Test Pattern Generation Based On Arithmetic Operations *

Sanjay Gupta, Janusz Rajski and Jerzy Tyszer
Microelectronics and Computer Systems Lab
McGill University, Montréal, Canada H3A 2A7

Abstract

Existing built-in self test (BIST) strategies require the use of specialized test pattern generation hardware which introduces significant area overhead and performance degradation. In this paper, we propose a novel method for implementing test pattern generators based on adders widely available in data-path architectures and digital signal processing circuits. Test patterns are generated by continuously accumulating a constant value and their quality is evaluated in terms of the pseudo-exhaustive state coverage on subspaces of contiguous bits. This new test generation scheme, along with the recently introduced accumulator-based compaction scheme facilitates a BIST strategy for high performance datapath architectures that uses the functionality of existing hardware, is entirely integrated with the circuit under test, and results in at-speed testing with no performance degradation and area overhead.

1 Introduction

The increasing complexity of VLSI circuits in the absence of a corresponding increase in the number of input and output pins, has made Built-in Self Test (BIST) an extremely successful test strategy in the last decade. The fundamental idea in BIST is to integrate the test pattern generation and test response compaction functions for the circuit-under-test (CUT) on the chip itself [1].

To guarantee a high quality of testing, the test pattern generators used for BIST must be able to exercise most of the faults in the CUT, and the compactor should be able to preserve this coverage. These functions till now have been performed by specialized dedicated hardware. Even though, as in BILBO [2] this hardware may share registers from the CUT, it results in substantial hardware overhead and significant performance degradation due to the addition of multiplexors in the signal paths.

*This work is supported by a Collaborative Research and Development Grant from the Natural Sciences and Engineering Research Council of Canada and Bell-Northern Research Ltd.

General purpose computing structures based on data-path architectures, as well as specialized digital signal processing circuits can perform powerful arithmetical and logical operations which can be exploited for test pattern generation and response compaction. A novel scheme for parallel compaction of test responses using existing accumulators with minimal area overhead, no performance degradation and similar aliasing probability to that of LFSRs has been proposed in [3]. If such existing arithmetical and logical structures could also be used to generate test patterns, then the need for additional hardware to implement a BIST strategy on such circuits would almost completely be eliminated. These vectors can then be distributed to different modules of the system since arithmetic and logic units (ALUs) or even simple adders usually constitute the core of such systems (Fig. 1).

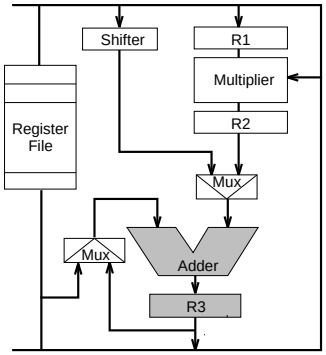


Figure 1: Typical Data-path Core

Extensive studies have been performed to analyze the testing properties of sequences generated by LFSRs and cellular automata [1], [4]. In contrast, though sequences generated using arithmetic and logical operations have been widely studied as sources of random numbers for Monte-Carlo simulation [5], their properties with respect to testing are largely unknown.

In this paper we propose a new test generation scheme that uses existing accumulators to generate parallel test patterns with no area overhead or performance degradation. In this scheme, the accumulator with an n-bit adder is used to generate a se-

quence of binary patterns by continuously accumulating a constant value. The patterns produced are pseudo-exhaustive [6], [7] in nature and can be used to test modules with physically adjacent input lines. The pseudo-exhaustive state coverage of such a scheme is analyzed and the best generators are identified.

2 Preliminaries

An n-bit ALU or accumulator featuring a binary adder can be used to implement a scheme in which a constant binary vector is successively transferred to the adder and added to the previous contents of the register R3 (Fig. 1). The state X_i of the register after i external vectors have been applied is given by

$$X_i = X_{i-1} + a \pmod{2^n} \tag{1}$$

The contents of the register R3 can now be used as a source of test patterns to test the accumulator itself or other modules in the circuit. The generation scheme (1) is described completely by its width n, initial state X_0 and the constant increment value a and can be compactly denoted by the triple (a, X_0, n).

An attractive method of evaluating the quality of the test pattern generator is to examine the state coverage that it provides. The state coverage gives the number of different patterns that can be produced and is an indication of the real capacity of the signals that appear on the output of the generator.

For an n-bit generator whose output bits are denoted as $b_{n-1}b_{n-2}\ldots b_1b_0$, consider the subspace formed by the k contiguous bits $b_{i+k-1}b_{i+k-2}\ldots b_{i+1}b_i$, denoted as S_i^k. Clearly, there are $n-k+1$ such subspaces of size k, for $i = 0, 1, \ldots, n-k$ (Fig. 2). The state coverage analysis will be with respect to such contiguous subspaces. Such an analysis is justified since data-path architectures have a strongly bit organized character and contain internal busses that are partitioned into physically adjacent lines.

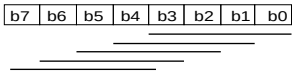


Figure 2: 4 bit contiguous subspaces for an 8 bit space

We say a sequence of n-bit vectors, $\mathcal{Q} = \{X_0, X_1, \ldots, X_{p-1}\}$, produced by the generator $\mathcal{A} = (a, X_0, n)$ under the scheme (1), exhaustively covers a specified k-bit contiguous subspace S_i^k, if all 2^k different patterns appear on the bit positions $b_{i+k-1}b_{i+k-2}\ldots b_{i+1}b_i$ defining S_i^k.

Fig. 3 shows how the state coverage for 12 and 14 bit subspaces increases with the number of patterns produced by the generator $\mathcal{A} = (13264529, 0, 32)$. The figure clearly shows how some subspaces are exhaustively covered fairly quickly while others (of the same size) require substantially more patterns. Note for instance that for $k = 14$, the subspaces $S_5^{14}, S_6^{14}, S_7^{14}, S_8^{14}$ and S_9^{14} get covered exhaustively within 32768 patterns while the subspace S_{15}^{14} requires as many as 1021199 patterns i.e. more than 30 times as many patterns.

In the following definitions, we provide a framework for evaluating different generators in terms of their ability to exhaustively cover subspaces consisting of contiguous bits.

Definition 1 *Let $P(S_i^k)$ denote the size of the smallest sequence (starting from X_0) that exhaustively covers the k-bit contiguous subspace, S_i^k. Then, the normalized number of patterns required to exhaustively cover S_i^k is defined as $\hat{P}(S_i^k) = P(S_i^k)/2^k$.*

Thus, $\hat{P}$ provides a means of comparing the efficiency of a generator in covering different subspaces independently of their size.

Definition 2 *The latency of the generator for k-bit subspaces is defined as*

a) $w_k = \max_{i=0}^{n-k}\{\hat{P}(S_i^k)\}$, in the worst case and

b) $v_k = \frac{1}{n-k+1}\sum_{i=0}^{n-k}\hat{P}(S_i^k)$, on average.

In other words, $w_k = q$ guarantees that all k-bit contiguous subspaces will be exhaustively covered in q normalized number of patterns.

Example *Consider the 4-bit sequence $\mathcal{Q} = \{0000, 1011, 0110, 0001, 1100, \ldots\}$ generated by $\mathcal{A} = (11, 0, 4)$. Here, the subspace S_0^2 consisting of bits b_1b_0 is exhaustively covered by the first four patterns in the sequence, while the subspaces S_1^2 and S_2^2 consisting of bits b_2b_1 and b_3b_2 respectively, require five patterns. Thus, $\hat{P}(S_0^2) = 4/4 = 1.0$ and $\hat{P}(S_1^2) = \hat{P}(S_2^2) = 5/4 = 1.25$. This indicates that S_0^2 can be exhaustively covered optimally, but S_1^2 and S_2^2 require 25% more patterns than the optimal. Now the worst case latency for the 2-bit subspaces is $w_2 = 1.25$ while the average latency is only $v_2 = (1+1.25+1.25)/3 = 1.167$.*

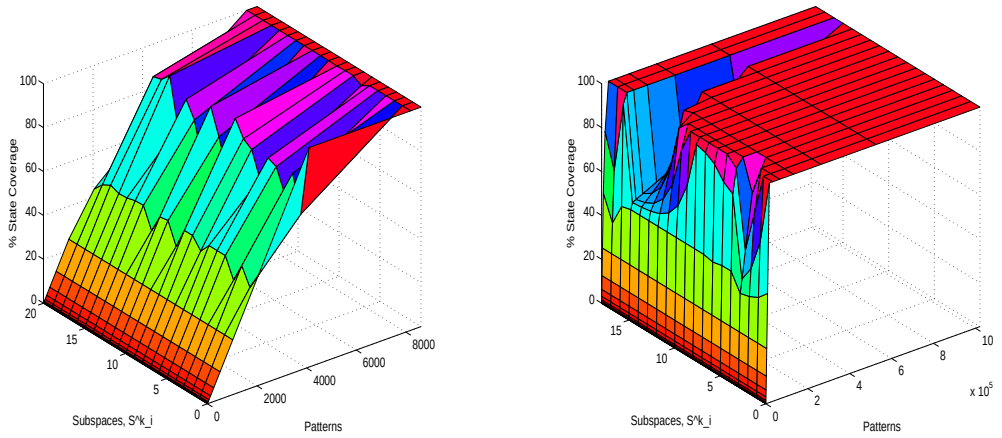
Theorem 1 *Given a generator $\mathcal{A}$, $w_k \leq 2w_{k+1}$ and $v_k \leq 2v_{k+1}$, for all k.*

Proof. Follows directly from the fact that $P(S_i^k) \leq P(S_i^{k+1})$ for all i and k. □

Definition 3 *We define several metrics to evaluate the quality of the generator $\mathcal{A} = (a, X_0, n)$ for subspaces of size r to s (both inclusive), $1 \leq r \leq s \leq n$, as follows*

a) The worst case latency $T(r, s)$, is given by

$$T(r, s) = \max_{k=r}^{s}\{w_k\} \tag{2}$$



(a) $k = 12$ (b) $k = 14$

Figure 3: State Coverage for the generator, $\mathcal{A} = (13264529, 0, 32)$

b) The average latency in the worst case $W(r,s)$, is defined as

$$W(r,s) = \frac{1}{s-r+1}\sum_{k=r}^{s} w_k \tag{3}$$

c) The average latency in the average case $V(r,s)$, is given by

$$V(r,s) = \frac{1}{s-r+1}\sum_{k=r}^{s} v_k \tag{4}$$

These metrics characterize different aspects of the behavior of the generator over the target range of subspace sizes. $T(r,s)$ describes the worst case behavior of the generator while the other metrics describe its average behavior. $T(r,s) = q$ implies that when $q2^j$ patterns have been produced, all subspaces of size j $(r \leq j \leq s)$ are exhaustively covered thus guaranteeing exhaustive coverage of all contiguous subspaces of size r to s within q times the optimal number of patterns for that size. $W(r,s)$ defines the average number of normalized patterns that need to be produced to exhaustively cover all contiguous subspaces of any size between r and s while $V(r,s)$ describes the average number of normalized patterns required to cover any subspace of any size between r and s.

For the metrics defined above, lower numbers imply better generators in the sense that such generators need to produce less number of patterns to exhaustively cover subspaces of different sizes, thus resulting in shorter test lengths for the same fault coverage. We can now state the problem as follows:

Problem Statement: Given the width of the generator n, and the sizes r, s $(1 \leq r \leq s \leq n)$, of the target subspaces, find values of a and X_0, for which the generator $\mathcal{A} = (a, X_0, n)$ is optimal under one of the metrics given in Definition 3. In other words, find that a, X_0 that minimizes the relevant metric. Such an optimal generator will be denoted as $\mathcal{A}_{r,s}$.

Clearly, no even number a, can make up a satisfactory generator since all subspaces of the form S_0^k will never be exhaustively covered. Also, if n is the width of the generator, then for all odd a, $\hat{P}(S_0^n) = 1$, since every odd number is relatively prime to 2^n. Thus, for $r = s = n$, every generator consisting of odd a is optimal. The search for optimal values of a can therefore, be restricted to odd numbers and for subspace sizes r and s such that $1 \leq r \leq s < n$.

3 Non-progressive Schemes

Non-progressive generators exhaustively cover only the k-bit contiguous subspaces optimally in 2^k steps, i.e. only $w_k = 1$. In other words, we restrict $r = s = k$ for the general problem statement above. Such schemes are called non-progressive since, as shown later, 2^{k-1} patterns need to be produced before even all the 1-bit subspaces are exhaustively covered.

Clearly, exhaustive coverage of k-bit subspaces also guarantees the exhaustive coverage of all subspaces of size less than k. However, from Theorem 1 the normalized number of steps required to exhaustively cover a j-bit subspace $(j < k)$ can, in the worst case, be as high as 2^{k-j}. Thus, for such schemes the optimal generator is of optimal quality only for the k-bit subspace, and the quality may deteriorate for other smaller sizes.

Theorem 2 *Given positive integers n and k $(k < n)$, the generator $\mathcal{A}_{k,k} = (a, 0, n)$ exhaustively covers all k-bit contiguous subspaces in exactly 2^k steps, i.e. $\hat{P}(S_i^k) = 1$, for all i, where a is given by*

$$a = \sum_{i=0}^{\lceil n/k \rceil - 1} 2^{ki} \tag{5}$$

Proof. The n-bit generator can be divided into $\lceil n/k \rceil$ disjoint subspaces (DS), each of which is k-bits wide, except for the last one. The constant a, has ones on bits $0, k, 2k, \ldots$ i.e. on the least significant position of each DS. Thus each step of the generation process is equivalent to adding 1 to each of the DSs. Since $X_0 = 0$, there is no carry out from bit position $ik-1$ to bit position ik before all 2^k patterns have been produced. Now consider a subspace that overlaps two adjacent DSs. It can easily be shown that this subspace consists of bits weighted from 0 to $k-1$ (Fig. 4), though the order depends on the relative position of the subspace with respect to the two adjacent DSs. Thus, the generator produces all 2^k patterns on the bits constituting the subspace in exactly 2^k steps. □

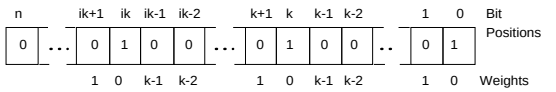


Figure 4: Non-progressive scheme

Theorem 3 *For the generator defined in Theorem 2, the worst case latency for i-bit subspaces is given by $w_i = 2^{-i}[2^{k-i}(2^i - 1) + 1]$, if $1 \leq i \leq k$.*

Proof. When $1 \leq i \leq k$, it can easily be shown that the i-bit subspace whose most significant bit is at position $jk-1$, for any j, requires the most patterns before it is exhaustively covered. These subspaces are exhaustively covered as soon as the all 1s pattern appears on the i bits. Since no carries are generated from position $jk-1$ to jk before 2^k steps, these subspaces are exhaustively covered (except for the initial pattern X_0) after $\sum_{j=0}^{i-1} 2^{k-1-j} = 2^{k-i}(2^i - 1)$ patterns, which concludes the proof. □

4 Progressive Schemes

In this section we will discuss generators which efficiently cover exhaustively all subspaces within a certain range of sizes. The selection of the required range of sizes depends, in general, on the width of the cones of logic that drive the outputs of the network under test. The upper limit of the range depends on the width of the widest cone of logic that needs to be tested and the lower limit depends on the minimum absolute number of test patterns that should be applied.

Using k generators, a scheme for which $w_i < 2\,(1 \leq i \leq k)$, can be defined as shown in the following theorem. An optimal (i.e. $T(1,k) = 1$) pseudo-exhaustive scheme was previously developed in [8]. That scheme, however, uses linear operations in $GF(2)$.

Theorem 4 *Given n and k $(k < n)$, the multiple generation scheme consisting of the k generators, $\mathcal{A}_{1,1}, \mathcal{A}_{2,2}, \ldots, \mathcal{A}_{i,i}, \ldots, \mathcal{A}_{k,k}$ used successively, requires at most 2 times the optimal number of patterns to exhaustively cover all subspaces of size 1 to k. The generators $\mathcal{A}_{i,i}$ are the optimal non-progressive generators as defined in Theorem 2 and are each used to generate the next 2^i patterns.*

Proof. From Theorem 2, it follows that the generator $\mathcal{A}_{i,i}$, exhaustively covers all i bit subspaces in exactly 2^i patterns. Since the $i-1$ previous generators have each generated $2^1, 2^2, \ldots, 2^{i-1}$ patterns respectively, a total of $2(2^i - 1)$ patterns have been generated when the i bit subspace is exhaustively covered. Thus, $w_i < 2$, for all i. □

For schemes using single generators, we will use the worst case latency $T(r,s)$ to evaluate the goodness of the generator. Clearly, this is the strongest criterion, in that for a generator with $T(r,s) = t$, we guarantee that all possible subspaces of size r to s will be exhaustively covered in at most t times the optimal number of patterns. Also, experimental data reveals that all the three metrics are very strongly correlated. Fig. 5 shows the values of the three metrics $T(1,n), W(1,n)$ and $V(1,n)$, for all generators of size 6 and 8 bits. Table 1 lists the correlation coefficients for $\rho(T(1,n), W(1,n))$ and $\rho(W(1,n), V(1,n))$ for all generators $\mathcal{A} = (a, 0, n)$ (a is odd and $8 \leq n \leq 16$). This data indicates that the goodness of a generator is

n	$\rho(T(1,n), W(1,n))$	$\rho(W(1,n), V(1,n))$
8	0.995207	0.987386
9	0.996179	0.985447
10	0.996674	0.983677
11	0.997054	0.982201
12	0.997299	0.980939
13	0.997480	0.979949
14	0.997645	0.979151
15	0.997755	0.978613
16	0.997836	0.978277

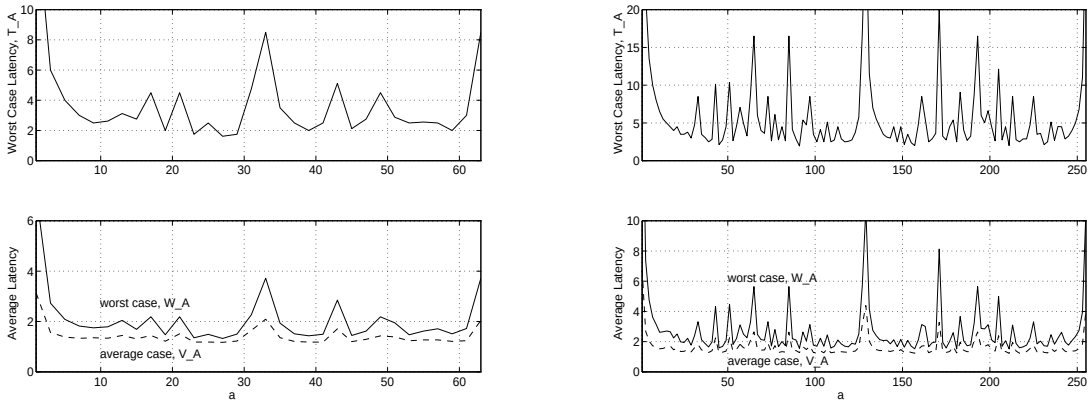
Table 1: Correlation coefficients for T, W and V

virtually independent of the specific metric used and leads to the following conjecture.

Conjecture 1 *The goodness of a generator $\mathcal{A}_{r,s} = (a, X_0, n)$, relative to other generators (with the same value of n, X_0), as determined by the worst case latency $T(r,s)$ is virtually similar to that determined by the average latency in the worst case $W(r,s)$, and the average latency in the average case $V(r,s)$.*

Thus, the search for good generators can be made with respect to any of the metrics.

Till now, it has been assumed that the initial value X_0 is equal to 0. This assumption is reasonable, since the impact of X_0 on the quality of the generator has



(a) $n = 6$ (b) $n = 8$

Figure 5: $T(1,n), W(1,n)$ and $V(1,n)$ vs. $\mathcal{A} = (a, 0, n)$

been observed to be greatest for the smallest size subspaces and diminishes rapidly for larger sizes. Also, experimental data reveals that the worst case latency T of a generator for the optimal value of X_0 is only marginally better than when $X_0 = 0$. In particular, Fig. 6 shows the impact of X_0 on the worst case latency, $T(1,n)$, of the generators $\mathcal{A} = (91, X_0, 8)$ and $\mathcal{A} = (467, X_0, 10)$.

Thus, we have the following conjecture.

Conjecture 2 *The goodness of the generator $\mathcal{A}_{r,s} = (a, X_0, n)$ for the optimal value of X_0 is only marginally better than for the value $X_0 = 0$.*

Corollary *Given n, the search for the optimum value of a and X_0 is orthogonal. Thus, the best generator can be located by first searching for the optimum value of a with $X_0 = 0$ and then using that value of a to search for the optimum value of X_0.*

The non-linearity of the addition operation makes a theoretical study of this generation scheme complex. Therefore, the next section presents the results obtained from extensive experiments to identify the best generators for a variety of generator widths.

5 Experimental Results

Extensive experiments have been conducted to identify the best generators for different widths and for different ranges of subspace sizes.

For $4 \le n \le 16$ the best generators $\mathcal{A}_{r,s} = (a, 0, n)$ have been identified for all subspaces of size r to s by exhaustive search of the entire solution space (i.e. by examining all odd $a < 2^n$). For $17 \le n \le 32$, an exhaustive search becomes infeasible. Hence, upto 50,000 values of a were randomly sampled for each n in order to identify the good generators. For these cases, the largest size subspaces that have been considered are 16 bits wide. In all experiments, it is assumed that the initial value $X_0 = 0$.

The best values of $T(r,s)$ and the corresponding values of a for different values of r, s are presented in Tables 2, 3 and 4. Only selected results are presented here; further details can be found in [9].

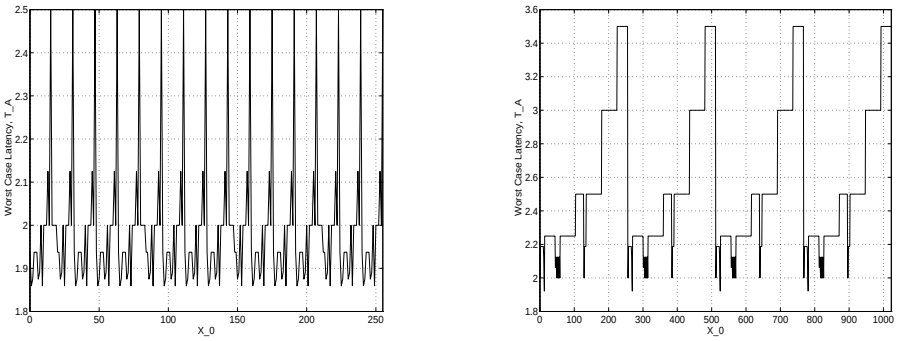
$T(r,s)$ values have been rounded up and are listed in the tables as $[t]$, while the values of a are listed in hexadecimal notation. For instance, the value of $T(5,8)$ for $n = 15$ is listed in the upper triangular half of Table 3 in the column labeled 5 and the third row of the group of rows labeled 8 and is equal to 2.079. The value of $\mathcal{A}_{5,8}$ is listed below that as 214F (hex).

s	n	r = 1	r = 2	r = 3	r = 4
5	7	[1.875] 5B	[1.750] 13	[1.688] 69	[1.469] 31
4	7	[1.875] 5B	[1.625] 1B	[1.438] 27	
	6	[1.625] 1B	[1.500] 25	[1.438] 19	
3	7	[1.625] 5B	[1.500] 25		
	6	[1.625] 1B	[1.500] 25		
	5	[1.625] 1B	[1.500] 5		
2	7	[1.500] 2D			
	6	[1.500] 15			
	5	[1.500] D			
	4	[1.500] 5			

Table 2: Best generators for $4 \le n \le 7$

It should be noted that values of a for $r = s$ are not listed since these can be synthesized using Theorem 2. Also, since $w_n = 1$ for all a, $T(r,n) = T(r,n-1)$. Thus, $\mathcal{A}_{r,n} = \mathcal{A}_{r,n-1}$. This implies that the the best generator for any range of subspace sizes can be obtained by only evaluating subspaces upto size $n-1$.

From the experimental data for $4 \le n \le 16$, it was observed that $\mathcal{A}_{r,n-1} = \mathcal{A}_{r,n-2}$, for all $r \le n-3$ i.e. the best generator for subspace sizes r to $n-2$



(a) $n = 8$, $a = 91$ (b) $n = 10$, $a = 467$

Figure 6: Impact of X_0 on the quality, $T(1, n)$ of the generators $\mathcal{A} = (a, X_0, n)$

is also the best generator for sizes r to $n-1$. It was also seen that the best generator $\mathcal{A}_{n-2,n-1}$, is always 3. Thus in the tables for $4 \leq n \leq 16$, the values for $\mathcal{A}_{r,n}, \mathcal{A}_{r,n-1}, \mathcal{A}_{n-2,n}$ and $\mathcal{A}_{n-2,n-1}$ are not listed.

The tables for $4 \leq n \leq 16$ show that the value of $T(1, n)$ increases from a low of 1.5 (for $n = 4$) to a high of 2.688 (for $n = 16$). The values for $T(r, s)$ for any other r, s are lower than these limits. Similarly, the value of $T(5, 16)$ increases to a maximum of 3.829 for $n = 32$. Thus, for a data-path of width n, if subspaces of sizes r to s are to be targeted, the best value of $\mathcal{A} = (a, 0, n)$ can be located from the relevant tables. The corresponding values of $T(r, s) = t$ then imply that as the number of patterns produced increases from $t \cdot 2^r$ to $t \cdot 2^s$ (subject to a maximum of 2^n), the sizes of the subspaces that are exhaustively covered increases progressively from r to s. Thus for a 32-bit data-path, only $3.829 \cdot 2^j$ patterns have to be produced to exhaustively cover a subspace of size j, $5 \leq j \leq 16$, while correspondingly for a 16-bit data-path only $2.329 \cdot 2^j$ patterns are needed.

6 Conclusions

In this paper we have demonstrated a completely original scheme for generating test patterns in data-path architectures having adders. To the best of our knowledge, this is the first time arithmetic units have been considered as sources of pseudo-exhaustive test patterns. We have shown that the sequence of patterns generated by continuously accumulating a constant value is an effective source of high quality parallel test patterns in terms of their pseudo-exhaustive state coverage on subspaces of contiguous bits. Such generators exhaustively cover all the contiguous bit subspaces in close to the optimal number of patterns. The selection of the range of sizes of subspaces that must be exhaustively covered then provides a flexible framework for tradeoffs between test application time and test quality. Since this test generation technique uses existing adders it has no impact on the area and performance of the circuit. The compaction scheme of [3] also uses existing accumulators to compact test responses. Thus, the arithmetic and logic units or even simple adders found in datapath architectures can be leveraged as test pattern generators and test response compactors to provide the key elements of a BIST strategy that results in at-speed testing with no area overhead or performance degradation.

Acknowledgments

The authors would like to thank Mr. Kevin Malakoff for providing some experimental results.

References

[1] P.H. Bardell, W.H. McAnney, and J. Savir. *Built-in Test for VLSI: Pseudorandom Techniques.* John Wiley & Sons, 1987.

[2] B. Koenemann, J. Mucha, and G. Zwiehoff. Built-in test for complex integrated circuits. *IEEE J. Solid State Circuits*, SC-15:315–318, June 1980.

[3] J. Rajski and J. Tyszer. Accumulator-based compaction of test responses. *IEEE Tr. Comp.*, 42(6):643–450, June 1993.

[4] P.D. Hortensius et al. Cellular-automata-based pseudo-random number generators for built-in self test. *IEEE Tr. CAD*, 7(8):842–859, Aug. 1989.

[5] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 2nd edition, 1981.

[6] D.T. Tang and L.S. Woo. Exhaustive test pattern generation with constant weight vectors. *IEEE Tr. Comp.*, 32(12):1145–1150, Dec. 1983.

[7] E.J. McCluskey. Verification testing - a pseudoexhaustive test technique. *IEEE Tr. Comp.*, 33(6):541–546, June 1984.

[8] J. Rajski and J. Tyszer. Recursive pseudo-exhaustive test pattern generation. *IEEE Tr. Comp.*, 42(12):1517–1521, Dec. 1993.

[9] S. Gupta, J. Rajski, and J. Tyszer. Arithmetic generators of pseudo-exhaustive test patterns. Submitted to *IEEE Tr. Comp.*

s	n	r = 1	2	3	4	5	6	7	8	9	10	11	12	13
14	16	[2.688] 92ED	[2.688] 92ED	[2.688] 92ED	[2.396] AF61	[2.329] 26C1	[2.329] 26C1	[2.117] A033	[2.106] ACC9	[2.000] 67FD	[1.963] B509	[1.778] 4009	[1.600] 8005	[1.455] 5559
13	16	[2.688] 92ED	[2.625] EC8B	[2.590] 1375	[2.396] AF61	[2.329] 26C1	[2.329] 26C1	[2.117] A033	[2.029] C26B	[2.000] 67FD	[1.880] CDE9	[1.601] 6AA9	[1.455] 4445	
	15	[2.590] 1375	[2.590] 1375	[2.590] 1375	[2.282] 2F61	[2.176] 26C1	[2.120] 127F	[2.104] 6B6B	[1.997] 3403	[1.962] E4B	[1.784] 2CCB	[1.601] 6AA9	[1.455] 4445	
12	16	[2.688] 92ED	[2.625] 62CB	[2.590] 1375	[2.396] AF61	[2.329] 26C1	[2.258] E91	[2.042] D6E1	[1.997] 3403	[1.882] 155B	[1.601] 1FFB	[1.489] A131		
	15	[2.590] 1375	[2.590] 1375	[2.590] 1375	[2.282] 2F61	[2.176] 26C1	[2.120] 127F	[2.042] 56E1	[1.997] 3403	[1.882] 155B	[1.601] 1FFB	[1.455] 1D89		
	14	[2.500] 1375	[2.500] 1375	[2.391] 2BD1	[2.102] 109F	[2.102] 109F	[2.102] 109F	[2.000] 1575	[1.963] 1ED3	[1.780] 1491	[1.600] 2005	[1.455] 1D89		
11	16	[2.688] 92ED	[2.625] 62CB	[2.547] 6791	[2.282] 509F	[2.282] 509F	[2.120] 127F	[2.000] 6603	[1.882] E927	[1.615] 3AA9	[1.489] 7C59		[1.500] 555	12
	15	[2.590] 1375	[2.590] 1375	[2.547] 6791	[2.172] 20F5	[2.172] 20F5	[2.120] 127F	[2.000] 19FD	[1.882] 6927	[1.601] 4557	[1.489] 3A7		[1.500] 2D5	11
	14	[2.500] 1375	[2.500] 1375	[2.391] 2BD1	[2.102] 109F	[2.102] 109F	[2.020] A71	[2.000] 1575	[1.882] 2927	[1.601] 557	[1.455] B		[1.500] 155	10
	13	[2.500] 1375	[2.500] 1375	[2.391] BD1	[2.102] 109F	[2.079] 114F	[1.985] D03	[1.946] D69	[1.778] 7F7	[1.601] 557	[1.455] B		[1.500] B5	9
10	16	[2.657] AEE7	[2.625] 62CB	[2.547] 6791	[2.282] 509F	[2.211] F64B	[2.032] 8189	[1.950] 3663	[1.600] 8805	[1.498] AB		[1.500] 5A5	[1.625] 6DB	12
	15	[2.590] 1375	[2.590] 1375	[2.532] 6989	[2.172] 20F5	[2.157] 193F	[1.985] 4D03	[1.874] 7663	[1.600] 805	[1.490] 3E5D		[1.500] 5A5	[1.625] 2DB	11
	14	[2.500] 1375	[2.500] 1375	[2.391] 2BD1	[2.102] 109F	[2.084] 314F	[1.985] D03	[1.874] 3663	[1.600] 805	[1.490] 1A3		[1.500] 1A5	[1.625] 2DB	10
	13	[2.500] 1375	[2.500] 1375	[2.391] BD1	[2.102] 109F	[2.079] 114F	[1.985] D03	[1.874] 1663	[1.600] 805	[1.457] 5A5		[1.500] 1A5	[1.625] DB	9
9	16	[2.657] AEE7	[2.375] 51D3	[2.375] 2E2D	[2.172] 20F5	[2.172] 20F5	[1.985] 4281	[1.727] 6A9	[1.499] 301			[1.500] A5	[1.625] 5B	8
	15	[2.590] 1375	[2.375] 51D3	[2.375] 2E2D	[2.172] 20F5	[2.118] 3E0B	[1.985] 3D7F	[1.727] 6A9	[1.499] 301					
	14	[2.500] 1375	[2.375] 11D3	[2.375] 11D3	[2.102] 109F	[2.084] 314F	[1.946] 22D	[1.633] 6A9	[1.491] 7CD		[1.688] 169	[1.750] 2D3	[2.000] 2D3	12
	13	[2.500] 1375	[2.375] 11D3	[2.375] 5E9	[2.102] 109F	[2.000] 1983	[1.922] 34D	[1.633] 6A9	[1.491] 7CD		[1.625] D9	[1.750] 169	[2.000] 1A7	11
8	16	[2.465] 5AE7	[2.250] 3769	[2.125] 3769	[2.125] 3769	[2.118] 41F5	[1.891] 36A5	[1.602] 1E05			[1.625] D9	[1.750] 97	[2.000] D3	10
	15	[2.465] 1AE7	[2.250] 3769	[2.125] 3769	[2.125] 3769	[2.079] 214F	[1.602] 1E05	[1.497] 3E7F			[1.625] D9	[1.750] 97	[2.000] 5B	9
	14	[2.465] 1AE7	[2.250] 3769	[2.125] 3769	[2.102] 109F	[1.891] 36A5	[1.602] 1E05	[1.497] 181			[1.500] D9	[1.625] E5	[1.938] 5B	8
	13	[2.465] 1AE7	[2.250] 1769	[2.125] 1769	[2.102] 109F	[1.875] 1AF7	[1.602] 1FB	[1.489] 1FD5		[1.594] 285	[1.813] 4D9	[1.813] 4D9	[2.000] 9D3	12
7	16	[2.438] 5AE7	[2.250] 3769	[2.125] 3769	[1.938] 50A1	[1.922] 50A1	[1.524] C815			[1.594] 85	[1.813] 463	[1.813] 4D9	[2.000] 1D3	11
	15	[2.438] 1AE7	[2.250] 3769	[2.125] 3769	[1.938] 10A1	[1.922] 50A1	[1.524] 4815			[1.594] 85	[1.750] D9	[1.750] D9	[2.000] 1D3	10
	14	[2.438] 1AE7	[2.063] 3727	[2.063] 8D9	[1.938] 10A1	[1.860] 1F3	[1.524] 815			[1.500] D1	[1.750] D9	[1.750] D9	[2.000] 5B	9
	13	[2.438] 1AE7	[2.063] 1727	[2.063] 8D9	[1.938] F5F	[1.829] AF7	[1.500] F41			[1.469] 31	[1.688] 69	[1.750] 69	[1.938] 5B	8
6	16	[2.188] CE9D	[2.188] 51D3	[2.125] 3769	[1.875] 477B	[1.672] 17A1			[1.594] 885	[1.594] 885	[2.000] 1D3	[2.000] 1D3	[2.188] 69D	12
	15	[2.188] 4E9D	[2.188] 11D3	[2.125] 2E2D	[1.875] 3885	[1.672] 17A1			[1.485] 39F	[1.594] 85	[2.000] 1D3	[2.000] 1D3	[2.000] 1D3	11
	14	[2.188] E9D	[2.063] 3727	[2.063] 8D9	[1.594] 1885	[1.594] 1885			[1.485] 61	[1.594] 85	[1.813] D9	[1.813] D9	[2.000] 1D3	10
	13	[2.188] 69D	[2.063] 1727	[2.063] 8D9	[1.594] 1885	[1.594] 1885			[1.454] 1F5	[1.594] 85	[1.750] D9	[1.750] D9	[2.000] 1D3	9
5	16	[2.000] 39D3	[1.813] 64D9	[1.813] 64D9	[1.594] 4285				[1.438] F5	[1.594] 85	[1.750] D9	[1.750] D9	[1.938] 5B	8
	15	[2.000] 39D3	[1.813] 64D9	[1.813] C63	[1.594] 1885			[1.493] C1	[1.829] 3A9	[1.938] A1	[2.063] 727	[2.063] 727	[2.438] AE7	12
	14	[2.000] E5B	[1.813] 24D9	[1.813] 4D9	[1.594] 285			[1.493] C1	[1.610] 457	[1.938] A1	[2.063] D9	[2.063] 727	[2.375] 1D3	11
	13	[2.000] E5B	[1.813] 4D9	[1.813] 4D9	[1.594] 285			[1.461] 73	[1.610] 57	[1.813] D9	[1.813] D9	[1.813] D9	[2.000] 1D3	10
4	16	[2.000] 39D3	[1.750] 25A7	[1.688] 3769				[1.461] 73	[1.610] 57	[1.750] D9	[1.750] D9	[1.750] D9	[2.000] 1D3	9
	15	[2.000] 1697	[1.750] 12D3	[1.688] 969			[1.489] 2B	[1.602] 1FB	[1.875] AF7	[2.102] 9F	[2.125] 769	[2.250] 769	[2.438] AE7	12
	14	[2.000] E5B	[1.750] 969	[1.688] 969			[1.458] 123	[1.602] 1FB	[1.829] 2F7	[1.938] 343	[2.125] 769	[2.125] 769	[2.375] 1D3	11
	13	[2.000] 5A7	[1.750] 5A7	[1.688] 969			[1.454] 2DD	[1.594] 205	[1.750] 109	[1.938] 343	[2.000] 1D3	[2.000] 1D3	[2.000] 1D3	10
3	16	[1.625] B6DB	[1.500] A5A5			[1.454] 559	[1.606] 6A9	[1.868] 22D	[1.954] E0B	[2.102] 9F	[2.188] C87	[2.375] 1D3	[2.500] 375	12
	15	[1.625] 36DB	[1.500] 25A5			[1.454] 559	[1.606] 6A9	[1.829] 691	[1.915] 6F9	[1.938] 343	[2.157] 631	[2.375] 1D3	[2.375] 1D3	11
	14	[1.625] 16DB	[1.500] 25A5		[1.457] 5A5	[1.600] 805	[1.774] CB	[1.926] 79B	[2.000] 60B	[2.102] 9F	[2.188] C87	[2.500] 375	[2.500] 375	12
	13	[1.625] 16DB	[1.500] 5A5											
		9	8	7	6	5	4	3	2	1				
		r											n	s

Table 3: Best generators for $8 \leq n \leq 16$

s	n	r = 5	r = 6	r = 7	r = 8	r = 9	r = 10	r = 11	r = 12	r = 13	r = 14	r = 15
16	32	[3.829] F5977DDB	[3.598] 8FDB079	[3.598] 8FDB079	[3.393] D36D3665	[3.393] D36D3665	[3.089] F5977DDB	[2.851] 95DED05	[2.851] 95DED05	[2.455] ABF9F71	[2.156] E54FCF03	[1.981] 514EB17
	31	[3.829] 75977DDB	[3.598] 8FDB079	[3.447] 16D7F351	[3.393] 536D3665	[3.142] 8BAB005	[3.089] 75977DDB	[2.851] 95DED05	[2.783] 1766BD89	[2.455] ABF9F71	[2.156] 654FCF03	[1.918] 5D8DD62D
15	32	[3.829] F5977DDB	[3.598] 8FDB079	[3.598] 8FDB079	[3.393] D36D3665	[3.196] D07FB05D	[2.946] 564FB7E7	[2.838] D07FB05D	[2.641] AA3C6E2B	[2.368] 622A03ED	[2.000] F8DF313B	
	31	[3.829] 75977DDB	[3.598] 8FDB079	[3.447] 16D7F351	[3.393] 536D3665	[3.142] 8BAB005	[2.946] 564FB7E7	[2.785] 145D51A1	[2.476] 33BF1299	[2.245] 1A9722A9	[2.000] 78DF313B	
14	32	[3.829] F5977DDB	[3.598] 8FDB079	[3.380] 8106F251	[3.371] 4BFCA057	[3.105] D4BD3983	[2.733] B583D4F1	[2.580] DB0C6F0B	[2.397] DB0C6F0B	[2.000] C94BBFFF		
	31	[3.829] 75977DDB	[3.598] 8FDB079	[3.380] 106F251	[3.321] 3DBAB9D9	[3.105] 54BD3983	[2.733] 3583D4F1	[2.580] 5B0C6F0B	[2.397] 5B0C6F0B	[1.916] 4BFCCCDB	[1.969] 11F8228D	30 6
13	32	[3.782] 8122A51B	[3.598] 8FDB079	[3.250] 610B5C0D	[3.133] 4B91764D	[2.794] 4B1D8423	[2.536] 65CAFFD7	[2.399] D112003B	[1.990] E8B8BC19		[1.969] 11F8228D	29
	31	[3.563] 273C08CB	[3.028] 273C08CB	[3.028] 273C08CB	[3.028] 273C08CB	[2.633] 4B91764D	[2.536] 65CAFFD7	[2.399] 5112003B	[1.990] 68B8BC19	[2.055] DFBF83	[2.469] CA06F75	30 7
12	32	[3.557] 43D5C1A9	[3.557] 43D5C1A9	[2.905] A4C201B9	[2.905] A4C201B9	[2.520] 4B91764D	[2.517] B343F77F	[1.988] 5C1553D7		[2.000] 187CC07F	[2.469] CA06F75	29
	31	[3.557] 43D5C1A9	[2.903] 273C08CB	[2.903] 273C08CB	[2.826] 763DCF71	[2.520] 4B91764D	[2.517] 3343F77F	[1.988] 5C1553D7	[1.969] 1A9AF957	[2.219] 30740ADD	[2.594] 11A4E615	30 8
11	32	[3.547] 43D5C1A9	[3.524] 21DD4E77	[2.905] A4C201B9	[2.618] D4274D3D	[2.287] 30F8DFF9	[2.000] 86400801		[1.954] 1A9AF957	[2.219] 10740ADD	[2.594] 11A4E615	29
	31	[3.520] 447EE427	[2.903] 273C08CB	[2.903] 273C08CB	[2.618] 54274D3D	[2.287] 30F8DFF9	[1.990] 4D9C73A3	[2.071] 135F55FF	[2.344] 2F0738B7	[2.641] 1F437F35	[2.747] 374F9F83	30 9
10	32	[2.973] EFC83B9B	[2.973] EFC83B9B	[2.805] C3B34EB1	[2.497] 80A03359	[1.993] CA008955		[2.016] 1EC88AA5	[2.344] F0738B7	[2.641] 1F437F35	[2.747] 174F9F83	29
	31	[2.973] 6FC83B9B	[2.903] 273C08CB	[2.805] 43B34EB1	[2.477] 4C15B7FB	[1.993] 4A008955	[1.993] A008955	[2.477] C15B7FB	[2.711] 1C075641	[2.857] 374F9F83	[2.857] 374F9F83	30 10
9	32	[2.973] EFC83B9B	[2.672] 61D66B17	[2.614] BE9D2EA7	[2.073] 58197AF1		[1.993] A008955	[2.469] 1887AFF	[2.469] 1887AFF	[2.857] 174F9F83	[2.857] 174F9F83	29
	31	[2.875] 32B1EF5F	[2.641] 2EE2FE03	[2.356] 1F7EBAA5	[2.071] 135F55FF	[1.990] D9C73A3	[2.287] 30F8DFF9	[2.618] 14274D3D	[2.857] 374F9F83	[2.857] 374F9F83	[2.857] 374F9F83	30 11
8	32	[2.875] 2698CF59	[2.485] 14A991B5	[1.985] 9A9AF957		[1.990] D9C73A3	[2.188] 7A53BB1	[2.563] 7A53BB1	[2.857] 174F9F83	[2.857] 174F9F83	[2.857] 174F9F83	29
	31	[2.610] 7DCD9F5	[2.485] 14A991B5	[1.969] 1A9AF957	[1.988] 1C1553D7	[2.517] 3343F77F	[2.520] B91764D	[2.826] 363DCF71	[2.857] 374F9F83	[2.857] 374F9F83	[2.857] 374F9F83	30 12
7	32	[2.500] EF3967A9	[2.149] CAE32543		[1.980] 5BD0C19	[2.334] 1E15A57F	[2.520] B91764D	[2.826] 163DCF71	[2.857] 174F9F83	[2.857] 174F9F83	[2.857] 174F9F83	29
	31	[2.469] CA06F75	[2.149] 4AE32543	[1.966] 48E39CB	[2.271] 222A01B5	[2.536] 25CAFFD7	[2.633] B91764D	[3.028] 273C08CB	[3.028] 273C08CB	[3.028] 273C08CB	[3.563] 273C08CB	30 13
6	32	[2.094] 3C0F3617		[1.956] 21F7263	[2.271] 22A01B5	[2.536] 5CAFFD7	[2.633] B91764D	[3.028] 73C08CB	[3.028] 73C08CB	[3.028] 73C08CB	[3.563] 73C08CB	29
	31	[1.969] 11F8228D	[1.916] BFCCCDB	[2.244] 38A508A5	[2.271] 222A01B5	[2.721] 392C4A43	[2.876] 23F1C995	[3.187] 3478A785	[3.380] 106F251	[3.598] 8FDB079	[3.829] 35977DDB	30 14
			[1.916] BFCCCDB	[2.174] A6B4A4B	[2.271] 22A01B5	[2.716] 978ED7F	[2.758] 31C19E9	[3.059] 31C19E9	[3.236] 15483B9B	[3.598] 8FDB079	[3.829] 15977DDB	29
		[1.980] 228314BF	[2.245] 1A9722A9	[2.434] AF964D	[2.744] 36AE5B03	[2.946] 164FB7E7	[3.142] 8BAB005	[3.393] 136D3665	[3.447] 16D7F351	[3.598] 8FDB079	[3.829] 35977DDB	30 15
		[1.964] 202FE13	[2.139] 1A9722A9	[2.402] 134AB615	[2.744] 16AE5B03	[2.946] 164FB7E7	[3.142] 8BAB005	[3.254] 15483B9B	[3.254] 15483B9B	[3.598] 8FDB079	[3.829] 15977DDB	29
		[1.914] 1D8DD62D	[2.126] 68D2697	[2.363] 68D2697	[2.553] 94A18D5	[2.851] 95DED05	[3.088] 7CBA9E5	[3.142] 8BAB005	[3.393] 136D3665	[3.447] 16D7F351	[3.598] 8FDB079	[3.829] 35977DDB	30 16
		[1.818] 737C467	[2.053] 9B9C41	[2.346] 17E6E38F	[2.553] 94A18D5	[2.851] 95DED05	[3.015] 35CD327	[3.142] 8BAB005	[3.393] 136D3665	[3.447] 16D7F351	[3.598] 8FDB079	[3.829] 15977DDB	29
		15	14	13	12	11	10	9	8	7	6	5	n	s
		r												

Table 4: Good generators for $29 \leq n \leq 32$