A Fast Signature Computation Algorithm for LFSR and MISR

Bin-Hong Lin, Shao-Hui Shieh, and Cheng-Wen Wu, *Senior Member, IEEE*

***Abstract*—A multiple-input signature register (MISR) computation algorithm for fast signature simulation is proposed. Based on the table look-up linear compaction algorithm and the modularity property of a single-input signature register (SISR), some new accelerating schemes—partial-input look-up tables and flying-state look-up tables—are developed to boost the signature computation speed. Mathematical analysis and simulation results show that this algorithm has an order of magnitude speedup without extra memory requirement compared with the original linear compaction algorithm. Though this algorithm is derived for SISR, a simple conversion scheme exists that can convert internal-EXOR MISR to SISR. Consequently, fast MISR signature computation can be done.**

***Index Terms*—Built-in self-test, fault simulation, LFSR, logic testing, MISR, signature analysis.**

I. Introduction

PSEUDORANDOM pattern based built-in self-test (BIST) has been a very popular test methodology for VLSI circuits due to its simplicity and effectiveness [1], [2]. Associated with the scan technique using internal flip-flops (FFs), BIST is considered one of the most powerful schemes for fault detection and classification. A general BIST configuration, as shown in Fig. 1, is composed of a pattern generator (PG), a signature analyzer (SA), and the circuit under test (CUT). During the BIST mode, test patterns from the PG are applied to the CUT for detecting possible faults, and the CUTs outputs are logged in the SA in a compressed form called the *signature*, which is then compared with that of a fault-free circuit to determine the CUTs correctness. *Aliasing* is said to exist if a faulty CUTs signature is the same as that of the fault-free circuit, though their output sequences are different.

An effective BIST design should have high fault coverage and low aliasing probability. Fault coverage evaluation is normally done by fault simulation, which estimates the percentage of all the faults considered that can be detected by the test patterns generated by the PG, assuming that we can observe the CUT outputs directly. In the BIST environment, however, we do not read the CUT outputs directly. Instead, we compress the outputs and read the final signature from the SA for comparison. Therefore, in addition to the fault coverage, the evaluation of the aliasing probability also is required. Aliasing probability estimation is usually done by probabilistic analysis [1], [3], based on assumptions that may not be realistic, such as uniform distribution of the faulty-output sequences, equal likelihood of the error patterns, and the application of all patterns from the PG. Moreover, a deviation of the actual design from the standard linear-feedback-shift-register (LFSR) implementation or a change in the input patterns (e.g., weighted patterns, multi-seeded patterns, embedded patterns, etc.) greatly complicates the analysis, and it is very difficult, if possible at all, to develop a design aid using this approach. An alternative practical approach is *BIST simulation*, which is a combination of fault simulation and signature computation. The research on fault simulation is relatively mature (see, e.g., [4]–[16]), and only combinational fault simulation is required for BIST simulation. On the other hand, the sequential nature of the SA makes the signature computation a much more time-consuming process and is the bottleneck of the BIST simulation [17], [18], i.e., fast BIST simulation relies on fast signature computation. In addition to aliasing prediction, finding the good signature of the circuit itself is also very important for the design and test engineers. Previous works extend the bit-by-bit (sequential) signature computation into multiple time-frame (multiple bit) computation to reduce the computation time [17], [19], [20]. Their basic idea is to decompose the state sequence into several independent subsequences, and then process the individual subsequences via state look-up tables (LUTs) for fast evaluation, followed by super-positioning of the results of individual responses with the input sequence. In their methods, the time-consuming sequential computation of the signature is replaced by table look-up operations, which can be done several bits at a time. A parallel algorithm based on a multi-processor environment was later developed to speed up the LFSR signature computation [21]. The major drawback of the parallel approach is the resource and communication overhead. Recently, in [22], a general *linear compaction algorithm* using superposition on the partial results of both the input partitions and state partitions by two individual sets of LUTs was proposed and shown to outperform the previous ones in speed. However, further speedup of the approach is limited by the excessive use of memory.

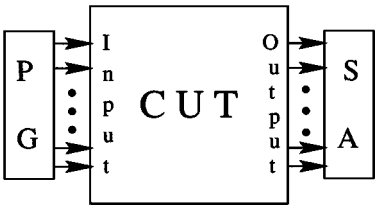

Fig. 1. A general BIST configuration.

In this paper, we propose several speedup techniques for LFSR signature computation. Based on the modulus property

Manuscript received June 23, 1999; revised January 13, 2000. This paper was recommended by Associate Editor K.-T. Cheng.

The authors are with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu 300, Taiwan (e-mail: cww@ee.nthu.edu.tw).

Publisher Item Identifier S 0278-0070(00)07480-7.

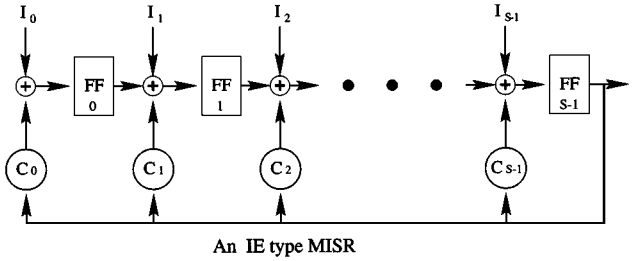

Fig. 2. A typical MISR configuration.

of the single-input signature register (SISR) or multiple-input signature register (MISR), we propose two partial-input LUT methods. The first uses the memory-driven approach to gain a certain speedup with less memory requirement as compared with the linear compaction algorithm. The second uses the timing-driven approach to gain a higher speedup with the same memory requirement as the linear compaction algorithm. Significant speedup can be achieved by exploring the sparsity of the error-domain inputs, i.e., long runs of contiguous zeros in the error syndromes from the CUT. The idea is to skip long runs of zeros in the error-domain input sequence in one iteration. Operating on the error-domain inputs, our flying-state LUT method can significantly reduce the signature computation time with an appropriate stride. However, the speedup of the method may be small if the stride is not properly selected. Consequently, a modification of the flying-state method using the pivot-checking (PC) technique is proposed to guarantee the speedup irrespective of the stride. Since the sparsity of the error-domain inputs depends on the faults of the CUT, we finally propose a dynamic pivot-checking (DPC) technique, which may stretch the pivot length dynamically according to the sparsity. It improves significantly the speedup in a general situation. As a result, our signature computation algorithm using the memory-driven approach and the (dynamic) PC technique is fast and economical in terms of memory space. Mathematical analysis and simulation results on ISCAS85 benchmark circuits show that, with the same memory requirement, this algorithm has an order of magnitude speedup over the linear compaction algorithm.

The rest of the paper is organized as follows. In Section II, we briefly review the linear compaction algorithm. We propose several improved MISR simulation algorithms in Section III, followed by their complexity analysis in Section IV. Experimental results are presented and discussed in Section V. Finally, Section VI concludes this work.

II. Preliminaries

Figure 2 shows a typical MISR configuration—the internal-EXOR MISR (IE-MISR). In the figure, s denotes the length of the MISR, i.e., the number of FFs in the register. Also, $C_0, C_1, \ldots, C_{s-1}$ are the binary coefficients of the characteristic polynomial of the MISR, i.e.,

$$C(x) = x^s + C_{s-1}x^{s-1} + \cdots + C_1 x + C_0. \tag{1}$$

Let r_i be the content of the i-th FF, then the state of the MISR (i.e., the sequence $r_0, r_1, \ldots, r_{s-1}$) can be represented by the state polynomial

$$S(x) = r_{s-1}x^{s-1} + \cdots + r_1 x + r_0. \tag{2}$$

Similarly, an m-bit input sequence $i_0, i_2, \ldots, i_{m-1}$ can be represented by the input polynomial.

$$I(x) = i_{m-1}x^{m-1} + \cdots + i_1 x + i_0 \tag{3}$$

The *signature* obtained by any SISR or MISR is defined as the final state of the register after the input sequence $I(x)$ has been entered into the register. Consider an IE-SISR with a characteristic polynomial $C(x)$. The signature obtained after the input bit stream ($I(x)$) is entered can be represented as [2]

$$S(x) = I(x) \bmod C(x) \tag{4}$$

i.e., the signature is equivalent to the remainder taken from the division of $I(x)$ by $C(x)$. Equation (4) generally provides an easy way for signature computation and aliasing probability analysis. Since an IE-MISR can be reduced to an equivalent SISR, it is a widely used technique to simulate a MISR by its corresponding SISR form. Parallel simulation for external-EXOR signature registers can be found in [21].

To reduce memory requirement during signature computation, the state and input polynomials can be partitioned into subpolynomials of equal length [22]. Specifically, for the s-bit state polynomial, we let

$$S(x) = \sum_{j=1}^{\frac{s}{l}} S_{\mathrm{sta}_j}(x) \tag{5}$$

where

$$S_{\mathrm{sta}_j}(x) \equiv r_{s-(j-1)l-1}x^{s-(j-1)l-1} + \cdots + r_{s-jl}x^{s-jl} \tag{6}$$

is the j-th partition (with length l) of the state polynomial, assuming s is a multiple of l. Similarly, the m-bit input sequence $I(x)$ can be partitioned into k-bit subpolynomials, i.e.,

$$I(x) = \sum_{i=1}^{\frac{m}{k}} I_{\mathrm{inp}_i}(x) \tag{7}$$

where

$$I_{\mathrm{inp}_i}(x) \equiv i_{m-(i-1)k-1}x^{m-(i-1)k-1} + \cdots + i_{m-ik}x^{m-ik} \tag{8}$$

is the i-th partition of the input polynomial and m is assumed to be a multiple of k.

Consider a linear compactor with initial state polynomial $S_0(x)$. Let $S^{+m}(x)$ represent the state polynomial after an m-bit input polynomial $I(x)$ is entered, i.e.,

$$S^{+m}(x) = I(x) \overset{m,s}{\gg} S_0(x) \tag{9}$$

where $\overset{m,s}{\gg}$ denotes the operator which shifts the m-bit input sequence into a linear compactor of length s. Also, let $S_{\mathrm{inp}}^{+m} \equiv$

$I(x) \overset{m,s}{\gg} 0$ and $S^{+m}_{\text{auto}} \equiv 0 \overset{m,s}{\gg} S_0(x)$ be the input and autonomous responses, respectively. In [22], it is shown that

$$S^{+m}(x) = S^{+m}_{\text{auto}}(x) \oplus S^{+m}_{\text{inp}}(x) = \sum_{j=1}^{\frac{s}{l}} S^{+m}_{\text{auto}_j}(x) \oplus \sum_{i=1}^{\frac{m}{k}} S^{+m}_{\text{inp}_i}(x) \tag{10}$$

where $S^{+m}_{\text{auto}_j}(x)$ is the autonomous response of the j-th state subpolynomial when we assume the input polynomial is a zero polynomial, and $S^{+m}_{\text{inp}_i}(x)$ is the input response of the ith input subpolynomial when we assume the initial state is a zero state, i.e.,

$$S^{+m}_{\text{auto}_j}(x) = 0 \overset{m,s}{\gg} S_{\text{sta}_j}(x) \tag{11}$$

and

$$S^{+m}_{\text{inp}_i}(x) = I_{\text{inp}_i} \overset{m,s}{\gg} 0. \tag{12}$$

The autonomous response $S^{+m}_{\text{auto}}(x)$ is the contribution of the current state to the state m time-frames from now, and the input response $S^{+m}_{\text{inp}}(x)$ is the contribution of the incoming m-bit input sequence to the state m time-frames from now. For an M-bit input sequence, the final state of the linear compactor can be evaluated by iteratively applying the preceding procedure $\lceil M/m \rceil$ times. Note that the polynomial additions/subtractions in this paper are all modulo-2 additions/subtractions.

III. Fast MISR Simulation Algorithm

In this section we will present several novel approaches to speeding up the linear compaction algorithm, and then propose a fast MISR simulation algorithm. The approaches and techniques will be described in detail in the following subsections.

A. Memory-Driven Partial-Input-LUT Approach

An IE-SISR with all-zero initial state can be viewed as a polynomial divider or modulus evaluator [23], [2], i.e., the state of the SISR obtained after the application of the input sequence $I(x)$ can be written as $S(x) = I(x) \bmod C(x)$, where $C(x)$ is the characteristic polynomial of the SISR. Let $\deg()$ denote the degree of a given polynomial. We know that $S(x) = I(x)$ if $\deg(I(x)) < \deg(C(x))$. We can use the modulus evaluation property of the IE-SISR to boost the signature computation speed in two different ways.

We first consider the contribution of an incoming m-bit input sequence in each iteration, and write the input response as

$$S^{+m}_{\text{inp}}(x) = \sum_{i=1}^{\frac{m}{k}} S^{+m}_{\text{inp}_i}(x) = S^{+m}_{\text{inp}_1}(x) \oplus \cdots \oplus S^{+m}_{\text{inp}_{\frac{m}{k}}}(x). \tag{13}$$

Let D be the maximum integer such that $\deg(I_{\text{inp}_{(m/k)-D+1}}) < \deg(C(x))$, i.e., $\deg(C(x)) > kD - 1$, then, according to (12), we have

$$S^{+m}_{\text{inp}_{\frac{m}{k}}}(x) = I_{\text{inp}_{\frac{m}{k}}}(x) \tag{14}$$

$$S^{+m}_{\text{inp}_{\frac{m}{k}-1}}(x) = I_{\text{inp}_{\frac{m}{k}-1}}(x) \tag{15}$$

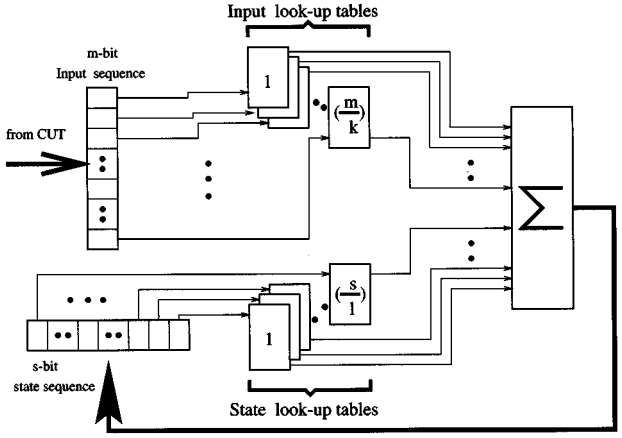

Fig. 3. The original linear compaction algorithm.

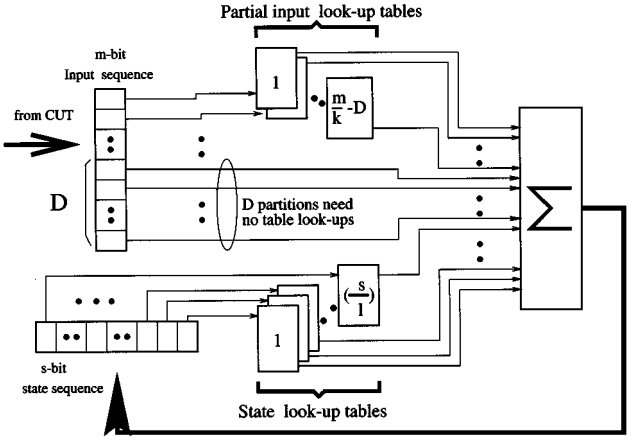

Fig. 4. The memory-driven partial-input LUT scheme.

$$\vdots$$

$$S^{+m}_{\text{inp}_{\frac{m}{k}-D+1}}(x) = I_{\text{inp}_{\frac{m}{k}-D+1}}(x). \tag{16}$$

Consequently, the input response of (13) is simply

$$S^{+m}_{\text{inp}}(x) = S^{+m}_{\text{inp}_1}(x) \oplus \cdots \oplus S^{+m}_{\text{inp}_{\frac{m}{k}-D}}(x) \oplus \underbrace{I_{\text{inp}_{\frac{m}{k}-D+1}} \oplus \cdots \oplus I_{\text{inp}_{\frac{m}{k}}}(x)}_{\text{no table look-up/construction required}}. \tag{17}$$

Generally, we have $D = \lfloor s/k \rfloor$. In each iteration, the contribution of the last D input partitions $I_{\text{inp}_{(m/k)-D+1}}, \ldots, I_{\text{inp}_{m/k}}$ (each of which has a degree less than $C(x)$) is the direct input subsequence without any computation. This memory-driven scheme benefits the signature computation in both the computation speed and memory requirement.

The speedup can be delineated by Figs. 3 and 4. In the original linear compaction scheme as shown in Fig. 3 [22], each nonzero input partition $I_{\text{inp}_i}(x), 1 \le i \le m/k$, is used as the key to the input LUTs to obtain the input contribution $S^{+m}_{\text{inp}_i}(x)$. On the other hand, in our memory-driven partial-input LUT scheme as shown in Fig. 4, only a portion of the nonzero input partitions [i.e., those with degrees higher than $C(x)$] need table references. This reduces the number of table look-ups as well as XOR operations for the trailing input sequences with degrees no more

than $\deg(C(x))$. Therefore, it increases the speed of the linear compaction algorithm.

Since table references are eliminated for those input partitions with degrees less than $\deg(C(x))$, the memory storage for the input LUTs is, therefore, reduced from $(m/k)2^k$ to $((m/k) - D)2^k$ s-bit words.

This memory-driven approach is especially useful for a long MISR with $\deg(C(x)) \approx m$, where m is the degree of each input sequence partition. In such a case, the input look-up operations are completely eliminated, so a better performance improvement can be observed.

B. Timing-Driven Input-LUT Approach

Speedup also can be achieved if we apply the modulus evaluation property of the SISR to the linear compaction algorithm in another way. The memory-driven partial-input-LUT approach presented above reduces the number of table look-ups, while the timing-driven input-LUT approach to be presented next reduces the number of iterations. The idea is to retire more input bits (increasing from m to $m+s$ bits) in each iteration, thus reducing the number of iterations. For an input sequence (represented by $I(x)$) of $m+s$ bits, the signature after each iteration is

$$\begin{aligned} S^{+(m+s)}(x) &= \left(I(x) \overset{(m+s),s}{\gg} 0\right) \oplus \left(0 \overset{(m+s),s}{\gg} S_0(x)\right) \\ &= \left((I_s(x) \oplus I_m(x)) \overset{(m+s),s}{\gg} 0\right) \\ &\quad \oplus \left(0 \overset{(m+s),s}{\gg} S_0(x)\right) \\ &= I_s(x) \oplus \left(I_m(x) \overset{(m+s),s}{\gg} 0\right) \\ &\quad \oplus \left(0 \overset{(m+s),s}{\gg} S_0(x)\right) \\ &= I_s(x) \oplus \sum_{i=\lceil \frac{s}{k} \rceil}^{\lceil \frac{m+s}{k} \rceil} S_{\mathrm{inp}_i}^{+(m+s)}(x) \\ &\quad \oplus \sum_{j=1}^{\frac{s}{l}} S_{\mathrm{auto}_j}^{+(m+s)}(x) \end{aligned} \tag{18}$$

where $I_s(x)$ is the least significant part of the input polynomial with degree less than $\deg(C(x))$, and $I_m(x) = I(x) \oplus I_s(x)$, i.e., $I_m(x)$ represents a sequence of $m+s$ bits with s trailing zero bits.

By (18), we use the input LUTs for the $m+s$ input bits (with s trailing zero bits), i.e., our algorithm can virtually retire $m+s$ bits in one iteration without requiring extra memory. This approach outperforms the linear compaction algorithm in that the number of iterations is reduced from $\lceil M/m \rceil$ to $\lceil M/(s+m) \rceil$ for an input sequence of M bits in total, at the cost of only an extra EXOR operation for $I_s(x)$ in each iteration. The table look-ups for $I_s(x)$ also are eliminated since its degree is less than $\deg(C(x))$.

Generally, this approach gains higher speedup than the memory-driven approach discussed above, especially for a dense (nonsparse) input sequence. For a sparse input sequence (i.e., an input sequence with a high percentage of zero segments), the linear compaction algorithm can skip input table look-up operations due to the null effect of the zero partitions. It reduces the computation time of the input contribution just like the direct substitution of the least significant input subsequence in the memory-driven approach—the memory-driven approach thus has a fair speedup under such a situation. On the other hand, the speedup of the timing-driven approach mainly results from the reduction of the number of iterations, so the speedup is pattern independent and guaranteed.

C. Error-Domain Flying-State-LUT Approach

Speedup techniques also can be developed from the exploration of the *error-domain inputs*. The error-domain inputs are obtained by taking the EXOR of the inputs from the fault-free circuit and the CUT. Let n be a positive integer that denotes the *stride*. For a zero input sequence of nm bits in length, the signature is the autonomous response, i.e.,

$$S^{+nm}(x) = 0 \overset{nm,s}{\gg} S_0(x) = \sum_{j=1}^{\frac{s}{l}} S_{\mathrm{auto}_j}^{+nm}(x). \tag{19}$$

By checking the zero-runs of length nm in the input sequence, we can jump directly to the signatures of the nm-bit input subsequences later. Starting from $S_0(x)$, the signature $S^{+nm}(x)$ can be obtained by state look-ups using a set of extra LUTs containing states nm bits ahead, which is called the *flying-state LUTs*. Note that the number of input bits processed in each iteration is n times that in the linear compaction algorithm, and the computation of the intermediate states from $S^{+m}(x)$ to $S^{+(n-1)m}(x)$ is dropped. A high speedup can be expected if there is a high percentage of nm-bit zero input runs.

In BIST simulation, what we usually care is the difference between the CUT signature and the fault-free one, not the signature itself (unless a dictionary is to be constructed for diagnosis). Let the error-domain inputs to the MISR be

$$I_e(x) = I_{\mathrm{cut}}(x) \oplus I(x) \tag{20}$$

where $I_{\mathrm{cut}}(x)$ and $I(x)$ are the output sequences of the CUT and the fault-free circuit, respectively. Generally, a circuit may contain many untestable faults and hard-to-detect faults. Therefore, in pseudorandom-pattern based BIST, $I_e(x)$ is usually sparse, i.e., it usually contains only a very small percentage of nonzero bits. Let p_i denote the probability that an input partition is nonzero. It is clear that by using $I_e(x)$, p_i can be greatly reduced as compared with that of $I(x)$. Table I shows the comparison of p_i among the bigger ISCAS-85 benchmark circuits. In this experiment, 8-bit partitions and single stuck-at faults were assumed, and 10 000 pseudorandom patterns were simulated for each fault. From the table, we see that a significant reduction of p_i is observed if we switch from $I(x)$ to $I_e(x)$, especially for large circuits.

Now let the error-domain signature of the MISR be

$$S_e^{+m}(x) = S_{\mathrm{cut}}^{m}(x) \oplus S^{+m}(x) \tag{21}$$

TABLE I
COMPARISON OF THE PROBABILITY OF NONZERO PARTITIONS BETWEEN $I(x)$ AND $I_e(x)$

Circuit	$p_i(I(x))$	$p_i(I_e(x))$	Faults
c1355	0.9908	0.0211	1574
c1908	0.9914	0.0088	1879
c2670	0.9942	0.0007	2747
c3540	0.5493	0.0034	3428
c5315	0.9946	0.0007	5350
c6288	0.8904	0.0009	7744
c7552	0.9950	0.0003	7550

where $S_{\text{cut}}^{+m}(x)$ and $S^{+m}(x)$ are the signatures of the CUT and the fault-free circuit, respectively. Then, we have

$$\begin{aligned} S_e^{+m}(x) &= S_{\text{cut}}^{+m}(x) \oplus S^{+m}(x) \\ &= \left(\left(I_{\text{cut}}(x) \overset{m,s}{\gg} 0\right) \oplus \left(0 \overset{m,s}{\gg} S_0(x)\right)\right) \\ &\quad \oplus \left(\left(I(x) \overset{m,s}{\gg} 0\right) \oplus \left(0 \overset{m,s}{\gg} S_0(x)\right)\right) \\ &= (I_{\text{cut}}(x) \oplus I(x)) \overset{m,s}{\gg} 0 \\ &= I_e(x) \overset{m,s}{\gg} 0. \end{aligned} \tag{22}$$

This shows how $S_{\text{cut}}^{+m}(x)$ can be obtained from the $I_e(x)$. Note that $I_e(x)$ can easily be obtained from any fault simulator. Based on our observation from Table I and (22) for $I_e(x)$, we propose the flying-state LUT approaches that take advantage of the observation. They are discussed in the following subsections.

D. Backward Zero-Checking Flying-State-LUT Approach

To effectively utilize the flying-state scheme, the input sequence has to satisfy the following two conditions: 1) the input sequence must be sparse; 2) there is an efficient method to identifying the nm-bit zero runs. We have shown previously that the error-domain inputs meet the first condition. For the second condition, we propose a *backward zero-checking* (BZC) technique. The idea is to find the last nonzero partition of the input sequence as soon as possible (if there is one) to minimize the zero-partition checking overhead, and to reserve the maximal runs of contiguous zeros for the next iteration to take full advantage of the zero runs. The procedure starts by reading an nm-bit input sequence and storing it clockwise in a circular queue of $2nm$ bits, with the *Head* pointer pointing to the starting position and the *Tail* pointer to the ending position of the input sequence, as shown in Fig. 5. We use the *Done* pointer to track the last partition to be checked—it points initially to *Head*. The *Start* pointer is associated with the first partition to be checked, and it points initially to *Tail*. The BZC procedure is summarized in Fig. 6.

In each iteration, the partitions from *Start* to *Done* are checked backward, i.e., counterclockwise, for zero partitions. Then, BZC updates *Head*, *Tail*, *Start*, and *Done* according to Fig. 6. During the backward checking process, if no nonzero partition is located, then *Head* will be equal to *Done* in the next iteration. However, if a nonzero partition is found, then *Head* will be less than *Done* in the next iteration. There are two reasons for doing it this way: 1) the checking time will be greatly reduced in the next iteration; 2) the partitions from *Head* to *Done* will be all zero partitions. Note that *Tail* always points to the last input partition in the queue so that we know the starting location for the next input sequence. Also, *Head* always points to the first of the input partitions being processed in the current iteration, and *Start* points to the $((m/k)-1)$st partition after it.

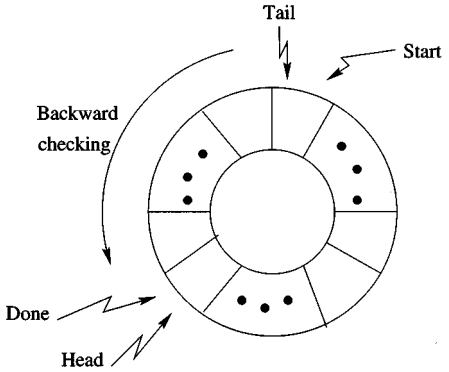

Fig. 5. Backward zero-checking procedure for an input sequence of length nm—initial condition or after flying-state operation.

```
BZC( ) /* Initially, Done = Head; Start = Tail; */
{
   if (no nonzero partition found from Start to Done) {
      flying_state_mode_from_Head_to_Tail();
      Head = Tail + 1; Done = Head;
      read_input_after_Tail();
      Tail=last_input_partition();
      Start = Head + (nm)/k − 1; }
   else { /* A nonzero partition found at Hit */
      if (Done == Head)
         normal_mode_from_Head_to_Hit();
      else {
         autonomous_contribution_from_Head_to_Done();
         normal_mode_from_Done_to_Hit(); }
      Head = Hit + 1; Done = Start + 1;
      read_input_after_Tail();
      Tail=last_input_partition();
      Start = Head + (nm)/k − 1; }
}
```

Fig. 6. The backward-zero-checking (BZC) algorithm.

If no nonzero partition between *Start* and *Done* is identified, then the final state is obtained by (19), i.e., the flying-state approach. This greatly reduces the number of table look-ups as compared with the linear compaction algorithm. After that, we update *Head*, *Tail*, *Start*, and *Done* for the next iteration as shown in Fig. 6.

If, on the other hand, a nonzero partition is found, say partition *Hit*, during the backward checking process, then *Hit* will be the last nonzero partition counting from *Head* (since the zero checking process is done backward toward *Done*), and we can locate it with minimal effort. The input sequence from *Head* to *Hit* will have to be processed by the m-bit mode operation. Under such circumstances, there are two different cases (due to the different outcomes in the previous iteration) to be considered.

The first case is when a nonzero partition was found in the previous iteration. In such case, *Done* is located after *Head*, and the partitions from *Head* to *Done* have been checked and known to be zeros during the previous iteration. Then, we can evaluate

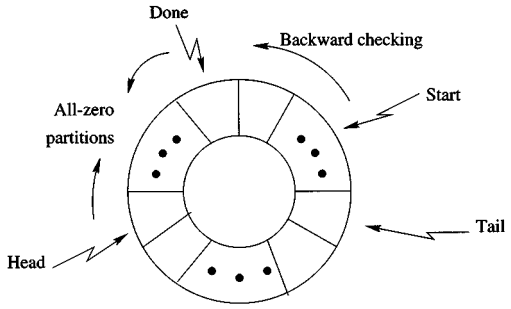

Fig. 7. Backward zero-checking procedure for an input sequence of nm bits, with nonzero partitions.

the state of partition *Hit* by the autonomous response of the current state from *Head* to *Done*-1 and the m-bit mode operation (using the memory-driven approach) from *Done* to *Hit*. After that, the pointers are updated for the next iteration. The partitions from *Head* to *Done* are now all zero, and need no checking in the next iteration. This situation is illustrated in Fig. 7.

The second case is when no nonzero partition is found in the previous iteration. In this case, *Done* is equal to *Head*, and the last nonzero partition of the nm-bit input sequence after *Head* is *Hit*. We then have to evaluate the contribution of the partitions from *Head* to *Hit* by the m-bit mode operation (using the memory-driven approach), and update the pointers afterwards. Again, after updating the pointers, the partitions from *Head* to *Done* are all zero and need no checking in the next iteration.

The advantages of our BZC technique can be summarized as follows.

1) It provides efficient checking of the nm-bit nonzero input sequence. The backward checking procedure can identify the maximal length of nonzero input sequence beginning at *Head* by checking only one nonzero partition, even if there are more than one nonzero partitions in the sequence. This greatly reduces the checking effort as compared with forward checking of all the nonzero partitions.
2) For a nonzero input sequence, the longest train of zero partitions from *Hit* to *Tail* is reserved for later flying-state operation, if possible.
3) Since we only need to check the input partitions from *Start* to *Done*, which in most cases is only a small portion of the nm-bit input sequence, the checking effort is greatly reduced.
4) It provides adaptive flying-state operation, and makes good use of nm-bit zero runs even if they are not read in the same nm-bit iteration.

The BZC technique can capture most runs of nm-bit zeros in the input sequence with little effort. By using the error-domain inputs, the flying-state technique with BZC results in a very high computation speed. The storage overhead of the BZC technique includes the flying-state LUTs of $(s/l)2^l$ s-bit words and the input queue of $2nm$ bits.

E. PC Flying-State-LUT Approach

Though the BZC technique can efficiently utilize the zero runs of the error-domain input sequence, there are situations when BZC is not very effective. With BZC, a fixed stride of n is selected. It seems that a longer stride will result in a faster computation. However, if the stride is too long, say nm/k is much larger than $1/p_i$, it is then hard to find such a long run of contiguous zeros. This will degrade the performance. As a result, its speedup is dependent on the stride.

We now present a simple extension to the BZC technique whose performance is not dependent on the stride. Consider an nm-bit input sequence with $h-1$ leading zero partitions. The signature obtained after this input sequence is

$$S^{+nm}(x) = \sum_{i=1}^{\frac{nm}{k}} \left(I_{\text{inp}_i}(x) \overset{nm,s}{\gg} 0 \right) \oplus 0 \overset{nm,s}{\gg} S_0(x)$$
$$= \sum_{i=h}^{\frac{nm}{k}} \left(I_{\text{inp}_i}(x) \overset{nm,s}{\gg} 0 \right) \oplus 0 \overset{nm,s}{\gg} S_0(x). \quad (23)$$

It can be seen that the signature computation in fact starts from the first nonzero partition (partition h in this case), i.e., the first $(h-1)$ table look-ups are eliminated.

The PC approach is based on this concept. Let the *pivot* be the leading P partitions of the current nm-bit input sequence. The PC technique checks if the pivot is zero by using the BZC algorithm. For a zero pivot, the signature is calculated by the flying-state operation as given in (23). Otherwise, the normal m-bit mode operation is performed from the first to the last nonzero partitions in the pivot, which are identified by using the BZC procedure on the pivot, i.e., *Start* always points to the Pth partition after *Head*. In BZC, an nm-bit input sequence is guaranteed to be retired in at most two iterations, hence the input queue is at most $2nm$-bit long. However, only P partitions of the nm-bit input sequence are guaranteed to be retired in two successive iterations for PC, so an adaptive input procedure should be used which takes into account the available space in the input queue. One simple method that simplifies the fault simulator procedure is to keep an input queue of $2nm$ bits in length as in BZC—read a new nm-bit input sequence if there is enough space, otherwise go to the next iteration.

The advantage of PC over BZC lies in its robustness, i.e., it is also good for input sequences which are nonsparse. Under such circumstances, PC can still gain speedup with a shorter pivot. This releases the strong dependence on the stride as in BZC. The choice of the pivot length P is determined by the speedup, overhead, and input sparsity. In fact, BZC can be considered as a special case of PC with $P = (nm/k)$.

F. DPC Flying-State-LUT Approach

For a highly sparse input sequence, a long pivot is desired. On the other hand, for a not-so sparse input sequence, a shorter pivot should be used. Since the sparsity of the input sequence depends on the faults, we propose a dynamic (adaptive) scheme which can stretch the pivot according to the sparsity of the error-domain input sequence. It is called the DPC approach. Let P be the pivot length. In the beginning, $P = (nm/k)$. During the signature computation process, if a zero pivot is detected, then

$$P = \begin{cases} P+1, & \text{if } P < \frac{nm}{k} \\ P, & \text{otherwise.} \end{cases} \quad (24)$$

If the pivot is nonzero, then

$$P = \begin{cases} P-1, & \text{if } P > 1 \\ 1, & \text{otherwise.} \end{cases} \tag{25}$$

G. Fast Signature Computation Algorithm

Our fast signature computation algorithm results from the preceding speedup approaches. Keeping the memory requirement as small as the linear compaction algorithm, our signature computation algorithm operates on the error-domain inputs $I_e(x)$. The algorithm is a combination of the memory-driven approach and the flying-state approach with BZC (for practical cases, DPC also can be used).

Although the discussion has been made on the IE-SISR, applying our algorithm to a MISR can be done by at least the following two different ways.

1) Convert the MISR into an equivalent IE-SISR and then apply our algorithm to the SISR to obtain the signature [17], [24].
2) Run our algorithm for each nonzero connection of the MISR and then combine the individual signatures according to the location of the nonzero connection to the MISR.

IV. Complexity Analysis

A. Memory Complexity

The memory space requirement of our algorithm includes the input LUTs, state LUTs, flying-state LUTs, and input queue, with their memory sizes denoted as $M_{i\cdot\text{lut}}$, $M_{s\cdot\text{lut}}$, $M_{f\cdot\text{lut}}$, and $M_{i\cdot\text{qeu}}$, respectively. For BZC, the memory sizes in terms of s-bit words are, respectively,

$$M_{i\cdot\text{lut}} = \frac{m}{k}2^k \tag{26}$$

$$M_{s\cdot\text{lut}} = \frac{s}{l}2^l \tag{27}$$

$$M_{f\cdot\text{lut}} = \frac{s}{l}2^l \tag{28}$$

$$M_{i\cdot\text{qeu}} = \frac{2nm}{s}. \tag{29}$$

Using the memory-driven approach, M_{inp} can be reduced by $(s/k)2^k$. Consequently, the total memory space of our algorithm is

$$\begin{aligned} M_{\text{total}} &= M_{i\cdot\text{lut}} + M_{s\cdot\text{lut}} + M_{f\cdot\text{lut}} + M_{i\cdot\text{qeu}} - \frac{s}{k}2^k \\ &= \frac{m-s}{k}2^k + \frac{2s}{l}2^l + \frac{2nm}{s}. \end{aligned} \tag{30}$$

Typically, $k = l$. Neglecting the small overhead of $M_{i\cdot\text{qeu}}$, this algorithm requires the same memory space as the linear compaction algorithm [22], i.e., $M_{\text{total}} = (m+s)2^l/l$.

B. Time Complexity

The BZC technique uses dynamic framing for determining the maximum runs of zero partitions. Let p_i denote the probability that a k-bit input partition is nonzero, and p_s denote the probability that an l-bit state partition is nonzero. For every nm-bit input polynomial in each iteration, the time complexity of our algorithm, including the LUT operation time and the BZC operation time, is calculated as follows.

1) LUT operation time:
   a) Non-flying-state operation:
   If there is a nonzero partition *Hit*, since the partitions from *Head* to *Done* have been checked and known to be zero in the BZC process, only autonomous responses are accumulated during this interval. The memory-driven operations are executed from *Done* to *Hit*, and the time required is

$$\begin{aligned} T_{nf\cdot\text{luo}} = {} & \frac{(\text{Done} - \text{Head})k}{m} T_{M\cdot\text{auto}} + \frac{(\text{Hit} - \text{Done} + 1)k}{m} \\ & \times (T_{M\cdot\text{auto}} + T_{M\cdot\text{inp}} + c_{\log}) \end{aligned} \tag{31}$$

   where

$$T_{M\cdot\text{auto}} = \frac{s}{l}p_s c_{\text{look}} + \left(\frac{s}{l} - 1\right)p_s c_{\log} + \frac{s}{w}c_{\text{check}} \tag{32}$$

   is the time for autonomous response accumulation in the memory-driven approach, and

$$\begin{aligned} T_{M\cdot\text{inp}} = {} & \left(\frac{m}{k}p_i - D\right)c_{\text{look}} + \left[\left(\frac{m}{k} - D\right) - 1\right]p_i c_{\log} \\ & + \frac{m - Dk}{w}c_{\text{check}}. \end{aligned} \tag{33}$$

   Note that c_{look} and $c_{\log}$ represent the average time required to perform a table look-up and a logical operation, respectively, c_{check} represents the average time to check if a subpolynomial is zero, and w is the word length. In general, $c_{\text{look}} > c_{\log} > c_{\text{check}}$.
   b) Flying-state operation:
   If nm/k consecutive zero partitions are found, then we perform the flying-state operation on this nm-bit input sequence. The operation time is

$$T_{f\cdot\text{luo}} = \frac{s}{l}p_s c_{\text{look}} + \left(\frac{s}{l} - 1\right)p_s c_{\log}. \tag{34}$$

2) BZC operation time:
   a) Non-flying-state operation:
   The time for the consecutive nonzero partitions is that spent during the checking process from *Done* to *Head* in the preceding iteration, i.e.,

$$T_{nf\cdot\text{check}} = \frac{(\text{Done} - \text{Head})k}{w}c_{\text{check}}. \tag{35}$$

   b) Flying-state operation:
   In this case, nm/k consecutive zero partitions in the input queue have been checked, and

$$T_{f\cdot\text{check}} = \frac{nm}{w}c_{\text{check}}. \tag{36}$$

Let p_0 be the probability that the nm-bit input polynomial is zero. Assume the input partitions are statistically independent, then

$$p_0 = (1 - p_i)^{\frac{nm}{k}}. \tag{37}$$

In each iteration, the probability that the algorithm operates in the flying-state mode is p_0. Consequently, the total iteration time is

$$T_{\text{total}} = p_0(T_{f\cdot\text{luo}} + T_{f\cdot\text{check}}) + (1-p_0)(T_{nf\cdot\text{luo}} + T_{nf\cdot\text{check}}) \tag{38}$$

where

$$\text{Head} \le \text{Done} \le \text{Hit} \le \text{Tail}. \tag{39}$$

The total iteration time is affected by the three random pointers, which may differ from iteration to iteration. Statistically, the mean (expectation) of the total iteration time is

$$\begin{aligned} T_m =& E\{T_{\text{total}}\} \\ =& p_0(T_{f\cdot\text{luo}} + T_{f\cdot\text{check}}) \\ &+ (1-p_0)\left[E\frac{\{(\text{Done}-\text{Head})\}k}{m}T_{M\cdot\text{auto}} \right. \\ &+ \frac{E\{(\text{Hit}-\text{Done}+1)\}k}{m} \\ &\times (T_{M\cdot\text{auto}} + T_{M\cdot\text{inp}} + c_{\log}) \\ &\left. + E\{T_{nf\cdot\text{check}}\}\right]. \end{aligned} \tag{40}$$

Let $L = \text{Head} - \text{Tail}$ be the number of partitions to be processed in an iteration, then $L = (nm/k)$. Assume *Done* is uniformly distributed between *Head* and *Tail*, then we have the probability

$$P\{(\text{Done}-\text{Head}) = x\} = \frac{1}{L+1} \tag{41}$$

for all x, $0 \le x \le L$. Obviously the mean is $E\{(\text{Done} - \text{Head})\} = L/2$. Now let $L_d = \text{Tail} - \text{Done}$. By (39), $0 \le Hit - Done \le L_d$. Also

$$E\{\text{Hit}-\text{Done}+1\} = E\{L_d/2\} + 1 = \frac{L}{4} + 1 \tag{42}$$

and

$$E\{T_{nf\cdot\text{check}}\} = \frac{L}{2}\frac{k}{w}c_{\text{check}}. \tag{43}$$

Therefore, we obtain

$$\begin{aligned} T_m =& p_0(T_{f\cdot\text{luo}} + T_{f\cdot\text{check}}) + (1-p_0) \\ &\times \left[\frac{Lk}{2m}T_{M\cdot\text{auto}} \right. \\ &+ \frac{\left(\frac{L}{4}+1\right)k}{m}(T_{M\cdot\text{auto}} + T_{M\cdot\text{inp}} + c_{\log}) \\ &\left. + \frac{L}{2}\frac{k}{w}c_{\text{check}}\right]. \end{aligned} \tag{44}$$

If it is a flying-state iteration, nm bits are processed, else the partitions from *Head* to *Hit* are processed. On the average, the number of bits processed in each iteration is

$$\begin{aligned} B &= p_0 nm + (1-p_0)E\{(Hit - Head + 1)k\} \\ &= p_0 nm + (1-p_0)\left(\frac{3L}{4}+1\right)k. \end{aligned} \tag{45}$$

TABLE II
COMPARISON OF MEMORY COMPLEXITY AND TIME COMPLEXITY FOR THE INPUT SEQUENCE $I(x)$

Circuit	[22]		Memory-driven LUT		Timing-driven LUT	
	Time	Mem	Time	Mem	Time	Mem
c1355	37	2048	18	1024	14	2048
c1908	36	2048	19	1024	20	2048
c2670	50	2048	28	1024	28	2048
c3540	68	2048	31	1024	32	2048
c5315	102	2048	54	1024	52	2048
c6288	156	2048	76	1024	77	2048
c7552	157	2048	61	1024	71	2048

TABLE III
COMPARISON OF MEMORY COMPLEXITY AND TIME COMPLEXITY FOR THE ERROR-DOMAIN INPUT SEQUENCE $I_e(x)$

Circuit	[22]		Memory-driven LUT		Timing-driven LUT	
	Time	Mem	Time	Mem	Time	Mem
c1355	19	2048	18	1024	6	2048
c1908	19	2048	15	1024	8	2048
c2670	22	2048	27	1024	10	2048
c3540	24	2048	31	1024	14	2048
c5315	50	2048	54	1024	24	2048
c6288	67	2048	80	1024	42	2048
c7552	74	2048	75	1024	34	2048

For an M-bit input sequence, the average total time thus is

$$T_{\text{ave}} = T_m M/B. \tag{46}$$

It can be seen that a significant speedup can be achieved when we use the error-domain inputs.

V. EXPERIMENTAL RESULTS

We use a 32-bit LFSR with characteristic polynomial $C(x) = 1+x^{15}+x^{32}$ in our simulations [22]. The ISCAS-85 benchmark circuits are used as the functional circuits under test, and the fault model assumed is the single stuck-at fault. For each fault, 10 000 random patterns are applied. For ease of discussion, the results shown below were taken from the sequence of the first primary output of each functional circuit. The results are similar if we pick a random primary output. We also assume that $m = 32$, $k = 8$, and $l = 8$. All the experiments were done on a SUN Sparc-20/71 workstation with 256 M-byte of RAM. The time was measured in *ticks* (a tick is 1/60 s), and the memory space was measured in 32-bit words.

Tables II and III list the experimental results for the original inputs $(I(x))$ and error-domain inputs $(I_e(x))$, respectively. Both the memory-driven and timing-driven approaches were simulated. Improvement can be observed for both approaches as compared with the linear compaction algorithm [22]. Note that there is no performance improvement for the memory-driven approach on the error-domain inputs (though memory space is reduced by half). This is because for the error-domain inputs the linear compaction algorithm also neglects as many table look-ups (for the zero input partitions) due to small p_i, so the check time for each partition is approximately that for the assignment operation on the partition in the memory-driven approach. From the tables, we can see that the timing-driven approach is faster than the memory-driven approach, especially for

TABLE IV
COMPUTATION TIME OF THE BZC PROCEDURE OVER A RANGE OF n (STRIDE)

Circuit	$n=5$	$n=10$	$n=15$	$n=20$	$n=25$	$n=30$	$n=35$
c1355	12	7	7	7	7	7	8
c1908	8	5	5	5	5	4	5
c2670	10	6	6	5	3	3	3
c3540	16	7	10	9	6	10	11
c5315	16	11	10	7	7	10	10
c6288	26	16	15	14	6	6	7
c7552	63	40	29	25	24	20	20

TABLE V
COMPUTATION TIME OF THE PC PROCEDURE OVER A RANGE OF P (PIVOT LENGTH), ASSUMING $n = 30$

Circuit	$P=5$	$P=10$	$P=15$	$P=20$	$P=25$	$P=30$
c1355	16	15	12	10	8	8
c1908	15	15	13	10	6	6
c2670	18	14	12	10	7	4
c3540	20	18	15	12	11	11
c5315	44	35	28	20	13	10
c6288	58	43	32	25	15	8
c7552	66	57	44	37	27	20

the error-domain inputs. Also, it is input-pattern independent as expected.

Table IV lists the computation time (in terms of ticks) of our algorithm over a range of stride n. Compared with that of the linear compaction algorithm shown in Table II, a great reduction in computation time can be seen—the speedup ranges from about 5 to 20 for $5 \leq n \leq 35$. It is interesting to note that the speedup tends to saturate (and even decay) for large n. This is because 1) the check time (c_{check}) dominates when p_i is very small if we use $I_e(x)$ and the BZC technique, and 2) the sparsity of $I_e(x)$ has been exploited. The memory requirement is the same as the linear compaction algorithm.

Table V shows the computation time for a range of P (pivot length) when we use the PC technique, assuming $n = 30$. From the table, we can see that the performance is very dependent on the value of P. A larger P means more intermediate states will be removed when we detect a zero pivot. Consequently, for a sparse input sequence, a large P should be used. For a sparse input sequence and a fixed n, PC is about as good as BZC if $P \approx n$.

VI. CONCLUSION

We have proposed several approaches to improving the LFSR/MISR signature computation performance. Based on the table look-up linear compaction algorithm and the modularity property of SISR, new accelerating schemes—partial input look-up tables and flying-state look-up tables—have been developed to boost the signature computation speed. We proposed two partial-input LUT methods—the memory-driven approach and the timing-driven approach. Significant speedup also has been observed when we explore the sparsity of the error-domain inputs. Operating on the error-domain inputs, the flying-state LUT method can significantly reduce the signature computation time with an appropriate stride which can be kept by using the DPC technique. Our signature computation algorithm using the memory-driven approach and the DPC technique is fast and requires small memory space. Mathematical analysis and simulation results on ISCAS85 benchmark circuits showed that, with the same memory requirement, this algorithm has an order of magnitude speedup over the linear compaction algorithm on the average. Although this algorithm was derived for SISR, a simple method exists that converts MISR to SISR. Consequently, fast MISR signature computation can be done.

REFERENCES

[1] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI*. New York, NY: John Wiley & Sons, 1987.

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

[3] S. Pilarski and T. Kameda, *A Probabilistic Analysis of Test-Response Compaction*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[4] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing: An alternative to fault simulationy," *IEEE Design & Test of Computers*, vol. 1, no. 1, pp. 83–93, Feb. 1984.

[5] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, "Fault simulation for structured VLSI," *VLSI Systems Design*, vol. 6, no. 12, pp. 20–32, Dec. 1985.

[6] K. J. Antreich and M. H. Schulz, "Accelerated fault simulation and fault grading in combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 6, no. 9, pp. 704–712, Sept. 1987.

[7] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault simulation in a distributed environment," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, 1988, pp. 686–691.

[8] W.-T. Cheng and M.-L. Yu, "Differential fault simulation—A fast method using minimal memory," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, June 1989, pp. 424–428.

[9] B. Underwood and J. Ferguson, "The parallel-test-detect fault simulation algorithm," in *Proc. Int. Test Conf. (ITC)*, 1989, pp. 712–717.

[10] F. Maamari and J. Rajski, "A method of fault simulation based on stem region," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 9, no. 2, pp. 212–220, Feb. 1990.

[11] H.-K. Lee and D.-S. Ha, "An efficient forward fault simulation algorithm based on the parallel single fault propagation," in *Proc. Int. Test Conf. (ITC)*, 1991, pp. 946–955.

[12] T. M. Niermann, W.-T. Cheng, and J. H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 11, no. 2, pp. 198–207, Feb. 1992.

[13] F. Maamari and J. Rajski, "The dynamic reduction of fault simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 12, no. 1, pp. 137–148, Jan. 1993.

[14] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proc. IEEE Int. Conf. Computer Design (ICCD)*, 1995, pp. 616–621.

[15] I. Pomeranz and S. M. Reddy, "On fault simulation for synchronous sequential circuits," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 335–340, Feb. 1995.

[16] M. B. Amin and B. Vinnakota, "ZAMBEZI: A parallel pattern parallel fault sequential circuit fault simulator," in *Proc. IEEE VLSI Test Symp. (VTS)*, 1996, pp. 438–443.

[17] C. F. See and K. K. Saluja, "An efficient method for computation of signatures," in *Proc. Int. Conf. VLSI Design*, Jan. 1992, pp. 245–250.

[18] C.-P. Kung, C.-J. Huang, and C.-S. Lin, "Fast fault simulation for BIST applications," in *Proc. Fourth Asian Test Symp. (ATS)*, Bangalore, Karnataka, Nov. 1995, pp. 93–99.

[19] G. Griffiths and G. C. Stones, "The tea-leaf reader algorithm: An efficient implementation of CRC-16 and CRC-32," *Communications of the ACM*, vol. 30, no. 7, pp. 617–620, July 1987.

[20] D. V. Sarwate, "Computation of cyclic redundancy checks via table look-up," *Communications of the ACM*, vol. 31, no. 8, pp. 1008–1013, Aug. 1988.

[21] N. Narendran, M. Franklin, and K. K. Saluja, "Parallel computation of LFSR signatures," in *Proc. Second Asian Test Symp. (ATS)*, Beijing, Nov. 1993, pp. 75–80.

[22] D. Lambidonis, A. Ivanov, and V. K. Agarwal, "Fast signature computation for BIST linear comparators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 14, pp. 1037–1044, Aug. 1995.

[23] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1983.
[24] K. K. Saluja and C. F. See, "An efficient signature computation method," *IEEE Design & Test of Computers*, vol. 9, no. 4, pp. 22–26, Dec. 1992.

Bin-Hong Lin received the B.S. and M.S. degrees both in electrical engineering, in 1990 and 1992, respectively, from National Tsing Hua University, Hsinchu, Taiwan. He is currently a principal engineer at TSMC, Hsinchu, Taiwan. Mr. Lin's research interests include VLSI design and testing, CAD tools for VLSI, and memory testing.

Shao-Hui Shieh received the B.S. degree in electronic engineering and the M.S. degree in automatic control engineering from Feng Chia University, Taichung, Taiwan, in 1977 and 1981, respectively. He is currently pursing the Ph.D. degree in electrical engineering at National Tsing Hua University. His research interests include VLSI design and testing, computer arithmetic and architecture, and redundant number representations.

Cheng-Wen Wu (S'86–M'87–SM'95) received the B.S.E.E. degree in 1981 from National Taiwan University, Taipei, Taiwan, and the M.S. and Ph.D. degrees, both in electrical and computer engineering, in 1985 and 1987, respectively, from the University of California, Santa Barbara. Since 1988 he has been with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, where he is currently a professor. He also has served as the director of the university's Computer and Communications Center from 1996 to 1998, and the director of the university's Technology Service Center from 1998 to 1999. He is a Guest Editor of the Journal of Information Science and Engineering, Special Issue on VLSI Testing.

Dr. Wu was the Technical Program Chair of the IEEE Fifth Asian Test Symposium (ATS'96), and is the General Chair of the Ninth ATS (ATS'00). He received the Distinguished Teaching Award from NTHU in 1996 and the Outstanding Electrical Engineering Professor Award from the Chinese Institute of Electrical Engineers (CIEE) in 1997. He is interested in design and test of high performance VLSI circuits and systems. He is a member of CIEE and a senior member of IEEE.