Czech Technical University
in Prague
Faculty of Electrical Engineering
Department of Computer Science
and Engineering

MSc. Thesis

2002 **Petr Fišer**

Czech Technical University
in Prague
Faculty of Electrical Engineering
Department of Computer Science
and Engineering

Minimization of Boolean Functions

Petr Fišer

May 2002

Statement

I declare herewith, that this M.S. Thesis is my own work and that all used sources are listed as references. Further, I agree that the Department of Computer Science and Engineering, FEE CTU may in the future use the results of my work for non-commercial purposes.

Acknowledgement

I would like to express my great thanks to my supervisor Prof. Jan Hlavička for his guidance, for enabling me to work in an interesting area of the logic design and diagnostics and for all his help during my MSc. studies.

Also, I would like to thank to Hlávkova Foundation for my financial support.

Abstract

This thesis presents two new methods of test-per-clock BIST design for combinational circuits. One of them is based on a transformation of the PRPG code words into test patterns generated by an ATPG tool. This transformation is done by a combinational circuit. For a design of such a circuit two major tasks have to be solved: first, the proper matching between the PRPG code words and the test patterns has to be found, and then the resulting Boolean function that is described by a truth table needs to be minimized. Such a Boolean minimization is a rather difficult task, as the number of input and output variables is often very large. Standard minimization tools, like ESPRESSO, often cannot efficiently minimize functions with a large number of input variables in a reasonable time. Therefore a novel Boolean minimizer BOOM that is capable to handle such function was developed.

The BOOM system is based on a new implicant generation paradigm. In contrast to all previous minimization methods, where the implicants are generated bottom-up, the proposed method uses a top-down approach. Thus, instead of increasing the dimensionality of implicants by omitting literals from their terms, the dimension of a universal hypercube is gradually decreased by adding new literals, until an implicant is found. The function to be minimized is defined by its on-set and off-set listed in a truth table; the don't care set, which normally represents the dominant part of the truth table, need not be specified explicitly.

The second BIST method, called the coverage-directed assignment, combines the pattern assignment and Boolean minimization together. The implicants of the combinational function performing the pattern transformation are generated directly from the PRPG code words, while the process is directed by the coverage of the ones in the test patterns.

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

Abstrakt

V této práci jsou uvedeny dvě nové metody test-per-clock vestavěného testování obvodů (BIST). První z nich je založena na transformaci kódových slov produkovaných generátorem pseudo-náhodných vektorů na předem dané testovací vektory. Tato transformace je vykonávána kombinačním obvodem. Při syntéze tohoto obvodu je nutné řešit dva problémy: nejprve musí být nalezeno správné přiřazení kódových slov k testovacím vektorům a posléze je nutné zminimalizovat výslednou booleovskou funkci. Provést takovouto minimalizaci je poměrně komplikovaná úloha díky velkému počtu vstupních a výstupních proměnných. Proto standardní minimalizátory, jako např. ESPRESSO, často nedokáží zjednodušit takovou funkci v rozumném čase. Proto jsme vyvinuli nový minimalizátor booleovských funkcí BOOM, který je schopný zpracovávat funkce velkých rozměrů.

BOOM je založen na novém principu generování implikantů. Na rozdíl od ostatních metod, kde jsou implikanty generovány z původních zadaných termů, BOOM vytváří implikanty redukcí univerzální hyperkrychle přidáváním nových literálů, dokud nevznikne nový implikant. Funkce je zadaná jejím on- a off-setem, don't care set, který obvykle představuje dominantní část pravdivostní tabulky, nemusí být explicitně specifikován.

Druhá BIST metoda, nazvaná „coverage-directed assignment", kombinuje přiřazování testovacích vektorů s booleovskou minimalizací. Implikanty kombinační funkce provádějící transformaci jsou generovány přímo z kódových slov generátoru, přičemž mechanizmus jejich vytváření je řízen pokrytím jedniček v testovacích vektorech.

Složitost výsledného BIST obvodu je vyhodnocena pro několik ISCAS benchmarků.

Table of Contents

1 Introduction

With the growing complexity of the logic circuits, testing gains an ever-increasing importance. There often arise faulty chips during the manufacturing process due to an inaccurate technology and such chips should be detected and eliminated. Moreover, there may continuously occur physical faults in a chip due to its aging, due to a cosmic radiation or an inappropriate use. Thus, the chips should be tested also during their functionality. The external testers are often very expensive and they cannot be used for a continuous testing. The concept solving this problem is a BIST (Built-in Self-test) approach. It allows the logic circuit (chip) to be tested without using any external testers; the testing mechanism is included in the circuit itself. Moreover, the use of the BIST equipment is becoming inevitable with the growth of the complexity of the VLSI devices. The growth of the amount of logic accessible through one chip pin follows approximately the Moore's law (for the feature size of 0.1 micron it reaches some 100 000 transistors per pin), and thus testing the chips via their external access points is often rather demanding. The BIST allows us to test individual functional blocks of a chip separately, as their inputs and outputs can be easily accessed by the BIST structures; the test is not restricted to the external outputs.

The problem of BIST design has been studied for more than two decades and the results can be found in several survey papers [McC85, Aga93]. Obviously, to implement a BIST some additional circuitry is necessary, thus the BIST causes an area overhead of a chip and this means also bigger power consumption of a circuit. Especially nowadays, when the low-power design is desired, the BIST should be implemented in a minimal area. The second aspect is duration of a test – the longer time the system spends with testing, the more power is consumed, and thus also the length of a test should be reduced.

The general BIST structure consists of the three main parts – see Figure 1.1. The TPG (*Test Patterns Generator*) produces the *test patterns* that are fed to the inputs of a C*ircuit under Test* (CUT) and the responses of a circuit are then evaluated in a *Response Evaluator* (RE).

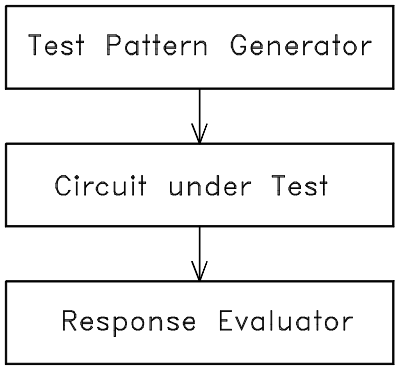

Figure 1.1. BIST structure

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

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

1.1 Test Modes

As the TPG can be constructed to have both parallel and/or serial outputs the BIST can be implemented in two general ways: test-per-clock and test-per-scan. The *test-per-clock* BIST feeds the CUT with the parallel outputs of the TPG, and thus each test pattern is processed in one clock cycle. A LFSR is often used as a TPG (see Subsection 1.2), either standalone or connected with some combinational logic modifying its code words.

The response of the CUT goes in parallel to the response evaluator, which is often a MISR (*Multi-Input Shift Register*). Such a structure is illustrated by Fig. 1.2.

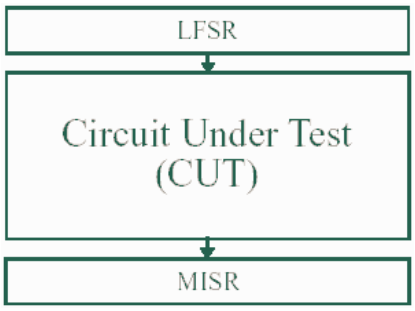

Figure 1.2. Test-per-clock BIST

A second typical structure, suitable especially for testing sequential circuits is denoted as a *test-per-scan* BIST. It is used in connection with CUTs having a scan chain, i.e., the circuit's flip-flops are connected into a chain making one scan register for testing purposes. Here the test patterns are shifted into the scan register of the CUT and applied by activating the functional clock after every full scan-in of one test pattern. The response is then scanned out and typically evaluated by a serial signature analyzer (signature register). The basic structure of a test-per-scan BIST is shown in Fig. 1.3.

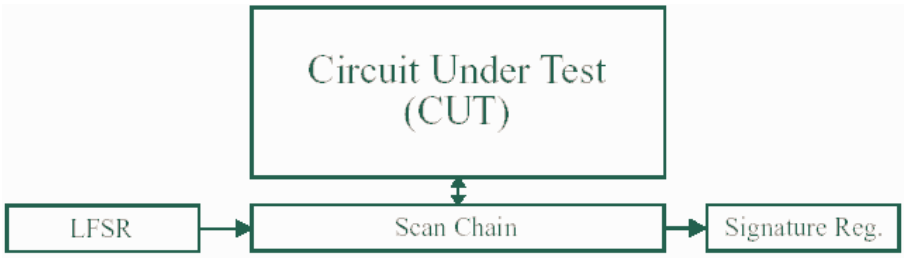

Figure 1.3. Test-per-scan BIST

1.2 The PRPG Structure

The design of the test pattern generator (TPG) is obviously of a key importance in the BIST design, as the area overhead of a TPG determines the area overhead of the whole BIST. Generally, the TPG is a *sequential* circuit that produces test patterns. There exist many TPG architectures (see Subsection 1.3) and most of them exploit sequential structures called *pseudo-random pattern generators* (PRPG). The PRPGs are generally very easily implementable circuits that generate deterministic code words having random characteristics. These code words are then either fed directly to the CUT inputs, or they are modified by some additional circuitry before that.

The most common PRPG structures are *linear feedback shift registers* (LFSR) 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 (*k*, *k*-*n*). The structure of an *n*-stage LFSR with internal XORs is shown in Fig. 1.4.

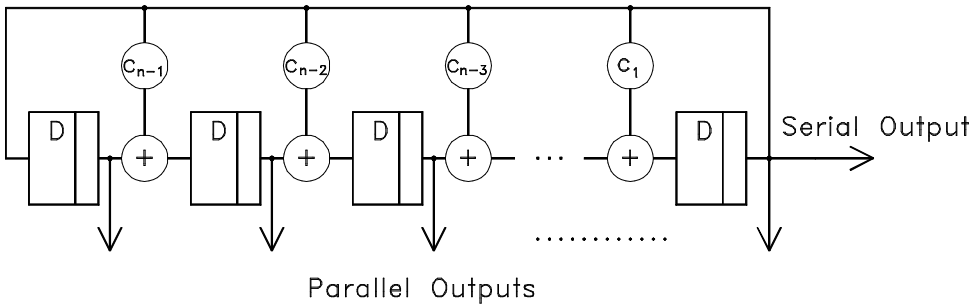

Figure 1.4. LFSR structure

The register has *n* parallel outputs drawn from 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 not (0), thus it determines whether there exists a XOR gate or the flip-flops are directly connected.

The sequence of code words that are produced by a LFSR can be described by a *generating polynomial* g(x) in GF(2^n):

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

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

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

The cellular automata are sequential structures similar to a LFSR, but mostly they are not linear and also their period is shorter. An example of a CA performing multiplication of the polynomials corresponding to code words by the polynomial x+1 (the rule 102 for each cell) is shown in Fig. 1.5. For more information see [Alo93].

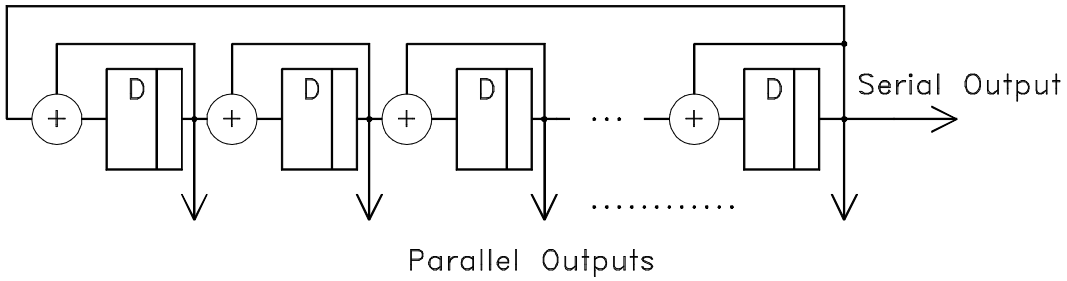

Figure 1.5. Cellular automaton example

In general, the LFSR code words have more balanced numbers of ones and zeros (both in the very code words and in the outputs when observed during several cycles) than the cellular automata. For some purposes the more balanced patterns are advantageous, however, the properties of some specially designed cellular automata sometimes can be of advantage. Several studies were made describing a design of a cellular automaton that produces the required test patterns, see e.g., [Nov98].

1.3 Survey of the BIST Methods

1.3.1 Exhaustive Testing

There exist several testing approaches differing in their successfulness and area overhead. The most naive method – the *exhaustive testing* – feeds the circuit with all the 2^n patterns and checks the responses. Obviously, for a combinational circuit the exhaustive test provides the full fault coverage, and can be very easily implemented (the area overheat is often the lowest possible), but it is extremely time demanding and thus very inefficient. It is applicable to the circuits with up to 30 inputs (10^9 patterns, which takes 1 sec on the frequency of 1 GHz), for more inputs the exhaustive testing is not feasible. The test patterns are mostly generated by an LFSR, as 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 the need to use all the 2^n test patterns. The circuit is divided into several possibly overlapping *cones* (see Fig. 1.6), which are logic elements that influence the individual outputs of the circuit. Then, all the cones are exhaustively separately tested, 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.

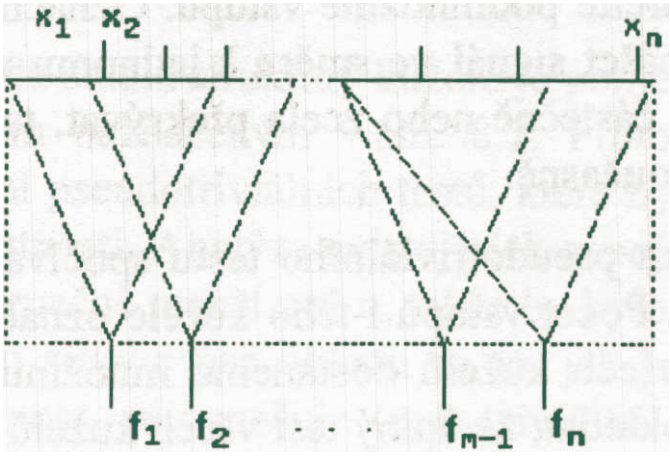

Fig. 1.6. The circuit's cones

1.3.2 Pseudo-random Testing

The second group of BIST methods exploits the test patterns that are pre-computed by an ATPG (*Automatic Test Pattern Generator*) tool. For a given circuit the required number of test patterns providing a full fault coverage test is computed and those patterns are to be fed to the circuit's inputs. In the simplest approach these patterns are stored in ROM memory, from where they are directly lead to the circuit under test. However, the area overhead of a ROM is often prohibitively large. Here some trade-off methods come to place – namely the pseudo-random BIST methods.

In a simple *pseudo-random testing* the test patterns are generated by some pseudo-random pattern generator (PRPG) and lead directly to the circuit's inputs. The difference from the exhaustive testing is that if the PRPG structure and seed are properly chosen, only several test patterns (less than 2^n) are necessary to generate to completely test the circuit. The pseudo-random testing is also widely used in a case when the complete fault coverage is not required, as the pseudo-random patterns often successfully detect most of the easy-to-detect faults.

In more complicated pseudo-random testing methods the pseudo-random code words generated by a PRPG are being transformed by some additional logic (combinational or sequential) in order to reach better fault coverage. Here the main area overhead consists in the combinational logic.

1.3.3 Mixed-Mode Testing

The combination of using a PRPG and the ROM memory is known as a *mixed-mode testing*. In the simplest case, a plain PRPG is used to produce several test patterns detecting easy-to-detect faults and then the random pattern resistant faults are detected by patterns stored in ROM. However, the size of a memory is often large, even when using this approach. Here some pattern compression techniques have to be used [Aga81].

More successful mixed-mode BIST methods use a LFSR that is seeded with more than one computed seeds during the test, thus only the seeds need to be stored in a ROM [Koe91]. The seeds are often smaller than the test patterns themselves and, most importantly, more than one test patterns are derived from one seed. This significantly reduces the 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 that produces the required test patterns. A solution of this problem is using a multi-polynomial LFSR (MP-LFSR), where the feedback network of a LFSR is reconfigurable [Hel92]. Here both the seeds and polynomials are stored in a ROM memory and for each LFSR seed also a unique LFSR polynomial is selected. The structure of such a TPG is shown in Fig. 1.7.

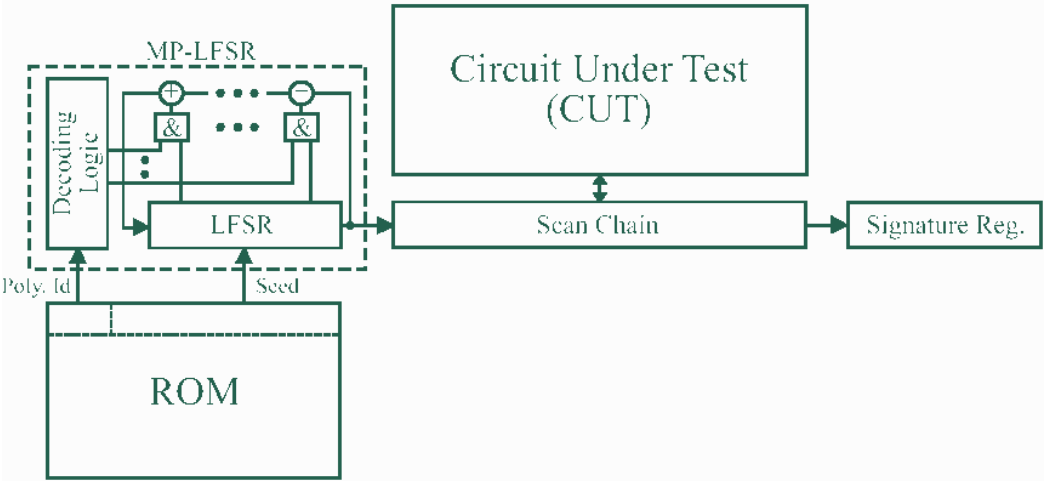

Figure 1.7. Multi-polynomial BIST

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

More complicated mixed-mode approach minimizing the memory requirements and area overhead was proposed in [Nov01]. A special cellular automaton constructed of T-flip-flops is used as a PRPG whose code words are being influenced by *modifying bits* stored in a serial memory.

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

1.3.4 Weighted Pattern BIST

One of such approaches, namely the *weighted pattern testing*, biases the PRPG patterns by defining a *signal probability* of each of the PRPG outputs (the probability of a 1 value) in order to reach the required test patterns. In the weighted pattern testing method two problems have to be solved: first, the weight sets have to be computed and then the problem how to generate the weighted signals. Many weight set computation

methods were proposed [Bra87] and it was shown that multiple weight sets are necessary to produce patterns with the sufficient fault coverage [Wun88]. These multiple weight sets have to be stored on chip and also the logic providing switching between them is complicated, thus this method often implies a large area overhead.

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

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

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

It can be said in general, that the weighted pattern BIST methods either require large area overhead, or they are restricted to circuits with special properties.

1.3.5 Mapping Function

The main principle of all the weighted pattern testing methods consists in placing some combinational logic at the LFSR outputs in order to reach a better fault coverage. However, the design of this combinational block can be generalized to perform any other mapping function. The method proposed in [Tou95a] consists in modifying some of the PRPG patterns in order to detect hard-to-detect faults – see Fig. 1.8.

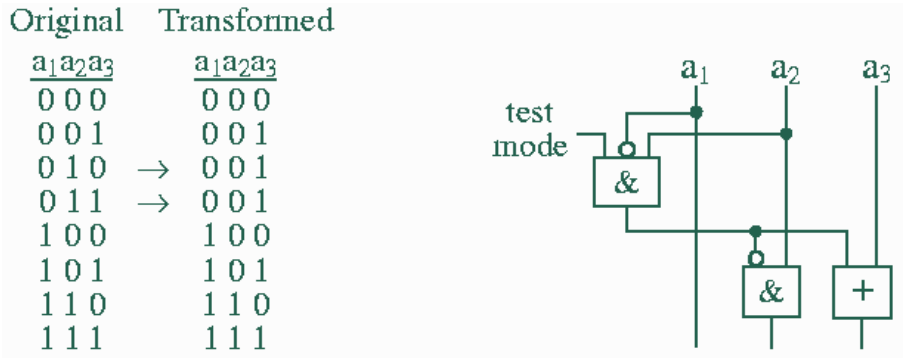

Figure 1.8. Modifying the PRPG patterns

This idea was generalized in [Tou95b], where the problem of finding a mapping function is transformed into finding a minimum rectangle in a binate matrix. Procedures used in ESPRESSO [Bra84] were used to find a mapping logic.

In [Tou96] this method was extended for testing sequential circuits – the test sequence entering the scan chain is altered by a *bit-fixing* function modifying some of the bits in order to obtain the required test patterns (see Fig. 1.9).

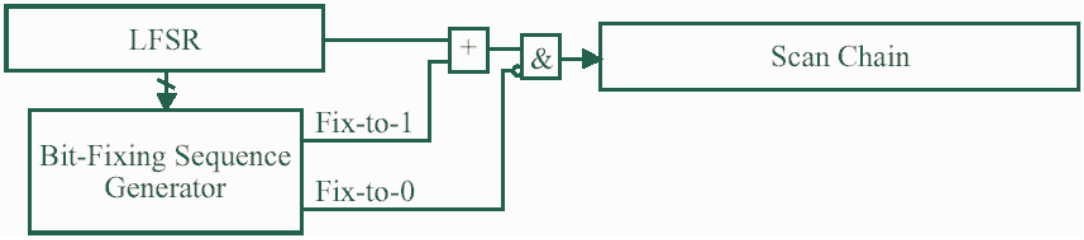

Figure 1.9. Bit-fixing function

As it is a more general method than the weighted pattern BIST, in most cases it reaches lower area overhead. However, the logic function modifying only several PRPG code words, while the rest remains unchanged, may be rather complicated. In general it may be more advantageous to design a simple logic function that transforms *all* the code words to the test patterns with the required fault coverage, as it is done in a row matching method.

1.3.6 Row Matching

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

The row matching means finding an assignment of these test patterns to the code words, as it is shown in Fig. 1.10. 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. Such an idea is also exploited in our BIST methods presented in this thesis.

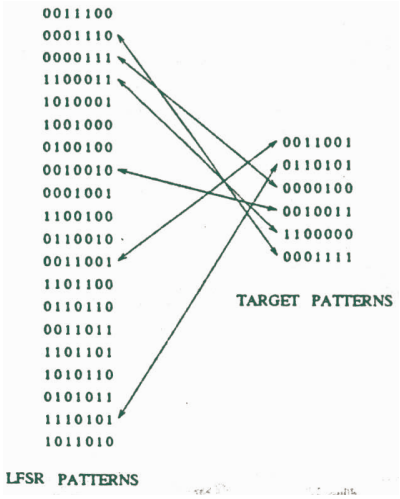

Figure 1.10. Row matching principle

In [Cha95] the *cost function* of the row matching is used as a criterion for finding a row match. The cost function is an estimation of the complexity of the combinational function performing the pattern transformation. The cost of a matching M for a n-input CUT (and thus the combinational block has n outputs) is defined as follows:

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

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

The aim is to find a row matching that minimizes this function. This is, however, an NP hard problem [Gar79] 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. 1.10 is shown in the following Figure:

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

Figure 1.11. The final truth table

There are several drawbacks of this method: first, computing the input indexes is an NP hard problem itself, and thus another nested heuristic has to be used. The one proposed by Chatterjee is a greedy heuristic, which is rather inefficient. The second problem of the algorithm is the fact that the number of input variables needed to produce one single output is taken as a criterion of minimality – for a multi-output minimization it may not be truth.

The main drawback consists in the fact that the row matching is done by a simple greedy heuristic that cannot change the matches previously made – we have found that the matching must be taken in global in order to reach good results. The method proposed in this thesis efficiently avoids all these drawbacks.

2 Problem Statement

In this thesis we propose a test-per-clock BIST method for combinational circuits, where the CUT is fed with test patterns that are pre-computed by an ATPG tool. These patterns are generated by a TPG circuit consisting of two parts, as is shown in Fig. 2.1. First, pseudo-random vectors are produced by a PRPG and then these vectors are converted into the required test patterns by a combinational circuit – the *output decoder*.

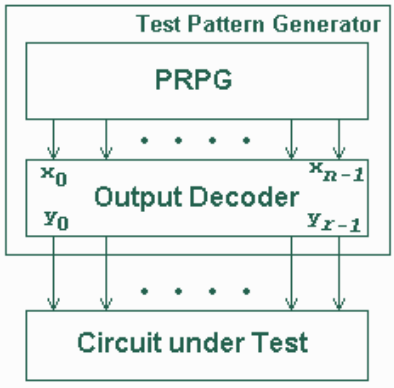

Figure 2.1. TPG structure

Designing a simple BIST means reducing the complexity of all its parts, namely using a short shift register, simplifying (or eliminating) the control unit and memory, and most importantly simplifying the LFSR output decoder. In our method any control logic and memory are totally eliminated and our goal is just to synthesize the output decoder to be as small as possible. In this thesis only the methods describing the synthesis of an output decoder are described and the it is mainly taken as a combinational problem; the methods are applicable also in other areas of logic design and Boolean function theory. The structure of the decoder is strictly dependent on the PRPG patterns as well as on the required test patterns. Thus, the inputs of the design algorithm are these two pattern sets.

Let us have an *n*-stage (*n*-bit) PRPG running for *p* cycles. Then the code words produced by this PRPG can be described by a ***C** matrix* (code matrix) with the dimensions (*n*, *p*). The total length of a test is then equal to *p*. The parallel outputs of the PRPG are entering the output decoder as input variables x_0 - x_{n-1}. Thus, the columns of a **C** matrix will be sometimes denoted as the values of the *input variables* of the output decoder, while the rows of the matrix can be taken as *input vectors* or *input minterms*. An example of a **C** matrix produced by a 5-stage LFSR with generating polynomial $x^5 + x^2 + 1$ seeded with a vector 00010 running for 10 clock cycles is shown in Fig. 2.2.

x_0-x_4

C =

00010
00001
10100
01010
00101
10110
01011
10001
11100
01110

Figure 2.2. Example of a C Matrix

The test set pre-computed by an ATPG tool is described by a ***T** matrix* (test matrix). For an *r*-input CUT the output decoder has to have *r* outputs denoted as y_0-y_{r-1}. For a test set with *s* vectors the **T** matrix will have dimensions (*r*, *s*). The columns of the **T**-matrix will be denoted also as *output variables* of the decoder, the rows as the *output vectors*.

Designing the output decoder means finding a combinational logic that transforms these two matrices – the **C** matrix vectors entering the decoder should be transformed into the **T** matrix vectors (see Fig. 2.3).

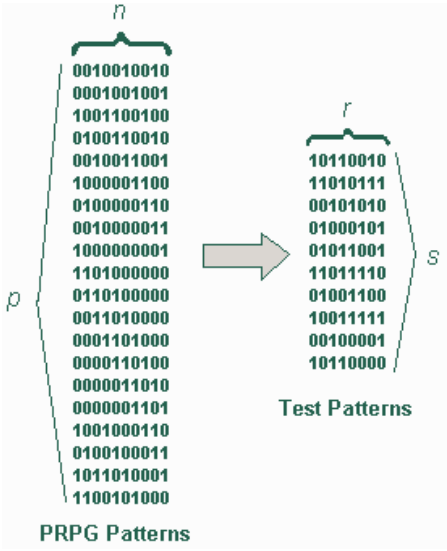

Figure 2.3. Transformation of matrices

There are some obvious relationships valid for the values mentioned above, like $p \leq 2^n - 1$ (the maximum number of distinct patterns that can be generated by a PRPG) and $p \geq s$, because there must be enough patterns to implement all test vectors generated by the ATPG. On the other hand, there are no strict requirements regarding the relationship of *n* and *r*, because the number of LFSR stages can be even smaller than the number of CUT inputs (the LFSR outputs can be split). For a larger number of **C** matrix columns the transformations are often easier and the resulting combinational logic is less complex, but the size of PRPG grows, and thus there must be a trade-off found.

As all the methods proposed here are restricted to *combinational circuits*, they are based on the following important fact: when testing combinational circuits, the order of test patterns generated by an ATPG tool is insignificant and thus the patterns can be reordered in any way. In other words, any vector (row) from a **T** matrix can be assigned to any vector of a **C** matrix. Moreover, the rows in the **C** matrix need not form a compact block. The excessive patterns that are not transformed into test vectors just represent the idle cycles of the PRPG. They do not disturb the testing, but only extend its length. If a low-power testing is required, we may use the pattern inhibition techniques - see [Gir99]. Finding a transformation from **C** matrix to **T** matrix means finding a *matching* of all *s* rows of **T** matrix with any distinct *s* rows of **C** matrix – thus finding a *row assignment* (see Fig. 2.4), i.e., determining which **C** matrix rows are matched with which **T** matrix rows.

The output decoder is than a combinational block that converts *s n*-dimensional vectors of a **C** matrix into *s r*-dimensional vectors of **T** matrix. The decoder is represented by a Boolean function with *n* inputs and *r* outputs, where only values of *s* terms are defined, the rest are don't cares. This Boolean function can easily be described by a truth table.

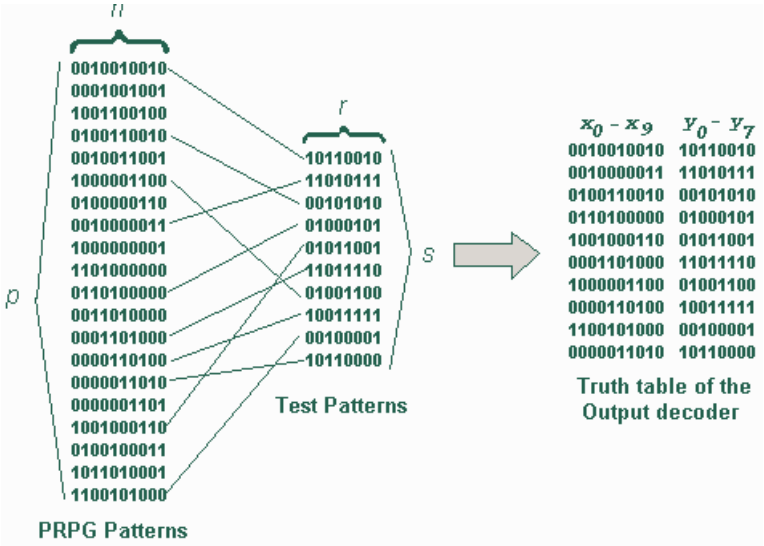

Figure 2.4. Example of a row matching

Although the output decoder is just a combinational circuit, its design may represent a rather difficult problem, because of the high number of input and output variables. Until recently, the problems of this size were not solvable because the state-of-the-art tools, like, e.g., ESPRESSO [Bra84, ESPRESSO], were unable to handle them. We succeeded in designing these decoders only thanks to the existence of the proprietary minimization system BOOM [1-7], which is briefly described in Section 6. Its runtimes were sufficiently short to allow experimental verification of several different solutions.

3 Column Matching

The assignment of the rows of **T** matrix to the rows of **C** matrix is of a key importance for the design of the output decoder. Our task is to find an assignment that reduces the combinational logic of the output decoder to minimum.

An indirect inspiration for the proposed method was the paper [Cha95], where the LFSR outputs and the required test patterns are matched row-wise. Some very interesting BIST solutions for the ISCAS benchmark circuits, although with incomplete fault coverage, were found in this way.

The proposed method on the contrary works with *column matching*. It is based on the fact, that if in the final assignment the *i*-th column of the **C** matrix is exactly the same as the *j*-th column of the **T** matrix (the values of *j*-th output variable are equal to the values of *i*-th input variable), there is no combinational logic required to implement the *j*-th variable in hardware. Thus, we try to find as many column matches as possible, after that all the **T** matrix rows are assigned to the **C** matrix rows by solving a group (the row matching is restricted by the column matches) of row matching problems using some of the row matching methods [Cha95], which yields a truth table. From this truth table the resulting combinational logic is synthesized by a Boolean function minimizer (BOOM). The whole process is illustrated by Fig. 3.1:

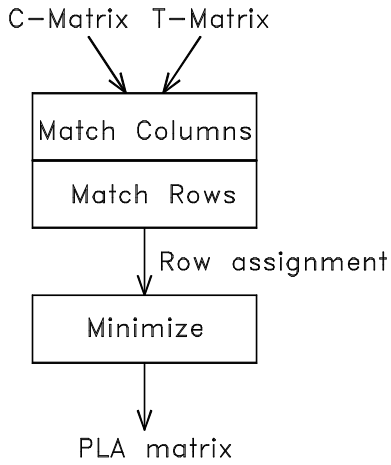

Figure 3.1. The column matching

An example in Fig. 3.2 shows the principle of the column matching: ten 5-dimensional vectors of the **T** matrix have to be assigned to the 5-dimensional vectors of the **C** matrix and the logic of the output decoder synthesized. When the **T** matrix vectors are properly reordered and the corresponding vectors matched row-wise, we get two column matches which yields no hardware to implement two outputs of the decoder (y_0, y_4). The remaining output functions (y_1, y_2, y_3) are synthesized using a standard design method.

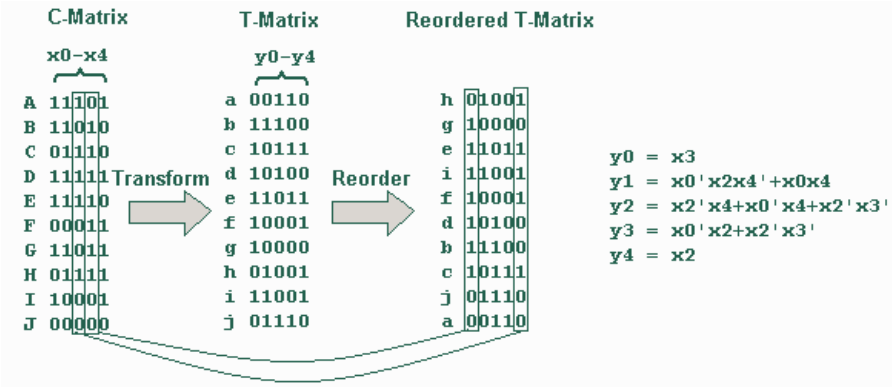

Figure 3.2. Example of column matching

We have developed several methods implementing the column matching principle, their applicability depends on the nature of the problem. Some of them are described in the following subsections.

3.1 One-to-One Assignment

As a one-to-one assignment will be denoted the case where $p = s$, thus all the PRPG vectors are assigned to the test vectors and no idle PRPG cycles are present. This is the fastest BIST method (only the necessary number of PRPG cycles is needed), however, the amount of logic needed to implement the output decoder is often large.

The most important feature of the one-to-one assignment is the fact that all PRPG vectors that are to be transformed into test patterns are known in advance. Determining a column match is a simple task then: the match is possible if the counts of ones (and zeros) in the corresponding columns are *equal*. In our previous example (Fig. 3.2) 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 finding a column match the two matrices are decomposed into two disjoint parts containing the rows with zeros and ones respectively in the matching columns, let the sub-matrices 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}.

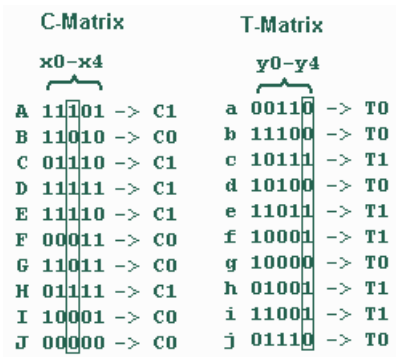

Figure 3.3. The first assignment to the submatrices

Finding all possible column matches consists in a successive decomposition of each of the original matrices into set systems until no other 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 selection of the candidate columns for a match is controlled by a heuristic, which measures the proportion of zeros and ones in both the candidate columns and selects the most balanced decomposition. Another possibility is an *exhaustive column match search*, where all the possible combinations of column matches are tried. This is applicable for problems with a low number of possible column matches.

The output of this algorithm are two systems of subsets of the **C** and **T** matrices. Each two corresponding subsets contain vectors that can be assigned to each other in any order. Then some row-matching based method, e.g. [Cha95], is applied to all these subsets to make a final assignment. At the end all rows of the **T** matrix are matched with all the rows of **C** matrix.

3.2 Scoring Matrix

Exact matching of columns minimizes the logic needed to implement some of the output variables of the output decoder. However, it is often not possible to find more than one or two matches using this method. Moreover, finding an exact match sometimes does not mean the reduction of the entire logic. The amount of logic needed to implement other outputs may surpass the savings gained by exact matches. Hence, an idea of non-exact matching was introduced together with a scoring matrix **S**. As was said above, when one exact column match is found (two matching columns with the same number of ones), the logic needed to implement one output variable of output decoder is reduced to zero. The non-exact column matching is based on the idea, that when two columns with *nearly the same* number of ones are matched, there will be some logic needed to implement the output variable, but not too complex. Next, this method considers the fact that one column matching may negatively affect another one. Thus the previously found matches eliminate some of the new ones, even if they might be better. These previous matches should rather be suppressed. All these aspects are considered in the **S** matrix based column-matching method. The structure of the **S** matrix used for the one-to-one assignment is shown in Fig. 3.4. The rows of the **S** matrix represent the vectors of a **C** matrix and the columns represent the vectors of a **T** matrix. The value of an **S**[*i, j*] element expresses the measure of likelihood that the *i*-th row of the T matrix will be matched with the *j*-th row of C matrix.

Figure 3.4. Scoring matrix for 10 rows of C and 10 rows of T

At the beginning of the column matching process the **S** matrix is filled with some constant non-zero value. Then we are looking successively for the best possible column match, i.e., for two columns with the nearest counts of ones and zeros respectively. If, e.g., column *k* of the **T** matrix is matched with column *l* of the **C** matrix, then the values of all cells [*i, j*] in the **S** matrix are reduced by a *penalization factor* when **T**[*i, k*] ≠ **C**[*j, l*] (when the corresponding rows contain opposite values in the selected columns). The penalization factor depends mainly on the similarity of the matched columns.

This procedure is performed for all columns of the **T** matrix and then the matches with the highest values in the **S** matrix are selected, so that the best row matches are finally picked out from the **S** matrix.

3.3 Generalized Column Matching

Both methods proposed above assumed a one-to-one row matching where $p = s$. 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. 2.4). Then the idle test cycles are present, however this approach significantly reduces the complexity of the output decoder.

The exact column matching method is very efficiently applicable here. Unlike the method described in subsection 3.1, we cannot determine a column match by comparing the number of ones in the corresponding columns, because we do not know in advance which **C** matrix vectors will be included in the final row assignment. However, when we can freely choose among the code words (if $p >> s$), finding an exact match is a trivial problem; for several initial matches practically any two columns can be successfully matched.

The method of generalized column matching is then very similar to the set system based method described in 3.1. Both **C** and **T** matrices are being divided into two disjoint parts, while in this case their sizes need not be equal, the number of vectors in each $\mathbf{C_i}$ must be *greater or equal* to the number of vectors in the corresponding $\mathbf{T_i}$. If not, there would exist some test patterns that cannot have a code 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.

3.4 Column Matching Exploiting Test Don't Cares

Until now, we have assumed that the **T** matrix contains only test patterns in their compact 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 the don't cares can be advantageously exploited in the design of the output decoder.

The problem of constructing the output decoder is in this case similar to the previous ones: all vectors from the **T** matrix are to be assigned to vectors of the **C** matrix, while $s \leq p$. The **T** matrix contains don't care states, the **C** matrix contains only minterms as concrete vectors are produced by a PRPG. We have found that using the set system approach here is rather time-consuming, although it is not impossible. More advantageous is a *binary scoring matrix* approach. Here the scoring matrix **S** contains only 0 and 1 values in each cell. Initially the matrix is filled with 1 values. If, again, column *k* of the T matrix is matched with column *l* of the **C** matrix, then the value of the element [*i, j*] in the scoring matrix is set to zero when **T**[*i, k*] ≠ **C**[*j, l*] and

simultaneously **T**[*i, k*] ≠ DC. Thus the matrix element is set to zero when the corresponding rows of **T** and **C** matrices contain opposite values in the selected columns, while the don't cares do not restrict the row match.

The fact that the rows of the **C** matrix are not divided into disjoint sets complicates the final assignment to some extent. The *i*-th row of the **T** matrix can be matched with the *j*-th row of the **C** matrix if the cell [*i, j*] in the scoring matrix is equal to one. We need to find a match for all rows in the **T** matrix while any row of the **C** matrix can be used only once. Finding such an assignment from the scoring matrix is an NP hard problem, as well as even determining whether there exists such an assignment. This problem has to be solved by some heuristic method.

The second consequence of this fact is that it is hard to determine the end of the column matching process, as the previously used criteria are not applicable here. In practice we perform the final assignment after each of the successful column matching and, if it fails, the procedure ends.

3.5 Inverse Column Matching

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

In most cases the PRPG outputs are drawn directly from the outputs of flip-flops. These flip-flops often have also the negative value of their outputs provided. Then, also the negative exact matching should be considered as a possibility of implementing some variable of the output decoder as a simple wire. This happens when the value of the matched output variable is a complement to the value of some input variable in all care terms. The possibility of a negative column matching has to be considered in all the previously described column matching methods (Subsections 3.1-3.4).

3.6 An ISCAS Benchmark Example

To illustrate the principles of the method we have chosen the c17 ISCAS benchmark [ISCAS] for its simplicity. As an input we have a complete test set generated by an ATPG tool. The set consists of 10 test patterns (see Fig. 3.5). Our goal is to implement a BIST structure applying the given test set to the c17 benchmark circuit.

Before starting the solution, it should be mentioned that the complete test set from Fig. 3.5 is used here for strictly illustrative purposes. It is well known that c17 can be completely tested with 4 patterns and that, on the other hand, if we used an exhaustive test (which would be easy to implement due to the small size of the circuit), the output decoder would completely disappear.

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

Figure 3.5. ISCAS c17 test vectors

As a PRPG we have selected a 5-stage LFSR with generating polynomial $x^5 + x^2 + 1$ seeded with a vector 00010. In the following two subsections we first let the LFSR run only the necessary 10 cycles to find a one-to-one assignment, then we let it run for 19 cycles in order to reach a better solution.

3.6.1 One-to-One Assignment for c17 Benchmark

In this example we show in detail how is the decomposition of matrices into set systems done for the one-to-one assignment. As an input we have two matrices: the **C** matrix represents the patterns generated by the LFSR, the **T** matrix contains pre-generated test patterns from Fig. 3.5.

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 inverse 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 we have found three exact column matches.

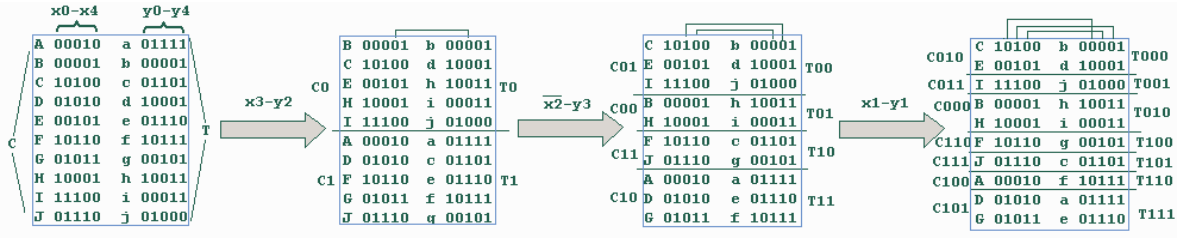

Figure 3.6. One-to-one exact column matching example

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. 3.7.

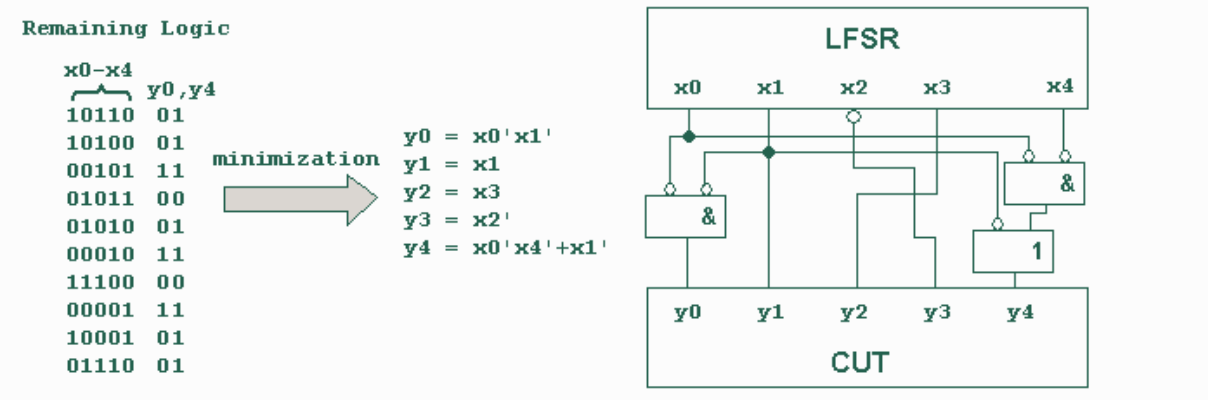

Figure 3.7. BIST implementation for c17 circuit

3.6.2 Generalized Column Matching Example

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

We have found experimentally that, when we retain the LFSR generating polynomial and seed from the previous example, 19 LFSR cycles are needed to match *all* the columns. Thus, absolutely no additional logic is needed to build the LFSR output decoder for the BIST. In Fig. 3.8 we show one of the possible assignments of the test patterns to 10 of the 19 LFSR patterns and the resulting combinational logic of the output decoder, which in this case is just a permutation of wires. 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.

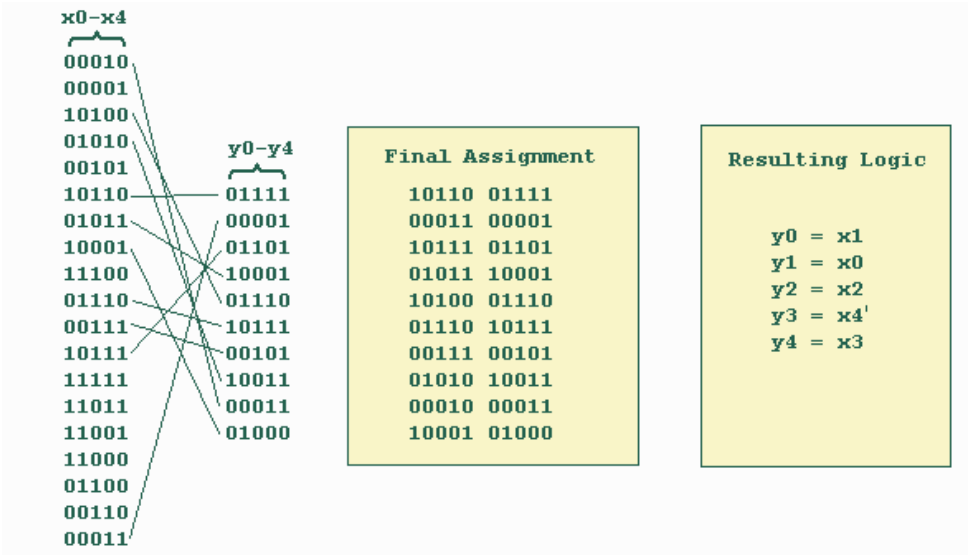

Figure 3.8. Assignment of rows for c17 circuit

4 Experimental Results of the Column Matching Method

4.1 Influence of the PRPG on the Result

The influence of the PRPG structure on the complexity of the output decoder is analyzed here. Intuitively, the more cycles the PRPG runs, the more vectors are available to choose from when finding column matches (using the generalized column matching method) and thus the better solution can be found. Similarly, the more bits has the PRPG, the more columns we have to choose from and there are more possibilities to find the exact column matches.

As an example we have chosen the c432 ISCAS benchmark circuit [Brg85], whose test vectors were generated by an ATPG tool ATOM [ATOM]. Test vectors are in their non-compressed form, thus they contain don't cares. The c432 circuit has 36 inputs, thus $r = 36$ and the complete test set contains 520 vectors ($s = 520$).

The **C** matrix vectors were produced by a LFSR with variable width n (10 – 100) seeded with a random non-zero vector, and the LFSR ran for a given number of cycles p (520 - 10000). The entries in Tab. 4.1 indicate the number of exact matches achieved and the complexity of the final output decoder. As the complexity measure we selected the sum of the number of literals in the sum-of-products form of the Boolean function and the output cost. The output cost is the total number of wires entering the OR-gates in the two-level implementation of a Boolean function. The results confirm the previous assumption: the number of exact matches reached increases and the complexity of the output decoder decreases with the increasing number of patterns generated by a LFSR. However, the width of a PRPG is not of a big significance.

Table 4.1. Influence of the LFSR on the result

n / p	520	1000	2000	5000	10000
10	5 / 2667	11 / 1961	-	-	-
13	4 / 2392	12 / 1840	14 / 1509	19 / 1436	-
22	6 / 2069	14 / 1459	16 / 1400	17 / 1084	21 / 794
37	5 / 1861	15 / 1284	19 / 1014	22 / 739	24 / 744
50	6 / 1736	15 / 1190	17 / 1241	22 / 699	22 / 727
100	4 / 1810	16 / 1075	18 / 855	22 / 612	24 / 698

Entry format: exact matches / literals + output cost

A similar experiment was made using a cellular automaton instead of a LFSR, while all the other parameters are kept the same. The CA chosen was an automaton performing multiplication of the polynomials corresponding to code words by the polynomial x+1 (the rule 102 for each cell) – see Fig. 1.5. The results are shown in Table 4.2. Let us note that the missing entries in the Table imply that the period of the given cellular automaton was not long enough to produce the required number of patterns.

Table 4.2. Influence of the CA on the result

n / p	520	1000	2000	5000	10000
13	4 / 2473	-	-	-	-
22	6 / 2108	-	-	-	-
37	6 / 1951	14 / 1469	18 / 887	20 / 971	23 / 737
50	4 / 1941	9 / 1647	18 / 919	21 / 959	21 / 841
100	4 / 2034	12 / 1479	16 / 1122	17 / 1121	22 / 921

Entry format: exact matches / literals + output cost

We can see that the values obtained are very similar to the situation when an LFSR is used as a PRPG. Thus, we can make a conclusion that the type of a PRPG does not influence the column matching algorithm significantly.

In the third measurement again the LFSR was used as a PRPG and also the same (c432) benchmark was used, but the test was compacted, thus the test vectors didn't contain don't cares. The length of a test is equal to 100, and thus smaller LFSR can be used (with the minimum number of stages equal to 7) to produce the test. Table 4.3 shows that even when less column matches were found, the resulting function is often simpler for lower number of LFSR cycles.

Table 4.3. Influence of the LFSR on the result – compacted test

w / l	100	500	1000	5000	10000
7	2 / 1987	-	-	-	-
10	2 / 1775	7 / 1544	8 / 1473	-	-
36	3 / 1461	7 / 1445	8 / 1289	10 / 1224	11 / 1150
50	3 / 1377	7 / 1366	8 / 1306	10 / 1212	11 / 1183

4.2 ISCAS Benchmarks

The results of the column matching method are illustrated on several combinational ISCAS benchmarks. Next, the impact of using a test with don't cares is studied here.

The complete test sets were generated by the APTG tool ATOM [ATOM] for all benchmark files. Both test sets – with and without don't cares - were used to study the influence of the nature of the source on the resulting complexity of the output decoder.

An LFSR with the width equal to the number of primary inputs of the CUT was used as a pseudorandom pattern generator, and the number of patterns generated was set to 5000. The LFSR was seeded with a random vector.

Table 4.4 compares the results of the two test sets. For each test its size is shown in the table, for non-compacted tests the portion of don't cares in the test is indicated in parentheses. For each benchmark circuits and its test set the number of exact column matches obtained is shown, as well as the complexity of the resulting output decoder (cost) in terms of the sum of the number of literals and the output cost (like in the previous example).

Table 4.4. ISCAS Benchmarks

benchmark	LFSR (n / p)	Test with DCs			Compacted test		
		test size (r / s)	matches	cost	test size (r / s)	matches	cost
c1355	41 x 5000	41 x 1566 (14.8 %)	6	10944	41 x 192	8	1475
c1908	33 x 5000	33 x 1870 (48.2 %)	10	9277	33 x 210	10	2043
c432	36 x 5000	36 x 520 (68.5 %)	17	908	36 x 100	10	1180
c499	41 x 5000	41 x 750 (5.4 %)	12	5375	41 x 127	9	698
c880	60 x 5000	60 x 942 (80.5 %)	39	1009	60 x 133	10	3024

We can see that in two cases (shaded cells) the use of a non-compressed test led to a smaller hardware overhead. These were the cases where the portion of don't cares in the test was large. In other cases, where the portion of don't cares is relatively small and the test with don't cares nears its compacted form, the use of the compacted test is more advantageous since the number of test patterns is smaller.

5 Coverage-Directed Assignment

In the previous Section the column matching based approach to finding an assignment of the rows of the **T** matrix to the rows of the matrix **C** was discussed. When the column match of the *i*-th column from the **C** matrix to the *j*-th column of the **T** matrix is found (direct or inverse), the implicant x_i (or x_i' respectively) is selected to be a part of the final solution. This implicant covers all the ones in the *j*-th column of the **T** matrix, thus all the values of the *j*-th output variable of the output decoder are produced by this implicant. For example, in Figure 12 when the match x_3-y_0 is found, the implicant x_3 covers all the ones in the first column of the **T** matrix. The column matching method tries to find the simplest implicants possible (which can be implemented without any logic) in order to cover as many complete **T** matrix columns as possible. The values of the other output variables have to be generated by other terms, which are determined by some logic minimizer.

However, as was shown above, after finding the column match the possibility of matching the rows is somehow restricted. Then, the row assignment need not be ideal in terms of complexity of the final result measured by the number of additional terms that need to be generated. In other words, these terms can complicate the solution, so that the advantages of the column matching are lost. The way to solve this problem is using the Coverage-Directed Assignment (CD-A) method.

5.1 The Principles of CD-A

The principles of a Coverage-Directed Assignment method are based on a simple fact: when all the rows of the **C** matrix are assigned to the rows of the **T** matrix (e.g., after the column and row matching) the Boolean minimization has to be performed. During this process implicants are sequentially added to the solution, until all the ones in the output matrix (**T** matrix) are covered by them – see the covering problem solution. Thus for each implicant the set of **T** matrix ones it covers is evaluated in the minimization process. Let us denote this set as a *coverage* of an implicant, the *coverage* of the output matrix will be a set of the coverages of all the implicants in the solution.

In the Coverage-Directed Assignment method we proceed backwards – first, we find a coverage of the output matrix and then we find the implicants that meet this coverage. Then, the Boolean minimization process can be completely omitted and even performing the row assignment is not necessary – the very implicants are produced by this method. The final PLA matrix is produced by joining the implicants with their coverage.

The whole process of the Coverage-Directed Assignment is illustrated by Figure 5.1.

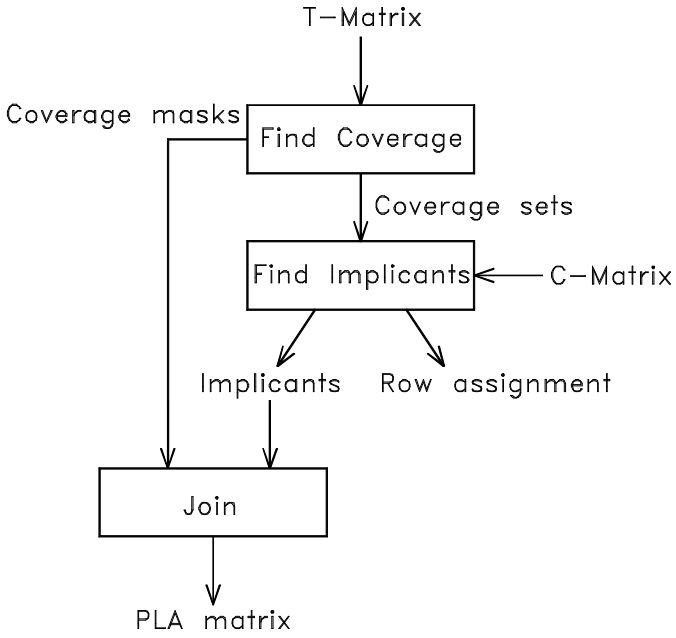

Figure 5.1: CD-A Structure

5.2 Find Coverage Phase

In this phase we try to find the coverage of the **T** matrix, thus the implicants that cover all ones in this matrix. But, the structure of these implicants is not known yet, only their properties are determined – namely the coverage sets together with the coverage masks.

Definition

Let t_i be an implicant. The *coverage set* $C(t_i)$ of the implicant t_i is a set of such vectors (rows) of the **T** matrix, that at least one one in the vector is covered by this implicant. In other words, the coverage set is a set of vectors of the output matrix for which t_i is an implicant for at least one output variable.

■

Definition

The *coverage mask* $M(t_i)$ of the implicant t_i is the set of columns of the **T** matrix, in which all vectors included in $C(t_i)$ contain ones.

The coverage mask $M(t_i)$ can also be expressed as a vector in the output matrix respective to the term t_i. In the following text we will use both representations of the coverage mask.

■

Condition – validity of the coverage

For every t_i the following equation must be valid:

Let

$C(t_i) = \{c_1, c_2, ..., c_u\}$, where $0 \le c_i < s$ for all $1 \le i \le u$, s is the number of rows in the **T** matrix

and

$M(t_i) = \{m_1, m_2, ..., m_r\}$, where $m_i \in \{0, 1\}$ for all i, r is the number of columns in the **T** matrix.

Then for all elements $\{c_i, m_j\}$ from the Cartesian product $C(t_i) \times M(t_i)$ there must be

$\mathbf{T}[c_i, j] = m_j$.

■

Condition – completeness of the coverage

For the complete coverage of the **T** matrix all the ones in the matrix must be generated by the couples of the Cartesian product $C(t_i) \times M(t_i)$.

■

4.2.1 Illustrative Example

We will illustrate the whole CD-A process by a simple example. Let us have the following **T**- and **C**-matrices:

```
     A 10000              a 10000
     B 11100              b 11100
     C 00001              c 01100
     D 10101              d 01010
C =  E 01111         T =  e 00111
     F 01001              f 00000
     G 01110              g 00011
     H 10110              h 01101
     I 00110              i 10111
     J 11010              j 10100
```

Figure 5.2. The C- and T-matrices

In this phase we will find the coverage of the **T** matrix:

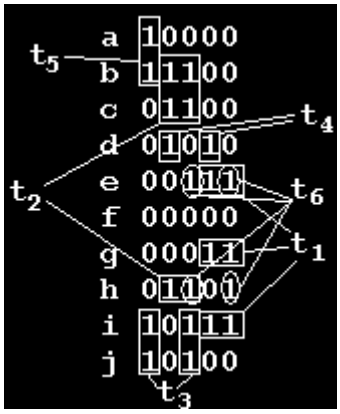

Figure 5.3. The coverage of the T matrix

This coverage contains 6 implicants t_1-t_6, the ones in the matrix that are covered by the individual implicants are shown in Fig. 5.3. All the ones are covered, while no zero is covered, thus this coverage is complete and valid. The coverage sets and coverage masks are shown in the following Table. Note, that the rows of the matrix are labeled a-j instead of numbering, as later the numbers could be confusing.

Table 5.1. The coverage sets and masks

Implicant	$C(t_i)$	$M(t_i)$
t_1	{e, g, i}	{0, 0, 0, 1, 1} ≡ 00011
t_2	{b, c, h}	{0, 1, 1, 0, 0} ≡ 01100
t_3	{i, j}	{1, 0, 1, 0, 0} ≡ 10100
t_4	{d}	{0, 1, 0, 1, 0} ≡ 01010
t_5	{a, b}	{1, 1, 0, 0, 0} ≡ 11000
t_6	{e, h}	{0, 0, 1, 0, 1} ≡ 00101

4.2.2 The Find Coverage Algorithm

Obviously, there exist many possible covers of a particular function. There are several trivial solutions: one of them is the coverage, where each single one of the matrix is covered by one coverage set. Thus, there are as many coverage sets, as there are ones in the **T** matrix. This solution is, however, extremely inefficient.

Another trivial solution is a coverage, in which each vector of the **T** matrix is covered by exactly one coverage set (and thus by a single term). Then each of the sets contains exactly one vector and their number is equal to the number of rows of the **T** matrix. As all these coverage sets have empty intersections, it is always possible to find the corresponding implicants and thus the size of such a coverage determines the *upper bound* of the number of implicants (see the following Subsection).

The third special case is the situation where all the ones in a column of the **T** matrix are covered by a single implicant. This corresponds to the column match discussed in Section 3. Thus, the column matching method is just a special case of a CD-A.

Finding the *minimum cover*, i.e., the minimum number of implicants is a NP hard problem, thus some heuristic must be used to generate the coverage sets. In praxis, we use a heuristic that sequentially tries to find the coverage sets that cover the maximum yet uncovered ones in the **T** matrix. At the beginning, the **T** matrix vector with the maximum number of ones is selected and included into the first coverage set, while the coverage mask is set equal to this vector. Next, we systematically try to add such vectors to this set, so the number of ones covered by them is increasing. After completing one set, another coverage set is constructed, until all the ones are covered. Note that the corresponding coverage mask is being reduced after every inclusion of the vector, and thus the number of covered ones may decrease after the vector selection.

4.3 Find Implicant Phase

Now, when the coverage of the **T** matrix is found, the implicants that have the required coverage sets should be generated from the **C** matrix. Such implicants will be denoted as implicants that *fulfil* this coverage. The properties of the implicants that fulfil the coverage are studied in this section.

This phase is completely independent on the **T** matrix, only its coverage sets are processed and only their structures are important. The implicant generation is based on the following Definitions and Theorems.

In the following text the rows C[*i*] of the **C** matrix will be taken as minterms.

Definition

Let us introduce an *inclusion function* $\varphi(t_1, t_2)$ for two terms t_1 and t_2 of the same dimension:

$\varphi(t_1, t_2) = 1$ if $t_2 \subseteq t_1$, thus t_2 is included in t_1.

$\varphi(t_1, t_2) = 0$ otherwise.

■

Theorem 1

The implicant t_i fulfils the coverage $C(t_i)$ if the number of minterms in the **C** matrix that are included in t_i is equal to the size of the $C(t_i)$ set, i.e.:

$$\sum_{j=0}^{p} \varphi(t_i, \mathbf{C}[j]) = |C(t_i)|$$

■

Proof

If a minterm of the **C** matrix is included in a term t_i, the term will have a value 1 for the values of the input variables corresponding to this minterm. If exactly j minterms of the **C** matrix are included in t_i, the term will have a value 1 for j rows of the **C** matrix. The size of the $C(t_i)$ set gives the number of rows of the **T** matrix, for which the term t_i has at least one value 1. Thus, if all the rows of the **C** matrix need to be assigned to the rows of the **T** matrix, the relation stated above must be valid. In the final row assignment the rows of the **C** matrix that are contained in t_i will be assigned to the rows of the **T** matrix included in the $C(t_i)$ set.

■

This condition for selecting the implicants is still not sufficient. It often happens that one row of the **T** matrix is covered by more than one implicant. Thus, the terms covering this row need to have a non-empty intersection. The number of the minterms of the **C** matrix that are included in this intersection must be equal to the size of the set that is an intersection of the corresponding coverage sets. Next, implicants that

correspond to the sets that have an empty intersection must also have an empty intersection.

These observations can be concluded as follows: when we search for the implicants that fulfil the coverage, first we determine all the intersections of the coverage sets. Terms that fulfil the coverage requirements must have the same number of **C** matrix minterms included in them as the sizes of the coverage sets are, as well as the number of **C** matrix minterms included in their intersections must be equal to the sizes of their corresponding coverage sets intersections.

4.3.1 Illustrative Example

Now we will show the implicant generation phase in our continuing example. First, we compute the intersections of the coverage sets:

$C(t_1) \cap C(t_3) = \{i\}$

$C(t_1) \cap C(t_6) = \{e\}$

$C(t_2) \cap C(t_5) = \{b\}$

$C(t_2) \cap C(t_6) = \{h\}$

Other set intersections than those listed above are empty.

We start, e.g., with the term t_1. We try to find a term that includes exactly 3 minterms from the **C** matrix (because $|C(t_1)| = 3$). The possible term is:

t_1 = (-01--)

Now the minterms that are contained in this term are assigned to it:

```
     A 10000
     B 11100
     C 00001
     D 10101 -> t1
C =  E 01111
     F 01001
     G 01110
     H 10110 -> t1
     I 00110 -> t1
     J 11010
```

Figure 5.4

The term t_2 has to contain 3 minterms, while no vector assigned to t_1 must be assigned to it, as $C(t_1) \cap C(t_2) = \varnothing$. The possibility is:

t_2 = (--00-)

C =
A 10000 -> **t_2**
B 11100
C 00001 -> **t_2**
D 10101 -> t_1
E 01111
F 01001 -> **t_2**
G 01110
H 10110 -> t_1
I 00110 -> t_1
J 11010

Figure 5.5

The size of $C(t_3)$ is equal to 2 and $|C(t_1) \cap C(t_3)| = 1$, thus exactly one minterm assigned to t_3 has to be assigned to t_1 as well, while the second one need not be assigned yet. We will select:

t_3 = (--10-)

C =
A 10000 -> t_2
B 11100 -> **t_3**
C 00001 -> t_2
D 10101 -> t_1, **t_3**
E 01111
F 01001 -> t_2
G 01110
H 10110 -> t_1
I 00110 -> t_1
J 11010

Figure 5.6

We continue this way, until all implicants are found. The result will be as follows:

t_1 = (-01--)
t_2 = (--00-)
t_3 = (--10-)
t_4 = (---11)
t_5 = (1-0--)
t_6 = (00---)

C =
A 10000 -> t_2, t_5
B 11100 -> t_3
C 00001 -> t_2, t_6
D 10101 -> t_1, t_3
E 01111 -> t_4
F 01001 -> t_2
G 01110
H 10110 -> t_1
I 00110 -> t_1, t_6
J 11010 -> t_5

Figure 5.7. The resulting terms

5.3.2 Implicant Generation Algorithm

Unlike the previous phase, where the solution can always be found, the implicant generation phase may not succeed in finding a solution. Firstly, as the implicants are being produced incrementally, it may happen that in some phase of the process the appropriate implicant cannot be found. Then, we use a backtracking approach; one of the previously constructed implicants is deleted and the search continues.

Second, and the most serious problem is that for a given coverage and the **C** matrix a solution need not exist. In this case a new coverage must be generated, i.e., the previous phase must be re-run.

Until now we have not yet explained the way in which the appropriate implicants are looked for. The method is based on finding proper supercubes of the minterms of the **C** matrix. In our example, when the t_3 term is being searched for after t_1 and t_2 were selected, it has to contain one minterm contained in t_3 (i.e., D, H or I) and one minterm, which is not yet contained in any previously created terms (i.e., B, E, G or J). Thus, t_3 must be a supercube of any two minterms from these two sets. We find the minimum supercubes of all the pairs of minterms from these two sets and try to find the appropriate term among them or their supercubes (expanded terms). Note that the found supercubes need not meet the requirements for the searched term; they only meet the requirements for minimum number of contained minterms, however they can contain more terms than needed and thus they are rejected. The minimum supercubes of the (D, H, I) and (B, E, G, J) minterm couples are listed below:

```
D-B: 1-10-    H-B: 1-1-0    I-B: --1-0
D-E: --1-1    H-E: --11-    I-E: 0-11-
D-G: --1--    H-G: --110    I-G: 0-110
D-J: 1----    H-J: 1--10    I-J: ---10
```

Figure 5.8. Candidates for the implicants

We can see that the term t_3 = (--10-) was created by expansion of the supercube of the D and B minterms. Similarly, the term t_1 was created by expansion of one of the supercubes of any three minterms.

5.4 The Final Phase

After the implicants are produced, the only thing we need to do is to assign these implicants to the outputs, i.e., to create the PLA matrix. This can be done simply by joining these implicants with their coverage masks found in the Find Coverage phase. The final PLA matrix and the SOP forms of the example circuit are shown below.

```
-01-- 00011
--00- 01100
--10- 10100
---11 01010
1-0-- 01010
00--- 00101
```

$y_0 = x_2x_3'$

$y_1 = x_2'x_3' + x_3x_4 + x_0x_2'$

$y_2 = x_2'x_3' + x_2x_3' + x_0'x_1'$

$y_3 = x_1'x_2 + x_3x_4 + x_0x_2'$

$y_4 = x_1'x_2 + x_0'x_1'$

Figure 5.9. The final result

5.5 Assignment of Rows

As was said before, in this method it is not necessary to determine which **C** matrix rows are assigned to which rows of the **T** matrix, as this knowledge is not needed to produce the final circuit. However, sometimes we may want to know the order in which the test patterns are generated and then the row assignment must be found.

For the following discussion the term of a reduced coverage set has to be introduced.

Definition

The *reduced coverage set* $C'(t_i)$ of a coverage set $C(t_i)$ will be defined as follows:

$$C'(t_i) = C(t_i) - \bigcup_{\substack{j=1 \\ j \neq i}}^{k} C(t_i) \cap C(t_j)$$

where k is the total number of coverage sets.

■

By this definition, the reduced coverage set contains only those vectors, that are not included in any intersection of the $C'(t_i)$ with any other set(s).

After computing all the reduced coverage sets and the reduced coverage sets of all their intersections each vector of the **T** matrix is contained in exactly one of the resulting sets (or in none). The reduced sets and the assignment of the vectors to the sets in our example is shown below:

Table 5.2. The final result

$C'(t_1) = \{g\}$

$C'(t_2) = \{c\}$

$C'(t_3) = \{j\}$

$C'(t_4) = \{d\}$

$C'(t_5) = \{a\}$

$C'(t_6) = \varnothing$

$C(t_1) \cap C(t_3) = \{i\}$

$C(t_1) \cap C(t_6) = \{e\}$

$C(t_2) \cap C(t_5) = \{b\}$

$C(t_2) \cap C(t_6) = \{h\}$

T matrix row	contained in
a	$C'(t_5)$
b	$C(t_2) \cap C(t_5)$
c	$C'(t_2)$
d	$C'(t_4)$
e	$C(t_1) \cap C(t_6)$
f	-
g	$C'(t_1)$
h	$C(t_2) \cap C(t_6)$
i	$C(t_1) \cap C(t_3)$
j	$C'(t_3)$

Now we have to find the assignment of the **C** matrix rows to the reduced coverage sets and the set intersections. This assignment can be easily observed from the assignment of the terms to the **C** matrix rows after the implicant generation phase; the

rows are covered by the terms and their intersections in order to meet the coverage requirements:

Table 5.3. Assignment of the C matrix rows

```
     A 10000 -> t2, t5
     B 11100 -> t3
     C 00001 -> t2, t6
     D 10101 -> t1, t3
C =  E 01111 -> t4
     F 01001 -> t2
     G 01110
     H 10110 -> t1
     I 00110 -> t1, t6
     J 11010 -> t5
```

Set	**C** matrix row
$C'(t_1)$	H
$C'(t_2)$	F
$C'(t_3)$	B
$C'(t_4)$	E
$C'(t_5)$	J
$C'(t_6)$	-
$C(t_1) \cap C(t_3)$	D
$C(t_1) \cap C(t_6)$	I
$C(t_2) \cap C(t_5)$	A
$C(t_2) \cap C(t_6)$	C
none	G

Then, the final row assignment is done by simple joining the two assignments to the coverage sets. If the **C** matrix rows are ordered the same way as they are produced by a PRPG, the order of the **T** matrix rows in this assignment determines the order in which the test patterns are fed into the CUT.

Table 5.4. The final assignment

Set	**C** matrix row	**T** matrix row
$C'(t_1)$	H	g
$C'(t_2)$	F	c
$C'(t_3)$	B	j
$C'(t_4)$	E	d
$C'(t_5)$	J	a
$C'(t_6)$	-	-
$C(t_1) \cap C(t_3)$	D	i
$C(t_1) \cap C(t_6)$	I	e
$C(t_2) \cap C(t_5)$	A	b
$C(t_2) \cap C(t_6)$	C	h
none	G	f

```
A 10000   b 11100
B 11100   j 10100
C 00001   h 01101
D 10101   i 10111
E 01111   d 01010
F 01001   c 01100
G 01110   f 00000
H 10110   g 00011
I 00110   e 00111
J 11010   a 10000
```

5.6 Generalized Coverage-Directed Assignment

As well as the column matching based method, also the CD-A approach can be easily modified for a longer PRPG run, when there are more **C** matrix rows than the **T** matrix rows ($p > s$). Again, only the suitable **C** matrix rows are assigned to all the **T** matrix rows, while the remaining rows in **C** are ignored.

Let us remark, that this CD-A modification consists only in altering the Implicant Generation phase, as the Find Coverage phase is completely independent on the **C** matrix. Thus, even if the number of the **C** matrix rows is greater than necessary, the number of implicants in the final solution cannot be reduced directly – it is determined purely from the Find Coverage phase. However, as was said before, the Implicant Generation phase need not always be successful for a given coverage. So, the improvement consists in the fact, that there may exist a better coverage of a given **T** matrix that cannot be implemented in the standard way (the Implicant Generation phase fails for this coverage), but we can find the appropriate implicants when the generalized CD-A method is used. The second improvement is that often "bigger" implicants can be found, containing fewer literals.

The modification consists above all in modifying Theorem 1. As not all **C** matrix minterms will be included in the solution, the condition of equality of the size of the coverage set and the number of minterms covered by the searched term is not required; the term may cover more minterms, while the excessive ones are omitted in the final solution. Thus, the equation from Theorem 1 is modified to:

$$\sum_{j=0}^{p} \varphi(t_i, \mathbf{C}[j]) \geq |C(t_i)|$$

Similarly, this modification applies also to the intersections of the coverage sets, except of the fact that some **C** matrix vectors can be assigned to non-existent coverage set intersections. These vectors will also be omitted in the final assignment.

However, these conditions are not yet sufficient – it may happen that the candidate term contains so many **C** matrix minterms, that there are not enough "free" minterms for the not yet processed coverage sets. So the additional condition must apply:

$$p - \sum_{j=0}^{p} \varphi(t_i, \mathbf{C}[j]) \geq \left| \bigcup_{j \notin \mathbf{P}} C(t_j) - \bigcup_{j \in \mathbf{P}} C(t_j) \right|$$

where **P** is the set of the already processed implicants.

5.6.1 Generalized CD-A Example

The principles of the Generalized CD-A can be illustrated with an example. The **T** matrix is kept the same as before, and the previously found coverage is also retained. The new **C** matrix has 20 vectors and 5 variables – thus it is representing a 5-stage PRPG running for 20 cycles:

```
     A 01111
     B 01101
     C 00011
     D 01011
     E 00001
     F 10001
     G 11001
     H 01110
     I 01000
C =  J 11000
     K 10111
     L 11010
     M 00100
     N 10011
     O 00010
     P 01001
     Q 00000
     R 01100
     S 00101
     T 10100
```

Figure 5.10. The C matrix

We will give the solution without explaining the particular steps, as the procedure is very similar to the classical CD-A algorithm. Figure 5.11 shows the resulting terms with the **C** matrix rows covered by them. Note that some minterms are contained in the intersections of terms that do not correspond to any coverage set intersections. Those minterms represent the idle cycles of the PRPG.

t_1 = (---1-)
t_2 = (--00-)
t_3 = (1-1--)
t_4 = (-1---)
t_5 = (0---1)
t_6 = (1-0--)

A 01111 -> t_1, t_4, t_5
B 01101 -> t_4, t_5
C 00011 -> t_1, t_5
D 01011 -> t_1, t_4, t_5
E 00001 -> t_2, t_5
F 10001 -> t_2, t_6
G 11001 -> t_2, t_4, t_6
H 01110 -> t_1, t_4
I 01000 -> t_2, t_4
C = J 11000 -> t_2, t_4, t_6
K 10111 -> t_1, t_3
L 11010 -> t_4, t_6
M 00100
N 10011 -> t_1, t_6
O 00010 -> t_1
P 01001 -> t_2, t_4, t_5
Q 00000 -> t_2
R 01100 -> t_4
S 00101 -> t_5
T 10100 -> t_3

Figure 5.11. Assignment of the C matrix rows to the terms

The following Table shows the resulting assignment of the **C** matrix rows to the reduced coverage sets and to the **T** matrix rows:

Table 5.5. The final assignment

Set	**C** matrix row	**T** matrix row
$C'(t_1)$	O	g
$C'(t_2)$	Q	c
$C'(t_3)$	T	j
$C'(t_4)$	R	d
$C'(t_5)$	S	a
$C'(t_6)$	-	-
$C(t_1) \cap C(t_3)$	K	i
$C(t_1) \cap C(t_6)$	N	e
$C(t_2) \cap C(t_5)$	E	b
$C(t_2) \cap C(t_6)$	F	h
none	M	f

```
A 01111
B 01101
C 00011
D 01011
E 00001     b 11100
F 10001     h 01101
G 11001
H 01110
I 01000
J 11000
K 10111     i 10111
L 11010
M 00100     f 00000
N 10011     e 00111
O 00010     g 00011
P 01001
Q 00000     c 01100
R 01100     d 01010
S 00101     a 10000
T 10100     j 10100
```

6 The BOOM Minimizer

As was said before, after the column matching and the row assignment, the logic of the output decoder has to be minimized to obtain the final PLA circuit. However, in most cases this is not a trivial problem to solve, as the numbers of input and output variables are often quite large. The current Boolean minimizers often were not able to complete the minimization in a reasonable time. The number of prime implicants of such functions is extremely large, and thus performing the exact minimization [McC56] is not possible at all. Thus, an efficient *heuristic* Boolean minimizer had to be used. The second feature required for such a minimizer is the capability to handle functions with a large portion of don't cares (highly unspecified functions) without enumerating them – only the on-sets and off-sets of the functions are explicitly defined.

The systematic Boolean minimization methods mostly copy the structure of the original method by Quine and McCluskey [Qui52, McC56], implementing the 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 and thus all the PIs need not be evaluated.

One of the most successful Boolean minimization methods is ESPRESSO and its later improvements. The original ESPRESSO generates near-minimal solutions, ESPRESSO-EXACT [Rud87] was developed in order to improve the quality of the results, mostly at the expense of 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].

A sort of combination of PI generation with solution of the CP, leading to a reduction of the total number of PIs generated, is also used in our BOOM (BOOlean Minimizer) approach used here. An important difference between the approaches of ESPRESSO and BOOM is the way 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. BOOM, on the other hand, uses the input sets (on-set and off-set) only as a reference that lead the algorithm to the correct solution. The second main difference is the top-down approach in generating implicants. Instead of expanding the source cubes in order to obtain better coverage, BOOM reduces the universal *n*-dimensional hypercube until it no longer intersects the off-set, while it covers the most terms of the source function. Some features of the proposed method were published in several conference proceedings [1-7].

6.1 BOOM Structure

When minimizing a single-output function, the BOOM system uses the following three phases:

1. *Coverage-Directed Search* (CD-Search) that generates implicants needed to cover the function
2. *Implicant Expansion* (IE) expands those implicants into primes (PIs)
3. *Solution of the covering problem* (CP).

For multi-output functions, instead of phase 3, phases 4, 5 and 6 are executed:

4. *Prime Implicant Reduction* (IR) generates the group implicants from the PIs
5. *Solution of the group covering problem*
6. *Output Reduction* (OR), which eliminates redundant implicants of individual outputs (partially corresponding to ESPRESSO's MAKE_SPARSE procedure [Bra84]).

All these parts will be briefly described in the following Subsections. Fig. 6.1 shows the block schematic of the BOOM system:

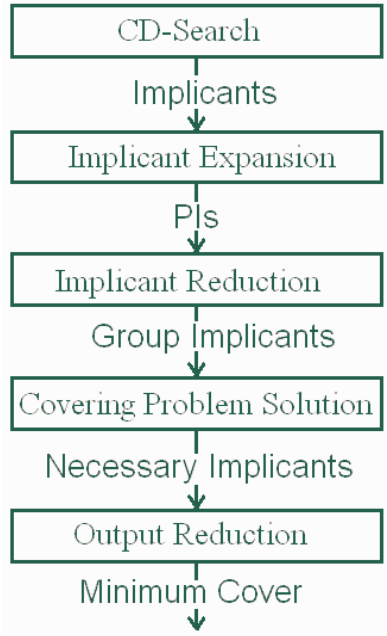

Figure 6.1. Structure of BOOM

6.2 Coverage-Directed (CD) Search

The idea of confining the implicant generation to those really needed gave rise to the CD-Search method, which is the most innovative feature of the BOOM system. It consists in a directed search for the most suitable literals that should be added to some previously constructed term in order to convert it into an implicant of the given function. Thus instead of increasing the dimension of an implicant starting from a 1-minterm (or any other 1-term given in the function definition), we reduce the *n*-dimensional hypercube by adding literals to its term, until it becomes an implicant of the given function. This happens at the moment when this hypercube does not intersect with any 0-term any more.

The implicant generation method aims at finding a hypercube that covers as many 1-terms as possible. We start by selecting the most frequent input literal from the given on-set. The selected literal describes an *n-1* dimensional hypercube, which may be an

implicant, if it does not intersect with any 0-term. If there are some 0-minterms covered, we add one more literal and verify whether the new term already corresponds to an implicant. After each literal selection we temporarily remove from the on-set the terms that cannot be covered by any term containing the selected literal - the terms containing that literal with the opposite polarity. In the remaining on-set we repetitively find the most frequent literal and include it into the previously found product term until it is an implicant. Then we remove from the original on-set the terms covered by this implicant. Thus we obtain a reduced on-set containing only uncovered terms. Now we repeat the procedure from the beginning and apply it to the uncovered terms, selecting the next most frequently used literal, until the next implicant is generated. In this way we generate new implicants, until the whole on-set is covered. The algorithm that takes the on-set (F) and offset (R) as an input and produces a set of product terms (H) covering all 1-terms and intersecting no 0-term can be described by the following pseudo-code:

Algorithm 6.1

```
CD_Search(F,R) {
  H = ∅
  do
    F′ = F
    t = true
    do
      v = most_frequent_literal(F′)
      t = t AND v
      F′ = F′–cubes_not_including(t)
    while (t ∩ R ≠ ∅)
    H = H ∪ t
    F = F - F′
  until (F == ∅)
  return H
}
```

6.2.1 CD-Search Example

Let us have a partially defined Boolean function of ten input variables $x_0..x_9$ and ten defined minterms given by a truth table in Tab. 6.1. The 1-minterms are highlighted.

Table 6.1

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

As the first step we count the occurrence of literals in the 1-minterms. The "0"-line and "1"-line in Tab. 6.2 give counts of x_i' and x_i literals respectively. In this table we select the most frequent literal.

Table 6.2

```
var: 0123456789
0:   3435322444
1:   3231344222
```

The most frequent literal is x_3' with five occurrences. This literal alone describes a function that is a superset of 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 our search by suppressing 1-minterms containing the selected literal with the opposite polarity (in Tab. 6.3 shaded dark). An implicant containing a literal x_3' cannot cover the 5th minterm, because it contains the x_3 literal. Thus, we temporarily suppress this minterm. In the remaining 1-minterms we find the most frequent literal.

Table 6.3

```
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 must be applied. We use these literals tentatively as implicant builders and create four product terms using the previously selected literal x_3': x_3'x_1', x_3'x_5, x_3'x_6, x_3'x_7'. Then we check 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. Now one of the remaining 3 terms representing implicants must be chosen. We should choose a term covering the maximum of yet uncovered 1-minterms (in Tab. 6.3 shaded lightly). As each of these implicants covers four 1-minterms, we can select randomly – 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. 6.4. We omit minterms covered by the selected implicant x_3'x_6 (dark shading) and select the most frequent literal in the remaining minterms.

Table 6.4

```
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. 6.4, we have four equal possibilities, so we choose one 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 $x_3'x_6 + x_5'x_6'$.

6.3 Implicant Expansion (IE)

The implicants generated during the CD-Search need not be prime. To make them prime, we have to increase their size by IE, which means by removing literals (variables) from their terms. When no literal can be removed from the term any more, we get a PI. The expansion of implicants into PIs can be done by several methods differing in complexity and quality of results obtained. We tested several approaches, from the simplest sequential search (which is linear) to the most complex exhaustive (exponential) search.

A **sequential Search** systematically tries to remove from each term all literals one by one, whereas the first literal is chosen randomly. Every removal is made permanent if no 0-minterm is covered. Only one PI is generated from each implicant, even if it could yield more PIs. A Sequential Search obviously does not reduce the number of product terms. On the other hand, experimental results show that it reduces the number of literals by approximately 25%.

With a **Multiple Sequential Search** we try all possible starting positions within an implicant, which thus expands into several PIs. This method produces more primes than a Sequential Search, while the time complexity is acceptable.

Even the Multiple Sequential Search algorithm cannot expand an implicant into all possible PIs. To do so, an **Exhaustive Implicant Expansion** must be used. Using recursion or queue, all possible literal removals are then tried until all primes are obtained. Unfortunately, the complexity of this algorithm is exponential.

All these expansion strategies have been tested and evaluated from the point of view of runtime and result quality. Finally the Multiple Sequential Search was selected as the best method for standard problems.

The Implicant Expansion phase is more thoroughly described in [3].

6.4 Solution of the Covering Problem

The quality of the final solution strongly depends on the CP solution algorithm. An efficient CP solution algorithm has to be used especially in connection with the iterative minimization (see Subsection 6.6). With a large number of PIs an exact solution is impossible and some heuristic must be used. Here the large number of implicants may misguide the CP solution algorithm and thereby lead to a non-minimal solution.

An exact CP solution is mostly rather time-consuming, especially when it is performed after several iterations during which many implicants were accumulated. In this case, a heuristic approach is the only possible solution. Out of several possible approaches we used two. The first one, denoted as the LCMC cover (Least Covered, Most Covering) is a common heuristic algorithm for solution of the covering problem. The implicants covering minterms covered by the lowest 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, 6]. This means that a weight is assigned to every implicant, which expresses its potential contribution to the minimal solution. For every term to be covered the weight of each unity entry is computed as a reciprocal value of the number of ones. The weights of all ones connected with one implicant are then summed up and assigned to the implicant as its weight. The implicant with the highest weight is then selected for the solution.

6.5 Minimization of Multi-Output Functions

6.5.1 Prime Implicant Reduction

When minimizing a multi-output function, each of the output functions is first treated separately. After performing the CD-search and IE phases we have a set of PIs sufficient for covering all the output functions. However, to obtain the minimum solution we may need group implicants, i.e., implicants of more than one output function that are not primes of any.

During the implicant reduction all obtained primes are tried for reduction (by adding some literals) in order to become implicants of more output functions. The method of implicant reduction is similar to a CD-Search. Literals are repetitively added to each term until there is no chance that the implicant will be used for more functions. We prefer literals that prevent intersecting with most of the 0-terms of all functions. When no further reduction yields any possible improvement, the reduction is stopped, the implicant is recorded and assigned to all functions, whose off-set it does not intersect.

6.5.2 Solution of the Group Covering Problem

After assigning implicants to the output functions the group covering problem is solved. The solution in this case is a set of implicants needed to cover all output functions. For each output we may find all implicants that do not intersect the off-set of the output function.

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. Sometimes this is harmless (e.g., in PLAs), or it can even prevent hazards. Nevertheless, for hardware-independent minimization the redundant outputs should be removed. This is done at the end of the minimization by solving covering problems for each of the output functions separately.

6.6 Iterative Minimization

When selecting the most frequent literal during the CD search, it may happen that two or more literals have the same frequency of occurrence. When no other criterion can be applied to select one literal, the BOOM system chooses at random. Thus there is a chance that repeated application of the same procedure to the same problem would yield different solutions.

The iterative minimization concept takes advantage of the fact that each iteration produces a new set of implicants satisfactory for covering all minterms. The newly created implicants are added to the previous ones and the covering problem is solved using all of them. The 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. 6.2 (thin line). This curve plots the values obtained during the solution of a problem with 20 input variables and 200 minterms. Theoretically, the more primes we have, the better the solution that can be found, but the maximum set of primes is often extremely large. In reality, the quality of the final solution improves rapidly during the first few iterations and then remains unchanged, even though the number of PIs grows further. This fact can be observed in Fig. 6.2 (thick line).

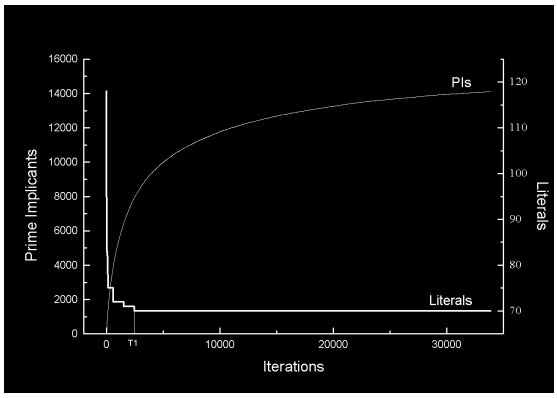

Figure 6.2. Growth of PI number and decrease of SOP length during iterative minimization

When the solution meets the requirements, the minimization is stopped. The whole iterative minimization process can be illustrated by the following algorithm. The inputs are the on- and off-sets of the source multioutput function, the output of the algorithm is a minimizad SOP form `G` meeting the `stop` condition.

Algorithm 6.2

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

6.7 Experimental Results

Many experiments have been made to evaluate the performance and applicability of the BOOM system. The tests were aimed above all at the problems of a very large size, especially for problems with a large number of input variables and an extremely high number of implicit don't cares – i.e., with only a few terms specified. For these problems the BOOM was found to be extremely efficient, as it highly overwhelms the other minimization methods like ESPRESSO.

The system was programmed in C++ Builder and tested on AMD Athlon 900MHz PC with 256 MB RAM. Some of the results are shown in following subsections.

6.7.1 The BOOM Benchmarks – Comparison with ESPRESSO

The effectiveness of the algorithm was tested on a set of artificial benchmarks we have developed for testing Boolean minimizers and other function manipulation tools capable of handling large Boolean functions. As the benchmarks were made in order to test the capabilities of the BOOM minimizer, we have named them BOOM Benchmarks [8]. The current benchmark set consists of 720 test problems of various sizes and with various portions of don't cares in their terms. For each problem size ten benchmarks were produced. The benchmark files are specified in a standard Berkeley PLA file format (see the Appendix ?), where the on-sets and off-sets of the function are specified, the don't care sets are not explicitly defined (type fr). Let us note that this format fully corresponds to the truth table specification of a highly unspecified Boolean function obtained, e.g., after the column matching algorithm. The BOOM benchmarks can be downloaded from the address [BENCH].

In this subsection we present a comparison of the BOOM minimizer with the ESPRESSO performed on a subset of these benchmarks. The test functions contain in the average 20% of don't cares in the input arrays of the care terms and 5 outputs. The

number of input variables (n) and the number of care terms (p) vary from 50 to 200. First, ESPRESSO was run in order to minimize each of the benchmark files and then BOOM was run iteratively until the result of the same of better quality in terms of the minimality was obtained. Then the runtimes of BOOM and ESPRESSO were compared. We can see from Table 6.5 that in most cases BOOM obtained even better results in a much shorter time than ESPRESSO did. Such cases are highlighted by shading. The quality of a result was measured by the total sum of the literals in the SOP form and the output cost (number of ones in the output PLA matrix), as it nearly exactly represents the number of 2-input gates (AND-OR or NAND, NOR) needed to implement the function. For each problem size 10 benchmarks were performed, the table contains the average values.

Table 6.5. Comparison of BOOM and ESPRESSO

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	427/1186,51 (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) 586/3872,66

Table entry format: BOOM results: *#of literals+output cost, solution time[s] (# of iterations)*

ESPRESSO results: *#of literals+output cost / solution time[s]*

6.7.2 Time Complexity Evaluation

As for most heuristic and iterative algorithms, it is difficult to evaluate the time complexity of the proposed algorithm exactly. We have observed the average time needed to complete one pass of the algorithm for various sizes of input truth table. The truth tables were generated randomly, while in this case only the single-output functions were studied and the input matrix contained only minterms. Fig. 6.3 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. 6.3 can be approximated with the square of the number of care minterms. Fig. 6.4 shows the runtime growth depending on the number of input variables (20-300) for various numbers of defined minterms (20-300). Although there are some fluctuations due to the low number of samples, the time complexity is almost linear. Fig. 6.5 shows a three-dimensional representation of the above curves.

These observations can be concluded as follows: when the BOOM minimizer is used in connection with the column matching algorithm (or another output decoder design method that requires a Boolean minimization at the end of the process), the number of the PRPG stages do not affect the design runtime significantly, while the length of the test affects the runtime to some extent, however not exponentially.

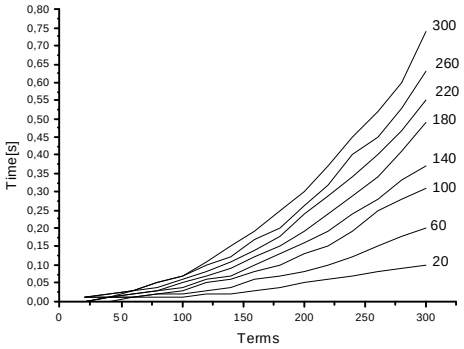

Figure 6.3 Time complexity (1)

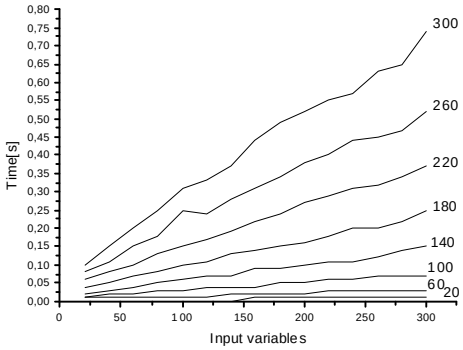

Figure 6.4 Time complexity (2)

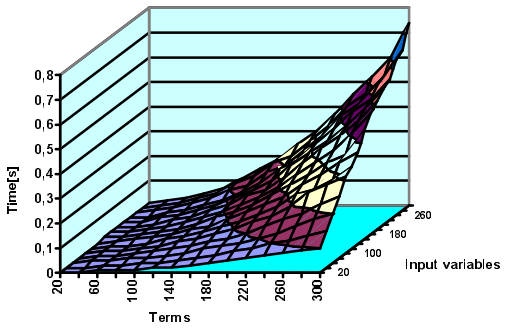

Figure 6.5 Time complexity (3)

7 Conclusions

A new test-per-clock BIST method for combinational circuits was described in this thesis. The pseudorandom patterns are generated by a PRPG and then transformed into test patterns that are pre-computed by an ATPG tool. In the method proposed here each of the test patterns has to be assigned to some of the PRPG patterns in order to generate the required test patterns by the test pattern generator. This transformation is done by a combinational block denoted as an output decoder. The method aims at simplifying this decoder as much as possible by proper assigning the test patterns to the PRPG patterns. However, the complexity of the decoder is strictly determined by the given test patterns. The assignment problem is treated as a general combinatorial problem and the algorithms can be exploited also in other areas of logic design.

Both the LFSR and cellular automata were tried to use as a PRPG and it was shown that the properties of the PRPG do dot influence the final solution significantly. Thus, both the LFSR generating polynomial and seed can be randomly chosen. Of course, such a generating polynomial must be chosen so that the LFSR produces the required number of code words, however, the polynomial needs not be irreducible. One possibility to improve the final result is to run the assignment algorithm repeatedly, while in each cycle different LFSR polynomial and/or seed is chosen. This principle was tested experimentally, but only a slight improvement was reached.

Two general assignment methods were proposed in this thesis: the column matching method and the novel coverage-directed assignment method. In the column matching method as many outputs of the decoder as possible are tried to directly match to the inputs, which minimizes the combinational logic needed to produce the matched outputs to zero. The logic producing the values of the unmatched output variables has to be synthesized by some Boolean minimizer. The principles of column matching algorithm were presented in [9].

Several approaches based on the decomposition into set systems and approaches based on the scoring matrix were proposed. The method is applicable to one-to-one matching where the PRPG runs only the most necessary number of cycles, as well as to the situation where idle PRPG cycles are inserted. The number of PRPG bits (stages) can be also freely chosen, as far as the PRPG produces the required umber of different code words. The test patterns can be presented as vectors containing don’t care states, as well as in their compacted form, where the test patterns are in the form of minterms and their number is reduced to minimum.

It was shown experimentally, that the more cycles the PRPG runs, the better solution in terms of the complexity of the combinational logic can be found. However, the length of a test should be reduced to minimum too, thus a trade-off has to be found. Similarly, increasing the width of a PRPG (number of PRPG stages) reduces the output decoder logic, but not significantly.

The method was tested on a set of combinational ISCAS benchmarks whose complete test sets were generated by an ATPG tool and the amount of logic needed to implement the output decoder was determined.

The coverage-directed assignment method produces the very implicants of the output decoder, thus no minimization is required. First, the coverage of the test matrix is determined, and then the implicants fulfilling this coverage are looked for. The assignment of the PRPG code words to the test patterns can be computed, however,

performing this process is not necessary for the synthesis of the output decoder. This method is the most general approach to the logic synthesis and it can be used in many other areas of logic design. Further research will be driven towards exploiting the method in a minimization of Boolean functions and generation of logic design benchmarks with defined properties.

The problem of finding the implicants fulfilling the required coverage is very complex. For larger problems finding the solution is very time-consuming and often the solution even may not exist. In this case another coverage has to be found and the process is repeated in order to obtain a result. The method fails for most of the ISCAS benchmarks so far and thus the results are not presented in the thesis. The development of algorithms that will allow us to handle more complex problems is yet in progress.

In connection with the pattern assignment methods an efficient Boolean minimizer BOOM was developed. Its most important features are its applicability to functions with several hundreds of input variables and very short minimization times for sparse functions, i.e., functions with only several care terms defined. The function to be minimized is described by the truth table where the on-set and off-set of the function is defined, whereas the don't care set need not be specified explicitly. The entries in the truth table may be minterms or terms of higher dimensions. The implicants of the function are constructed by reduction of *n*-dimensional cubes; hence the terms contained in the original truth table are not used as a basis for the final solution.

The properties of the BOOM minimization tool were demonstrated on examples. Its application is advantageous above all for problems with large dimensions and a large number of don't care states where it beats other methods, like ESPRESSO, both in minimality of the result and in runtime. The implicant generation method is very fast, hence it can easily be used in an iterative manner.

The principles of BOOM were published in many conference proceedings – see [1-7].

The BOOM minimizer together with the tool performing the pattern assignment using the proposed methods is provided along with this thesis, the usage of the tools is described in the Appendices.

References

[Aga81] Agarwal, V.K., - Cerny, E.: Store and Generate Built-In Testing Approach, Proc. of FTCS-11, pp. 35-40, 1981

[Aga93] Agarwal, V.K. – Kime, C.R. - 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] Aloke, K. – Chaudhuri, D.P.: 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] AlShaibi, M.F. - Kime, C.R.: Fixed-Biased Pseudorandom Built-In Self-Test for Random Pattern Resistant Circuits, Proc. of International Test Conference, pp. 929-938, 1994

[Bra84] Brayton, R.K., et al.: Logic minimization algorithms for VLSI synthesis. Boston, MA, Kluwer Academic Publishers, 1984

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

[Brg85] Brglez, F. - Fujiwara, H.: 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

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

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

[Gar79] Gary, M. – Johnson, D.: Computers and Interactibility: A guide to the theory of NP Completenes, Freeman, 1979

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

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

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

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

[Hel95] Hellebrand, S. 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

[Hel96] Hellebrand, S. - Wunderlich, H-J. - Hertwig, A.: Mixed Mode BIST Using Embedded Processors. Proc. of IEEE ITC, 1996

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

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

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

[McC84] McCluskey, E.J.: 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] McCluskey, E.J.: 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] McGeer, P. et al.: ESPRESSO-SIGNATURE: A new exact minimizer for logic functions, In Proc. of the Design Automation Conf.'93
[Nee93] Neebel, D.J. - Kime, C.R.: Inhomogeneous Cellular Automata for Weighted Random Pattern Generation, Proc. of International Test Conference, pp. 1013-1022, 1993
[Ngu87] Nguyen, L. – Perkowski, M. - Goldstein, N.: Palmini – fast Boolean minimizer for personal computers, In Proc. of the Design Automation Conf.'87, pp.615-621
[Nov98] Novák, O. – Hlavička, J.: Design of a Cellular Automaton for Efficient Test Pattern Generation. Proc. IEEE ETW 1998, Barcelona, Spain, pp. 30-31
[Nov99] Novák, O.: Weighted Random Patterns for BIST Generated in Cellular Automata, Proc. of 5-th IOLTW, Rhodes, Greece, July 1999, pp. 72-76
[Nov01] Novák, O. – Hlawiczka, A. at al.: Low Hardware Overhead Deterministic Logic BIST with Zero-Aliasing Compactor, Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'01), Gyor (Hungary), 18-20.4.2001, pp. 291-298
[Pom93] Pomeranz, I. - Reddy, S.M.: 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
[Qui52] Quine, W.V.: The problem of simplifying truth functions, Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531
[Rud87] Rudell, R.L. - Sangiovanni-Vincentelli, A.L.: Multiple-valued minimization for PLA optimization, IEEE Trans. on CAD, 6(5): 725-750, Sept.1987
[Rud89] Rudell, R.L: Logic Synthesis for VLSI Design, Ph.D. Thesis, UCB/ERL M89/49, 1989
[Ser75] Servít, M.: A Heuristic method for solving weighted set covering problems, Digital Processes, vol. 1. No. 2, 1975, pp.177-182
[Tou94] Touba, N.A. - McCluskey, E.J.: Transformed Pseudo-Random Patterns for BIST, CRC Technical Report No. 94-10, 1994
[Tou95a] Touba, N.A. - McCluskey, E.J.: Transformed Pseudo-Random Patterns for BIST, Proc. of VLSI Test Symposium, pp. 410-416, 1995
[Tou95b] Touba, N.A. : Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST, Proc. of International Test Conference, pp. 674-682, 1995
[Tou96] Touba, N.A. - McCluskey, E.J.: Altering a Pseudo-Random Bit Sequence for Scan-Based BIST, Proc. of International Test Conference, 1996, pp. 167-175
[Tro96] Trouborst, P.: LFSR Reseeding as a Component of Board Level BIST. Proc. Int' l. TesConf. 1996, Washington, D.C., pp. 58-67
[Wun87] Wunderlich, H.J.: Self-Test Using Unequiprobable Random Patterns, Proc. of FTCS-17, pp. 258-263, 1987
[Wun88] Wunderlich, H.J.: Multiple Distributions for Biased Random Test patterns, Proc. of International Test Conference, pp. 236-244, 1988.
[Zac95] Zacharia, N. – Rajski, J. – Tyszer, J.: Decompression of Test Data Using Variable-Length Seed LFSRs. Proc. VLSI Test Symp., pp. 426-433, 1995

[ATOM] http://www.crhc.uiuc.edu/IGATE/
[BOOM] http://service.felk.cvut.cz/vlsi/prj/BOOM/
[BENCH] http://service.felk.cvut.cz/vlsi/prj/BoomBench/
[ESPRESSO] http://eda.seodu.co.kr/~chang/ download/espresso/
[ISCAS] http://www.crhc.uiuc.edu/IGATE/

[1] Hlavička, J. - Fišer, P.: 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, pp.130-133

[2] Fišer, P. - Hlavička, J.: Efficient Minimization Method for Incompletely Defined Boolean Functions, Proc. 4th Int. Workshop on Boolean Problems, Freiberg (Germany), Sept. 21-22, 2000, pp. 91-98

[3] Fišer, P. - Hlavička, J.: 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

[4] Hlavička, J. - Fišer, P.: 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

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

[6] Fišer, P. - Hlavicka, J.: BOOM - a Boolean Minimizer. Research Report DC-2001-05, Prague, CTU Publishing House, June 2001, 37 pp.

[7] Hlavička, J. - Fišer, P.: BOOM - a Heuristic Boolean Minimizer, Proc. ICCAD-2001, San Jose, Cal. (USA), 4-8.11.2001, pp. 439-442

[8] Fišer, P. - Hlavička, J.: 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 (poster)

[9] Fišer, P. - Hlavička, J.: Column-Matching Based BIST Design Method, Proc. 7th IEEE Europian Test Workshop (ETW'02), Corfu (Greece), 26.-29.5.2002 (poster, in print)

APPENDIX A

The BOOM Minimizer User‘s Manual

BOOM 1.3 Manual

BOOM (BOOlean Minimization) is a tool for minimizing two-valued Boolean functions. The output is a near-minimal or minimal two-level disjunctive (SOP) form. The input and output of BOOM are compatible with Berkeley standard PLA format (see Appendix B).

BOOM 1.3 runs as a Win32 console application.

Minimum requirements:

- Microsoft Windows 95 or higher
- Intel Pentium processor or higher

Command line syntax

```
BOOM [options] source [destination]
```

OPTIONS

-CMn	Define CD-search mutations ratio n (0-100)
-RMn	Define implicant reduction mutations ratio n (0-100)
-Ex	Select implicant expansion type: 0 – Sequential search 1 – Distributed multiple IE 2 – Distributed exhaustive IE 3 – Multiple IE (default) 4 – Exhaustive IE
-CPx	Select the covering problem solution algorithm: 0 - LCMC 1 - Contributive selection (default) 2 - Contributive removal 3 - Exact

-Sxn	Define stopping criterion x of value n: t - stop after n seconds (floating point number is expacted) i - stop after n iterations (default is Si1) n - stopping interval equal to n the minimization is stopped when there is no improvement of the solution for n-times more iterations than it was needed for the last improvement q - stop when the quality of solution meets n more criteria can be specified at the same time
-Qx	Define quality criterion x: t - number of terms l - number of literals o - output cost b - number of literals + output cost (default)
-Mn	Specify the mutation rate for the CD-search in percents. An integer number in the range 0-100 is expected. Default is 0.
-Ex	Choose the Implicant Expansion method: 0 - Sequential Search (default) 1 - Distributed Multiple Expansion 2 - Distributed Exhaustive Expansion 3 - Multiple Implicant Expansion 4 - Exhaustive Expansion
-endcov	Cover at the end only
-single	Use single-output minimization only (good for PALs)
-c	Checks the input function for consistence, i.e., checks if the off-set doesn't intersect the on-set.

CD-Search Mutations

This argument specifies the ratio of forced randomness of the CD-search. In some cases, by increasing this value the better solution can be reached faster. For more detailed description see [5].

IR Mutations

This argument specifies the ratio of forced randomness of the IR phase. In some cases, by increasing this value the better solution can be reached faster. For more detailed description see [5].

Implicant Expansion Method

You can specify the implicant expansion method. For more detailed description see [3].

Covering Problem Solution

This switch selects the covering problem solution algorithm. For more detailed description see [6].

Stopping Criteria Specification

BOOM performs an iterative minimization. The more iterations are processed, the better the result that can be archieved. The number of iterations can be specified by the user.

Quality Criteria Specification

During iterative minimization the best result found so far is remembered. The measure of the quality can be specified by the user.

Cover at the End Only

When this option is selected, the covering problem is solved only once at the end of the minimizaion process. This option cannot be combined with optimization criteria based stopping condition.

Source

The source for minimization will be a PLA file. Type .fr is required (on-set and off-set specified).

Destination

The output of BOOM is a PLA file type .fd. This corresponds to the physical PLA representation of the function. When no destination is specified, the function is printed to the standard output.

EXAMPLE

We have a Boolean function of 4 input variables and 3 output variables described by the following Karnaugh maps:

1	1	1	X
0	0	1	0
0	0	1	0
0	1	0	X

0	1	0	X
0	1	1	1
1	1	1	0
1	0	1	X

0	0	1	X
0	0	0	1
0	1	1	0
0	1	0	X

This function can be described by a PLA file as follows:

```
.i 4
.o 3
.p 14
.type fr
1100 010
1011 010
0100 000
1110 000
1000 010
1111 111
0000 100
0011 101
0110 011
0101 010
0111 110
1101 011
0001 110
1001 101
.e
```

This will be an input for the BOOM minimizer. This function can be minimized by the following command:

```
BOOM fnct.pla
```

The function will be minimized to the minimal form:

```
.i 4
.o 3
.p 9
-111 110
00-- 100
1-00 010
0-10 011
001- 001
1-11 010
11-1 011
0-01 010
1001 101
.e
```

APPENDIX B

The PLA Format Description

PLA Format Accepted by BOOM

BOOM accepts as an input and also produces in the output a two-level description of a Boolean function. This is described as a character matrix (truth table) with keywords embedded in the input to specify the size of the matrix and the logical format of the input function.

KEYWORDS

The following keywords are recognized by BOOM. The list shows the probable order of the keywords in a PLA description. The symbol *d* denotes a decimal number and *s* denotes a text string. The minimum required set of keywords is .i, .o and .e. Both keywords .i and .o must precede the truth table.

.i *d*	Specifies the number of input variables (obligatory)
.o *d*	Specifies the number of output functions (obligatory)
.ilb *s1 s2 . . . sn*	Gives the names of the binary valued variables. This must come after .i. There must be as many tokens following the keyword as there are input variables
.ob *s1 s2 . . . sn*	Gives the names of the output functions. This must come after .o. There must be as many tokens following the keyword as there are output variables
.type *s*	Sets the logical interpretation of the character matrix. This keyword (if present) must come before any product terms. *s* is either fr or fd (which is default)
.p *d*	Specifies the number of product terms
.e (.end)	Marks the end of the PLA description

LOGICAL DESCRIPTION OF A PLA

When we speak of the ON-set of a Boolean function, we mean those minterms which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

A Boolean function can be described in one of the following ways:

1. By providing the ON-set. In this case the OFF-set can be computed as the complement of the ON-set and the DC-set is empty.

2. By providing the ON-set and DC-set. The OFF-set can be computed as the complement of the union of the ON-set and the DC-set. This is indicated with the keyword **.type fd** in the PLA file. This Boolean function specification uses BOOM as the output of the minimization algorithm.
3. By providing the ON-set and OFF-set. In this case the DC-set can be computed as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the "consistency check" option. This type is indicated with the keyword **.type fr** in the input file. This is the only possible Boolean function specification for the input of BOOM.

SYMBOLS IN THE PLA MATRIX AND THEIR INTERPRETATION

Each position in the input plane corresponds to an input variable where a 0 implies that the corresponding input literal appears complemented in the product term, a 1 implies that the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term.

With **type .fd** (the default), for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term has no meaning for the value of this function.

With **type .fr**, for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, and a - means this product term has no meaning for the value of this function.

Regardless of the type of PLA, a ~ implies the product term has no meaning for the value of this function.

EXAMPLE

A two-bit adder which takes in two 2-bit operands and produces a 3-bit result can be described completely in minterms as:

```
.i 4
.o 3
.p 16
0000 000
0001 001
0010 010
0011 011
0100 001
0101 010
0110 011
0111 100
1000 010
1001 011
1010 100
1011 101
1100 011
1101 100
```

```
1110 101
1111 110
.e
```

Note that BOOM **does not** accept all features of the current Berkeley PLA format. When any fatures of this format not described here are used, they are ignored or the error is returned.

APPENDIX C

The ASSIGN Program User‘s Manual

ASSIGN 1.0 Manual

The ASSIGN is a program that performs the row assignment between the **C** matrix and **T** matrix rows. The input formats are the pattern files, the output is a PLA file describing the resulting Boolean function.

ASSIGN 1.0 runs as a Win32 console application.

Minimum requirements:

- Microsoft Windows 95 or higher
- Intel Pentium processor or higher

Command line syntax

```
ASSIGN [options] C-matrix_source T-matrix_source
[destination]
```

OPTIONS

-Tx	Select assignment type: 0 - Scoring Matrix 1 - Column Matching - random selection 2 - Column Matching - heuristic selection (default) 3 - Column Matching – exact 4 - Column Matching - with DCs 5 - CD-A
-X	Extract column matches
-I	Forbid inverse matching

The Source Matrices

As the source matrices the *pattern files* are accepted. The pattern file consists of the listed patterns in each row only. For example the pattern file of a **C** matrix from the CD-A example looks as follows:

```
10000
11100
00001
10101
01111
01001
01110
10110
00110
11010
```

The Destination File

The destination file is a PLA file that represent the resulting function. If this parameter is omitted, the output will be written to the standard output.

Assignment Type

You can select by this option the assignment method:

0 – the scoring matrix approach (see Subsection 3.2). No column matches are being looked for, the assignment is performed in order to simplify the resulting PLA function. It cannot be used for a test containing don't cares. It can be used only for one-to-one matching.

1 - The column matching approach with a random column match selection. It is suitable for most of cases. It cannot be used for a test with don't cares.

2 - The column matching using a heuristics to find a proper column match. Sometimes it reaches the optimal result in a shorter time. It cannot be used for a test containing don't cares.

3 - The exact column matching always finds the maximum number of column matches, however it is sometimes extremely time-demanding. It cannot be used for a test containing don't cares.

4 - The column matching method exploiting don't cares. A heuristic is used to find the column matches.

5 - The Coverage-Directed Assignment method. This method cannot be used for a test containing don't cares. Unlike in other methods the output of this algorithm is a final PLA file (type fd), thus the post-process minimization is not required.

Extract Exact Matches

When this switch is present, the output variables that were matched with some inputs are omitted from the result, as they can be implemented as simple wires. Using this option sometimes significantly reduces the time needed to minimize the function. It is applicable to the assignment methods 1-4.

Forbid Inverse Matches

This option restricts the inverse column matching. Use it in a situation when the PRPG outputs have not their inverse value provided. It is applicable to the assignment methods 1-4.

Successfulness of the Method

Let us mention that all the assignment methods are driven by some heuristic that is often dependent on random events. Thus it may happen that the repeated execution of the program may yield in better results.

Moreover, some of the assignment methods – namely the column matching method exploiting don't cares and the Coverage-Directed Assignment method (4 and 5) sometimes need not be successful at all. In this case the failure is announced and no PLA file is produced.

Example

Let us have the two pattern files `C.pat` and `T.pat` describing the **C** and **T** matrices. To design an output decoder the assignment of the rows has to be performed first by a command:

```
ASSIGN C.pat T.pat assig.pla
```

The `assig.pla` file will contain the resulting function described by its on-set and off-set. Now it has to be minimized by BOOM:

```
BOOM assig.pla result.pla
```

Now the `result.pla` file contains the final PLA logic to implement the output decoder.

If the CD-A method is used, no minimization is required, as the ASSIGN program directly produces the minimized form (the implicants):

```
ASSIGN –T5 C.pat T.pat result.pla
```