EFFICIENT PATTERN MAPPING FOR DETERMINISTIC LOGIC BIST

Valentin Gherman, Hans-Joachim Wunderlich
Harald Vranken
Friedrich Hapke, Michael Wittke, Michael Garbers

Universität Stuttgart
Pfaffenwaldring 47
D-70569 Stuttgart
Germany

Philips Research
Prof. Holstlaan 4-WAY-41
5656 AA Eindhoven
The Netherlands

Philips Semiconductors
Georg-Heyken-Strasse 1
D-21147 Hamburg
Germany

ghermanv@informatik.uni-stuttgart.de
wu@informatik.uni-stuttgart.de
harald.vranken@philips.com
friedrich.hapke@philips.com
michael.wittke@philips.com
michael.garbers@philips.com

Abstract

Deterministic logic BIST (DLBIST) is an attractive test strategy, since it combines advantages of deterministic external testing and pseudo-random LBIST. Unfortunately, previously published DLBIST methods are unsuited for large ICs, since computing time and memory consumption of the DLBIST synthesis algorithms increase exponentially, or at least cubically, with the circuit size.

In this paper, we propose a novel DLBIST synthesis procedure that has nearly linear complexity in terms of both computing time and memory consumption. The new algorithms are based on binary decision diagrams (BDDs). We demonstrate the efficiency of the new algorithms for industrial designs up to 2M gates.

Keywords: Logic BIST, BDDs

1. Introduction

Logic Built-In Self-Test (LBIST) for random logic is becoming an attractive alternative in IC testing. Recent advances in nanometer IC process technology and core-based IC design are leading to more widespread use of LBIST since external testing is becoming more and more difficult and costly. Also requirements on in-field testing and limited access into ICs that contain secure information, are demanding LBIST solutions.

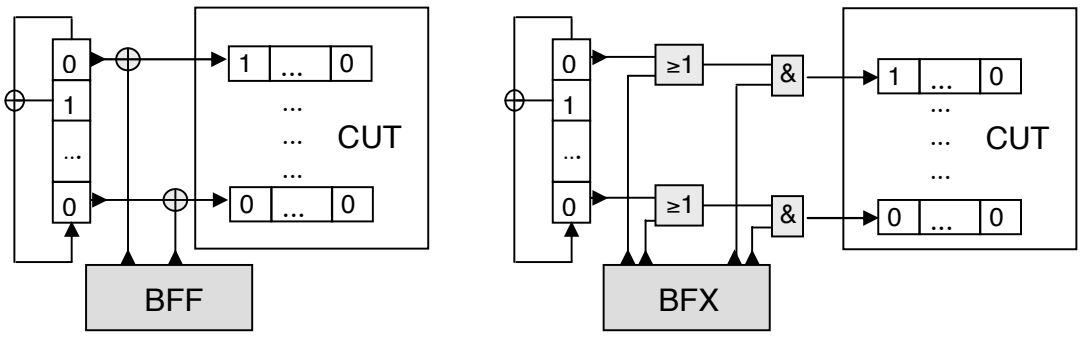
There is a wide range of deterministic logic BIST methods that apply deterministic test patterns and hence improve the low fault coverage often obtained by pseudo-random patterns. Straightforward is the application of additional external deterministic patterns on top of the pseudo-random test [8]. Unfortunately, the very last percentages of fault coverage require the largest amount of deterministic patterns, so the benefits of LBIST are severely reduced by this approach.

More efficient are compression and decompression methods, where a small amount of external test data is continuously fed into the circuit [12][14]. However, this approach is no longer a BIST method; it requires still external ATE and looses some benefits of BIST like in-field testing. An alternative for increasing the fault coverage is inserting test points, which has been proposed for both LBIST and external testing [4][5][15][17]. While the area increase due to test points may be tolerable, they may also introduce additional delays, which could require complete resynthesis and new timing verification [18].

In contrast to the abovementioned LBIST methods, pure deterministic LBIST schemes try to avoid both modifying the core under test (CUT) and applying additional patterns. Their underlying methods can be classified into “store and generate” schemes and “test set embedding” schemes [20].

“Store and generate” schemes consist of hardware structures which store the test patterns on-chip in a compressed form and implement an algorithm for decompression.

Widely known representatives of this method are LFSR-reseeding [12], multi-polynomial reseeding [6][7] and folding counter based-LBIST [13].



(a) (b)

Figure 1: (a) Bit-flipping and (b) bit-fixing BIST schemes.

"Test set embedding" schemes rely on a pseudo-random test pattern generator plus some additional circuitry that modifies the pseudo-random sequence in such a way that a set of deterministic patterns is embedded. Widely known techniques are bit-flipping [9][10][11][19] and bit-fixing [16].

In the bit-flipping approach, the output sequences of an LFSR are inverted at a few bit positions in order to increase fault coverage (Figure 1.a), while the bit-fixing approach applies constant values (Figure 1.b). The test generation process is controlled by a bit-flipping function (BFF) or a bit-fixing function (BFX), respectively.

We use the term *pattern mapping* for referring to the embedding of a set of deterministic patterns into a sequence of pseudo-random patterns. A DLBIST synthesis procedure consists of pattern mapping and generation of the hardware structure to implement the mapping, e.g. by means of a BFF or BFX. The synthesis procedure for generating the BFX as published in [16], is based on rectangle covering, while the synthesis procedure for generating the BFF as published in [9][19][10], is based on manipulating sets of test cubes. In both cases, the procedures are based on heuristics that generally require at least cubical, but often exponential, effort in terms of memory consumption and computing time.

In this paper, we present a BDD-based algorithm for test pattern mapping that outperforms previously published algorithms by several orders of magnitude. The paper is organized as follows: in Section 2 a more formal definition of the pattern mapping problem is given. Section 3 describes the BDD-based synthesis in detail and Section 4 shows the significant improvements with the help of a set of industrial benchmark circuits.

2. The pattern mapping problem

The "test set embedding" schemes provide both pseudo-random and deterministic test stimuli. Usually, some of the pseudo-random patterns generated by an LFSR, are altered into deterministic test stimuli. Most of the pseudo-random test patterns do not contribute to the fault coverage, since they only detect faults that already were detected by other pseudo-random patterns. Such *useless* pseudo-random test patterns may therefore be skipped or modified in any arbitrary way. The key idea is to modify some useless pseudo-random patterns into useful deterministic test patterns to improve the fault coverage. The deterministic test patterns are determined by an ATPG tool, and they target those faults that are not detected by pseudo-random test stimuli. In such a deterministic test pattern, only few bits are actually specified, while most of the bits are don't care and hence can arbitrarily be set to 0 or 1.

The method presented here can be applied to both the bit-flipping and bit-fixing approach, assuming a few modifications. For the sake of simplicity, we will explain the method by using the bit-flipping approach; also the experimental results are given for this method.

In the bit-flipping approach, the modification of the pseudo-random patterns is realized by inverting *(flipping)* some of the LFSR outputs, such that the deterministic stimuli are obtained. The flipping is implemented by combinational logic, called bit-flipping function (BFF). The BFF

can be kept quite small by exploiting the large number of useless pseudo-random test patterns that may be modified, and carefully selecting the pseudo-random test patterns on which deterministic test patterns are mapped.

As shown in Figure 2, the BFF inputs are connected to the LFSR, the pattern counter, and the shift counter, while the BFF outputs are connected to the XOR-gates at the scan inputs. The BFF determines whether a bit has to be flipped based on the states of the LFSR, the pattern counter, and the shift counter. The pattern counter is part of the test control unit, and counts the number of test patterns applied during the self-test. The shift counter is also part of the test control unit, and counts the number of scan shift cycles for shifting data in/out the scan chains.

The BFF realizes the mapping of deterministic test stimuli to pseudo-random test stimuli. Every specified bit (i.e. care bit) in a deterministic stimulus either matches to the corresponding bit in the pseudo-random stimulus, in which case bit-flipping must not be performed, or the bit does not match, in which case bit-flipping is required. For all unspecified bits (i.e. don't-care bits) in the deterministic stimulus, the corresponding bits in the pseudo-random stimulus may be arbitrarily flipped or not. The BFF should provide that (1) all conflicting bits are flipped, (2) all matching bits are not flipped while (3) the don't-care bits may be arbitrarily flipped or not. We first consider a CUT with a single scan chain. The LFSR generates a pseudo-random sequence of test stimuli that is shifted into the scan chain. The LFSR and shift counter (SC) are updated in every clock cycle, while the pattern counter (PC) is updated when applying a new test pattern. In every clock cycle, the DLBIST hardware therefore has a unique state identified by the states of the LFSR, PC, and SC. The set S denotes the set of all possible states of LFSR_PC_SC (here the symbol '_' indicates concatenation).

The *on-set* is the set of LFSR_PC_SC states that correspond to the clock cycles in which the pseudo-random LFSR output should be flipped. Similarly, the *off-set* is the set of LFSR_PC_SC states that correspond to clock cycles in which the pseudo-random LFSR output should not be flipped. The *don't-care set* (*dc-set*) is the set of LFSR_PC_SC states that correspond to clock cycles in which the pseudo-random LFSR output may be arbitrarily flipped or not. The *on-set* and *off-set* are disjoint ($on\text{-}set \cap off\text{-}set = \varnothing$). The *dc-set* contains all states that are not in the union of the *on-set* and *off-set*: $dc\text{-}set = S \setminus (on\text{-}set \cup off\text{-}set)$. The *on-set*, *off-set*, and *dc-set* specify the operation of the BFF. The *dc-set* is exploited to minimize the logic implementation of the BFF.

The *on-set*, *off-set*, and *dc-set* express a Boolean function $\varphi(\{0,1\}^k) \rightarrow \{0,1,\text{-}\}$ where k corresponds to the size (i.e. number of bits) of the LFSR, PC, plus SC. The symbol '-' indicates don't care. For instance, consider a simple example of DLBIST hardware with a 2-bit LFSR, a 2-bit PC, and a 2-bit SC. $\varphi(01\_10\_01) = 1$ now indicates that the pseudo-random bit should be flipped when the LFSR state is 01, the PC state is 10, and SC state is 01. The state 01_10_01 is therefore part of the

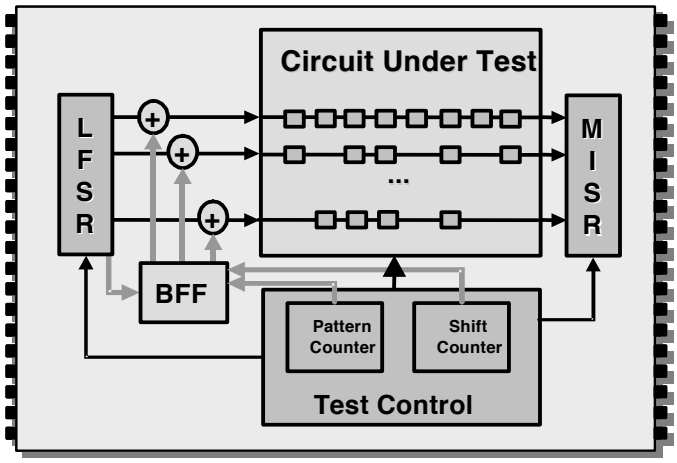


Figure 2: Bit-flipping DLBIST architecture.

on-set. φ(01_10_11) = 0 indicates that the pseudo-random bit should not be flipped when the LFSR state is 01, the PC state is 10, and SC state is 11. The state 01_10_11 is therefore part of the *off-set*. φ(10_01_01) = '-' indicates that the pseudo-random bit may be flipped or not when the LFSR state is 10, the PC state is 01, and SC state is 01. The state 10_01_01 is therefore part of the *dc-set*.

In case of a CUT with multiple scan chains, there are separate on-sets, off-sets, and dc-sets associated with each scan chain. For a CUT with n scan chains, the sets are $on\text{-}set_i$, $off\text{-}set_i$, and $dc\text{-}set_i$ for scan chain i, $1 \le i \le n$. The BFF now consists of the n bit-flipping logics BFF_i for each scan chain. The size of the BFF implementation can be minimized by sharing logic between the BFF_i for various scan chains.

In the original bit-flipping synthesis [19] [9] [10], the sets are represented as sets of k-bit cubes. A cube is element of the set $\{0,1,-\}^k$, and corresponds to a sequence of k bits that are '0', '1', or '-'. The original synthesis procedure is based on Espresso-like logic optimization using the cube-representation [2], and results in a two-level logic implementation of the BFF.

The size of the *on-set* and *off-set* increases with the number of specified bits and in contrast to standard logic synthesis problems, the cubes in these sets are very irregular. Hence, logic minimization exploiting the *on-set*, *off-set* and *dc-set*, may have exponential complexity in terms of the number of specified bits.

3. BDD-based pattern mapping

A binary decision diagram (BDD) is a well-known representation of a logic function [1]. A BDD is a tree-like directed graph, starting from a root vertex. A BDD contains non-terminal vertices that have two outgoing edges and terminal vertices that only have incoming edges. For example, Figure 3 shows the BDD representation of a parity function. The function *parity(a,b,c)* operates on the input variables a, b, and c. The function result is 0 if there is an even number of input variables that have value 1, while the function result is 1 if there is an odd number of input variables that have value 1. For instance, *parity*(011) = 0 and *parity*(010) = 1. The labels at the edges correspond to the variable value of the parent vertex. The BDD-based representation of the parity function with n input variables contains $2n+1$ vertices, while a cube-based representation of the same function would require 2^{n-1} cubes. The example illustrates that a BDD may be a very compact representation for certain logic functions. A second advantage of BDDs is that the complexity of many operations on a BDD scales linearly with the number of input variables [3].

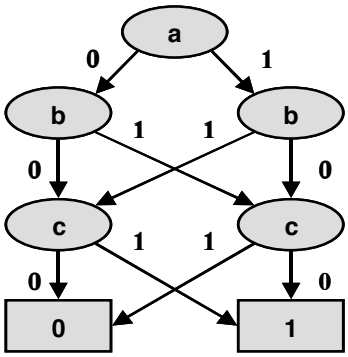


Figure 3: BDD representation of parity function.

In the BDD-based bit-flipping synthesis procedure, the *on-set* and *off-set* of the BFF are represented by characteristic functions, the *on-BDD* and the *off-BDD*. The *on-BDD* will output the value '1' if the input is taken from the on-set, otherwise the output is '0'. Similarly, the *off-BDD* will output '1', only if the input is selected from the off-set. Checking whether an assignment is element of the off-set or the on-set is linear in the number of input variables of the BDD, whereas the cube-based representation requires an effort linear in the cardinality of the sets.

In the presented approach, the sequence of test stimuli is partitioned into two parts. The first part of the sequence is used only for pseudo-random fault detection, and no deterministic stimuli are embedded into this part. The outputs of the BFF should be disabled during this part. The LFSR_PC_SC states for this first part of pseudo-random test stimuli are included in the *dc-set*, since increasing the *dc-set* gives more room for logic optimization of the BFF. However, the BFF will arbitrarily flip some pseudo-random pattern-bits, and some detected faults may no longer be detected by the modified sequence with bit-flipping. In general, most faults are quickly detected by the first few hundreds or thousands pseudo-random test patterns. Disabling the BFF can be achieved using some simple additional circuitry that considers the most significant bits of the PC.

All deterministic patterns are embedded into the second part of the sequence, during which the BFF is enabled. The second part usually is 1/2, 1/4, or 1/8 of the total test sequence.

In general, the sizes of BDDs may grow exponentially with the number of input variables. For BDD-based synthesis of the BFF, the only input variables are the LFSR, the PC, and the SC. In practice the LFSR size is typically below 64 bits, the PC size is below 18 bits (which allows to generate 2^{18} = 256k patterns), and the SC size is below 12 bits (which allows scan chains with maximum 2^{12} = 4,096 flip-flops). Hence, the number of inputs to the BFF is below 64+18+12=94 bits. The sizes of the BDDs for representing such BFF are therefore within practical limits that can be handled by state-of-the-art computers and BDD software packages, and the complexity of the main operations used here are linear in the BDD size. In contrast to that, the sizes of the *on-set* and *off-set* in the cube-based representation increase linearly with the number of specified bits, which is increasing with test set and CUT size, and the operations on the cube sets are up to exponential.

The BDD-based bit-flipping synthesis procedure is outlined in Figure 4. The steps are in detail:

1. Fault simulation is performed with the sequence of pseudo-random test patterns as generated by the LFSR, to determine which faults are detected by the pseudo-random patterns.

2. ATPG is used to generate compact deterministic test patterns for all faults that are not detected by the pseudo-random patterns. The deterministic patterns contain a large number of don't-care bits.

3. The deterministic ATPG patterns are mapped onto pseudo-random test patterns. The BFF is created such that the identified pseudo-random test stimuli are modified into the deterministic patterns by flipping the appropriate bits. The mapping is done such that the size of the subsequent BFF implementation is minimized, which can be achieved by exploiting the *dc-set*.

   The first attempt is to assign a pseudo-random stimulus to a deterministic stimulus such that a minimum number of bits are conflicting. The mapping is further optimized using a combination of the following attempts:

   - Minimize the number of clock cycles with both matching and conflicting bits. This attempts to maximize the sharing of logic between BFF_i implementations for different scan chains.

   - Minimize the number of matching and conflicting bits per scan chain. This attempts to decouple the *on-BDD* and *off-BDD* for each scan chain with respect to the state of the PC. This increases the degrees of freedom for optimizing the corresponding BFF_i implementations.

4. The BDD-based representation of the BFF is transformed into a hardware structure description (e.g. RTL VHDL or Verilog). The RTL description can be synthesized using commercial logic synthesis tools.

5. Fault simulation is performed with the sequence of test stimuli as generated by the LFSR and the BFF.

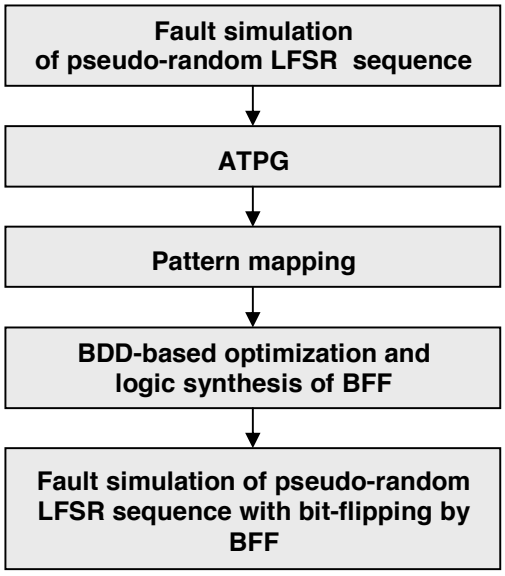


Figure 4: Bit-flipping synthesis procedure.

4. Experimental results

Below, experiments are reported performed on Linux GNU machines equipped with one GB of memory and an AMD Atlon-XP processor running at 1500 MHz. The BDD-based computations were implemented using the *CUDD* package [21].

The benchmark circuits are industrial designs described in Table 1. The first column reports the circuit name encoded like *pN*, where *N* denotes the number of gates in the circuit. The second column gives the number of scan flip-flops contained in each design. The last two columns report the fault coverage and the fault efficiency obtained after applying 10,000 pseudo-random patterns, which are the percentage of detected faults and the percentage of detected and redundant faults, respectively, with respect to the total number of faults.

While the original cube-based pattern mapping is an iterative algorithm [19], where ATPG, pattern mapping and fault simulation are alternating, the BDD-based algorithm is a single pass algorithm, which involves ATPG and fault simulation less often. Hence computing time savings are not only due to substituting the cube-based approach by BDD-based algorithms, also for ATPG and fault simulation computing time is saved.

Table 2 shows that mapping time is reduced from several days down to a few minutes, and that also the other tasks have significant improvements. The overall computing time (including also the time spent during the BDD- based synthesis) and the memory consumption are given in Table 3. The BDD-based approach reduces computing time from more than a week down to several hours, while also the memory requirements scale quite well with the circuit size.

Design	# Flip-flops	Random fault coverage [%]	Random fault efficiency [%]
p19k	1407	63.11	71.68
p59k	4730	87.30	97.30
p127k	5116	82.14	84.30
p278k	9967	79.92	81.61
p333k	20756	93.64	95.65
p951k	104624	92.91	92.92
p2074k	58835	64.11	94.83

Table 1: Benchmark characteristics.

No results are available in Table 2 and 3 with the cube-based approach for the two largest designs due to excessive run-time and memory requirements.

Finally, the amazing improvements should not be paid by less quality in terms of fault efficiency and

	Cube-based			BDD-based		
Design	ATPG time [h:m]	Mapping time [h:m]	Fault simulation time [h:m]	ATPG time [h:m]	Mapping time [h:m]	Fault simulation time [h:m]
p19k	00:00	02:57	00:33	00:00	00:02	00:01
p59k	00:05	02:20	00:30	00:01	00:02	00:03
p127k	02:22	76:54	18:25	03:10	00:14	00:12
p278k	05:20	193:10	37:23	02:29	00:09	00:22
p333k	00:48	116:15	47:45	00:37	00:14	00:17
p951k	-	-	-	01:14	03:12	00:57
p2074k	-	-	-	02:55	03:59	00:35

Table 2: Run-time for different tasks of the cube-based and BDD-based algorithm. For the design 'p2074k' a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz was used.

	Cube-based		BDD-based	
Design	Total time [h:m]	Total memory [MB]	Total time [h:m]	Total memory [MB]
p19k	03:30	58	00:27	58
p59k	02:55	138	00:11	66
p127k	97:41	368	11:13	211
p278k	235:53	584	15:21	318
p333k	164:48	660	09:07	290
p951k	-	-	14:22	1106
p2074k	-	-	18:37	1865

Table 3: Run-time and memory consumption of the cube-based and BDD-based algorithm. For the design 'p2074k' a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz was used.

hardware overhead. Table 4 reports the fault efficiencies obtained in both cases. In order to have comparable results of time and memory, the fault efficiency of the BDD-based approach was limited to the one reached by the cube-based approach. By spending more resources, even higher fault efficiency could be obtained, only limited by the resources given to the ATPG tool. The last column (C*ell area*) shows the logic overhead of the BFF implementation relative to the cell area of the CUT, obtained using a commercial synthesis tool and a proprietary library. Only logic overhead of the BFF implementation is given; the overhead of the other parts of the DLBIST hardware may be neglected. Again, the BDD-based approach outperforms the cube-based approach.

Table 5 illustrates how the computational resources are scaling when the targeted fault efficiency is increased to levels allowed by the ATPG tool. Most of the additional run-time is consumed during the deterministic pattern generation and the BDD-based synthesis, while the time spent for fault simulation remains constant. These final fault efficiencies are practically not reachable by the cube-based approach in the case of the last four designs. Additionally, Table 5 shows that the overhead ratio decreases significantly for the larger designs. The presented approach does not only scale very well in terms of computing time and memory, but also in terms of area overhead.

During the generation of the BDD-based representation, no static or dynamic variable reordering was used. The variables were a priori and optimally arranged in groups corresponding to the states of the LFSR, PC and SC. The reported experimental results were obtained with the same variable order for all the designs.

	Cube-based		BDD-based	
Design	Fault efficiency [%]	Cell area [%]	Fault efficiency [%]	Cell area [%]
p19k	96.86	89.67	97.68	21.71
p59k	99.06	7.64	99.15	3.59
p127k	94.61	27.86	95.57	9.81
p278k	90.83	25.77	91.62	9.66
p333k	97.46	12.07	97.52	3.56

Table 4: Fault efficiency and logic overhead of the cube-based and BDD-based algorithm.

Design	# Embedded patterns	Fault efficiency [%]	Total time [h:m]	Memory [MB]	Cell area [%]
p19k	181	99.26	00:32	91	25.36
p59k	137	99.19	00:11	68	3.75
p127k	582	99.27	18:20	295	21.81
p278k	1549	98.89	55:37	536	34.58
p333k	1298	99.31	23:00	359	7.00
p951k	259	99.67	14:22	1106	1.49
p2074k	302	99.29	18:37	1865	2.64

Table 5: Results obtained with the BDD-based approach reaching a fault efficiency level close to 100%. For the designs 'p278k' and 'p2074k' a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz was used.

5. Conclusions

A new pattern mapping algorithm for "test set embedding" deterministic BIST schemes was proposed which exploits standard BDD operations. This way, improvements of several order of magnitude are obtainable compared with the cube-based approach, e.g., in terms of both run-time and memory requirements. With this approach, computing and memory resources for DLBIST synthesis are in the same order of complexity as the resources required for ATPG or fault simulation. The gains of efficiency can also be used to obtain even better solutions in terms of hardware overhead and fault coverage.

Acknowledgments

This research work was supported by the German Federal Ministry of Education and Research (BMBF) in the Project AZTEKE under the contract number 01M3063C.

References

[1] S.B. Akers „Binary Decision Diagrams," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 509-516.

[2] R.K. Brayton, G.D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli „Logic Minimization Algorithms for VLSI Synthesis," *Kluver Academic Publishers*, 1997.

[3] R.E. Bryant „Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, C-35-8, August 1986, pp. 677-691.

[4] M.J. Geuzebroek, J.Th. van der Linden, A.J. van de Goor „Test Point Insertion for Compact Test Sets," *Proceedings of International Test Conference*, IEEE, 2000, pp. 506-514.

[5] J.P. Hayes, A.D. Friedman „Test Point Placement to Simplify Fault Detection," *IEEE Transactions on Computers*, Vol. C-33, July 1974, pp. 727-735.

[6] S. Hellebrand, S. Tarnik, J. Rajski, B. Courtois „Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers," *Proceedings of International Test Conference*, 1992, pp. 120-129.

[7] S. Hellebrand, B. Reeb, S. Tarnick, H.-J. Wunderlich „Pattern Generation for a Deterministic BIST Scheme," *Proceedings ACM/IEEE International Conference on CAD-95 (ICCAD95)*, San Jose, CA, November 1995, pp. 88-94.

[8] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, J. Rajski „Logic BIST for Large Industrial Designs: Real Issues and Case Studies," *Proceedings of International Test Conference*, IEEE, 1999, pp. 358-367.

[9] G. Kiefer, H.-J. Wunderlich „Using BIST Control for Pattern Generation," *Proceedings International Test Conference*, IEEE, 1997, pp. 347-355.

[10] G. Kiefer, H.-J. Wunderlich „Deterministic BIST with Multiple Scan Chains," *Proceedings International Test Conference*, IEEE, 1998, pp. 1057-1064.

[11] G. Kiefer, H. Vranken, E. J. Marinissen, H.-J. Wunderlich „Application of Deterministic Logic BIST on Industrial Circuits," *Proceed-*

ings IEEE International Test Conference, ITC 2000, Atlantic City, NJ, October 3-5, 2000, pp. 105-114.

[12] B. Koenemann „LFSR-Coded Test Patterns for Scan Designs," *Proceedings of European Test Conference*, 1991, pp. 237-242.

[13] H. Liang, S. Hellebrand, H.-J. Wunderlich „Two-Dimensional Test Data Compression for Scan-Based Deterministic BIST," *Proceedings IEEE International Test Conference*, *Journal of Electronic Testing - Theory and Applications (JETTA)*, Vol. 18, No. 2, April 2002, pp. 157-168.

[14] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide, J. Qian „Embedded deterministic test for low cost manufacturing test", Proceedings of *International Test Conference*, IEEE, 2002, pp. 301-310.

[15] B.H. Seiß, P.M. Trouborst and M.H. Schulz „Test Point Insertion for Scan-Based BIST," European Test Conference (ETC), April 1991, pp. 253-262.

[16] N.A. Touba, and E.J. McCluskey „Altering a pseudo- random bit sequence for scan-based BIST," *Proceedings IEEE International Test Conference*, 1996, pp.167-175.

[17] H. Vranken, F. Meister, H.-J. Wunderlich „Combining Deterministic Logic BIST with Test Point Insertion," *The Seventh IEEE European Test Workshop*, May 26-29, 2002.

[18] H. Vranken, H.-J. Wunderlich, F. Syafei Sapei „Impact of Test Point Insertion on Silicon Area and Timing During Layout," *Design, Automation and Test in Europe*, Paris, 16-20 February 2004.

[19] H.-J. Wunderlich, G. Kiefer „Bit-Flipping BIST," *Proceedings International Conference on Computer Aided Design*, IEEE, 1996, pp. 337-343.

[20] H.-J. Wunderlich „BIST for Systems-on-a-Chip," *INTEGRATION, the VLSI journal*, 1998, pp. 57-78.

[21] http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html