Czech Technical University in Prague
Faculty of Electrical Engineering

Doctoral Thesis

June, 2007

Petr Fišer

Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science and Engineering

COLUMN-MATCHING BASED MIXED-MODE BIST TECHNIQUE

by

Petr Fišer

A doctoral thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Ph.D. Programme: Electrical Engineering and Information Technology
Branch of study: Information Science and Computer Engineering

June 2007

Thesis Supervisor:
Doc. Ing. Hana Kubátová, CSc.
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

Abstract

A novel test-per-clock built-in self-test (BIST) equipment design method for combinational or full-scan circuits, together with necessary supplementary algorithms, is proposed in this Thesis. This method is mostly based on a design of a combinational block - the Decoder, transforming pseudo-random code words into deterministic test patterns pre-computed by some ATPG tool. The Column-Matching algorithm to design the decoder is proposed. 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 the BIST area overhead.

Since quite a large number of test vectors is often needed to sufficiently test a particular circuit, synthesizing all these vectors deterministically would involve a very large area overhead. Thus, the basic Column-Matching method has been extended to support the mixed-mode testing. Here the BIST execution is divided into two disjoint phases – the pseudo-random phase, where the pseudo-random patterns are being applied to the circuit unmodified, and the deterministic phase detecting all the yet undetected faults. This enables us to reach high fault coverage in a short test time and with a low area overhead. The novelty of this approach comprises of the fact that these phases are disjoint. As a consequence of this, the BIST control logic is significantly reduced, when compared to other state-of-the-art methods. The choice of the lengths of the two phases directly influences the test time, BIST design time and BIST area overhead. The Column-Matching algorithm is described in details in the Thesis and several heuristic methods solving some of the major NP-hard problems involved are proposed. 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.

A truth table description of a Boolean function is obtained as a result of the Column-Matching algorithm. This function describes the Output Decoder logic. In order to maximally reduce the BIST area overhead, the function has to be minimized. Since it is usually a function of many input and output variables (hundreds, thousands), available Boolean minimizers were not able to handle it in a reasonable time. Thus, an efficient Boolean minimizer has been developed for this reason. BOOM, as a minimizer capable to process functions having many input variables, is proposed. The implicants of a function are generated by a top-down way: the universal hypercube is being gradually reduced, until it becomes an implicant. This approach becomes very advantageous for sparse functions, i.e., functions where values of only few minterms are defined. This is exactly the case of the Column-Matching BIST design. The efficiency of Boom for such functions is documented on standard benchmark circuit, as well as on practical Column-Matching examples.

The proposed BIST design method was tested on ISCAS and ITC'99 benchmarks and the results were compared with the results obtained by some of the state-of-the-art methods. The complete resulting BIST equipment logic was synthesized and its complexity evaluated. A complete (100%) stuck-at fault coverage was considered in all the experiments performed.

The main contributions of this Thesis are the following:

- *A new BIST design methodology is proposed*
- *The BIST process is divided into two separate phases, unlike in other methods*
- *A new and very efficient Boolean minimizer is proposed*

Keywords

Built-in self-test, test pattern generation, pseudo-random testing, mixed-mode BIST, logic functions, implicant generation, Boolean minimization.

Acknowledgement

First of all, I would like to express my gratitude to my former supervisor, Prof. Jan Hlavička, who has, to a grief of all who has known him, deceased in September, 2002. He has been my great guidance in my research and a big source of motivation. He has introduced me into the scientific life and showed me the way how to be acquainted in this area. I really appreciate the friendship that has been between us.

Then, I would like to give many thanks to Hana Kubátová, my Thesis supervisor. She has been a constant source of encouragement and insight during my research. She provided me with numerous opportunities for professional advancements. Her efforts as the Thesis supervisor contributed substantially to the quality and completeness of the thesis.

The staff of our department has provided me a pleasant and flexible environment for my research. My work has been partially supported by grants from GACR grant agencies.

Last but not least, I would like to thank to my family and to my friends for their everlasting support and patience during my research.

Table of Contents

1 Introduction 1
1.1 Basic Principles of Built-In Self-Test 1
1.2 Structure of the Thesis 5
2 State-of-the-Art Methods 6
2.1 Exhaustive Testing 6
2.2 Pseudo-Random Testing 6
2.3 Reseeding-Based Techniques 7
2.4 Weighted Pattern BIST 7
2.5 Bit-Fixing and Bit-Flipping 8
2.6 Row Matching 9
3 Aims of the Dissertation and Its Contributions 11
4 Proposed BIST Method 12
4.1 Basic Principles of the Proposed Mixed-Mode BIST 13
4.2 The BIST Design Process 14
4.3 The Pseudo-Random Phase 15
4.3.1 Influence of the Pseudo-Random Phase Length 17
4.4 Influence of the LFSR Structure and Seed 18
4.5 The Deterministic Phase 19
4.5.1 Problem Statement 19
4.5.2 The Column-Matching Algorithm 20
4.5.3 One-to-One Assignment 21
4.5.4 Generalized Column-Matching 22
4.5.5 Column Matching Process Example 23
One-to-One Assignment for c17 Benchmark 24
Generalized Column-Matching Example 24
4.6 Column-Matching Exploiting Test Don't Cares 25
4.6.1 Row Assignment Algorithms 26
4.6.2 Column-Matching Algorithms 28
4.6.3 The Basic Fast Search Algorithm 30
4.6.4 Overview of the Column-Matching Alternatives in Mixed-Mode BIST 31
4.7 Multiple-Vector Column-Matching 32
4.7.1 ATPG Modes 32
4.7.2 Simple Test Set Compaction 33
4.7.3 Multiple-Vector Column-Matching Principles 35
4.7.4 Multiple-Vector Test Set Compaction 35
Example 35
4.7.5 Modified Row Assignment 36
4.7.6 Modified BIST Design Process 36
4.8 Influence of the Length of the Deterministic Phase 37
4.9 Summary Discussion on the Lengths of the Two Phases 37
4.10 Comparison with Other State-of-the-Art Methods 38
4.11 Column-Matching Results for Standard Benchmarks 39
5 BOOM – The Boolean Minimizer 44
5.1 Motivation 44
5.2 Introduction 44
5.3 Problem Statement 45
5.4 BOOM Structure 46
5.5 Coverage-Directed Search 47
5.5.1 Basis of the Method 47
5.5.2 Immediate Implicant Checking 48
5.5.3 CD-Search Example 49
5.6 Iterative Minimization 50
5.6.1 The Effect of the Iterative Approach 50
5.6.2 Accelerating the Iterative Minimization 51
5.7 Implicant Expansion 52
5.7.1 Checking the Removal of a Literal 53
5.7.2 Expansion Strategy 53

5.7.3 Evaluation of Expansion Strategies 54
5.8 Minimizing Multi-Output Functions 55
5.9 Implicant Reduction (IR) 56
5.10 Solution of the Covering Problem 56
5.11 The BOOM Algorithm 57
5.12 BOOM Experimental Results 58
5.12.1 Standard MCNC Benchmarks 58
5.12.2 Test Problems Having $n \geq 50$ 59
5.12.3 Solution of Very Large Problems 61
5.12.4 Column-Matching Practical Examples 61
5.13 Time Complexity Evaluation 63
5.13.1 Influence of the Problem Size 63
5.13.2 Influence of Don't Cares 65
5.14 BOOM Conclusions 67
6 Implementation 68
6.1 Implementation of Column-Matching BIST Equipment Design Method 68
6.2 Implementation of BOOM 71
7 Conclusions and Future Work 72
7.1 Future Work 73

List of Figures

Figure 1.1: BIST structure 1
Figure 1.2: LFSR structure 2
Figure 1.3: Example of a cellular automaton 3
Figure 1.4: Test-per-clock BIST structure 3
Figure 1.5: Test-per-scan BIST structure 4
Figure 1.6: Multiple scan chain BIST structure 4
Figure 1.7: STUMPS structure 4
Figure 2.1: Multi-polynomial BIST 7
Figure 2.2: Modifying the LFSR patterns 8
Figure 2.3: Bit-fixing scheme 9
Figure 2.4: Bit-flipping scheme 9
Figure 2.5: Row matching principle 10
Figure 2.6: The resulting truth table 10
Figure 4.1: Test-per-clock BIST structure 12
Figure 4.2: Mixed-mode BIST structure 13
Figure 4.3: Test sequence generation 15
Figure 4.4: Resulting BIST circuitry 15
Figure 4.5: Pseudo-random fault coverage 16
Figure 4.6: Fault coverage saturation curve 16
Figure 4.7: Assignment of the rows 20
Figure 4.8: Column matching example 21
Figure 4.9: The first assignment to the submatrices 21
Figure 4.10: ISCAS c17 test vectors 23
Figure 4.11: One-to-one exact Column-Matching example 24
Figure 4.12: BIST implementation for c17 circuit 24
Figure 4.13: Assignment of rows for c17 circuit 25
Figure 4.14: Row assignment histograms 27
Figure 4.15: Close-up view of Fig. 4.14 28
Figure 4.16: Thorough search progress 29
Figure 4.17: Repetitive fast search 30
Figure 5.1: Structure of the BOOM system 46
Figure 5.2: Growth of PI number and decrease of SOP length during iterative minimization 51
Figure 5.3: Iterative minimization schematic plan 52
Figure 5.4: I-buffer tree structure 52
Figure 5.5: Growth of time for different IE methods 54
Figure 5.6: Growth of PIs for different IE methods 55
Figure 5.7: Growth and fall of the number of non-primes 56
Figure 5.8: Time complexity (1) 64
Figure 5.9: Time complexity (2) 64
Figure 5.10: Time complexity (3) 65
Figure 5.11: Runtimes for ESPRESSO (dashed lines) and for BOOM (solid line) 66
Figure 5.12: Relative slowdown [%] for various percentages of DCs 66
Figure 6.1: BIST equipment synthesis dataflow 69
Figure 6.2: BIST structure 70

List of Tables

Table 4.1: Influence of the pseudo-random phase on the result 17
Table 4.2: Influence of the LFSR seed 18
Table 4.3: Row assignment algorithms 28
Table 4.4: Test compaction results 34
Table 4.5: Influence of the deterministic phase length on the result 37
Table 4.6: Influence of the test lengths 38
Table 4.7: Comparison results 39
Table 4.8: ISCAS & ITC benchmarks 41
Table 5.1: Immediate implicant checking effects 48
Table 5.2: CD-Search Example (1) 49
Table 5.3: CD-Search Example (2) 49
Table 5.4: CD-Search Example (3) 49
Table 5.5: CD-Search Example (4) 50
Table 5.6: Runtimes and minimum solutions for the standard MCNC benchmarks 59
Table 5.7: Solution of Boom Benchmarks - comparing the result quality 60
Table 5.8: Solution of Boom Benchmarks - comparing the runtime 61
Table 5.9: Time for one iteration on very large problems 61
Table 5.10: Output Decoder design examples 62

List of Algorithms

Algorithm 4.1: Set System Based Column-Matching 23
Algorithm 4.2: Fast Search Column-Matching 31
Algorithm 4.3: Multiple-vector row assignment 36
Algorithm 5.1: The CD-Search 48
Algorithm 5.2: Minimization of a group of functions 58

List of Abbreviations

ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generator
BIST	Built-in Self-Test
BISTE	Built-in Self-Test Equipment
CA	Cellular Automaton
CD-Search	Coverage-Directed Search
CM	Column-Matching
CP	Covering Problem
CUT	Circuit under Test
DC	Don’t Care
FSM	Finite State Machine
GE	Gate Equivalent
GLFSR	Generalized Linear Feedback Shift Register
GURT	Generator of Unequiprobable Random Tests
IE	Implicant Expansion
IR	Implicant Reduction
LFSR	Linear Feedback Shift Register
MISR	Multiple-Input Shift Register
MP-LFSR	Multi-Polynomial Linear Feedback Shift Register
PI	Prime Implicant
PRPG	Pseudo-Random Pattern Generator
RE	Response Evaluator
TPG	Test Pattern Generator

List of Symbols

A ..... *Covering matrix*
C ..... *PRPG code words matrix*
Det ..... *the length of the deterministic phase*
D_i ..... *Don't* care set
e ..... *Number of rows of* ***A***
f ..... *Number of columns of* ***A***
F_i ..... *Single-output Boolean function*
F_i ..... *On*-set
i ..... *a particular* ***C*** *matrix column (to be matched); general index*
j ..... *a particular* ***T*** *matrix column (to be matched); general index*
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; number of input variables*
p ..... *number of PRPG cycles; number of* ***C*** *matrix rows*
PR ..... *the length of the pseudo-random phase*
R_i ..... *Off*-set
r ..... *number of CUT inputs; number of* ***T*** *matrix columns*
s ..... *number of deterministic test vectors; number of* ***T*** *matrix rows*
T ..... *Test vectors matrix*
v ..... *number of output variables*
x_i ..... *input variable (LFSR output, Decoder input)*
y_j ..... *output variable (Decoder output, CUT input)*

1 Introduction

With the ever-increasing complexity of present VLSI circuits, their testing is becoming more and more important. There often arise faulty chips during the manufacturing process due to an inaccurate technology and such chips should be detected and eliminated. Using only external test equipment (ATE) to test the chips is becoming impossible, mainly due to a huge amount of test vectors, long test time and very expensive test equipment. Incorporating the Built-in Self-Test Equipment (BISTE) becomes inevitable. It requires no external tester to test the circuit, since all the circuitry needed to conduct the test is included in the very circuit. This is paid by an area overhead, long test time and often low fault coverage. Up to now, many BIST methods have been developed [Aga93, Tou96a, Tou96b], all of them trying to find some trade-off between these four aspects that are mutually antipodal:

- Fault coverage, i.e., the percentage of detected faults, in a chosen fault model
- Test time
- BIST area overhead
- BIST design time

To reach a high fault coverage, either a long test time (exhaustive test), or a high area overhead (ROM-based BIST) is involved. A pseudo-random testing established the simplest trade-off between all these 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. To improve the fault coverage and to reduce the test time, many enhancements of this pseudo-random principle have been 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.

1.1 Basic Principles of Built-In Self-Test

The general Built-in Self-Test structure consists of three main parts [McC85] – see Figure 1.1. 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).

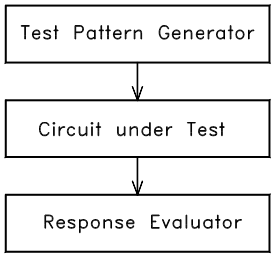

Figure 1.1: BIST structure

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

There are two basic testing strategies: *functional testing* and *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 are several kinds of faults caused by various physical defects, like 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 applied to the input (of their total number 2^n, where n is the number of the primary inputs of the circuit). However, there are faults that are hard to detect (*random pattern resistant faults, hard faults*), as only few test vectors 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.

In most of cases, some kinds of pseudo-random pattern generators (PRPGs) are used as test pattern generators (TPGs), either stand-alone or modified somehow. Generally, PRPGs are simple sequential circuits generating code words, according to the generating polynomial [Str02]. These code words are then either fed directly to the CUT inputs, or they are modified by some additional circuitry.

The most common PRPG structures are *linear feedback shift registers* (LFSRs) or *cellular automata* (CA). An *n*-bit (*n*-stage) LFSR is a linear sequential circuit consisting of D flip-flops and XOR gates generating code words (patterns) of a cyclic code. The structure of an *n*-stage LFSR-I (with internal XORs) is shown in Fig. 1.2.

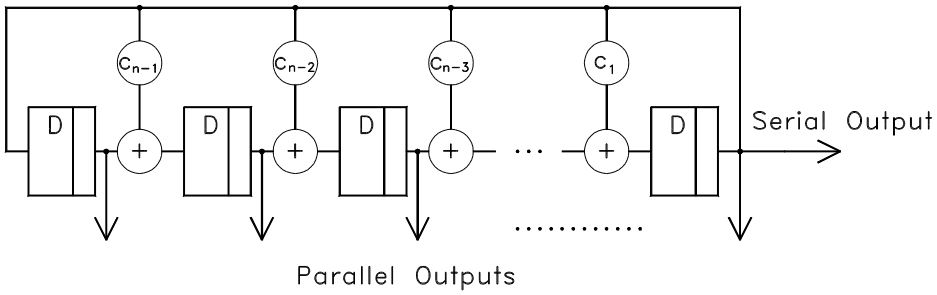

Figure 1.2: LFSR structure

The register has n parallel outputs corresponding to the outputs of the D flip-flops, and one flip-flop output can be used as a serial output of a register.

The coefficients $c_1 - c_{n-1}$ express whether there exists (1) a connection from the feedback to the corresponding XOR gate or no connection (0). Thus it determines whether there is a respective XOR gate present or the flip-flops are connected directly. The feedbacks leading to the XOR gates are also called *taps*.

The sequence of code words produced by an LFSR can be described by a *generating polynomial* g(x) in GF(2^n), [Ada91].

$$g(x) = x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + ... + c_1x^1 + 1$$

If the generating polynomial is primitive, the LFSR has a maximum period 2^n-1, thus it produces 2^n-1 different patterns.

The initial state of the register (initial values of the flip-flops) is called the seed.

The second LFSR type, the LFSR-II is implemented with XORs in the feedback. Its generating polynomial is dual to the LFSR-I polynomial. Only LFSR-I will be considered in this Thesis, since these two LFSRs types are mutually convertible.

Cellular automata [Hor90, Alo93] are sequential structures similar to LFSRs. Their periods are often shorter, but code words generated by CA are sometimes more suitable for test patterns with preferred numbers of ones or zeros at the outputs.

An example of a CA performing multiplication of the polynomials corresponding to code words by the polynomial x+1 (rule 60 for each cell [Cha97]) is shown in Fig. 1.3.

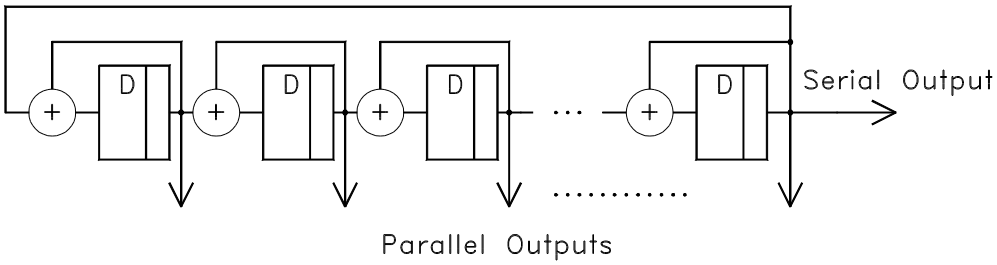

Figure 1.3: Example of a cellular automaton

Since the TPG can be constructed to have both parallel and/or serial outputs, the BIST can be designed in two general ways: the test-per-clock and test-per-scan BIST. In the *test-per-clock* BIST the CUT is being fed by parallel outputs of the TPG, which is mostly the linear feedback shift register (LFSR) or a cellular automaton (CA). 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 Multi-Input Shift Register (MISR). A general structure of the test-per-clock BIST is shown in Fig. 1.4.

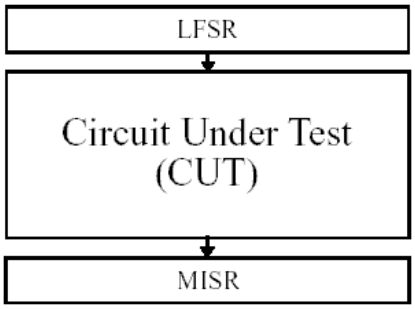

Figure 1.4: Test-per-clock BIST structure

The second typical structure, suitable especially to test sequential circuits, is denoted as a *test-per-scan* BIST (Fig. 1.5). 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 only the test-per-clock is considered, however the method can be adapted to test-per-scan as well [Cha03].

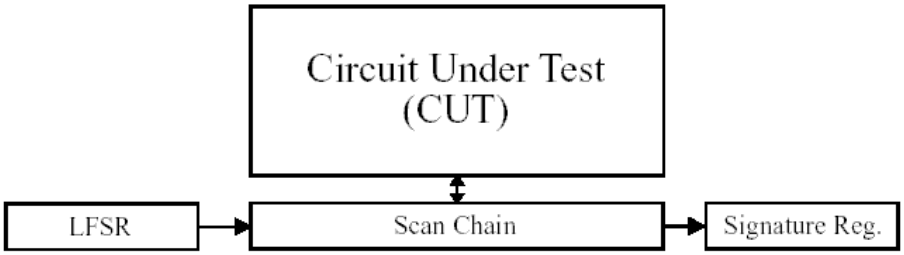

Figure 1.5: Test-per-scan BIST structure

The two above-mentioned methods suffer from their specific drawbacks: in the test-per-clock BIST case there is often a very "wide" PRPG register and heavy wiring. On the other hand, the test application time is the shortest possible. When the test is applied using the test-per-scan method, there is a big test time overhead, since each of the test patterns has to be serially loaded into the scan chain. Moreover, the circuit often cannot be tested at its work frequency, since the flipping activity on the signals exceeds the activity occurring during its normal operation. This would cause the tested chip dissipate more heat, which could "burn" the chip during the test.

The present trend in the BIST design area is a combination of these two approaches: the *multiple scan chain BIST* [Kei98]. Here the CUT registers form more than one scan chains, which are fed in parallel, see Fig. 1.6.

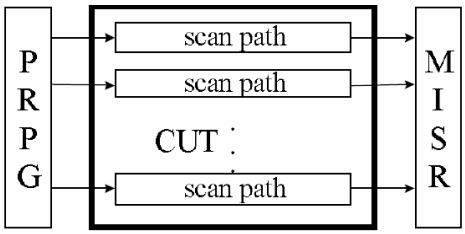

Figure 1.6: Multiple scan chain BIST structure

The state-of-the-art serially-parallel BIST trend is the STUMPS (Self-Test Using MISR and Parallel Shift-Register Sequence Generators) architecture [Bar87], where the PRPG code words are modified by so called "phase shifter" (PS) and then fed in parallel into multiple scan chains. The output response is evaluated in MISR. See Fig. 1.7.

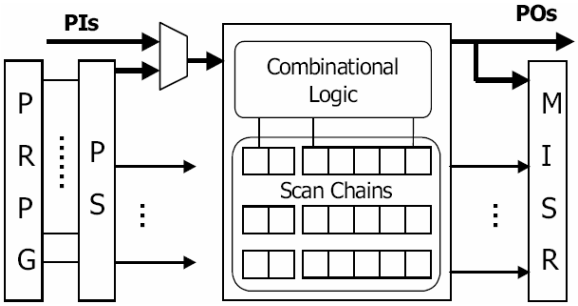

Figure 1.7: STUMPS structure

1.2 Structure of the Thesis

After a brief introduction to the BIST in Section 1, a survey of state-of-the-art BIST design methods follows (Section 2). Section 3 describes the aims of the Dissertation. The proposed BIST design method is presented in Section 4. Section 5 describes BOOM, the Boolean minimizer which has been developed as a necessary part of the proposed BIST design process. The Implementation section (Section 6) describes the engineer work done, thus the programs developed in the process. Section 7 concludes the Dissertation and proposes several ways of future research.

2 State-of-the-Art Methods

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, the basic BIST methods will be introduced, for better understanding to the latter ones.

2.1 Exhaustive Testing

There are several testing approaches differing in their successfulness, in terms of the BIST area overhead, design time and test duration. In the most naive method – the *exhaustive testing* – the circuit is fed with all the possible 2^n (where n is the number of CUT inputs) patterns and the responses is checked. Obviously, for a combinational circuit the exhaustive test provides complete fault coverage, and it can be very easily implemented (an area overhead is often the lowest possible), but it is extremely time consuming and thus inefficient and practically unusable. It is applicable to circuits with up to 30 inputs (10^9 patterns, which takes cca. 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 comprise of 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.

2.2 Pseudo-Random Testing

In a simple *pseudo-random testing* the test patterns are generated by a pseudo-random pattern generator (PRPG) and led directly to the circuit's inputs. It differs from the exhaustive testing by a test length. If the PRPG structure and seed are properly chosen, only several test patterns (less than 2^n) are needed to completely test the circuit. The pseudo-random testing is also widely used in a case where the complete fault coverage is not required, since pseudo-random patterns often successfully detect most of the easy-to-detect faults. Linear feedback shift registers (LFSRs) or cellular automata (CA) are usually used as PRPGs. As an improvement of an LFSR, a generalized LFSR (GLFSR) has been proposed [Pra99], however it involves an increase of the area overhead.

A combination of a pseudo-random and deterministic BIST is being referred to as a *mixed-mode BIST*. The easy-to-detect faults are tested by pseudo-random test patterns, and the deterministic patterns are generated to test the remaining, undetected faults. The popular bit-fixing [Tou95, Tou96a, Tou01] and bit-flipping [Wun96] techniques belong to this category.

2.3 Reseeding-Based Techniques

In the basic *reseeding* technique, the LFSR is seeded with more than one seeds during the test, while the seeds need to be stored in ROM [Koe91]. The seeds are sometimes 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 using a *multi-polynomial LFSR* (MP-LFSR), where the feedback network of an LFSR is reconfigurable [Hel92, Hel95]. 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. 2.1.

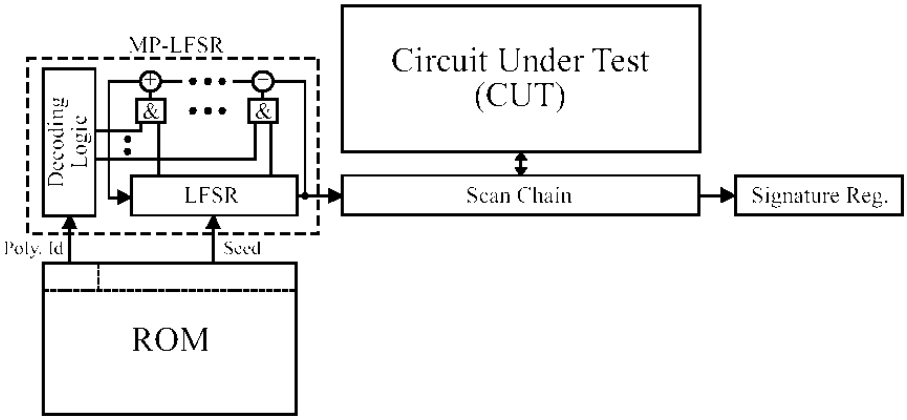

Figure 2.1: Multi-polynomial BIST

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.

2.4 Weighted Pattern BIST

To one of the approaches, where the pseudo-random patterns are modified so that better fault coverage is reached, belongs the *weighted pattern testing.* Here the PRPG patterns are being biased by a *signal probability* of some of the PRPG outputs (the

probability of a “1” value occurrence). In the weighted pattern testing method two problems have to be solved: first, the weight sets have to be computed and then the weighted signals have to be generated. Many weight set computation methods were proposed [Bar87] and it was shown that multiple weight sets are needed to produce patterns with sufficient fault coverage [Wun88]. These multiple weight sets have to be stored on a chip and also the logic accomplishing 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 BIST were proposed - one of them is a *Generator of Unequiprobable Random Tests* (GURT) presented in [Wun87]. The area overhead is reduced, however it is restricted to one weight set only. 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 usually have very balanced weights, 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].

In [Wan01] the weighted random BIST technique is accompanied by a special ATPG producing suitable test vectors. Such a combination yields a very low BIST area overhead.

2.5 Bit-Fixing and Bit-Flipping

The *bit-fixing* [Tou95, Tou96a, Tou01] and *bit-flipping* [Wun96] methods are based on a modification of some LFSR 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 an example in Fig. 2.2.

This idea was generalized in [Tou96b], where the problem of finding a mapping function is transformed into finding a minimum rectangle cover of 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 2.3 and 2.4 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.

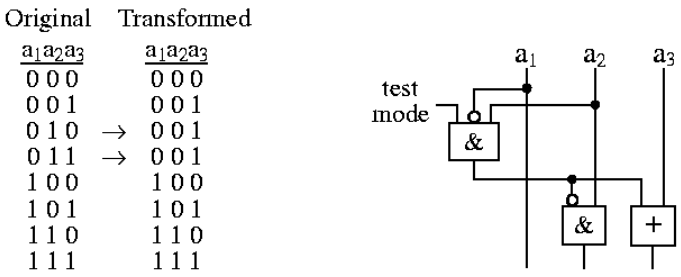

Figure 2.2: Modifying the LFSR patterns

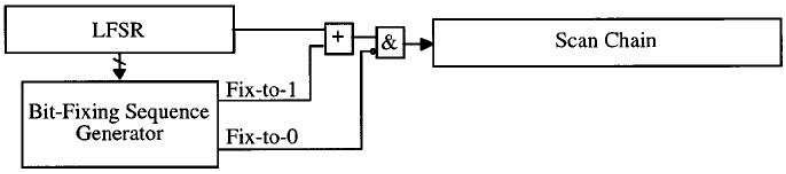

Figure 2.3: Bit-fixing scheme

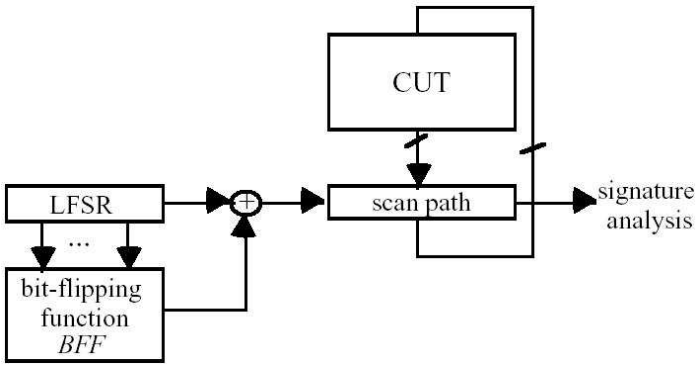

Figure 2.4: Bit-flipping scheme

2.6 Row Matching

The *row matching* approach proposed in [Cha95, Cha03] is based on a similar idea. A simple combinational function transforming some of the PRPG patterns into test patterns is 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 – 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 comprises of finding an assignment of deterministic test patterns to the PRPG code words, as it is shown in Fig. 2.5. 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.

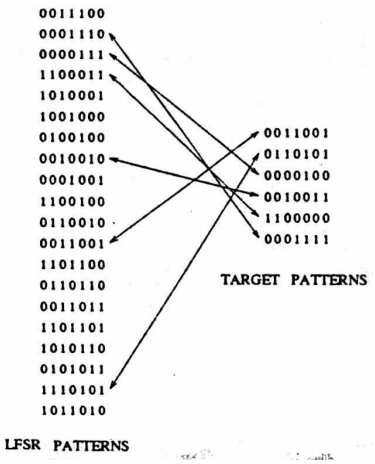

Figure 2.5: Row matching principle

The aim of the algorithm is to find a row matching that minimizes the cost function, which is a rough measure of the complexity of the final BIST design [Cha95]. This is, unfortunately, an NP-hard problem and thus some heuristic must be used. In the proposed algorithm 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. 2.5 is shown in Figure 2.6:

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 2.6: The resulting 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 [Pra99]. However, this modification introduces an additional logic to the whole BIST structure, and thus it disturbs otherwise good results.

3 Aims of the Dissertation and Its Contributions

The main aim of my research, described in this Dissertation Thesis, was to develop a new BIST design method, as a better alternative to other state-of-the-art techniques, like bit-flipping, bit-fixing, etc. The designed method was primarily targeted to the test-per-clock BIST for combinational or full-scan circuits (or circuits provided by a slight modification of a full-scan, respectively). Competitive methods that are commonly used in industrial applications are used for a comparison in this Thesis.

A big importance should be given to the scalability of the algorithm. There are four antipodal aspects in the BIST equipment design:

- BISTE design time
- BISTE area overhead
- Duration of the BIST (number of clock cycles needed to test the circuit)
- Fault coverage

Different ASIC designers integrating BIST equipment into their circuits have different requirements. Sometimes there is a requirement to design the BIST equipment as fast as possible, regardless the area overhead and the fault coverage (to some extent, of course). For low-power designs, the BISTE area overhead should be kept as small as possible, while the BIST design time is not that important. Or, and this is the most common case in practice, high fault coverage is important, whereas the BIST design time plays a small role. This is underlain by a fact that the design time of the tested circuit is mostly significantly higher than the BIST equipment design time.

Thus, the aim of the Dissertation is to propose a flexible way how to design test pattern generators (TPGs) meeting *any* of the above-mentioned restrictions (or, better, quality measures). The designer should be able to freely adjust the BIST equipment design runtime, BISTE area overhead and BIST run time, according his preferences.

The next contribution is the way how the mixed-mode TPG is designed: it contains no additional space-demanding memory elements (except of the PRPG flip-flops). Methods described in Section 2 (bit-fixing, bit-flipping) require additional flip-flops to store signatures, in order to be able to recognize pseudo-random patterns that are to be transformed into deterministic ones. On the other hand, the Mixed-Mode Column-Matching method strictly divides the pseudo-random and deterministic phases. Their switching requires minimum of logics, namely only one additional pattern counter stage, in the optimum case.

The proposed algorithm should serve as a basic guideline how to design more complex BIST designs, i.e., the multiple-scan chain based BIST, the STUMPS architecture, etc. The method should be as general, like the other state-of-the-art methods are (e.g., bit-flipping, bit-fixing).

BOOM, the two-level Boolean minimizer has been developed as a part of this work. It is a very efficient minimization tool able to efficiently handle functions having up to thousands input variables. It offers a big scalability too – the result quality may be improved by iterating the minimization process, thus for a cost of a longer runtime.

4 Proposed BIST Method

A novel test-per-clock BIST design method is proposed in this Thesis. The test patterns are applied to the primary inputs of the circuit-under-test (CUT) in parallel, thus one test vector is being processed in one clock cycle. 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 decreases the BIST area overhead by simplification of the test pattern generator (TPG). Deterministic test patterns generated by some ATPG (Automatic Test Pattern Generator) tool are used, thus the fault coverage achieved strictly depends on these patterns. The method is that universal, so that any test vector set can be used. This implies that the method can be adapted to any fault model, as long as basic requirements for the test vectors are held. For example, delay faults cannot be tested using the simple Column-Matching, since test vector pairs have to stay together here. However, after a slight algorithm modification, even this could be possible. On the other hand, e.g., IDDQ testing [Raj00] may be supported without any algorithm modification.

No memory is used for storing test patterns, since the memory mostly causes a big area overhead on a chip. From a global point of view [Str02], the method is based on a synthesis of a finite state machine (FSM) that produces deterministic 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.

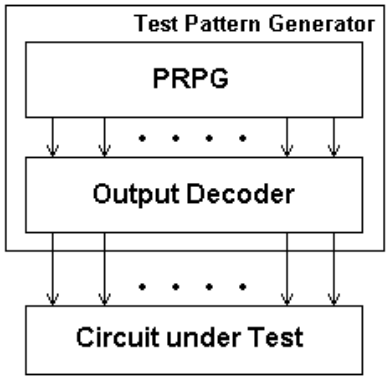

Figure 4.1: Test-per-clock BIST structure

The principles of the proposed mixed-mode BIST method are presented in this section. There are several aspects involved with the BIST design. These will be described in the following Subsections:

- The general mixed-mode BIST principles are shown in Subsection 4.1.
- Then the whole BIST design process is described, for the basic method.

- The BIST process is being executed in two separate phases: the pseudo-random and the deterministic one. These two phases will be discussed more thoroughly in Subsections 4.3, 4.4, 4.5 and 4.8.
- The Column-Matching principle exploiting test don't cares is described in Subsection 4.6.
- Section 4.7 describes the newest algorithm enhancement, namely the Multiple-Vector Column-Matching.
- Experimental results are presented at the end of this Section.

4.1 Basic Principles of the Proposed 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 the proposed mixed-mode BIST design is shown in Fig. 4.2. The pseudo-random code words are produced by an LSFR (or any PRPG in general). 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).

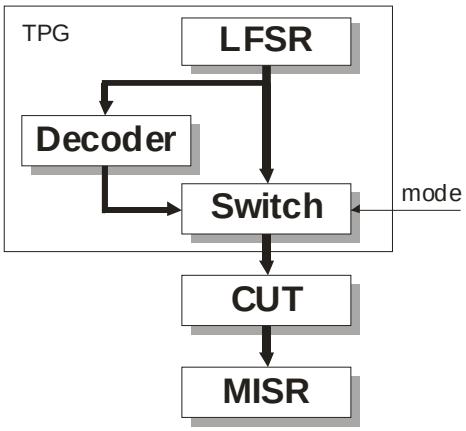

Figure 4.2: Mixed-mode BIST structure

The main difference between the proposed algorithm and competitive methods [Tou95, Tou96b, Wun96, 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. 2.3, or a XOR gate for bit flipping [Wun96] – See Fig. 2.4.

In practice, several initial pseudo-random vectors detect many 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 to switch to the deterministic one at once, as it is being done in the proposed 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 large, since even these multiplexers can be efficiently eliminated by using a Modified Column-Matching method as well (see Subsection 4.6.4). Moreover, the size of a multiplexer, when implemented using transmission gates, is 1.5-times the size of a standard NAND gate

[DeM94]. Moreover, the array of multiplexers has to be present in any test-per-clock BIST design. If another layer of multiplexers is added by the method, the 2-1 MUXes are changed to 3-1 MUXes only (see the Implementation section).

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 (internally). Even in this case the area overhead of this logic can be negligible, since the BIST-controller pattern counter can be exploited very efficiently here. For instance, when the lengths of the two phases are equal, the *mode* signal can be driven by one additional stage (D flip-flop) of a pattern counter only. If not, just extra comparator logic has to be present. Hence, the separation of the two BIST phases completely eliminates the pattern recognition logic.

4.2 The BIST Design Process

The Decoder logic is synthesized using the Column-Matching algorithm. The *Output Decoder* is a combinational block transforming some of the PRPG patterns into deterministic patterns pre-computed by an ATPG. The 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 PRPG 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 by a wired connection, thus without any logic involved. 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 represents non-testing "gaps". This gives a big freedom to select appropriate matches. The values of the non-matched outputs have to be synthesized by some Boolean minimizer, i.e., BOOM [Hla01, Fis03b] or ESPRESSO [Bra84]. The Column-Matching algorithm will be described into detail in Subsection 4.5).

The whole mixed-mode Column-Matching based TPG design process can be summarized as follows:

1. Simulate several (*PR*) pseudo-random patterns for the CUT and determine the undetected faults (by fault simulation)
2. Compute deterministic test patterns detecting these faults by an ATPG tool
3. Perform the Column-Matching using the subsequent LFSR pseudo-random patterns (*Det*) and the deterministic tests
4. Synthesize the unmatched decoder outputs using a two-level Boolean minimizer.

An artificial illustrative example is shown in Fig. 4.3. A 5-bit LFSR is run for 5 cycles first and the easily testable faults are detected. Then the fault simulation was run 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 subsequent 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 outputs there is only the switching logic needed (y_2, y_3). Such cases should be preferred when the BIST is being designed.

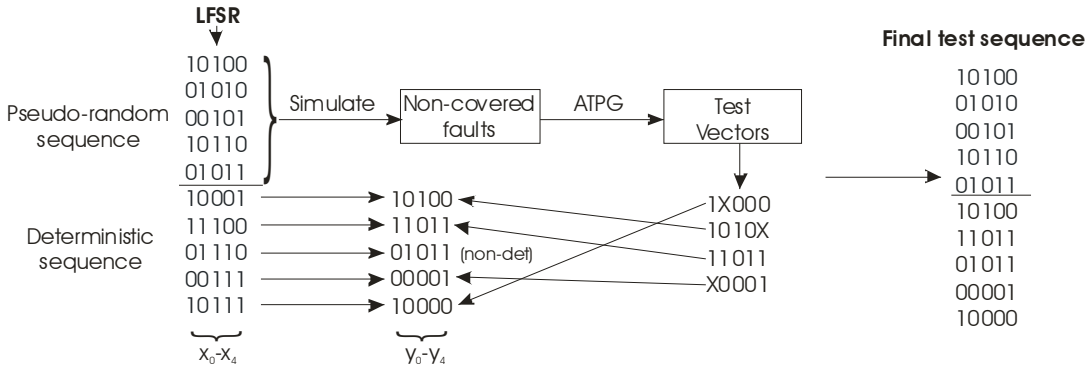

Figure 4.3: Test sequence generation

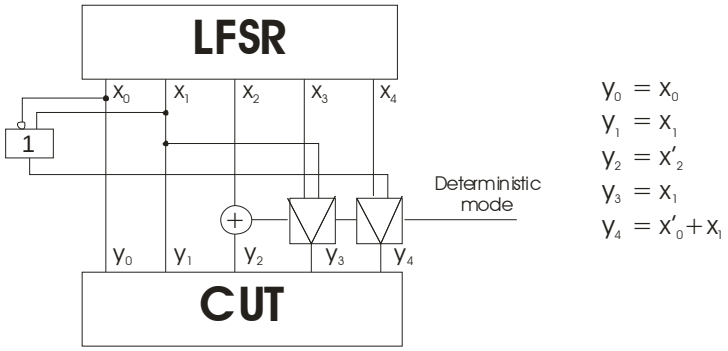

Figure 4.4: Resulting BIST circuitry

4.3 The Pseudo-Random Phase

The aim of the pseudo-random phase is to detect 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 the LFSR polynomial and seed in order to achieve good fault coverage is an extremely computationally demanding problem, thus the seed is selected at random and its effectiveness evaluated experimentally.

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. 4.5). Each LFSR polynomial and seed were selected randomly. The distribution of the number of faults which remained undetected is shown. We can see that it well 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 polynomial and seed on the fault coverage is negligible.

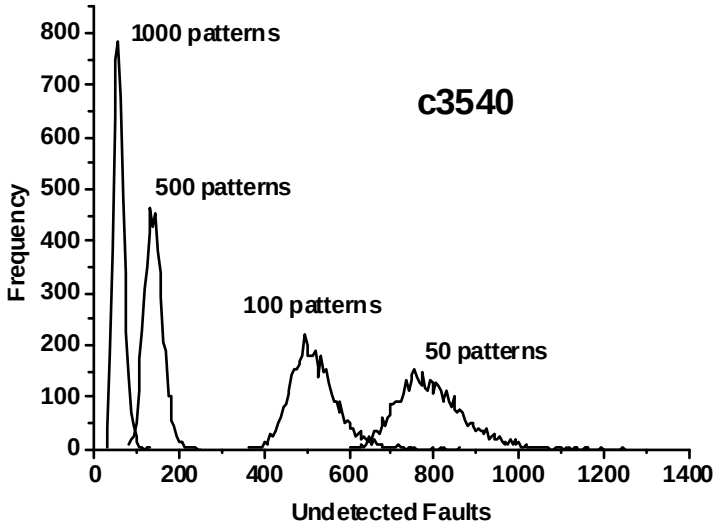

Figure 4.5: Pseudo-random fault coverage

The number of the covered faults as a function of the number of LFSR cycles applied to the CUT follows the 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.

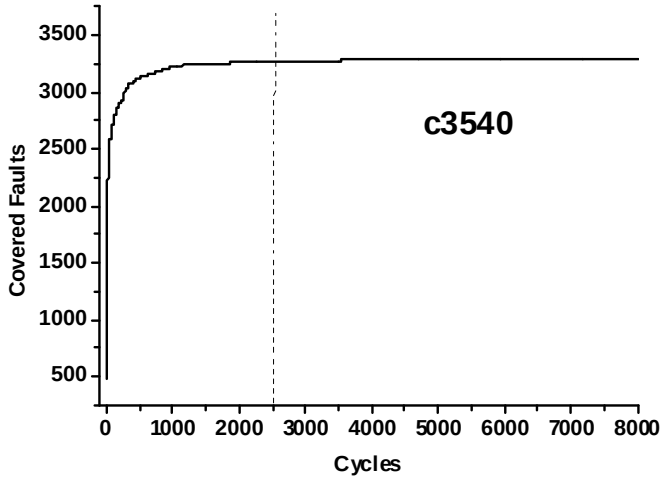

Figure 4.6: Fault coverage saturation curve

A conclusion can be made from these two graphs: in order to reach a satisfactory fault coverage in the pseudo-random phase, the fault coverage saturation curve for the CUT should be determined by a 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 does not improve 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, this threshold is set to 1000 cycles. Thus, for the c3540 benchmark *PR = 2500* cycles is determined (see Fig. 4.6).

4.3.1 Influence of the Pseudo-Random Phase Length

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

The results are shown in Table 4.1. The benchmark name is shown in the first column. 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 by this phase. *"vct."* gives then the number of deterministic vectors testing these faults, produced by an ATPG. The length of the deterministic phase was set constantly to 1000 cycles, except of the s38417 benchmark, where it was set to 2000 cycles (because of the size of the test set). The *"GEs"* column shows the total complexity of the BIST design, in terms of the gate equivalents [DeM94]. The time needed to complete the Column-Matching procedure is indicated in the last column. The experiment was run on a PC with 1 GHz Athlon CPU, Windows XP.

Table 4.1: Influence of the pseudo-random phase on the result

bench	*PR*	*UD*	*vct.*	*GEs*	*Time [s]*
c2670	1 K	309	86	199.5	166
	2 K	306	86	189.5	166
	5 K	216	73	194.5	143
	10 K	154	69	166.5	123
c3540	300	165	66	109.5	10.26
	500	92	42	56.5	3.88
	1 K	36	26	28	1.02
	2 K	9	9	13.5	0.19
	5 K	1	1	1.5	0.02
s1196	200	228	104	110.5	5.05
	500	141	79	77	3.87
	1 K	90	51	50.5	2.00
	2 K	52	37	37	1.20
	5 K	23	17	17	0.48
	10 K	9	4	6	0.04
s5378	5 K	89	49	65.5	2259
	10 K	63	23	31.5	767
	20 K	48	8	16.5	104
s9234.1	1 K	1674	215	883	52 300
	50 K	773	99	333.5	4 400
	200 K	599	52	212.5	1 600
s13207.1	1 K	1793	197	699	208 K

bench	*PR*	*UD*	*vct.*	*GEs*	*Time [s]*
	10 K	617	74	280	3 480
	50 K	182	21	36	128
s38417	10 K	2067	1391	15106.5	3 417
	100 K	780	520	3259.5	2 263

A big trade-off between the test length and the area overhead can be seen here. The longer the pseudo-random phase runs, the less area overhead is reached. Consequently, the BIST synthesis time reduces as well.

It can be concluded from this table that the pseudorandom phase plays a very important role. 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 size of the Decoder logic cannot be influenced by this phase that much.

The influence of the length of the pseudo-random phase on the final result is discussed more thoroughly in [Fis04c].

4.4 Influence of the LFSR Structure and Seed

The fault coverage reached in the first phase is not influenced by the length of the pseudo-random test only. 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. Significantly different results are produced for different LFSRs, even when the lengths of the phases are retained equal. For illustration, see the BIST design for the c1908 ISCAS benchmark circuit [Brg85] results shown in Table 4.2. The pseudo-random phase was run for 2000 cycles, the LFSR polynomial was set constant (1-tap, see [Fis04b, Fis05a]), the LFSR was repeatedly randomly reseeded. Then the deterministic phase was run for 1000 clock cycles. 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.

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

It is impossible to compute the proper LFSR seed and/or generating polynomial analytically for practical examples, due to the complexity of this problem. Thus, in praxis the LFSR is repeatedly reseeded several times and the fault simulation is conducted. Then the best seed (covering most of faults) is selected. The fault simulation is often a very fast process, thus it does not significantly influence the BIST design time.

Table 4.2: Influence of the LFSR seed

Run #	*ud.*	*vct.*	*GEs*
1	19	10	7.5
2	21	9	19.5
3	24	13	23.5

Run #	*ud.*	*vct.*	*GEs*
11	33	15	37
12	34	16	33
13	36	18	38

Run #	*ud.*	*vct.*	*GEs*
4	26	15	28
5	26	13	25
6	28	15	37.5
7	28	14	22.5
8	30	14	36
9	32	16	31
10	33	17	27.5

Run #	*ud.*	*vct.*	*GEs*
14	37	20	40.5
15	39	22	53
16	44	26	40
17	46	22	42.5
18	48	24	44
19	52	28	63.5
20	62	34	69

4.5 The Deterministic Phase

In the deterministic phase 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.

4.5.1 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 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. These don't cares can be very efficiently exploited, since they give more freedom to select the column matches.

There are some obvious restrictions for the matrices dimensions. The number of test patterns *p* must be $2^n - 1$ at most (the maximum number of distinct patterns generated by a LFSR) and $p \geq s$, because there must be enough patterns to implement all the 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, some pattern inhibition techniques may be used - see [Gir99]. The proposed 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 having *n* inputs and *r* outputs, where only values of *s* terms are

defined and the rest are don't care values. Such a 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* (see Fig. 4.7).

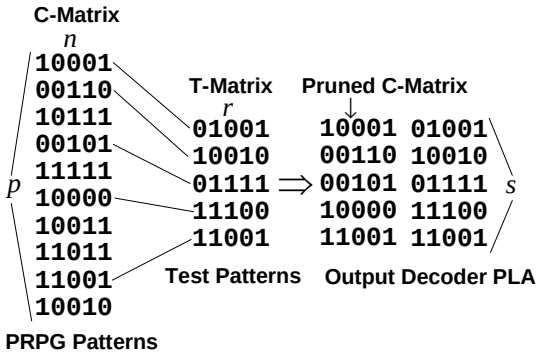

Figure 4.7: Assignment of the rows

4.5.2 The Column-Matching Algorithm

The task is now, how to assign the rows to each other to reach maximum area overhead reduction. The aim of the Column-Matching method is to assign 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 of the pruned **C** matrix columns in the result. This would yield no logic necessary to implement these **T** matrix columns (outputs of the decoder); they would be implemented as mere wires.

In most cases the PRPG outputs are drawn directly from the outputs of flip-flops. These flip-flops often also have 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.

An illustrative example is shown in Fig. 4.8. The matched columns of the pruned **C** matrix and **T** matrix from Fig. 2 are shown here. The **T** matrix column y_1 is matched with the **C** matrix column x_3 (negatively), then y_3 with x_1 (negatively) and y_4 with x_4 (positively).

Thus, the outputs y_1, y_3 and y_4 are implemented without any combinational logic, while the remaining outputs have to be synthesized using some standard two-level Boolean minimization tools, like ESPRESSO [Bra84] or BOOM [Hla01, Fis03b].

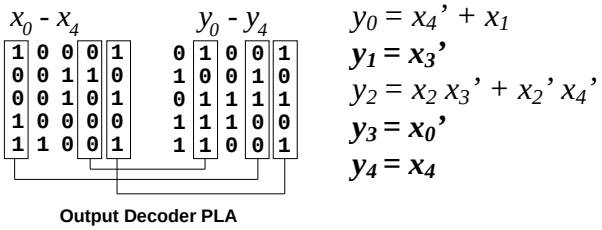

Figure 4.8: Column matching example

4.5.3 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 possible 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 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.

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 **C** matrix 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 the 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 the 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}.

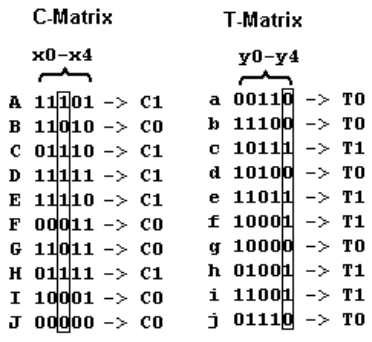

Figure 4.9: The first assignment to the submatrices

Finding all possible column matches consists in a successive decomposition of the 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 number of combinations of possible column matches grows exponentially with r (number of **T** matrix vectors). Particularly, there are n^r possible combinations (where n is the number of **C** matrix columns). Thus, the selection of the candidate columns for a match is driven by a heuristic, measuring the ratio of zeros and ones in both the candidate columns. The most balanced decomposition is then selected. 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.

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

4.5.4 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 in the method described in the previous subsection, we cannot determine a column match by comparing the number of ones (and zeros) in the corresponding columns, because it is not known 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 a column 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 be 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) = SelectTwoColumnsToBeMatched(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 (𝒮ᵤᶜ[k, i] == 0) C⁰ = C⁰ ∪ 𝒮ᵤᶜ[k];
                  else C¹ = C¹ ∪ 𝒮ᵤᶜ[k];
            T⁰ = ∅;
            T¹ = ∅;
            for (l = 0; l < T_matrix_rows; l++)
                  if (𝒮ᵤᵀ[l, j] == 0) T⁰ = T⁰ ∪ 𝒮ᵤᵀ[l];
                  else T¹ = T¹ ∪ 𝒮ᵤᵀ[l];
            if (|C⁰| < |T⁰| || |C¹| < |T¹|) return 𝒮;
            𝒮' = 𝒮' ∪ {[C⁰, T⁰]; [[C¹, T¹]};     // add the split sets
            𝒮 = 𝒮';
      }
   }
}
```

4.5.5 Column Matching Process Example

To illustrate the principles of the method, the c17 ISCAS benchmark [Brg85] was chosen 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). The 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 purely illustrative purposes. It is known that c17 can be completely tested with 4 patterns and that, on the other hand, if an exhaustive test was used (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

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

One-to-One Assignment for c17 Benchmark

In this example the decomposition of matrices into set systems is shown for the one-to-one assignment. There are 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.

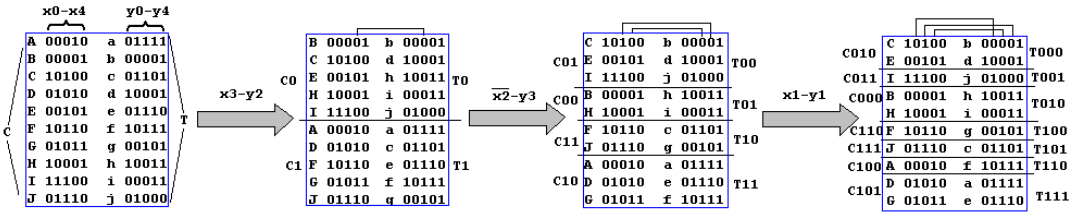

Figure 4.11: One-to-one exact Column-Matching example

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.

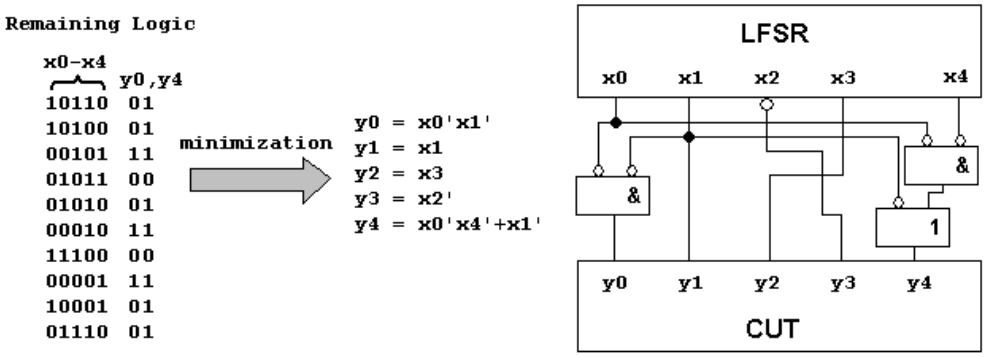

Figure 4.12: BIST implementation for c17 circuit

Generalized Column-Matching Example

Three exact column matches have been found for the one-to-one assignment in the previous example, whereas the decoder for the remaining two variables (y_0, y_4) had to be synthesized. Now let's try to let the LFSR run for more than the minimum required 10 cycles and see if more exact matches will be achieved.

It can be found experimentally, that when the LFSR generating polynomial and seed are retained from the previous example, 19 LFSR cycles are needed to match *all* the columns. Thus, no additional logic is needed to build the output decoder. In Fig. 4.13

one of the possible assignments of the test patterns is shown. The combinational logic of the Output Decoder is completely eliminated, since the decoder 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.

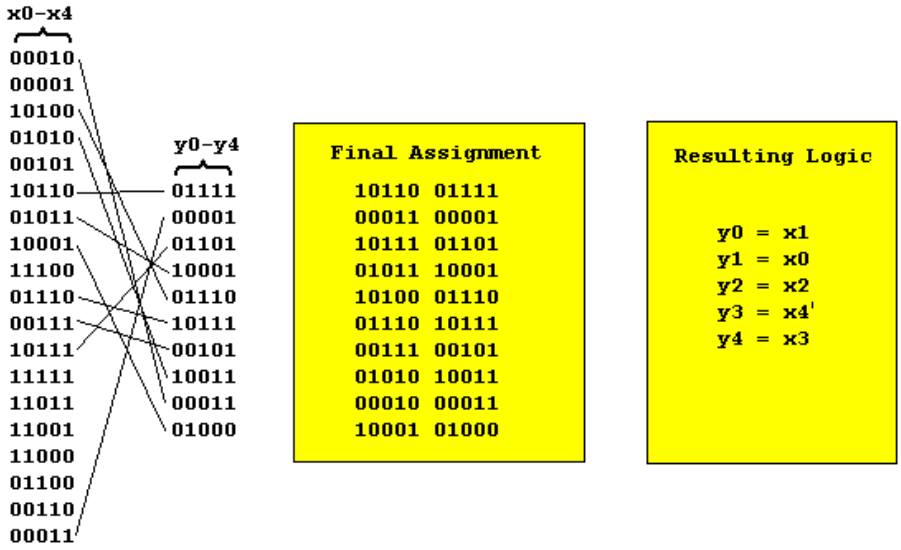

Figure 4.13: Assignment of rows for c17 circuit

4.6 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, since they give more freedom to the matching process.

The process 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 particular vectors are produced by the 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. When the test don't cares are present, these sets become non-disjoint. This is because it cannot be decided what values should be assigned to the don't care values, until all the matches are performed. Thus the algorithm consists of two linked NP-hard problems. Using the set system approach here is rather time-consuming, although using it is not impossible. The disadvantage of this approach consists in the need of duplicating of the **C** matrix vectors after every column match, if they are assigned to **T** matrix vectors containing don't cares.

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 "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, the **B** matrix cells [*k, l*] are set to "0" when the *k*-th input row contains in the *i*-th column the opposite value to the *l*-th output row in the *j*-th column. Thus, rows that contain opposite values in the matched columns cannot be assigned to each other.

$$\mathbf{B}[k, l] := \text{“0” when } (\mathbf{C}[k, i] \neq \mathbf{T}[l, j] \wedge \mathbf{T}[l, j] \neq \text{don’t care}) \qquad (4.1)$$

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 NP-hard. 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.

4.6.1 Row Assignment Algorithms

It would be often extremely time-consuming to solve the row assignment problem exactly, thus greedy incremental heuristic is used. Since the Column-Matching algorithm needs to solve this problem after performing every column match, the row assignment heuristic should be as fast as possible. 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 several methods has been proposed and the results compared.

One method (LCLR – *least in column, least in row*) is a simple greedy heuristic. The **B** matrix column with the least number of "1" values is found (because the respective **T** matrix vector could be difficult 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 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.

The efficiency of the algorithms is shown on the results obtained by processing 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 was done, 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 the 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.

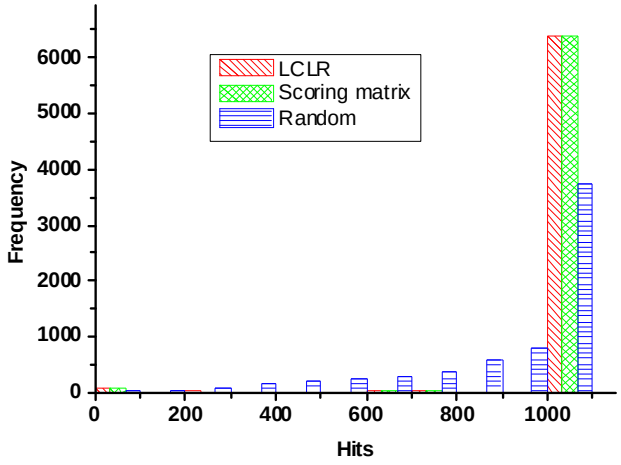

Figure 4.14: Row assignment histograms

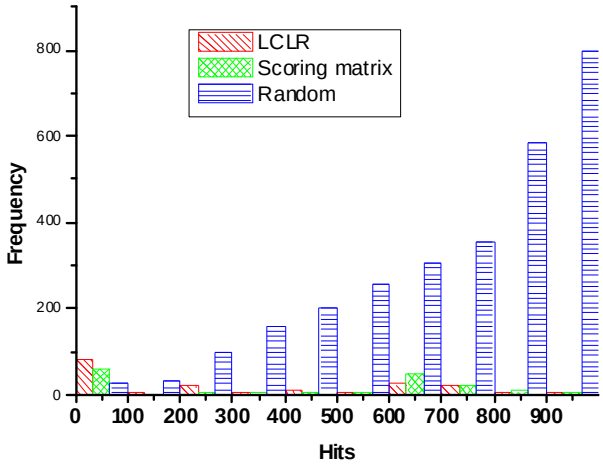

Figure 4.15: Close-up view of Fig. 4.14

We can see that in most cases both the LCLR and scoring matrix based heuristics found a solution, while the randomized method was not that successful. Particularly, 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

It can be concluded from these results that both the LCLR and scoring matrix based algorithms are quite efficient, unlike the random approach. Both the algorithms are almost equally successful, however the scoring matrix based method is more than 10 times slower. For this reason, the LCLR row assignment algorithm has been chosen as a good compromise. 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 and there are s possible assignments, the asymptotic time complexity of the algorithm is $O(p \cdot s^2)$. The algorithms are described in [Fis04d] more thoroughly.

4.6.2 Column-Matching Algorithms

Three major algorithms driving the whole Column-Matching process have been developed:

- Exact search
- Thorough search
- Fast search

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 for practical problems and it won't be discussed any more in this Thesis.

The "simplest" possible column matching method can be described as follows: when a non-valid column match is encountered (during the row assignment process), the whole process is stopped. This is the fastest algorithm developed. It is often suitable for problems with a large number of variables. Because of the row assignment is repeated after each column match and there could be r column matches at most, the asymptotic complexity of this algorithm is $O(r \cdot p \cdot s^2)$. It corresponds to a case where all the r 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. This algorithm was named a ***thorough search***. The asymptotic complexity increases to $O(n \cdot r^2 \cdot p \cdot s^2)$, 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 3 column matches only (after 30 ms), while the *thorough search* ran for 198 cycles, but reached 21 column matches (in 200 ms). It is obvious from this example that the thorough search significantly outperforms the fast search in the quality.

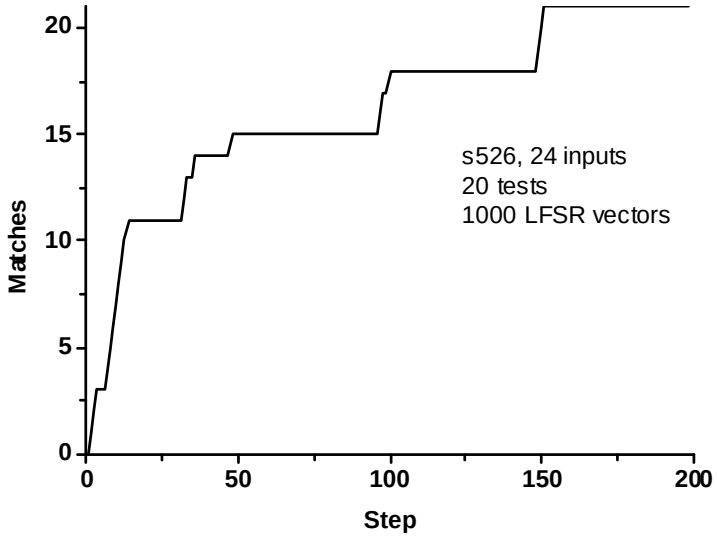

Figure 4.16: Thorough search progress

Several modifications can be yet done to improve the result quality. The selection of column matches is being done purely at random. Thus, when the whole Column-Matching process is repeated several times, there is a chance that a better solution will be obtained. 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 good solutions. 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 (randomly) 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. Only 5 column matches were obtained in the first run, 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.

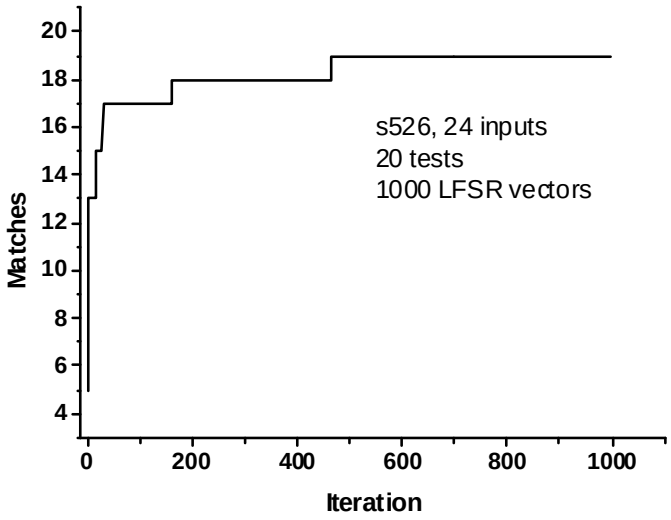

Figure 4.17: Repetitive fast search

4.6.3 The Basic Fast Search 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 than 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 (by performing the row assignment). 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 the 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 a backup of the row assignment
      A = MakeRowAssignment(B);   // do a 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;
}
```

The Thorough Search algorithm is very similar to this one, but a one-step backtracking is involved there.

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

It has been assumed up to now that applying the 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 pseudo-random and deterministic phase, the Switch is present as well. The aim is to minimize both the Output Decoder and the switching logic. There are five alternatives that can occur when designing the logic 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 the example in Fig. 4.4 it is the 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 the example in Fig. 4.4 it is the 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, 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 proper output values, while an additional multiplexer has to be present in the switching block. This is the case of y_4 in Fig. 4.4.

The first case mentioned is, of course, the one with the least 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, the indirect ones are being made. When no matches are possible, Column-Matching stops and the resulting outputs are synthesized by BOOM [Hla01, Fis03b].

4.7 Multiple-Vector Column-Matching

The BIST area overhead becomes an essential issue now. For ASIC designers the area becomes more important than the design time, since the overall chip design time significantly surpasses the BIST design time. Thus, any improvement of the BIST design methods, in terms of the area overhead, is beneficial. Such an improvement of the Column-Matching algorithm is proposed in this Subsection. A significant area overhead reduction is involved, for a cost of a longer design time. The improvement consists in a generalization of the basic method, to fully exploit capabilities of ATPGs. The ATPG generates more than one test vectors for each tested fault in the proposed enhancement, thus the algorithm has more freedom in generating the test sequence [Fis06a].

4.7.1 ATPG Modes

The Column-Matching method is so universal, that any test vectors set can be used. Most of the available non-commercial ATPGs can be influenced, so that they produce various sets of test vectors. The only and necessary requirement for the ATPG tool used is the capability to produce test vectors for a specified set of faults.

In the most general case, possible test sets that can be obtained may be divided as follows:

1. *Non-compacted test without don't cares*. Such a test set is usually obtained by a random-pattern simulation and subsequent deterministic test generation. The test is usually long and far from optimum length.
2. *Compacted test set without don't care values*. Here the test comprises of minimum of test vectors (in the optimum case), obtained after deterministic test set generation and compaction, followed by the don't care substitution.
3. *Compacted test set with don't cares*. The test comprises of minimum of test vectors (in the optimum case), obtained after deterministic test set generation followed by a compaction. The don't care values are retained. However, their number is usually negligible.
4. *Non-compacted test with don't cares*. The test set is produced by a deterministic test set generator only. No test compaction is executed.
5. *One test vector for each fault.* The test pattern generation is usually accompanied by a fault simulation. Thus, after one test vector is produced during the test generation process, fault simulation is executed for this vector and faults detected by it are removed from the processed fault list. This was the case assumed in the preceding cases. However, the ATPG may proceed in the simplest way, by generating one deterministic vector for each fault. No fault simulation or test compaction is involved. Test vectors with many don't care values are usually obtained. The test set is often large, comparing to previous cases, however many don't care values are present in the test, which is usually beneficial.
6. *More than one test vectors for each fault.* As a generalization of the previous item, more test vectors for each fault can be produced, if possible. The test is then even longer, but offers much greater flexibility.
7. *All the possible test vectors for each fault.* This is the most general case. Some ATPGs are able to produce all the possible test vectors for each fault. However, the test set size is then prohibitively large, thus such a case usually cannot be used in practice.

The don't cares present in the test set (**T** matrix) are beneficial, since they bring more freedom to the column matches choosing, and consequently reduce the BIST area overhead. Thus, the test generation alternatives 3-7 are more advantageous than the topmost ones.

Atalanta [Lee93] ATPG tool was used throughout this work. It is able to generate all the test sets listed above. However, is has been found during the experiments, that the test set compaction method, which is used in the Atalanta, does not perform well, thus a new static compaction method based on joining test vectors was introduced.

Then, the Column-Matching algorithm has been extended to be able to handle test sets having more than one test vectors for each fault, to improve the quality of its results.

4.7.2 Simple Test Set Compaction

Due to the fact that the test set compaction performed by an ATPG is often lacking in quality, a new static compaction method was introduced to the BIST design process. Maximum of the don't care values in the test set should be retained after the compaction. An exact test compaction algorithm is usually not applicable, since its time

complexity is prohibitively large (it is an NP-hard problem). Thus, a heuristic method has to be used.

The proposed algorithm is simple but effective. It is based on joining pairs of test vectors. Two test vectors may be joined, when they have a non-empty intersection. The result of their joining will be that intersection. Considering that a test vector t_1 detects a fault set F_1 and a test vector t_2 detects a fault set F_2, their intersection $t_1 \cap t_2$ detects faults $F_1 \cup F_2$.

Let us have a test set comprising of v vectors. Each vector is compared with each other and the size (dimensionality) of their intersection is computed. Two vectors having the "biggest" intersection will be joined. In other words, two vectors differing in at least one bit cannot be joined (since they have an empty intersection); two vectors having minimum collisions with a don't care on one side and a '0' or '1' value on the other side are joined. Test vectors loosing minimum number of don't cares by their joining are joined. This is being repetitively performed until there is no chance to join any more vectors. The complexity of such an algorithm is $O(v^3)$, which sometimes means a significant computational time increase. The number of test vectors can be significantly reduced by this method, see Table 4.4. First, "*PR*" pseudo-random patterns were simulated. Test sets for the undetected faults were computed by Atalanta ATPG [Lee93]. The ATPG was set to generate "*vct/vlt*" vectors for each fault. The total number of test vectors is shown in the "*ATPG*" column. After the compaction, their number was reduced to "*compact*". The amount of don't care values in the final compacted test set is shown in the last column. It can be well observed that by increasing the number of test vectors the number of don't cares decreases. This is due to the fact that the compaction algorithm preferably selects vectors having many don't cares to be joined. However, this "disadvantage" is compensated by the freedom offered by the number of vectors more than enough.

Table 4.4: Test compaction results

bench	*PR*	*vct/flt*	*ATPG*	*compact*	*DC*
c1908	1000	1	42	36	50 %
		10	382	340	25 %
c2670	10 K	1	201	74	83 %
		10	1824	825	77 %
c3540	1000	1	31	25	72 %
		10	117	101	65 %
		100	663	555	56 %
c7552	10 K	1	215	106	69 %
		10	2141	1206	68 %
s1196	1000	1	93	55	59 %
		100	392	259	56 %
s1238	3000	1	45	33	57 %
		100	140	95	52 %
s5378	10 K	1	23	19	92 %
		100	289	258	92 %
s9234.1	50 K	1	321	99	82 %
		10	2899	1003	81 %
s13207.1	10 K	1	466	74	96 %
		10	1538	362	96 %

4.7.3 Multiple-Vector Column-Matching Principles

The more “freedom” has the Column-Matching algorithm in selection of the matches, the better it performs. Particularly, more don’t care values in the test set induce more column matches and thus a less area overhead. Let us consider an example where two test vector sets are to be mapped onto PRPG patterns, one set generated by the 3-rd APTG mode (compacted test set with don’t cares), the second one as the 5-th one (one test vector for each fault, with don’t cares). The second test set will be much larger than the first one. On the other hand, more don’t care values will be present in the second one (in general). Practical examples have shown that even when there are more test vectors to be generated by the Output decoder, the BIST area overhead is less if the test vectors have many don’t care values. Thus, the second case will perform better, in terms of the area overhead.

The above-mentioned notion can be extended, so that there will be more than one test vectors available to choose from. The aim of the Column-Matching algorithm wouldn’t be to synthesize all the test vectors then; the aim would be to synthesize vectors that cover all faults (from the given fault set), regardless by what vectors. Thus, even more freedom will be given to the matching algorithm, which yields better results. This is the main idea of the *Multiple-Vector Column-Matching*.

In order to adapt basic Column-Matching principles to be able to exploit more test vectors for each fault, several modifications have to be done. First of all, each test vector has to be accompanied by a *fault mask*. The fault mask is a binary vector identifying faults that are detected by the test vector. First, the fault list for the tested circuit is determined. The size of the fault mask is then equal to the number of faults, each position in the fault list corresponding to one fault. A ‘1’ value indicates that the respective fault is detected by the test vector, ‘0’ the contrary. The fault mask is obtained by a fault simulation of the respective test vector. After the fault masks are generated, all the test vectors are put together; there is no need to distinguish between them. Information on what vector was generated for what fault may be lost.

4.7.4 Multiple-Vector Test Set Compaction

Since the number of test vectors generated by an ATPG is often large, static test compaction should be performed. Basically, the algorithm described in Subsection 4.7.2 is used. The only modification is that after joining the test vectors their fault masks have to be joined too. The resulting fault mask is obtained by OR-ing the two fault masks, since faults detected by both joined vectors are detected by the resulting vector.

Even when the test set compaction reduces the freedom given to the Column-Matching, it has been found experimentally that it is advantageous to perform it. When the test set compaction is not performed, the Column-Matching runtime is prohibitively long and usually there is no improvement in the quality of the result.

Example

Let us consider a 5-input CUT having 10 faults. The two example test vectors together with their fault masks will be joined as follows:

```
10-0-  1100101001
1-10-  0100100100
1010-  1100101101
```

4.7.5 Modified Row Assignment

The basic Column-Matching algorithm remains unchanged but the row assignment. Since there are more test vectors detecting each fault, the test set is extremely redundant. Thus, not all test vectors have to be synthesized by the Output decoder; the primary aim is to detect all faults now, regardless by what vectors. Not all the **B** matrix columns have to be assigned then.

The heuristic used to solve the row assignment problem (see 4.6.1) is modified in this way: the heuristic function for a selection of a **B** matrix column (test vector) is the number of yet undetected faults it detects. At the beginning of the algorithm, the **B** matrix column detecting most of faults (i.e., test vector having most '1's in the fault mask) is selected. For this column a row having a '1' value in the respective position in the **B** matrix is found, so that this row has a minimum number of '1's in other positions. It is the row (**C** matrix vector), that may be transformed into the required test vector and simultaneously may be transformed into a minimum of others. The selected column and row are assigned to each other, removed from the **B** matrix and the detected faults are removed from the fault list. The column selection is repeated, until the fault list is empty or an undetectable fault is encountered (which means an invalid assignment). When an invalid assignment is returned, the last column match is taken back and another column matches are tried.

Basic principles of the row assignment are outlined by the following pseudo-code:

Algorithm 4.3: Multiple-vector row assignment

```
Assign {
  set(fl); // create a complete fault list
  do {
     c = FindBestColumn(B, faultmasks, fl);
      // find column detecting most faults from fl
     r = FindBestRow(B, c);
      // find a row, so that B[r, c] = 1 and has a minimum of 1's
     if (r != NULL) {   // a row B[r, c]=1 was found
        MakeMatch(c, r);
        RemoveFaults(fl, c);
          // remove faults detected by c from fl
        RemoveColumn(B, c);
          // remove c from B matrix
     } else return(FAIL);
  } while(!empty(fl));
  return(SUCCEED);
}
```

4.7.6 Modified BIST Design Process

Summarizing all the modifications needed to be done to extend the BIST design method to support Multiple-Vector Column-Matching, the whole process consists of these phases:

1. Simulation of several (*PR*) pseudo-random patterns for the CUT and determination of undetected faults.
2. Computation of the deterministic test patterns for these faults by an ATPG tool, generating more than one test pattern for each fault.
3. Fault simulation for each of the test vectors, i.e., computing fault masks.

4. Test set compaction.
5. Performing the multiple-vector column-matching.
6. Synthesis of the decoder for the unmatched outputs.

4.8 Influence of the Length of the Deterministic Phase

In the deterministic phase deterministic vectors are synthesized from the PRPG patterns (following the pseudo-random phase). By increasing the number of PRPG patterns the chance to find more column matches increases. This is due to having more freedom for selecting the PRPG vectors to be assigned to the deterministic vectors. Unfortunately, the Column-Matching runtime rapidly increases with the number of vectors (see Subsection 4.6.2).

This is illustrated in Table 4.5. The benchmark name is shown in the first column. The "*PR*" and "*Det.*" columns indicate the lengths of the pseudorandom and deterministic phases, in the "*SW GEs*" and "*OD GEs*" columns the overhead of the Switch and Output decoder are shown. These are then summed together to obtain the total BIST combinational logic overhead, in terms of gate equivalents [DeM94]. The BIST design time is shown in the last column. The experiment was run on a PC with Athlon CPU, on 1 GHz, Windows XP.

Table 4.5: Influence of the deterministic phase length on the result

bench	*PR*	*Det.*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*
c1908	1000	500	36	54.5	90.5	1.6
		1000	33	48	81	4.88
		2000	30	50	80	8.47
		5000	30	38.5	68.5	25.78
c3540	1000	200	28.5	5.5	34	0.32
		500	28.5	1	29.5	0.52
		1000	27	1	28	1.02
		2000	16.5	**0**	16.5	1.47
		5000	7.5	**0**	7.5	2.93
s1196	5000	200	15	10.5	25.5	0.17
		500	18	7	27	0.32
		1000	10.5	6.5	17	0.48
		2000	9	8	17	1.52
		5000	7.5	1.5	9	2.16
		10000	4.5	**0**	4.5	5.83

It can be well 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.

4.9 Summary Discussion on the Lengths of the Two Phases

In subsections 4.3.1 and 4.8 the influence of the lengths of the two BIST phases on the resulting logics and design time were discussed separately. As it was said in the Introduction, four important aspects play role in the BIST design:

- BIST design time
- Fault coverage
- BIST area overhead
- BIST run length

The method proposed in this Thesis offers a big scalability in all these four aspects. This is briefly summarized in the following table. The fault coverage aspect is not considered, since 100% s-a fault coverage is considered throughout this Thesis. Anyway, downgrading the requirements for the fault coverage would decrease all the BIST design time, area overhead and BIST run length, according the needs of the BIST designer.

By increasing the length of the pseudorandom phase the number of undetected faults is decreased. Thus, the number of deterministic test vectors that are needed to be generated in the deterministic phase is decreased as well. As a consequence of this, the BIST design time is often significantly decreased (even though the fault simulation time is higher) and the area overhead is reduced as well.

On the other hand, by prolonging the run of the deterministic phase the Column-Matching algorithm runtime is increased, see Subsection 4.6.2. The algorithm has more freedom in the selection of matches on the other hand, so the area overhead of the decoder is decreased.

Table 4.6: Influence of the test lengths

	Longer PR phase	Longer Det. phase
BIST design time	decreased	increased
BIST area overhead	decreased	decreased
BIST run length	increased	increased

4.10 Comparison with Other State-of-the-Art Methods

In this section the proposed Column-Matching method is compared with four different state-of-the-art methods, namely the basic bit-fixing method [Tou95], the bit-fixing accompanied by a "bit-correlating" ATPG [Tou01], the "3-Weight Weighted Random BIST" proposed in [Wan01] and the row matching method proposed in [Cha03]. The comparison is shown in Table 4.7. The *"TL"* columns indicate the total length of the test, the *"GEs"* columns give the number of gate equivalents (or 2-input NAND gates) of the BIST combinational circuits and the "*lit.*" columns indicate the number of literals in the SOP form of the decoding logic (the Switch logic is not considered in Column-Matching here).

Let us note here, that a special kind of a PRPG (GLFSR) 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. The Column-Matching method is independent on a PRPG used, in general, thus in all the cases an LFSR with one XOR gate only was used. Thus, sometimes bigger area overhead of our method could be compensated by a small area of the PRPG used.

In the bit-fixing and weighted BIST methods several registers (flip-flops) are used. In the Column-Matching method no flip-flops are needed.

The Column-Matching results describe the overall test length and the number of gate equivalents of the decoder. The number of GEs approximately corresponds to the number of SOP literals, thus a comparison with the Bit-fixing and Weighted BIST methods can be freely made.

The empty cells indicate that the data for the respective circuit was not available.

The Column-Matching algorithm was run in the Thorough Search mode, using one test vector per one fault. The area overhead thus can be yet improved by using Multiple-Vector Column-Matching, for a cost of a longer algorithm runtime (see Table 4.8).

Table 4.7: Comparison results

	Bit-fixing [Tou95]		Bit-fixing [Tou01]		Weighted BIST [Wan01]		Row-matching [Cha03]		Column-Matching	
Bench	*TL*	*GEs*	*TL*	*lit.*	*TL*	*lit.*	*TL*	*GEs*	*TL*	*GEs*
c880	1 K	27	-	-	-	-	640	21	1 K	15
c1355	3 K	11	-	-	-	-	1.8 K	0	1.5 K	15
c1908	4 K	12	-	-	-	-	4.7 K	8	3 K	10.5
c2670	5 K	121	10 K	385	8 K	269	6 K	119	5 K	113
c3540	4.5 K	13	-	-	-	-	4.8 K	4	5.5 K	1.5
c7552	10 K	186	10 K	806	6.7 K	641	8 K	297	8 K	586
s420	1 K	28	10 K	59	1.4 K	67	-	-	1 K	24.5
s641	10 K	12	10 K	98	768	45	7.7 K	6	4 K	15
s713	-	-	-	-	-	-	4.8 K	4	5 K	16.5
s838	10 K	37	10 K	183	3.1 K	108	-	-	6 K	130
s1196	-	-	10 K	97	16.8 K	67	10 K	36	10 K	6
s1238	-	-	-	-	17 K	33	-	-	4 K	26.5
s5378	-	-	10 K	332	18.4 K	68	-	-	11 K	19.0

4.11 Column-Matching Results for Standard Benchmarks

Since the comparison shown in the previous table describes results for a few benchmark circuits only, a more exhaustive result table is presented here, for the ISCAS [Brg85, Brg89] and ITC'99 [Cor99] benchmarks. For each benchmark the BIST circuitry was synthesized in several different parameters setting. Generally, each benchmark is processed in three ways: first, the testing time is kept low, thus the area overhead is higher and the BIST design time is short. Then the length of the BIST phases is increased, yielding a reduction of the BIST logic, for a cost of a longer test time. The last result for each benchmark represents the Multiple-Vector Column-Matching, where the BIST area overhead is reduced furthermore, for a cost of a bigger BIST design time. The trade-off between the test length, an area overhead and the BIST design time can be seen in the presented examples.

The "*inps*" column indicates the number of the benchmark inputs. The "*GEs*" column shows the complexity of the benchmark circuit, in terms of gate equivalents [DeM94]. In the "*PRand*" column the number of pseudo-random vectors needed to be applied to the CUT to be completely tested is shown, just for comparison. The "*TL*" column gives the lengths of the pseudorandom and deterministic phases. The "*vcts/flt*" column indicates the number of test vectors generated for each fault. The number of faults that were undetected by the pseudo-random phase is shown in the "*undet*" column. After that the total number of deterministic test vectors that have to be generated by the deterministic phase is shown. 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 "*Time*" column. The BIST area overhead is shown in the last column.

The experiments were run on a CPU AMD Athlon, 900 MHz.

Table 4.8: ISCAS & ITC benchmarks

Bench	*inps*	*GEs*	*PRand*	*TL (PR + Det.)*	*vcts/flt*	*undet*	*vcts*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*	*Overhead*
c880	60	364.5	2.5 K	100 + 100	1	30	28	53	22	57	12.5	69.5	0.50	19.0%
				500 + 500	1	10	10	**60**	50	15	**0**	15	0.4	4.12%
				500 + 500	100	13	1201	**60**	51	13.5	**0**	13.5	33.8	3.70%
c1355	41	532	2 K	500 + 500	1	40	13	24	6	52.5	27	79.5	3.26	15.0%
				1000 + 500	1	10	1	**41**	31	15	**0**	15	0.05	2.82%
				1000 + 500	100	10	8	**41**	33	12	**0**	12	0.10	2.2%
c1908	33	749	4 K	1000 + 1000	1	51	36	28	15	27	16	43	6.76	5.74%
				2000 + 1000	1	16	6	**33**	26	10.5	**0**	10.5	0.27	1.40%
				2000 + 1000	100	17	189	**33**	31	3	**0**	3	2.49	0.40%
c2670	233	1038	4.5 M	4000 + 1000	1	320	104	197	166	100.5	146.5	247	606	23.80%
				10 000 + 1000	1	321	74	201	180	79.5	77.5	157	896	15.13%
				10 000 + 1000	10	321	825	202	166	100.5	59.5	160	5889	15.41%
c3540	50	1469.5	15 K	1000 + 1000	1	167	21	**50**	40	15	**0**	15	1.60	1.02%
				2000 + 1000	1	147	9	**50**	43	10.5	**0**	10.5	0.35	0.71%
				2000 + 1000	10	147	9	**50**	45	7.5	**0**	7.5	3.33	0.51%
c7552	207	3072	> 100 M	5000 + 1000	1	416	207	135	28	268.5	417	685.5	3892	22.31%
				10 000 + 1000	1	362	106	152	25	273	250.5	523.5	1105	17.04%
				10 000 + 1000	10	362	1206	159	33	261	192.5	453.5	161 K	14.76%
s420.1	34	191.5	165 K	1000 + 1000	1	61	32	29	20	21	14.5	35.5	4.34	18.54%
				5000 + 1000	1	46	19	**34**	21	19.5	**0**	19.5	1.48	10.18%
				5000 + 1000	10	46	50	**34**	22	18	**0**	18	5.01	9.40%
s641	54	269.5	200 K	500 + 500	1	20	13	**54**	36	27	**0**	27	1.16	10.02%
				1000 + 1000	1	15	14	**54**	40	21	**0**	21	2.27	7.79%
				1000 + 1000	10	15	14	**54**	42	18	**0**	18	14.02	6.68%
s713	54	352.5	300 K	500 + 500	1	61	15	52	37	25.5	3	28.5	1.29	8.09%
				1000 + 1000	1	53	11	**54**	40	21	**0**	21	1.90	5.96%
				1000 + 1000	10	53	126	**54**	41	19.5	**0**	19.5	13.36	5.53%

Bench	*inps*	*GEs*	*PRand*	*TL (PR + Det.)*	*vcts/flt*	*undet*	*vcts*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*	*Overhead*
s838	67	393.5	> 100 M	5000 + 1000	1	101	61	43	18	73.5	33.5	107	50.3	27.19%
				50 000 + 1000	1	91	50	51	13	91	22	113	34.23	28.72%
				50 000 + 1000	10	91	510	52	10	85.5	25	110.5	348	28.08%
s953	45	458.5	15 K	500 + 500	1	156	45	41	34	16.5	11	27.5	3.43	6.0%
				1000 + 1000	1	70	25	43	39	9	4.5	13.5	2.47	2.94%
				1000 + 1000	10	70	73	44	37	12	2	14	16.83	3.05%
s1196	32	504.5	200 K	1000 + 1000	1	93	55	27	25	10.5	45.5	56	5.53	11.10%
				3000 + 1000	1	114	33	27	24	12	26.5	38.5	2.96	6.70%
				3000 + 1000	100	114	95	28	25	10.5	16	26.5	16.68	4.61%
s1238	32	574.5	20 K	1000 + 1000	1	171	60	27	21	16.5	28.5	65	7.85	11.31%
				3000 + 1000	1	114	45	27	24	12	26.5	38.5	2.96	6.70%
				3000 + 1000	10	114	140	28	25	10.5	16	26.5	16.69	4.61%
s5378	214	2134.5	50 K	5000 + 1000	1	89	31	213	171	64.5	1	64.5	22.59	3.07%
				10 000 + 1000	1	63	19	**214**	193	31.5	**0**	31.5	7.68	1.48%
				10 000 + 1000	100	64	258	213	203	15	4	19	181.5	0.89%
s9234.1	247	3985.5	10 M	1000 + 1000	1	1674	215	172	86	241.5	641.5	883	5238	22.16%
				200 000 + 1000	1	599	52	224	147	150	62.5	212.5	161	5.33%
				200 000 + 1000	10	599	564	225	162	125.5	66	193.5	3509	4.86%
s13207.1	700	5596.5	100 K	10 000 + 1000	1	617	74	696	532	252	28	280	348	5.00%
				50 000 + 1000	1	182	21	**700**	676	36	**0**	36	12.86	0.64%
				50 000 + 1000	100	182	31	**700**	678	33	**0**	33	42.50	0.59%
s15850.1	611	6824	> 10 M	10 000 + 1000	1	940	313	515	407	306	188	494	152 K	7.24%
				100 000 + 1000	1	674	180	563	444	250.5	78.5	329	21 K	4.82%
s38584.1	1464	16454	> 1 G	10 000 + 1000	1	1890	307	1000	949	772.5	155	927.5	5600	5.64%
				100 000 + 1000	1	1558	45	1000	961	754.5	16.5	771	146.18	4,68%
b04	77	545.5	15 K	1000 + 1000	1	56	37	68	54	34.5	19.5	54	16.6	9.90%
				2000 + 1000	1	34	24	75	51	39	4	41	7.79	7.88%
				2000 + 1000	10	34	253	76	60	25.5	1	26.5	71.11	4.86%
b05	35	518	10 K	1000 + 1000	1	65	33	28	21	21	36	57	5.25	11.00%
				2000 + 1000	1	32	20	30	20	22.5	15	37.5	2.24	7.24%

Bench	*inps*	*GEs*	*PRand*	*TL (PR + Det.)*	*vcts/flt*	*undet*	*vcts*	*M*	*DM*	*SW GEs*	*OD GEs*	*Total GEs*	*Time [s]*	*Overhead*
				2000 + 1000	10	32	200	33	24	16.5	4	20.5	19.81	3.96%
b07	50	378	200 K	1000 + 1000	1	45	41	45	29	31.5	10	41.5	10.65	10.98%
				1000 + 1000	100	45	2622	**50**	33	25.5	**0**	25.5	14.8 K	6.75%
b12	126	940.5	5 M	1000 + 1000	1	172	128	115	106	30	94	124	7730	13.18%
				1000 + 1000	10	172	955	119	108	27	44.5	71.5	108 K	7.60%

5 BOOM – The Boolean Minimizer

5.1 Motivation

The last phase of the proposed BIST synthesis method consists in the synthesis of the Output decoder. It is a combinational block “producing” the unmatched CUT inputs. In particular, the decoder is a multi-output combinational block having *n* inputs (where *n* is the number of PRPG outputs) and *r-m* outputs (where *r* is the number of CUT inputs and *m* the number of column matches reached). Such a multi-output Boolean function is described by a truth table.

This function usually has a large number of input variables, since the PRPG width is equal to the number of CUT inputs (both the primary and pseudo-primary) in this work. The number of its outputs is usually not that large, since many CUT inputs is matched in the previous (Column-Matching) phase. The number of inputs sometimes reaches hundreds or thousands. Thus, standard Boolean minimizers (like ESPRESSO [Bra84]) are unusable here, since their runtime is prohibitively large for such functions. For this reason a novel Boolean minimizer BOOM was developed. It is capable to handle functions having thousands of input variables in a very reasonable time.

5.2 Introduction

The problem of two-level minimization of Boolean functions is old, but surely not dead. It is encountered in many design environments like PLA design, multi-level logic design, design of control systems, or design of built-in self-test (BIST) equipment, and also in software engineering, artificial intelligence problems, etc. The systematic Boolean minimization methods mostly copy the structure of the original method by Quine and McCluskey [Qui52, McC56], implementing two basic phases known as prime implicant (PI) generation and covering problem (CP) solution. Some more modern methods, including the well-known ESPRESSO [Bra84, Hac96], try to combine these two phases because the problems encountered in up-to-date application areas often require minimization of functions with a prohibitively large number of inputs. Also the number of don't care states is mostly very large, hence the modern minimization methods must be able to take advantage of all don’t care states without enumerating them.

One of the most successful Boolean minimization methods is ESPRESSO [Bra84, Hac96] and its later improvements. ESPRESSO-EXACT [Rud87] was developed in order to improve the quality of the results, mostly at the expense of much longer runtimes. Finally, ESPRESSO-SIGNATURE [McG93] was developed, accelerating the minimization by reducing the number of prime implicants to be processed by introducing the concept of a “signature”, which is an intersection of all primes covering one minterm. This in turn was an alternative name given to the concept of “minimal implicants” introduced in [Ngu87]. Other Boolean minimization methods

exploiting the implicit set manipulation techniques were proposed in, e.g., [Cou92, Cou93]. The idea of meta-products was proposed, which allows the manipulation with extremely large sets of PIs.

A sort of a combination of prime implicant (PI) generation with a solution of the covering problem (CP), leading to a reduction of the total number of PIs generated, is also used in the BOOM (BOOlean Minimizer) approach proposed here. An important difference between the approaches of ESPRESSO and BOOM is in the way how they work with the on-set received as a function definition. ESPRESSO uses it as an initial solution, which has to be modified (improved) by expansions, reductions, etc., on the other hand BOOM uses the input sets (on-set and off-set) only as a reference that determines whether a tentative solution is correct or not. This allows reducing the dependence on the original function coverage. The second main difference is the top-down approach in implicant generation. Instead of expanding the source cubes in order to obtain better coverage (like in ESPRESSO), BOOM reduces the universal *n*-dimensional hypercube until it no longer intersects the off-set, while it covers as many 1-terms of the source function as possible. This phase is denoted as a CD-Search and represents the most innovative idea of the proposed method. Beyond this, some other commonly known algorithms (Implicant Expansion, Covering Problem solution, etc.) are used together with the CD-Search to obtain the final solution.

The algorithm is efficient above all for functions with a large number of input variables, where other minimization tools often fail to give a result in a reasonable time.

The principles of the proposed minimization method were published in [Fis00, Fis01a, Fis01b, Fis01c, Fis02c, Fis03b, Fis03c].

5.3 Problem Statement

Let us have a set of *v* Boolean functions of *n* input variables $F_1(x_1, x_2, \dots x_n)$, $F_2(x_1, x_2, \dots x_n)$, … $F_v(x_1, x_2, \dots x_n)$, whose output values are defined by truth tables. These truth tables describe the *on-set* $F_i(x_1, x_2, \dots x_n)$ and *off-set* $R_i(x_1, x_2, \dots x_n)$ for each of the functions F_i. The terms not represented in the input field of the truth table are implicitly assigned don't care values for all output functions. The *don't care set* $D_i(x_1, x_2, \dots x_n)$ of the function F_i is thus represented by all the terms not used in the input part of the truth table and by the terms to which don't care values are assigned in the *i*-th output column. The don't care values can be also specified explicitly in the truth table. Listing the two care sets instead of an on-set and a don't care set, which is usual, e.g., in MCNC benchmarks [MCNC], is more practical for problems with a large number of input variables, because in these cases the size of the don't care set heavily exceeds the two care sets. We will assume that *n* is of the order of hundreds and that only a few of the 2^n minterms have an output value assigned, i.e., the majority of the minterms are don't care states. Moreover, using off-set in the function definition simplifies checking whether a term is an implicant of the given function. Without the explicit off-set definition, more complicated methods using, e.g., tautology checking as in ESPRESSO [Bra84], must be used, which slows down the minimization process. And, most importantly, exactly such many-input incompletely defined functions specified by on-sets and off-sets are to be minimized to design the Output Decoder.

The task is to minimize the multi-output function F, so that the output of the algorithm will be an SOP (sum-of-products) expression describing the function, while the complexity of the resulting SOP form should be kept minimal. The measure of the complexity is sometimes vague, since it depends on the final implementation,

technology mapping, etc. Almost any quality criterion can be specified in the proposed method, since it is very flexible.

5.4 BOOM Structure

Like most other Boolean minimization algorithms, BOOM consists of two major phases: *generation of implicants* (PIs for single-output functions, group implicants for multi-output functions) and the subsequent *solution of the covering problem*. The generation of implicants for single-output functions is performed in two steps: first the *Coverage-Directed Search (CD-Search)* generates a sufficient set of implicants needed for covering the on-set of the source function, and the subsequent *Implicant Expansion (IE)* phase converts them into PIs.

Multi-output functions are minimized in a similar manner. Each of the output functions is first treated separately; the CD-Search and IE phases are performed in order to produce primes covering all output functions. However, to obtain a satisfactory solution, we may need implicants of more than one output function that are not primes of any (group implicants). Here, *Implicant Reduction* (IR) takes place. Then the *Group Covering Problem* is solved and the *Output Reduction* is performed. Fig. 5.1 shows a block diagram of the BOOM system, where each block corresponds to one minimization step and the data sets described between these blocks correspond to the products of these steps.

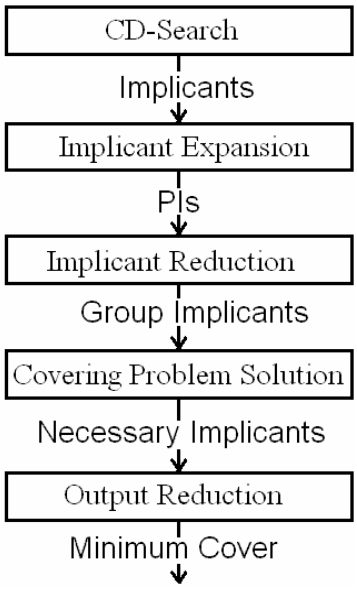

Figure 5.1: Structure of the BOOM system

The BOOM system may improve the quality of the solution by repeating the implicant generation phase several times and recording all different implicants that has been found. At the end of each iteration a set of implicants that is sufficient for covering all the output functions is obtained. In each of the following iterations, another sufficient set is generated and new implicants are added to the previous ones (if the solutions are not equal). This process is treated more thoroughly in Section 5.6.

5.5 Coverage-Directed Search

5.5.1 Basis of the Method

The idea of combining implicant generation with the solution of the covering problem was the basis of the Coverage-Directed Search (CD-Search) method used in the BOOM system. This consists in a search for the most suitable literals that should be added to some previously constructed term. Thus, instead of increasing the dimension of an implicant starting from a 1-minterm, an *n*-dimensional hypercube is being gradually reduced by adding literals to its term, until it becomes an implicant of F_i. This happens at the moment when the resulting hypercube does not intersect any 0-term.

The search for suitable literals that should be added to a term is directed towards finding an implicant that covers as many 1-terms as possible. To do this, the implicant generation starts with selecting the most frequent literal from the given on-set, because the (*n-1*) dimensional hypercube covering most of 1-minterms is described by the most frequent literal appearing in the on-set. The (*n-1*) dimensional hypercube found in this way may be an implicant, if it does not intersect any 0-term. If there are some 0-minterms covered, another literal is added and it is verified whether the new term already corresponds to an implicant by comparing it with 0-terms that may intersect this term. Again, the literal appearing in most of 1-terms is selected. After each literal selection the terms that cannot be covered by any term containing the selected literal are temporarily removed from the on-set, for more efficient search. These are the terms containing that literal with the opposite polarity. Literals are gradually added to a term under construction, until an implicant is generated. Then the term is recorded and the 1-terms that are covered by this term are removed from the on-set. Thus, a reduced on-set containing only yet uncovered terms is obtained. Now the whole procedure is repeated from the beginning. The search for implicants continues, until the whole on-set is covered.

The output of this algorithm is a set of product terms covering all 1-terms and not intersecting any 0-term. This algorithm is greedy and thus the obtained implicants need not be prime. In order to expand them into primes, the IE phase must be performed after the CD-Search.

The basic CD-Search algorithm for a single-output function can be described by the following function in pseudo-code. The on-set (F) and the off-set (R) are the inputs to the algorithm; the output is the sum of products (H) that covers the given on-set.

Algorithm 5.1: The CD-Search

```
CD_Search(F, R) {
    H = ∅;              // H is being created
    do
      F′ = F;           // F′ is the reduced on-set
      t = 1;            // t is the term in progress
      do
        v = most_frequent_literal(F′);
        t = t.v;
        F′ = F′ – cubes_not_including(t);
      while (t ∩ R ≠ ∅);
      H = H ∪ t;
      F = F - F′;
    until (F == ∅);
    return H;
}
```

5.5.2 Immediate Implicant Checking

When selecting the most frequent literal, it may happen that two or more literals have the same frequency of occurrence. In these cases either one is selected at random or another decision criterion is applied – namely the *immediate implicant checking*. The idea consists in constructing terms as candidates for implicants by multiplying all newly selected literals (those with the same frequency) by the previously selected one(s). Among these terms only implicants (if any) are selected and the rest is rejected. When there are still more possibilities to choose from, one is taken at random.

Sometimes this feature prevents a term from being unnecessarily "prolonged", because it would have to be "shortened" during the IE phase. The effects of using this immediate implicant checking are as follows:

- The runtime of CD-Search and the whole minimization runtime is decreased
- The number of PIs that are generated is slightly reduced.

This can be illustrated by Tab. 5.1. A single-output function having 20 input variables and 500 defined terms was minimized in 1000 iterations (see Subsection 5.6). In the first experiment, immediate implicant checking was not used, while in the second it was used.

Table 5.1: Immediate implicant checking effects

	not used	used
Total CD-Search time [s]	318,9	265,1
Total minimization time [s]	6688,3	4782,8
Number of PIs found	27194	21741

5.5.3 CD-Search Example

Let us have a single-output incompletely defined Boolean function of ten input variables $x_0..x_9$ and ten defined minterms given by a truth table Tab. 5.2. The 1-minterms are highlighted.

Table 5.2: CD-Search Example (1)

var:	0123456789	
0.	0000000010	1
1.	1000111011	1
2.	0000011001	1
3.	1111011000	0
4.	1011001100	0
5.	1111000100	1
6.	0100010100	0
7.	0011011011	0
8.	0010111100	1
9.	1110111000	1

In the first step the occurrence of literals in the 1-minterms is counted. The "0"-line and "1"-line in Tab. 5.3 give the counts of x_i' and x_i literals respectively. The most frequent literal is selected.

Table 5.3: CD-Search Example (2)

var:	0123456789
0:	3435322444
1:	3231344222

The most frequent literal is x_3' with five occurrences. This literal alone describes a term that is not an implicant, because it covers the 6th minterm (0-minterm) in the original function. Hence another literal must be added. When searching for the next literal, we can reduce the scope of the search by suppressing 1-minterms containing the selected literal with the opposite polarity (in Tab. 5.4 shaded dark). An implicant containing a literal x_3' cannot cover the 5th minterm, because it contains the x_3 literal. Thus, this minterm is temporarily suppressed. In the remaining 1-minterms the most frequent literal is found.

Table 5.4: CD-Search Example (3)

var:	0123456789	
0.	0000000010	1
1.	1000111011	1
2.	0000011001	1
3.	1111011000	0
4.	1011001100	0
5.	1111000100	1
6.	0100010100	0
7.	0011011011	0
8.	0010111100	1
9.	1110111000	1

var:	0123456789
0:	343-211433
1:	212-344122

As there are several literals with maximum frequency of occurrence 4 (x_1', x_5, x_6, x_7'), the second selection criterion (immediate implicant checking) must be applied. These literals are tentatively used as implicant builders and four product terms are generated (using the previously selected literal x_3': x_3'x_1', x_3'x_5, x_3'x_6, x_3'x_7'). Then it is checked for which of them are already implicants. The term x_3'x_5 is not an implicant (it covers the 6th minterm), so it is discarded and among the remaining three terms one is selected at random, e.g., x_3'x_6. This implicant is stored and the search continues.

The search for literals of the next implicants is described in Tab. 5.5. We omit minterms that are covered by the selected implicant x_3'x_6 (dark shading) and select the most frequent literal in the remaining minterms.

Table 5.5: CD-Search Example (4)

```
var: 0123456789
0.   0000000010 1
1.   1000111011 1
2.   0000011001 1
3.   1111011000 0
4.   1011001100 0
5.   1111000100 1
6.   0100010100 0
7.   0011011011 0
8.   0010111100 1
9.   1110111000 1

var: 0123456789
0:   1111222112
1:   1111000110
```

As seen in the lower part of Tab. 5.4, there are four equal possibilities, so one is chosen randomly – e.g. x_5'. In a similar way we can find another literal (x_6') needed to create an implicant covering the remaining two 1-minterms.

The resulting expression covering the given function is x_3'x_6 + x_5'x_6'.

5.6 Iterative Minimization

Most of current heuristic Boolean minimization tools use deterministic algorithms. The minimization process then always leads to the same solution, never mind how many times it is repeated. On the contrary, in the BOOM system the result of minimization depends to a certain extent on random events, because when there are several equal possibilities to choose from, the decision is made randomly. Thus there is a chance that repeated application of the same procedure to the same problem instance would yield different solutions and thus we can pick out the best solution. Moreover, the PIs and group implicants can be cumulated during the process and afterwards the CP solved using all of them, which sometimes yields a better final result.

5.6.1 The Effect of the Iterative Approach

The iterative minimization concept takes advantage of the fact that each iteration produces a new set of implicants sufficient for covering all 1-terms. This set

of implicants gradually grows until a maximum reachable set is obtained. The typical growth of the size of a PI set as a function of the number of iterations is shown in Fig. 5.2 (thin line). This curve plots the values obtained during the solution of a single-output function having 20 input variables and 200 care minterms. Theoretically, the more primes we have, the better is the solution that can be found after solving the covering problem, but the maximum set of primes is often extremely large. In reality, the quality of the final solution, measured by the number of literals in the resulting SOP form, improves rapidly during first few iterations and then remains unchanged, even though the number of PIs grows further. This fact can be observed in Fig. 5.2 (thick line).

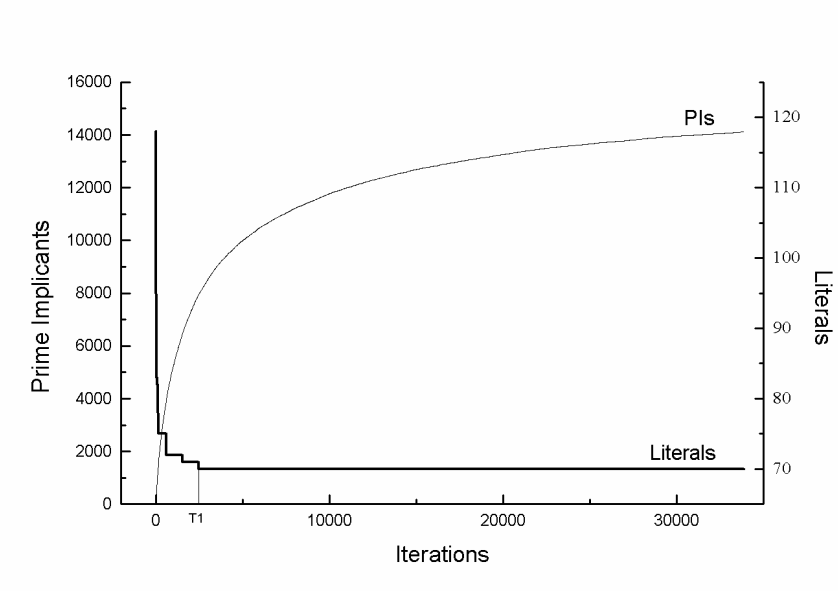

Figure 5.2: Growth of PI number and decrease of SOP length during iterative minimization

It is obvious from curves in Fig. 5.2 that selecting a suitable moment T1 for terminating the iterative process is of key importance for the efficiency of the minimization. The approximate position of the stopping point can be found by observing the relative change of the solution quality during several consecutive iterations. If the solution does not change during a certain number of iterations (e.g., twice as many iterations as were needed for the last improvement), the minimization is stopped. The amount of elapsed time may be used as an emergency exit for the case of unexpected problem size and complexity.

5.6.2 Accelerating the Iterative Minimization

When the CD-Search phase is being repeated, identical implicants are quite often generated in different iterations. These are then passed to the Implicant Expansion phase (see Subsection 5.7), which might be unnecessarily repeated. To prevent this, all implicants that were ever produced by the CD-Search are stored in the *I-buffer* (Implicant buffer). A diagram of the whole minimization algorithm for a multi-output function is shown in Fig. 5.3.

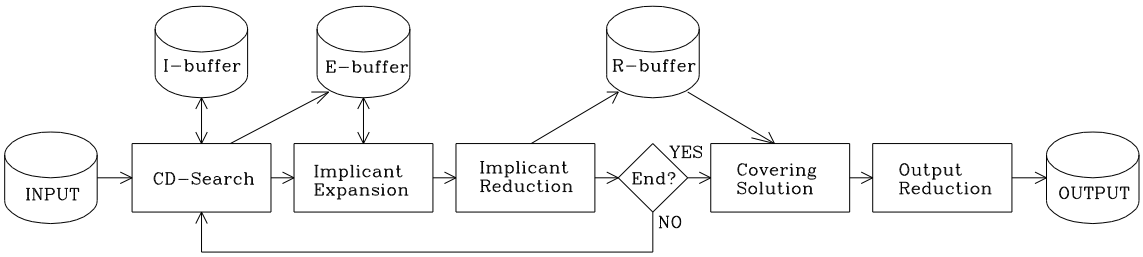

Figure 5.3: Iterative minimization schematic plan

Each newly generated implicant is first looked up in the I-buffer and, if it is already present, its further processing is stopped. Otherwise it is stored in both the I-buffer and E-buffer (Expansion buffer). The E-buffer serves as a storage of implicants that are candidates for expansion into PIs. After the expansion, they are removed from the E-buffer. Then they are reduced to group implicants and, after duplicity and dominance checks, the newly created group implicants are stored in the R-buffer (Reduced implicants buffer). Finally, the covering problem is solved using all the implicants from the R-buffer. There are separate I- and E-buffers for each output for multioutput functions. The R-buffer is common for all.

The main implementation requirement for the I-buffer is a high look-up speed, enhanced especially by early detection of an absence of a term. A novel structure, which was named a *tree buffer* is proposed here. The buffer is structured as a ternary tree with depth *n*. During the search in the tree, the direction at the *k*-th level is chosen according to the type of occurrence (0,-,1) of the *k*-th variable in the searched term. The presence of an implicant is represented by the existence of its corresponding leaf. The tree is dynamically constructed during the addition of implicants into the buffer. An example of such a tree for $n = 3$ is shown in Fig. 5.4. The buffer contains implicants 0-0, 10- and 11-. If, e.g., an implicant 0-1 is looked for, the search will fail in the node 0-, where no path leading to 0-1 is present.

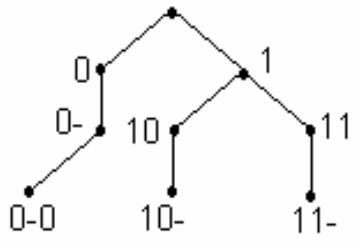

Figure 5.4: I-buffer tree structure

5.7 Implicant Expansion

As mentioned above, the implicants constructed during the CD-Search need not be prime. To reduce the number of implicants needed to cover all 1-terms of the given function, their dimension has to be increased by the implicant expansion (IE). The expansion is done by removing literals (variables) from the terms. When no literal can be removed from the term any more, we get a prime implicant (PI).

There are basically two problems to be solved in connection with implicant expansion. One of them is the mechanism that effectively checks whether a tentative literal removal is acceptable. The other is the selection of the literals and the order in which they are to be removed from the implicant term. First let us discuss the checking mechanism.

5.7.1 Checking the Removal of a Literal

Removing a variable from a term doubles the number of minterms covered by the term. The newly covered minterms may be 1-minterms or DC-minterms, but none of them should be a 0-minterm. In BOOM, individual literals are tried for removal and checked whether the expanded term does not intersect the off-set (therefore the DC terms need not be enumerated explicitly). If a non-empty intersection with a 0-term is found, the removal is rejected. The checking is done by a simple comparison of the term with all the off-set terms, thus in a linear time.

5.7.2 Expansion Strategy

The second problem to be solved is the selection strategy for the literals to be removed. The expansion of one implicant may yield several different prime implicants. To find them all, we have to try systematically to remove each literal, whereas the order of the literals selected plays an important role. Trying all possible combinations of literals to be removed is denoted as an *Exhaustive Implicant Expansion***.** Using recursion or queue, all possible literal removals are systematically tried until all primes are obtained. Unfortunately, the complexity of this algorithm is exponential with the number of input variables. Hence this method is usable only for problems with up to cca. 20 input variables.

Two local heuristic IE methods, differing in speed and quality of results are proposed here.

The simplest one, namely a *Sequential Expansion*, systematically tries to remove from each term all literals one by one, starting from a randomly chosen position. Each removal is checked against the off-set as above, but if the removal is successful, it is made permanent. If, on the contrary, some 0-minterm is covered, the literal is put back and the algorithm proceeds to the next one. After removing all possible literals one prime implicant covering the original term is obtained. This algorithm is greedy and we stay with one PI even if there is more than one PI that can be derived from the original implicant. The complexity of this algorithm is linear with the number of input variables and the number of processed terms.

A sequential expansion obviously cannot reduce the number of product terms, but it reduces the number of literals. The experimental results have shown that this reduction may reach approximately 25%.

With a *Multiple Sequential Expansion* all the possible starting positions are tried and each implicant thus may expand into several PIs. The upper bound of the number of PIs that can be produced from one implicant is n-d, where n is the number of input variables and d is the dimension of the original implicant. The complexity of this algorithm is $O(n.p)$, where p is the number of processed terms.

5.7.3 Evaluation of Expansion Strategies

The properties of the proposed IE methods and their influence on the minimization process (runtime and quality of the final solution) will be discussed in this subsection.

Fig. 5.5 shows the time of the minimization of a single-output function of 30 input variables and 500 defined minterms as a function of the number of iterations. The growth for the sequential expansion is linear, which means that an equal time is needed for each iteration. The time needed for the multiple sequential expansion and the exhaustive expansion grows faster at the beginning and then turns to linear with a slower growth. At this point the CD-Search no longer produces new implicants and thus the IE and the following phases are not executed. This causes simple sequential expansion, which is seemingly the fastest one, to become the slowest one after a certain number of iterations.

Fig. 5.6 illustrates the growth of the PI set as a function of time. We can see that the Sequential Expansion achieves the lowest values, although it is the fastest implicant expansion method. This is because it cannot take advantage of the I-buffer. The implicants are repeatedly expanded, even when they have already been expanded in all possible ways. The two other methods achieve higher values, because they put an implicant into the E-buffer only once and then they are blocked by the I-buffer. Hence when the same implicants are generated repeatedly by the CD-Search, they are not processed any more, which speeds up the whole minimization. We can see that the most complex method, namely exhaustive expansion, produces PIs at the fastest rate.

Practice shows that the more complex IE methods are advantageous for functions with large care sets, where the number of implicants in the final solution is big, while the simplest sequential expansion is better for very sparse functions.

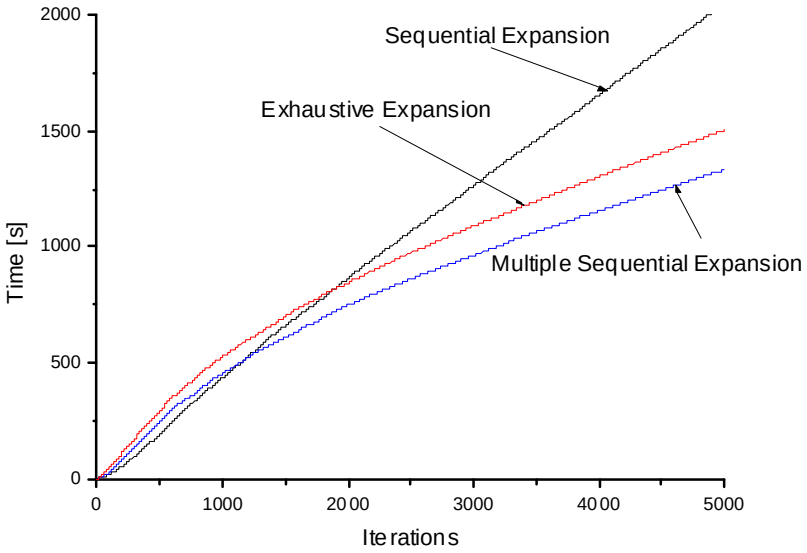

Figure 5.5: Growth of time for different IE methods

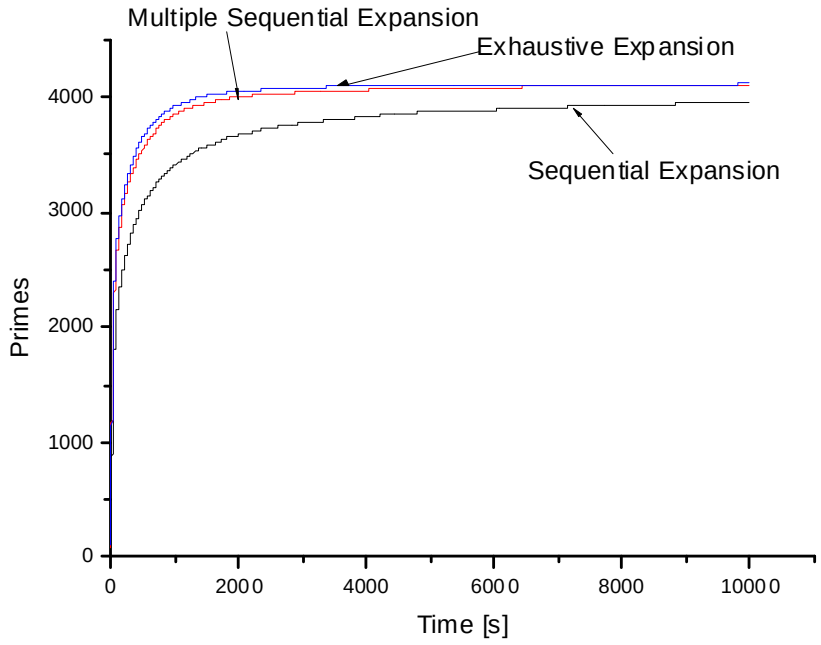

Figure 5.6: Growth of PIs for different IE methods

5.8 Minimizing Multi-Output Functions

To minimize multi-output functions, only a few modifications of the basic single-output algorithm must be done.

At the beginning, each of the output functions F_i is treated separately, and the CD-Search and IE phases are performed. After that, a set of primes sufficient for covering all *v* functions is obtained. However, for obtaining the (theoretically) minimal solution we may need implicants of more than one output function that are not primes of any F_i. Here the next part of the minimization – *Implicant Reduction* (IR) finds its place. After the IR phase is performed, the group covering problem is solved. Its solution is a set of implicants needed to cover each of the output functions F_1 ... F_m. These implicants are assigned to the individual output functions, so they do not intersect the functions' off-sets. However, to generate the required output values, some of these implicants may not be necessary. These implicants would create redundant inputs into the output OR gates (when implemented as a two-level AND-OR network). Sometimes this is harmless (e.g. in PLAs), moreover it could prevent hazards. Nevertheless, for hardware-independent minimization the redundant outputs should be removed. This is done at the end of the minimization by solving *v* covering problems for all *v* functions independently. This phase corresponds to ESPRESSO's MAKE_SPARSE procedure [Bra84].

5.9 Implicant Reduction (IR)

All the obtained prime implicants are tried for reduction by adding literals to them, in order to become implicants of more than one output functions. The method of implicant reduction is similar to CD-Search. Literals are sequentially being added to the previously obtained implicants until there is no chance that the implicant will be used for more output functions. Preferably, literals that prevent intersecting with most of the terms of the off-sets of all F_1 ... F_m (i.e., yielding reduced terms that cover the least zeros in all the functions) are selected. When no further reduction leads to any possible improvement, the reduction is stopped and the term is recorded. A term that no longer intersects with the off-set of any F_i becomes its implicant. All implicants that were ever found are stored and output functions are assigned to them – it is checked for each term produced, what output functions from F_i it is an implicant of.

Then simple dominance checks are performed in order to eliminate implicants that are dominated by other implicants. Fig. 5.7 shows the typical growth of the number of group implicants (non-primes) as a function of the number of iterations. Here the function of 13 input variables, 13 output variables and 200 defined terms was used for demonstration. We can see that the number of the reduced implicants first grows rapidly, but then it falls to approx. 15% of the maximum value. This is due the fact that new prime implicants are being constantly produced and they absorb most of the previously generated group implicants in the preliminary dominance checks.

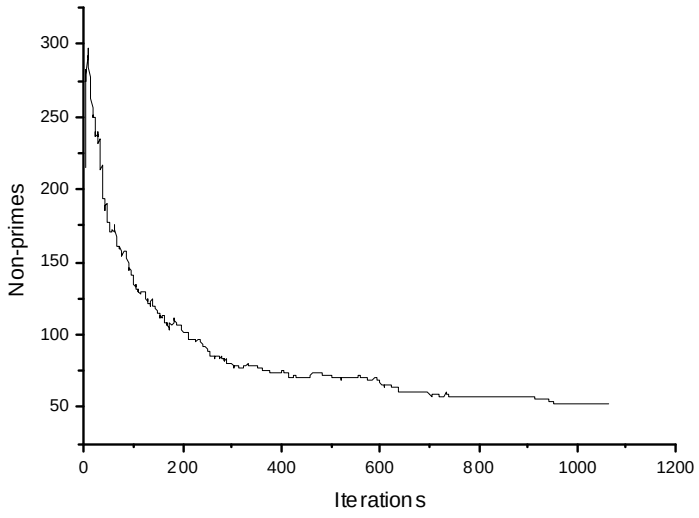

Figure 5.7: Growth and fall of the number of non-primes

5.10 Solution of the Covering Problem

It was shown in subsection 5.6.1 that even a small subset of implicants may give the minimum solution. However, the quality of the final solution strongly depends on the covering problem (CP) solution algorithm. With a large number of implicants it is impossible to obtain an exact solution, since the covering problem is NP-hard. Thus some kind of a heuristic must be used.

Moreover, a large number of implicants may sometimes misguide the CP solution algorithm and thereby lead to a solution, which is even worse than the solution obtained using only few implicants.

An exact CP solution is mostly rather time-consuming, especially when it is performed after several iterations during which many implicants had accumulated. In this case, a heuristic approach is the only possible solution. Out of several possible approaches two has been used in BOOM. The first one, denoted as *LCMC* (Least Covered, Most Covering) is a common greedy heuristic algorithm for solution of the covering problem. The implicants covering minterms covered by the least number of other implicants are preferred. If there are more than one such implicants, implicants covering the highest number of yet uncovered 1-minterms are selected.

More sophisticated heuristic methods for CP solution are based on computing the contributions (scoring functions) of terms as a criterion for their inclusion into the solution [Ser75, Rud89, Cou94]. Such a method is also used in BOOM as is has been found very efficient and not too time-consuming.

Here the covering matrix $\boldsymbol{A}$ is constructed, its dimension will be denoted as (e, f). The columns correspond to the implicants, rows to the on-set terms that are to be covered. $\boldsymbol{A}[i, j] = 1$ if the implicant j covers the on-set term i, $A[i, j] = 0$ otherwise. For each row its *strength of coverage* is computed as

$$\mathrm{SC}(x_i) = \frac{1}{\sum_{j=1}^{f} A[i, j]} \tag{1}$$

Then the *column contribution* is computed for each column:

$$\mathrm{CC}(y_j) = \sum_{i=1}^{e} A[i, j] \cdot SC(x_i) \tag{2}$$

The implicant (column) with the maximum contribution value is selected into the solution, the contribution values are recomputed and the process is repeated until the whole on-set is covered.

5.11 The BOOM Algorithm

The iterative minimization algorithm for a group of functions $\boldsymbol{F}_i$ (i = 1, 2,…, v) can be described by the following pseudo-code. The inputs are the on-sets F_i and off-sets R_i of the v functions, and the output is a minimized disjunctive form G = (G_1, G_2,...G_v), where G_j stands for a particular implicant.

Algorithm 5.2: Minimization of a group of functions

```
BOOM(F[1..v], R[1..v]) {
  G = ∅;
  do
    I = ∅;
    for (i = 1; i <= v; i++)
        I′ = CD_Search(F[i], R[i]);
        Expand(I′, R[i]);
        Reduce(I′, R[1..v]);
        I = I ∪ I′;
        DominanceCheck(I);
    G′ = Group_cover(I, F[1..v]);
    Reduce_output(G′, F[1..v]);
    if (Better(G′, G)) then G = G′;
  until (stop);
  return G;
}
```

5.12 BOOM Experimental Results

Extensive experimental work has been done to evaluate the efficiency of the proposed algorithm, especially for problems of large dimensions. Both the runtime in seconds and result quality were evaluated. The quality of the results was measured by three parameters: total number of literals, output cost and number of product terms (implicants). One of them or the combination of two (sum of the number of literals and output cost) had always to be chosen as a minimization criterion. The results of the experiments are listed in the following subsections. All the experiments were performed on a standard PC with a 900 MHz Athlon processor and 256 MB RAM.

5.12.1 Standard MCNC Benchmarks

A set of 139 standard MCNC benchmarks [Yan91, MCNC] was solved by ESPRESSO v2.3, ESPRESSO-EXACT and BOOM. As the benchmark functions were specified by the on set and don't care set (type fd PLA), the source files had to be converted into the format where the on-set and off-set are defined (type fr PLA). This was done by ESPRESSO (using the -ofr switch) before the minimization was performed. Thus, both ESPRESSO and BOOM could use the same input files. The presented runtimes do not include the time needed for the conversion.

Of all the 139 standard problems, 67 (48.2%) were solved by BOOM in a shorter time than by ESPRESSO. In all cases only one iteration of BOOM was used, and thus the results may not be optimal. However, this option was chosen in order to have a better comparison of the results. In 52 cases (37.4%) BOOM gave the same result as ESPRESSO, while 30 (57.7%) of these equal results were reached faster than by ESPRESSO.

Table 5.6 shows the minimization results of a selected set of "harder" benchmarks. The benchmarks were also solved by ESPRESSO-EXACT in order to obtain the minimum solution for comparison. Note that in this case the minimality criterion is the number of terms only and thus some "exact" solutions are even worse than those reached by ESPRESSO or BOOM. Some benchmarks were not solved by ESPRESSO-EXACT because of its extremely long runtimes (blank entries

in Tab. 5.6). ESPRESSO solutions that are equal to the exact ones are shaded in the ESPRESSO column. The column *i/o/p* describe the number of input/output variables and care terms of a particular benchmark, the *time* columns indicate the computational time in seconds, the *lit/out/terms* columns show the quality of the results, i.e., the number of literals in the final SOP form, the output cost and the number of terms. The shadowed cells indicate that the benchmark was solved by BOOM in a shorter time than by ESPRESSO, or the same result was reached respectively.

Since the MCNC benchmark circuits mostly have a relatively low number of inputs and many care terms defined, the features of BOOM couldn't be fully exploited here. Thus, the results are not that promising, when compared with ESPRESSO. However, BOOM is much more efficient for more complex problems (see the following Subsections), which ESPRESSO often cannot solve in a reasonable time at all.

Table 5.6: Runtimes and minimum solutions for the standard MCNC benchmarks

		ESPRESSO		ESPRESSO-EXACT		BOOM – 1it.	
bench	*i/o/p*	*time*	*lit/out/terms*	*time*	*lit/out/terms*	*time*	*lit/out/terms*
alu2	10/8/241	0.07	268/79/68	0.18	268/79/68	0.02	268/79/68
alu3	10/8/273	0.08	279/70/65	0.19	278/74/64	0.02	279/68/66
alu4	14/8/1184	0.59	4445/644/575	12.24	4495/648/575	1.02	4449/636/577
b9	16/5/292	0.08	754/119/119	0.89	754/119/119	0.09	754/119/119
br1	12/8/107	0.05	206/48/19	0.07	206/48/19	0.02	215/45/20
br2	12/8/83	0.06	134/38/13	0.07	134/38/13	0.01	134/38/13
chkn	29/7/370	0.14	1598/141/140	0.25	1602/142/140	0.41	1598/141/140
cordic	23/2/2105	1.86	13825/914/914	3.59	13843/914/914	4.05	13825/914/914
ex4	128/28/654	0.62	1649/279/279			14.01	1649/279/279
e64	65/65/327	0.11	2145/65/65	0.11	2145/65/65	15.06	2145/65/65
exep	30/63/643	0.17	1175/110/110	0.55	1170/108/108	3.66	1175/110/110
ibm	48/17/499	0.11	882/173/173			0.82	882/173/173
mark1	20/31/72	0.25	97/57/19	1.45	97/57/19	0.04	93/46/23
misex2	25/18/101	0.07	183/30/28	0.06	183/30/28	0.10	183/30/28
misex3c	14/14/1566	0.98	1306/253/197			0.59	1335/242/209
misj	35/14/55	0.07	54/48/35			0.03	54/48/35
shift	19/16/200	0.07	388/105/100			0.06	388/105/100
spla	16/46/837	0.71	2558/643/251	6.65	1564/450/181	1.54	2821/517/285
vg2	25/8/304	0.08	804/110/110	0.54	804/110/110	0.15	804/110/110
x9dn	27/7/315	0.08	1138/120/120	0.49	1138/120/120	0.22	1138/120/120

5.12.2 Test Problems Having $n \geq 50$

The MCNC benchmarks have relatively few defined terms, few input variables (only for 9 standard benchmarks n exceeds 50) and also a small number of don't care terms. To compare the performance and result quality achieved by the minimization programs on larger problems, a set of problems having up to 200 input variables and up to 200 terms was solved. In order to accomplish this, a set of artificial benchmark problems is proposed, which was named as BOOM Benchmarks [Fis02, BOOMBench]. The truth tables of these problems were generated by a random number generator, for which only the number of input variables and the number of care terms were specified. The number of outputs was set equal to 5, and the input matrix contained 20% of don't cares. The on-set and off-set of each function were kept approximately of the same size. Such a type of problems usually occurs in the Output Decoder design as well, with an exception of input don't cares present, since deterministic PRPG are considered.

However, introducing don't cares makes the problem only harder to solve. On the other hand, the distribution of zeros and ones in the input matrix is pseudorandom (generated by a PRPG), thus a random distribution nears to practical problems very well. Moreover, the randomness of the benchmarks used here was chosen in order to have functions with no special properties. This allows us to determine more easily the properties and scalability of the algorithms.

For each problem size (number of variables, number of terms) in Tab. 5.7 and 5.8, ten different samples were generated and solved and average values of the ten solutions were computed.

First the minimality of the result was compared. BOOM was always run iteratively, using *the same total runtime* as ESPRESSO needed to obtain the solution. In the following three tables, the number of input variables *n* increases horizontally and the number of input terms *p* is increased vertically. The first row of each cell in Tab. 5.7 contains the BOOM results, the second row shows the ESPRESSO results. The quality criterion selected for BOOM was the sum of the number of literals and the output cost, which approximates the gate equivalents (GEs) [DeM94]. We can see that for all but one problem size (shaded cell) BOOM found a better solution than ESPRESSO.

Table 5.7: Solution of Boom Benchmarks - comparing the result quality

p / n	50	100	150	200
50	110/41/25 (58) 122/54/27/3.89	96/35/23 (90) 104/45/23/10.29	90/32/21 (147) 92/41/21/24.87	84/29/20 (199) 89/39/20/41.99
100	284/86/52 (46) 289/104/51/19.31	229/68/42 (94) 231/84/42/77.07	217/61/40 (140) 213/80/39/199.17	207/57/38 (140) 201/74/37/246.21
150	474/132/76 (43) 481/158/76/54.76	389/101/63 (101) 384/125/62/282.80	362/92/61 (116) 345/113/56/646.20	381/90/64 (64) 322/107/52/1066.14
200	678/177/101 (51) 686/209/101/162.62	553/137/83 (116) 539/165/81/730.91	492/125/75 (207) 480/149/72/1913.65	469/110/71 (277) 450/136/68/3372.66

Entry format: BOOM: # of literals / output cost / # of implicants (# of iterations)
ESPRESSO: # of literals / output cost / # of implicants / time in seconds

A second group of experiments for $n \geq 50$ was performed to compare the runtimes. Again, the randomly generated problems from [Fis02, BOOMBench] were solved, but this time BOOM was running until a solution of *the same or better quality* as ESPRESSO was reached. The quality criterion selected was again the sum of the number of literals and the output cost. The results given in Tab. 5.8 show that for all samples the same or better solution was found by BOOM in a shorter time than by ESPRESSO.

Table 5.8: Solution of Boom Benchmarks - comparing the runtime

p / n	50	100	150	200
50	170/0,64 (12) 176/3,89	145/1,89 (21) 149/10,29	131/14,52 (73) 133/24,87	126/3,26 (25) 128/41,99
100	388/7,15 (23) 393/19,31	313/25,5 (48) 315/77,07	291/38,91 (56) 293/199,17	273/86,51 (83) 275/246,21
150	631/20,38 (25) 639/54,76	506/153,84 (70) 509/282,8	456/374,68 (105) 458/646,20	427/974,40 (161) 429/1066,14
200	890/71,97 (31) 895/162,62	697/467,63 (86) 704/730,91	625/1026,28 (149) 629/1913,65	582/1759,27 (220) 586/3372,66

Entry format: BOOM: # of literals+output cost / time in seconds (# of iterations)

ESPRESSO: # of literals+output cost / time in seconds

5.12.3 Solution of Very Large Problems

The third group of experiments aims at establishing the limits of applicability of BOOM. For this purpose, a set of very large test problems was generated and solved by BOOM. Each problem was a single-output function in this case. For problems with more than 200 input variables ESPRESSO could not be used, because of prohibitively long the runtimes (several hours). Hence when investigating the limits of applicability of BOOM, it was not possible to verify the results by any other method. The results of this test are listed in Tab. 5.9 where the average time in seconds needed to complete one iteration for various problem sizes is shown. We can see that a problem with 1000 input variables and 2000 care minterms was solved by BOOM in about 20 seconds.

Table 5.9: Time for one iteration on very large problems

p / n	200	400	600	800	1000
200	0.06	0.11	0.17	0.26	0.26
400	0.25	0.34	0.52	0.77	0.88
600	0.45	0.80	1.15	1.44	1.96
800	0.88	1.43	2.05	2.69	3.35
1000	1.32	2.10	3.07	4.21	4.42
1200	1.91	3.28	4.69	6.30	7.27
1400	2.69	4.48	6.04	7.72	8.96
1600	3.56	5.78	8.58	10.57	11.69
1800	4.51	7.73	10.56	12.52	16.34
2000	5.64	10.02	13.17	17.45	20.17

5.12.4 Column-Matching Practical Examples

In order to fully justify the need for BOOM in the proposed BIST synthesis method (Column-Matching), several Output Decoder design examples are presented in this Subsection. Particular decoders were minimized both by BOOM and ESPRESSO and the result quality and runtime compared. The results are shown in Table 5.10.

In all the cases BOOM was run for 100 iterations. The "*Bench*" column indicates the name of the benchmark circuit, which is the Output Decoder to be synthesized for a particular benchmark (which name follows after "d_"). For most of benchmark circuits the Column-Matching procedure was run with different parameters set (namely lengths of the phases), yielding different decoder complexities. In the "*i/o/p*" column are listed the numbers of its inputs, outputs and defined terms. Then, the results obtained by BOOM and ESPRESSO are shown. The resulting Decoder complexity is given in terms of gate equivalents [DeM94]. The shadowed cells indicate cases where BOOM outperformed ESPRESSO, both for the result quality and runtime.

Table 5.10: Output Decoder design examples

		BOOM		ESPRESSO	
Bench	*i/o/p*	*GEs*	*Time [s]*	*GEs*	*Time [s]*
d_c1355 (1)	41/18/13	31.5	0.69	37.0	0.19
d_c1355 (2)	41/21/14	35.0	0.74	42.5	0.26
d_c1908	33/3/29	17.0	0.49	18.0	0.12
d_c2670 (1)	233/32/60	113.0	165.95	313.0	4838.62
d_c2670 (2)	233/31/52	61.0	159.06	260.0	2329.44
d_c2670 (3)	233/36/104	159.5	740.18	344.5	24 710.07
d_c7552 (1)	207/48/81	196.5	807.84	373.0	27 574.93
d_c7552 (2)	207/72/207	389.5	23 933.46	-	> 24 h
d_s420.1 (1)	34/6/42	22.0	0.75	26.0	1.58
d_s420.1 (2)	34/5/33	19.5	0.75	24.5	0.95
d_s838 (1)	67/24/61	35.5	3.15	58.5	27.94
d_s838 (2)	67/15/46	29.0	1.65	44.0	14.94
d_s953 (1)	45/2/25	4.5	0.13	7.0	0.11
d_s953 (2)	45/4/45	10.5	0.42	10.5	0.16
d_s1196	32/4/48	29.5	2.12	37.0	1.04
d_s1238 (1)	32/5/60	46.5	9.71	68.5	3.15
d_s1238 (2)	32/4/58	28.5	5.23	36.5	0.53
d_s5378 (1)	214/3/36	12.0	2.81	15.5	6.58
d_s5378 (2)	214/2/22	7.0	0.66	7.0	1.70
d_s9234 (1)	247/77/216	655.0	18835.6	-	> 24 h
d_s9234 (2)	247/38/99	186.5	266.78	252.5	17 298.0
d_s9234 (3)	247/23/52	64.5	29.09	108.0	659.25
d_s13207.1 (1)	700/8/96	88.5	93.65	95.5	1251.0
d_s13207.1 (2)	700/58/197	293.5	1550.25	316.5	190 038.74
d_ s15850.1 (1)	611/96/313	197.5	3416.4	-	> 24 h
d_ s15850.1 (2)	611/48/180	78.5	516.3	120.0	37 818.65
d_s38417	1664/1454/520	759.0	1923.0	-	> 24 h
d_s38584.1 (1)	1664/464/307	158.0	321.9	-	> 24 h
d_s38584.1 (2)	1664/464/45	11.0	46.6	31.0	20 361.71
d_b04 (1)	77/9/37	24.5	1.75	28.5	4.19
d_b04 (2)	77/4/29	9.0	0.32	11.0	0.58
d_b05 (1)	35/7/33	28.5	2.94	49.0	0.85
d_b05 (2)	35/2/15	4.0	0.07	4.5	0.06
d_b07 (1)	50/5/41	11.5	2.01	12.0	3.43
d_b07 (2)	50/1/24	1.0	0.02	1.0	0.8

		BOOM		ESPRESSO	
Bench	*i/o/p*	*GEs*	*Time [s]*	*GEs*	*Time [s]*
d_b12 (1)	126/11/128	95.0	118.14	118.0	379.93
d_b12 (2)	126/7/66	45.5	15.52	57.5	18.27

It can be seen that BOOM outperformed ESPRESSO in the result quality in all the cases and mostly in the runtime as well. In some more complex cases ESPRESSO did not return a result in more than one day, thus the measurement was terminated.

5.13 Time Complexity Evaluation

As for most heuristic and iterative algorithms, it is difficult to evaluate the time complexity of the proposed algorithm analytically. A vast experimental evaluation has been therefore performed.

5.13.1 Influence of the Problem Size

The average time needed to complete one iteration of BOOM for various sizes of the input truth table was measured here. The number of experiments of each instance size was 10. The truth tables were generated randomly, following the same rules as in paragraphs 5.12.2 and 5.12.3. Fig. 5.8 shows the growth of an average runtime as a function of the number of care minterms (20-300) where the number of input variables is changed as a parameter (20-300). The curves in Fig. 5.8 can be approximated with the square of the number of care minterms.

The minimization time thus grows relatively rapidly with the number of care terms, however, it is not exponential. This fact complicates the minimization of functions with a large number of defined terms to some extent. Hence, BOOM is more suitable for minimizing very sparse functions, where the number of care terms is low.

Fig. 5.9 shows the runtime growth as the function of the number of input variables (20-300) for varying number of defined minterms (20-300). Although there are some fluctuations, the time complexity is almost linear. Fig. 5.10 shows a three-dimensional representation of the above curves.

The fact that the time complexity grows *linearly* with the number of input variables (while keeping the number of defined terms) expresses the main advantage of the BOOM algorithm. As the size of the Boolean space of the function grows exponentially with the number of input variables, the time complexity of most of the common minimization algorithms grows exponentially too. In BOOM there is no chance for an exponential time grow, as there are no algorithms with an exponential complexity used in BOOM (except of the case when the exhaustive IE is used). This allows to minimize functions with an extremely large number of inputs very efficiently.

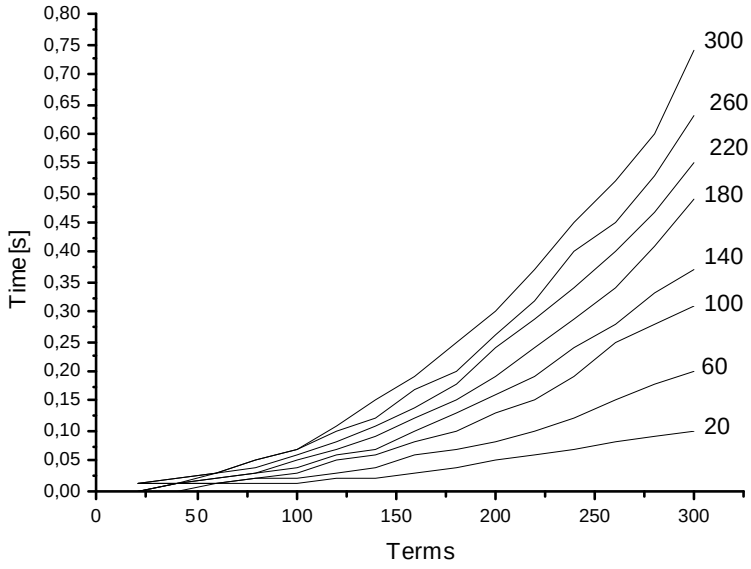

Figure 5.8: Time complexity (1)

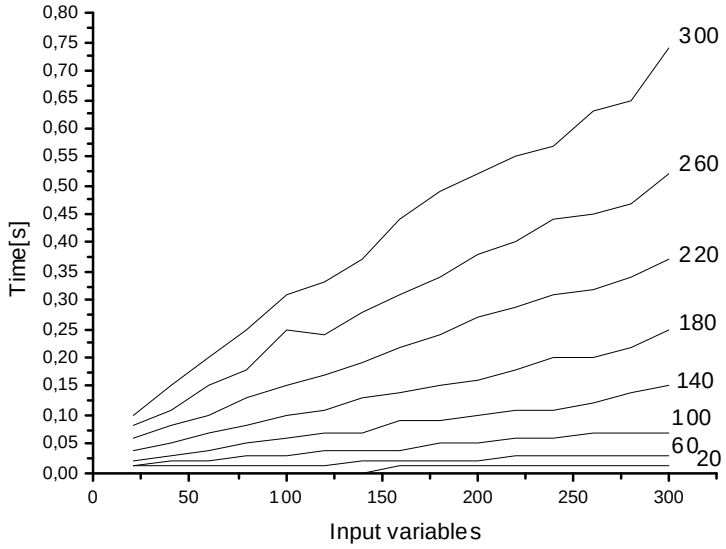

Figure 5.9: Time complexity (2)

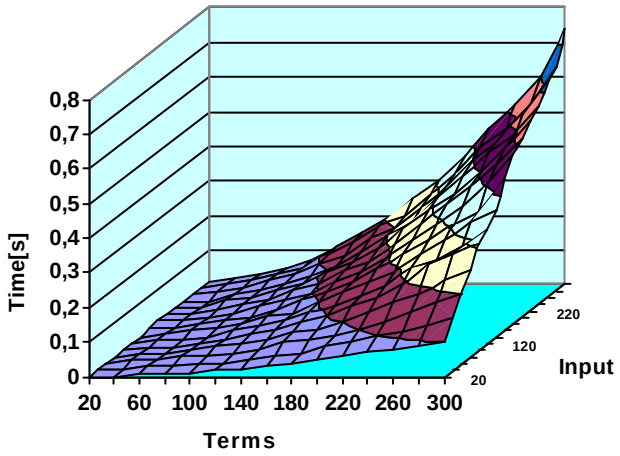

Figure 5.10: Time complexity (3)

5.13.2 Influence of Don't Cares

The influence of the number of don't care states in the field of input variables on the runtime was studied on a set of problems generated by a random number generator for n = 20, 50 and 80, with 5 output variables and 200 defined terms. The percentage of don't cares was varied in the range from 0 to 35%. Here 0% denotes the situation when only minterms were used in the function definition. At the other end, 35% of don't cares means that slightly more than one third of all values of the input variables were undefined.

The growth of the runtime for ESPRESSO and for BOOM is shown in Fig. 5.11, where the number of input variables is indicated in parentheses. We can see that although ESPRESSO runtime grows to 5000 s for 80 input variables, BOOM runtime remains almost constant within the used scale for all problem sizes. The influence on the runtime is visualized even more clearly in Fig. 5.12, showing the relative slowdown of BOOM and of ESPRESSO caused by the don't cares. We can see that the relative slowdown of BOOM for the highest percentage of don't cares is about 7.5, whereas for ESPRESSO it is up to 100.

We can conclude from this observation that ESPRESSO is extremely sensitive to the dimensionality of the source terms; the minimization time grows rapidly with the growing number of input don’t cares. On the other hand, BOOM is almost insensitive to the dimension of the terms. Thus, BOOM can be efficiently used to minimize functions with a large portion of don’t cares in the source terms.

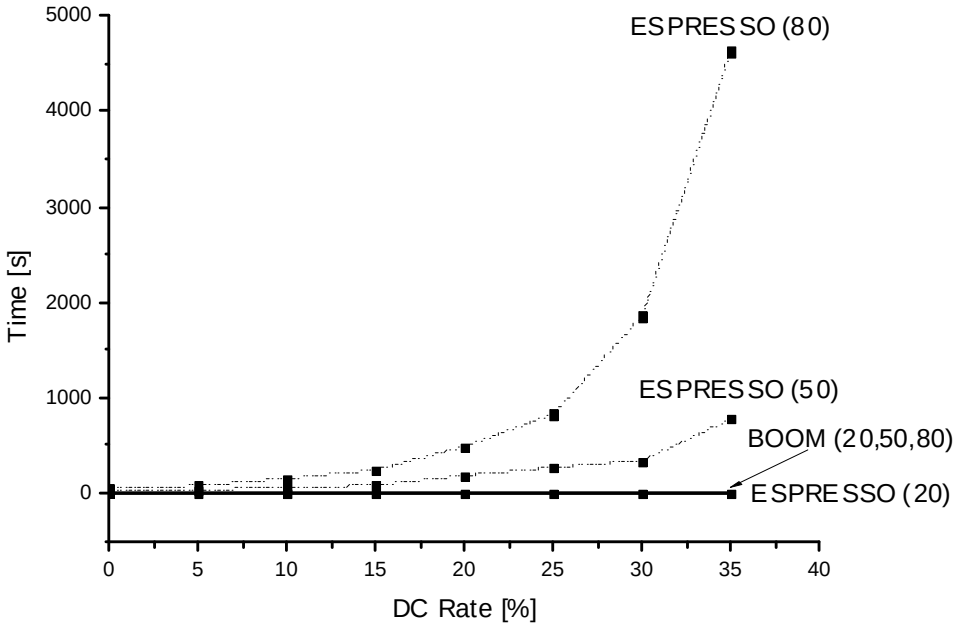

Figure 5.11: Runtimes for ESPRESSO (dashed lines) and for BOOM (solid line)

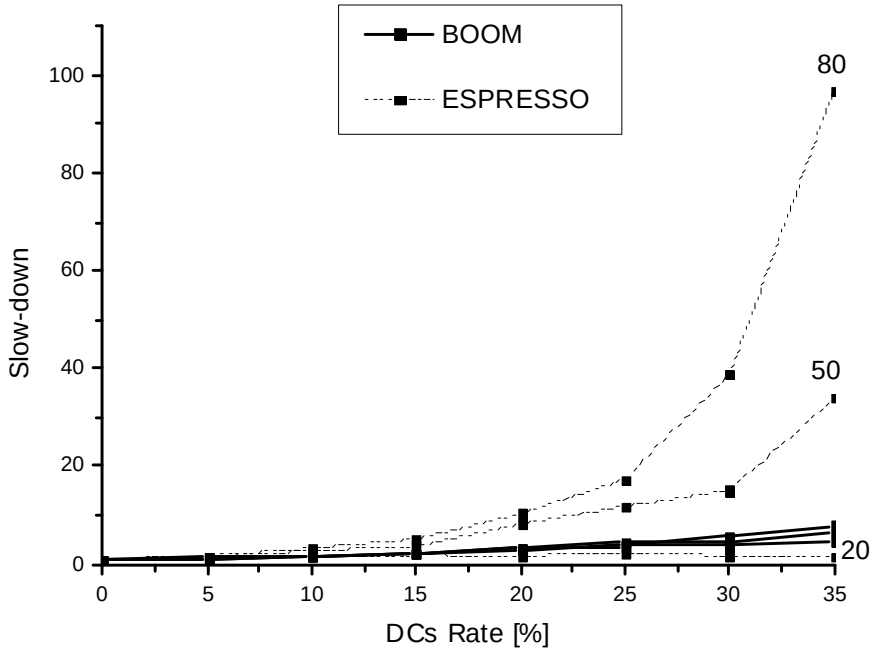

Figure 5.12: Relative slowdown [%] for various percentages of DCs

5.14 BOOM Conclusions

The original Boolean minimization method, based on a new approach to implicant generation has been proposed. Its most important features are its applicability to functions with several hundreds or thousands of input variables and very short minimization times for sparse functions. The function to be minimized is defined by its on-set and off-set (which may consist of minterms and terms of higher dimensions), whereas the don't care set needs not be specified explicitly. Properties of the BOOM minimization tool were demonstrated by many examples. Its application is advantageous above all for functions with a large number of input variables and a large number of don't care states where it beats other methods, like ESPRESSO, as concern to the quality of the results and in method runtime, too. The PI generation method is very fast and can easily be used by an iterative manner. Extensive tests on different benchmarks (MCNC, randomly generated problems, practical Column-Matching problems) were performed in order to determine the strengths and weaknesses of the BOOM system.

The dimension of the problems that can be solved by BOOM can reach thousands input variables, since the runtime grows linearly with the number of input variables. For problems of very high dimension, the success largely depends on the size of the care set, because the runtime grows roughly with the square of its size.

BOOM is applicable very efficiently to the design of the Output Decoder needed in the Column-Matching BIST design process. It overpowers ESPRESSO in general. ESPRESSO is often completely unusable, due to its prohibitively long runtime for problems having a large number of inputs.

6 Implementation

6.1 Implementation of Column-Matching BIST Equipment Design Method

In order to accomplish the proposed BIST equipment synthesis process, several programs had to be created. First of all, the ***Colmatch*** tool has been developed [Colmatch]. It implements the very Column-Matching algorithms. The program takes two matrices as an input – the C-matrix and the T-matrix. The output of the program is a PLA file describing the Output Decoder, and a report file containing information on the matched CUT inputs. Using this information it is possible to design the BISTE circuitry.

A simple program ***LFSR*** generating code words of a given LFSR (i.e., the **C**-matrix) has been made, to accompany *ColMatch*.

Then, additional tool has been programmed (***CMBIST***), producing the VHDL code of the whole circuit, together with the BIST equipment. The input to this algorithm is the circuit-under-test and the information obtained by *Colmatch*, the output is a synthesizable VHDL code of the circuit with BIST. This implies the scan-chain insertion (or inserting MUXes into the circuit, respectively), converting a sequential circuit into a combinational one (by introducing pseudo-primary inputs and outputs), the BIST controller design and, last but not least, the top-entity design, describing the whole circuitry.

The whole BIST design process is then described by the dataflow diagram shown in Fig. 6.1.

First, the sequential description of the CUT (in ISCAS'89 format [Brg89]) is converted into a combinational one. Simultaneously, VHDL description of the circuit, including multiplexers, is generated. Then the LFSR seed and polynomial are generated and the C-Matrix produced. These pseudo-random vectors are simulated by FSIM and a test for undetected faults is produced by Atalanta (the T-Matrix). Then the Column-Matching process is run, using the two matrices. The Output Decoder is minimized by BOOM and the resulting data are put together to form a synthesizable VHDL code describing the whole self-testable circuit.

The block schematic of the whole BIST, generated by ***CMBIST*** is shown in Fig. 6.2.

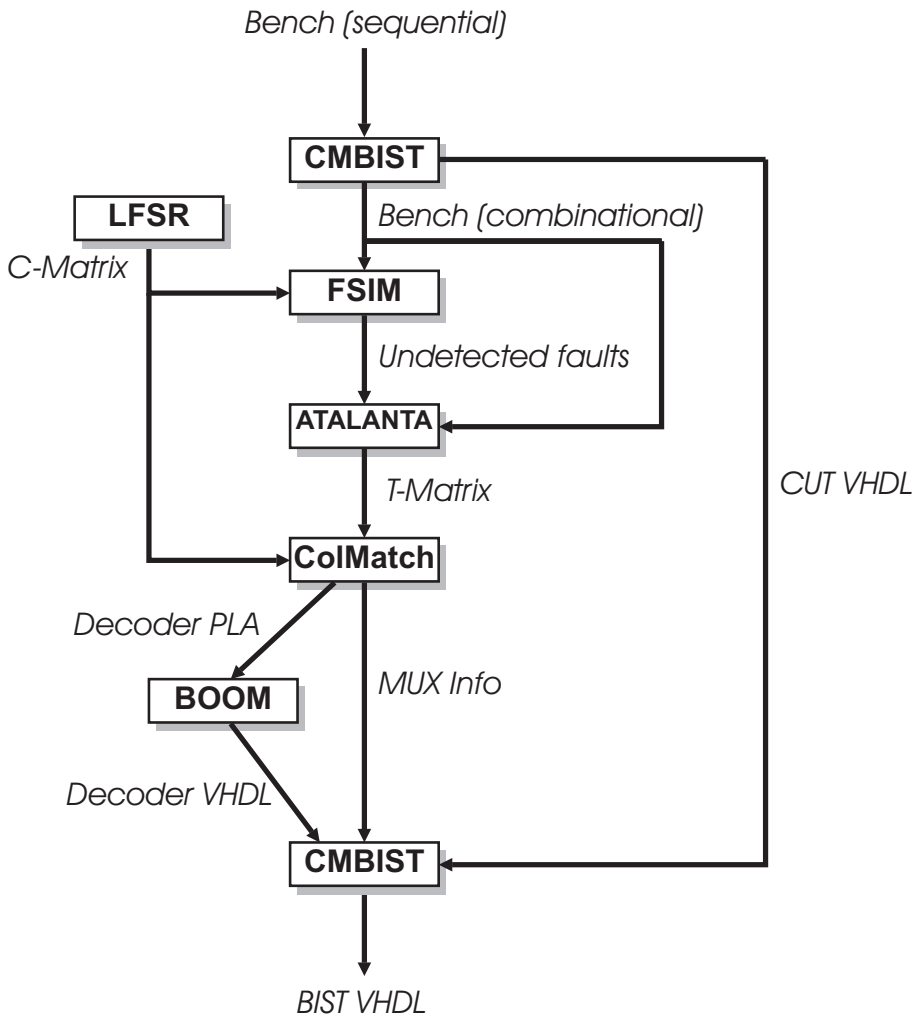

Figure 6.1: BIST equipment synthesis dataflow

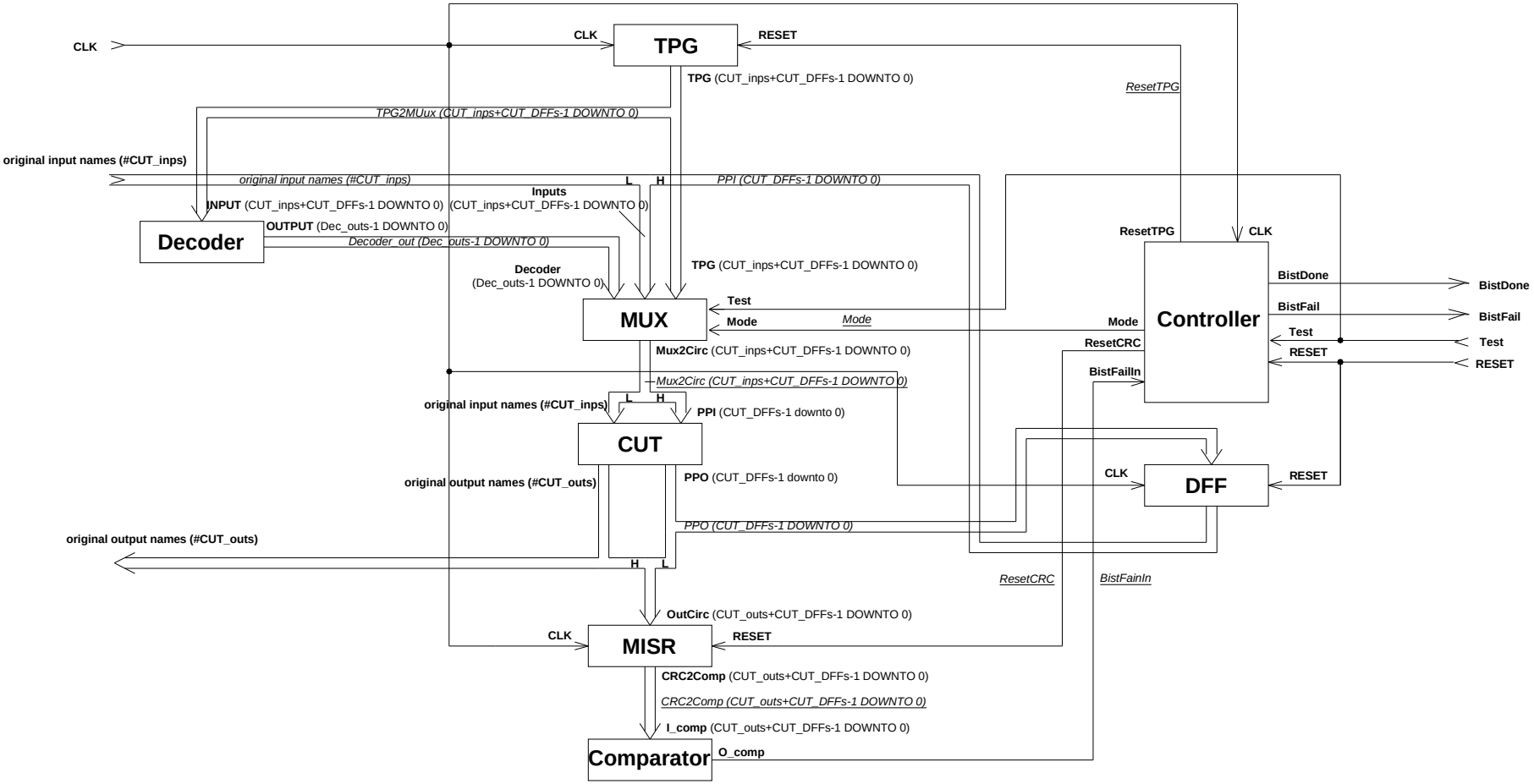

Figure 6.2: BIST structure

6.2 Implementation of BOOM

BOOM has been implemented as a stand-alone tool and released for publics in 2003, for the first time [BOOM]. Since that time is has overcame many improvements. Nowadays (version 2.8), it supports these features:

- PLA, *type fr* input (on-set and off-set specified), *type f* output (on-set specified)
- Output in: PLA, BLIF, VHDL, Verilog, Equations, HTML table
- Very fast minimization of functions having many input and output variables
- Low memory demands
- Adjustable tradeoff between runtime and result quality
- Many different optimization criteria (terms, literals, output cost, GEs, …)
- Windows and Linux compatible versions are available

7 Conclusions and Future Work

A mixed-mode BIST equipment design method based on a newly developed *Column-Matching* principle has been proposed. Here pseudorandom LFSR code words are being transformed into deterministic test patterns computed by an ATPG tool. The transformation is being done by a purely combinational block; no additional registers are needed to perform the transformation. The algorithm tries to "match" maximum of decoder outputs with its outputs, which yields no logic necessary to implement these outputs. The Thesis primarily describes the method of a design of the test pattern generator producing test vectors for the tested circuit.

The method is primarily designed for a test-per-clock BIST, however it can be easily adopted to test-per-scan for full-scan or multiple-scan circuits.

The pseudo-random and deterministic phases are separated, which enables to reach less area overhead of the control logic. 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.

The test is divided into two phases, the pseudo-random and deterministic one. The lengths of both phases can 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. This issue is discussed in this Thesis as well.

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

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

The BIST synthesis method may be used for almost any fault model, if a proper fault simulator and ATPG tool are provided. The fault coverage reached depends only on the ATPG tool as well; a trade-off between the fault coverage and BIST area overhead may be adjusted too.

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

The principles of the Column-Matching method were published in [Fis02a, Fis03a, Fis04a, Fis04b, Fis04c, Fis04d, Fis05a, Fis05b, Fis05d and Fis06a].

Not only the Column-Matching algorithm has been considered during the research. The "*Coverage-Directed Assignment (CD-A)*" algorithm has been developed as an alternative to the Column-Matching. The aim of this algorithm is the same as in Column Matching is, i.e., to transform an excessive set of pseudo-random vectors into a set of deterministic vectors. However, the principles of method are completely different. It is based on a generalization of Boolean minimization process. The results were published in [Fis03d and Fis03f]. However, this approach has been abandoned after some time, due to prohibitively high time complexity of the CD-A algorithm.

A novel Boolean minimizer BOOM has been developed as a necessary part of the BIST test pattern generator synthesis process, since standard minimizers, like

ESPRESSO, were not able to handle functions having hundreds of inputs in a reasonable time. The minimization method is based on a completely different approach to implicant generation: instead of processing the on-set terms and trying to increase their size, the implicants are generated by reducing a universal hypercube, until it becomes an implicant of the source function. The on-set provided serves just as a "guideline" to implicant generation, the on-set terms are not processed explicitly.

This idea yielded an efficient heuristic two-level Boolean minimizer capable to minimize functions having up to thousands of inputs in an acceptable time.

BOOM was tested on standard MCNC benchmarks, randomly generated benchmarks and circuits that were generated by the Column-Matching method. The results were compared with results obtained by ESPRESSO. For a majority of the benchmark circuits BOOM returned the result in a shorter time, while the result qualities were equal. "Bigger" benchmarks (having hundreds of input variables) were solved by BOOM in a significantly shorter time than by ESPRESSO, while the result obtained by BOOM was better in quality. For PLAs obtained by the Column-Matching algorithm ESPRESSO returned significantly worse results than BOOM, in a much longer time. Some PLAs could not be minimized by ESPRESSO at all, for a prohibitively long runtime.

The principles of BOOM (with later extensions supporting the decomposition of combinational circuits) have been published in [Hla00, Fis00, Hla01a, Hla01b, Fis01a, Fis01b, Fis01c, Fis02b, Fis02c, Fis02e, Hla02, Fis03b and Fis03c].

Another Boolean minimizer has been developed as a more successful byproduct of the CD-A algorithm development: *FC-Min*. Its principles were published in [Fis03e, Fis03g and Fis04g, Fis05b]. Finally, BOOM and FC-Min have been combined together to obtain a universal Boolean minimizer *BOOM-II* [Fis04e, Fis04f, Fis06b].

7.1 Future Work

As the future work several minor modifications are planned to be done, helping to reduce the complexity of the resulting BIST. Namely it is the use of cellular automata or other more complex structures as PRPGs.

More essential modification of the algorithm will enable to adjust the width of the PRPG. Until now, it has been assumed that the number PRPG outputs is equal to the number of CUT inputs, at least in the mixed-mode version. If the width of the PRPG is different, there would be no modification of the algorithm involved. 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 a hint for another possibility of improvement of the algorithm – to consider a proper 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 it will be possible 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.

Incorporating the weighted pattern testing would enable the use of a “shorter” PRPG as well, while the fault coverage reached by the pseudo-random phase would be increased. This, however, would be paid by an overhead caused by the weighting logic.

To be able to cope with most of VLSI core designs the method (or its implementation, respectively) should be modified 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, a kind of partitioning should be applied, to split large circuits into smaller ones, for which the BIST would be 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, it is planned to combine the proposed method with other methods, namely to exploit the reseeding and weighted pattern testing principles.

References

[Ada91] J. Adamek. *Foundations of Coding*. John Wiley & Sons, Inc. 1991, 336 p.

[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

[Cha97] P. P. Chaudhuri, et al.: *Additive Cellular Automata Theory and Applications*. Volume I. IEEE Computer Society Press, 1997, 340 p.

[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

[Cor99] F. Corno, M. Sonza Reorda and G. Squillero. *RT-Level ITC 99 Benchmarks and First ATPG Results*. IEEE Design & Test of Computers, July-August 2000, pp. 44-53

[Cou92] O. Coudert and J.C. Madre. *Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions*, Proc. of 29th DAC, Anaheim CA, USA, June 1992, pp. 36-39

[Cou93] O. Coudert, J.C. Madre and H. Fraisse. *A New Viewpoint on Two-Level Logic Minimization*, Proc. of 30th DAC, Dallas TX, USA, June 1993, pp. 625-630

[Cou94] O. Coudert. *Two-level logic minimization: an overview*, Integration, the VLSI journal, 17-2, pp. 97-140, Oct. 1994

[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.

[Hac96] G.D. Hachtel and F. Somenzi. *Logic synthesis and verification algorithms*, Boston, MA, Kluwer Academic Publishers, 1996, 564 pp.

[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

[Hor90] Hortensius, et al. *Cellular automata circuits for BIST*. IBM J. R&Dev, vol 34, no 2/3, pp. 389-405, 1990

[Kei98] G. Kiefer and H.-J. Wunderlich. *Deterministic BIST with Multiple Scan Chains*, Proc. IEEE International Test Konference (ITC), Washington, DC, pp. 1057-1064, October 1998

[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

[McC56] E.J. McCluskey. *Minimization of Boolean functions*, The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[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

[McG93] P. McGeer et al.: *ESPRESSO-SIGNATURE: A new exact minimizer for logic functions*, In Proc. of the Design Automation Conf.'93

[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

[Ngu87] L. Nguyen, M. Perkowski and N. Goldstein. *Palmini – fast Boolean minimizer for personal computers*, In Proc. of the Design Automation Conf.'87, pp.615-621

[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

[Pra99] D. K. Pradhan and M. Chatterjee, GLFSR-A New Test Pattern Generator for Built-in-Self-Test, IEEE Trans. on CAD, vol. 18, no. 2, pp. 319-328, 1999.

[Qui52] W.V. Quine. *The problem of simplifying truth functions*, Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531

[Raj00] R. Rajsuman: *Iddq testing for CMOS VLSI*, Proceedings of the IEEE, Volume 88, Issue 4, April 2000 Page(s):544 - 568. This is a summary of the basic ideas behind Iddq testing, the history of the technique, and many of its characteristics

[Rud87] R.L. Rudell and A.L. Sangiovanni-Vincentelli. *Multiple-valued minimization for PLA optimization*, IEEE Trans. on CAD, 6(5): 725-750, Sept.1987

[Rud89] R.L. Rudell. *Logic Synthesis for VLSI Design*, Ph.D. Thesis, UCB/ERL M89/49, 1989

[Sen92] E.M. Sentovich et al. *SIS: A System for Sequential Circuit Synthesis*, Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992

[Ser75] M. Servít. *A Heuristic method for solving weighted set covering problems*, Digital Processes, vol. 1. No. 2, 1975, pp.177-182

[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

[Wan01] S. Wang. *Low Hardware Overhead Scan Based 3-Weight Weighted Random BIST*. In Proceedings of the 2001 IEEE international Test Conference (October 30 - November 01, 2001). ITC. IEEE Computer Society, Washington, DC, 868.

[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

[Yan91] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide*, Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991

[MCNC] ftp://ic.eecs.berkeley.edu

[BOOM] http://service.felk.cvut.cz/vlsi/prj/Boom

[BOOMBench] http://service.felk.cvut.cz/vlsi/prj/BoomBench

[ColMatch] http://service.felk.cvut.cz/vlsi/prj/ColMatch

Refereed 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 (DSD‘01) Warsaw (Poland) 4.-6.9.2001, pp. 300-305

[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

[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

[Fis03f] P. Fišer, J. Hlavička and H. Kubátová. *CD-A Based BIST Method*, Proc. 6th International Workshop on Electronics, Control, Measurement and Signals (ECMS'03), Liberec (CR), 2.-4.6.2003, pp. 279-283

[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, pp. 201-204

[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, pp. 221-228

[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

[Fis05a] P. Fišer and H. Kubátová. *Pseudorandom Testability - Study of the Effect of the Generator Type*, Acta Polytechnica, Vol. 45, No. 2, August 2005, CVUT, ISSN 1210-2709, pp. 47-54

[Fis05b] P. Fišer and H. Kubátová. *Improvement of the Fault Coverage of the Pseudo-Random Phase in Column Matching BIST*, Proc. 31th Euromicro Symposium on Digital Systems Design (DSD'05), Porto, (Portugal), 30.8. - 3.9.05, pp. 56-63

[Fis06a] P. Fišer and H. Kubátová. *Multiple-Vector Column-Matching BIST Design Method*, Proc. 9th IEEE Design and Diagnostics of Electronic Circuits and Systems 2006 (DDECS'06), Prague, CZ, 18.-21.4.2006, pp. 268-273

[Fis06b] P. Fišer and H. Kubátová. *Flexible Two-Level Boolean Minimizer BOOM II and Its Applications*, Proc. 9th Euromicro Conference on Digital Systems Design (DSD'06), Cavtat, (Croatia), 30.8. - 1.9.2006, pp. 369-376

[Fis06c] P. Fišer, P. Kubalík and H. Kubátová. *Output Grouping Method Based on a Similarity of Boolean Functions*, Proc. 7th Int. Workshop on Boolean Problems (IWSBP'06), Freiberg, Germany, 21.-22.9.2006, pp. 107-113

[Hla00] J. Hlavička and P. Fišer. *Algorithm for Minimization of Partial Boolean Functions*. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS‘00) Workshop, Smolenice, (Slovakia) 5.-7.4.2000, ISBN 80-968320-3-4, pp.130-133

[Hla01a] 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

[Hla01b] 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

[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

Unrefereed Publications of the Author

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

[Fis02e] P. Fišer. *Minimization of Boolean Functions*, MSc. Thesis, Prague, CTU, May 2002, 70 pp.

[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

[Fis03g] P. Fišer and H. Kubátová. *FC-Min: The Iterative Boolean Minimizer FC-Min*, Pracovní seminář Počítačové Architektury & Diagnostika 2003, Zvíkov (CR), 24.-26.9.2003, pp. 57-62

[Fis05d] P. Fišer. *Mixed-Mode BIST Based on Column Matching*, Počítačové Architektury & Diagnostika 2005, Lázně Sedmihorky, ČR, 21. – 23. 9. 2005, pp. 45-50

Citations

Paper [Hla01a] has been cited in

- R. Lysecky, F. Vahid, *On-chip logic minimization*. In Proceedings of the 40th Conference on Design Automation (Anaheim, CA, USA, June 02 - 06, 2003). DAC '03. ACM Press, New York, NY, 334-337.
- S. Sapra, M. Theobald, E. Clarke: *SAT-Based Algorithms for Logic Minimization*, 21st International Conference on Computer Design (ICCD 2003), San Jose CA, 12.-15.10.2003, pp. 510-519
- S. Verma, K. Permar: *A Novel Method for Minimization of Boolean Functions using Gray Code and development of a Parallel Algorithm*, Proc. 6th Int. Workshop on Boolean Problems (IWSBP'04), Freiberg, Germany, 23.-24.9.2004, pp. 229-236

Paper [Fis03b] has been cited in

- Y. Novikov, R. Brinkmann, *Foundations of Hierarchical SAT-Solving*, 6th International Workshop on "Boolean Problems" 2004, Freiberg University of Mining and Technology, Institute of Computer Science, September 23-24, pp. 103-142, 2004