Non-Intrusive BIST for Systems-on-a-Chip

Silvia CHIUSANO*, PAOLO PRINETTO*, HANS-JOACHIM WUNDERLICH [+]

(*) **Politecnico di Torino**
Dipartimento di Automatica e Informatica
Corso duca degli Abruzzi 24 - I-10129, Torino, Italy
Email: {chiusano, prinetto}@polito.it
http://www.testgroup.polito.it

(+) **University of Stuttgart**
Computer Architecture Lab,
Stuttgart, Germany
Email: wu@informatik.uni-stuttgart.de
http://www.ra.informatik.uni-stuttgart.de

Abstract[1]

The term "functional BIST" describes a test method to control functional modules so that they generate a deterministic test set, which targets structural faults within other parts of the system. It is a promising solution for self-testing complex digital systems at reduced costs in terms of area overhead and performance degradation. While previous work mainly investigated the use of functional modules for generating pseudo-random and pseudo-exhaustive test patterns, the present paper shows that a variety of modules can also be used as a deterministic test pattern generator via an appropriate reseeding strategy. This method enables a BIST technique that does not introduce additional hardware like test points and test registers into combinational and pipelined modules under test. The experimental results prove that the reseeding method works for accumulator based structures, multipliers, or encryption modules as efficiently as for the classic linear feedback shift registers, and some times even better.

1. Introduction

Embedded Test is today widely recognized as an effective approach to Systems-on-Chip (SoC) testing, while traditional methods based solely on an external automatic test equipment (ATE) become more and more expensive or even unfeasible. *Built-in Self-Test* (BIST) strategies embed the functions needed for testing a given *Unit Under Test* (UUT) into the chip itself. These functions consist of a *Test Pattern Generator* (TPG) and a *Test Response Compactor* (TRC) at least, and until now, they have been performed by specialized dedicated hardware mainly based on Linear Feedback Shift Register (LFSRs) or cellular automata.

Traditionally, the TPG generates pseudo-random or pseudo-exhaustive patterns [1][2], and test points may be required either for circuit segmentation or for increasing controllability and observability in order to obtain sufficient fault coverage [3][4][5]. Test points do not only increase the hardware overhead but they may also put additional delays on critical paths and slow down the system performance.

An alternative to test point insertion is not modifying the UUT but the pattern generators. For this purpose, test methods based on weighted random patterns [6][7][8] and, more recently, based on deterministic test patterns have been developed [9][10][11][12]. The reseeding technique presented in [9][13] computes initial values of an LFSR so that the output sequence includes pre-computed deterministic test patterns.

Recently an innovative BIST technique has been proposed which exploits the system functionalities for test generation and is less intrusive than using test registers. The main idea of the *Arithmetic BIST (ABIST)* methodology is to perform test pattern generation by exploiting the available system structure, which consists of both the modules present in the system and the available connections between the modules [14].

In Figure 1, the modules M_i and M_j are part of the system mission logic; they are functionally connected in such

[1] This work was supported in part by the *Deutscher Akademischer Austauschdienst* (DAAD), in part by the *Conferenza dei Rettori delle Università Italiane* (CRUI), under the *Vigoni Project* 1999-2000 and by DFG Wu 245/1-3

a way that the M_i outputs are the input signals of M_j. In the ABIST approach, M_i is controlled appropriately in order to generate output values that are suitable test patterns for M_j.

Typically, M_i is a sequential circuit used as TPG for a given Unit Under Test (UUT), in our case M_j. The additional test hardware mainly consists in the control logic: the multiplexer (MUX) must be controlled so that the best candidate for pattern generation is selected from the connected components (in our case M_i instead of M_k or M_t), and M_i has to run autonomously.

Also the test responses may be observed and compacted by modules already available in the system. This has already been dealt with in [15][16] and is not part of the present paper.

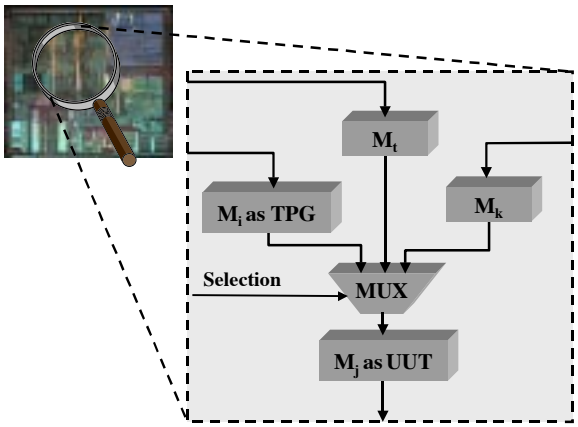

Figure 1: The ABIST approach

The most relevant advantage of the ABIST approach is the avoidance of dedicated test hardware, since now system modules are used for pattern generation, and neither additional hardware nor additional delays are introduced into the data path.

The ABIST technique is comprehensively described in textbooks [17]. The technique addresses general purpose computing structures based on data-path architectures, as well as specialized digital signal processing circuits, performing arithmetic operations, which can be exploited for test pattern generation. Typically, ABIST-based approaches use accumulator-based structures for generating *pseudo-random* and *pseudo-exhaustive patterns* [14][18][19][20]. Modules containing hard-to-detect faults still require extra test hardware either by inserting test points into the mission logic or by storing additional deterministic test patterns.

The goal of the present paper is to combine the advantages of the ABIST approach and the reseeding method for deterministic pattern generation, which was restricted to linear feedback shift registers until now. A method is presented to compute initial values for general functional modules so that they are able to produce deterministic test patterns with complete fault coverage. The method is based on Genetic Algorithms and can handle arbitrary sequential modules as pattern generators. Since it is not restricted to arithmetic modules but can work of any type of functions, we call the resulting test method *Functional BIST*.

Moreover, the paper describes the implementation of a tool, which supports the selection of the best candidate of the functional modules for deterministic pattern generation. An example of a commercial SoC is analyzed, and it is shown that the functional modules available there can be used for reseeding. Finally some additional, more complex, modules are analyzed as well.

The rest of the paper is organized in the following way: the next section describes the state of the art in functional BIST and introduces some basic concepts. Section 3 describes the trade-off in functional BIST pattern generation; Section 4 maps the reseeding problem for general functional modules to a genetic algorithm and gives an overview of the resulting tool. In Section 5, the experimental results focus on units that are most commonly available in SoCs. Results are presented for several standard accumulator-based arithmetic modules (i.e., *adder*, *multiplier*, *subtracter*), that are usually embedded in the systems either as single modules or integrated in an arithmetic logic unit (ALU) or a multiply accumulator block (MAC). Moreover, as an example of non-arithmetic circuits, an *encryption* unit [21] and an *LFSR* are investigated.

2. State of the Art

As functional BIST is a rather new technique, only a few papers focus on covering not random testable faults via deterministic test patterns generated by system modules during BIST.

[22] proposes two computation methods for the initial values (a simulation-based and an analytic one) using an adder as arithmetic unit, both methods do not mainly target complete fault coverage but test length minimization.

The method presented in [23] also applies to adder-based accumulator structures, and is able to compute seeds so that the resulting test sequences obtain complete fault coverage for all the ISCA85 circuits and the combinational parts of the ISCAS89 circuits [24][25]. The performance of this method is in general better in terms of test length and number of seeds than the LFSR-based reseedings. The serious drawback of this technique is the restriction to adder-based structures. It is not expected that this limitation will generally be overcome in the future as the method models the function of the TPG by binary decision diagrams (BDDs) symbolically. An extension already fails if the pattern generator is a multiplier-based accumulator structure, and the solution presented there is optimized especially and applicable only for adders.

In [26], the authors proposed a *universal* algorithm to control and initialize sequential structures so that they work as a *deterministic* test pattern generator for a given

UUT. An algorithm called *GATSBY (*Genetic Algorithm based Test Synthesis tool for BIST applications) as implemented, and experiments were performed considering adder-based structures. This method does not depend on the function of the pattern generating module and was applied to both adder based structures and LFSRs. The results presented there reached the same efficiency as the specialized methods presented in [13][23], and often outperformed them in terms of test size and test time.

The present paper intends to exploit the flexibility of the method [26] to investigate the usability of the functional BIST approach for testing SoCs. Actual SoCs include a variety of functional units, library modules (e.g., ALU, MAC, LFSR, etc.), as well as custom blocks. Moreover, these modules usually form a strongly connected network, in which each unit is functionally linked to many other system modules either by bus-oriented or by multiplexer-oriented interconnections. In the present paper, for several UUTs, we analyze candidate TPGs, taking into account the parameters *test length*, *area overhead*, and *fault coverage*. The trade-off of each functional unit when exploited as TPG is determined. According to the design requirements the "best" functional unit to be used as TPG can be identified, among the ones functionally connected to the UUT itself.

3. Functional BIST Pattern Generation

The *Test Pattern Generator* (TPG) is generally considered as an accumulator-based unit with an input register and a state register, which are assumed to be partially or fully accessible, either via parallel load or in full-scan mode. The *test sets* generated by the TPG depend on the functionality of the TPG itself, on the initial content of the TPG registers, and on the number of clock cycles the TPG is let evolve.

The test pattern generation mechanism consists of two main phases. First, the TPG *state register* is set to an initial value (δ) and its *inputs register* is fixed at a constant value (σ). The pair of values (δ, σ) is often referred to as the *Seed* of the TPG. Then, the TPG runs autonomously for τ clock cycles. A new pattern p_j appears on the TPG outputs at each clock cycle t_j, $0 \le j < \tau$, and is applied to the Unit Under Test (UUT). The resulting test set (TS) is therefore a function *f* of the *triplet* (δ, σ, τ) and of the functionality φ_{TPG} of the TPG itself:

$$TS = \{ p_j \mid 0 \le j < \tau \} = f(\varphi_{TPG}, triplet(\delta, \sigma, \tau)).$$

Figure 2 sketches the test pattern generation mechanism in the functional BIST approach. This is similar to the classical LFSR reseeding where either the seeds [27], or both the seeds and the feedback function have to be stored [9].

The fault coverage (FC%) and the test length (τ) define the quality of the test set. In the functional BIST approach, the test set effectiveness is correlated to the percentage of patterns that really contribute to increase the fault coverage. Since the test set TS is obtained by the autonomous evolution of the TPG after initial seeding, not all the generated patterns may be useful for testing the UUT. A pattern p_j is a *dummy pattern* if it just covers faults already detected by at least one of the patterns p_k, generated in a previous instant of time ($0 \le k < j$). High percentages of dummy patterns affect the TS effectiveness, since most of the testing time is lost in applying useless patterns to the UUT. In Figure 2, only the patterns p_0, p_2, p_j and $p_{\tau-1}$ (colored in gray) are useful for the testing purpose.

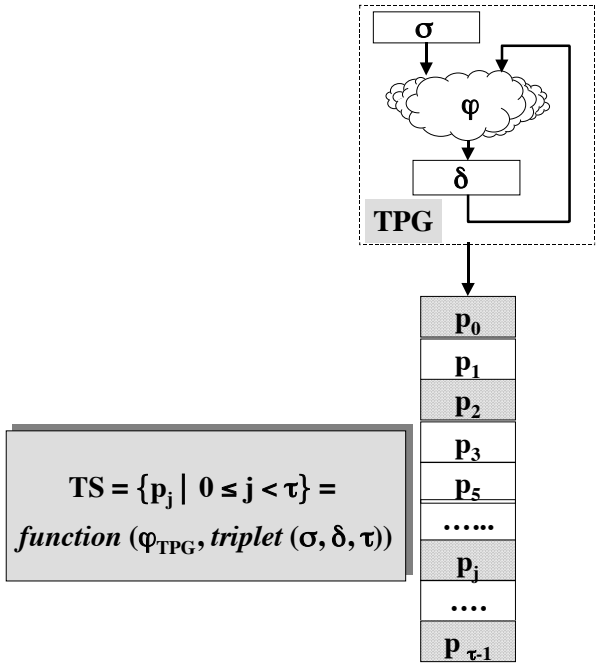

Figure 2: The Test Pattern Generation

The computation of an optimal solution consists in finding a suitable triplet such that the target fault coverage is achieved, while both the test length (τ) and the number of dummy patterns inside the test set are minimized.

A test set provided by a single triplet does not always guarantee to fulfill all the goals at the same time. Experiment showed that random testable faults are detectable in relatively few clock cycles, with an acceptable percentage of dummy patterns inside the test set. On the other hand, a higher number of clock cycles may be required to generate patterns testing not random testable faults. In addition, the TPG functionality may prevent the generation of certain patterns when the TPG is seed by (δ, σ); due, for instance, to states not reachable by the TPG starting from (δ, σ).

In order to reduce the test length or to reach a target fault coverage, the TPG evolution can be periodically stopped and restarted with a new triplet $(\delta, \sigma, \tau)_i$, until the target fault coverage is reached. Such a re-initialization process is called *TPG reseeding*. Figure 3 shows the quali-

tative behavior of the reseeding process. Here, reseeding occur after t_i and t_j clock cycles. Reseeding increases the fault coverage obtained within a fixed number of clock cycles and/or reduce the test length required to get a given fault coverage.

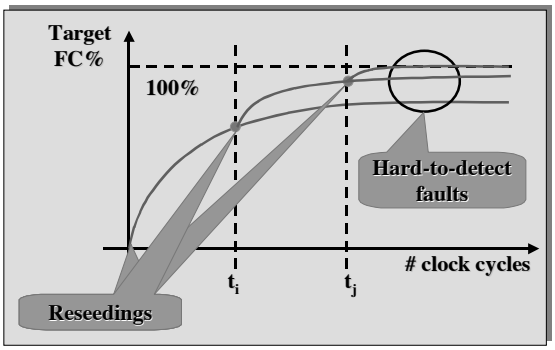

Figure 3: Qualitative behavior

In the case of reseeding, the solution is a set of *N* triplets $\cup_{0 \le j < i}(\delta, \sigma, \tau)_i$, which are sequentially applied to drive the TPG evolution. The overall test set TS is therefore the union of the test sets TS_i generated by each single triplet $(\delta, \sigma, \tau)_i$: $TS = TS_0 \cup TS_1 \cup TS_2 \cup ... TS_{N-1}$. The test set TS is characterized by a global test length $T=\Sigma_{0 \le i < N} \tau_i$, and fault coverage $FC\%=\Sigma_{0 \le i < N} \Delta FC\%_i$. The value $\Delta FC\%_i$ is the percentage of faults detected by TS_i and not covered by the test set $TS_0 \cup TS_1$ È...TS_{i-1}, generated by the subset $\cup_{0 \le j < i}(\delta, \sigma, \tau)_j$. The number N of seeds directly corresponds to the storage effort.

According to the design specification, an optimal reseeding solution can be computed by trading-off the number of reseedings vs. area overhead and test length. In particular,

- *A low number of reseedings* allows minimizing the *extra area* needed to store the triplets (e.g., in a ROM), but usually a larger test length is necessary and the 100% of testable fault coverage is not always guaranteed
- *A large number of reseedings* implies more area overhead, but a shorter test length. The reseeding can be repeated until all the target faults have been detected.

4. The Test Synthesis Tool

The *Optimization Parameters* of GATSBY to be specified drive the computation process; they include the target fault coverage value (FC%), and specify if the solution must minimize either the global test length (T) or the number of reseedings (N).

As a result of the computation, GATSBY provides a set of N triplets, $\cup_{0 \le i < N}(\delta, \sigma, \tau)_i$, tuned towards the Optimization Parameters; in particular, it guarantees the target coverage of non-redundant faults in the UUT. The algorithm is not customized towards a specific functional TPG unit and absolutely general.

Inputs of the program are a behavioral description of the functional TPG module that can easily be extracted if only the structure is known, and the gate-level description of the UUT. At the current state of research only combinational test pattern generation is considered, and the UUT has to be combinational, pipelined or equipped with a scan-path. In order to be compatible with commercial ATPG tools the single stuck-at fault model is targeted, extensions to any other combinational fault models are straight-forward.

The kernel of GATSBY is the *Triplets Generator*, which is implemented by using Genetic Algorithms based procedure and sketched in Figure 4. Genetic Algorithms (GAs) [28] aim at evolving a *population* of *individuals* in order to increase their quality (*fitness*). The population evolves through *generation*s, based on a mechanism that mimics nature. In each generation, the reproduction is performed by the exchange of genetic material *(crossover, mutation*), and the new individuals must compete with their parents for survival: only individuals having higher fitness values will appear in the next generation.

In our case, an individual encodes a single triplet $(\delta, \sigma, \tau)_i$; the population is thus a set of *K* candidate triplets $(\cup_{0 \le i < K}(\delta, \sigma, \tau)_i)$ and the evolution process aims at improving the quality of all of them with respect to the Optimization Parameters specified by the user. The provided optimal reseeding is a minimal subset of *N* triplets $((\cup_{0 \le i < N}(\delta, \sigma, \tau)_i)$, extracted from the last population of *K* individuals, $N \le K$. It has the fault coverage $FC\%=\Sigma_{0 \le i < N} \Delta FC\%_i$, which has to reach the target value, and the global test length $T=\Sigma_{0 \le i < N} \tau_i$. The value $\Delta FC\%_i$ is the percentage of faults detected by $(\delta, \sigma, \tau)_i$ and not covered by the subset of triplets $\cup_{0 \le j < i}(\delta, \sigma, \tau)_j$.

The implemented GA-based procedure traces a quite typical GA structure. At each generation a set of new individuals (triplets) is created, starting from the existing ones, through the *Genetic Operator*s. Standard operators, such as the *horizontal two-cut crossover* and bit-flip based *mutation operators* are adopted. Then the quality of the individuals is assessed and the *Fitness Function* values are used to rank the population. The fitness value for each triplet $(\delta, \sigma, \tau)_i$ is expressed as quality of the test set obtained by processing the TPG with this triplet. As shown in Figure 4, the *Triplet Simulator* computes the test set by seeding the TPG with $(\delta, \sigma)_i$ and running it for τ_i clock cycles. A standard gate-level event-driven *Fault Simulator* applies the test set to the UUT and investigates a target fault list. Three different evaluation parameters contribute in measuring the fitness of an individual:

- The percentage of detected faults of the target fault list ($\Delta FC\%_i$)

- The *circuit sensitization* parameter, to estimate how close the test set is in incrementing its actual fault coverage. In the presence of a target fault not detected by the test set, the *circuit sensitization* counts the number of logic differences between the good and faulty machine injected by a pattern of the test set. For a test set, the value is obtained summing up the maximum circuit sensitization values of its test patterns.
- The number of *dummy pattern*s.

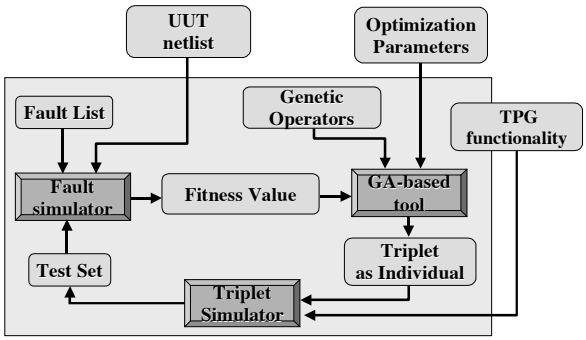

Figure 4: The Triplets Generator

The fitness function uses a multiple reordering procedure, where individuals are ordered mainly based on their contribution to the fault coverage of the whole population. First, $triplet_i$, with highest fault coverage, is stored into the next population. Among the remaining individuals, $triplet_j$, covering the largest subset of faults not detected by $triplet_i$, is then selected. The following choice, $triplet_k$, is done to guarantee the coverage of faults undetected by $triplet_i$ and $triplet_j$. The circuit sensitization and the number of dummy patterns are used to distinguish among individuals having the same fault coverage.

To further increase the efficiency of GATSBY, preprocessing and post-processing phases support the computations by the Triplets Generator and optimize the reseeding solution further, respectively.

The *pre-processing phase* deals with the fault list and the deterministic test set generated by a gate-level ATPG.

The test set is instrumented with the additional information obtained by fault simulation, which evaluates the detection capability of each pattern with respect to the whole fault list. During the GA evolution, some new individuals are set up using patterns extracted from the *instrumented test set*: in particular, at each time patterns covering faults not detected yet by the actual population are selected.

Moreover, the faults are ranked according to their hardness to be tested *(sorted fault list*), and an "incremental" fault list strategy is applied. The GA population is first customized towards a small subset of very hard-to-detect faults, and it is progressively slightly modified to cover easier ones as well, until the whole target fault list is taken into account.

Finally, in the *post-processing phase* the *Triplets Optimizer* performs fault-simulation with each triplet of the reseeding solution in *reverse order*, to reduce the global test length.

5. Experimental Results

The methodology presented so far was evaluated by using the embedded cores of a commercially distributed system-on-a-chip and some behavioral descriptions of even more complex cores.

The next subsection presents the cores used as TPGs and the test method in some detail. In the actual implementation, the TPG functionality is described at the behavioral level in C++. Section 5.2 describes the experimental setup, and section 5.3 compares the results obtained from the different cores and evaluates the trade off in terms of test length, fault coverage and number of seeds to be stored.

The goal of the present paper was to exploit the flexibility of the method presented in [26] to consider different functional blocks as possible TPGs. For a detailed analysis of the proposed approach w.r.t. the state-of-the-art approaches refer to [26].

5.1. Test pattern generating cores

As a demonstrator we use the OAK$^+$ DSP Core™ by VLSI Technology[2]. This is a widely used commercial digital signal processor core for communication applications, and includes a variety of modules: operational blocks (e.g., ALU, multiplier, barrel shifter, etc., a Program Control Unit, and memory). A block diagram is shown in Figure 5.

In the next subsection we investigate the ability of the embedded functions, such as the ALU or the multiplier to test external units under test, which are connected to one of the buses.

As UUTs, a subset of the ISCAS'85 [24] and ISCAS'89 [25] (full scan version) benchmarks circuits that are not randomly testable by 10K patterns has been considered.

A programmable, bus organized system as shown in Figure 9 has the advantage that memory is available and randomly accessible for storing the triplets for reseeding. If the external memory is not available, also the internal one (Status Registers) can be used. Moreover, for this type of systems an external BIST controller is not mandatory, as the Program Control Unit may be available for controlling the test.

Systems which are not as flexible as this test vehicle may require addition memory space for storing the seeds in a ROM, e.g., and may need modifications of the control logic.

[2] OAK$^+$ DSP Core, by VLSI Technology, is a trademark of DSP Group.

Besides the multiplier, the adder and the subtractor from the system describes above, we investigated a *linear-feedback shift register* (LFSR), and an *encryption* unit. Modules for data encryption are today common in a variety of digital systems, developed for different applications.

In our experiments, we use the ANSI C procedure available in the Sun Solaris vers. 2.6 [21] as behavioral description of the unit. Such a function encodes an input string, based on a one-way encryption algorithm, and is primarily used by the Operating System for user's password encryption.

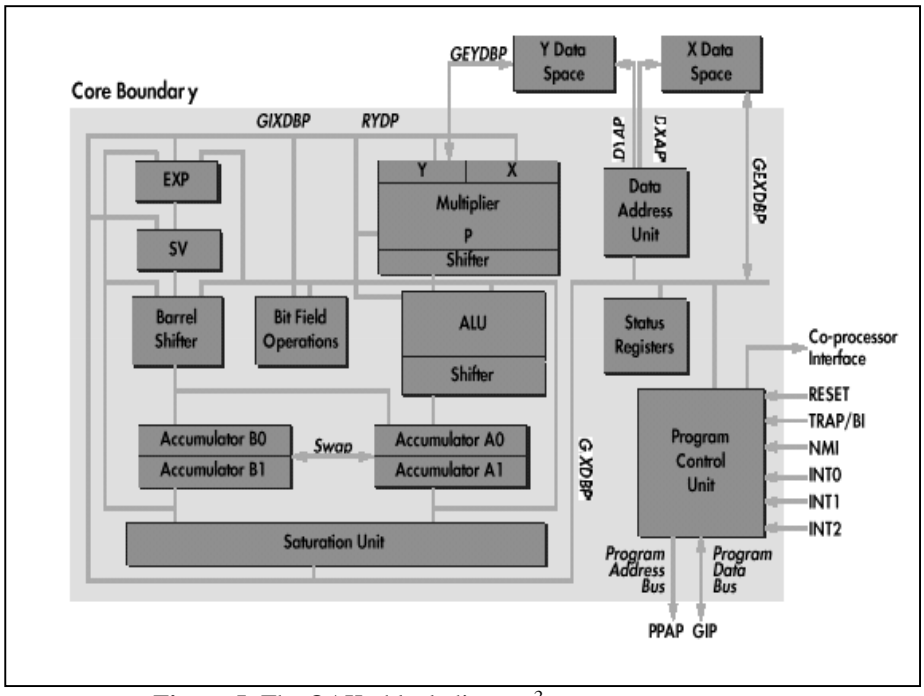

Figure 5: The OAK + block diagram[3]

5.2. Experimental Setup

The employed test synthesis tool GATSBY is implemented in ANSI C and counts up to 3,500 lines of code.

Results were obtained by running GATSBY on a Sparc-Station 5/110 with 64Megabytes of RAM, and limiting the computation time to 24 hours.

The optimal reseeding solution is computed by using an optimization parameter, which minimizes of the number of triplets. This value defines the storage requirements for the testing purpose, but it also contributes in increasing the global test time.

The gate-level ATPG Sunrise [29] is used to compute the fault list and the deterministic test set exploited in the computation process. Moreover, Sunrise provides the target fault coverage to be reached by the reseeding solution.

[3] The scheme is reproduced in the present paper with the authorization of Philips Semiconductor [30]

The value of the genetic parameters has been experimentally tuned: they must drive the GATSBY computation in order to provide an optimal reseeding achieving the target fault coverage. Genetic parameters consist of a population of 16 individuals, let evolving for about 100 generations; at each new evolution process, 20 new individuals are created. Finally, in performing the experiments we assumed that the TPG primary outputs always correspond to the content of the TPG state register.

5.3. Test efficiency

The performed experiments show that a variety of different functional blocks are efficient test pattern generators.

Reseeding results are collected in Table 1 and Table 2. To a better understanding of the results and to allow a comparison among the different TPGs, Table 1 reports the number of triplets for optimal reseedings (N) and Table 2 the corresponding global test length (T). The test length of each triplet τ is not directly taken into account since it is kept constant for all the triplets of the population.

For sake of completeness, Table 1 also reports the target fault coverage computed running Sunrise, achieved by all the reseeding solutions presented below, and Table 2 includes the test lengths of the deterministic test sets pro-vided by Sunrise.

Performed experiments show that in general LFSRs are not superior to any of the other modules investigated, neither in terms of number of reseedings nor with respect to the test length. Therefore, there is no need to include additional LFSRs only for testing purposes.

Focusing first on TPGs including arithmetic functions, most of the ISCAS'85 circuits (c432, c499, c880, c1355, c1908, c3540, c6288) are tested with a single triplet (Table 1). On the remaining examples (excepted c2670), the adder-based TPGs outperform the other arithmetic-based units, requiring the lowest number of triplets. Moreover, depending on the circuits, the multiplier-based TPG sometimes implies less reseedings then the subtracter-based one (s641, s713, s953, s1238), or vice versa (c2670, s838, s1196), while on two circuits the two TPGs are equivalent (s420, s820).

In terms of test length (Table 2), the minimum values are provided by both the adder- (9 cases out of 17) and multiplier-based TPGs (5 cases out of 17), while on three circuits (s641, s713, s1196) the subtracter-based ones give the optimal solutions. Making a comparison normalized

on the number of reseedings, instead, the multiplier-based TPG usually requires the lowest test lengths, with respect to the other arithmetic units (c499, c880, c1908, s641, s713).

LFSR-based TPGs provide test lengths lower than the ones required by arithmetic units in 8 cases out of 17 (c432, c880, c6288, s641, s713, s820, s953, s1423). In terms of number of triplets, instead, LFSRs are worse than the arithmetic units in 4 cases out of 17 (s713, s820, s953, s1423), while they provide solutions equivalent to their highest reseeding numbers for two circuits (s420, s641). In the remaining 11 cases, the reseeding number is in between the optimal and the worst solutions by the arithmetic units.

Finally, using the encryption unit, the results obtained with some circuits are comparable with both arithmetic units and the LFSR but, generally, do not outperform those solutions.

Figure 6 analyses the effectiveness of the test set in terms of number of dummy patterns; the case of s1423 as UUT has been considered. Many vectors of the test set are dummy patterns; however, for all the TPGs, the number of useful patterns included into the test set is comparable with the Sunrise test length. This behavior has been experimentally validated with almost all the considered circuits.

Since the functional BIST approach is an at-speed test, reducing the number of dummy patterns and reducing the overall test length compared to an external scan based test was not a target. Allowing a larger number of dummy patterns and increasing the test length may even improve the defect coverage, and reduce the memory requirements for the seeds while the test application time will still be shorter than the time for external testing.

6. Conclusions

The present paper works inside the context of the functional BIST strategy and investigates different functional units as possible deterministic test generators.

Experiments proved that accumulator-based units including various arithmetic functions, as well as user-defined modules, can be efficiently employed for test pattern generation. According to each context the designer can therefore select the "best" candidate test generator among the units functionally connected to the module under test.

The functional BIST approach is less intrusive then traditional BIST techniques, since no test points and no additional registers are introduced into the modules to be tested. The efficiency of all the functional units investigated so far is as least as high as efficiency of the classical LFSR reseeding technique, and in many cases it is higher.

		GATSBY				
	Target	# Triplets (N)				
		Arithmetic units			LFSR	Encr
Circuit		Adder	Multiplier	Subtractor		
c432	99.12	1	1	1	1	1
c499	98.84	1	1	1	1	1
c880	100	1	1	1	1	1
c1355	99.47	1	1	1	1	1
c1908	99.61	1	1	1	1	1
c2670	95.64	33	36	30	34	43
c3540	96.05	1	1	1	1	2
c6288	99.56	1	1	1	1	1
S420	100	7	10	10	10	13
S641	100	5	6	7	7	7
S713	93.46	5	5	7	8	6
S820	100	3	3	3	35	3
S838	100	11	92	30	44	89
S953	100	3	3	4	5	4
s1196	100	4	6	5	5	6
s1238	94.74	4	6	7	6	7
s1423	98.99	3	4	3	5	3

Table 1: Comparison among TPGs in terms of Reseedings

	Test length (T)					
	Sunrise	GATSBY				
		Arithmetic Units			LFSR	Encryptor
Circuit		Adder	Multiplier	Subtractor		
c432	81	243	260	273	210	316
c499	74	368	354	420	417	472
c880	91	2,104	1,966	2,123	1,829	2,737
c1355	118	1,151	1,241	1,177	1,334	1,314
c1908	184	3,773	3,348	3,641	3,759	3,245
c2670	163	10,179	8,310	10,401	10,206	6,090
c3540	210	3,467	6,106	4,568	4,505	6,078
c6288	57	56	59	56	23	54
s420	110	5,510	9,711	8,851	10,843	9,735
s641	92	4,475	2,730	2,665	2,430	3,179
s713	94	9,082	3,588	3,091	2,759	3,518
s820	176	5,311	5,401	7,485	527	4,785
s838	215	6,694	10,988	17,192	9,273	17,098
s953	114	7,871	5,803	6,572	4,834	8,043
s1196	232	10,000	7,025	6,920	18,776	6,408
s1238	236	7,356	9,071	8,533	7,713	8,183
s1423	93	3,100	4,474	3,439	1,308	3,461

Table 2: Comparison among TPGs in terms of Test Length

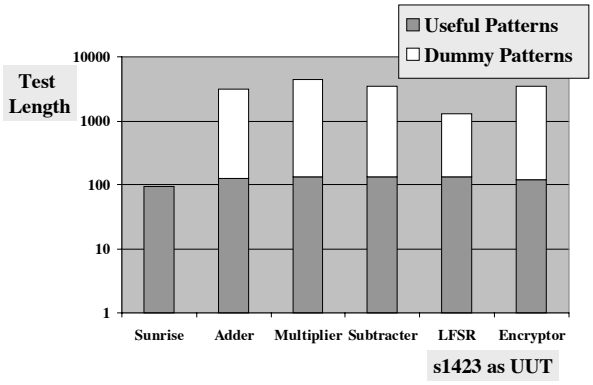

Figure 6: Dummy patterns inside the test set

7. References

[1] B. Koenemann, J. Mucha, G. Zwiehoff, *Built-in Logic Block Observation Techniques*, IEEE International Test Conference, Cherry Hill, NJ, 1979, pp. 37– 41

[2] L. T. Wang, E. J. McCluskey, *Circuits for Pseudo-Exhaustive Test Pattern Generation*, IEEE International Test Conference, Washington, DC, 1986, pp. 25-37

[3] K.-T. Chen, C.-J. Lin, *Timing Driven Test Point Insertion for Full-Scan and Partial-Scan BIST*, IEEE International Test Conference, Washington, DC, 1995, pp. 506-514

[4] 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

[5] B.H. Seiss, P.M. Trousborst, M.H. Schulz, *Test Point Insertion for Scan-Based BIST*, IEEE European Test Conference, 1991, pp. 253-262

[6] H.-J. Wunderlich, *PROTEST: A Tool for Probabilistic Testability Analysis*, IEEE/ACM 22nd Design Automation Conference, Las Vegas, 1985, pp. 204-211

[7] J. A. Waicukauski, E. Lindbloom et al., *WRP: A Method for Generating Weighted Random Patterns*, IBM Journal of Research and Development, Vol. 33, No. 2, March 1989, pp. 149-161

[8] F. Brglez et al., *Hardware-Based Weighted Random Pattern Generation for Boundary-Scan*, IEEE International Test Conference, Washington, DC, 1989, pp. 264-274

[9] S. Hellebrand, S. Tarnick, J. Rajski, B. Courtois, *Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers*, IEEE International Test Conference, Baltimore, MD, September 1992, pp. 120-129

[10] N. A. Touba and E. J. McCluskey, *Altering a Pseudo-Random Bit Sequence for Scan-Based BIST*, IEEE International Test Conference, Washington, DC, 1996, pp. 167-175

[11] H.J Wunderlich, G. Kiefer, *Bit-flipping BIST*, IEEE International Conference on Computer-Aided Design, 1996, pp. 337-343

[12] S. Chiusano, F. Corno, P. Prinetto, M. Sonza Reorda, *Cellular Automata for Deterministic Sequential Test Pattern Generation*, IEEE VLSI Test Symposium, 1997, pp.60-65

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

[14] S. Gupta, J.Rajski, J. Tyszer, *Test Pattern Generation Based on Arithmetic Operations*, IEEE International Conference on Computer-Aided Design 1994

[15] J. Rajski, J. Tyszer, *Accumulator-Based Compaction of Test Responses*, IEEE Transactions on Computers, vol. 42, no. 6, pp. 643-650, June 1993

[16] A. P. Stroele, *Test Response Compaction Using Arithmetic Functions*, VLSI Test Symposium, 1996, pp. 380-386

[17] J. Rajski, J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998

[18] S. Gupta, J. Rajski, J. Tyszer, *Arithmetic Adaptive Generators of Pseudo-Exhaustive Test Patterns*, IEEE Transactions on Computers, 8(45): 939-949, August, 1996

[19] J.Rajski, J.Tyszer, *Multiplicative Window Generators of Pseudo Random Test Vectors*, IEEE European Design & Test Conference, 1996

[20] A. P. Stroele, *Arithmetic Pattern Generation for Built-In Self Test,* IEEE International Conference Computer Design, 1996

[21] Sun Solaris vers. 2.6, *User Reference Manual*, 1997

[22] A. P. Stroele, F. Mayer, *Methods to reduce Test Application Time for Accumulator-Based Self –Test*, IEEE VLSI Test Symposium, 1997, pp. 48-53

[23] R. Dorsch, H.-J. Wunderlich, *Accumulator Based Deterministic BIST*, IEEE International Test Conference, 1999

[24] F. Brglez and H. Fujiwara, *A Neutral Netlist of 10 Combinatorial Benchmark Circuits*, in Proceedings of the IEEE International Symposium on Circuits and Systems, 1985

[25] F. Brglez, D. Bryan, K. Kozminski, *Combinatorial Profiles of Sequential Benchmark Circuits*, IEEE International Symposium on Circuits and Systems, 1989, pp.1129-1234

[26] S. Cataldo, S. Chiusano, P. Prinetto, H-J. Wunderlich, *Optimal Hardware Pattern Generation for Functional BIST*, IEEE Design Automation and Test in Europe (DATE), 2000, pp.292-297

[27] B. Koenemann, *LFSR-Coded Test Patterns for Scan Designs*, IEEE European Test Conference, Munich (1991) pp. 237-242

[28] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989

[29] Sunrise Reference Manual, Sunrise Test Systems, 1995

[30] http://www.semiconductors.com/acrobat/other/technology/embedtech/ oakplus.pdf