POLITECNICO DI TORINO
Repository ISTITUZIONALE

AFSM-based deterministic hardware TPG

Original
AFSM-based deterministic hardware TPG / Benso, A., DI CARLO, S., DI NATALE, G., Prinetto, P.E.. - STAMPA. - (2005), pp. 178-181. (IEEE 8th Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) Sopron, HU 13-16 Apr. 2005).

Availability:
This version is available at: 11583/1499954 since:

Publisher:
IEEE Computer Society

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

27 November 2026

DDECS
DESIGN & DIAGNOSTICS of ELECTRONIC CIRCUITS & SYSTEMS

8th IEEE Workshop on
Design and Diagnostics of Electronic Circuits and Systems
Sopron, Hungary, April 13 - 16, 2005

AFSM-BASED DETERMINISTIC HARDWARE TPG

Alfredo BENSO, Stefano DI CARLO, Giorgio DI NATALE , Paolo PRINETTO
Politecnico di Torino
Corso Duca degli Abruzzi 24 – 10129 - Torino (Italy)
{alfredo.benso, stefano.dicarlo, giorgio.dinatale, paolo.prinetto}@polito.it

Abstract. *This paper proposes a new approach for designing a cost-effective, on-chip, hardware pattern generator of deterministic test sequences. Given a pre-computed test pattern (obtained by an ATPG tool) with predetermined fault coverage, a hardware Test Pattern Generator (TPG) based on Autonomous Finite State Machines (AFSM) structure is synthesized to generate it. This new approach exploits "don't care" bits of the deterministic test patterns to lower area overhead of the TPG. Simulations using benchmark circuits show that the hardware components cost is considerably less when compared with alternative solutions.*

1 Introduction

Several approaches have been proposed in literature to build TPGs in BIST architectures: (i) exhaustive testing, where a simple counter is used to generate exhaustively all combinations of test patterns [1]; (ii) pseudo-random testing that exploits Linear Feedback Shift Registers (LFSRs) [2], or linear Cellular Automata (CAs) [3] to generate pseudo random patterns; (iii) hardware generators of deterministic test sequences, where a set of pre-computed test patterns, generated by an Automatic Test Pattern Generator (ATPG), are internally generated by ad-hoc logic [4], or by CA-based structures [5] [6]; and (iv) mixed-mode test pattern generation, that combines the test features of the pseudo-random and deterministic generation approaches [7] [8].

This paper proposes a new approach for designing a cost-effective, on-chip hardware pattern generator of deterministic test sequences. Given pre-computed test patterns (computed by an ATPG tool) with predetermined fault coverage, a hardware Test Pattern Generator based on Autonomous Finite State Machines (AFSM) is synthesized to generate the given test set. This new approach exploits the experimental observation that not all bits of deterministic test patterns generated by ATPG are specified. In many cases, deterministic test patterns consist of a large number of "don't care" bits and a small number of care bits. Studies on compacted test sets showed a "care" bit density of 1% -5% [9].

2 Algorithm description

The basic idea is to use an Autonomous Synchronous Finite-State Moore Machine as shown in Figure 1a. If we assume no test vectors are identical, then there is a one-to-one correspondence between states and outputs (if this condition is not true, it is however

possible to implement a different machine able to generate the same test vectors, adding extra flip-flops). In this case the Moore machine can be simplified as shown in figure 1b.

Given a deterministic binary sequence, our goal is to find an AFSM (that we call AFSM_TPG) that evolves exactly through that sequence. Therefore the problem is to design a combinational circuit (the δ function) implementing the correct state transitions function.

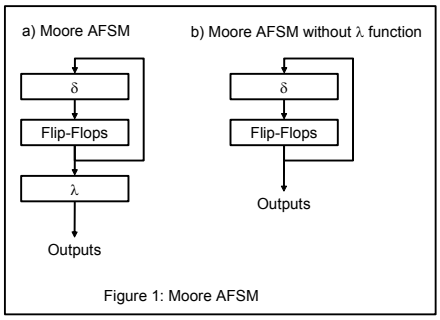


Figure 1: Moore AFSM

The deterministic sequence can be represented using a matrix M(r,c) where r and c represent the row and the column of the matrix respectively, and M(r,c) represents the bit that has to be generated in the r^{th} clock cycle by the c^{th} flip-flop of the AFSM_TPG.

The goal of the proposed algorithm is therefore to compute M(r,c) as a function of M(r-1, "*some other c*").

To do so, we first need to map the concept of collision on the matrix M; we say that there is a collision in column *c* when there are two rows r1 and r2 of the matrix M so that:

$$[r1/r2,c] \equiv \begin{cases} M(r1,c) = M(r2,c) \\ M(r1+1,c) \neq M(r2+1,c) \end{cases}$$

We start from an AFSM_TPG in which each cell depends only on itself. A new dependency is inserted whenever a collision is detected. If adding a new dependency does not achieve the solution a new column is inserted in the matrix M. Purpose of the algorithm is to minimize the number of dependencies and the number of additional columns. To solve a collision, the algorithm goes through the following steps:

```
∀ column C {
   Classification of transitions
   Don't Care Substitution
   If (Collision detected) {
     Don't Care Substitution
     Collision Matrix
     While Not (SetCovering) {
       Add Column
     }
   }
}
```

We detail each routine, considering C as the column under elaboration:

- **Classification of transitions**: Since a collision is detected whenever there is a pair of rows in which the matrix has a transition from 0 to 0 (or 1 to 0) and another row in which there is a transition form 0 to 1 (or 1 to 1), the algorithm classifies each row of the matrix based on the performed boolean transition (Transitions Classes). For each transition we define a transition class including all the rows subject to that transition (class Z_0: transition from 0 to 0; class Z_1: from 0 to 1; class O_0: from 1 to 0; class O_1: from 1 to 1).
- **Collisions detection**: Each possible pair of rows from classes Z0 and Z1 and from classes O0 and O1 generates a collision. If no collisions are present in the system, the AFSM_TPG is able to generate the sequence of bits in column c without any other dependency.
- **Collisions matrix**: All the collisions are stored in a matrix called Collision Matrix T. Each row of this matrix represents one of the possible collisions [r1/r2,C] whereas the columns Cx are those of the Matrix M (except C) that could be used as new

dependencies for column C. Each element of the matrix (T) is a boolean value whose value is 'true' if column Cx in the Matrix M assumes two different values in the collision rows (r1-r2).

- **Set Covering**: It is used to find the minimum number of columns to be used as new dependencies in order to remove all the collisions. When the problem has no solution, a new "dummy" column is added to the original matrix M. In this case the values of the bits of the new column are set to "don't care" except for the two bits corresponding to the colliding rows, which are set to '0' and '1', respectively.
- **Don't care substitution**: The "don't care substitution" is performed in two situations:
  1) Each time a new column is considered by the algorithm, all the "don't cares" present in the column are set in order to minimize the product of the cardinalities of classes Z0 and Z1, and of classes O0 and O1.
  2) When a collision is detected between two rows, if, on the same row, it is possible to find two "don't cares", setting one to '0' and the other to '1' will assure that the set covering will find a solution to resolve this collision.

3 Experimental results

In our experiments we used ISCAS-85 data and obtained test vectors sequences using the ATPG tool[1] from Synopsys. The proposed algorithm is implemented in C language and we run the experiments on a Intel Pentium IV with 256 Megs of RAM.

We compare our results with those presented in [5] and [6] that are pure deterministic methods as our AFSM_TPG method. We calculated the size of the resulting AFSM_TPG using the same equations defined by the authors in [5] and [6]:

Area = (# of inputs) * (# of product terms) * (# of outputs)

In our experiments, the total area is the sum of the areas of the δ function of each column. We computed the area of each δ function as:

δ Area = (# of dependencies) * (# of product terms of the δ Truth Table) * (1)

Table 1 shows the results of our experiments, whereas Table 2 compares our results with the ones presented in [5] and [6].

Table 1: Results of our experiments

CUT	Execution Time	Rows (Patterns)	Columns	Added Columns	% Don't Care	Size
c432	462m	90	36	0	57,22222	9776
c499	452m	67	41	0	16,01747	3727
c880	80m	53	60	0	58,27044	7500
c1355	634m	81	41	0	7,768744	9007
c1908	31m	68	33	0	41,39929	6324
c3540	243m	180	50	1	57,86667	29698
c5315	128m	90	178	50	73,05243	41315
c6288	3m	59	32	0	4,502119	5348

[1] In order to obtain "don't care" bits in the generated sequence, the command used with this tool is "`testgen -autotime -norandomfill -combcompact`".

Table 2: Comparison between our results and [5] and [6]

CUT	Size	Size [6]	Size [5]	Red. [6]	Red. [5]
c432	9776	18668	29376	48%	67%
c499	3727	8457	37720	56%	90%
c880	7500	31708	47520	77%	84%
c1355	9007	26842	52808	67%	83%
c1908	6324	53791	48576	88%	87%
c3540	29698	115948	252000	74%	88%
c5315	41315	288007	3262384	86%	99%
c6288	5348	6797	9472	21%	44%

All the benchmarks show that the proposed approach is able to produce hardware TPGs considerably smaller than the ones presented in [5] and [6].

4 Conclusions

Despite the excellent results obtained so far, the proposed approach can still benefit from many other possible optimizations that we are investigating. Among them the authors would like to mention:

- Trade-off between number of dependencies and number of additional columns. In general, adding a dependency will add combinational logic, whereas adding a column will add a flip-flop. It is not necessarily true that one addition is better than the other. We are therefore exploiting the performances of the algorithm when constraining the number of additional columns on the number of dependencies.
- A lot of work can still be done in the optimization of the "don't care" substitution routine
- Experimental data show reasonable run time for the tiny benchmarks. To scales with large industrial circuits with large number of scan cells a mixed-mode method has to be investigated

5 References

[1] E. J. McCluskey, "Built-In Verification Test", Digest, 1982, IEEE Test Conference, pp.183-190, Nov. 1982

[2] M. Abramovici, M. A. Breuer, A. D. Friedman, "Digital Systems Testing and Testable Design"; Computer Science Press, New York, 1990

[3] P. D. Hortsenius, "Cellular automata based signature analysis for Built-In Self-Test", IEEE Trans. Computer, 1990, Vol. 39, No. 10, pp. 1273-1283

[4] R. Dandapani, J. H. Patel, J. A. Abrham, "Design of Test Pattern Generator for Built-In Self-Test", IEEE International Test Conference, 1984, pp.315-319

[5] S. Boubezari, B. Kaminska, "A deterministic Built-In Self-Test Generator based on Cellular Automata Structures", IEEE Trans. Computer, 1995, Vol. 44, No. 6, June 1995, pp.805-816

[6] M. Guler, H. Kilic, "Built-In Self-Test Generator Desing using Nonuniform Cellular Automata Model", IEE Proceedings of Circuits Devices Systems, Vol. 145, No. 3, June 1998

[7] M. Karkala, N. Touba, H.J. Wunderlich, "Special ATPG to Correlate Test Patterns for Low-Overhead Mixed-Mode BIST", IEEE 7th. Asian Test Symposium, pp. 492, December 1998

[8] F. Brglez, C. Gloster, G. Kedem, "Hardware-Based Weighted Random Pattern Generation for Boundary Scan", IEEE International Test Conference, 1989, pp. 264-274

[9] T. Hiraide, K. O. Boateng, H. Konishi, K. Itaya, M. Emori, H. Yamanaka, Y. Mochiyama, "BIST-Aided Scan Test – A New Method for Test Cost Reduction", Proc. IEEE VLSI Test Symposium, 2003, pp. 359