An Efficient Seeds Selection Method for LFSR-based Test-per-clock BIST[†]

E. Kalligeros, X. Kavousianos, D. Bakalis and D. Nikolos

Dept. of Computer Engineering & Informatics, Univ. of Patras, 26500, Patras, Greece
Computer Technology Institute, 61 Riga Feraiou Str., 26221, Patras, Greece

e-mail: kalliger@ceid.upatras.gr, kabousia@ceid.upatras.gr, bakalis@cti.gr, nikolosd@cti.gr

Abstract

In this paper we propose a new algorithm for seeds selection in LFSR-based test-per-clock BIST. The proposed algorithm uses the well-known concept of solving systems of linear equations and, based on heuristics, minimizes the number of seeds and test vectors while achieving 100% fault coverage. Experimental results indicate that it compares favorably to the other known techniques.

1. Introduction

Built-In Self-Test (BIST) is an effective approach for testing large and complex circuits [1-2]. When BIST is used, a *Test Pattern Generator* (TPG), a *Test Response Verifier* and a BIST controller accompany the Circuit Under Test (CUT) in the chip, creating this way a self-testable circuit. Minimal test application time and hardware overhead as well as minimal performance degradation are essential in many BIST applications whereas, in most applications, complete (100%) fault coverage is also desirable.

BIST schemes can be classified into two general categories: test-per-scan and test-per-clock. In the test-per-scan scheme a complete or partial scan path is serially filled by the TPG [3-5], while in the test-per-clock scheme a new test vector is applied to the CUT at each clock cycle [5-8]. In this paper we consider only test-per-clock BIST schemes.

Linear Feedback Shift Registers (LFSRs) have been by far the most popular devices for pseudo-random test pattern generation in BIST schemes [2]. They have the advantage of very low hardware overhead. However, for circuits with random pattern resistant faults, high fault coverage cannot be achieved within an acceptable test length. Among others, reseeding is a technique which has been proposed to solve this problem.

[6] describes a design of an LSSD-based LFSR, which is capable of changing seeds by applying a pair of clock pulses at the time of change. The seeds cannot be predetermined, they are randomly selected and they have the property of being uniformly distributed over the entire LFSR pattern space. Mixed-mode BIST schemes [5, 7-8] impose in the pseudorandom sequence deterministic test vectors for detecting the random pattern resistant faults.

The concept of solving systems of linear equations for finding seeds which ensure the embedding of deterministic test vectors in sequences of pseudorandom vectors was presented for test-per-scan BIST in [3-4]. The straightforward application of this technique in test-per-clock BIST schemes does not give good results with respect to the number of seeds and the test sequence length. Using the above concept, in this paper we give a simple but effective algorithm which, based on heuristics, leads to very small number of seeds, short test sequences and complete fault coverage. Experimental results show that the proposed technique compares favorably to the other already known techniques.

The remaining of the paper is organized as follows: Sections 2 and 3 present the LFSR-based reseeding algorithm. In Section 4 the effectiveness of the proposed algorithm is evaluated with experiments on benchmark circuits and comparisons are made with other techniques. Conclusions are given in Section 5.

2. Reseeding algorithm

Consider that the test sequence consists of the parts P_0, P_1, P_2, …, as shown in Figure 1. Each one of these parts consists of a set of successive vectors generated by the LFSR used as TPG. The first vector of P_0 is the initial seed of the test sequence, while the first vector of each one of the other parts (shaded areas in Figure 1) is a new seed. The initial seed is selected randomly, while a method for the selection of the rest of the seeds will be given shortly.

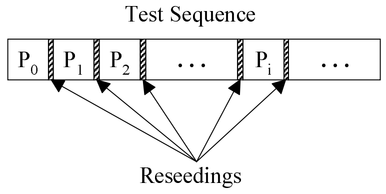

Figure 1. Test sequence

[†] This research was financially supported by the Public Benefit Foundation "Alexander S. Onassis" via its scholarships programs and by the Research Committee of Patras University, within the framework of "K. Karatheodoris" scholarships programs.

At first we perform equivalence and dominance fault collapsing [1] so as to reduce the set of faults to be analyzed by the algorithm and then we proceed to define set P_0. P_0 contains at most *EASYVECTORS* vectors, where *EASYVECTORS* is a user-defined parameter. The value of *EASYVECTORS* must be appropriately selected so as the vectors of set P_0 to detect the majority of the easily detectable, by random testing, faults. The LFSR is initially set to a random state and is let to run for *EASYVECTORS - 1* clock cycles. The produced vectors comprise set P_0 and the faults detected by those vectors are considered as easy-to-detect. If there are any vectors at the end of the P_0 sequence that do not detect any additional faults, we exclude them from P_0.

The faults that remain undetected after the application of set P_0 to the CUT are considered as hard-to-detect. For each one of these faults, e.g. f_i, we choose a representative test vector v_{fi}, which detects that fault. At first we extract Q test vectors for each f_i using a deterministic test pattern generation tool and we modify them to contain don't care bits. We then choose as v_{fi} the one with the maximum number of don't care bits. A representative can be viewed as an indication of how hard the corresponding fault is. The more don't care bits a representative has, the easier it is for the corresponding fault to be detected.

In order to minimize the number of the seeds required to fully test the CUT, we should try to detect, in each part P_i, as many hard-to-detect faults as possible. The proposed algorithm exploits the existence of don't care bits in the test vectors of the undetected faults, so as to select an appropriate set P_i, the vectors of which can detect several hard-to-detect faults.

Let L be the maximum number of vectors that can be included in a part P_i ($i \geq 1$). To determine a set P_i we consider a window of L successive states of the LFSR and we try to locate this window in such a way so as to maximize the number of vectors v_{fi}, v_{fj}, …, that are covered by the states of the LFSR belonging to the window. We choose as last vector of P_i one of the representatives, e.g. v_{fi}, and by moving backwards we try to find, among the L states of the window, the most appropriate to be used as the seed of set P_i. The success of the method depends heavily on the selected representative test vector v_{fi} (the selection procedure will be given in the next section).

Based on the characteristic polynomial of an LFSR and its current state, we can derive all its history, that is, the states through which the LFSR has stepped before reaching the current state. Generally, if we have a k-stage LFSR with external XOR gates that implements the characteristic polynomial $P(x) = x^k + a_{k-1}x^{k-1} + \ldots + a_1x + 1$ with a_1, a_2, ..., $a_{k-1} \in \{0, 1\}$, then:

$$S_i(j-1) = S_{i+1}(j), \quad 1 \leq i \leq k - 1 \text{ and}$$

$$S_k(j-1) = S_1(j) \oplus a_1S_2(j) \oplus a_2S_3(j) \oplus \ldots \oplus a_{k-1}S_k(j),$$

where $S_i(j-1)$ is the value of the i-th stage of the LFSR at the $(j-1)$-th clock cycle. That is, we get the $j-1$ state of the LFSR from its j-th state. Similar equations can be derived if we have an LFSR with internal XOR gates. Hence, knowing the last state of the window we can step by step determine all its previous states.

After the selection of the last vector of P_i, which is the starting state of the procedure, we generate successively the previous L-1 LFSR states, that is the states that the window contains. The don't care bits of the last vector are considered as variables. Since the chosen last state contains variables, some of the bits of the states of the window will be a function of some of those variables too.

The next step is to compare all the representative test vectors we have derived for testing the remaining hard-to-detect faults, with each one of the L states. We note that the last vector of P_i is excluded from these comparisons, since the corresponding fault is already covered. It is evident that in order to check if a test vector v matches a state s_j of the window, we have to solve a system of k equations of the form:

$$X_{t1} \oplus X_{t2} \oplus \ldots \oplus X_{tn} \oplus b_t = c_t,$$

where b_t, $c_t \in \{0, 1\}$ and k is the number of the bits of vector v that have specific binary values. Taking into account that the variables and the constants take binary values and that they are related with the XOR operation only, we conclude that the above procedure is trivial compared to that of solving a conventional system of equations.

Example. Assume that we have a CUT with 4 primary inputs and as a TPG we use the LFSR with characteristic polynomial x^4+x+1.

Consider as last vector of P_i the vector 1xx0. This vector, as far as our procedure is concerned, is equivalent to the state $1X_2X_30$. For $L = 6$ we get a window consisting of the states shown in Figure 2.

State Number	**LFSR state**				
	$S_1(j-1) = S_2(j)$	$S_2(j-1) = S_3(j)$	$S_3(j-1) = S_4(j)$	$S_4(j-1) = S_1(j) \oplus S_2(j)$	
6	1	X_2	X_3	0	***Last State***
5	X_2	X_3	0	$X_2 \oplus 1$	
4	X_3	0	$X_2 \oplus 1$	$X_2 \oplus X_3$	
3	0	$X_2 \oplus 1$	$X_2 \oplus X_3$	X_3	
2	$X_2 \oplus 1$	$X_2 \oplus X_3$	X_3	$X_2 \oplus 1$	
1	$X_2 \oplus X_3$	X_3	$X_2 \oplus 1$	$X_3 \oplus 1$	

Figure 2. Backward state generation

We remind that the states are generated from the last one to the first one and the last state ($1X_2X_30$) is numbered as state L (state 6). If we want to check if test vector 0x01 matches state 1 of Figure 2, we should solve the following system of equations:

$$\{X_2 \oplus X_3 = 0, X_2 \oplus 1 = 0, X_3 \oplus 1 = 1\}$$

From the first equation we have $X_2 = X_3$. We eliminate variable X_2 by replacing it with X_3 in the remaining equations and as a result the second equation becomes $X_3 \oplus 1 = 0$ or equivalently $X_3 = 1$. Using this result in the third equation we get $1 \oplus 1 = 1$, which is a contradiction. Thus, vector 0x01 does not match the LFSR state 1.

On the contrary, if we check 0x01 against state 3, the following system of equations is derived:

$$\{0 = 0, X_2 \oplus X_3 = 0, X_3 = 1\}$$

The above equations do not lead to a contradiction and therefore test vector 0x01 matches state 3. ■

Definition 1. As elimination of a variable X_i we define the replacement of that variable by an expression of the form $X_{j1} \oplus X_{j2} \oplus \dots \oplus X_{jn} \oplus b$, where X_{j1}, X_{j2}, …, X_{jn} are variables different from X_i and $b \in \{0, 1\}$. ■

Definition 2. A vector v matches a state m of the LFSR window when the corresponding system of equations does not lead to any contradictions. ■

If test vector v matches state m of set P_i, this means that P_i has the potential to cover v in its m-th state. In order for vector v to be covered by state m of P_i, a number of variables in the generated LFSR states must be replaced by expressions which include variables and/or constant logic values. The variables that must be replaced are the eliminated variables in the "state m = vector v" system of equations. If this is the case, set P_i must include state m and consequently all states between m and L (Figure 3). The test vectors of the undetected faults are categorized according to the number of don't care bits they contain and are checked against the window states according to their category. This categorization will be clarified in Section 3.

By solving systems of equations like those shown above, we check every test vector against all the L states of the window. If a vector matches a state, then we count the number of variables that must be eliminated in order for the test vector to be covered by that specific state. When we have checked all the test vectors against all the L states, we choose to include in set P_i the test vector which eliminates the fewer variables in one of the above L states, by replacing those variables with the appropriate expressions resulting from the corresponding "state - vector" equations. The argument behind this choice is that by keeping as many variables as possible in the states of the window, we increase the probability of a test vector detecting another undetected fault to match a window state. Therefore, the number of hard-to-detect faults covered by set P_i is maximized.

After having selected the appropriate vector (and hence its corresponding fault) to be covered, we update the L states of the window and we repeat the same process using the test vectors that test the remaining hard-to-detect faults. When no test vector matches any of the L states or all the undetected faults are covered, set P_i is finally defined. As seed we choose the state with the smallest state number (see Figure 3), in which a test vector was included in set P_i (P_i contains all states between that state and state L). We should note that the whole process of checking all the vectors against all the states can be significantly speeded up by stopping solving a "state - vector" system when it eliminates more variables than the, up to that point, best state - vector pair.

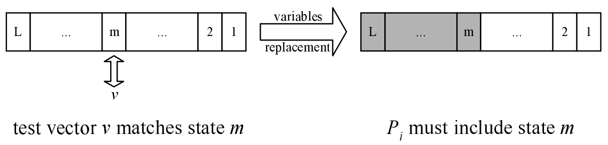
L
...
m
...
2
1
variables
replacement
v
test vector v matches state m
P_i must include state m

Figure 3. Inclusion of vector v in set P_i

Since some of the vectors may detect more than one hard-to-detect faults, after determining a new part P_i of the test sequence, we run fault simulation, so as to drop all faults detected by that part.

By repeating the above procedure we determine parts P_{i+1}, P_{i+2} … until all the faults of the CUT are covered.

Possibly, some of the easy-to-detect faults that were tested by set P_0 can also be detected by some of the vectors of the sets P_1, P_2, ... Thus some of the first vectors of the test sequence may be redundant. In order to minimize the cardinality of the test set, after the determination of all of the P_i parts of the test sequence, we perform reverse simulation [9] and we adjust the initial seed so as to exclude the redundant test vectors.

3. Optimizations of the proposed algorithm

In this section we will clarify the criteria used to determine the values of some critical parameters of the algorithm and specifically the last state of each set P_i and the size L of the window of successive states (L includes the last state of the window). We will also define how the representative test vectors are categorized according to the number of don't care bits they contain. The selection of these parameters must be done very carefully in order to balance the number of the seeds and the total number of test vectors needed to fully test the CUT.

We will at first discuss how the selection of the last state of set P_i is made. Remember that the more don't care bits a representative has, the easier the corresponding fault is detected. There are two scenarios that can be followed. According to the first one, among all the representatives, the one with the maximum number of don't cares (which corresponds to the easiest of the undetected faults) is chosen to be the last state of P_i. Since, following this scenario, we first choose the representatives with the higher number of don't cares, a great number of matches take place at the first parts P_1, P_2, …, P_k, where k is a small number, and a lot of undetected faults are covered in these parts of the sequence. On the contrary, at the last parts of the procedure, representatives with very few don't cares remain. So the number of matches is restricted and the number of faults detected by each part P_i, with i high, is limited. The above imply that excluding the last parts from the resulting test sequence, the number of undetected faults will be small. Therefore, it is expected that this scenario will give better results, in terms of number of seeds, in the cases that complete (100 %) fault coverage is not required.

The second scenario is to select among the representatives, the one with the smallest number of variables as the last state of P_i. The notion behind this choice is that by choosing to cover the hardest of the undetected faults in the beginning, we will avoid detecting them at the end of the test sequence by performing successive reseedings. By starting with the fewest don't cares we do not anticipate covering as many faults as in the first steps of the first scenario. On the contrary we expect the easiest of the undetected faults to be more uniformly distributed over the parts P_i of the test sequence, so as all faults to be detected without having to waste a few reseedings at the end of the test sequence for the hardest of them. The comments concerning the two possible scenarios were verified with experiments. Taking into account that our target is the 100% fault coverage of the CUT, we selected the second one.

An example is given in Figure 4 for the benchmark circuit s420. We can see that the second scenario leads to a more uniform distribution of the faults which are detected by each part P_i, as well as to complete (100%) fault coverage with a smaller number of seeds. On the other hand, if we decide to limit the number of reseedings, following the first scenario and using the parts P_1 to P_6 we achieve 97.62 % fault coverage, while for achieving the same fault coverage with the second scenario we need two more seeds (parts P_1 to P_8).

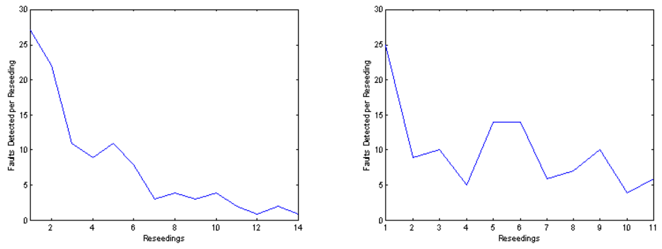

Figure 4. Hard-to-detect faults distribution over the reseedings for the circuit s420 using as last vector of each set P_i: a) the representative with the maximum number of don't care bits and b) the representative with the minimum number of don't care bits (*EASYVECTORS* = 4000, L the same for both experiments)

The determination of parameter L is also critical for the performance of the proposed algorithm. Our goal is to try to cover as many undetected faults as possible in each set P_i of the test sequence ($i > 0$). Apart from keeping as many variables as possible in the states of the window, another technique to achieve this goal is to distribute all the variables in all the stages of the window states. In this way, the bit values of the states of the window will be given by a variety of expressions containing many variables. As a result, it would be easier for the test vectors to match the window states. Therefore, L should be equal to or greater than the number of the primary inputs of the CUT, since, in that case, some of the states of the window are derived after a full-length rotation of the LFSR and thus variables appear in every bit of the window states. We experimentally confirmed the above considerations and we decided that parameter L should be a multiple of the number of the primary inputs of the CUT.

One final adjustment to the proposed algorithm is to classify the representatives v_{fi}, with $f_i \in$ {set of hard-to-detect faults}, to two categories or priority classes. Since the representatives with few don't care bits are the hardest to match a state of the window and if they do so, a lot of variables should be eliminated so as to be included in a set P_i, those representatives are rarely selected by the algorithm. This can cause reseedings that detect very few faults at the end of the test sequence, as it was described above. Let U_{min} be the smallest number of undefined bits that a representative v_{fi} has and U_{avg} the average number of undefined bits of all the representatives. $(U_{avg} - U_{min}) / 2$ is the middle of the distance between U_{min} and U_{avg}. Before we begin defining sets P_i, with $i > 0$, we scan all the representatives and those that the number of their don't care bits lies between U_{min} and $U_{min} + (U_{avg} - U_{min}) / 2$, are marked to be of high priority. All the other representatives have normal priority. Thus, we first check the representatives of high priority against the window states and if some of them match any of the window states, we select the one that eliminates the fewer variables as described in the previous section. If this is not the case, we proceed to those of normal priority.

4. Experimental results

In order to validate the effectiveness of the proposed technique, we implemented the algorithm described in Sections 2 and 3 in C programming language and performed experiments using ISCAS '85 and ISCAS '89 benchmark circuits with random pattern resistant faults. In the case of ISCAS '89, we considered only their combinational part. The primitive polynomials of the LFSRs used in the experiments were taken from [2].

In Table 1 we present results for 2 different values of parameter *EASYVECTORS* and 4 different values of parameter L. In each pair of columns, after the first column of each table, we give the number of seeds and the total number of test vectors required to achieve complete (100%) fault coverage of the CUT. The notation "$L = k$ x" (k = 3, 6, 20, 40) means that the parameter L is equal to k times the number of the primary inputs of the CUT.

For circuits with many primary inputs, increasing the value of L does not improve the performance of the algorithm. For that reason, some results are omitted from Table 1. For each circuit, the best among all the results is shaded. As best result we consider the one with the fewer seeds or the one with the smallest number of test vectors among results with similar numbers of seeds. Each result in Table 1 was taken after a single run of the proposed algorithm, starting from a random initial seed.

In general we expect that, as *EASYVECTORS* and L increase, we will get test sequences requiring less seeds and more test vectors for fully testing the CUT. The results given in Table 1 follow this general trend. Exceptions to this rule are due to the random selection of the initial seed, the reverse simulation process and the fact that for some

Table 1. The results of the proposed technique for *EASYVECTORS* = 3000 and *EASYVECTORS* = 5000

Circuit	*EASYVECTORS* = 3000								*EASYVECTORS* = 5000							
	L = 3x		*L* = 6x		*L* = 20x		*L* = 40x		*L* = 3x		*L* = 6x		*L* = 20x		*L* = 40x	
	seeds	vecs	seeds	vecs	seeds	vecs	seeds	vecs	seeds	vecs	seeds	vecs	seeds	vecs	seeds	vecs
c2670	13	6225	11	11297	-	-	-	-	13	6225	11	11328	-	-	-	-
c7552	26	14178	24	18389	-	-	-	-	25	11261	22	22809	-	-	-	-
s420	16	2687	14	3234	14	3343	12	6636	16	2635	13	4160	11	3450	10	5097
s641	5	2650	4	2701	4	1499	4	2844	5	3010	4	3395	4	2981	4	2364
s713	5	1965	4	2249	5	1472	4	2795	4	2547	4	1820	4	4537	4	3305
s820	6	3033	6	2916	7	3294	6	4373	6	4610	4	4964	6	5919	5	5108
s838	25	2395	23	4497	-	-	-	-	21	5235	21	6646	-	-	-	-
s953	5	3071	6	3466	3	3159	4	3721	4	4340	5	4671	5	5963	4	6164
s1196	9	3432	9	3183	9	4290	7	5745	7	4837	5	5060	8	5037	5	6852
s1238	13	3343	10	3322	8	4498	7	5689	9	4641	7	4934	8	4909	4	6592
s1423	3	2341	3	1457	2	3376	3	2957	3	4068	3	4462	3	3614	3	3619
s5378	4	4222	5	5132	-	-	-	-	4	6579	4	6435	-	-	-	-
s9234	20	13785	-	-	-	-	-	-	20	15813	-	-	-	-	-	-

Table 2. Seeds and test vectors comparisons for complete fault coverage

Circuit	Proposed		Twisted-ring counters [5]			Gauss elimination [7]			LFSR-based TPGs of [8]		
	Seeds	Test vectors	Seeds	Test vectors	Test Vectors reduction	Seeds + polyn.	Test vectors	Test Vectors reduction	Seeds	Test vectors	Test Vectors reduction
c2670	13	6225	70	58930	89.4 %	15	7300	14.7 %	34	10206	39.0 %
c7552	25	11261	107	76447	85.3 %	19	31282	64.0 %	-	-	-
s420	11	3450	8	10816	68.1 %	8	5775	40.3 %	10	10843	68.2 %
s641	4	1499	9	11458	86.9 %	7	2345	36.1 %	7	2430	38.3 %
s713	4	1820	8	11296	83.9 %	6	2069	12.0 %	8	2759	34.0 %
s820	6	2916	-	-	-	6	6036	51.7 %	35	527	-81.9 %
s838	21	5235	29	15742	66.7 %	13	17526	70.1 %	44	9273	43.5 %
s953	3	3159	6	10810	70.8 %	6	7146	55.8 %	5	4834	34.7 %
s1196	5	5060	12	11152	54.6 %	6	7991	36.7 %	5	18776	73.1 %
s1238	4	6592	9	10864	39.3 %	8	8185	19.5 %	6	7713	14.5 %
s1423	3	1457	-	-	-	5	2993	51.3 %	5	1308	-10.2 %
s5378	4	4222	1	10642	60.3 %	6	8400	49.7 %	-	-	-
s9234	20	13785	-	-	-	23	108638	87.3 %	-	-	-

Table 3. Hardware overhead comparisons

Circuit	Number of Primary Inputs	Proposed technique		Twisted-ring counters [5]		Gauss elimination [7]			LFSR-based TPGs of [8]	
		ROM bits (gate equiv.)*	Control logic	ROM bits (gate equiv.)*	Control logic (gate equiv.)	ROM bits (gate equiv.)*	Control logic	P-LFSR (gate equiv.)	ROM bits (gate equiv.)*	Control logic
c2670	233	757	108	4078	65	874	H1	1025	1981	H2
c7552	207	1294	164	5537	65	983	H1	911	-	-
s420	34	94	96	68	42	68	H1	150	85	H2
s641	54	54	59	122	46	95	H1	238	95	H2
s713	54	54	54	108	42	81	H1	238	108	H2
s820	23	36	73	-	-	35	H1	101	201	H2
s838	66	347	119	479	54	215	H1	290	726	H2
s953	45	34	58	68	42	68	H1	198	56	H2
s1196	32	40	73	96	42	48	H1	141	40	H2
s1238	32	32	66	72	42	64	H1	141	48	H2
s1423	91	68	54	-	-	114	H1	400	114	H2
s5378	214	214	68	54	41	321	H1	942	-	-
s9234	247	1235	143	-	-	1420	H1	1087	-	-

*: *We have taken into account the assumption made in [7], that, on average, 0.25 gate equivalents are required for each memory cell of a ROM.*

circuits the seeds are already too few and any further reduction is difficult to be done.

In Tables 2 and 3 we compare the proposed method against the methods presented in [5], [7] and [8]. Among the results given in Table 1 the best, for each circuit, was chosen. We note that a dash (-) in the comparison tables means that no results have been provided by the authors of the referenced paper for the corresponding circuit.

In Table 2 we compare the four techniques with respect to the number of seeds and test vectors they require for fully testing the CUT. For all of them we assume serial loading of the seeds in the LFSR register. However, in the comparisons we do not consider the clock cycles needed for loading the seeds, but we just compare the cardinalities of the applied test sets. We note that the Gauss elimination approach [7] needs a programmable LFSR (P-LFSR), since, along with each seed, it also calculates a new characteristic polynomial and reprograms the LFSR at each reseeding. Thus, in the seventh column of Table 2 we give the sum of the seeds and the characteristic polynomials that need to be stored. From Table 2 we can see that, in the majority of cases, the proposed technique requires less seeds and shorter test sequences than those of [5], [7] and [8].

The hardware overhead comparisons are given in Table 3. Since in all four techniques the seeds are loaded serially, we do not take into account the bit counter required for controlling the loading process. Also, we do not consider the cost of modifying a register to a shift register, since this cost is common for all the techniques we compare. The hardware overhead is given in terms of gate equivalents, assuming that 1 gate equivalent corresponds to a 2-input NAND gate.

The twisted-ring counter-based scheme of [5] imposes very small hardware overhead for the control module but it has to use many ROM bits, especially for circuits with many random pattern resistant faults (c2670, c7552, s838). For these circuits, we can see from Table 3 that the proposed technique is far better than that of [5]. For the rest, the use of our method results in significantly smaller test sequences, with less hardware overhead (except for s420 and s5378). For the realization of the P-LFSRs, the technique of [7] requires, additionally to the logic of the register, two gates and one D flip-flop per LFSR stage. Thus, this technique, although it achieves to significantly reduce the use of ROM, leads to an increased hardware overhead, due to the P-LFSRs, compared to the proposed technique. Also the hardware overhead H1 of the logic that controls the TPG scheme must be added to the overall hardware overhead required by this method. Since not enough information has been given in [7], we were unable to calculate this hardware overhead. Finally, our approach compares favorably to that of [8] with respect to the required ROM (except for the case of s420), but, as in the case of [7], we cannot estimate the cost for the logic that controls the TPG scheme (H2). We expect though, that this logic would be similar or greater than that used by the proposed technique, since in most cases the technique of [8] has to perform more reseedings.

We should finally mention that the execution time of the proposed algorithm is not very long. For the larger circuits we needed only a few hours for each simulation, while for the smaller circuits less than 0.5 hours in an Intel Pentium III, 500 MHz system, excluding the run-time for the ATPG. Although a home ATPG tool was used, it can be easily replaced by any other, more efficient commercial tool.

5. Conclusions

An efficient algorithm for seeds selection in LFSR-based test-per-clock BIST was presented. Experimental results using ISCAS'85 and the combinational part of ISCAS'89 benchmark circuits with random pattern resistant faults show the superiority of the proposed method, with respect to the number of seeds and test vectors as well as the hardware overhead, against the already known methods.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, NY, 1990.

[2] P. H. Bardell, W. H. McAnney and J. Savir, *Built-In Test for VLSI: Pseudo-Random Techniques*, Johh Wiley & Sons, New York, NY, 1987.

[3] B. Koenemann, "LFSR-Coded Test Patterns for Scan Design", Proc. of European Test Conference, Munich, Germany, April 1991, pp. 237-242.

[4] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman and B. Courtois, "Built-In Test for Circuits with Scan Based on Reseeding of Multiple-Polynomial Linear Feedback Shift Registers", *IEEE Trans. on Computers*, vol. 44, no. 2, February 1995, pp. 223-233.

[5] K. Chakrabarty, B. T. Murray and V. Iyengar, "Built-in Test Pattern Generation For High-Performance Circuits Using Twisted-Ring Counters", Proc. of 17th IEEE VLSI Test Symposium, Dana Point, CA, USA, April 1999, pp. 22-27.

[6] J. Savir and W. McAnney, "A Multiple Seed Linear Feedback Shift Register", *IEEE Trans. on Computers*, vol. 41, no. 2, February 1992, pp. 250-252.

[7] L. R. Huang, J. Y. Jou and S. Y. Kuo, "Gauss-Elimination-Based Generation of Multiple Seed-Polynomial Pairs for LFSR", *IEEE Trans. on CAD*, vol. 16, no. 9, September 1997, pp. 1015-1024.

[8] S. Chiusano, P. Prinetto and H. J. Wunderlich, "Non-Intrusive BIST for Systems-on-a-Chip", Proc. of International Test Conference, Atlantic City, NJ, USA, October 2000, pp. 644-651.

[9] A. P. Stroele & F. Mayer, “Methods to Reduce Test Application Time for Accumulator-based Self-Test”, Proc. of 15th IEEE VLSI Test Symposium, Monterey, CA, USA, April-May 1997, pp. 48-53.