A Survey of Recursive Pseudo-Exhaustive Two-Pattern Generation Using BIST

B.Kamalasoundari M.E.,
Assistant Professor, PSNA College of Engineering Technology, Tamilnadu, India
kamalasoundari@gmail.com

***Abstract*—Pseudo-exhaustive pattern generators for built-in self-test (BIST) provide high fault coverage of detectable combinational faults with much fewer test vectors than exhaustive generation. In (n,k)-adjacent bit pseudo-exhaustive test sets, all 2^k binary combinations appear to all adjacent k-bit groups of inputs. With recursive pseudoexhaustive generation, all (n,k)-adjacent bit pseudoexhaustive tests are generated for $k \leq n$ and more than one modules can be pseudo-exhaustively tested in parallel. In order to detect sequential (e.g., stuck-open) faults that occur into current CMOS circuits, two-pattern tests are exercised. Also, delay testing, commonly used to assure correct circuit operation at clock speed requires two-pattern tests. In this paper a pseudoexhaustive two-pattern generator is presented, that recursively generates all two-pattern (n,k)-adjacent bit pseudoexhaustive tests for all $k \leq n$. To the best of our knowledge, this is the first time in the open literature that the subject of recursive pseudoexhaustive two-pattern testing is being dealt with. A software-based implementation with no hardware overhead is also presented.**

***Index Terms*—Built-in self-test (BIST), pseudoexchaustive two-pattern testing, test pattern generation.**

I. Introduction

In current IC technology, highly complex chips have low accessibility of internal nodes; this makes traditional testing techniques costly and ineffective. Built-in self-test (BIST) schemes have been proposed as a powerful alternative to external testing. BIST techniques employ on-chip test generation and response verification; therefore the need for expensive external testing equipment is reduced. Furthermore, with BIST at-speed testing can be achieved; thus, the quality of the delivered ICs is increased [1].

Exhaustive and pseudoexhaustive test generators provide for complete fault coverage without the need for fault simulation or deterministic test pattern generation. Numerous publications address the problem of pseudoexhaustive testing as an alternative to competing schemes, e.g., exhaustive or pseudorandom testing. Srinivasal *et al.* have posed bounds on the length of pseudoexhaustive tests [2] and proposed BIST pattern generators [3]. Chattopadhay proposed cellular automata pseudoexhaustive test generators in [4]. Kagaris and Tragoudas proposed pseudoexhaustive test generators using linear feedback shift registers (LFSRs) whose polynomials are

BIST is used to make faster, less-expensive integrated circuit manufacturing tests. The IC has a function that verifies all or a portion of the internal functionality of the IC. In some cases, this is valuable to customers, as well. For example, a BIST mechanism is provided in advanced fieldbus systems to verify functionality.

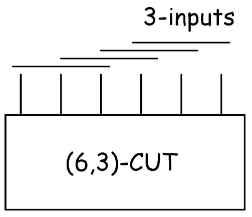

Fig. 1. (6,3)-pseudo-exhaustive testing.

not primitive, using a low number of seeds in [5]. Shaer *et al.* proposed partitioning for combinational as well as sequential [6] circuits. Novak *et al.* proposed generators to generate CA-based pseudoexhaustive generators in [7]. Gupta *et al.* [8], Stroele [9], and [10] propose other pseudoexhaustive BIST pattern generators.

Testing of blocks of certain types of circuits, such as digital signal processing systems, data path architectures, embedded memories and others, involves partitioning of the inputs into physically adjacent groups [8]. In this case, the pseudo-exhaustive objective can be reformulated in such way that the n-bit space is covered if for all $n - k + 1$ contiguous k-bit subspaces, each of the 2^k patterns occurs at least once (see Fig. 1). An (n,k)-adjacent bit pseudo-exhaustive test set (PETS) is a set of n-bit patterns in which all k-bit patterns appear into all adjacent k-bit groups. A module C that can be pseudo-exhaustively tested with an (n,k)-pseudo-exhaustive test set is called (n,k)-pseudo-exhaustively testable and k is called the cone size of C. For example, $n \times m$ array [11] and Booth [12] multipliers have been shown to be $(n+m$, 8)-pseudo-exhaustively testable modules.

Iterative logic arrays (ILAs) [13] are another type of modules where application of pseudo-exhaustive testing has been proved efficient. ILAs are structures consisting of identical logic cells, connected in a regular manner. The inherent regularity of ILAs utilizes the fast derivation of compact and efficient test sets. In many cases ILAs can be tested with a constant number of test vectors irrespective of their size, a property known as C-testability [14]. Test pattern generation and design-for-testability for ILAs has attained extensive consideration during the last decades [15]–[17].

In modern VLSI circuits, containing millions of transistors, the utilization of the same BIST pattern generator to test more than one module can drive down the cost of BIST hardware [18]. Modules whose inputs are driven (during BIST) from the same pattern generator may have different cone sizes. Two solutions have been proposed to this direction.

One solution is to utilize recursive pseudoexhaustive testing [18]; with recursive pseudoexhaustive testing, (n,k)-PETS are generated for all $k = 1, 2, 3, \ldots, n$. In the literature, recursive pseudoexhaustive testing was introduced by Rajski and Tyszer in [18], where the utilization of an array of XOR gates and a binary counter was proposed, to recursively generate all (n, k) pseudoexhaustive test sets for $k \leq n$ in optimal time. Dasgupta *et al.* [19] proposed a cellular automata-based generator for the same purpose.

An alternative solution is generic pseudoexhaustive testing, introduced in [10]. A generic pseudoexhaustive generator can generate a (n, k)-PETS for any value of k by enabling a respective signal PE$[k]$. The pseudoexhaustive generator of [10] was also utilized for recursive pseudoexhaustive testing (also termed progressive pseudoexhaustive testing in [10]) and was shown to outperform the schemes proposed in [18], [19] in terms of hardware overhead—time to complete the test tradeoff.

BIST pattern generators are commonly discerned into one-pattern and two-pattern. One-pattern generators target the detection of combinational (mainly stuck-at) faults. However, it has been proved that many failure mechanisms that appear in CMOS circuits cannot be modelled by the stuck-at fault paradigm [20], [21]. Furthermore, increasing performance requirements emphasize the need to operate digital circuits at their highest possible speeds. This motivates testing for the correct temporal behavior, commonly known as delay testing. The detection of these faults requires two pattern tests. Various published schemes target the efficient generation of two-pattern tests [22]–[35]. BIST two-pattern generators have been proposed in [25] and [31]–[35].

In this paper, we start by presenting a generic pseudoexhaustive two-pattern generation scheme. A generic pseudoexhaustive two-pattern generation scheme generates an (n, k)-pseudoexhaustive two-pattern test for any value of k, by enabling a proper input signal PE$[k]$, $1 \leq k \leq n$. To the best of our knowledge, this is the first generic pseudoexhaustive two-pattern generator to be presented in the open literature. Next, we generalize the generic pseudoexhaustive two-pattern generation scheme into a progressive two-pattern generator that generates all (n, k)-pseudoexhaustive two-pattern tests for all k, for $k = 1$ to n. We call this scheme recursive pseudoexhaustive two-pattern generation scheme. To the best of our knowledge, the proposed recursive pseudoexhaustive two-pattern generation scheme is the first to be presented in the open literature, therefore it comes to complement the one-pattern recursive pseudoexhaustive generators presented in [10], [18], and [19] in the territory of two-pattern testing. Although no recursive pseudoexhaustive two-pattern generation scheme has been proposed in the open literature, we perform a comparison with possible extension of the schemes proposed in [10], [18], and [19] to provide for recursive two-pattern pseudoexhaustive testing. From the comparison, it is derived that the proposed scheme presents lower hardware overhead than the possible extensions of the schemes in [10], [18], and [19].

The layout of this paper is as follows. In Section II, the proposed generic pseudoexhaustive two-pattern generator is presented. In Section III, the generic pseudoexhaustive generator is extended to recursively generate all (n, k)-pseudoexhaustive two-pattern tests. In Section IV, the hardware implementation and the test latency of the presented scheme are calculated. In Section V, we consider schemes that have been proposed in the open literature for recursive pseudoexhaustive one-pattern generation and possible extensions in the two-pattern domain; comparisons of such extensions reveal that the proposed scheme presents lower hardware overhead. A software implementation of the presented scheme is presented in Section VI. Finally, in Section VII we conclude the paper.

II. Generic Pseudoexhaustive Two-Pattern Testing

A *generic* pseudo-exhaustive two-pattern generator is a module with n inputs (PE$[n:1]$) and n outputs $(A[n:1])$ that can generate a two-pattern (n, k)-pseudo-exhaustive test set for any value of k, $(k \leq n)$.

At each time at most one of the PE$[i]$ signals may be enabled. When PE$[i]$ $1 \leq i \leq n$ is enabled, then a $(n, k-1)$-pseudoexhaustive two-pattern test is generated. For example, for $n = 12$, Table I presents the pseudoexhaustive test set generated for each value of the PE$[i]$ signals. In Table I, in the first column we present the value of the PE$[k]$ signals; in the second column we present the generated pseudoexhaustive test, while in the third column we present the span of the exhaustive subspaces. For example, in the fifth row of Table I, where PE[5] is enabled, a (12,4)-pseudoexhaustive test set is generated and a 4-bit exhaustive test set is applied to the 4-bit groups $A[12:9]$, $A[8:5]$, and $A[4:1]$. It is trivial to see that, in this case, the span of outputs $A[11:8]$ also receives all 4-bit combinations (since the value of the $A[12]$ output is equal to the value of the $A[8]$ output); the same holds true for all 4-bit spans, i.e., $A[10:7]$, $A[9:6]$, etc., therefore, all 4-bit groups receive an exhaustive 4-bit test set.

It should be noted that, in case the size of the pseudoexhaustive test set does not exactly divide n (for $n = 12$, this is the case for $k = 5, 7, 8, 9, 10, 11$) then the span of n bits is divided into $\lfloor n/k \rfloor$ groups of k bits, that take the same valus and the remaining $(n \bmod k)$ high-order bits have the same values with the $(n \bmod k)$ low-order bits of the low-order groups. For example, for $k = 8$, the $A[12:9]$ bits of the output of the generator have the same values with the $A[4:1]$ bits. Henceforth, if an exhaustive k-bit two-pattern test is generated at the low-order bits of the output of the generator, then an (n, k)-pseudoexhaustive two-pattern test is generated at the outputs of the generator.

The proposed generic pseudoexhaustive two-pattern generator is presented in Fig. 2.

It consists of an n-stage accumulator, comprising an adder and a register (the carry-in signal of the adder is driven by the output of a module termed carry-generator, or c_gen for short), an n-stage generic counter, and a control module.

In the sequel, we shall present the implementation and functionality of the modules comprising the generic pseudoexhaustive two-pattern generator of Fig. 2.

A. Generic Counter Module

An n-stage generic counter takes as inputs a basic clock signal (clk) and n signals PE$[n:1]$; if all signals PE$[i]$, $1 \leq i \leq n$ are disabled, then the generic counter operates as an n-stage binary counter. The same holds true in case PE$[1] = 1$.

When PE$[k]$ is activated, for some value of k, $1 \leq k \leq n$, the stages k, $2k$, $3k$, etc., are clocked by the basic clock signal. Therefore, the generic counter generates all $2^{k-1} \times (2^{k-1} - 1)$ combinations to all groups of $k-1$ adjacent bits, i.e., operates as $\lceil n/(k-1) \rceil$ consecutive k-stage counters. When C_clk_disable

TABLE I
GENERIC $(12, k)$-PSEUDOEXHAUSTIVE GENERATOR

PE[12:1]	(n, k)	Operates as...
0000 0000 0001	(12,12)	(XXXXXXXXXXXX)
0000 0000 0010	(12,1)	(X) (X) (X) (X) (X) (X) (X) (X) (X) (X) (X) (X)
0000 0000 0100	(12,2)	(XX) (XX) (XX) (XX) (XX) (XX)
0000 0000 1000	(12,3)	(XXX) (XXX) (XXX) (XXX)
0000 0001 0000	(12,4)	(XXXX) (XXXX) (XXXX)
0000 0010 0000	(12,5)	(XX) (XXXXX) (XXXXX)
0000 0100 0000	(12,6)	(XXXXXX) (XXXXXX)
0000 1000 0000	(12,7)	(XXXXX) (XXXXXXX)
0001 0000 0000	(12,8)	(XXXX) (XXXXXXXX)
0010 0000 0000	(12,9)	(XXX) (XXXXXXXXX)
0100 0000 0000	(12,10)	(XX) (XXXXXXXXXX)
1000 0000 0000	(12,11)	(X) (XXXXXXXXXXX)

TABLE II
OPERATION OF A 12-STAGE GENERIC COUNTER

PE[1:12]	Operates as...	In each clock increased by...
0000 0000 0001	1×12-stage counter	000000000001
0000 0000 0010	12×1-stage counters	111111111111
0000 0000 0100	6×2-stage counters	010101010101
0000 0000 1000	4×3-stage counters	001001001001
0000 0001 0000	3×4-stage counters	000100010001
0000 0010 0000	2×5-stage counters + 1×2-stage counter	010000100001
0000 0100 0000	2×6 stage counters	000001000001
0000 1000 0000	1×7 stage counter + 1×5-stage counter	000010000001
0001 0000 0000	1×8 stage counter + 1×4-stage counter	000100000001
0010 0000 0000	1×9 stage counter + 1×3-stage counter	001000000001
0100 0000 0000	1×10 stage counter + 1×2-stage counter	010000000001
1000 0000 0000	1×11 stage counter + 1×1-stage counter	100000000001

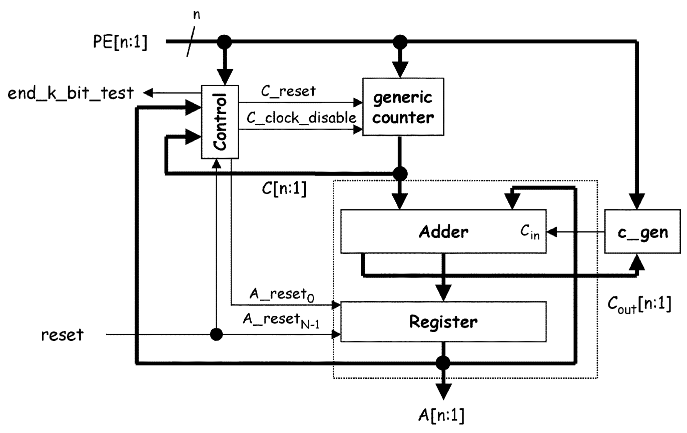

Fig. 2. Generic pseudoexhaustive two-pattern generator.

is enabled, the clk signal is disabled, and the generic counter remains idle. In Table II, we present the operation of a 12-stage generic counter for the various values of the $\mathrm{PE}[i]$ signals.

From the operation of the generic counter we can see that it can generate either one-pattern exhaustive test set (when $\mathrm{PE}[1]$ is enabled), or one-pattern (n, k)-pseudoexhaustive test set when $\mathrm{PE}[k+1]$ is enabled.

A 12-stage generic counter is presented in Fig. 3(a). It consists of 12 generic counter cells (GCC) and an OR grid. The GCC is a modified version of the typical counter cell and its implementation for the case of the ripple counter is also presented in Fig. 3(a). Implementations for other counter designs can be found in [10].

If Sel_i is enabled, the stage i of the counter is clocked by the basic clock signal. Therefore, Sel_i is enabled if and only if i is a multiple of k. Therefore, the select signal of every stage i is the output of an OR gate, whose inputs are the signals $\mathrm{PE}j$, for all j that divide i. If i is a prime number (is divided only by 1) there is no need for an OR gate, and the Select signal is driven by the signal PE_i. For example, the OR grid of a 12-stage generic counter is presented in Fig. 3(b).

B. c_gen Module

The purpose of the c_gen module is to provide for the carry input of the accumulator, depending on the value of the $\mathrm{PE}[i]$ signals. When $\mathrm{PE}[k]$ is enabled $(1 \leq k \leq n)$ then the input of the accumulator is fed from the carry output of its kth stage. Hence, the k low-order stages of the accumulator operate as a k-stage accumulator comprising a 1's complement adder, while the $(n-k)$ higher-order bits also take as a carry input the output of the k-stage carry output of the accumulator.

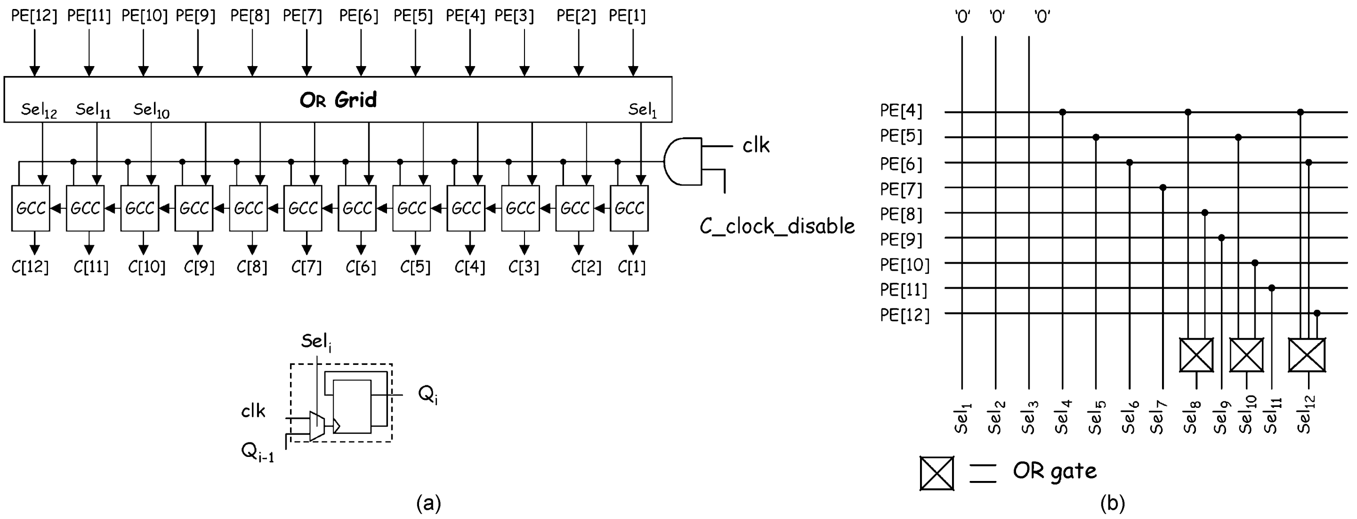
PE[12] PE[11] PE[10] PE[9] PE[8] PE[7] PE[6] PE[5] PE[4] PE[3] PE[2] PE[1]
OR Grid
Sel12 Sel11 Sel10 Sel1
clk
C_clock_disable
GCC
C[12] C[11] C[10] C[9] C[8] C[7] C[6] C[5] C[4] C[3] C[2] C[1]
Seli
clk
Qi
Qi-1
(a)
'0' '0' '0'
PE[4] PE[5] PE[6] PE[7] PE[8] PE[9] PE[10] PE[11] PE[12]
Sel1 Sel2 Sel3 Sel4 Sel5 Sel6 Sel7 Sel8 Sel9 Sel10 Sel11 Sel12
= OR gate
(b)

Fig. 3. (a) 12-stage selective counter and GCC (ripple counter implementation). (b) 12-stage OR grid.

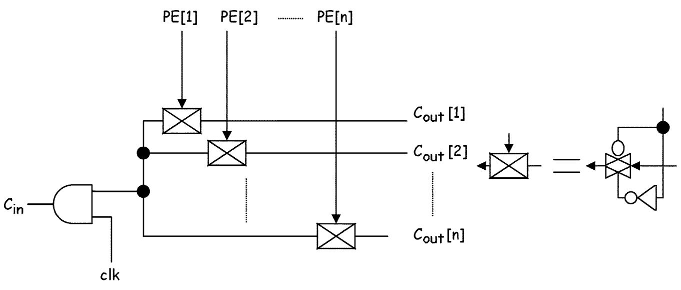

Fig. 4. c_gen (carry generator) module.

Phase	Step	Algorithm TPG(k)
		`{ int K=2^k, A=K-1;`
		`do {`

1	1	`C=0;`
	2	`do {`
	3	`C++; A = Acc(A, C, K);`
	4	`} while (C!= K-3);`
	5	`} while (A!=K-1);`
	6	`C++;`
2	1	`do { A = Acc(A, C, K); } while (A!=K-1);`
	2	`C=0;`
3	1	`do {`
	2	`C++;`
	3	`A=0;`
	4	`A = Acc(A, C, K);`
	5	`} while (A!=K-1);`
	6	`}`

		`int Acc(int A, C, K)`
		`{ return(A+C<K ? A+C: (A+C)%K+1); }`

Fig. 5. Algorithm for two-pattern test generation.

The c_gen module takes as inputs the carry outputs ($C_{\text{out}}[n:1]$) of the n stages of the accumulator and the $\text{PE}[i]$ signals $1 \leq i \leq n$. When $\text{PE}[i]$ is enabled, the $C_{\text{out}}[i]$ is enabled to feed the inputs of the carry-in signal of the accumulator. The c_gen module can be implemented in hardware utilizing pass transistor logic as shown in Fig. 4. The AND gate is disabled during the first semi-period of the clock to avoid oscillations and sequential behavior of the adder [35].

For example, let us consider the case where $\text{PE}[5]$ is enabled in a 12-stage generator. In this case, the accumulator operation can be emulated by three 4-stage sub-accumulators, where the carry input of each sub-accumulator is driven by the carry-output of the previous sub-accumulator. It is trivial to show that if a 4-stage pseudoexhaustive two-pattern test is generated at the 4 low-order stages of the accumulator, then a (12,4)-pseudoexhaustive test is generated at the outputs of the generator. In the sequel we shall present how the k-stage exhaustive two-pattern test is generated at the k-stage low-order stages when $\text{PE}[k+1]$ is generated, using the control module.

C. Control Module

The purpose of the control module is to assure that a k-stage two-pattern test is generated at the k low-order stages of the generator. If this is achieved, then, following the above reasoning, (n,k)-pseudoxhaustive test is generated. The operation of the control module is based on the algorithm presented in Fig. 5 in a C-like notation.

The Acc function performs the accumulation operation with one's complement addition. Therefore, the TPG algorithm simulates the operation of an accumulator whose inputs are driven by a binary counter. The counter counts from 1 to $K-3$ (Phase 1, steps 3–5) and then it is reset; this is repeated until the outputs of the counter are equal to $K-3$ and the outputs of the accumulator are equal to $K-1$ (Phase 1, step 6). Next (Phase 2) the counter is incremented to $\text{K}-2$ and the accumulator repeatedly accumulates $K-2$ until its output is equal to $K-1$. Finally (Phase 3) all transitions to and from zero are generated, by resetting the accumulator and incrementing the counter every second clock cycle. In [35] the TPG algorithm was proved to generate all n-bit 2-pattern tests within $2^k \times (2^k-1)+1$ clock cycles, i.e., within the theoretically minimum time. The operation of the TPG algorithm for $k=3$ stages is illustrated in Table III.

The control module takes as inputs the signals reset, $\text{ACC}[n:1]$, $C[n:1]$, $\text{PE}[n:3]$, and generates the signals C_clock_disable, C_reset, A_reset_0, end_k_bit_test. It operates as follows (in the sequel, $C[k:1]$ denotes the k low-order stage of the counter module, while $\text{ACC}[k:1]$ denotes the k low-order stages of the accumulator).

TABLE III
TWO-PATTERN TEST GENERATED BY TPG(3)

Phase #	Cycle #	C	A
	0		111
1	1	001	001
1	2	010	011
1	3	011	110
1	4	100	011
1	5	101	001
1	6	001	010
1	7	010	100
1	8	011	111
1	9	100	100
1	10	101	010
1	11	001	011
1	12	010	101
1	13	011	001
1	14	100	101
1	15	101	011
1	16	001	100
1	17	010	110
1	18	011	010
1	19	100	110
1	20	101	100
1	21	001	101
1	22	010	111
1	23	011	011
1	24	100	111
1	25	101	101
1	26	001	110
1	27	010	001
1	28	011	100
1	29	100	001
1	30	101	110
1	31	001	111
1	32	010	010
1	33	011	101
1	34	100	010
1	35	101	111
2	36	110	110
2	37	110	101
2	38	110	100
2	39	110	011
2	40	110	010
2	41	110	001
2	42	110	111
3	43	001	000
3	44	001	001
3	45	010	000
3	46	010	010
3	47	011	000
3	48	011	011
3	49	100	000
3	50	100	100
3	51	101	000
3	52	101	101
3	53	110	000
3	54	110	110
3	55	111	000
3	56	111	111

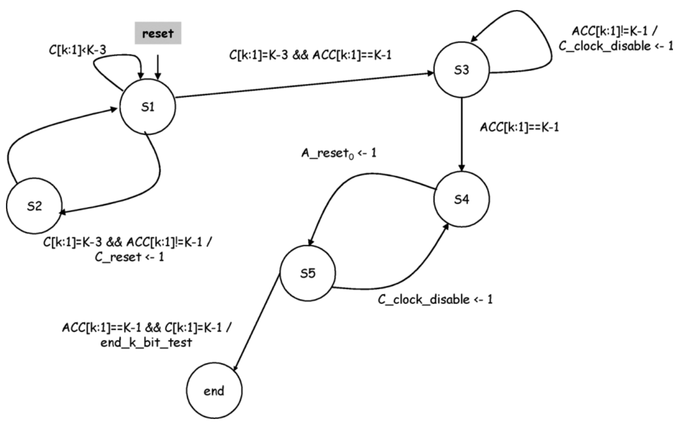

Fig. 6. State diagram of the operation of the control module.

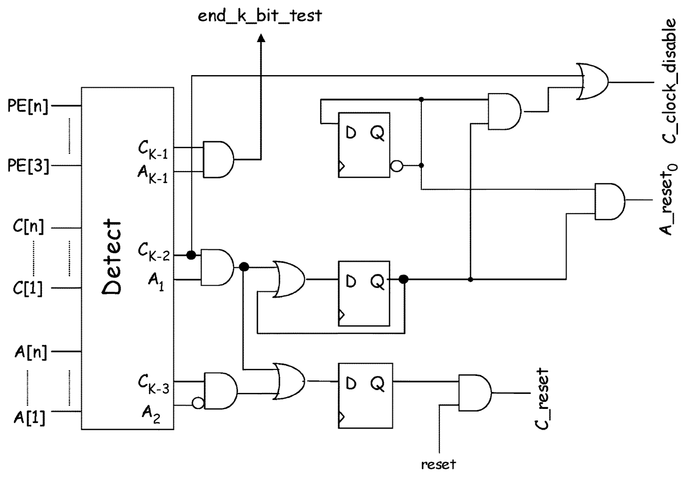

Fig. 7. Control module.

- When $C\%K == K-3$ (i.e., $C[k:1] = K-3$), the generic counter is reset in the next cycle (Phase 1—Step 5 of the TPG algorithm, Fig. 4).
- When $C\%K == K-3$ (i.e., $C[k:1] = K-3$) and $\text{ACC}\%K == K-1$ (i.e., $\text{ACC}[k:1] = K-1$) the counter is clocked one more time and the counter clock is disabled (from the next cycle), (Phase 1—Step 6).
- When $\text{ACC}\%K == K-1$, then the third phase of the PET algorithm commences, during which the clock of the selective counter is driven by the divided-by-two clock signal and the accumulator is reset to 0 every second clock cycle.
- When both the accumulator and the generic counter reach the value $K-1$, the end_k_bit_test signal is enabled, to indicate the end of the (n,k)-pseudoexhaustive test.

The state diagram of the operation of the control module is presented in Fig. 6. An implementation of the control module is presented in Fig. 7.

The detect module is a series of OR-AND gates that detect the occurrence of certain values at the outputs of the $A[n:1]$, $C[n:1]$, and $\text{PE}[n:1]$ buses. An implementation of the detect module is presented in Fig. 8 for the case $n=7$. Please note that, in Fig. 8 the signal A_{N-1} is calculated instead of the signal A_{K-1}. The two signals are equivalent, since when $A[k:1] = K-1$, then the k low-order outputs of the accumulator are equal to 1, hence all outputs of the accumulator are equal to 1. The implementation of A_{N-1} is preferred over the implementation of A_{K-1}, since the former requires less hardware overhead (n gates instead $2n$ gates required for the calculation of A_{K-1}).

Example: In Fig. 8, the case where a (7,4)-Pseudo exhaustive test is aimed; hence, $\text{PE}[5] = 1$. Given that $\text{PE[i]} = 0$ for all i $\neq 5$, the shaded AND gates are disabled, therefore, only $C[4:1]$ and $A[4:1]$ affect the values of the signals C_{K-3}, C_{K-2}, C_{K-1}, A_1, A_2, detecting the values C_{13}, C_{14}, C_{15}, A_1, A_2, respectively, calculated as illustrated in Table IV ($'$ denotes the negation operation). Signal $A_{N-1} = A_{K-1}$ is an n-input AND gate.

III. RECURSIVE PSEUDOEXHAUSTIVE TWO-PATTERN TESTING

In order to generate (n,k)-PETS for all $k \leq n$, one can utilize the module presented in Fig. 2 and recursively enable $\text{PE}[k]$ for all values of k, $2 \leq k \leq n$. In order to accomplish this, the module presented in Fig. 9 comprises an $m = \lceil \log_2 n \rceil$-stage

TABLE IV
ILLUSTRATION OF THE OPERATION OF THE 7-STAGE DETECT MODULE OF FIG. 8

Signal	Is enabled when	It detects	Comment
C_{K-3} = C[1] . C[2]' . C[3] . C[4]	C[7:1] = XXX1101	$(1101)_2=(13)_{10}$= K-3	(since K=16)
C_{K-2} = C[1]' . C[2] . C[3] . C[4]	C [7:1] = XXX1110	$(1110)_2=(14)_{10}$= K-2	(since K=16)
C_{K-1} = C[1] . C[2] . C[3] . C[4]	C[7:1] = XXX1111	$(1111)_2=(15)_{10}$= K-1	(since K=16)
A_1 = A[1] . A[2]' . A[3]' . A[4]'	A[7:1] = XXX0001	A=$(0001)_2=(1)_{10}$= 1	
A_2 = A[1]' . A[2] . A[3]' . A[4]'	A[7:1] = XXX0010	A=$(0010)_2=(2)_{10}$= 2	
A_{N-1}= A[1] . A[2] . A[3] . A[4] . A[5] . A[6] . A[7]	A[7:1] = 1111111	A=$(1111111)_2=(127)_{10}$= N-1	(since N=128)

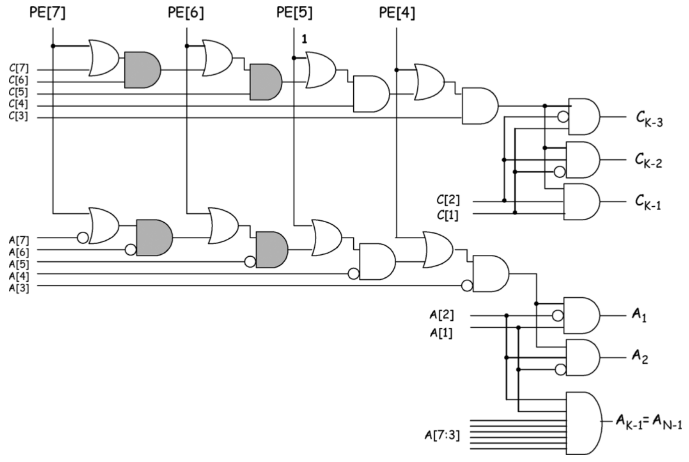

Fig. 8. 7-stage detect module.

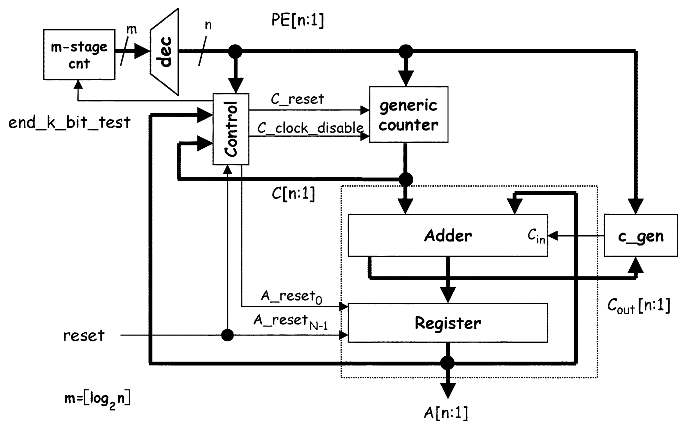

Fig. 9. Recursive pseudo-exhaustive two-pattern generator.

counter driving the inputs of an m-to-n decoder and operates as follows. Initially, the m-stage counter is set to 3; therefore the output of the decoder is $\mathrm{PE}[\mathrm{n}:1] = 000\ldots1000$. The selective counter operates as consecutive 3-stage counters and increments by $\ldots$ 001 001 every time it is clocked.

Furthermore, the $\mathrm{PE}[n:1]$ signal is driven to the c_gen module indicating that $C_{\mathrm{out}}[3]$ (i.e., the output of the third stage of the accumulator) will drive the C_{in} input of the accumulator. In this way, a two-pattern $(n,3)$-PETS is generated at the outputs of the accumulator. When the $(n,3)$-PETS is complete (i.e., when $\mathrm{C}[3:1] = 2^3-1 = 7$ and $\mathrm{ACC}[3:1] = 2^3-1=7$), the control module increments the m-stage counter to 4; therefore, the output of the decoder becomes $0\ldots01000$, the generic counter is reset to 0, the accumulator is reset to $N-1$, and a two-pattern $(n,4)$-PETS commences in a similar fashion. When the two-pattern (n,n)-PET is complete, the recursive pseudoexhaustive test is also complete.

TABLE V
CALCULATION OF THE HARDWARE OVERHEAD OF THE PRESENTED SCHEME

Module	Hardware overhead	Gates
m-stage counter	m×DFF	8×m
m-to-n decoder	≤2×n	2×n(1)
Control + Detect	9 gates + 3 DFF + 5×n gates	9 + 24 + 5×n
Generic counter	n×DFF + n×MUX21 + OR-grid	8×n + 3×n + 2×n(2)
Accumulator	n × (DFF+2×XOR+3*NAND)	19×n
c_gen	n transmission gates	n

(2) In [31] it was shown that the hardware overhead of the m-to-2 decoder is less than $2n$ gates.
(2) It was shown in [10] that the hardware overhead of the OR grid is less than $2n$ gates.

IV. HARDWARE AND TEST LATENCY

A. Hardware Overhead

In order to calculate the hardware overhead of the proposed generator, we have considered that a D-type flip-flop accounts for eight gate equivalents and a XOR gate for four gate equivalents. The implementation of the generic pseudoexhaustive two-pattern generator requires the control module and the generic counter. The recursive pseudoexhaustive generator additionally requires the m-stage counter and the m-to-n decoder. The hardware overhead of the modules is presented in Table V.

In Table VI, we present the hardware overhead of the generic and recursive pseudoexhaustive two-pattern generators for the following cases: 1) none of the required modules exists; 2) an accumulator exists in the data path; 3) an accumulator whose inputs are driven by the outputs of a register exists; in this case, the register can be transformed into a generic counter by adding $2\times n$ 2-way multiplexers; 4) an accumulator exists whose inputs are driven by an n-stage counter.

In case this hardware overhead is not acceptable, a software implementation of the proposed scheme may be utilized, as presented in a subsequent section, to generate the recursive two-pattern PETS with actually no hardware overhead.

B. Test Latency

The presented generator is not optimal in terms of time required to complete the recursive pseudoexhaustive two-pattern test. We define the *latency* of the recursive pseudoexhaustive generator over an optimal pseudoexhaustive generator as the

TABLE VI
HARDWARE OVERHEAD OF THE GENERIC AND RECURSIVE AND PSEUDOEXHAUSTIVE TWO-PATTERN GENERATION SCHEME

#	Existing Modules	Generic scheme		Recursive scheme	
		Hardware overhead	Gates	Hardware overhead	Gates
(1)	None	Control + Detect + Accumulator + c_gen + Generic Counter	38×n	m-stage counter + m-to-n decoder + Control + Detect + Accumulator + c_gen + Generic Counter	40×n+8×m
(2)	Accumulator	Control + Detect + c_gen + Generic Counter	21×n	m-stage counter + m-to-n decoder + Control + Detect + c_gen + Generic Counter	23×n+8×m
(3)	Accumulator +register	Control + Detect + c_gen + Transform register into generic counter	16×n	m-stage counter + m-to-n decoder + Control + Detect + c_gen + Transform register into generic counter	18×n+8×m
(4)	Accumulator +counter	Control + Detect + c_gen + Transform counter into generic counter	12×n	m-stage counter + m-to-n decoder + Control + Detect + c_gen + Transform counter into generic counter	15×n+8×m

TABLE VII
TIME OVERHEAD FOR RECURSIVE PSEUDOEXHAUSTIVE TWO-PATTERN GENERATION

k	$2^k \times (2^k-1)$	$\sum_{i=1}^{k} 2^i \times (2^i-1)$	Latency
2	12	14	14,28%
3	56	70	20,00%
4	240	310	22,58%
5	992	1.302	23,80%
6	4.032	5.334	24,40%
7	16.256	21.590	24,70%
8	65.280	86.870	24,85%
9	261.632	348.502	24,92%
10	1.047.552	1.396.054	24,96%
11	4.192.256	5.588.310	24,98%
12	16.773.120	22.361.430	24,99%
13	67.100.672	89.462.102	24,99%
14	268.419.072	357.881.174	24,99%

fraction of the clock cycles required to generate all the i-subspaces for $i < k$ over the number of clock cycles required for the generation of the vectors required to cover all i-subspaces for $i \leq k$, given by the following formula:

$$\text{Latency} = \frac{\sum_{i=1}^{k-1} 2^i \times (2^i - 1)}{\sum_{i=1}^{k} 2^i \times (2^i - 1)}.$$

In order to calculate the latency of the proposed recursive scheme, we shall approximate $2^n \times (2^n - 1)$ with 2^{2n} (for $n > 4$ this approximation results in an error of less than 4%). Then

$$\sum_{i=1}^{k-1} 2^{2i} = \frac{4 \times 2^{2k} - 2^2}{3} = \frac{4}{3} \times (2^{2k} - 1).$$

Therefore

$$\text{Latency} = \frac{\sum_{i=1}^{k-1} 2^{2i}}{\sum_{i=1}^{k} 2^{2i}} = \frac{\frac{4}{3} \times (2^{2k} - 1)}{\frac{4}{3} \times \left(2^{2(k+1)} - 1\right)} \approx \frac{1}{4} = 25\%. \quad (1)$$

In Table VII, we have calculated the latency for various values of k. From Table VII, the value calculated in (1) is a good approximation for $n > 4$.

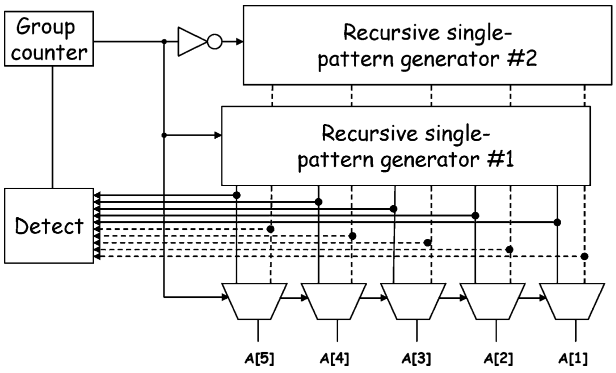
Group counter
Recursive single-pattern generator #2
Recursive single-pattern generator #1
Detect
A[5]
A[4]
A[3]
A[2]
A[1]

Fig. 10. Extension of one-pattern recursive pseudoexhaustive generators to generate two-pattern recursive pseudoexhaustive test sets.

TABLE VIII
PATTERNS GENERATED BY THE GENERATOR OF [18], [19] FOR $n = 4$

Pattern	Group				Pattern
T1	G1	G2	G3	G4	0000
T2					1111
T3	▨				1010
T4	▨				0101
T5	▨	▨			1100
T6	▨	▨			0011
T7	▨	▨			0110
T8	▨	▨			1001
T9	▨	▨	▨		1000
T10	▨	▨	▨		0111
T11	▨	▨	▨		0010
T12	▨	▨	▨		1101
T13	▨	▨	▨		0100
T14	▨	▨	▨		1011
T15	▨	▨	▨		1110
T16	▨	▨	▨		0001

V. COMPARISONS

Although, to the best of our knowledge, no recursive pseudoexhaustive two-pattern generation scheme has been presented in the open literature, some of the schemes that have been proposed for two-pattern generation could be extended to recursively generate all pseudoexhaustive tests. Two-pattern generation schemes have been proposed by Starke [20], Vuksic and Fuchs [34], and Chen and Gupta [32].

TABLE IX
RECURSIVE PSEUDOEXHAUSTIVE TWO-PATTERN GENERATORS: COMPARISON

Scheme	Existing modules	Transformations	Gate equivalents
[18]	Two n-stage registers	Transform two registers into counters + XOR gates + n 2-input multiplexers + m-stage group counter + Detect	7×n + XOR gates + 3×n + 8×m
[19]	Two n-stage registers	Transform two registers into CA + n 2-input multiplexers + m-stage group counter + Detect	7×n + XOR gates + 3×n + 8×m
[10]	Two n-stage registers	Transform two registers into generic counters + 2 m-to-n decoders + 2 m-stage counters + n 2-input multiplexers + m-stage group counter + Detect	21×n+24×m
RPET	Accumulator +register	m-stage counter + m-to-n decoder + Control + Detect + c_gen + Transform register into generic counter	18×n+8×m

TABLE X
HARDWARE OVERHEAD OF RECURSIVE PSEUDOEXHAUSTIVE GENERATORS: CALCULATION OF HARDWARE OVERHEAD IN GATES

	[18]		[19]		[10]	RPET
n	#xor	#gates	#xor	#gates		
3	3	70	7	102	111	67
4	4	88	7	112	132	84
5	8	138	15	194	177	109
6	10	164	15	204	198	126
7	12	190	15	214	219	143
8	13	208	15	224	240	160
9	21	290	31	370	285	185
10	25	332	31	380	306	202
11	29	374	31	390	327	219
12	31	400	31	400	348	236
13	35	442	31	410	369	253
14	37	468	31	420	390	270
15	39	494	31	430	411	287
16	40	512	31	440	432	304
17	56	658	63	714	477	329
18	64	732	63	724	498	346
19	72	806	63	734	519	363
20	76	848	63	744	540	380
21	84	922	63	754	561	397
22	88	964	63	764	582	414
23	92	1006	63	774	603	431
24	94	1032	63	784	624	448
25	102	1106	63	794	645	465
26	106	1148	63	804	666	482
27	110	1190	63	814	687	499
28	112	1216	63	824	708	516
29	116	1258	63	834	729	533
30	118	1284	63	844	750	550
31	120	1310	63	854	771	567
32	121	1328	63	864	792	584

Starke has proposed PETT [20]. In PETT, a nonlinear feedback shift register with $2n$ stages is used for the testing of an n-bit CUT. Assuming that before the insertion of the BIST circuitry an n-stage register existed in the inputs of the CUT, with PETT n additional flip-flops are inserted (for the formation of the $2n$-stage NFSR). Furthermore, n multiplexers are inserted to the inputs of the register flip-flops, and logic gates (OR) with totally $2n$ inputs must be included in order to implement the non-feedback operation.

Vuksic and Fuchs proved [34] that a multiple input shift register (MISR) can generate all transitions if it receives all the 2^n input combinations. Assuming the existence of n flip-fops (n stage register) at the inputs of the CUT, the BIST circuitry requires the insertion of n multiplexers, n XOR gates and n flip-flops (that will generate the 2^n input combinations).

Chen and Gupta [32] investigated how an exhaustive two-pattern test can be generated using either a linear feedback shift register (LFSR) or a cellular automaton (CA). Their results show that for an n-input CUT, an LFSR, or CA with at least $2n$ stages is required. Assuming the existence of n flip-flops at the inputs of the CUT, the implementation of the LFSR version requires n flip-flops (for the formation of the $2n$ stage LFSR) and n multiplexers at the inputs of the existing flip-flops.

The CA-version of the technique requires n flip flops (for the formation of the $2n$-stage CA), n multiplexers at the inputs of the existing flip-flops and a number of XOR gates for the formation of the CA rules. In order to calculate the number of XOR gates, we assume that half of the stages implement rule 90, while the others implement rule 150. This assumption is justified since these two rules are the most commonly used in Cellular Automata applications [19]. Rule 90 requires one 2-input XOR gate, while Rule 150 requires two 2-input XOR gates.

The above-mentioned schemes have been shown to perform very well in the field of exhaustive, or pseudorandom testing; furthermore, [32] has been also shown to be effective for (n, k)-pseudo-exhaustive testing for a specific value of k; however, since they are based on LFSRs or CA, their extension to recursively generate two-pattern tests, would require the formation of n different feedback polynomials for an n-stage generator, as well as n multiplexers, each one having n inputs, in order to allow for the different feedback polynomials to generate the required pseudo-exhaustive tests. The hardware overhead of these multiplexers is $3 \times n^2$ gates, which is prohibitive; for example, for $\mathrm{n} = 32$, the hardware overhead is ≈3000 gates, which is about 5 times the overhead of the proposed scheme.

An alternative solution to the problem would be to utilize one of the recursive pseudoexhaustive one-pattern generation schemes proposed in [10], [18], [19], generate two-pattern tests using two generators and multiplex their outputs as presented in Fig. 10. In this case, one out of two approaches could be adopted.

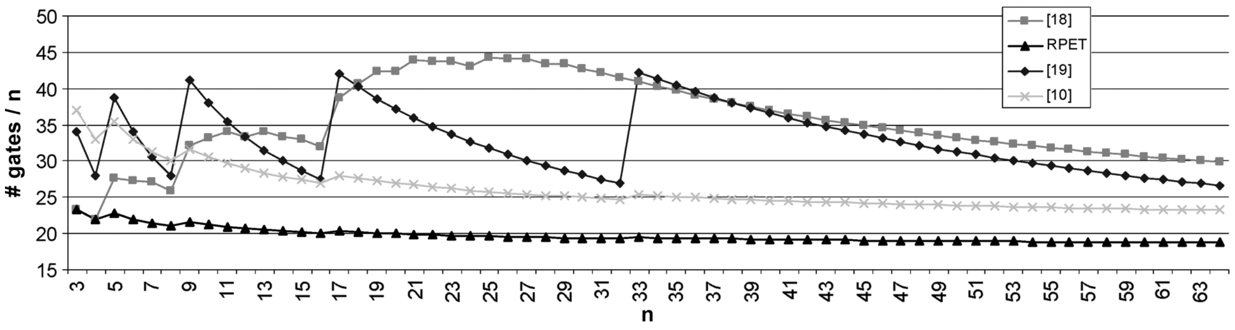

Fig. 11. Recursive pseudoexhaustive two-pattern generations schemes: comparison.

```
PROC RPET
    MOV k, 3
    MOV K, 8
L11: CALL PET
    INC k
    SL  K
    CMP k,n
    JNE L11
ENDP RPET

PROC k_ACC
    ADD A, C
    MOV AK, A
    AND AK, K-1
    MOV CK, C
    AND CK, K-1
    ADD AK, CK
    CMP AK, K
    JL L12
    INC A
L12:
ENDP k_ACC

PROC CALC_MASK
    MOV MASK, 0
    MOV I, 1
L31:ADD MASK, I
    SL I, k
    CMP I, N
    JL L31
ENDP CALC_MASK
```

```
void RPET(n){
    k=3;
    K=8;
L11: PET(n,k);
    k++;
    K=K*2;
    if (k<n)
    goto L11;
}

int k_acc(int A,C,N,K){
  A = (A + C)%N;
  AK = A%K;

  CK = C%K;

  AK=AK+CK;
  if (AK >= K)
   A++;
 return (A);
}


int cal_mask(int n, k)
{int i, mask=0;
     I=1;
L31: mask += I;
     I = I * K;
     if (I<N)
     goto L31;
return (mask);}
```

```
PROC PET
    MOV AK,K-1
    MOV A, N-1
    CALL CALC_MASK
    MOV I_MASK, MASK
    XOR I_MASK, N-1
L1:
    MOV C,0
L2:
    ADD C, MASK
    CALL k_ACC
    CMP CK, K-3
    JNE L2
    CMP A, N-1
    JNE L1

    ADD C, MASK
L3:
    CALL k_ACC
    CMP A, N-1
    JNE L3

    MOV C, 0
L4:
    ADD C, MASK
    MOV A, 0
    CALL k_ACC
    CMP A, N-1
    JNE L4
ENDP PET
```

```
void PET(n, k)
{ int i, N=2^n, K=2^k,
  mask, i_mask,
  A = N-1;
  mask=calc_mask(n,k);
  i_mask = N-1 - mask;
L1:
    C=0;
L2:
    C += mask;
    A=k_acc(A,C,N,K);
    if (C%K!= K-3)
    goto L2
    if (A!=N-1)
    goto L1

    C += mask;
L3:
    A=k_acc(A,C,N,K); }
    if (A!=N-1)
    goto L3

    C=0;
L4:
    C += mask;
    A=0;
    A=k_acc(A,C,N,K);
    if (A!=N-1)
    goto L4;
}
```

Fig. 12. Software implementation of the proposed recursive pseudoexhaustive generator.

According to the first approach, all n-bit patterns of the first generator are combined with all n-bit patterns generated by the second generator. This approach is optimal with respect to the n-bit two-pattern generation scheme, in the sense that all $2^n \times (2^n - 1)$ pairs are generated within $2^n \times (2^n - 1)$ clock cycles, however the k-bit subspaces for $k < n$ are not covered in optimal time. In order to illustrate this approach, let us consider an $n = 4$ input CUT and the 16 patterns generated by the generators of [18], [19] which are presented in Table VIII. In Table VIII, groups G1, G2, G3, G4 represent the patterns required to cover 1-, 2-, 3-, and 4-bit subspaces, respectively. According to the first approach all patterns T1–T16 of the first generator are paired with all patterns T1–T16 of the second generator, covering the 4-bit space within $2^4 \times (2^4 - 1) + 1$ clock cycles. However, this solution does not guarantee that subspaces for $k = 1, 2, 3$ are covered within $2^1 \times (2^1 - 1) + 1$, $2^2 \times (2^2 - 1) + 1$, and $2^3 \times (2^3 - 1) + 1$ cycles, respectively. In fact, all subspaces are only guaranteed to be covered within $2^4 \times (2^4 - 1) + 1$ clock cycles. Therefore, this approach cannot be considered as a recursive pseudo-exhaustive two-pattern BIST scheme.

The second approach is to combine all patterns of each group (i.e., G1, G2, G3, and G4) of the first generator with all patterns of the respective group of the second generator. Therefore, the time required to cover 1-bit subspaces is $2^1 \times (2^1 - 1) + 1$, the time required to cover 2-bit subspaces is $2^2 \times (2^2 - 1) + 1$, the time required to cover 3-bit subspaces is $2^3 \times (2^3 - 1) + 1$ and the time required to cover 4-bit subspaces is $2^4 \times (2^4 - 1) + 1$. Therefore, the total time required for the completion of the test is given by: $2^1 \times (2^1 - 1) + 1 + 2^2 \times (2^2 - 1) + 1 + 2^3 \times (2^3 - 1) + 1$ $+ 2^4 \times (2^4 - 1) + 1$. It is trivial to see that, in this case, the time required to cover all subspaces is sub-optimal; in fact it is equal to the time required by the proposed scheme.

In the latter approach, the hardware overhead required in order to recursively generate all two-pattern tests is two-times the hardware overhead of the recursive one-pattern generator,

n 2-input multiplexers, and the additional logic. In order to implement the control logic, a structure similar to the one utilized in the proposed scheme will be issued, comprising at least a group counter with m stages, where $m = \lceil \log_2 n \rceil$, and a module needed to detect the end of the test for each sub-group (i.e., G1, G2, G3, etc.) with $4 \times n$ gates overhead.

In Table IX, we compare the proposed scheme with the described possible extensions of the schemes proposed in [18], [19], and [10] in terms of hardware overhead. In the second column of Table IX we present the modules whose existence we assume for the implementation of the schemes. In the third column we present the required transformations, while in the fourth column we present the gate equivalents required for the modifications. The term "XOR gates" refers to the number of

for the schemes [18] and [19] and have been quoted from the respective references. They have been included in the Table for the sake of completeness. The columns that present the hardware overhead of the four scheme are the columns denoted "[18]-#gates", "[19]-#gates", "[10]" and "RPET".

In Fig. 11, we present graphically the quantity HO(n)/n, i.e., the hardware overhead as a function of the number of stages, over the number of stages (we have chosen to divide the hardware overhead over the number of the generator outputs in order for the comparisons to be more clearly presented). It is evident that the proposed scheme presents lower hardware overhead from the possible extensions of the schemes proposed in [10], [18] and [19] to provide for recursive pseudo-exhaustive two-pattern testing.

VI. Software Implementation

In the case that the accumulator belongs to the datapath of a processor [8] then, instead of implementing the control module, the BIST program can be integrated into the memory of the processor. In Fig. 12 the segment of code that emulates the BIST operation is presented. The C-like notation is also given for illustrative purposes.

VII. Conclusion

Pseudoexhaustive test pattern generators provide very high fault coverage without the need for fault simulation or deterministic test pattern generation. Various techniques have been proposed for pseudoexhaustive test pattern generation for combinational faults. Adjacent bit pseudoexhaustive testing is mainly targeted to data path architectures that have a strongly bit-organized character and contain internal buses that are partitioned into physically adjacent lines [8].

In modern VLSI circuits, containing millions of transistors, the utilization of the same BIST pattern generator for testing more than one modules can drive down the hardware overhead, increasing the applicability of the BIST concept [18]. Modules whose inputs are driven (during BIST) from the same pattern generator may have different cone sizes. Recursive pseudoexhaustive testing has been proposed as a solution to this problem; in [10], [18], and [19] recursive and generic pseudoexhaustive generators for the detection of stuck-at faults have been proposed. On the contrary, no recursive pseudoexhaustive two-pattern generator has been presented to date.

In this paper we have presented a two-pattern generation scheme that can generate both generic and recursive pseudoexhaustive tests; with this scheme, more than one circuit under test, possibly having different cone sizes (k) can be tested in parallel. To the best of our knowledge, this is the first time in the open literature that the problem of recursive pseudoexhaustive two-pattern generation is addressed.

Comparisons of the proposed scheme with schemes proposed previously to recursively generate pseudoexhaustive one-pattern tests properly extended to generate two-pattern tests, reveal that the proposed scheme generates the recursive pseudoexhaustive two-pattern tests with lower hardware overhead.

References

[1] Ioannis Voyiatzis, Dimitris Gizopoulos, and Antonis Paschalis," *"Recursive Pseuo-exhaustive Two Pattern Generation",IEEE Trans Very Large Scale Integr. (VLSI) Syst., vol. 18, no. 1, Jan. 2010. 8,*

[2] R. Srinivasan, S. K. Gupta, and M. A. Breuer, "Novel test pattern generators for pseudoexhaustive testing," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1228–1240, Nov. 2000.

[3] S. Chattopadhyay, "Efficient circuit specific pseudoexhaustive testing with cellular automata," in *Proc. 11th IEEE Asian Test Symp.*, 2002, p. 188.

[4] D. Kagaris and S. Tragoudas, "Pseudoexhaustive TPG with a provably low number of LFSR seeds," in *Proc. IEEE Int. Conf. Comput. Des.: VLSI Comput. Processors, 2000, p. 42.*

[5] B. Shaer, K. Aurangabadkar, and N. Agarwal, "Testable sequential circuit design: Partitioning for pseudoexhaustive test," in *Proc. IEEE Comput. Soc. Ann. Symp. VLSI (ISVLSI)*, 2003, p. 244.

[6] O. Novak, "Pseudorandom, weighted random and pseudoexhaustive test patterns generated in universal cellular automata," in *Proc. 3rd Eur. Dependable Comput. Conf. Dependable Comput.*, 1999, vol. 1667, Lecture Notes In Computer Science, pp. 303–320.

[7] S. Gupta, J. Rajski, and J. Tyszer, "Arithmetic additive generators of pseudo-exhaustive test patterns," *IEEE Trans. Comput.*, vol. 45, no. 8, pp. 939–949, Aug. 1996.

[8] A. Stroele, "A self test approach using accumulators as test pattern generators," in Proc. Int. Symp. Circuits Syst., 1995, pp. 2120–2123.

generators," in *Proc. Int. Symp. Circuits Syst.*, 1995, pp. 2120–2123.

[10] I. Voyiatzis, "A counter-based pseudo-exhaustive pattern generator for BIST applications," *Microelectron. J.*, vol. 35, no. 11, pp. 927–935, 2004.

[11] D. Gizopoulos, A. Paschalis, and Y. Zorian, "An effective BIST BIST applications," *Microelectron. J.*, vol. 35, no. 11, pp. 927–935,

[12] A. D. Friedman, "Easily testable iterative systems," *IEEE Trans. Comput.*, vol. 22, no. 12, pp. 1061–1064, Dec. 1973.

[13] K. Teng, H. Takahashi, and Y. Takamatsu, "A general BIST-amenable *Symp.*, 2000, pp. 171–176.

[14] W. H. Kautz, "Testing for faults in cellular logic arrays," in *Proc. 8th*

[15] R. Parthasarathy and S. M. Reddy, "A testable design of iterative logic arrays," *IEEE Trans. Comput.*, vol. C-30, no. 11, pp. 833–841, Nov. 1981.

[16] A. D. Friedman, "A functional approach to efficient fault detection 1365–1375, Dec. 1994.

[17] J. Rajski and J. Tyszer, "Recursive pseudoexhaustive test pattern generation," *IEEE Trans. Comput.*, vol. 42, no. 12, pp. 1517–1521, Dec.

[18] P. Dasgupta, S. Chattopadhyay, P. P. Chaudhuri, and I. Sengupta, “Cellular automata-based recursive pseudo-exhaustive test pattern generation,” *IEEE Trans. Comput.*, vol. 50, no. 2, pp. 177–185, Feb. 2001.

[19] C. Starke, “Built-in test for CMOS circuits,” in *Proc. IEEE Int. Test Conf.*, Oct. 1984, pp. 309–314.

[20] R. Wadsack, “Fault modeling and logic simulation of CMOS and nMOS integrated circuits,” *Bell Syst. Techn. J.*, vol. 57, pp. 1449–1474, May–Jun. 1978.

[21] P. Girard, C. Landrault, S. Pravossoudovitch, and A. Virazel, “Comparison between random and pseudorandom generation for BIST of delay, stuck-at and bridging faults,” in *Proc. IEEE On-Line Test. Workshop*, 2000, pp. 121–126.

[22] A. Virazel, R. David, P. Girard, C. Landrault, and S. Pravossoudovitch, “Delay fault testing: Choosing between random SIC and random MIC sequences,” in *Proc. IEEE Eur. Test Workshop*, 2000, pp. 9–14.

[23] H. Rahaman, D. Das, and B. Bhattacharya, “Transition count based BIST for detecting multiple stuck-open faults in CMOS circuits,” in *Proc. 2nd IEEE Asia Pac. Conf. ASICs*, Aug. 2000, pp. 307–310.

[24] E. Gizdarski, “Detection of delay faults in memory address decoders,” *J. Electron. Test.*, vol. 16, no. 4, pp. 381–387, Aug. 2000.

[25] M. K. Michael and S. Tragoudas, “Functions-based compact test pattern generation for path delay faults,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 8, pp. 996–1001, Aug. 2005.

[26] I. Pomeranz and S. M. Reddy, “On n-detection test sets and variable n-detection test sets for transition faults,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 3, pp. 372–383, Mar. 2000.

[27] Y. Shao, I. Pomeranz, and S. M. Reddy, “On generating high quality tests for transition faults,” in *Proc. 11th ATS*, 2002, pp. 1–8.

[28] K. Yang, K. T. Cheng, and L. C. Wang, “TranGen: A SAT-based ATPG for path-oriented transition faults,” in *Proc. ASP-DAC*, 2004, pp. 92–97.

[29] S. Neophytou, M. Michael, and S. Tragoudas, “Functions for quality transition-fault tests and their applications in test-set enhancement,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 12, pp. 3026–3035, Dec. 2006.

[30] I. Voyiatzis, T. Haniotakis, and C. Halatsis, “A novel algorithm for the generation of SIC pairs and its implementation in a BIST environment,” *IEE Proc. Circuits, Devices Syst.*, vol. 153, no. 5, pp. 427–432, Oct. 2006.

[31] C. Chen and S. Gupta, “BIST test pattern generators for two-pattern testing-theory and design algorithms,” *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 257–269, Mar. 1996.

[32] I. Voyiatzis, A. Paschalis, D. Nikolos, and C. Halatsis, “Accumulator-based BIST approach for two-pattern testing,” *J. Electron. Test.: Theory Appl.*, vol. 15, no. 3, pp. 267–278, Dec. 1999.

[33] A. Vuksic and K. Fuchs, “A new BIST approach for delay fault testing,” in *Proc. Eur. Des. Test Conf.*, Mar. 1994, pp. 284–288.

[34] I. Voyiatzis, “Accumulator-based pseudo-exhaustive two-pattern generation,” *J. Syst. Arch.*, vol. 35, no. 11, pp. 846–860, Nov. 2007.