Column-Matching Based Mixed-Mode Test Pattern Generator Design Technique for BIST

Petr Fišer, Hana Kubátová
Department of Computer Science and Engineering
Czech Technical University
Karlovo nam. 13, 121 35 Prague 2
e-mail: fiserp@fel.cvut.cz, kubatova@fel.cvut.cz

Abstract

A novel test-per-clock built-in self-test (BIST) equipment design method for combinational or full-scan sequential circuits is proposed in this paper. Particularly, the test pattern generator is being designed. The method is based on similar principles as are well known test pattern generator design methods, like bit-fixing and bit-flipping. The novelty comprises in proposing a brand new algorithm to synthesize the test pattern generator. In principle, we synthesize a combinational block - the Decoder, transforming pseudo-random code words into deterministic test patterns pre-computed by an ATPG tool. The Column-Matching algorithm to design the decoder is proposed. Here the maximum of output variables of the decoder is tried to be matched with the decoder inputs, yielding the outputs be implemented as mere wires, thus without any logic. No memory elements are needed to store the test patterns, which reduces the BIST area overhead.

Our BIST exploits mixed-mode testing principles. The BIST execution is divided into two disjoint phases – the pseudo-random phase and the deterministic phase. This enables to reach high fault coverage in a short test time and with a low area overhead. The choice of the lengths of the two phases directly influences the test time, BIST design time and BIST area overhead. A big effort has been put to a capability of trading-off the design criteria. The method allows for scaling the test time, BIST area overhead, BIST design time, etc. The time complexity of the algorithm is studied and experimentally evaluated.

Keywords. Built-in self-test, test pattern generator, mixed-mode testing, weighted random pattern testing, logic design.

1. Introduction

With the ever-increasing complexity of present VLSI circuits, their testing is becoming more and more important. There often arise faulty chips during the manufacturing process due to an inaccurate technology and such chips should be detected and eliminated. Using only external test equipment (ATE) to test the chips is becoming very time-consuming, mainly due to a huge amount of test vectors, long test time and very expensive test equipment. A solution of this problem consists in incorporating the Built-in Self-Test Equipment (BISTE) into the circuit. No external tester is required to test the circuit, since all the circuitry needed to conduct the test is included in the very circuit. This is paid by an area overhead, long test time and often low fault coverage.

The basic idea of Built-in Self-Test (BIST) is to design a circuit so that it is able to test itself and determine whether it is fault-free or faulty. This typically requires additional circuitry incorporated into the design. This additional logic must be capable to generate test patterns as well as to provide a mechanism to determine if the test responses of the circuit under test (CUT) correspond to those of a fault-free circuit. The basic architecture of the BIST circuitry as it might be incorporated into the CUT is shown in Fig. 1 [9].

Our aim is to design the Test Pattern Generator (TPG), so that it provides complete (100%) non-redundant stuck-at fault coverage and its size is maximally reduced.

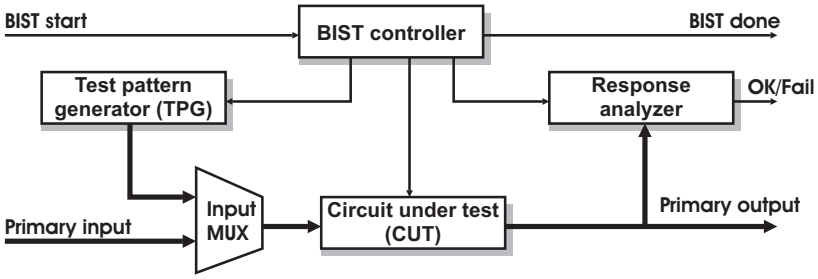

Fig. 1: Basic BIST architecture

Up to now, many BIST design methods have been developed [1-3, 9, 13, 18-23, 28], all of them trying to find some trade-off between these four aspects that cannot be all satisfied at one time:

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

To reach high fault coverage, either a long test time or a high area overhead is involved. A pseudo-random testing established the simplest trade-off between all these criteria. With an extremely low area overhead the circuit can be tested usually up to more than 90% in a relatively small number of clock cycles. To improve the fault coverage and to reduce the test time, many enhancements of this pseudo-random principle have been developed. All of them are accompanied by some additional area overhead.

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

We propose a flexible way how to design test pattern generators (TPGs) meeting *any* of the above-mentioned restrictions (or, better, quality measures). The designer is able to freely adjust the BIST logic design time, BIST logic area overhead and BIST run time, according his preferences. 100% fault coverage (of non-redundant faults) is considered in the following text. However, the method may be modified so that less fault coverage is reached, with a benefit of less area overhead and shorter design time.

The paper is structured as follows: the proposed test pattern generator design is described in Section 2. Section 3 describes the design of the Decoder, as an essential part of the proposed method. The possibility of scaling of the lengths of the BIST phases is discussed in Section 4, the way how to reduce the LFSR width is shown in Section 5. An improvement of the basic column-matching algorithm, yielding less BIST area overhead is briefly described in Section 6. Section 7 contains some experimental results, Section 8 concludes the paper.

2. Proposed Test Pattern Generator Design Method

The method is primarily intended for a test-per-clock BIST, thus the test patterns are applied to the primary inputs of the circuit-under-test (CUT) in parallel. However, the method can be modified for a test-per-scan as well [3]. Multiplexers separating combinational parts of sequential circuits have to be present, similarly like in full-scan designs.

The proposed test pattern generator (TPG) consists of two main parts: a linear feedback shift register (LFSR) producing pseudorandom patterns and the Decoder, which is a combinational block transforming these patterns into deterministic test patterns computed by an ATPG tool. Generating a fully deterministic test detecting 100% non-redundant stuck-at faults would involve a huge combinational logic (of the Decoder). Hence, a mixed-mode (or sometimes called hybrid) BIST [1, 2, 28] is used. The BIST run is divided into two phases: the pseudorandom and deterministic one. The difference between our mixed-mode BIST method and most of the others (e.g., bit-flipping [1], bit-fixing [2]) is that the two phases are separated in our approach. First, the easy-to-detect faults are covered in the *pseudo-random* phase run. Then, a set of deterministic test vectors covering the undetected faults is computed and these vectors are then generated by a transformation of the subsequent LFSR patterns. This significantly reduces both the Decoder and BIST control logic. No memory elements are needed to recognize patterns that are to be modified (like in [1, 2]); switching between the two phases is handled by the BIST controller counter. The proposed BIST scheme is shown in Fig. 2.

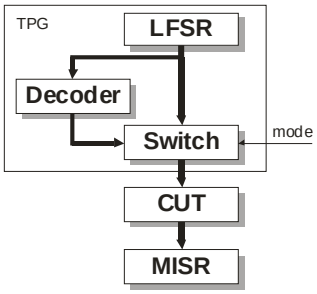

Fig. 2: Proposed BIST scheme

2.1. The LFSR Design

A *linear feedback shift register* (LFSR) is used as a pseudo-random pattern generator in our TPG. An *n*-bit (*n*-stage) LFSR is a linear sequential circuit consisting of D flip-flops and XOR gates generating code words (patterns) of a cyclic code. The structure of an *n*-stage LFSR-I (with internal XORs) is shown in Fig. 3.

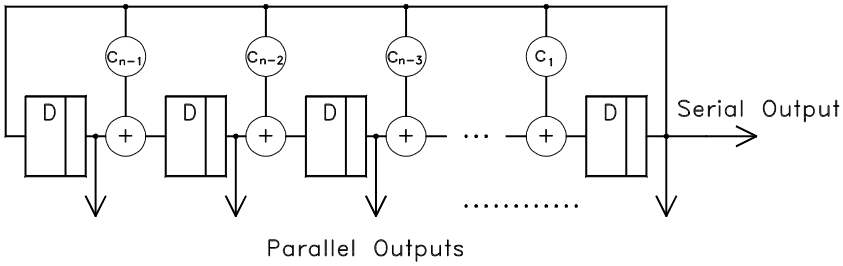

Fig. 3: LFSR structure

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

The coefficients $c_1, c_2, ...,c_{n-1}$ express whether there exists (1) a connection from the feedback to the corresponding XOR gate or there is no connection (0). Thus it determines whether there is a respective XOR gate present or the flip-flops are connected directly. The feedbacks leading to the XOR gates are also called *taps*.

The sequence of code words produced by an LFSR can be described by a *generating polynomial* g(x) in GF(2^n) [4].

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

If the generating polynomial is primitive (non-divisible by any other polynomial), the LFSR has a maximum period 2^n-1, thus it produces 2^n-1 different patterns. However, the use of a primitive polynomial is not needed for our purposes, since only few (i.e., thousands) patterns from all the possible 2^n-1 are needed to conduct the test. Thus, we use a randomly generated 1-tap LFSR (i.e., having only one XOR gate), to minimize the area overhead, see the study in [5]. For course, we have to assure that the LFSR period is long enough, e.g., by simulation.

3. The Decoder Design

The Decoder is designed by the column-matching algorithm proposed here. Let us have an *n*-bit LFSR running for *p* clock cycles. The code words generated by this LFSR are described by a **C** matrix (*code matrix*) of dimensions (*p, n*). These code words are to be transformed into deterministic test patterns computed by an ATPG tool. The deterministic patterns are described by a **T** matrix (*test matrix*). For an *r*-input CUT and the test consisting of *s* vectors the **T** matrix has dimensions (*s, r*). The rows of the matrices will be denoted as *vectors*. The Decoder logic modifies the **C** matrix vectors to obtain all the **T** matrix vectors. As the proposed method is restricted to combinational (or full-scan) circuits, the order of the test patterns is insignificant. Moreover, "gaps" in the transformation do not matter as well, they only represent non-testing vectors. Finding a transformation from the **C** matrix to the **T** matrix means finding a pairing of all the *s* rows of the **T** matrix with distinct rows of the **C** matrix – finding a *row assignment*, see Fig. 4. Here there are four **T** matrix rows assigned to five **C** matrix rows, the third **C** matrix row remains unassigned.

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

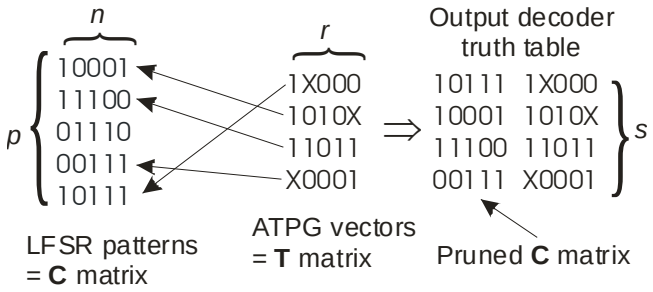

Fig. 4: Assignment of the rows

3.1. The Column-Matching Algorithm

The major task to solve is to assign the rows to each other in such a way, so that the Decoder logic will be as small as possible. The column-matching algorithm has been developed for this purpose. The principle of the algorithm is to assign all the **T** matrix rows to some of the **C** matrix rows so that some *columns* of the **T** matrix will be *equal* to some of the pruned **C** matrix columns in the result. This involves *no logic* needed to implement these **T** matrix columns (which correspond to respective output variables of the decoder); they are implemented as simple wired connections. This idea can be extended to a *negative matching*, by allowing negated columns to be matched. This would involve a negligible amount of additional logic as well.

The algorithm is thus based on finding columns that may be matched and subsequently assigning the rows to each other according to the selected column matches. The number of possible combinations of matches grows with a factorial of the number of the **C** matrix columns, thus finding an optimum solution is impossible. In practice, we use a simple heuristic method. Since the number of the **C** matrix rows is often much higher (typically by one or two orders of magnitude) than the number of the **T** matrix rows, finding several initial column matches is a trivial task: almost any two columns can be matched, because there is a big choice of possible assignments for the **C** matrix rows. Hence the selection of the rows to be matched is done at random.

Two random columns are chosen for matching and then we must determine whether the match can be really made. This is done by performing the row assignment (i.e., assigning distinct **C** matrix rows to all the **T** matrix rows). An efficient heuristic based on a *blocking matrix B* has been proposed in [10]. The blocking matrix is a binary matrix (it contains only "0" and "1" values) of dimensions (*p*, *s*). Thus, it has as many columns as there are **T** matrix rows and as many rows as there are **C** matrix rows. The value "1" in the cell **B**[*k*, *l*] indicates that the *k*-th **C** matrix row may be assigned to the *l*-th **T** matrix row, "0" value indicates the contrary.

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

$$\mathbf{B}[k, l] := \text{“0” when } (\mathbf{C}[k, i] \neq \mathbf{T}[l, j] \wedge \mathbf{T}[l, j] \neq \text{don't care}) \quad (2)$$

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

When performing the row assignment, distinct rows have to be assigned to each other. It is a trivial problem for a test without don't cares, since there does not exist a **B** matrix row having a "1" value in more than one column (one LFSR code word cannot be assigned to more than one test pattern). The final assignment then consists in selecting one row from the possible ones for each of the columns. Unfortunately, in the algorithm exploiting don't cares in the test, the **B** matrix rows may have ones in more than one column, since some values in the test patterns will be determined after the assignment. This makes the assignment NP-hard. Again, a simple but very efficient greedy heuristic is used to solve this problem [10].

The row assignment has to be performed after each trial for a column match, to determine whether the match is valid. If the assignment fails, the column-matching process is terminated and the last valid assignment is considered as the final

solution. The final valid row assignment forms a truth table, which has to be processed by a minimization tool [6-8] to synthesize the final Decoder logic.

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

Algorithm 1: Column-matching algorithm

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

This basic column-matching algorithm has been later extended to perform better, by introducing a one-step backtracking. If the match fails, the algorithm is not terminated but another match is looked for. The algorithm terminates once no more matches are possible to be done.

The asymptotic complexity of this extended algorithm (called a "*Thorough Search*") is $O(n \cdot r^2 \cdot p \cdot s^2)$. For more detailed description of the column-matching algorithm see [10, 11].

3.2. Column Matching Example

Let us assume our example from Fig. 4. We have four **T** matrix vectors (labeled 'A'-'D' now) that are to be assigned to some of the five **C** matrix vectors (labeled 'a'-'e'). The LFSR outputs (i.e., decoder inputs) will be labeled $x_0 - x_4$, the decoder outputs $y_0 - y_4$ (here the columns of **C** and **T** matrices). Since there are no matches selected at the beginning, any **T** matrix vector may be assigned to any **C** matrix vector, the **B** matrix is filled with 1-values, see Fig. 5a. Now we decide (randomly) to try to assign y_0 to x_0. Then the **B** matrix will be pruned to some extent, see Fig. 5b. For example, the **T** matrix row 'A' may be assigned to **C** matrix rows having a '1' value in the x_0 column only, since it has a '1' value in the y_0 column, i.e., {a, b, e}. Then we select the $x_1 - y_1$ match (Fig. 5c). More intensive **B** matrix pruning can be observed here. The 'C' row can be matched with the 'b' row only from now on. Next we proceed with negatively matching $x_2 - y_2$ (Fig. 5d) and positive $x_1 - y_3$ (Fig. 5e). There is no possibility for matching y_4 after performing these four matches, thus the algorithm ends up with four matches. There has been left two choices for matching the 'D' row ('d' and 'e'), 'd' is chosen randomly. The test don't care values are then substituted by exact values, so that the respective **C** and **T** matrix columns will contain equal values. The final row assignment is shown in Fig. 5f.

The first four Decoder outputs ($y_0 - y_3$) have been successfully matched, thus these outputs will be implemented without any logic. The output y_4 has to by synthesized, e.g., by using a Boolean minimizer. The resulting Decoder logic is shown in Fig. 6. We can see that only one logic gate is needed to perform the matrices transformation.

Let us note that the "random" choices for the column matches in this example have been made intentionally, to obtain the best possible result and in a direct way, without invalid trials. Any other combination of matches would yield worse results. However, the Thorough Search algorithm usually discovers a solution shown in this example soon.

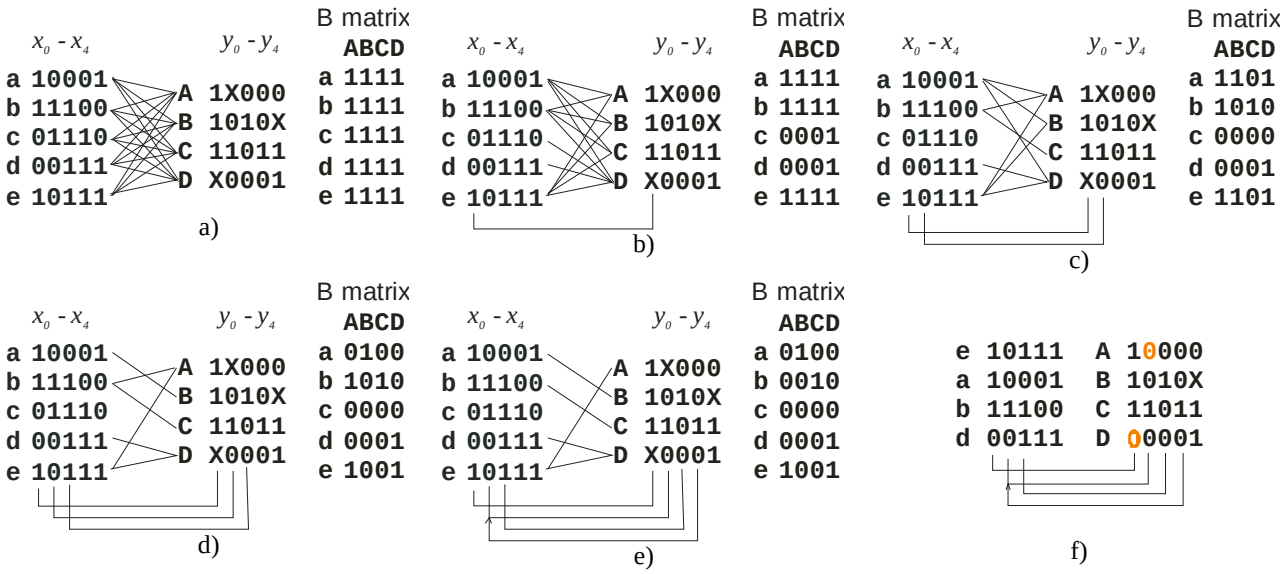

Fig. 5: Column Matching example

$$y_0 = x_0$$
$$y_1 = x_1$$
$$y_2 = x_2'$$
$$y_3 = x_1$$
$$y_4 = x_0' + x_1$$

Fig. 6: The Decoder logic

3.3. Mixed-Mode Column-Matching BIST

When the mixed-mode BIST technique is used, the test is divided into two phases – the pseudorandom and deterministic one. In the pseudorandom phase the LFSR code words are applied to the CUT unmodified, to detect the easy-to-detect faults. Faults that remained undetected are found by fault simulation and deterministic test vectors detecting them are computed by an ATPG. These deterministic vectors are then generated by a transformation of the LFSR code words that follow the pseudorandom sequence.

An artificial illustrative example is shown in Fig. 7. The BIST logic for a 5-input circuit is to be synthesized here, the LFSR is required to run for 10 cycles in total. A 5-bit LFSR is run for 5 cycles first (the “Pseudo-random sequence”), to detect the easily testable faults. Deterministic test vectors for the faults not detected by these five vectors are generated by an ATPG. At the end the decoder is synthesized as a combinational block transforming the subsequent five LFSR patterns into these four deterministic vectors. The resulting test sequence detecting 100% of faults is shown on the right side of the figure, the resulting TPG circuitry is shown in Fig. 8. First, the “Deterministic mode” signal is set to ‘0’, so that the LFSR sequence is applied to the CUT unmodified. After 5 clock cycles it is switched to ‘1’, to apply the deterministic patterns.

The Decoder logic synthesis process is described in the previous example.

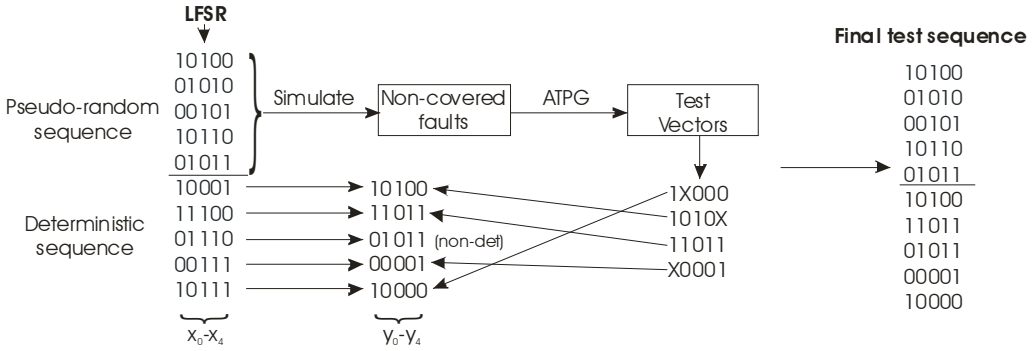

Fig. 7: Test sequence generation

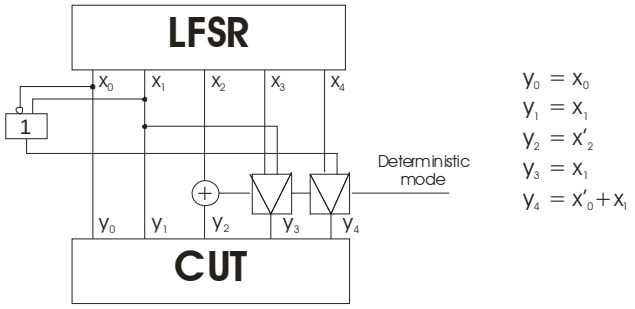

Fig. 8: Resulting BIST circuitry

4. Scaling the Lengths of the Phases

As it was said above, the BIST equipment design methodology should cope with the designer's needs, thus be as customizable as possible. The above mentioned four aspects (area overhead, fault coverage, test time, design time) play a big role in the overall design process and cannot be optimally satisfied all.

The column-matching based TPG design method is very well scalable in all these aspects. The ways of scaling are described in this section. The fault coverage aspect will not be discussed here, since 100% of detected non-redundant faults are assumed. Anyway, the fault coverage is determined by the ATPG used. If less than complete fault coverage is sufficient, less deterministic vectors may be used. Then both the TPG area overhead and BIST design time would be reduced.

Parameters that most essentially influence the TPG area overhead and BIST design time are the lengths of the two BIST phases. Of course, the duration of the BIST execution is given by the lengths of the phases directly.

4.1. Pseudo-Random Phase

The aim of the pseudo-random phase is to detect as many faults as possible, while keeping the test time acceptable. Two aspects play role here: the LFSR polynomial and seed and the test length. Several methods to compute the LFSR seed to achieve good fault coverage have been proposed [12, 13]. However, for simplicity, we just repeatedly select the seed randomly, evaluate the fault coverage reached by using it, and select the best one. This approach allows reaching good fault coverage as well, whereas the fault coverage may be almost arbitrarily improved, for a cost of the runtime (by increasing the number of repetitions). Moreover, it is generally applicable to any type of pseudo-random pattern generators, e.g., cellular automata [14].

The number of the covered faults as a function of the number of LFSR cycles applied to the CUT follows the saturation curve shown in Fig. 9 (for the c3540 circuit [15]). First few vectors detect the majority of faults and then the fault coverage increases only slightly.

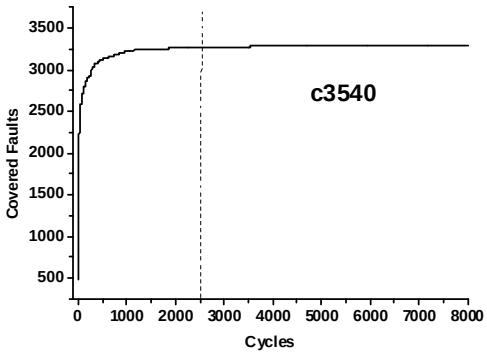

Fig. 9: Fault coverage saturation curve

In order to reach satisfactory fault coverage in the pseudo-random phase, the fault coverage saturation curve for the CUT should be determined by a fault simulation. The appropriate length of the *PR* phase can be easily derived from it. The pseudo-random phase should be stopped when the fault coverage does not improve for a given number of cycles. This number can be freely adjusted, according to the application specific requirements (the trade-off between the test time and area overhead).

To illustrate the scalability of the method in terms of the length of the pseudo-random phase, the BIST structure was designed for several ISCAS benchmarks [15, 16]. The length of the pseudo-random phase was varied, while the length of the deterministic phase was kept constant, 1000 cycles.

The results are shown in Tab. 1. The benchmark name is shown in the first column. The *"PR"* column indicates the length of the pseudo-random phase, the *"UD"* column shows the number of stuck-at faults that were left undetected by this phase. *"vct."* gives the number of deterministic vectors produced by an ATPG [24], to test these faults. The length of the deterministic phase was set constantly to 1000 cycles, except of the s38417 benchmark, where it was set to 2000 cycles (because of the size of the test set). The *"GEs"* column shows the total complexity of the BIST design, in terms of the gate equivalents [17]. The time needed to complete the column-matching procedure is indicated in the last column. The experiment was run on a PC with 1 GHz Athlon CPU, Windows XP.

Tab. 1: Influence of the length of the pseudo-random phase

bench	*PR*	*UD*	*vct.*	*GEs*	*Time [s]*
s1196	200	228	104	110.5	5.05
	2 K	52	37	37	1.20
	10 K	9	4	6	0.04
s5378	5 K	89	49	65.5	2259
	10 K	63	23	31.5	767
	20 K	48	8	16.5	104
s9234.1	1 K	1674	215	883	52 300
	50 K	773	99	333.5	4 400
	200 K	599	52	212.5	1 600
s13207.1	1 K	1793	197	699	208 K
	10 K	617	74	280	3 480
	50 K	182	21	36	128
s38417	10 K	2067	1391	15106.5	3 417
	100 K	780	520	3259.5	2 263

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

4.2. Deterministic Phase

In the deterministic phase deterministic vectors are synthesized from some of the LFSR patterns that follow the pseudo-random phase. With increasing the number of LFSR patterns, the chance for finding more column matches

increases as well. This is due to having more freedom in selecting the LFSR vectors to be assigned to the deterministic vectors. However, the algorithm runtime rapidly increases with the number of deterministic vectors.

This is illustrated by Tab. 2. Its format is partially retained from Tab. 1, the "*Det.*" column indicates the length of the deterministic phase.

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

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

Tab. 2: Influence of the the length of the deterministic phase

bench	*inps*	*PR*	*Det.*	*GEs*	*Time [s]*
c3540	50	1000	200	34	0.32
			500	29.5	0.52
			1000	28	1.02
			2000	16.5	1.47
s1196	32	5000	200	25.5	0.17
			500	25	0.32
			1000	17	0.48
			2000	17	1.52
			5000	9	2.16
			10000	4.5	5.83

4.3. Summary

By increasing the length of the pseudorandom phase, the number of undetected faults decreases. Thus, the number of deterministic test vectors that are needed to be generated in the deterministic phase is decreases as well. As a consequence of this, the BIST design time is often significantly decreased (even though the fault simulation time is higher) and the area overhead is reduced as well. To obtain best results, the pseudorandom phase should be run until most of the easy-to-detect faults are detected, according Fig. 9. Running this phase longer becomes ineffective.

By prolonging the run of the deterministic phase the column-matching algorithm runtime is increased. The algorithm has more freedom in the selection of matches on the other hand, so the area overhead of the decoder is decreased.

Tab. 3: Influence of the test lengths

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

5. Reducing the LFSR Width

As it was stated before, the column-matching algorithm is primarily designed for a test-per-clock BIST. Originally, the number of the LFSR bits had to be equal to the number of CUT inputs. However, the LFSR width may be arbitrarily scaled by using the weighted pattern testing. The effect of such a scaling will be shown in this section. The LFSR scaling affects both the total TPG area overhead and the TPG design time.

5.1. Weighted Pattern Testing

There have been many weighted pattern testing approaches proposed up to now [18-23]. Basically, all of them are based on computing the probability of occurrence of '1' and '0' values (weights) on particular inputs and modifying the pseudorandom sequence (which usually has this probability 0.5 at all the inputs) by some additional *weighting* logic, to meet these weights. In general, the weighting logic consists of AND and OR gates, which, when fed by LFSR outputs, produce weighted patterns. For example, when two LFSR outputs are led into an AND gate, the resulting probability of occurrence of '1' at the output of the AND gate will be 0.25.

It was shown that using only simple weighted pattern testing does not ensure sufficient fault coverage and multiple weight sets are needed [19]. Another alternative is to use a combination of weighted pattern testing with deterministic test,

as shown, e.g., in [23]. We propose such an approach too, particularly the combination of weighted pattern pseudorandom testing with column-matching.

The weights are usually being computed from the deterministic test set derived for the tested circuit. A common approach is to find a set of so called *random pattern resistant faults (RPRFs)*, which are faults that are difficult to detect by random patterns. Then, a test vector set is computed to test these faults. The weights are then derived by computing respective 0/1 value ratios for each CUT input.

The RPRFs are determined by repeatedly applying pseudorandom vectors to the CUT and recording the undetected faults. The number of RPRFs obtained thus strongly depends on the number of pseudorandom patterns applied. The higher their number is, the less faults remain undetected. There arises a question of what number of RPRFs should be considered to compute the weights, to obtain optimum results. One limit approach is to consider *all* faults and to derive the weights from a complete test for the circuit. This approach is often unusable for very large circuits, since the complete test set computation would take a very long time. We have performed experiments to estimate what number of RPRFs should be used to compute the test weights. We have used the s9234.1 ISCAS'89 benchmark circuit [16] for the following measurement. We have varied the number of pseudorandom vectors applied to the CUT to detect RPRFs, from 0 (all faults are used to compute weights) to 100 000. Then a 3-weight set was computed using the test vectors detecting these RPRFs. After that, the weighting logic was synthesized and the weighted test pattern generator was run for 10 000 cycles. The number of undetected faults was measured. Such an experiment was repeatedly run 10 000-times (using different LFSRs and keeping the computed weight set) and the average of the result was computed (for the number of undetected faults), for higher precision of the measurement. The results are shown in Tab. 4. First the number of pseudorandom vectors used to determine RPRFs is shown, then the number of RPRFs obtained. The "*vcts*" column indicates the number of test vectors used for the weight computation. Finally, the average number of faults undetected by the run of the resulting weighted pattern generator is shown ("*UD*"). The dependency of the number of undetected faults on the number of test vectors testing RPRFs is visualized by Fig. 10. There is an apparent global minimum to be seen, corresponding to the optimum number of test vectors to determine the test weights. However, it is hard to estimate this optimum in practice, for different circuits. It has to be found experimentally, by trying out different numbers of vectors that are applied to the circuit and picking the best trial.

Tab. 4: Computing test weights

vectors	*RPRFs*	*vcts.*	*UD*
0	6927	6470	852
100	3060	2605	817
1000	1997	1544	815
2000	1709	1257	805
2500	1614	1162	802
3000	1482	1030	781
4000	1347	895	762
5000	1280	828	808
10 000	1080	628	818
20 000	950	498	860
50 000	876	424	862
100 000	715	263	891

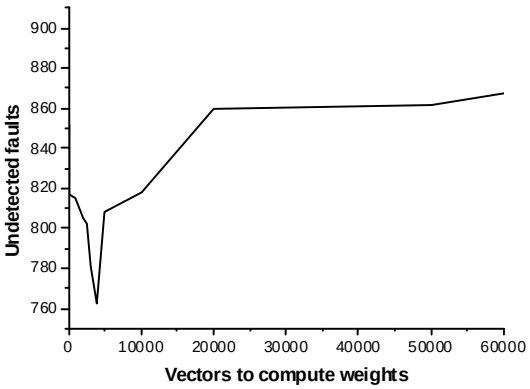

Fig. 10: Numbers of undetected faults

Another issue that has to be taken into account is the design of the weighting logic. The higher is the precision of weight generation, the higher the area overhead of the weighting logic. For our examples we will limit ourselves to the 3-weight and 5-weight logic only, since more weights (or even more weight sets) would involve a large area overhead caused by the weighting logic, while the gain in quality (fault coverage) is negligible.

There have been several weighting logic designs proposed. The simplest one (see, e.g., [18]) proposes weights like 0, 0.5, 1 for 3-weighted logic, 0, 0.25, 0.5, 0.75, 1 for 5-weight logic, etc. This approach has been found very inefficient in terms of the fault coverage reached, see Tab. 5. This is most probably due to constant 0 and 1 values in the test. However, such a 3-weight test does not require any additional weighting hardware, since the 0 and 1 weights are implemented as hard-wired connections to the ground or the voltage supply, respectively, and the 0.5 weight is constructed as a direct connection to an LFSR output. We use a 3-weight testing method where the weights are 0.25, 0.5 and 0.75. Thus, the two weights are constructed by AND-ing and OR-ing two LFSR outputs and, for a 5-weight method (0.125, 0.25, 0.5, 0.75, 0.875), the weights are generated by AND-ing and OR-ing two or three LFSR outputs. This approach involves some additional hardware, however the increase of the fault coverage reached by it is fully compensated by the reduction of the deterministic test generator logic.

The effect of the use of the weighting logic on the fault coverage is illustrated by Fig. 11. We have computed the weights for the ISCAS'89 s13207.1 circuit [16], repeatedly reseeded the LFSR and constructed the weighting logic (10 000-times). The number of faults that remained undetected by a sequence of 5000 vectors generated by the final TPG was measured. The curves represent the frequencies of the respective amounts of undetected faults (the area below the curves is equal to the number of vectors applied, i.e., 10 000). Four curves are shown, representing four different weighting logics. The most efficient one is the 5-weight logic, where no 0 and 1 weights are used (the leftmost curve). Here minimum of faults are left undetected. Similar 3-weight logic is illustrated by the neighboring curve. The number of undetected faults is almost the same. Then the case with no weighting logic used is shown, for comparison. It can be seen that the weighting logic significantly reduces the number of undetected faults, with respect to this case. The rightmost curve describes the 5-weight logic case, where constant 0 and 1 weights are used. Here the number of undetected faults is even increased. The 3-weight logic with 0 and 1 constants is not shown in the graph, since the number of undetected faults is extremely high (far to the right).

The results are summarized in Tab. 5. Minimum and average numbers of undetected faults are shown there, for all the five weighting logic cases. The s13207.1 circuit has 9815 faults in total.

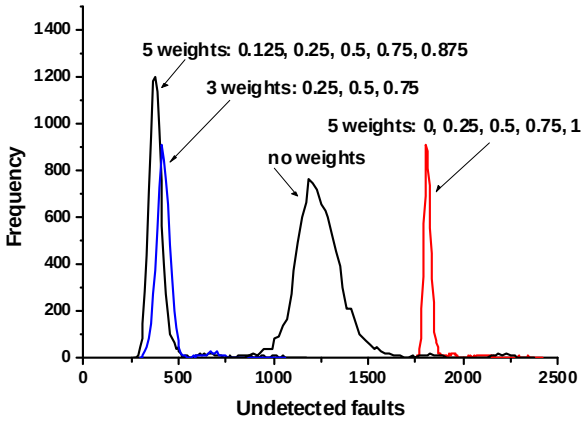

Fig. 11: Weighting logic effects

Tab. 5: Weighting logic effects

Weighting logic	Min.	Avg.
5-weights (0.125, 0.25, 0.5, 0.75, 0.875)	269	392
3-weights (0.25, 0.5, 0.75)	299	425
no weights	742	1250
5-weights (0, 0.25, 0.5, 0.75, 1)	1761	1823
3-weights (0, 0.5, 1)	3762	3771

5.2. Weighted Pattern Column-Matching

When the weighted pattern testing is used in connection with the column-matching method, a new block is introduced into the TPG – the *weighting logic* block. See Fig. 12. The LFSR width (*r*) may be then less than the number of CUT inputs (*m*), the number of TPG outputs is increased just by the weighting logic block.

In practice, an LFSR with a random polynomial and random seed is used. This ensures (to some extent) a uniform distribution of '1's and '0's in code words produced by it. In other words, the weights of all the *r* LFSR outputs are approximately 0.5. To generate the weighted patterns, the outputs of the weighting logic are generated by AND-ing or OR-ing randomly selected LFSR outputs. Thus, *m* weighted TPG outputs are generated by this way.

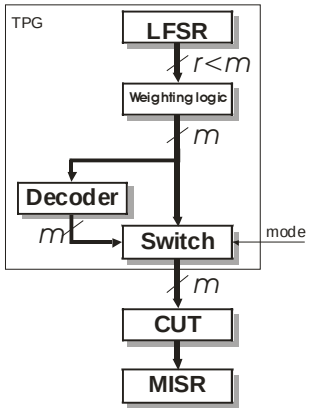

Fig. 12: Weighted column-matching BIST scheme

The effects of the LFSR scaling are shown in Tab. 6 for the ISCAS s13207.1 benchmark having 700 inputs and compared with a standard approach (the case when no weights are used). The "*LFSR*" column indicates the width of the LFSR used (*r).* In a case of weighted pattern testing the number of gates needed for the weighting logic is shown in the next column ("*w*"). The total TPG area overhead (including the LFSR) is computed in terms of gate equivalents [17]. An *n*-input NAND gate counts for 0.5*n* GEs, the two-input XOR gate (used in LFSR) is 2.5 GEs. The size of a D flip-flop is considered to be 4 GEs, which complies with the design of a standard flip-flop in CMOS logic. We have tried to scale the originally 700-bit LFSR down to 30 bits. It can be seen that optimum results are obtained for the 50-bit LFSR, the TPG area reduction is more than 70% with respect to the original (700-bit unweighted) case. When the LFSR width is scaled down more, the TPG area overhead rapidly increases, since the weighted LFSR is not able to cover enough faults, due to reduced randomness of the patterns. The column-matching results for the 30-bit LFSR are not present, due to very high column-matching algorithm runtimes.

See [25] for more details of the principles of weighted pattern column-matching testing.

Tab. 6: LFSR scaling

LFSR (r)	*method*	*w*	*time [s]*	*TPG GEs*
700	no weights	-	4720	2975
700	3-w	518	245	3361
200	3-w	365	653	1231
50	3-w	365	2835	671
45	3-w	365	3423	693
40	3-w	365	29434	1288
30	3-w	365	-	-

6. Multiple-Vector Column-Matching

The more "freedom" has the column-matching algorithm in selection of the matches, the better it performs. Particularly, more don't care values in the test set induce more column matches and thus less area overhead [10]. Some ATPG tools are able to produce rather complex (and redundant) tests, which can be efficiently exploited by the column-matching algorithm. E.g., the Atalanta ATPG [24] is capable producing more than one (or all) vectors testing one particular fault. The test set is then much larger, yielding the column-matching process be slower. However, due to more freedom for a column match selection, the area of the Decoder is less.

This is documented by Tab. 7. The "*vct/flt*" column indicates the number of processed test vectors per one fault. The total number of generated test vectors (T-matrix size) is shown in the next column. The TPG design time and its area

overhead (w.r.t. the original circuit) are shown next. The improvement with respect to the original, one-vector method is indicated in the last column. We can conclude that quite a significant area reduction may be obtained when multiple vectors per one fault are used, for a cost of a longer runtime.

See [26] for more details on multiple-vector column-matching.

Tab. 7: Multiple-vector column-matching

bench	*vct/flt*	*vcts.*	*time [s]*	*overhead*	*impr.*
c1908	1	36	6.7	5.7 %	
	10	340	55.9	3.0 %	48 %
c3540	1	31	3.9	2.2 %	
	10	101	19.1	1.6 %	27 %
	100	555	90.0	1.3 %	42 %
c7552	1	106	1104.8	17.0 %	
	10	1206	16124.7	14.8 %	13 %
s1196	1	55	5.5	11.1 %	
	10	259	109.0	7.8 %	30 %
s1238	1	33	2.9	6.7 %	
	100	95	16.7	4.6 %	31 %
s5378	1	19	7.7	1.5 %	
	100	258	181.5	0.9 %	40 %
s9241.1	1	52	160.7	5.3 %	
	10	564	3508.6	4.9 %	10 %

7. Experimental Results

7.1. Comparison with Other State-of-the-Art Methods

The proposed column-catching method is compared with three state-of-the-art methods in this section, namely the bit-fixing accompanied by a "bit-correlating" ATPG [2], the "3-weight weighted random pattern BIST" proposed in [19] and the row matching method proposed in [3]. The comparison results are shown in Tab. 8. The *"TL"* columns indicate the total length of the test, the *"GEs"* columns give the number of gate equivalents (or 2-input NAND gates) of the BIST combinational circuits and the *"lit."* columns indicate the number of literals in the sum-of-products (SOP) form of the decoding logic. Test lengths have been set approximately equal.

Let us mention here, that a special kind of a test pattern generator (GLFSR) is used in the row matching approach [3]. Using such a circuit causes quite a large area overhead in most cases, for many XOR gates involved. This overhead is not included in the table. Generally, the column-matching method is independent on a pseudorandom pattern generator employed, thus in all the cases an LFSR with one XOR gate only was used. Thus, sometimes bigger area overhead of our method could be compensated by a small area of the pseudorandom pattern generator used.

In the bit-fixing and weighted BIST methods several additional registers (flip-flops) are present in the test pattern generator. In the column-matching method no flip-flops are needed (except of those in the LFSR).

The column-matching results describe the overall test length and the number of gate equivalents of the Decoder. The number of GEs approximately corresponds to the number of SOP literals, thus a comparison with the bit-fixing and weighted BIST methods can be freely made.

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

Tab. 8: Comparison results

	Bit-fixing [2]		Weighted BIST [19]		Row matching [3]		Column matching	
Bench	*TL*	*lit.*	*TL*	*lit.*	*TL*	*GEs*	*TL*	*GEs*
c880	-	-	-	-	640	21	1 K	15
c1355	-	-	-	-	1.8 K	0	1.5 K	15
c1908	-	-	-	-	4.7 K	8	3 K	10.5
c2670	10 K	385	8 K	269	6 K	119	5 K	113
c3540	-	-	-	-	4.8 K	4	5.5 K	1.5
s420	10 K	59	1.4 K	67	-	-	1 K	24.5
s641	10 K	98	768	45	7.7 K	6	4 K	15
s713	-	-	-	-	4.8 K	4	5 K	16.5
s838	10 K	183	3.1 K	108	-	-	6 K	130
s1196	10 K	97	16.8 K	67	10 K	36	10 K	6
s1238	-	-	17 K	33	-	-	4 K	26.5
s5378	10 K	332	18.4 K	68	-	-	11 K	19.0

7.2. Results for Standard Benchmarks

Since the comparison shown in Tab. 8 describes results for a few small benchmark circuits only, we will present a more exhaustive result table (Tab. 9), for some of the bigger ISCAS [15, 16] and ITC'99 [27] benchmarks. The easy-to-test benchmarks were omitted (benchmarks 100% testable by less than 100 000 random vectors). The BIST circuitry was synthesized in two modes for each benchmark – in the first mode, the test length was set to be relatively small (the white rows). In the second mode a big effort has been put to obtain low area overhead in a reasonable BIST design time. The test is prolonged and some improvement techniques, like multiple-vector column-matching and weighted pattern testing, are sometimes used to design the TPG. This is indicated in the "*Method"* column. The legend to the values is below the table.

The *"Inps"* column indicates the number of the benchmark inputs, in the *"100% FC*" column the number of pseudorandom vectors needed to be applied to the CUT to achieve 100% fault coverage is shown, just to show the effectiveness of the method. The *"TL"* column gives the lengths of the pseudo-random and deterministic phases. Thousands of vectors are abbreviated by "K", millions by "M". The "*Match*" columns show the total number of column matches reached. The complexity of the switching logic is shown in the *"SW GEs*" column, the complexity of the output decoder is shown in *"OD GEs"*. These numbers are summed together (and with possible number of weighting gates) in the *"Total GEs*" column. The percentage of the area overhead of the Output Decoder and Switch, with respect to the CUT GEs is shown in the "*BIST Overhead*" column. The runtime needed to complete the column-matching process is indicated in the last column.

It can be seen that quite a large number of column matches is often reached and thus the Output Decoder logic is reduced to minimum, thus the overall BIST overhead is reduced to percents of the size of the original circuit.

The experiments were run on a PC with 2,6 GHz Athlon AMD CPU, Windows XP operating system.

Tab. 9: Results for standard benchmarks

Bench	*Inps*	*100% FC*	*TL (PR + Det.)*	*Method*	*Match*	*SW GEs*	*OD GEs*	*Total GEs*	*BIST Overhead*	*Time [s]*
c2670	233	2.4 M	1 K + 1 K		193	90	109.5	199.5	19 %	166
			4 K + 1 K	1)	201	82.5	71.5	154	15 %	437
c7552	207	> 100 M	7 K + 1 K		131	261	325	586	19 %	500
			10 K + 1 K		133	256.5	248.5	505	16 %	887
s420	34	165 K	400 + 600		32	21	3.5	24.5	13 %	0.75
			5 K + 1 K	1)	34	18	**0**	18	9 %	5.01
s641	54	200 K	500 + 500		52	21	2	23	9 %	0.47
			3 K + 1 K		**54**	15	**0**	15	6 %	0.21
s713	54	300 K	500 + 500		52	24	3	27	8 %	0.56
			3 K + 1 K		**54**	18	**0**	18	5 %	0.32
s838	67	> 100 M	1 K + 1 K		37	81	45	126	32 %	26.20
			10 K + 2 K		46	79.5	29	108.5	28 %	51.51
s1196	32	200 K	2 K + 1 K		28	13.5	23.5	37	7 %	1.20
			9 K + 1 K		**32**	6	**0**	6	1 %	0.04
s9234	247	10 M	50 K + 1 K		208	163.5	156	319.5	8 %	350
			200 K + 1 K	1)	225	127.5	66	193.5	5 %	3500
s13207.1	700	100 K	1 K + 1 K		638	456	294	750	13 %	4000
			50 K + 1 K		**700**	36	**0**	36	< 1 %	13
s15850.1	611	> 10 M	10 K + 1 K		478	397.5	187	584.5	9 %	812
			100 K + 2 K		553	306	66.5	372.5	5 %	1244
s38417	1664	> 10 M	10 K + 1 K		1240	1365	1389.5	2754.5	17 %	24 K
			100 K + 1 K	2)	1000	1239	142	2284	14 %	4660
s38584.1	1464	> 1 G	10 K + 1 K		1435	379.5	57.5	437	3 %	650
			100 K + 1 K		**1464**	165	**0**	165	1 %	34
b07	50	200 K	1 K + 1 K		45	33	8.5	42.5	11 %	4
			10 K + 1 K		**50**	24	**0**	24	6 %	0.5
b12	126	5 M	1 K + 1 K		117	37.5	45	82.5	9 %	40
			10 K + 1 K	1)	118	33	34	67	7 %	1080
b14	277	> 100 M	1 M / 2 K		84	318	8017	8335	141 %	170 K
			100 M / 1 K		90	328.5	2663.5	3319.5	56 %	100 K
b15	485	> 100 M	100 K / 1 K		241	558	4709	5267	67 %	50 K
			1 M / 1 K	2)	378	373.5	667	1355.5	17 %	4800

1) 10 vectors per fault
2) 3-weight logic

8. Conclusions

A scalable and customizable mixed-mode BIST equipment design method based on a *column-matching* principle has been proposed. Here pseudorandom LFSR code words are being transformed into deterministic test patterns computed by an ATPG tool. The transformation is being done by a purely combinational block; no additional registers are needed to perform the transformation. The algorithm tries to “match” maximum of decoder outputs with its inputs, which yields no logic needed to implement these outputs.

The pseudo-random and deterministic phases are separated, which enables to reach less area overhead of the control logic. The method is based on a design of a decoder transforming the LFSR code words into deterministic test vectors testing the hard-to-detect faults. Moreover, no memory elements are needed to recognize LFSR patterns that are to be modified, like in other state-of-the-art approaches (e.g. bit-fixing, bit-flipping).

Since the test is divided into two phases, the lengths of both phases may be freely adjusted, to find a trade-off between the test time and area overhead. It has been shown that the length of the pseudo-random phase has a crucial impact to the result. The length of the deterministic phase influences the result as well, though not that significantly. The impact of the test lengths to the duration of the BIST design process is considered too.

A big scalability of the method, in terms of the area overhead, test time and design time is shown. A weighted pattern testing principle is used to reduce the LFSR width. Next, multiple-vector column-matching method reducing the area overhead for a cost of a longer runtime is proposed.

The proposed algorithm should serve as a basic guideline how to design more complex BIST designs, i.e., the multiple-scan chain based BIST, the STUMPS architecture, etc. The method should be as general, as the other state-of-the-art methods are (e.g., bit-flipping, bit-fixing). The obtained results, in terms of the area overhead, are comparable to the other methods, sometimes they are significantly better.

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

The proposed method could be very suitable for complex, e.g, SoC designs – here only the Decoder logic has to be synthesized for each combinational block (core); the LFSR and the BIST control logic may be shared among the blocks. The investigation of such possibilities would be a part of our future research.

Acknowledgement

This research has been supported by MSMT under research program MSM6840770014.

References

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

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

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

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

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

[6] R.K. Brayton, et al, *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA, Kluwer Academic Publishers, 1984, 193 p.

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

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

[9] E.C. Stroud, A Designer's Guide to Built-In Self-Test, Kluwer Academic Publishers, Boston, MA, 2002, 344 p.

[10] P. Fišer, H. Kubátová, J. Hlavička, Column-Matching BIST Exploiting Test Don't-Cares, Proc. 8th IEEE European Test Workshop, Maastricht, 2003, pp. 215-216

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

[12] C. Fagot, O. Gascuel, P. Girard, C. Landrault, On calculating efficient LFSR seeds for built-in self test, Proc. IEEE European Test Workshop 1999 (ETW'99), Constance, Germany, 1999, pp. 7-14

[13] S. Hellebrand, H.-J. Wunderlich, A. Hertwig, Mixed-Mode BIST Using Embedded Processors; Journal of Electronic Testing Theory and Applications (JETTA), Vol. 12, Nos. 1/2, February/April 1998, pp. 127-138

[14] P. Fišer, H. Kubátová, Improvement of the Fault Coverage of the Pseudo-Random Phase in Column Matching BIST, Proc. 31th Euromicro Symposium on Digital Systems Design (DSD'05), Porto, (Portugal), 2005, pp. 56-63

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

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

[17] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994, 576 p.

[18] H.J. Wunderlich, Self Test Using Unequiprobable Random Patterns, Proc. International Symposium on Fault-Tolerant Computing, 1987, pp. 258-263

[19] S. Wang, Low Hardware Overhead Scan Based 3-Weight Weighted Random BIST, Proc. IEEE International Test Conference, Washington, DC, 2001, p. 868.

[20] F. Muradali, V.K. Agarwal, B. Nadeau-Dostie, A New Procedure for Weighted Random Built-In-Self-Test, Proc. International Test Conference (ITC'90), 1990, pp. 660-668

[21] M.A. Miranda, et al., Generation of Optimized Single Distributions of Weights for Random BIST, Proc. International Test Conf. (ITC), 1993, pp. 1023- 1030

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

[23] A. Jas, C.V. Krishna, N.A. Touba, Hybrid BIST Based on Weighted Pseudo-Random Testing: A New Test Resource Partitioning Scheme, Proc. 19th IEEE VLSI Test Symposium (VTS). Washington, DC, 2001, pp. 114-120
[24] H.K. Lee and D.S. Ha. Atalanta: an Efficient ATPG for Combinational Circuits. Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993
[25] P. Fišer, H. Kubátová: Pseudo-Random Pattern Generator Design for Column Matching BIST, Proc. 10th Euromicro Conference on Digital Systems Design (DSD'07), Lübeck (Germany), 27. - 31.8.2007, pp. 657-663
[26] P. Fišer, H. Kubátová: Multiple-Vector Column-Matching BIST Design Method, Proc. 9th IEEE Design and Diagnostics of Electronic Circuits and Systems 2006 (DDECS'06), Prague, CZ, 18.-21.4.2006, pp. 268-273
[27] F. Corno, M. Sonza Reorda and G. Squillero, "*RT-Level ITC 99 Benchmarks and First ATPG Results*". IEEE Design & Test of Computers, July-August 2000, pp. 44-53
[28] G. Jervan, P. Eles, Z. Peng, R. Ubar, M. Jenihhin: Hybrid BIST Time Minimization for Core-Based Systems with STUMPS Architecture. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 225-232