Postgraduate Study Report DC-PSR-2004-14

Mixed-Mode BIST Based on Column Matching

Petr Fišer

Supervisor: *Ing. Hana Kubátová, CSc.*

September 2004

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
CZ-121 35 Prague 2
Czech Republic

email: fiserp@fel.cvut.cz
WWW: cs.felk.cvut.cz/~fiserp

This report was prepared as a part of the project

Design of Highly Reliable Control Systems Built on dynamically Reconfigurable FPGAs

This research has been supported by grant GA102/01/0566, GA102/03/0672, GA102/04/2137 and MSM 212300014.

. . . . . .
Ing. Petr Fišer
postgraduate student

. . . . . .
Ing. Hana Kubátová, CSc.
supervisor

Contents

MIXED-MODE BIST BASED ON COLUMN MATCHING

Petr Fišer
fiserp@fel.cvut.cz
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Abstract

A test-per-clock BIST method for combinational or full-scan circuits is proposed. The method is based on a design of a combinational block - the decoder, transforming pseudo-random LFSR code words into deterministic test patterns pre-computed by some ATPG tool. We propose a column-matching algorithm to design the decoder. Here the maximum of output variables of the decoder is tried to be matched with the decoder inputs, yielding the outputs be implemented as mere wires, thus without any logic. No memory elements are needed to store the test patterns, which reduces an area overhead.

We describe the Column-Matching algorithm into detail and propose several heuristic methods solving some of the major NP-hard problems. The tradeoff between the duration of the execution of BIST, the solution quality and runtime is discussed. The time complexity of the algorithm is studied and experimentally evaluated.

Since quite a large number of test vectors if often needed to sufficiently test a particular circuit, synthesizing all these vectors deterministically would mean a large area overhead. Thus, the Column-Matching method was modified to support a mixed-mode testing. The BIST is divided into two disjoint phases – the pseudo-random phase, where the LFSR patterns are being applied to the circuit unmodified, and the deterministic phase detecting all the yet undetected faults. This enables us to reach a high fault coverage in a short test time and with a low area overhead.

The choice of the lengths of the two phases directly influences both the test time and area overhead. This issue is discussed here as well.

The complexity of the resulting BIST is evaluated for the ISCAS benchmarks.

Keywords

built-in self-test, test-per-clock, pseudo-random testing, deterministic BIST

Chapter 1

Introduction

The complexity of present VLSI circuits rapidly grows. Their testing is becoming more and more important, together with the tests complexity and total costs. Using only external test equipment (ATE) is becoming impossible, mainly due to a huge amount of test vectors, long testing time and very expensive test equipment. Incorporating the Built-in Self-Test (BIST) becomes inevitable. It requires no external tester to test the circuit, since all the circuitry needed to conduct a test is included in the very circuit. This is paid by an area overhead, long test time and often a low fault coverage. Up to now, many BIST methods were developed [Aga93, Tou96a, Tou96b], all of them trying to find some trade-off between these four aspects that are mutually antipodal:

- Fault coverage
- Test time
- Area overhead
- BIST design time

A high fault coverage means either a long test time (exhaustive test), or a high area overhead (ROM-based BIST). A pseudo-random testing established the simplest trade-off between all the three criteria. With an extremely low area overhead the circuit can be tested usually up to more than 90% in a relatively small number of clock cycles (thousands). To improve fault coverage and to reduce the test time, many enhancements of this pseudo-random principle were developed. Of course, all of them are accompanied by some additional area overhead. Here the BIST design time comes to importance – a design of a BIST structure achieving high fault coverage with a low area overhead often takes a long time to synthesize.

We propose a novel BIST method based on our Column-Matching principle. We introduce an Output decoder transforming the pseudo-random patterns into deterministic patterns pre-computed by an ATPG (Automatic Test Pattern Generator) tool. Using it the desired fault coverage is obtained (100%), for a cost of the Decoder logic. To reduce it, we try to implement as many Decoder outputs as possible as wires, without any logic. This is being done by application of the Column-matching algorithm.

Moreover, we extend the method to support a mixed-mode testing. Here the test is divided into two phases: the pseudo-random one and the deterministic one. This enables us to significantly reduce the Decoder logic.

The study is structured as follows: the theoretical background and basic BIST principles are described in Section 2, the related state-of-the-art work is given in Section 3. Major principles of our newly proposed method are presented in Section 4, the experimental results are presented in Section 5, Section 6 contains the conclusions.

Chapter 2

Theoretical Background - BIST

The general Built-in Self-Test structure consists of three main parts [McC85] – see Figure 2.1. The TPG (*Test Pattern Generator*) produces *test patterns* that are fed to the inputs of a *Circuit under Test* (CUT) and the responses of a circuit are then evaluated in a *Response Evaluator* (RE).

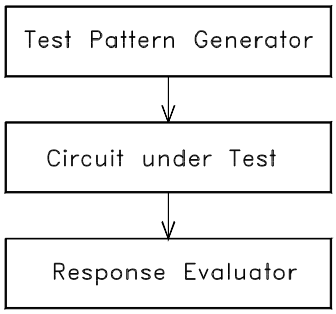

Figure 2.1: BIST structure

During the test the test patterns are sequentially fed to the primary inputs of a logic circuit and the response at the primary outputs is checked. If the response is different from the expected value, a fault is detected.

There are two basic testing strategies: the *functional testing* and the *structural testing*. The functional testing checks the circuit's response to the input patterns to test the functionality of the circuit, while its inner structure needs not be known. On the other hand, the structural test tries to find physical defects of the circuit by propagating faults to the output (by finding a sensitive path). There may exist several kinds of physical faults, namely the *stuck-at* faults (stuck-at-one, stuck-at-zero), bridging faults, opens and other technology dependent faults. Most of the faults are easy to detect, as they can be propagated to the circuit's outputs by many possible vectors (of their total number 2^n, where n is the number of the primary inputs of a circuit). However, there are faults that are hard to detect (*random resistant faults*), as only few test patterns propagate these faults to the outputs. Thus, the amount of faults that can be detected by a particular test set depends on the test patterns. Thus we always have to specify the set of faults on which we concentrate. If a test set detects all faults from the given fault set, it is denoted as *complete*. The most commonly accepted fault set consists of all stuck-at faults.

Since the TPG can be constructed to have both parallel and/or serial outputs the BIST can be designed in two general ways: test-per-clock and test-per-scan. In the *test-per-clock* BIST the CUT is being fed by parallel outputs of the TPG, and thus each test pattern is processed in one clock cycle. The response of the CUT goes to the response evaluator in parallel, which is often a MISR (*Multi-Input Shift Register*). A general structure of the test-per-clock BIST is shown in Fig. 2.2.

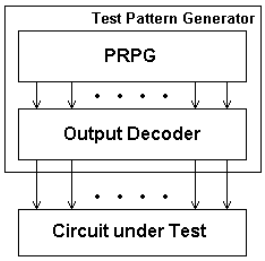

Figure 2.2: Test-per-clock BIST structure

A second typical structure, suitable especially for testing sequential circuits, is denoted as a *test-per-scan* BIST. It is used in connection with CUTs having a scan chain, i.e., the circuit's flip-flops are connected into a chain making one scan register for testing purposes. Here the test patterns are shifted into the scan register of the CUT and applied by activating the functional clock after every full scan-in of one test pattern. The response is then scanned out and typically evaluated by a serial signature analyzer (signature register).

In this work we deal with the test-per-clock only, however the method can be adapted to test-per-scan as well.

Chapter 3

Related Work

Before describing the principles of the state-of-the art methods, namely the Reseeding, Weighted pattern testing, Bit-fixing, Bit-flipping and Row-marching methods, we introduce the naive BIST methods, mainly for better understanding to the latter ones.

3.1 Exhaustive Testing

There are several testing approaches differing in their successfulness and area overhead. The most naive method – the *exhaustive testing* – feeds the circuit with all the 2^n patterns and checks the responses. Obviously, for a combinational circuit the exhaustive test provides complete fault coverage, and can be very easily implemented (an area overhead is often the lowest possible), but it is extremely time demanding and thus very inefficient. It is applicable to circuits with up to 30 inputs (10^9 patterns, which takes 1 sec on the frequency of 1 GHz), for more inputs the exhaustive testing is not feasible. The test patterns are mostly generated by an LFSR (Linear Feedback Shift Register), since it produces 2^n-1 different patterns during its period and it can be very easily implemented on the chip.

A slight modification of this method called a *pseudo-exhaustive testing* [McC84] allows us to test a circuit exhaustively without a need to use all the 2^n test patterns. The circuit is divided into several possibly overlapping *cones*, which are logic elements that influence individual outputs of the circuit. Then, all the cones are separately tested exhaustively, and hereby also the whole circuit is completely tested. The only fault type not covered by pseudo-exhaustive tests are bridging faults between elements belonging to different non-overlapping cones. If such an efficient decomposition is possible, the circuit can be tested with much less than 2^n test patterns. However, for more complex circuits the cones are rather wide (the cones have a large number of inputs) and thus the pseudo-exhaustive testing is often not feasible either.

3.2 Pseudo-random Testing

In a simple *pseudo-random testing* the test patterns are generated by some pseudo-random pattern generator (PRPG) and lead directly to the circuit's inputs. It differs from the exhaustive testing with a test length. If the PRPG structure and seed are properly chosen, only several test patterns (less than 2^n) are necessary to generate to completely test the circuit. The

pseudo-random testing is also widely used in a case when the complete fault coverage is not required, since the pseudo-random patterns often successfully detect most of the easy-to-detect faults.

In more sophisticated pseudo-random testing methods the pseudo-random code words generated by a PRPG are being transformed by some additional logic (combinational or sequential) in order to reach better fault coverage. Here the main area overhead consists in the combinational logic. To such methods belong the reseeding-based techniques, weighted testing, bit-fixing, bit-flipping, and others. These methods are often being referenced as a *mixed-mode BIST*.

3.3 Reseeding

In this technique the LFSR is seeded with more than one computed seeds during the test, the seeds need to be stored in ROM [Koe91]. The seeds are often smaller than the test patterns themselves and, most importantly, more than one test patterns are derived from one seed. This significantly reduces memory requirements.

One problem is that if a standard LFSR is used as a pattern generator, it may always not be possible to find the seed producing the required test patterns. A solution of this problem is to use a multi-polynomial LFSR (MP-LFSR), where the feedback network of a LFSR is reconfigurable [Hel92, Hel95]. Here both the seeds and polynomials are stored in a ROM memory and for each LFSR seed also a unique LFSR polynomial is selected. The structure of such a TPG is shown in Fig. 3.1.

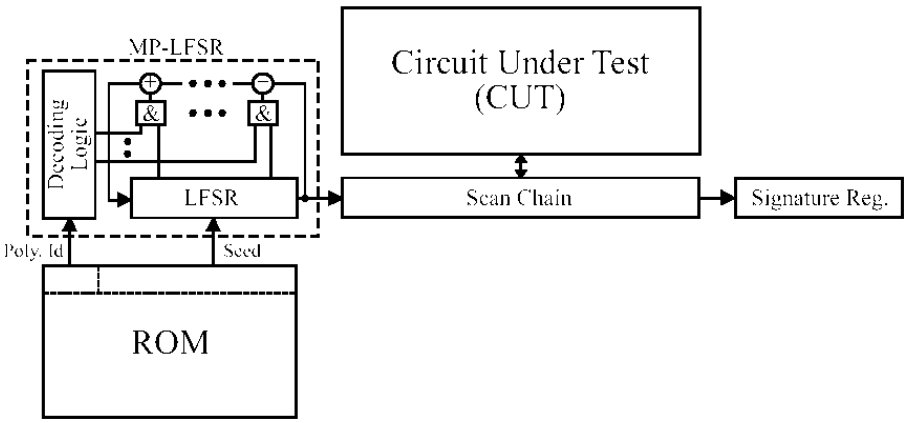

Figure 3.1: Multi-polynomial BIST

This idea has been extended in [Hel00] where the *folding counter*, which is a programmable Johnson counter, is used as a PRPG. Here the number of folding seeds to be stored in ROM is even more minimized.

In spite of all these techniques reducing memory overhead, implementation of a ROM on a chip is still very area demanding and thus the ROM memory should be completely eliminated in BIST.

3.4 Weighted Pattern BIST

One of such approaches is the *weighted pattern testing.* Here the PRPG patterns are being biased by a *signal probability* of each of the PRPG outputs (the probability of a 1 value) in order to reach required test patterns. In the weighted pattern testing method two problems have to be solved: first, the weight sets have to be computed and then how to generate the weighted signals. Many weight set computation methods were proposed [Bar87] and it was shown that multiple weight sets are necessary to produce patterns with a sufficient fault coverage [Wun88]. These multiple weight sets have to be stored on chip and also the logic providing switching between them is complicated, thus this method often implies a large area overhead.

Several techniques reducing the area overhead of a weighted pattern testing were proposed – one of them is a *Generator of Unequiprobable Random Tests* (GURT) presented in [Wun87]. The area overhead is reduced to minimum, however it is restricted to only one weight set. Also the more general method based on modifying the GURT [Har93] uses only one weight set and thus it is also limited to special cases of the tested circuits and cannot be used in general.

Special methods using multiple weight sets that can be easily implemented were proposed in [Pom93] and [AlS94]. In [Pom93] three different weight values can be applied by adding a very simple combinational logic to the PRPG outputs, [AlS94] on the other hand uses specially designed PRPG flip-flops.

As the LFSR code words have very balanced properties, the design of the logic generating a weighted signal can be rather difficult. Some approaches using cellular automata instead of an LFSR were studied, and good results were reached using this approach for some circuits [Alo03, Nov98, Nov99]. Methods using inhomogeneous cellular automata to produce weighted pattern sets are presented in [Nee93].

3.5 Bit-Fixing and Bit-Flipping

Principles of the bit-fixing [Tou95, Tou96a, Tou01] and bit-flipping [Wun96] methods consist in a modification of some bits by some additional logic, in order to increase the fault coverage. Both of them introduce a *mapping function* that transforms the LFSR pseudo-random code words into deterministic patterns – see Fig. 3.2.

This idea was generalized in [Tou96b], where the problem of finding a mapping function is transformed into finding a minimum rectangle in a binate matrix. Procedures used in ESPRESSO [Bra84] were used to find a mapping logic.

General schemes of test-per-scan bit-flipping and bit-fixing BIST methods are shown in Figures 3.3 and 3.4 respectively. The bit-fixing method modifies the pseudo-random sequence by AND and OR gates, the bit-flipping method augments the sequence by flipping some bits by a XOR gate.

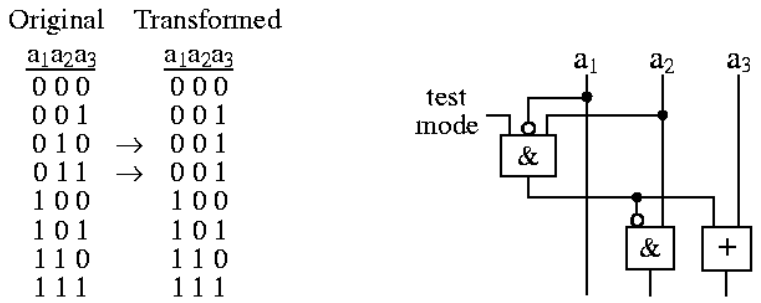

Figure 3.2: Modifying the LFSR patterns

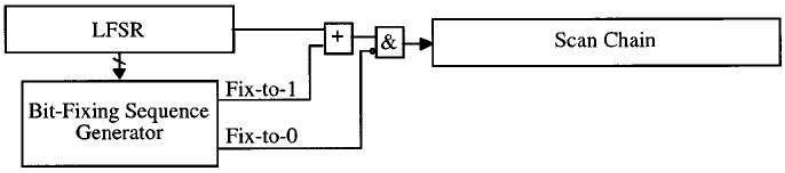

Figure 3.3: Bit-fixing scheme

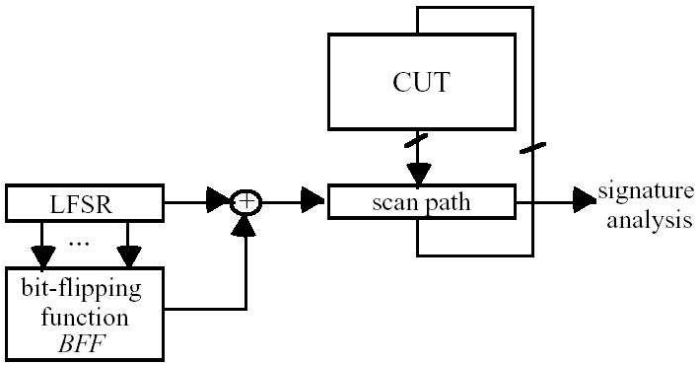

Figure 3.4: Bit-flipping scheme

3.6 Row Matching

The *row matching* approach proposed in [Cha95, Cha03] is based on a very similar idea. A simple combinational function that transforms some of the PRPG patterns into test patterns is being designed in order to reach better fault coverage. Here, the test patterns are independent on the PRPG code words in a sense of a similarity of the patterns – the proper test vectors are pre-computed by an ATPG tool; they are not derived from the original PRPG code words as it was being done in the previous methods.

The row matching means finding an assignment of these test patterns to the code words, as it is shown in Fig. 3.5. Each of the test patterns has to be assigned to some PRPG pattern to generate the required test. Here the problem to be solved consists in finding such a row matching that the pattern transformation function is as simple as possible. Similar idea is also exploited in our BIST methods presented in this report.

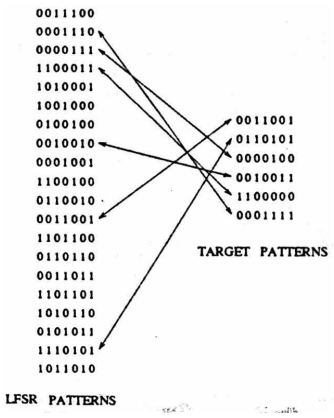

Figure 3.5: Row matching principle

The *cost function* of the row matching is used as a criterion for finding a row match. The cost function is an estimation of the complexity of the combinational function performing the pattern transformation. The cost of a matching M for an n-input CUT (and thus the combinational block has n outputs) is defined as follows:

$$C(M) = \sum_{i=0}^{n} \left(|I_i| \times W(|I_i|) \right) \tag{3.1}$$

where I_i is called an *input index* of the output variable i and it is defined as a set of input variables of an output decoder that are needed to obtain the values of the i-th output – i.e., the support of the i-th output variable. The weight W is used to take into account a non-linear relation between the size of the I_i and the area overhead.

The aim is to find a row matching that minimizes this function. This is, however, an NP-hard problem and thus some heuristic must be used. In the proposed algorithm [Cha95] the rows are being matched sequentially (one-by-one) preferring the match that locally minimizes the cost function. After the matching is done, the result is in a form of a truth table, which has to be minimized by some Boolean minimizer (ESPRESSO) to obtain the final solution. The truth table corresponding to the example from Fig. 3.5 is shown in the following Figure:

Input Vector	Output values required at						
	0	1	2	3	4	5	6
0 0 1 1 0 0 1	0	0	1	1	0	0	1
1 1 1 0 1 0 1	0	1	1	0	1	0	1
0 0 0 0 1 1 1	0	0	0	0	1	0	0
0 0 1 0 0 1 0	0	0	1	0	0	1	1
1 1 0 0 0 1 1	1	1	0	0	0	0	0
0 0 0 1 1 1 0	0	0	0	1	1	1	1

Figure 3.6: The final truth table

In addition to introducing a mapping function, a special kind of a PRPG is exploited here – a GLFSR (generalized LFSR). In principle, it behaves similarly to a weighted-pattern TPG, however the weighted patterns are being generated by a modification of a LFSR. However, this modification introduces an additional logic to the whole BIST structure, and thus it disturbs otherwise good results.

Chapter 4

Overview of New Approach

4.1 Column-matching Mixed-Mode BIST method

We propose a novel test-per-clock BIST method. The test patterns are applied to the primary inputs of the circuit-under-test (CUT) in parallel, thus in each clock cycle one test vector is being processed. The response is then drawn from the primary outputs and analyzed in the response evaluator (RE), which is mostly a multi-input shift register (MISR).

This method aims at the decrease of the area overhead that may be achieved by the simplification of the test pattern generator (TPG). We have used deterministic test patterns generated by some ATPG (Automatic Test Pattern Generator) tool, thus the fault coverage achieved strictly depends on these patterns. No memory is used for their storage, since the memory mostly causes a big area overhead on a chip. From a global point of view [Str02], our method is based on a synthesis of a finite state machine (FSM) that produces algorithmic test patterns.

The test pattern generator consists of two blocks: the pseudo-random pattern generator (PRPG) and the output decoder, which is a combinational block transforming the PRPG patterns into deterministic tests. The PRPG is mostly constructed as a linear feedback shift register (LFSR) with an appropriate generating polynomial, or as a cellular automaton [Nee93, Nov98, Nov99, Alo03,]. The basic structure of such a test-per-clock BIST is shown in Fig. 4.1.

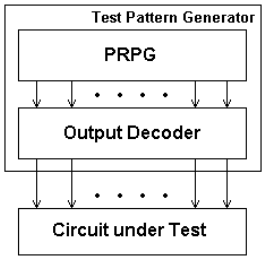

Figure 4.1: Test-per-clock BIST structure

Synthesis of the combinational logic transforming the pseudo-random patterns into deterministic tests is based on our column-matching algorithm [Fis02, Fis03a]. We try to implement most of the outputs of the decoder logic by assigning them to the inputs, thus implement them without any circuitry. An enhancement of this method enabled us to support a mixed-mode BIST, which significantly reduces the output decoder logic [Fis04a]. The issue of adjusting the BIST synthesis parameters, namely the influence of the ratio of the test don't cares and the durations of the pseudo-random and deterministic phases are discussed here as well, and in [Fis04b, Fis04c].

The method was extensively tested on standard ISCAS benchmarks. Here a big scalability of the method, in terms of the trade-off between the test time and area overhead, was observed.

4.2 Principles of Mixed-Mode BIST

Most of the mixed-mode BIST techniques involve using some kind of transformation and switching logic accompanying the pseudo-random pattern generator (PRPG). A general structure of our mixed-mode BIST design is shown in Fig. 4.2. The pseudo-random code words are produced by an LSFR. Then they are transformed by the Decoder into deterministic vectors. The Switching logic selects the patterns to be applied to the CUT. After that the circuit's response is evaluated, usually in the multi-input shift register (MISR).

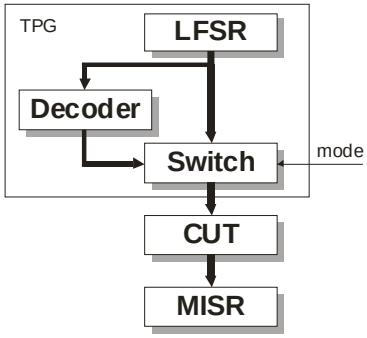

Fig. 4.2: Mixed-mode BIST structure

The main difference between our algorithm and the competitive methods [Tou95, Tou96b, Cha03] consists in a separation of the pseudo-random and deterministic phases. In the other methods the LFSR patterns that do not detect any faults are identified and modified. Here the switching logic consists of coupled AND and OR gates in the bit-fixing method [Tou95] – see Fig. 3.3, or a XOR gate for bit-flipping [Wun96] – See Fig. 3.4.

In praxis, several initial pseudo-random vectors detect faults, but the fault detection capability of the latter ones quickly drops to zero. Thus, it could be more advantageous to run the unmodified pseudo-random phase for several clock cycles and then switch to the deterministic one at once, as it is being done in our approach. The switching logic then consists of *multiplexers, in the most general case*. The area overhead caused by the switching logic needs not be too big, since we try to eliminate even these multiplexers using a modified column-matching method. Moreover, the size of a multiplexer, when implemented using transmission gates, is 1.5-times the size of a standard NAND gate [DeM94].

In the first, pseudo-random phase, all the multiplexers are set so they feed the circuit with the unmodified LFSR patterns; the Decoder is cut off. Subsequently, in the deterministic phase, all the MUXes switch to the Decoder outputs and only the modified patterns are

applied to the CUT. The *mode* signal driving the multiplexers can be generated externally (by ATE), or some kind of a counter can be used. Even in this case the area overhead of this counter can be negligible, since the BIST-controller counter can be exploited, or we can use an extra counter that can be shared by many IP cores in a complex design.

4.3 The BIST Design Process

The Decoder logic is synthesized using our column-matching algorithm. The Decoder is a combinational block transforming some of the PRPG patterns into deterministic patterns pre-computed by an ATPG. Our aim is to design the decoder to be as small as possible. Its design is based on "matching" maximum of the decoder outputs with its inputs. Particularly, when the test vectors are reordered and assigned to the LFSR vectors in such a way that the values in the respective matched columns (i.e., input and output variables) are equal, the matched output will be implemented as a wire, without any logic. Since the BIST is designed for combinational circuits, any reordering can be freely done. Moreover, the deterministic test can be much longer than the computed test sequence. Only few of the PRPG patterns produce the required test vectors and the rest represent the non-testing "gaps". This gives us a big freedom how to select the appropriate matches. The values of the non-matched outputs have to be synthesized by some Boolean minimizer, i.e. BOOM [Hla01, Fis03b].

The whole BIST design process can be divided into four phases:

1. Simulate several (*PR)* pseudo-random patterns for the CUT and determine the undetected faults (by a fault simulator).
2. Compute deterministic test patterns for these faults by an ATPG tool.
3. For the following pseudo-random LFSR patterns (*Det*) and the deterministic tests do the column matching (see Section 6).
4. Synthesize the unmatched decoder outputs by BOOM.

An artificial illustrative example is shown in Fig. 4.3. The 5-bit LFSR is run for 5 cycles first and the easily testable faults are detected. Then we run the fault simulation to find the undetected faults, for which the test vectors are generated by an ATPG. At the end the decoder logic is synthesized for these tests and the succeeding LFSR patterns. The resulting circuitry is shown in Fig. 4.4. Here we can see that for some outputs (y_0, y_1) there is no decoder and switching logic needed, for some there is the switching logic only (y_2, y_3). Such cases should be preferred when the BIST is being designed.

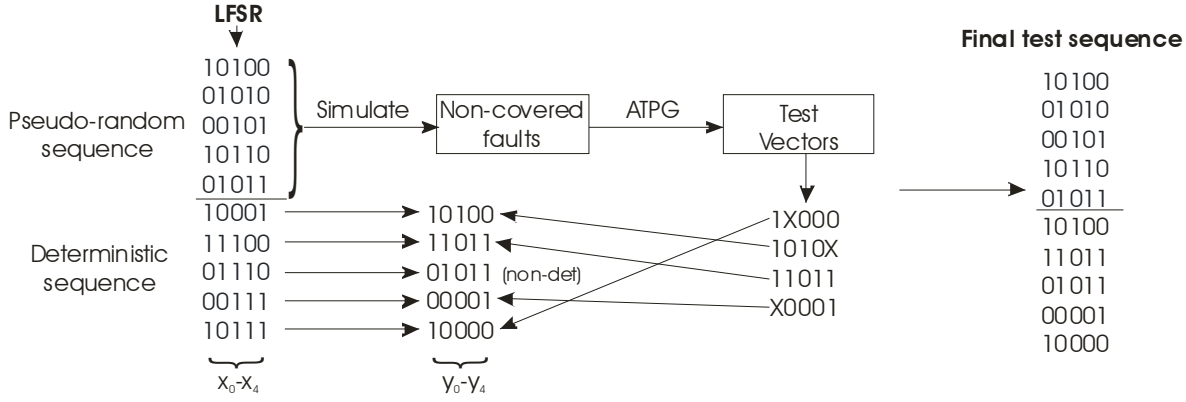

Figure 4.3: Test sequence generation

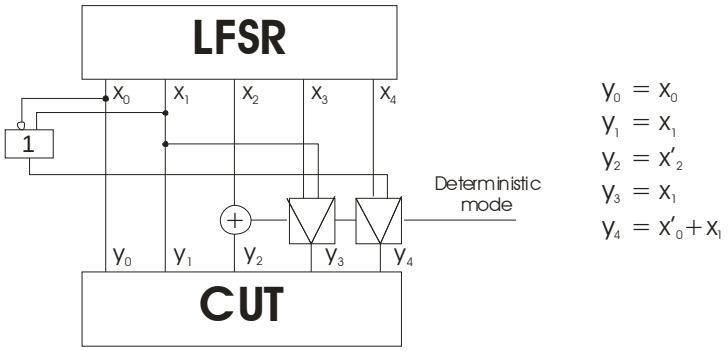

Figure 4.4: Resulting BIST circuitry

4.4 The Pseudo-Random Phase

The aim of the pseudo-random phase is to cover as many faults as possible, while keeping the test time acceptable. Two aspects play role here: the LFSR polynomial and seed and the test length. Computing a LFSR polynomial and seed in order to achieve good fault coverage is an extremely computationally demanding problem, thus we select it at random and evaluate the effectiveness.

Selection of a LFSR and a seed might significantly influence the fault coverage. The frequency distribution of covering a particular number of faults is illustrated by Fig. 4.5. Here sets of 50, 100, 500 and 1000 LFSR patterns were applied to the c3540 ISCAS circuit [Brg85], 1000 samples for each test size (see the four curves in Fig. 5.1). Each LFSR and its seed were selected randomly. The distribution of the number of faults, which remained undetected, is shown. We can see that it follows the Gaussian distribution. For a low number of patterns many faults are left undetected, while also their number varies a lot. When increasing the number of the test patterns the number of undetected faults rapidly decreases, while the variation of this number decreases as well. This means that when a high fault coverage is obtained by a long test sequence, the influence of the LFSR and seed on the fault coverage is negligible.

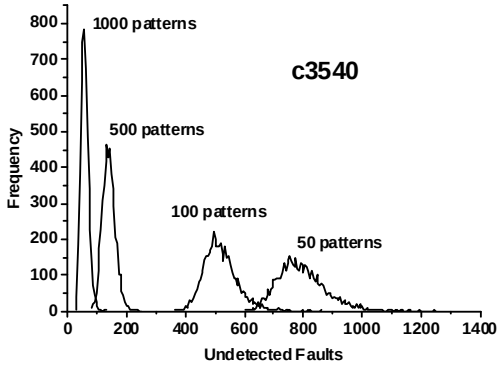

Figure 4.5: Pseudo-random fault coverage

The number of the covered faults as a function of the number of LFSR cycles applied to the CUT follows the well-known saturation curve shown in Fig. 4.6 (for the c3540 circuit [Brg85]). First few vectors detect the majority of faults, and then the fault coverage increases only slightly. The total number of detectable stuck-at faults is 3428. This number was not reached even after applying 50 000 LFSR cycles.

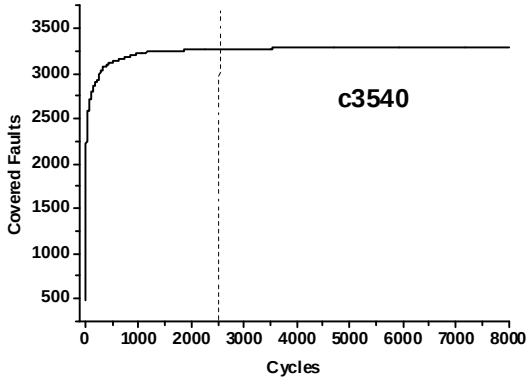

Figure 4.6: Fault coverage saturation curve

A conclusion can be made from these two graphs: in order to reach a satisfactory fault coverage by the first phase, we should determine the fault coverage saturation curve for the CUT by fault simulation. The appropriate length of the *PR* phase can be easily derived from it. The pseudo-random phase should be stopped when the fault coverage is not improving for a given number of cycles. This number can be freely adjusted, according to the application specific requirements (the trade-off between the test time and area overhead). Usually, we set this threshold to 1000 cycles. Thus, for the c3540 benchmark we determine *PR = 2500* cycles (see Fig. 4.6). The influence of the test length on the final result is discussed in [Fis04c].

4.5 Influence of the LFSR on the BIST Design Process

The fault coverage reached in the first phase is not influenced only by the length of the pseudo-random test. The number of detected faults also depends on the properties of the pseudo-random sequence, thus it is influenced by the LFSR polynomial and seed. For different LFSRs, significantly different results are produced, even when the lengths of the phases are retained. For illustration, we have designed a BIST for the c1908 ISCAS benchmark circuit [Brg85]. The pseudo-random phase was run for 2000 cycles, the LFSR polynomial was set constant (1-tap, see [Fis04b]) and we have repeatedly randomly reseeded it. Then the deterministic phase was run for 1000 clock cycles. The simulation results are shown in Table 4.1. The *"ud."* column indicates the number of undetected faults in the first phase, *"vct."* gives the number of deterministic vectors, *"GEs"* shows the complexity of the resulting BIST structure, in terms of the gate equivalents [DeM94]. The entries are sorted by the number of faults not detected in the pseudo-random phase.

We can see that the complexity of the final circuit strictly depends on the LFSR seed selected – it varies from 7.5 GEs up to 69 GEs.

To compute a proper LFSR seed and/or generating polynomial analytically is impossible for practical examples, due to the complexity of this problem. Thus, in praxis we repeatedly reseed the polynomial and conduct the fault simulation several times, while we pick out the

best seed for further processing. The fault simulation is often a very fast process, thus it does not significantly influence the BIST design time.

Table 4.1: Influence of the LFSR seed

ud.	*vct.*	*GEs*	*ud.*	*vct.*	*GEs*
19	10	7.5	33	15	37
21	9	19.5	34	16	33
24	13	23.5	36	18	38
26	15	28	37	20	40.5
26	13	25	39	22	53
28	15	37.5	44	26	40
28	14	22.5	46	22	42.5
30	14	36	48	24	44
32	16	31	52	28	63.5
33	17	27.5	62	34	69

4.6 The Deterministic Phase

In the deterministic phase the deterministic vectors are synthesized from some of the LFSR patterns that follow the pseudo-random phase. To do so, the *Column-Matching* algorithm is used. First, let us state the problem formally.

4.6.1 Problem Statement

Let us have an *n*-bit PRPG running for *p* clock cycles in the deterministic phase. The code words generated by this PRPG can be described by a **C** matrix (*code matrix*) of dimensions (*p, n*). These code words are to be transformed into the test patterns pre-computed by some ATPG tool. They are described by a **T** matrix (*test matrix*). For an *r*-input CUT and the test consisting of *s* vectors the **T** matrix will have dimensions (*s, r*). The rows of the matrices will be denoted as *vectors*.

The tests can be presented either in a form of deterministic patterns (minterms) or they may contain don't care values, depending on the ATPG algorithm used for the test set generation. We can take advantage of these don't cares in our algorithm, since they give us more freedom to select the column matches.

There are some obvious restrictions for the matrices dimensions. The number of test patterns *p* must be maximally $2^n - 1$ (the maximum number of distinct patterns generated by a LFSR) and $p \geq s$, because there must be enough patterns to implement all test vectors generated by the ATPG. On the other hand, there are no strict requirements regarding the relationship of *n* and *r*, since the number of LFSR stages can be even smaller than the number of CUT inputs.

The output decoder logic modifies the **C** matrix vectors in order to obtain all the **T** matrix vectors. As the proposed method is restricted to combinational circuits, the order in which the test patterns are fed to the CUT is insignificant. Thus, the **T** matrix vectors can be reordered in any way. Finding a transformation from the **C** matrix to the **T** matrix means finding a coupling of each of the *s* rows of **T** matrix with rows of the **C** matrix – thus finding a *row assignment* (see Fig. 4.7), i.e., to determine which **C** matrix rows will be transformed to **T** matrix rows and how. The excessive patterns do not disturb testing, they only extend the

test length. If a low-power testing is required, we may use some pattern inhibition techniques - see [Gir99]. Our method can be easily modified under these considerations.

The ***Output Decoder*** is a combinational block that converts *s n*-dimensional vectors of the **C** matrix into *s r*-dimensional vectors of the **T** matrix. The decoder is represented by a Boolean function with *n* inputs and *r* outputs, where only values of *s* terms are defined and the rest are don't cares. This Boolean function can be easily described by a truth table, where the output part corresponds to the **T** matrix, while the input part consists of *s* **C** matrix vectors assigned to the **T** matrix rows. The set of such vectors will be denoted as a *pruned **C** matrix*.

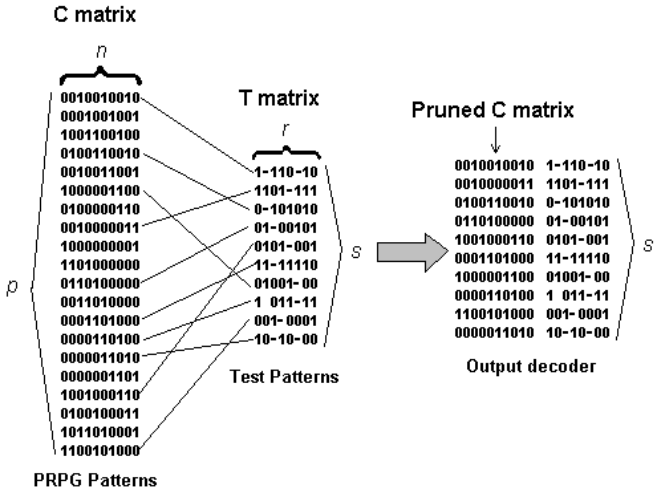

Figure 4.7: Assignment of the rows

4.6.2 The Column-Matching Method

The column-matching method is based on assigning all the **T** matrix rows to some of the **C** matrix rows so that some columns of the **T** matrix will be ***equal*** to some columns in the pruned **C** matrix. This yields absolutely no logic necessary to implement these **T** matrix columns (output variables of the decoder); they are implemented as mere wires.

The principles of the column matching are shown in Fig. 4.8. The ten LFSR patterns represented by a **C** matrix are to be transformed into 10 deterministic test vectors described by a **T** matrix. The PRPG outputs are entering the output decoder as variables $x_0 - x_4$, the outputs of the decoder (thus the CUT inputs) are denoted as y_0 - y_4. A case of a test without don't cares was chosen for simplicity. Two column matches can be made in this example. The **C** matrix column x_2 has been matched with the **T** matrix column y_4, similarly x_3 with y_0 has been matched. Thus, the outputs y_0 and y_4 are implemented without any combinational logic, while the remaining outputs have to by synthesized using some standard two-level Boolean minimization tools, like ESPRESSO [Bra84] or BOOM [Hla01, Fis03b]. The matches were obtained by reordering the **T** matrix rows, in order to have equal values in the corresponding columns. The **C** matrix rows are then assigned to the **T** matrix rows in the ascending order (A-h, B-g, C-e, etc.). Thus, the whole TPG generates the test patterns in the order h-g-e-i-, etc.

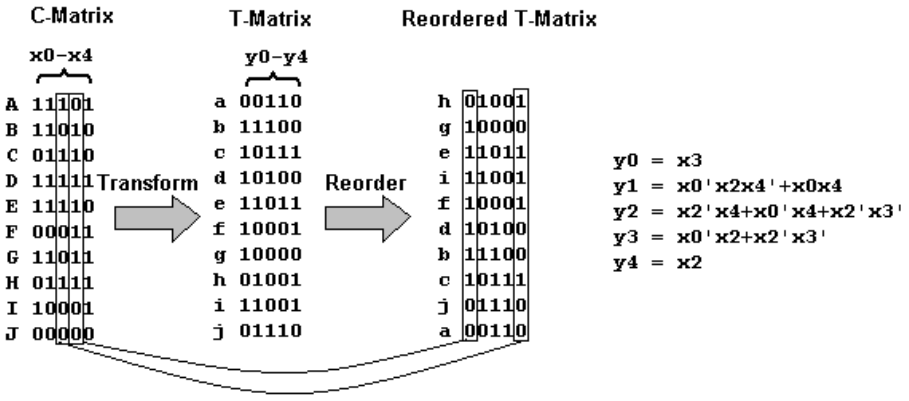

Figure 4.8: Column matching example

4.6.3 One-to-One Assignment

As a one-to-one assignment will be denoted the case where $p = s$, thus all the PRPG vectors are to be assigned to the test vectors and no idle PRPG cycles are present. In this case the minimum number of PRPG vectors is needed to generate the deterministic test vectors, however, the amount of logic needed to implement the output decoder is often large.

Generally, when doing the column matching, some restrictions for the **C** and **T** matrix rows that are to be assigned to each other must be applied every time a column match is done. If, e.g., the *i*-th **C** matrix column is matched with the *j*-th **T** matrix column, the **C** matrix rows containing "1" value in the *i*-th column can be assigned only to the **T** matrix rows containing "1" value in the *j*-th column and vice versa. If the test set contains don't cares, the **T** matrix rows having a don't care in the *j*-th column can be assigned to *any* **C** matrix row (while respecting the restrictions given by the previously made matches). The don't cares are substituted by "0" or "1" values form the **C** matrix after the final row assignment is known. The column-matching process for a test with don't cares will be described into detail in Section 7.

The most important feature of the one-to-one assignment is the fact that all the PRPG vectors that are to be transformed into test patterns are known in advance; there are no excessive vectors. Determining a column match is then a simple task: it is possible to make a match if the counts of ones (and zeros) in the corresponding columns are *equal*. In our previous example (Fig. 4.8) the counts of ones in the **C** matrix for columns x_0-x_4 are {6, 7, 5, 7, 6}, the counts of ones in the **T** matrix for columns y_0-y_4 are {7, 5, 5, 4, 5}, thus there are five possible column matches {x_1-y_0, x_3-y_0, x_2-y_1, x_2-y_2, x_2-y_4}.

After selecting a column match the two matrices are decomposed into two disjoint parts containing the rows with zeros and ones respectively in the matching columns, let the submatrices be denoted as $\mathbf{C_0}$, $\mathbf{C_1}$ and $\mathbf{T_0}$, $\mathbf{T_1}$. Then any vector from the $\mathbf{T_0}$ submatrix can be assigned to any vector from $\mathbf{C_0}$, as well as any vector from the $\mathbf{T_1}$ submatrix can be assigned to any vector from $\mathbf{C_1}$, but not otherwise. In our example, when the x_2-y_4 match is selected first, $\mathbf{C_0}$ = {B, F, G, I, J}, $\mathbf{C_1}$ = {A, C, D, E, H}, $\mathbf{T_0}$ = {a, b, d, g, j}, and $\mathbf{T_1}$ = {c, e, f, h, i}.

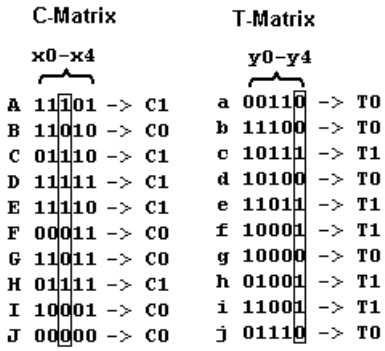

Figure 4.9: The first assignment to the submatrices

Finding all possible column matches consists in a successive decomposition of each of the original matrices into set systems until no further decomposition is possible. This happens when no more columns with equal one and zero counts are available in any two $\mathbf{C_i}$ and $\mathbf{T_i}$ submatrices.

The problem of selecting a proper set of column matches is NP-hard. Thus, the selection of the candidate columns for a match is controlled by a heuristic, which measures the proportion of zeros and ones in both the candidate columns and selects the most balanced decomposition. Another possibility is to use an *exhaustive column match search*, where all the possible combinations of column matches are tried. This method is applicable only to problems with a low number of possible column matches.

The output of this algorithm are two systems of subsets of the **C** and **T** matrices. Each two corresponding subsets contain vectors that can be assigned to each other in any order. We do the final assignment at random, since it influences the final result only negligibly (it influences only the final minimization).

4.6.4 Generalized Column Matching

In practice, it is often more advantageous to let the PRPG run more cycles than needed and pick out only several suitable vectors (see Fig. 4.7). Then idle test cycles are present, however this method significantly reduces the complexity of the output decoder.

The column matching principle is very efficiently applicable here. Unlike the method described in the previous subsection, we cannot determine a column match by comparing the number of ones in the corresponding columns, because we do not know in advance which **C** matrix vectors will be included in the final row assignment. However, we can freely choose among the code words (if $p >> s$). Finding an exact match is then a trivial problem; for several initial matches practically any two columns can be successfully matched.

Making an assignment of the **T** matrix rows to the **C** matrix rows is then very similar to the set system based method proposed above. Both the **C** and **T** matrices are being divided into two disjoint parts, while in this case their sizes need not be equal; the number of vectors in each $\mathbf{C_i}$ must be *greater or equal* to the number of vectors in the corresponding $\mathbf{T_i}$. If not, there would exist some test patterns that cannot have a **C** matrix vector assigned and then the matching procedure ends. After that, like in the original algorithm, some row-matching method is used to accomplish the final assignment of vectors.

The set system based column-matching algorithm is shown below. The inputs to the algorithm are the **C** and **T** matrices, the output is a valid system of sets $\mathscr{S}$ describing the total decomposition of the **C** and **T** matrix vectors. From this decomposition, the rows are assigned

to each other randomly and then the final result is obtained after completing a Boolean minimization.

Algorithm 4.1: Set System Based Column Matching

```
ColumnMatching(C, T) {
   𝒮 = {[C, T]};                          // initialize system of sets
   do {
      (i, j) = SelectColumnsToBeMatched(C, T);
      𝒮' = ∅;
      for (u = 0; u < |𝒮|; u++) {         // for all items in set system
            C⁰ = ∅;                       // generate subsets
            C¹ = ∅;
            for (k = 0; k < C_matrix_rows; k++)
                  if (𝒮_u^C[k, i] == 0) C⁰ = C⁰ ∪ 𝒮_u^C[k];
                  else C¹ = C¹ ∪ 𝒮_u^C[k];
            T⁰ = ∅;
            T¹ = ∅;
            for (l = 0; l < T_matrix_rows; l++)
                  if (𝒮_u^T[l, j] == 0) T⁰ = T⁰ ∪ 𝒮_u^T[l];
                  else T¹ = T¹ ∪ 𝒮_u^T[l];
            if (|C⁰| < |T⁰| || |C¹| < |T¹|) return 𝒮;
            𝒮' = 𝒮' ∪ {[C⁰, T⁰]; [[C¹, T¹]};     // add the split sets
            𝒮 = 𝒮';
      }
   }
}
```

4.6.5 Negative Column Matching

As we have described above, the idea of the column matching is based on finding a maximum of the decoder outputs that can be implemented just as wires, thus without any logic. This happens when the value of the matched output variable is equal to the value of some input variable in all care terms.

In most cases the PRPG outputs are drawn directly from the outputs of flip-flops. These flip-flops often have also the negative value of their outputs provided. Then, also the *negative matching* should be considered as a possibility to implement some variable of the output decoder as a simple wire. This happens when the value of the matched output variable is complement to the value of some input variable in all care terms. The possibility of a negative column matching should be then considered.

4.6.6 An ISCAS Benchmark Example

To illustrate the principles of the method we have chosen the c17 ISCAS benchmark [Brg85] for its simplicity. As an input to the algorithm we have a complete test set generated by an ATPG tool. The test consists of 10 test patterns (see Fig. 4.10). Our goal is to implement a BIST structure applying the given test set to the c17 benchmark circuit.

It should be mentioned that the test set shown in Fig. 4.10 is used here for strictly illustrative purposes. It is well known that c17 can be completely tested with 4 patterns and that, on the other hand, if we used an exhaustive test (which would be easy to implement due to the small size of the circuit), the output decoder circuitry would completely disappear.

```
01111
00001
01101
10001
01110
10111
00101
10011
00011
01000
```

Figure 4.10: ISCAS c17 test vectors

As a PRPG we have selected a 5-stage LFSR with generating polynomial $x^5 + x^2 + 1$ seeded with a vector 00010. In the following two subsections we will illustrate both the one-to-one assignment and the generalized matching process.

4.6.7 One-to-One Assignment for c17 Benchmark

In this example we show how the decomposition of matrices into set systems is being done for the one-to-one assignment into detail. We have two matrices as an input: the **C** matrix represents the patterns generated by the LFSR, the **T** matrix contains pre-generated test patterns shown in Fig. 4.10.

First, the counts of ones in all columns in both matrices are enumerated: for the **C** matrix these counts are {4, 4, 5, 5, 4}, for **T** matrix {3, 4, 5, 5, 8}. Thus, all possible column matches are {x_0-y_1, x_1-y_1, x_2-y_2, x_2-y_3, x_3-y_2, x_3-y_3, x_4-y_1}. At the beginning we select x_3-y_2 match and perform the decomposition of the matrices. Then the negative column match x'_2-y_3 is chosen and at the end we select the match x_1-y_1. No exact matches are possible any more, thus there has been three exact column matches found.

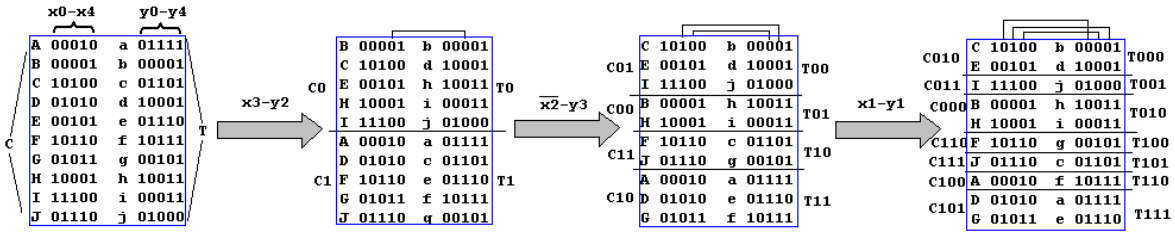

Figure 4.11: One-to-one exact column matching example

In all the subsets the $\mathbf{C_i}$ vectors are assigned to $\mathbf{T_i}$ vectors and the remaining logic is minimized by BOOM or ESPRESSO. The resulting schematic is shown in Fig. 4.12.

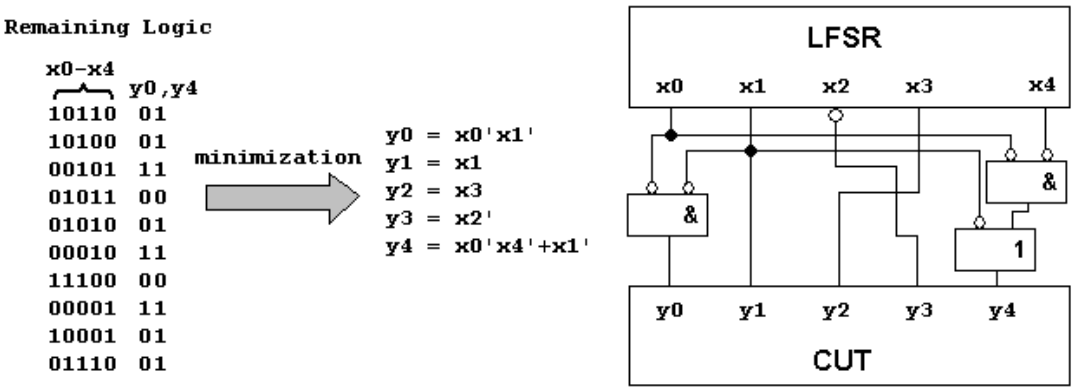

Figure 4.12: BIST implementation for c17 circuit

4.6.8 Generalized Column Matching Example

We have found three exact column matches for a one-to-one assignment in the previous example, whereas the decoder for the remaining two variables needed to be synthesized. Now we will try to let the LFSR run for more than the minimum required 10 cycles and see if more exact matches will be achieved.

We have found experimentally, that when we retain the LFSR generating polynomial and seed from the previous example, 19 LFSR cycles are needed to match *all* the columns. Thus, absolutely no additional logic is needed to build the output decoder. In Fig. 4.13 we show one of the possible assignments of the test patterns to 10 of the 19 LFSR patterns and the resulting combinational logic of the output decoder, which is formed just as a permutation of wires in this case. For comparison, let us note that an exhaustive test set having an equally simple output decoder would require 32 patterns. The exact column matches found for our example are obvious from the final solution.

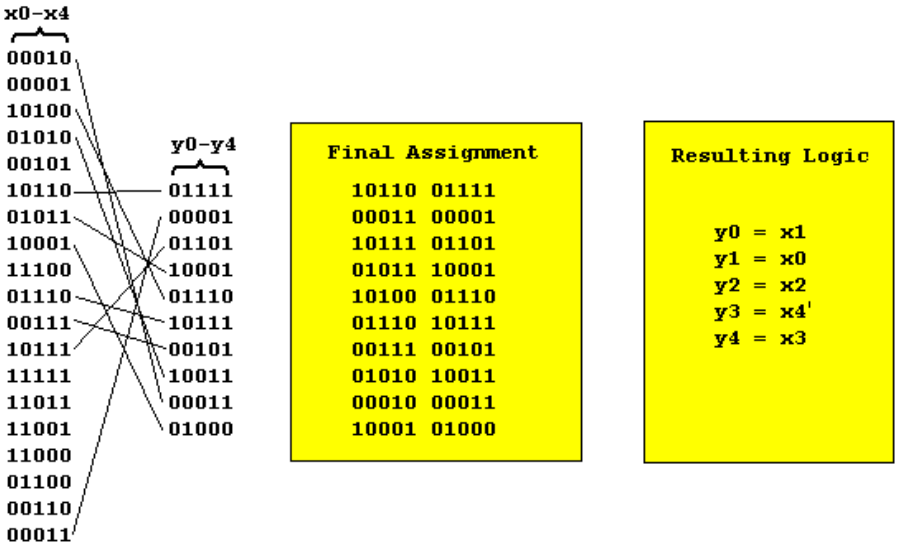

Figure 4.12: Assignment of rows for c17 circuit

4.7 Column Matching Exploiting Test Don't Cares

Until now, we have assumed that the **T** matrix contains only test patterns in their compacted form, i.e., minterms. Some ATPG tools produce test patterns containing don't care values (DCs). Such a test is often significantly longer than the compacted one, but on the other hand the don't cares can be advantageously exploited in the output decoder design.

The problem of constructing the output decoder is in this case similar to the previous one: all the **T** matrix vectors are to be assigned to the **C** matrix vectors, while $s \leq p$. The **T** matrix contains don't care values, the **C** matrix contains only minterms, since concrete vectors are produced by a PRPG.

When the don't cares are not present in the test set, each of the test vectors can be assigned to a set of PRPG patterns at every instant, while all these sets are disjoint. But when the don't cares are present, these sets become non-disjoint. This is because we cannot decide what values to assign to the don't cares, until all the matches are done. Thus the algorithm consists of two linked NP-hard problems. We have found that using the set system approach here is rather time-consuming, although it is not impossible.

An efficient heuristic based on a ***blocking matrix B*** has been proposed in [Fis03a]. The blocking matrix is a binary matrix (it contains only "0" and "1" values) of dimensions (p, s). Thus, it has as many columns as there are **T** matrix rows and as many rows as there are **C** matrix rows. The value "1" in the cell **B**[k, l] indicates that the k-th **C** matrix row may be assigned to the l-th **T** matrix row, "0" value indicates the contrary.

At the beginning of the algorithm all the **B** matrix cells are filled with a "1" value, since there are no restrictions for row assignments. After the i-th **C** matrix column is matched with the j-th **T** matrix column column, the **B** matrix cells [k, l] are set to "0" when the k-th input row contains in a i-th column the opposite value to the l-th output row in a j-th column. Thus, rows that contain opposite values in the matched columns cannot be assigned to each other.

$$\mathbf{B}[k, l] := \text{“0” when } (\mathbf{C}[k, i] \neq \mathbf{T}[l, j] \wedge \mathbf{T}[l, j] \neq \text{don't care}) \qquad (4.2)$$

If the negative column match is to be performed, the **B** matrix cells are set to "0" when equal values are present in the respective positions.

When making the row assignment, distinct rows have to be assigned to each other. It is a trivial problem for a test without don't cares, since there does not exist a **B** matrix row having "1" value in more than one column (one PRPG code word cannot be assigned to more than one test pattern). The final assignment then consists in selecting one row from the possible ones for each of the columns. Unfortunately, in the column matching exploiting don't cares the **B** matrix rows may have ones in more than one column, since some values in the test patterns will be determined after the assignment. This makes the assignment to be a NP-hard problem. An example of an assignment is shown in Table 4.2. Here all the output vectors t_1-t_6 are to be assigned to the LFSR vectors c_1-c_6. There are two possible solutions to this problem:

Table 4.2: Row assignment using a B matrix

	t_1	t_2	t_3	t_4	t_5
c_1	1	0	0	1	0
c_2	0	1	0	0	0
c_3	0	1	0	0	0
c_4	0	0	1	0	0
c_5	0	0	1	0	1
c_6	0	0	0	1	1

$t_1 - c_1$
$t_2 - c_2$ or c_3
$t_3 - c_4$
$t_4 - c_6$
$t_5 - c_5$

Since the **B** matrix is mostly rather large, solving this problem exactly becomes impossible. Thus some heuristic has to be used. Selecting a proper algorithm is of a key importance for reaching good results. For instance, if an assignment of c_1 to t_4 in Table 4.2 was chosen at the beginning, the algorithm would yield no solution – there won't be any possible assignment for t_1.

4.7.1 Row Assignment Algorithms

It would be often extremely time-consuming to solve this problem exactly, thus we use a greedy incremental heuristic. Since the column-matching algorithm needs to solve this problem after every column match, the row assignment heuristic should be fast. Moreover, the whole process is being guided by the result of the assignment. If the assignment fails, the column-matching will stop. Thus, the algorithm should be precise enough as well. For this reason we have tried out several methods and compared the results to select the best one.

One method (LCLR – *least in column, least in row*) uses a simple greedy heuristic. The **B** matrix column with the least number of "1" values is found (because the respective **T** matrix vector would be hard to assign) and the row having a "1" value in this column and the least "1"s in other columns is assigned to it (because the respective **C** matrix vector is not so "useful" for other assignments). If a column without any "1" value is found at some instant, the algorithm returns a failure and the whole column matching process is stopped (when no backtracking is used). The algorithm has not succeeded in finding an assignment in this case, however, there is still a possibility that there exists a solution.

The second, more sophisticated heuristic constructs a ***scoring matrix*** from which the best row assignments are being picked-up. It is similar to the **B** matrix, but any values can be contained in its cells. Each cell contains a value defining a "score" of a particular row assignment. It is computed by dividing the number of ones in a respective **B** matrix row by the number of ones in a respective **B** matrix column. An assignment having a biggest score is done, the matched row and column is removed from the **B** matrix and all the values are recomputed. The process is repeated until all test columns are assigned or an all-zero column is encountered.

We have tested the efficiency of the algorithms on the s526 ISCAS benchmark [Brg89] having 24 inputs, 1000 LFSR vectors were to be matched with 20 tests. We have run the column-matching algorithm 300 times in its thorough search mode (see later), while in each step a row assignment was performed repeatedly 1000 times, using both methods, plus a purely random assignment, just for a comparison. In those 300 iterations 80 000 runs of the row-assignment algorithm were required, from which 6500 were successful (there was a solution). Figures 4.14 and 4.15 show histograms of the frequencies of the successful hits in the 6500 row assignment passes for the three algorithms. Figure 4.15 is a close-up view on the unsuccessful tries.

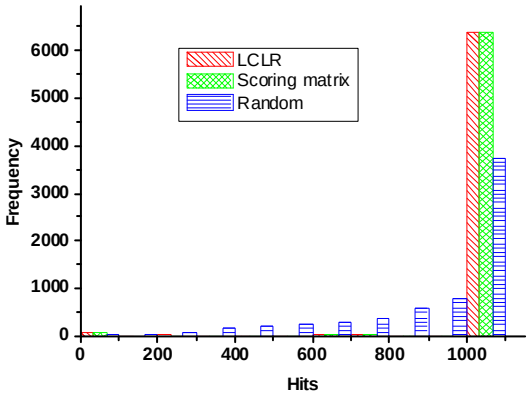

Figure 4.14: Row assignment histograms

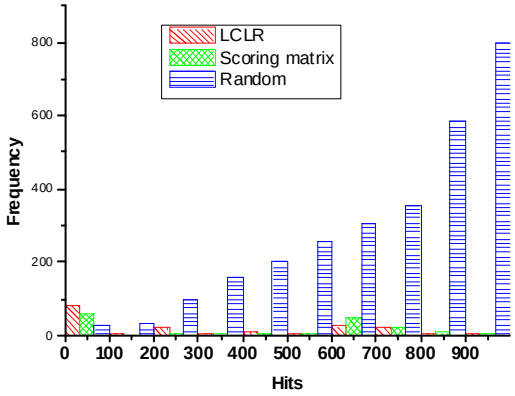

Figure 4.15: Close-up view of Fig. 4.14

We can see that in most cases both the LCLR and scoring matrix based heuristics found a solution, while the randomized method was not so successful. Particularly, the LCLR found an assignment in 97.3% of the possible cases, the scoring matrix based method in 97.6% and the random method in 57.2% only. The average runtimes with the percentage of the efficiency of all the heuristics are shown in Table 4.3. All the experiments were run on a PC with a 1200 MHz Athlon processor.

Table 4.3: Row assignment algorithms

algorithm	successfulness	runtime
LCLR	97.3%	0.28 ms
scoring matrix	97.6%	2.94 ms
random	57.2%	0.09 ms

We can conclude from these results that both the LCLR and scoring matrix based algorithms are extremely efficient, unlike the random approach. Both the algorithms are almost equally successful, however the scoring matrix method is more than 10 times slower. For this reason, in all our experiments we use the LCLR row assignment algorithm. Since for all the columns of the **B** matrix rows values in all the rows have to be examined in a case

of a successful assignment, the time complexity of the algorithm is O($p \cdot s$). The algorithms are described in [Fis04d] as well.

4.7.2 The Column Matching Algorithms

We have developed several algorithms driving the whole column-matching process. In the ***exact search*** all the possibilities for all the matches are explored, which always yields the optimum solution, in terms of the number of matches achieved. However, the time complexity of this algorithm grows exponentially with the number of output variables, thus it is not feasible to use it for practical problems.

Then a simple heuristic can be used: when a non-valid column match is encountered (during the row assignment process), the whole process could be stopped. This is the fastest algorithm, which is often suitable for problems with a large number of variables. Because the row assignment is repeated after every column match and there could exist at most m column matches, the worst-case complexity of this algorithm would be O($m \cdot p \cdot s$). It corresponds to a case where all the m column matches were found. This algorithm will be denoted as a ***fast search***.

The result may be further improved by trying other possibilities for a column match if one column match fails. This would significantly increase the runtime. We call this algorithm a ***thorough search***. The worst-case complexity increases to O($n \cdot m^2 \cdot p \cdot s$), however the best-case complexity is equal to the *fast search* case. A typical progress of a thorough search is shown in Fig. 4.16. Here the s526 ISCAS benchmark [Brg89] having 24 inputs was solved. The test set consisted of 20 vectors and these had to be matched to 1000 LFSR vectors. A simple *fast search* would end after only 3 column matches (after 30 ms), while the *thorough search* ran for 198 cycles, but reached 21 column matches (in 200 ms). From this example it is obvious that the thorough search significantly outperforms the fast search.

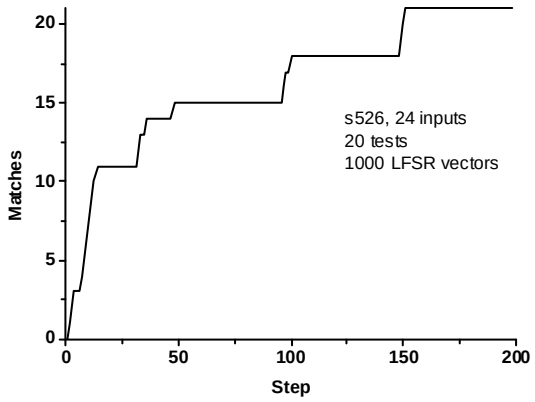

Figure 4.16: Thorough search progress

Several modifications can be yet done to improve the result quality. The selection for column matches is being done purely at random. Thus, when the whole column matching process is repeated several times, there is a chance that we will reach a better solution. After every repetition the number of column matches reached is compared with the previously reached one, and if it is bigger, it is recorded as the so far best solution. For the *fast search* it is the only possibility to reach a good solution. Here the column matching can be even further sped up: it is not necessary to perform a row assignment after each column match – the number of up to now obtained maximum of the column matches is performed and after that

it is checked for validity (by making a row assignment). When it is not valid, the whole solution is rejected, since it cannot improve the overall solution. The ***repetitive fast search*** might be a good way to improve the result quality for problems with a large number of variables, however it often never outperforms the thorough search, in terms of the number of column matches reached.

The improvement of the number of column matches reached is visualized by Fig. 4.17. Here the same problem as in the previous example was solved by a fast search repetitively 1000 times. After the first run only 5 column matches were obtained, however in the 464th pass 19 matches were found. More matches were not found in the following passes.

The whole process had run 11.5 seconds. Let us remind for comparison that the thorough search had found 21 matches in 200 ms.

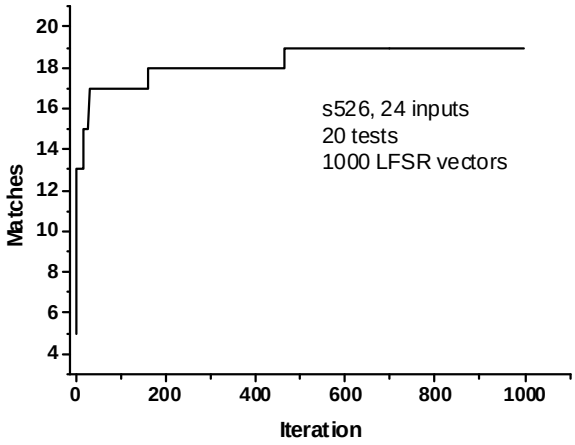

Figure 4.17: Repetitive fast search

The thorough search can be augmented by repetition as well. Unfortunately, the speedup method mentioned above cannot be used here. On the other hand, other techniques can be applied. Since the row assignment is quite a time-consuming process, we try to avoid it at any cost. One possibility is to keep a *history*. After each run of the whole algorithm we store the column matches obtained into a special buffer. In all the following runs we check the possibility for a column match in this buffer, and only if it is not found, the row assignment is performed. The buffer can be efficiently constructed as a tree, where the result is obtained in *m* steps at most.

Further improvement of the thorough search algorithm consists in applying a *backtracking* technique. At the end of the search, when everything fails, one of the column matches is taken back and the search continues (while the removed match cannot be repeated). This means a big increase of a runtime – consider that all the unsuccessful matches have to be repeated. Hence, the backtracking technique was found to be not this efficient, the repetitive thorough search yields better results.

4.7.3 The Basic Algorithm

The summary of the basic *fast search* column matching algorithm is presented in this subsection.

Since the number of the **C** matrix rows is often much higher then the number of the **T** matrix rows, finding several initial column matches is a trivial problem: almost any two

columns can be matched, because there is a big choice of possible assignments for the **C** matrix rows. Thus the selection of the rows to be matched is done at random.

When two columns to be matched are selected, the match must be checked for validity using a **B** matrix. Thus, after each column match the row assignment has to be performed to determine whether the match is valid. If the assignment fails the column matching process is terminated and the last valid assignment is considered as a final result. The row assignment forms a truth table, which has to be further processed. Firstly, the test don't cares in the matched **T** matrix columns are substituted by "0" and "1" values according to the values of the corresponding **C** matrix columns. Since most of the tests including don't cares are not in a compacted form (e.g., there is one test pattern for each of the s-a faults), some test compaction technique [Ham98] should be applied after the column matching. This often reduces the length of the BIST, and it reduces the amount of the output decoder logic as well. Then the matched output variables are removed from the truth table and the values of the remaining output variables are synthesized by some standard Boolean minimizer [Bra84, Fis03b].

The algorithm can be described by the following pseudo-code. The inputs of the algorithm are the **C** and **T** matrices, the output is in the form of a minimized Boolean function.

Algorithm 4.2: Fast Search Column Matching

```
ColumnMatching(C, T) {
   for (k = 0; k < C_matrix_rows; k++)          // initiallize B matrix
      for (l = 0; l < T_matrix_rows; l++)
         B[k, l] = "1";
   A = ∅;
   do {
      i = random(C_matrix_columns);             // randomly select columns
      j = random(T_matrix_columns);
      for (k = 0; k < C_matrix_rows; k++)       // modify blocking matrix
       for (l = 0; l < T_matrix_rows; l++)
                if (T[l, j] ≠ DC && C[k, i] ≠ T[l, j]) B[k, l] = "0";
      A' = A;                          // make backup of the row assignment
      A = MakeRowAssignment(B);        // make row assignment
   } while (A ≠ FAILED);
   Substitute_DCs(T);                  // substitute test DCs with "0" or "1"
   CompactTest(T);                     // make test compaction
   ExtractMatches(C, T);               // remove matched outputs
   F = Minimize(A')                    // synthesize the remaining logic
   return F;
}
```

4.7.4 Overview of the Column-Matching Alternatives in Mixed-Mode BIST

Up to now it has been assumed that applying a column match means no hardware to implement one output. Obviously, when no column match for a particular output is found, some combinational logic has to be added to the Output Decoder. For a mixed-mode BIST, namely when the test is divided into the two above-mentioned phases, the Switch is present as well. Our aim is to minimize both the Output Decoder and the switching logic. There are five possibilities for a particular output decoder output:

- There has been found a column match between the output variable y_i and the input variable x_i. Then y_i will be implemented as a wire, without any output decoder logic. Moreover, there will be no switching logic for this output; the CUT is being fed directly

by an LFSR output. In our example (Fig. 4.4) it is a case of y_0 and y_1. Such a case will be denoted as a ***direct column match***.

- There has been found a negative column match between the output variable y_i and the input variable x_i. Then the decoder logic for y_i could be implemented as a negator. The switching logic for y_i will be a multiplexer. In praxis, it is more advantageous to join these two gates into a single XOR gate. In our example (Fig. 4.4) it is a case of y_2. Such a case will be denoted as a ***negative direct column match***.
- The variable y_i has been matched with the x_j variable, while $i \neq j$. If the first BIST phase weren't present, y_i would be implemented as a wire. In mixed-mode BIST there has to be a multiplexer switching y_i between x_i and x_j LFSR outputs added. In Fig. 4.4 it is the y_3 case. Such a match will be denoted as an ***indirect column match***.
- An ***indirect negative column match*** is a similar case. Here an inverter has to be added to the matched LFSR output. However, the D flip-flops used in the LFSR are often provided with the negated output as well, so no additional inverter would be needed in this case.
- No column matching was found for some y_i. Here the output decoder has to synthesize the proper output values, while an additional multiplexer has to be present in the switching block. This is a case of y_4 in Fig. 4.4.

The first case mentioned is, of course, the one with the lowest BIST area overhead, in the latter ones the overhead gradually increases. Thus, the intention of the algorithm should be to prefer the direct matches, and only when no such are possible, the indirect column matches should be made. This is the way how the column-matching heuristic selects the candidates to match – it gradually scans all the unmatched output variables for a possibility for a direct column match. When one is found, it is performed and the search continues. When there is no possibility for a direct match any more, the indirect ones are being made. When no matches are possible, the resulting outputs are synthesized by BOOM [Hla01, Fis03b].

Chapter 5

Experimental Results

5.1 Influence of the Length of the Pseudorandom Phase

To illustrate the importance of properly choosing the parameters of the pseudo-random phase we have designed a BIST structure for several ISCAS benchmarks [Brg85, Brg89]. We have varied the length of the pseudo-random phase, while the length of the deterministic phase was kept constant, 1000 cycles. As a fault simulator FSIM was used [Lee91], as an ATPG we have used Atalanta [Lee93]. For all the benchmarks a test covering all the irredundant faults was produced by this tool.

The results are shown in Table 5.1. The benchmark name and the number of its inputs are shown in the first two columns. The *"PR"* column indicates the length of the pseudo-random phase, the *"UD"* column shows the number of s-a faults that were left undetected after the "*PR*" pseudo-random cycles. *"vct."* gives then the number of deterministic vectors generated to test these faults. The *"M"* column shows the total number of column matches obtained, *"DM"* the number of direct column matches. The *"SW GEs"* column describes the complexity of the Switch and *"OD GEs"* column of the Decoder, in terms of the gate equivalents [DeM94]. These two values are summed together in the next column, to obtain the total area overhead of the combinational block. The time needed to complete the column-matching procedure is indicated in the last column. The runtimes of the fault simulation and Boolean minimization were negligible comparing to the column-matching runtimes. The experiment was run on a PC with Athlon CPU, on 1 GHz, under Windows XP.

Table 5.1: Influence of the pseudo-random phase on the result

bench	*inps*	*PR*	*UD*	*vct.*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*
c1908	33	1000	65	39	20	11	33	48	81	4.88
		2000	23	10	33	23	15	**0**	15	0.18
c2670	233	1000	309	86	193	173	90	109.5	199.5	166
		2000	306	86	192	175	87	102.5	189.5	166
		5000	216	73	198	164	103.5	91	194.5	143
		10000	154	69	199	178	82.5	84	166.5	123
c3540	50	300	165	66	38	29	31.5	78	109.5	10.26
		500	92	42	44	29	31.5	25	56.5	3.88

		1000	36	26	49	32	27	1	28	1.02
		2000	9	9	50	41	13.5	**0**	13.5	0.19
		5000	1	1	50	49	1.5	**0**	1.5	0.02
s1196	32	200	228	104	26	25	10.5	100	110.5	5.05
		500	141	79	27	23	13.5	63.5	77	3.87
		1000	90	51	27	24	12	38.5	50.5	2.00
		2000	52	37	28	23	13.5	23.5	37	1.20
		5000	23	17	29	25	10.5	6.5	17	0.48
		10000	9	4	32	28	6	**0**	6	0.04

It can be concluded from this table that the pseudorandom phase plays a very important role here. If its length is selected so that many easy-to-detect faults are covered by it, only few faults are to be covered by the deterministic phase, thus the Decoder logic would be negligible. However, for circuits having a large number of hard-to-detect faults (c2670) the amount of the Decoder logic cannot be influenced by this phase too much.

5.2 The Deterministic Phase

In the deterministic phase deterministic vectors are synthesized from some of the LFSR patterns that follow after the pseudo-random phase. With increasing number of LFSR patterns the chance to find more column matches increases as well. This is due to having more freedom for selecting the LFSR vectors to be assigned to the deterministic vectors. However, with the number of vectors the design runtime rapidly increases.

This is illustrated by Table 5.2. Its format is retained from Table 5.1, the "*Det.*" column indicates the length of the deterministic phase.

It can be observed that a trade-off between the test time and area overhead can be freely adjusted here too, according to demands of the BIST designer.

The lengths of both the phases significantly influence the BIST design time as well. The design process is being sped up when increasing the length of the pseudo-random phase, since the number of deterministic vectors is being reduced this way. On the other hand, an increasing length of the deterministic phase slows down the process.

Table 5.2: Influence of the deterministic phase on the result

bench	*inps*	*PR*	*Det.*	*vct.*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*
c1908	33	1000	500	39	18	9	36	54.5	90.5	1.6
			1000		20	11	33	48	81	4.88
			2000		20	13	30	50	80	8.47
			5000		22	13	30	38.5	68.5	25.78
c3540	50	1000	200	26	48	31	28.5	5.5	34	0.32
			500		49	31	28.5	1	29.5	0.52
			1000		49	32	27	1	28	1.02
			2000		50	39	16.5	**0**	16.5	1.47
			5000		50	45	7.5	**0**	7.5	2.93
s1196	32	5000	200	23	27	22	15	10.5	25.5	0.17
			500		29	20	18	7	27	0.32
			1000		29	25	10.5	6.5	17	0.48

			2000		29	26	9	8	17	1.52
			5000		31	27	7.5	1.5	9	2.16
			10000		32	29	4.5	**0**	4.5	5.83

5.3 Comparison of the Results

We have compared our results with two state-of-the-art methods, namely the bit-fixing method [Tou95] and the row matching method proposed in [Cha03]. The comparison is shown in Table 5.3. The *"TL"* columns indicate the total length of the test, the *"GEs"* columns give the number of gate equivalents of the BIST combinational circuits. The column-matching GEs in bold indicate that our method was better than both the other methods in the particular case, in terms of the complexity of the transforming combinational logic. Let us note here, that a special kind of a PRPG is used in the row-matching approach [Cha03]. Such a circuit causes quite a large area overhead in most cases, for many XOR gates present. This overhead is not included in the table. Our method is independent on a PRPG used, in general, thus in all the cases we have used an LFSR with two XOR gates only, independently on its width. Thus, sometimes bigger area overhead of our method could be compensated by a small area of the PRPG used. The empty cells indicate that the data for the respective circuit was not available to us.

Table 5.3: Comparison results

	Column-matching		Bit-fixing		Row-matching	
Bench	*TL*	*GEs*	*TL*	*GEs*	*TL*	*GEs*
c880	1 K	**10.5**	1 K	27	1 K	21
c1355	2 K	15	3 K	11	2 K	0
c1908	3 K	**7.5**	4 K	12	4.5 K	8
c2670	5 K	172	5 K	121	5 K	119
c3540	5.5 K	**1.5**	4.5 K	13	4.5 K	4
c7552	8 K	586	10 K	186	8 K	297
s420	1 K	**24.5**	1 K	28	-	-
s641	4 K	15	10 K	12	10 K	6
s713	5 K	16.5	-	-	5 K	4
s838	6 K	130	10 K	37	-	-
s1196	10 K	**6**	-	-	10 K	36

5.4 Results for Standard Benchmarks

Since the comparison shown in Table 5.3 describes results for a few benchmark circuits only, we will present a more exhaustive result table, for most of ISCAS [Brg85, Brg89] benchmarks. For each benchmark the BIST circuitry was synthesized in two modes – for the first one, the test length was set to be relatively small (the white rows). In the second one the test was longer, to keep the area overhead as small as possible. Thus, the tradeoff between the test length and area overhead can be seen well.

The *"inps"* column indicates the number of the benchmark inputs, in the *"PRand"* column the number of pseudo-random vectors needed to be applied to the CUT to be completely tested is shown, just to show the effectiveness of the method. The *"TL"* column gives the

lengths of the pseudorandom and deterministic phases. The *"M"* and *"DM"* columns show the number of total and direct column matches reached. The complexity of the switching logic is shown in the *"SW GEs"* column, the complexity of the output decoder in *"OD GEs"*. These numbers are summed together in the *"Total GEs"* column. The runtime needed to complete the column-matching process is indicated in the last column.

Table 5.4: ISCAS benchmarks

Bench	*inps*	*PRand*	*TL*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*
c880	60	2.5 K	100 + 100	53	22	57	12.5	69.5	0.50
			500 + 500	**60**	50	15	**0**	15	0.04
c1908	33	3 K	1000 + 500	18	9	36	54.5	90.5	1.6
			2000 + 500	**33**	16	25.5	**0**	25.5	0.14
c2670	233	2.4 M	1000 + 1000	193	173	90	109.5	199.5	166
			10000 + 5000	204	179	81	73.5	154.5	673
c3540	50	5 K	1000 + 500	**50**	34	24	**0**	24	0.40
			2000 + 1000	**50**	41	13.5	**0**	13.5	0.19
c5315	178	2 K	500 + 500	168	121	85.5	20.5	106	6.43
			1000 + 500	**178**	154	36	**0**	36	0.13
c7552	207	> 100 M	7000 + 1000	131	33	261	325	586	500
			10000 + 2000	133	36	256.5	248.5	505	887
s420	34	165 K	400 + 600	32	21	21	3.5	24.5	0.75
			3000 + 1000	35	21	21	**0**	21	0.41
s526	24	5 K	500 + 500	21	20	6	8	14	0.20
			1000 + 1000	**24**	21	4.5	**0**	4.5	0.11
s641	54	200 K	500 + 500	52	40	21	**2**	23	0.47
			3000 + 1000	**54**	44	15	**0**	15	0.21
s713	54	300 K	500 + 500	52	38	24	3	27	0.56
			3000 + 1000	**54**	42	18	**0**	18	0.32
s820	23	10 K	1000 + 1000	20	19	6	9.5	15.5	0.50
			3000 + 1000	**23**	18	7.5	**0**	7.5	0.15
s832	23	10 K	1000 + 1000	19	19	6	8.5	14.5	0.40
			3000 + 1000	22	19	6	2	8	0.20
s838	67	> 100 M	1000 + 1000	37	13	81	45	126	26.20
			10000 + 2000	46	14	79.5	29	108.5	51.51
s953	45	15 K	1000 + 1000	42	38	10.5	6	16.5	1.23
			2000 + 1000	**45**	39	9	**0**	9	0.58
s1196	32	200 K	2000 + 1000	28	23	13.5	23.5	37	1.20
			9000 + 1000	**32**	28	6	**0**	6	0.04
s1238	32	20 K	1000 + 1000	27	22	15	43	58	2.51
			5000 + 1000	30	23	13.5	4.5	18	0.44
s1423	91	10 K	1000 + 1000	89	63	42	2	44	1.43
			5000 + 1000	91	82	13.5	**0**	13.5	0.06
s1488	14	2 K	300 + 200	9	8	9	55.5	64.5	0.13
			500 + 500	13	10	6	2	8	0.12
s1494	14	2 K	300 + 200	10	8	9	41	50	0.12
			500 + 500	12	12	3	13	16	0.12

Chapter 6

Conclusions and Future Work

A mixed-mode BIST method based on the *column-matching* approach has been proposed. Here the pseudorandom LFSR code words are being transformed into deterministic test patterns computed by some ATPG tool. The transformation is being done by a purely combinational block. Here we try to match as many of its outputs as possible with its inputs, which yields no logic necessary to implement these outputs.

The method is designed for a test-per-scan BIST, however it can be easily adopted to full-scan or multiple-scan circuits. The pseudo-random and deterministic phases are separated, which enables to reach smaller area overhead. The method is based on a design of a decoder transforming the LFSR code words into deterministic test vectors testing the hard-to-detect faults. In all the mixed-mode designs, some kind of switching logic is involved. A method reducing both the transformation and switching logic is proposed here.

The test is divided into two phases, the pseudo-random and deterministic. The lengths of both phases might be freely adjusted, to find a trade-off between the test time and area overhead. It has been shown that the length of the pseudo-random phase has a crucial impact on the result and we present a methodology for choosing its length efficiently.

The length of the deterministic phase influences the result as well, however not too significantly. The impact of the test lengths on the duration of the BIST design process is considered as well.

A big scalability of the method, in terms of the area overhead, test time and design time was shown.

Our BIST method can be used for any fault model, if a proper fault simulator and ATPG tool is provided. The fault coverage reached is dependent only on the ATPG tool as well; a trade-off between the fault coverage and BIST area overhead can be adjusted too.

The method was tested on the standard ISCAS benchmarks and the results were compared with other state-of-the-art methods.

As the future work we plan to do several minor modifications, which could help us to slightly reduce the complexity of the resulting BIST. Namely it is using cellular automata or other more complex structures as the PRPG.

More essential modification of the algorithm will enable us to adjust the width of the PRPG. Until now, we have assumed that the number PRPG outputs is equal to the number of CUT inputs, at least in the mixed-mode method. There would be no modification of the algorithm for this case. However, for a wider PRPG the algorithm cannot decide what PRPG outputs should be connected to the CUT inputs in the pseudo-random phase - until now they

are connected in an ascending order, however it is possible to choose any other order. This problem gives us a hint for another possibility of improvement of our algorithm – to consider a permutation of wires, not to just connect it straight.

This would be possible to do by incorporating the ATPG tool into the algorithm more extensively. Particularly, the deterministic test won't be generated in one step, but iteratively with a chance to change unwanted tests and to enable the column-matching algorithm to take hints from the ATPG. For example, for a particular set of faults we will be able to select a test vector having don't care values in the positions of the already matched columns. Thus, the restrictions put on the following column match will be reduced.

Such a major modification could significantly reduce both the area overhead and the test length.

To be able to cope with most of VLSI core designs we will modify our method to support the test-per-scan BIST, even for multiple scan-chains.

Larger circuits are often hard to test, especially for their huge number of inputs (arising from the scan-chain). Thus, we will try to propose a partitioning method, splitting large circuits into smaller ones, for which would be the BIST constructed separately. Such a partitioning should be done in such a way that the CUT performance should not be affected, nor the area overhead would significantly increase.

Then, after all, we plan to combine our method with other methods, namely to exploit the reseeding principle. This would make Column-matching a universal BIST design method.

List of Abbreviations

ATE .....Automatic Test Equipment
ATPG .....Automatic Test Pattern Generator
BIST .....Built-in Self-Test
CUT.....Circuit under Test
DC .....Don’t Care
FSM.....Finite State Machine
GE .....Gate Equivalent
LFSR .....Linear Feedback Shift Register
MISR.....Multiple-Input Shift Register
PRPG.....Pseudo-Random Pattern Generator
RE.....Response Evaluator
TPG .....Test Pattern Generator

List of Symbols Used

C.....*PRPG code words matrix*
T.....*Test matrix*
i .....*a particular **C** matrix column (to be matched)*
j .....*a particular **T** matrix column (to be matched)*
k.....*a particular **C** matrix row*
l .....*a particular **T** matrix row*
m.....*number of column matches*
n.....*number of PRPG bits; number of **C** matrix columns*
p.....*number of PRPG cycles; number of **C** matrix rows*
r.....*number of CUT inputs; number of **T** matrix columns*
s.....*number of deterministic test vectors; number of **T** matrix rows*
x_i.....*input variable (LFSR output, Decoder input)*
y_j.....*output variable (Decoder output, CUT input)*

References

[Aga93] V.K. Agarwal, C.R. Kime and K.K. Saluja. *A tutorial on BIST, part 1: Principles*, IEEE Design & Test of Computers, vol. 10, No.1 March 1993, pp.73-83, part 2: Applications, No.2 June 1993, pp. 69-77

[Alo93] K. Aloke, K. and D.P. Chaudhuri. *Vector Space Theoretic Analysis of Additive Cellular Automata and Its Application of Pseudoexhaustive Test Pattern Generation*, IEEE Transactions on Computers, Vol. 42, No. 3, March 1993, pp. 340-352

[AlS94] M.F. AlShaibi and C.R. Kime. *Fixed-Biased Pseudorandom Built-In Self-Test for Random Pattern Resistant Circuits*, Proc. of International Test Conference, pp. 929-938, 1994

[Bar87] P.H. Bardell, W.H. McAnney and J. Savir. *Buit-In Test for VLSI: Pseudorandom Techniques*, New York: Wiley, 1987

[Bra84] R.K. Brayton, et al. *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA, Kluwer Academic Publishers, 1984

[Brg85] F. Brglez and H. Fujiwara. *A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan*, Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985

[Brg89] F. Brglez, D. Bryan and K. Kozminski. *Combinational Profiles of Sequential Benchmark Circuits*, Proc. of International Symposium of Circuits and Systems, pp. 1929-1934, 1989

[Cha95] M. Chatterjee and D.K. Pradhan. *A novel pattern generator for near-perfect fault coverage*, Proc. of VLSI Test Symposium 1995, pp. 417-425

[Cha03] M. Chatterjee and D.K. Pradhan. *A BIST Pattern Generator Design for Near-Perfect Fault Coverage*, IEEE Transactions on Computers, vol. 52, no. 12, December 2003, pp. 1543-1558

[DeM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994

[Gir99] P. Girard, et al.: *A test vector inhibiting technique for low energy BIST design*. IEEE VLSI Test Symposium, May 1999, pp. 407-412.

[Ham98] I. Hamzaoglu and J.H. Patel. *Test Set Compaction Algorithms for Combinational Circuits*, Proceedings of the International Conference on Computer-Aided Design (ICCAD), November 1998.

[Har93] J. Hartmann and G. Kemnitz. *How to Do Weighted Random Testing for BIST*, Proc. of International Conference on Computer-Aided Design (ICCAD), pp. 568-571, 1993

[Hel92] S. Hellebrand, S. Tarnick and J. Rajski. *Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers*, Proc. of International Test Conference, pp. 120-129, 1992

[Hel95] S. Hellebrand, et al. *Built-In Test for Circuits with Scan Based on Reseeding of Multiple-Polynomial Linear Feedback Shift Registers*. IEEE Trans. on Comp., vol. 44, No. 2, February 1995, pp. 223-233

[Hel00] S. Hellebrand, H. Liang and H.J. Wunderlich. *A Mixed Mode BIST Scheme Based on reseeding of Folding Counters*, Proc. IEEE ITC, 2000, pp.778-784

[Koe91] B. Koenemann. *LFSR – Coded Test Patterns for Scan Designs*. Proc. Europian Test Conf., Munich, Germany, 1991, pp. 237-242

[Lee91] H.K. Lee and D.S. Ha. *An Efficient Forward Fault Simulation Algorithm Based on the Paralel Pattern Single Fault Propagation*, Proc. of the 1991 International Test Conference, pp. 946-955, Oct. 1991

[Lee93] H.K. Lee and D.S. Ha. *Atalanta: an Efficient ATPG for Combinational Circuits*. Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993

[McC84] E.J. McCluskey. *Pseudo-Exhaustive Testing for VLSI Devices*, CRC Technical Report No. 84-6, Dept. of Electrical Engineering and Computer Science, Stanford University, USA, August 1984

[McC85] E.J. McCluskey. *BIST techniques*. IEEE Design & Test of Computers, vol. 2 No.2 Apr. 1985, pp.21-28, BIST structures. vol. 2 No.2 Apr. 1985. pp. 29-36

[Nee93] D.J. Neebel and C.R. Kime. *Inhomogeneous Cellular Automata for Weighted Random Pattern Generation*, Proc. of International Test Conference, pp. 1013-1022, 1993

[Nov98] O. Novák and J. Hlavička. *Design of a Cellular Automaton for Efficient Test Pattern Generation*. Proc. IEEE ETW 1998, Barcelona, Spain, pp. 30-31

[Nov99] O. Novák. *Weighted Random Patterns for BIST Generated in Cellular Automata*, Proc. of 5-th IOLTW, Rhodes, Greece, July 1999, pp. 72-76

[Pom93] I. Pomeranz and S.M. Reddy. *3-Weight Pseudo-Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits*, IEEE Transactions on Computer-Aided Design, Vol. 12, No. 7, pp. 1050-1058, July 1993

[Str02] E.C. Stroud. *A Designer's Guide to Built-In Self-Test*, Boston, MA, Kluwer Academic Publishers, 2002

[Tou95] N.A. Touba. *Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST*, Proc. of International Test Conference, pp. 674-682, 1995

[Tou96a] N.A. Touba and E.J. McCluskey. *Synthesis Techniques for Pseudo-Random Built-In Self-Test*, Technical Report, (CSL TR # 96-704), Departments of Electrical Engineering and Computer Science Stanford University, August 1996

[Tou96b] N.A. Touba and E.J. McCluskey. *Altering a Pseudo-Random Bit Sequence for Scan-Based BIST*, Proc. of International Test Conference, 1996, pp. 167-175

[Tou01] N.A. Touba and E.J. McCluskey. *Bit-Fixing in Pseudorandom Sequences for Scan BIST*, IEEE Transactions on CAD, Vol. 20, No. 4, April 2001, pp. 545-555

[Wun87] H.J. Wunderlich. *Self-Test Using Unequiprobable Random Patterns*, Proc. of FTCS-17, pp. 258-263, 1987

[Wun88] H.J. Wunderlich. *Multiple Distributions for Biased Random Test Patterns*, Proc. of International Test Conference, pp. 236-244, 1988.

[Wun96] H.J. Wunderlich and G. Keifer. *Bit-Flipping BIST*, Proc. ACM/IEEE International Conference on CAD-96 (ICCAD96), San Jose, California, November 1996, pp. 337-343

Dissertation Thesis

Title: Mixed-Mode BIST Based on Column Matching

Abstract

Dissertation Thesis will focus on a design of built-in self-test circuitry for combinational or scan-based circuits. The method will be based on our novel method – the Column Matching. The mixed-mode BIST will be supported, while the test will be divided into two disjoint phases – the pseudo-random and deterministic. In the deterministic phase the test vectors are generated by a purely combinational block – the Output Decoder. When designing the Decoder we try to match as many of its outputs with the inputs as possible, which yields no logic needed to implement them.

Better incorporation of the ATPG tool into the algorithm will be studied, to improve the quality of the result. Partitioning of large tested circuits will be considered as well, to reduce the BIST design time and even the resulting BIST area overhead.

The methodology will be verified on standard benchmarks (ISCAS, ITC).

Keywords: built-in self-test, test-per-clock, pseudo-random testing, deterministic BIST

Publications of the Author

[Fis00] P. Fišer and J. Hlavička. *Efficient minimization method for incompletely defined Boolean functions*. Proc. 4th Int. Workshop on Boolean Problems, Freiberg (Germany) 21.-22.9.2000, pp.91-98

[Fis01a] P. Fišer and J. Hlavička. *Implicant Expansion Method used in the BOOM Minimizer*. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'01), Gyor (Hungary), 18.-20.4.2001, pp. 291-298

[Fis01b] P. Fišer and J. Hlavička. *On the Use of Mutations in Boolean Minimization*. Proc. Euromicro Symposium on Digital Systems Design, Warsaw (Poland) 4.-6.9.2001, pp. 300-305

[Fis01c] P. Fišer and J. Hlavička. *BOOM - a Boolean Minimizer*. Research Report DC-2001-05, Prague, CTU Publishing House, June 2001, 37 pp.

[Fis02a] P. Fišer and J. Hlavička. *Column-Matching Based BIST Design Method*. Proc. 7th IEEE Europian Test Workshop (ETW'02), Corfu (Greece), 26.-29.5.2002, pp. 15-16

[Fis02b] P. Fišer and J. Hlavička. *A Set of Logic Design Benchmarks*. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'02), Brno (Czech Rep.), 17.-19.4.2002, pp. 324-327

[Fis02c] P. Fišer and J. Hlavička. *A Flexible Minimization and Partitioning Method*. Proc. 5th Int. Workshop on Boolean Problems, Freiberg (Germany) 19.-20.9.2002, pp. 83-90

[Fis03a] P. Fišer, J. Hlavička and H. Kubátová. *Column-Matching BIST Exploiting Test Don't-Cares*. Proc. 8th IEEE Europian Test Workshop (ETW'03), Maastricht (The Netherlands), 25.-28.5.2003, pp. 215-216

[Fis03b] P. Fišer and J. Hlavička. *BOOM - A Heuristic Boolean Minimizer*, Computers and Informatics, Vol. 22, 2003, No. 1, pp. 19-51

[Fis03c] P. Fišer and J. Hlavička. *A Flexible Minimization and Partitioning Method*, Proc. of Workshop 2003 (web). Prague : CTU, 2003, vol. A, p. 312-313. ISBN 80-01-02708-2

[Fis03d] P. Fišer, J. Hlavička and H. Kubátová. *Coverage-Directed Assignment Approach to BIST*, Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'03), Poznan (Poland), 14.-16.4.2003, pp. 87-92

[Fis03e] P. Fišer, J. Hlavička and H. Kubátová. *FC-Min: A Fast Multi-Output Boolean Minimizer*, Proc. 29th Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 1.-6.9.2003, pp. 451-454

[Fis04a] P. Fišer and H. Kubátová. *An Efficient Mixed-Mode BIST Technique*, Proc. 7th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop 2004, Tatranská Lomnica, SK, 18.-21.4.2004, pp. 227-230

[Fis04b] P. Fišer and H. Kubátová. *Pseudorandom Testability - Study of the Effect of the Generator Type*, Proc. 6th International Scientific Conference on Electronic Computers and Informatics 2004 (ECI'04), Herľany, SR, 22.-24.9.04

[Fis04c] P. Fišer and H. Kubátová. *Influence of the Test Lengths on Area Overhead in Mixed-Mode BIST*, Proc. 9th Biennial Conference on Electronics and Microsystem Technology 2004 (BEC'04), Tallinn (Estonia), 3.-6.10.2004

[Fis04d] P. Fišer and H. Kubátová. *Survey of the Algorithms in the Column-Matching BIST Method*, Proc. 10th International On-Line Testing Symposium 2004 (IOLTS'04), Madeira, Portugal, 12.-14.7.2004, pp. 181

[Fis04e] P. Fišer and H. Kubátová. *Two-Level Boolean Minimizer BOOM-II*, Proc. 6th Int. Workshop on Boolean Problems, Freiberg, (Germany), 23.-24.9.2004

[Fis04f] P. Fišer and H. Kubátová. *Single-Level Partitioning Support in BOOM-II*, Proc. 2nd Descrete-Event System Design 2004 (DESDes'04), Dychów, Poland, 15.-17.9.04, pp. 149-154

[Fis04g] P. Fišer and H. Kubátová. *Boolean Minimizer FC-Min: Coverage Finding Process*, Proc. 30th Euromicro Symposium on Digital Systems Design (DSD'04), Rennes (FR), 31.8. - 3.9.04, pp. 152-159

[Hla00] J. Hlavička and P. Fišer. *Algorithm for Minimization of Partial Boolean Functions*. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS00) Workshop, Smolenice, (Slovakia) 5.-7.4.2000, ISBN 80-968320-3-4, pp.130-133

[Hla01] J. Hlavička and P. Fišer. *BOOM - a Heuristic Boolean Minimizer*. Proc. International Conference on Computer-Aided Design ICCAD 2001, San Jose, California (USA), 4.-8.11.2001, pp. 439-442

[Hla01a] J. Hlavička and P. Fišer. *A Heuristic method of two-level logic synthesis*. Proc. The 5th World Multiconference on Systemics, Cybernetics and Informatics SCI'2001, Orlando, Florida (USA) 22.-25.7.2001, pp. 283-288, vol. II

[Hla01b] J. Hlavička and P. Fišer. *BOOM - a Heuristic Boolean Minimizer*. Proc. International Conference on Computer-Aided Design ICCAD 2001, San Jose, California (USA), 4.-8.11.2001, pp. 439-442

[Hla02] J. Hlavička and P. Fišer. *Minimization and Partitioning Method Reducing Input Sets*. Proc. 1st International Workshop on Electronic Design, Test & Applications (DELTA 2002), New Zealand, 29.-31.1.2002, pp. 434-436