Military Technical College
Kobry El-Kobba,
Cairo, Egypt

11-th International Conference
on Aerospace Sciences &
Aviation Technology

A NEW SINGLE TEST PATTERN GENERATOR FOR PSEUDOEXHAUSTIVE TESTING

[1]Mohamed H. El-Mahlawy*, Winston Waller**

ABSTRACT

In this paper, we present a novel test pattern generator for pseudoexhaustive testing. This generator bridges the gap between convolved LFSR/SR and permuted LFSR/SR. It is considered to be the optimal pseudoexhaustive test pattern generator as far as the lengths of test set and hardware overhead are concerning. We present an efficient search to assign the residues for the inputs of the CUT to increase the chance to get several solutions and reduce the hardware overhead. With small number of permutations in the assigned residues, the chance of obtaining efficient results may be increased. The novel generator is considered the general form of the pseudoexhaustive test pattern generator. The simple LFSR/ SR, the permuted LFSR/SR, and convolved LFSR/SR are considered the special case of the novel generator. The experimental results indicate the superiority of this generator and the efficiency of our approach.

Key words: Design for testability of VLSI design, pseudoexhaustive testing.

1. INTRODUCTION

The pseudoexhaustive test retains almost all benefits of an exhaustive test [2-6]. The choice of pseudoexhaustive test technique depends on whether or not any combinational circuit outputs depend on all of the circuit inputs. If any circuit output depends on all of its inputs, a partitioning (or segmentation) test technique must be used to test these circuits [4-5, 17]. For circuits with restricted output dependency, the pseudoexhaustive test techniques provide an alternative test method. The combinational circuit with *n* inputs and *m* outputs is modelled as a direct acyclic graph. The nodes represent gates and the interconnection signals are represented by edges. Each output cone of the circuit forms a subgraph need not be disjoint. The *dependency set*, D_i, of the output cone *i* is considered the set of the primary inputs and the pseudo-primary inputs that feed it directly or affect it through another node. The *dependency,* $|D_i|$, of the output

***Egyptian Armed Forces,**
**** Senior Lecturer, VLSI group, Kent University at Canterbury**

cone i is the cardinality of its dependency set. Let k be the maximum value among the dependencies of the m output cones. The circuit can be characterized as an (n, m, k) circuit. The circuit is segmented into m output cones, and each cone is tested exhaustively. The test ensures detection of all irredundant combinational faults with a single pattern within individual cones of the circuit without fault simulation. The time required for pseudoexhaustive testing depends on the sizes of the output cones. So pseudoexhaustive testing reduces the testing time to a feasible workable value while retaining many of the advantages of exhaustive testing. Many test pattern generators have been proposed for pseudoexhaustive testing. Examples are *syndrome driver counters* (SDCs) [7], *constant-weight counters* (CWCs) [8], *condensed LFSRs* [9], *cyclic LFSRs* [10], *combined LFSR and XOR gates* (LFSR/XORs) [11-12], *combined LFSR and shift register* (simple LFSR/SRs) [13], *permuted LFSR/SRs* [18], *convolved LFSR/SRs* [14], and *modified convolved LFSR/SRs* [6]. It has been found the generators based on the universal test set method require longer test lengths than the generators based on the output-specific test set method. Convolved LFSR/SR is a generator based on the output-specific approach. It is considered to be the optimal pseudoexhaustive test pattern technique as far as the lengths of test set and hardware overhead are concerning. In [14], the size of shift register segment is constrained to have a desired minimum length to reduce the number of feed forward stages. This constraint weakens the potential of using convolved LFSR/SR and in a lot of cases, the number of XOR gates needed for convolved LFSR/SR is high [4, 6]. The algorithm presented in [6] presents search iteration for assigning residues to the inputs of the CUT so as to increase the number of potential solutions and thus reduce the hardware overhead. It reduces the constraint in the size of the shift register segment and makes an efficient search to restrict on the number of feed forward stages into two stages at most and no restriction on the size of the shift register (SR) segment. The residues are assigned such that minimum hardware overhead is achieved. This search generates several possible solutions for each case, from which the minimal hardware solutions may be chosen. Sometimes, the required search time to generate the minimal test set (if $w = k$, where, w is the order of the primitive polynomial) is large, or the search procedure requires test set length of greater than 2^k.

Dimitrios Kagaris and Spyros Tragoudas [18] have suggested a permuted LFSR/SR which is a simple LFSR/SR that drives a permuted set of inputs. A simple LFSR/SR, which is not capable of generating the optimal test set length because of fixed assignment of residues, may become feasible through reassigning the residues. From the results in [18], additional segmentation cells are required to find an applicable primitive polynomial.

This paper introduces an efficient approach to design a generator that generates a pseudoexhaustive test pattern set. It is required to design a generator with minimal test length and minimal hardware overhead. A new generator which bridges the gap between convolved LFSR/SR [4, 6] and permuted LFSR/SR [18] is presented. With small number of permutations in the assigned residues, the chance of obtaining efficient results may be increased. A simple efficient heuristic approach for permutation is introduced. The novel generator is considered the general form of the pseudoexhaustive test pattern generator. The simple LFSR/ SR, the permuted LFSR/SR, and the convolved LFSR/SR are considered the special case of the novel generator. The experimental results indicate the efficiency of this generator. The results also show that the insertion of additional segmentation cells required by the method outlined in [18] is not required.

We will begin with a background survey of the convolved LFSR/ SR as test pattern generator for pseudoexhaustive testing in section 2. Search for residues assignment will be in section 3. The algorithm is given in section 4. The experimental results will be in section 5 and the conclusion in section 6.

2. OVERVIEW OF THE STRUCTURE OF THE NOVEL GENERATOR

Let π_0 be an index default permutation of the *n* inputs of the CUT which signifies the order in which the corresponding flip-flops (stages) of the convolved LFSR/SR are linked. Consider an (*n*, *m*, *k*) CUT along with the notation that input I_i is assigned a unique index (label) *i* where $0 \leq i \leq n$. The default permutation π_0 of the inputs of the CUT is specified completely by *n*-tuple (0, 1, 2, 3,..., *n* - 1), let the set A = $\{I_0, I_1, I_2, ..., I_{n-1}\}$, and $\pi_0 = \{0, 1, 2, 3,..., n - 1\}$.

Fig. 1 is considered more general scheme of the pseudoexhaustive test pattern generator. In a convolved LFSR/SR [4, 6] and using the default permutation π_0, inputs 0 through *i* are assigned to the residues R_0 through R_i, and inputs *i* + 1 through *n* - 1 are assigned to the residues R_{i+j} through R_{i+n-2}. In the novel generator, it is possible to change the default permutation to another permutation π. In Fig. 1 using permutation π (for example), every CUT inputs are assigned to the residues according to permutation π_0 except I_{w+2} assigned R_{w+1}, I_{w+1} assigned R_{w+2}, I_{i+2} assigned R_{i+j}, and I_{i+1} assigned R_{i+j+1}. So, the convolved LFSR/SR is considered a special case of the novel generator. The simple LFSR/SR and the permuted LFSR/SR are considered special cases of the novel generator as well.

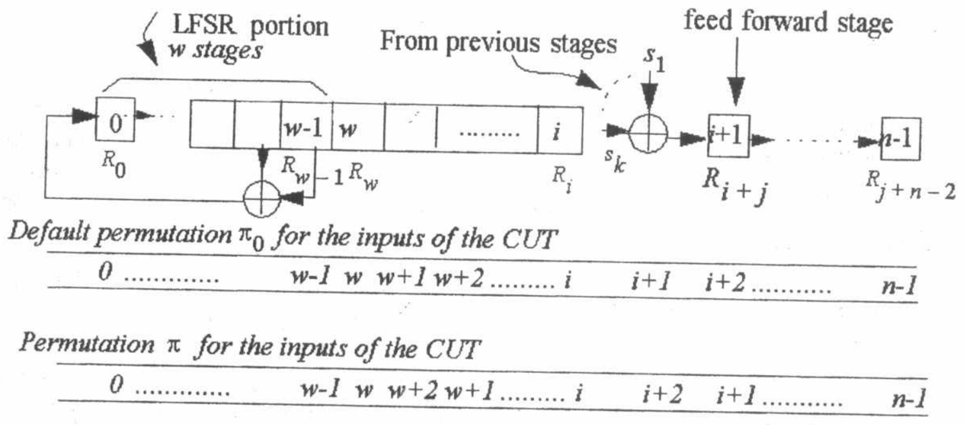
LFSR portion
w stages
From previous stages
feed forward stage
s_1
s_k
0
R_0
w-1
w
i
R_{w-1}
R_w
R_i
i+1
R_{i+j}
n-1
R_{j+n-2}
Default permutation π_0 for the inputs of the CUT
0 .... w-1 w w+1 w+2 .... i i+1 i+2 .... n-1
Permutation π for the inputs of the CUT
0 .... w-1 w w+2 w+1 .... i i+2 i+1 .... n-1

Fig. 1 Novel generator for pseudoexhaustive testing with permutation.

Using a specific primitive polynomial, let β be the set containing all dependency sets of the output cones whose assigned residues are linearly dependent and $|\beta|$ be the number of the dependency sets in β. In the case of the convolved LFSR/SRs, all dependency sets of the output cones must satisfy certain condition that all assigned residues for dependency sets of the output cones must be linearly independent so that β is an empty set. This restriction can be reduced to get a solution even if β is not an empty set. This is done by a small number of permutations of the assigned residues to the inputs of the CUT to make β an empty set. The permutation of the residues means the permutation of the CUT inputs assigned to those residues. The permutation of the CUT inputs may be useful in obtaining a pseudoexhaustive test pattern generator (PETPG) with minimal test set length. The permutation of the inputs results in a routing overhead so we assume that the generator synthesis is done prior to the layout phase of the CUT, and that, therefore any rewiring issues are tackled in the overall layout of the CUT and the BIST circuitry as a whole.

3. THE SEARCH TO OBTAIN MINIMUM $|\beta|$

The search to obtain the minimum $|\beta|$, referred to as $|\beta|_{min}$, is carried out either by searching in the candidate primitive polynomials or by using the proposed residue assignment, discussed in [6] for a specific primitive polynomial. Choosing a small value of $|\beta|_{min}$will reduce the required number of permutations which reduce the routing overhead. First, two useful definitions need to introduce.

Condition 1: For each output cone *i*, all residues R_j, $j \in D_i$, $0 \leq i \leq m$, must be linearly independent.

Definition 1: An *applicable primitive polynomial for permutation* is the polynomial for which at least (m - $|\beta|_{min}$) dependency sets of the output cones satisfy condition 1 before permutation.

Definition 2: The *proper residues for permutation* are produced by the residue assignment for the CUT inputs before permutation such that at least (m - $|\beta|_{min}$) dependency sets of the output cones satisfy condition 1.

The algorithm to design the novel generator (the steps of this algorithm will be discussed in section 5) consists of two phases. The first phase is dedicated to search from the candidate primitive polynomials to find the applicable primitive polynomial for permutation and, with a small number of permutations, the set β may be an empty set. The generator resulting from this phase is the permuted LFSR/SR (the simple LFSR/SR with permutation π). In the second phase, the proper residues for permutation can be found using the proposed residue assignment, discussed in [4, 6] and, with a small number of permutations, the set β may be an empty set. The generator resulting from this phase is the novel generator (convolved LFSR/SR with permutation). A combination of changing the assigned residues with a small number of permutations may increase the chance of obtaining a solution for the case where the convolved LFSR/SR cannot get a solution with minimal test set lengths and minimal hardware overhead (i.e, reduce the required number of XOR gates).

For simplicity, the algorithm divides the length of the generator into two parts. The first part is the LFSR portion with size *w* and the second part is either a simple LFSR/SR with size *n* - *w* (*n*-*w* $\geq$*w*) or shift register with size *n* - *w* (*n*-*w* < *w*). The residues of the second part of the generator can be changed according to the residue assignment, discussed in [4, 6] (refer to the convolved LFSR/SR in form A [4, 6] (Fig. 2)). This division is practically adequate and the experimental results in section 6 will indicate that situation.

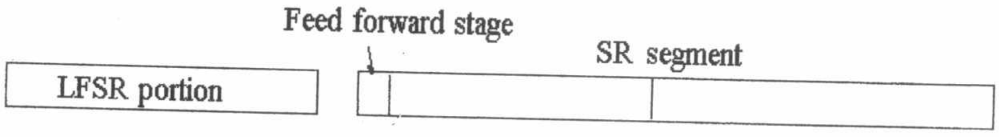

Fig. 2 Convolved LFSR/SR in form A [4, 6].

4. IDEA OF THE PERMUTATION

In this section, the idea of the permutation of one dependency set, whose assigned residues are linearly dependent, will be explained. The permutation approach of all dependency sets, whose assigned residues are linearly dependent, will be explained in section 5.

It is known from condition 1, that all residues assigned to the dependency set must be linearly independent. If the residues assigned to the dependency set are linearly dependent, one or more residues will exist which are linear combinations of the each other. The idea of the permutation will be demonstrated by an example.

Example 1: The (24, 6, 10) CUT with its dependency sets according to following table is considered.

D_0	{0, 1, 3, 4, 8, 9, 10, 13, 16, 22}
D_1	{0, 2, 3, 5, 6, 8, 11, 14, 17, 23}
D_2	{1, 2, 4, 5, 7, 9, 12, 15, 18, 22}
D_3	{0, 1, 2, 6, 7, 10, 11, 12, 19, 23}
D_4	{3, 4, 5, 6, 7, 13, 14, 15, 20, 22}
D_5	{8, 9, 10, 11, 12, 13, 14, 15, 21, 23}

This example consists of two phases. The first phase indicates the solution using the applicable primitive polynomial for permutation. The second phase indicates the solution using the proper residues for permutation for a specific primitive polynomial.

4.1. The first phase: The primitive polynomial with minimum terms, and $|\beta|_{min}$ is determined. This polynomial is $1+x^2+x^7+x^8+x^{10}$, $|\beta|_{min}$ equals 1 so the polynomial, $1+x^2+x^7+x^8+x^{10}$, is an applicable primitive polynomial for permutation. The dependency set whose assigned residues are linearly dependent, *b*, is {0 1 2 6 7 10 11 12 19 23}, let π_0 = {0, 1, 2,...., 23}. First, construct set *b'* = {0, 1, 2} (*b'* $\subset$ *b*) and check if its assigned residues are linearly independent or not. If the residues are linearly independent, add to *b'* the next element which is 6 in *b* (*b'*= *b'*$\cup${6}). The set *b'* is now {0, 1, 2, 6} (*b'* $\subset$*b*). If its assigned residues are linearly independent, add to *b'* the next element. This process is continued until the assigned residue for last added element to *b'* causes linear dependence with other assigned residues of elements in *b'*. The residue assigned to last added element is the first residue that causes the linear dependence of *b*. In this example, this residue is the residue assigned to I_{23} in *b* (input 23) which is R_{23}. Now, it is required to permute this residue with another residue, assigned to input *p* such that $p \in \pi_0$ and $p \notin b$. The first choice is the residue assigned to I_{22}. The residues assigned to set *b* will be linearly independent but this permutation will generate linear dependence for three other dependency sets of the output cones. This permutation is ignored. The residue assigned to I_{21} is then considered but again the residues assigned to set *b* are linearly dependent. The residue assigned to I_{20} is then considered, followed by I_{18} and so on until the residue assigned to I_{13} is permuted by the residue assigned to I_{23} and all residues assigned to all dependency sets of the output cones are linearly independent. The new input assignment of the permuted LFSR/ SR, π, is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, **23**, 14, 15, 16, 17, 18, 19, 20, 21, 22, **13**}. In this case, three XOR gates are required, and the required number of permutations is 1. The permuted LFSR/SR for that CUT with its initial seed and new permutation of the CUT inputs, π, is shown in Fig. 3.

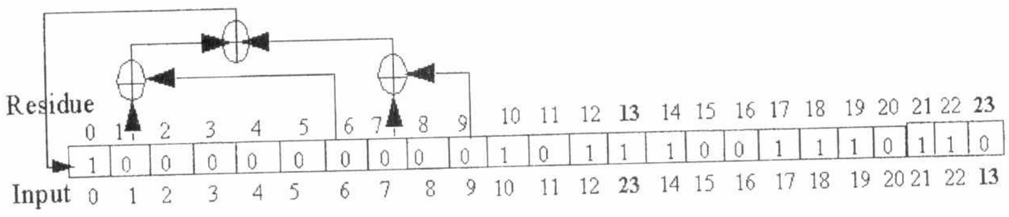

Fig. 3 Permuted LFSR/SR of phase 1.

4.2. The second phase: Set $|\beta|_{\min}$ to 1 and limit the number of terms of the primitive polynomial to 3 so as to use one XOR gate in the LFSR stages. The number of primitive polynomials of degree 10 with one XOR gates is 2 [4]. Unfortunately, $|\beta|$ of these primitive polynomials are greater than one (greater than $|\beta|_{\min}$). So, it is required to divide the generator into two parts, the first part with length 10 which is LFSR portion and the second part with length 14 which is a (14, 10) simple LFSR/SR (convolved LFSR/SR with form A). The residues of the second part are changed to find which equals 1 as in the search using pattern group A in [4, 6]. The solution in this case is the primitive polynomial $1+x^7+x^{10}$, and the residues of the generator are: 0-9, 40-53. The residue assignment, 0-9, 40-53, is the proper residues for permutation. By permuting residue 49 with residue 53, all dependency sets of the output cones satisfy condition 1 (this permutation is carried out as in the first part). R_{49}, assigned to I_{19}, is now assigned to I_{23} and R_{53}, assigned to I_{23}, is now assigned to I_{19}. The number of required XOR gates for this generator is 2 and the required number of permutations is 1. The generator with its initial seed and permutation of the CUT inputs, π, is shown in Fig. 4. The convolved LFSR/SR designed using algorithm [4, 6] requires three XOR gates to test all output cones exhaustively. The novel generator needs just two XOR gates.

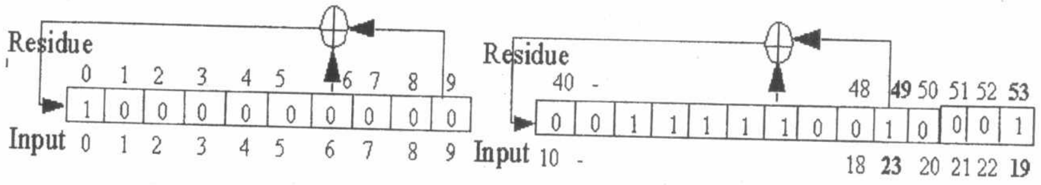

Fig. 4 Novel generator of phase 2.

5. THE PERMUTATION APPROACH AND THE ALGORITHM

Each phase in the novel generator design consists of other two phases. The first one searches for either the applicable primitive polynomial for permutation or the proper residues for permutation. The second is the permutation phase. In this section, the permutation approach for all dependency sets whose assigned residues are linearly dependent is presented.

All dependency sets whose assigned residues are linearly dependent are collected in β. (The number of the sets in β is $|\beta|_{min}$.) The first dependency set in β is chosen, referred to as set *b,* and the first assigned residue causing linear dependence in *b* is determined. Let input *s* be the input whose assigned residue is the first residue in *b* causing linear dependence. Let set *b'* ($b' \subset b$) be the set that covers the elements of *b* from the first element to element *s*. Now, it is required to permute the residue assigned to input *s* with the residue assigned to another input, *p*, such that $p \in \pi_0$ and $p \notin b$. The residues assigned to set *b'*, after permuting the residue assigned to input *p*, are checked for linear independence. In the case that the residues assigned to set *b'* are linearly independent and the residues assigned to set *b* after the previous permutation are linearly dependent, store *p* in buffer *pp* as the first permuted element in set *b*. The search to find other permuted elements in *b* to make the residues assigned to *b* linearly independent is repeated.

For example, if *b* = {0, 2, 3, 6, 7}, and π_0 = {0, 1, 2,...., 9} where these numbers are the indices of the CUT inputs. Let the residues assigned to set *b* be linearly dependent, and the first residue in *b* causing linear dependence be the residue assigned to input 6 (for example). It is required to permute the residue assigned to input 6 (which is input *s*) with another residue assigned to input *p* such that $p \in \pi_0$ and $p \notin b$. If the residue assigned to input 6 is permuted with the residue assigned to input 8 (which is input *p*) and the residues assigned to set *b'*, which are {0, 2, 3, 8}, are linearly independent and the residues assigned to set *b*, which is {0, 2, 3, 8, 7}, are still linearly dependent after permutation, then store *p*, which is input 8, in buffer *pp* as the first permuted element in set *b*. The first residue, causing linear dependence in *b* after the first permutation, is the residue assigned to input 7 (for example) and it is required to permute it with another residue assigned to input *p* such that $p \in \pi_0$ and $p \notin b$ until the residues assigned to set *b* are linearly independent. (Generally, input *p* is considered in the discussion as the permuted element.)

In the case that the residues assigned to *b* are linearly independent, there are three options:

(1) Where the number of linear dependence sets of the dependency sets of the output cones is less than $|\beta|_{min}$. A new β is constructed with a low number of dependency sets such that their assigned residues are linearly dependent and update with the new.

(2) If $|\beta| \geq |\beta|_{min}$ and there are possibilities for other inputs to be permuted, then ignore the last permutation in *b* and try the next possible input.

(3) If $|\beta| \geq |\beta|_{min}$ and there is no possibility for other inputs to be permuted, then ignore all previous permutations applied to *b* and retrieve the stored input from buffer *pp* (first permuted element in set *b*) and select the possible input next to the stored input, let this input be *p*, such that $p \in \pi_0$ and $p \notin b$.

I will state the steps of algorithm *Novel_gen* to design the novel generator.

Algorithm *Novel_gen*

Input: The dependency sets of output cones, w ($\geq k$), $|\beta|_{min}$, Q (number of generated residues), and S (number of required solutions).

Output: Residue assignment for the CUT inputs, the initial seed value, π, and the required number of permutations.

1. Phase 1

1.1 Retrieve all primitive polynomials or the subset of all primitive polynomials of degree *w* from a stored file and put them in queue *L*, set buffer *pp* to -1.

1.2 If *L* is not empty, then select the primitive polynomial, *p(x)*, from *L*. If *L* is empty, then go to phase 2.

1.3 Residues from R_0 through R_{n-1} are generated, based on *p(x)* [4, 6].

1.4 Condition 1 is checked for all dependency sets of the output cones.

1.5 If condition 1 is satisfied for at least (m - $|\beta|_{min}$) dependency sets of the output cones, then the corresponding primitive polynomial is an applicable primitive polynomial for permutation and set β is constructed. If *p(x)* is not an applicable primitive polynomial for permutation, go to step 1.2.

1.6 Take the first set in β, referred to as *b*, and set buffer *pp* to -1.

1.7 Determine the first residue (assigned to input *s*) in *b* causing linear dependence. Let set *b'* be a subset of *b* containing all elements in *b* from the first element up to input *s*. The residue assigned to input *s* is permuted with the residue assigned to input *p* such that $p \in \pi_0$ and $p \notin b$.

1.8 If the residues assigned to *b'* are linearly dependent, there are two options:

- **1.8.1** If buffer *pp* equals -1, try next possible input to be permuted. If there is no possibility for other inputs to be permuted, then go to step 1.2.
- **1.8.2** If buffer *pp* does not equal -1, try next possible input to be permuted. If there is no possibility for other inputs to be permuted, then ignore all previous permutations applied to *b* and retrieve the stored input from buffer *pp* (first permuted element in set *b*) and select the possible input next to the stored input, let this input be *p*, such that $p \in \pi_0$ and $p \notin b$, construct *b'*, set

the buffer *pp* to -1 and go to step 1.8. If there is no possible input next to the stored input as a permuted element, then go to step 1.2.

1.9 If the residues assigned to *b'* are linearly independent and the residues assigned to *b* are linearly dependent after the permutation, then store *p* in the buffer *pp* when *p* is the first permuted element (input) in *b* then go to step 1.7.

1.10 If the residues assigned to *b* are linearly independent and the number of linearly dependent sets of the dependency sets of the output cones is less than $|\beta|_{min}$ and do not equal zero, then construct the new β, update $|\beta|_{min}$ with the new β, and go to step 1.6.

1.11 If the residues assigned to *b* are linearly independent, and the number of linearly dependent sets of the dependency sets of the output cones is greater than or equal to $|\beta|_{min}$ and there are possibilities for other inputs to be permuted, then ignore the last permutation in *b*, try next possible input, construct *b'*, and go to step 1.8.

1.12 If the residues assigned to *b* are linearly independent, and the number of linearly dependent sets of the dependency sets of the output cones is greater than or equal to $|\beta|_{min}$ and there is no possibility for other inputs to be permuted, then ignore all previous permutations applied to *b* and retrieve the stored input from buffer *pp* (first permuted element in set *b*) and select the possible input next to the stored input, let this input be *p*, such that $p \in \pi_0$ and $p \notin b$, construct *b'*, set buffer *pp* to -1 and go to step 1.8. If there is no possible input next to the stored input as a permuted element (or buffer *pp* equals -1), then go to step 1.2.

1.13. If β is an empty set, print the residue assignment for the inputs, π, and store it as a solution and go to the subroutine that determines the initial seed of the generator [4, 6].

1.14 If the number of solutions is less than *S* and there are other primitive polynomials in *L*, select the next primitive polynomial in *L* and go to step 1.3. If the number of solutions is equal to *S*, exit.

1.15 If *L* is empty and no applicable primitive polynomial for permutation exists, go to phase 2.

2. Phase 2

2.1 Select a primitive polynomial with minimum terms from *L*, and generate all required residues (residue 0 through *Q* - 1) [4, 6].

2.2 Using the residue assignment [4, 6], find the proper residues for permutation.

2.3 Construct set β.

2.4 Repeat steps 1.6 through 1.13. (In step 1.8 and step 1.12 of phase 1, there is possibility to branch to step 1.2 but this step in phase 2 will branch to 2.1 when *L* is not empty)

2.5 If the number of solutions equals *S*, exit.

2.6 If the complete set of the generated patterns [4, 6] is finished and *L* is not empty, go to step 2.1.

2.7 If the complete set of the generated patterns [4, 6] is finished and *L* is empty, exit.

6. EXPERIMENTAL RESULTS OF THE NOVEL GENERATOR

Using algorithm *Novel_gen*, novel generators were designed for all combinational benchmark circuits in [1], after they had been segmented using the new algorithm presented in [5]. Table 1 presents the design of the novel generators for the segmented benchmark circuits with a cone size reduction, *l*, of 16, 20, 24, and 28 inputs. (After segmenting the circuit with the segmentation cells, the resulting *k* for the segmented circuits may be less than *l*. The difference between *l* and *k*, obtained after segmenting, is due to the nature of the circuit.) The first two columns provide the characteristics of the segmented circuits. The exponent terms for the primitive polynomial, *p(x)*, is given in the third column. The residue assignment for the stages of the generator, the total number of required XOR gates to realize the generator, and the run-time in seconds on a SUN Sparc II workstation are shown in the fourth and fifth column, respectively. The required number of permutations is given in the sixth column. For example, for the segmented c432 circuit in the first row, the primitive polynomial, *p(x)*, is $1+x^3+x^4+x^5+x^{16}$. Stages 0 through 62 have residues R_0 through R_{62}, respectively. The total number of XOR gates required to realize the generator is 3 in < 1 second with 4 input permutations. Referring to the corresponding design in [4, 6], the required number of XOR gates was 6 in 2 seconds.

The number of permutations in the last column, referred to as *num_per*, is calculated as follows. Let π_0 be {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, and π be {1, 0, 2, 3, 4, 9, 6, 7, 8, 5}. Then, α is {1, 1, 0, 0, 0, 1, 0, 0, 0, 1}. The elements in α are either 0 or 1. If element *i* of π_0 equals element *i* of π, then element *i* of α = 0, if the elements are not equal then element *i* of α = 1. The number of the permutations in π equals the number of 1's in α divided by 2 which equals 2 in this example.

From Table 1, note the following:

1. The test set lengths designed for all combinational benchmark circuits in Table 1 are optimal (minimal) test set lengths (the degree of *p(x)*, *w*, equals *k*) without requiring the insertion of segmentation cells. From the results in [18], additional segmentation cells are required to find an applicable primitive polynomial.

2. Comparing the number of XOR gates and the required run-time between the convolved LFSR/SR using *COV_G* in [4] and the novel generator in Table 1 indicates the efficiency of the novel generator where the required hardware overhead and the run-time are always less than that produced in [4]. In every case, there are several possible solutions which increase the chance of obtaining a result with minimal hardware overhead.

3. The number of required permutations of the CUT inputs is small compared to the length of the generator, *n*.

4. The novel generator has found solutions in cases where the convolved LFSR/SRs [4] could not find a solution in reasonable time. In c5315 and c7552, the algorithm in [4] does not provide any solution when *k* = 16 for any primitive polynomial with degree 16 (the algorithm in [4] found a solution when *w* = 17) but the novel generator succeeds with low hardware overhead. All results in Table 1 prove the efficiency of the novel generator where a solution in all cases is achieved. Table 1 demonstrates the potential of the novel generator approach, generating optimal (minimal) test set lengths with low hardware overhead for the segmented combinational benchmark circuits [5].

7. CONCLUSION

The novel pseudoexhaustive test pattern generator, that bridges the gap between the convolved LFSR/SR and the permuted LFSR/SR, has been introduced and demonstrated. An efficient heuristic approach has been introduced. It permutes the CUT inputs after the proper residues for permutation become available. The changing residues assigned to the CUT inputs followed by permutation gives the novel generator a special character in that the convolved LFSR/SR and the permuted LFSR/SR are special cases of the novel generator. The approach is successful because: (1) The test set lengths designed for all combinational benchmark circuits in Table 1 are optimal (minimal) test set lengths without requiring additional segmentation cells. (2) The required hardware overhead and the run- time are always less than that produced by the convolved LFSR/SR using *COV_G*. (3) The number of required permutations of the CUT inputs is small compared to the length of the generator, *n*. (4) The novel generator has found solutions for those cases where the convolved LFSR/SRs using *COV_G* could not find a solution (as in c5315 and c7552, the algorithm *COV_G* does not provide any solution when $k = w = 16$ (*COV_G* found a solution when $w = 17$)). Finally, the practical results indicate that a generator with minimal test set length, minimal hardware overhead, minimum routing overhead, and good performance can be obtained.

Table 1: Results of the novel generator for segmented benchmark circuits

	COV_G				TPG				
Ckt	*(n,m',k)*	*p(x)*	*Residue assignment*	*XOR / time*	*Residue assignment*	*XOR /time*	*Residue assignment*	*XOR / time*	*num _per*
c432	(63, 19, 16)	16 5 4 3 0	0-27 64-98	6* / 2 sec.	0-40 48-67 69-70	14 / < 1 sec.	0-62	3 / < 1 sec.	4
		16 6 4 1 0	0-34 64-91	6* / 4 sec.	0-40 44-61 63-66	14 / < 1 sec.	0-62	3 / < 1 sec.	4
		16 15 12 10 0	0-33 88-116	6* / 4 sec.	0-36 40-55 61-70	12 / < 1 sec.	0-62	3 / < 1 sec.	2
		16 15 13 4 0	0-29 52-84	6* / 3 sec.	0-41 50-68 70-71	10 / < 1 sec.	0-62	3 / < 1 sec.	4
	(55, 26, 20)	20 19 4 3 0	0-19 21-31 116-139	9* / 4 sec.	0-20 23-55 58	17 / < 1 sec.	0-19 27-61	6 / < 1 sec.	4
		20 3 0	0-33 216-236	2* / 4 sec.	0-37 243-258	7 / < 1 sec.	0 - 19 151 - 185	2 / 1 sec.	6
		20 9 5 1 0	0-19 64-98	6* / < 1 sec.	0-36 110-127	11 / 1 sec.	0 - 19 21 - 55	6 / < 1 sec.	2
		20 17 9 7 0	0-19 27-61	6* / < 1 sec.	0-46 59-66	12 / < 1 sec.	0-54	3 / < 1 sec.	1
	(51, 22, 24)	24 4 3 1 0	0-25 214-238	6* / 1 sec.	0-33 135-151	12 / < 1 sec.	0 - 23 208 - 234	6 / 1 sec.	2
	(47, 18, 28)	28 3 0	0-27 177-195	4* / < 1 sec.	0-29 380-396	10 / 1 sec.	0 - 27 53 - 71	2 / 12 sec.	6.5
	(49, 40, 14)	14 9 8 3 0	0-13 22-40 46-61	9* / 16 sec	0-23 55-69 81-90	13 / 2 sec	0 - 13 43 - 77	6 / 1 sec.	2
c499		14 13 11 4 0	0-13 18-36 169-184	9* / 21 sec	0-21 69-83 139-150	12 / 25 sec	0 - 13 156 - 190	6 / 3 sec.	2
c1355		14 11 7 1 0	0-13 15-32 91-107	9* / 12 sec	0-22 26-41 87-96	12 / 1 sec	0 - 13 24 - 58	6 / 2 sec.	2
		14 13 3 2 0	0-13 18-32 57-76	9* / 10 sec	0-16 23-39 72-86	9* / 24 sec	0 - 13 20 - 54	6 / 1 sec.	3
	(48, 39, 22)	22 11 2 1 0	0-21 185-192 264-281	17* / 646 sec.	no results after two days run	-	0 - 48	3 / 1 sec.	3.5
		22 15 12 9 0	0-21 82230-82255	6*/ 1359 sec.	no results after two days run	-	0 - 21 12689 - 12714	6 / 9193 sec.	1
c880	(71, 28, 16)	16 5 4 3 0	0-15 49-82 135-155	9* / 287 sec.	0-15 17-39 47-64 124-137	17 / 1827 sec.	0 - 15 62 - 116	6 / 3 sec.	4
		16 6 4 1 0	0-15 46-76 428-451	9* / 219 sec.	0-32 51-76 393-404	16 / 2 sec.	0 - 15 46 - 100	6 / < 1 sec.	5
		16 15 12 10 0	0-45 216-240	6* / 14 sec.	0-46 70-92 95	11 / 1 sec.	0 - 15 27 - 81	6 / < 1 sec.	3
		16 15 13 4 0	0-15 23-54 179-201	9* / 95 sec.	0-38 50-69 209-220	10 / 956 sec.	0 - 70	3 / < 1 sec.	5.5
	(70, 28, 17)	17 16 3 2 0	0-16 37-70 268-286	9* / 254 sec.	0-34 51-76 100-108	11 / < 1 sec.	0 - 16 42 - 94	6 / < 1 sec.	5.5
		17 3 0	0-16 110-143 214-232	3* / 569 sec.	0-23 117-151 397-407	10 / 3 sec.	0 - 16 97 - 149	2 / 9 sec.	6
		17 5 0	0-16 86-121 214-230	3* / 395 sec.	0-26 90-123 186-194	9 / 1 sec.	0 - 16 54 - 106	2 / 7 sec.	7
		17 6 0	0-16 91-120 180-202	3* / 442 sec.	0-25 89-122 197-206	10 / 1 sec.	0 - 16 71 - 123	2 / 4 sec.	7.5
	(69, 28, 24)	24 4 3 1 0	0-23 29-48 59-83	9* / 36 sec.	0-23 29-68 130-134	17 / 1 sec.	0 - 23 25 - 69	6 / < 1 sec.	5.5
	(68, 24, 28)	28 3 0	0-27 343-382	2* / < 1 sec.	0-27 308-338 410-418	14 / 28 sec.	0 - 27 146 - 185	2 / 9 sec.	3
c1908	(50, 27, 16)	16 5 4 3 0	0-15 220-235 446-463	9* / 594 sec.	0-18 326-341 364-378	11 / 1326 sec.	0 - 15 508 - 541	6 / 108 sec.	4
		16 11 3 2 0	0-15 170-187 231-246	9* / 214 sec.	0-17 149-164 219-234	9* / 831 sec.	0 - 15 570 - 603	6 / 295 sec.	6.5
		16 11 6 5 0	0-15 96-111 202-219	9* / 123 sec.	0-15 96-111 202-219	9* / 1839 sec.	0 - 15 6637 - 6670	6 / 2635 sec.	5.5
	(44, 27, 20)	20 17 0	0-19 316-330 449-457	15* / 613 sec.	0-19 316-330 449-457	15* / 5832 sec.	0 - 19 2507 - 2530	2 / 1923 sec.	2.5
		20 6 5 3 0	0-19 345-355 437-449	17* / 906 sec.	0-19 263-273 291-301 315-316	20 / 16072 sec.	0 - 19 714 - 737	6 / 400 sec.	6
		20 9 5 1 0	0-19 48-57 224-237	15* / 100 sec.	0-19 29-39 312-321 441-443	16 / 4583 sec.	0 - 19 1668 - 1691	6 / 1159 sec	3
		20 10 5 1 0	0-19 167-177 364-376	14* / 439 sec.	0-19 100-110 472-482 492-493	19 / 37994 sec.	0 - 19 107 - 130	6 / 65 sec.	3.5
	(42, 28, 21)	21 5 2 1 0	0-20 183-192 466-476	14* / 520 sec.	0-21 168-177 280-289	15 / 21436 sec.	0 - 20 566 - 586	6 / 121 sec.	1
	(40, 11, 28)	28 3 0	0-27 58 332-342	7* / 3 sec.	0-27 337-347 362	9 / 3 sec.	0 - 27 471 - 482	3 / 34 sec.	9.5
c2670	(265, 42, 16)	16 5 4 3 0	0-15 18-217 390-438	9* / 177 sec.	0-220 223-243 257-274 279-283	17 / 1 sec.	0-264	3 / < 1 sec.	3
		16 6 4 1 0	0-215 344-392	6* / 56 sec.	0-237 272-289 306-314	13 / 2 sec.	0-264	3 / < 1 sec.	9
		16 15 12 10 0	0-15 17-210 311-365	9* / 151 sec.	0-238 340-357 359-366	14 / 8 sec.	0 - 15 17 - 265	6 / 341 sec.	5
	(262, 40, 20)	20 9 5 1 0	0-80 88-268	6* / 46 sec.	0-209 211-260 264-265	16 / 1 sec.	0 - 261	3 / < 1 sec.	2
		20 15 5 4 0	0-203 207-264	6* / 171 sec.	0-242 266-284	16 / 2 sec.	0 - 261	3 / < 1 sec.	1
		20 15 7 4 0	0-87 115-288	6* / 57 sec.	0-238 261-282 285	13 / 2 sec.	0 - 261	3 / < 1 sec.	4
		20 15 8 7 0	0-205 207-262	6* / 164 sec.	0-242 249-267	10 / 1 sec.	0 - 261	3 / 1 sec.	3
		20 15 9 2 0	0-206 232-286	6* / 167 sec.	0-246 253-267	13 / < 1 sec.	0 - 261	3 / 1 sec.	3
	(256, 33, 24)	24 4 3 1 0	0-213 404-445	6* / 240 sec.	0-223 240-266 270-274	21 / 2 sec.	0 - 255	3 / < 1 sec.	4.5

	(254, 31, 27)	27 8 7 1 0	0-26 29-210 240-284	9* / 830 sec.	0-204 214-242 329-348	16 / 12 sec.	0 - 26 29 - 255	6 / 1159 sec.	7
c3540	(128, 61, 16) (107, 50, 20) (94, 40, 24) (82, 31, 28)	16 11 9 7 0 16 12 6 1 0 20 19 4 3 0 24 4 3 1 0 28 3 0	0-15 126-217 463-482 0-15 152-176 243-329 0-19 3908-3994 0-23 83-106 135-180 0-27 577-601 2556-2584	9* / 944 sec. 9* / 812 sec. 6* / 9 sec. 9* / 76 sec. 8* / 19396 sec.	0-24 26-41 43-66 76-112 310-332 412-414 0-23 28-77 81-103 106-132 192-195 0-22 48-71 86-117 191-212 221-226 0-24 82-106 109-133 138-156 0-27 739-766 1319-1344	19 / 811 sec. 19 / 5 sec. 18 / 8 sec. 18 / 1 sec. 12 / 100 sec.	0-127 0-127 0-106 0-23 30-99 0-27 310-363	3 / 2 sec. 3 / 1 sec. 3 / 20 sec. 6 / 56 sec. 2 / 1273 sec	6 6.5 9.5 11 14.5
c5315	(215, 76, 20) (204, 79, 24) (194, 68, 28)	20 6 5 3 0 20 9 5 1 0 20 19 4 3 0 24 8 5 2 0 28 6 4 1 0	0-19 156-282 424-491 0-19 103-268 300-328 0-146 12488-12555 0-23 26-141 182-245 0-142 448-498	9* / 9399 sec. 9* / 7238 sec. 6* / 10612 sec. 9* / 433 sec. 6* / 108 sec.	0-146 148-169 212-235 281-300 357-358 0-171 199-218 248-268 370 371 0-147 150-176 234-256 279-295 0-160 163-189 324-339 0-142 175-207 401-418	22 / 63 sec. 19 / 2376 sec. 18 / 17 sec. 19 / 245 sec. 15 / 823 sec.	0 - 15 47 - 258 0 - 19 27 - 221 0 - 19 22 - 216 0 - 23 25 - 204 0 - 193	6 / 12087 sec. 6 / 1634 sec. 6 / 576 sec. 6 / 177 sec. 3 / 37 sec.	13.5 11 7.5 5.5 14
c6288	(118, 36,16) (87, 25, 20) (65, 25, 24) (47, 8, 28)	16 10 9 6 0 20 6 5 3 0 24 4 3 1 0 28 3 0	0-15 210-241 437-506 0-19 669-705 917-946 0-23 96-112 1619-1642 0-27 1078-1096	9* / 270 sec. 9* / 1111 sec. 10* / 198 sec. 6 / 2 sec.	0-22 416-433 447-466 531-549 560-578 580-598 0-19 233-253 296-330 342-352 0-23 96-118 752-769 0-27 428-446	18 / 61 sec. 16 / 7 sec. 19 / 20 sec. 9 / 2 sec.	0 - 117 0 - 19 57 - 123 0 - 23 94 - 134 0 - 27 526 - 544	3 / 3 sec. 6 / 44 sec. 6 / 5 sec. 4 / 3 sec.	11.5 7.5 4.5 3
c7552	(274, 45, 20) (261, 34, 24) (255, 38, 28)	20 19 4 3 0 20 6 4 1 0 24 7 2 1 0 28 9 5 1 0	0-79 12609-12802 0-79 987-1180 0-23 163-399 0-27 127-353	6* / 2383 sec. 6* / 61 sec. 6* / 1 sec. 6* / < 1 sec.	0-172 180-209 213-260 295-315 318-319 0-167 169-189 224-281 322-341 377-383 0-177 184-214 221-260 264-275 0-202 211-240 259-280	21 / 4 sec. 21 / 9 sec. 21 / 4 sec. 15 / 5 sec.	0 - 304 0 - 273 0 - 260 0 - 254	3 / 8 sec. 3 / 4 sec. 3 / 8 sec. 3 / 7 sec.	21.5 6 4 4

REFERENCES

[1] F. Brglez and H. Fujiwara, "A neutral netlist on ten combinational benchmark circuits and a target translator in FORTRAN," *International Symposium on circuits and systems*, June, 1985.

[2] E. J. McCluskey and S. Bozorgui-Nesbat. "Design for autonomous test," *IEEE transaction on computers* vol. C-30, pp. 866-875, Nov. 1981.

[3] E. J. McCluskey, "Verification testing-A pseudoexhaustive test technique," *IEEE transaction on computers* vol. C-33, pp. 541-546, June 1984.

[4] Mohamed H. El-Mahlawy, *Pseudo-Exhaustive Built-In Self-Test for Boundary Scan*, Ph.D. thesis, Kent University, U.K., 2000.

[5] Mohamed H. El-Mahlawy, Winston Waller, "A New Segmentation Approach for Pseudoexhaustive Testing of Combinational Circuits." *4th International Conference in Electrical Engineering*, Military Technical College, Egypt, Nov. 2004.

[6] Mohamed H. El-Mahlawy, Winston Waller, "An efficient algorithm to design convolved LFSR/SR." *IEEE 17th National Radio Science Conference*, Minufiya, Egypt, pp. C23 (1-10), Feb. 2000.

[7] Zeev Barzilai, Jacob Savir, George Markowsky, and Merlin G. Smith, "The weighted syndrome sums approach to VLSI testing," IEEE Transactions on Computers, VOL. C-30, NO. 12, Dec. 1981.

[8] D. T. Tang and L. S. Woo, "Exhaustive test pattern generation with constant weight vectors," IEEE transaction on computers vol. C-32, pp. 1145-1150, Dec. 1983.

[9] L. -T. Wang and E. J. McCluskey, "Condensed linear feedfack shift register (LFSR) testing - A pseudoexhaustive test technique," IEEE transaction on computers vol. C-35, pp. 367-370, Apr. 1986.

[10] L. -T. Wang and E. J. McCluskey, "Circuits for pseudoexhaustive test pattern generation," IEEE transaction on computer-aided design vol. 7, pp. 1068- 1080 Oct. 1988.

[11] S. B. Akers, "On the use of linear sums in exhaustive testing," Digest of papers, 15th Annual International on Fault Tolerant Computing Symposium, pp. 148-153, 1985.

[12] N. Vasanthavada, P. N. Marinos, "An operationally efficient scheme for exhaustive test-pattern generation using linear codes," Proc. International Test Conference, pp. 476-482, Nov. 1985.

[13] Zeev Barzilai, Don Coppersmith, and Arnold L. Rosenberg, "Exhaustive generation of bit patterns with applications to VLSI self-testing," IEEE Transactions on Computers, VOL. C-32, NO. 2, Feb. 1983.

[14] Srinivasan, R., S. K. Gupta, and M. A. Breuer, "Novel test pattern generators for pseudoexhaustive testing," Proc. International Test Conference, pp. 1041-1050, 1993.

[15] W. Wesley Petrson, E. J. Weldon., "Error-correcting codes," second edition, 1972.

[16] Paul H. Bardell, Willian H. McAnney, Jacob Savir, "Built-In test for VLSI: pseudorandom techniques," John Wiley and Sons, 1987.

[17] Mohamed H. El-Mahlawy, Winston Waller, "A New Segmentation Approach for Pseudoexhaustive Testing of Combinational Circuits." To be appeared in the 4th International Conference of the Electrical Engineering, Military Technical College, Egypt, pp. 251-265, Nov. 2004.

[18] Dimitrios Kagaris and Spyros Tragoudas, "Cost-effective LFSR synthesis for optimal pseudoexhaustive BIST test sets," IEEE transactions on very large scale integration systems, Vol. 1, NO. 4, pp. 526-536, Dec. 1993.