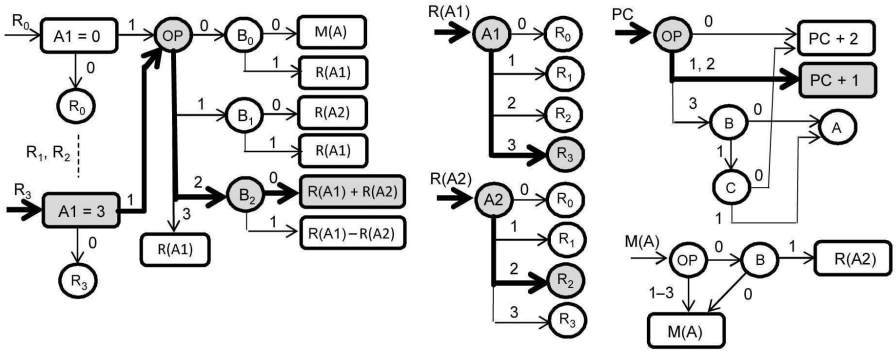

Fig. 2. HLDD model of the microprocessor.

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

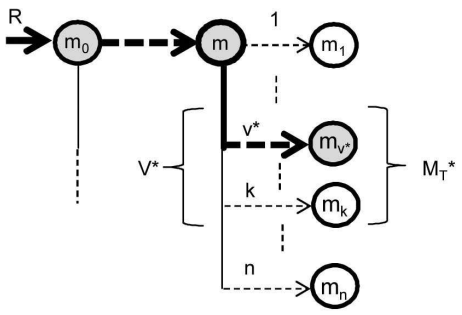

Fig. 3. Illustration of the conditions for testing the node m in the HLDD model G_R.

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

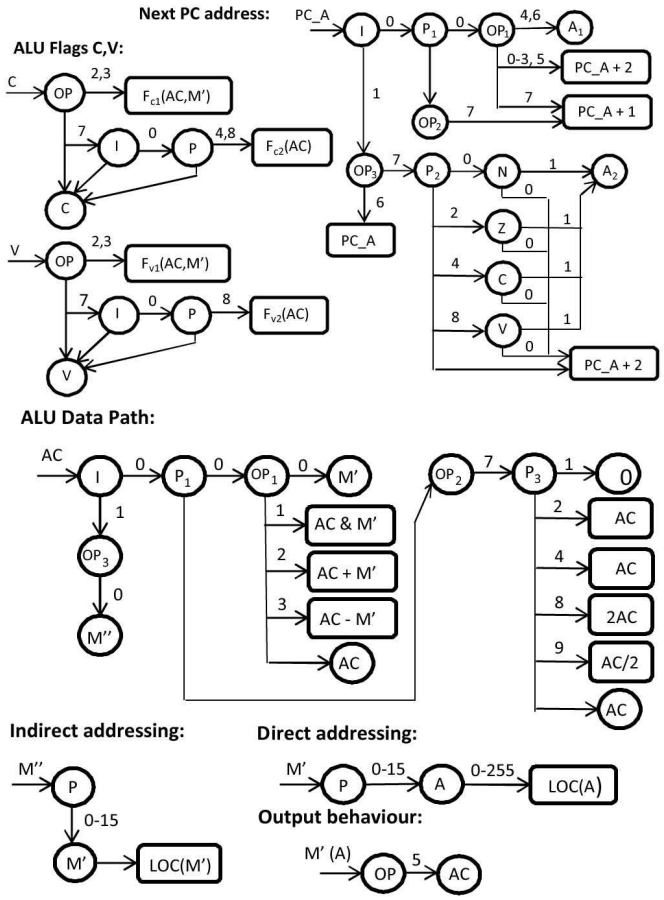

Fig. 4. HLDD model for microprocessor Parwan.

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.

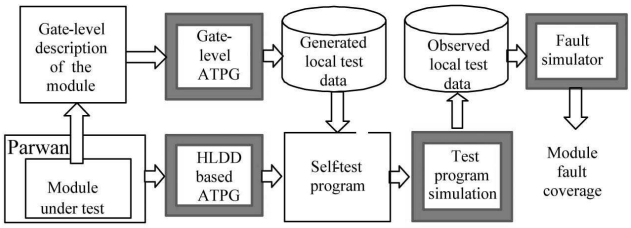

Fig. 6. Set-up of experiments.

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.