SCALABLE TEST GENERATORS FOR HIGH-SPEED DATAPATH CIRCUITS[1]

Hussain Al-Asaad†, John P. Hayes†, and Brian T. Murray††

† Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109-2122

†† Electrical and Electronics Department
General Motors R&D Center
30500 Mound Road, Warren, MI 48090-9055

ABSTRACT

This paper explores the design of efficient test sets and test-pattern generators for on-line BIST. The target applications are high-performance, scalable datapath circuits for which fast and complete fault coverage is required. Because of the presence of carry-lookahead, most existing BIST methods are unsuitable for these applications. High-level models are used to identify potential test sets for a small version of the circuit to be tested. Then a regular test set is extracted and a test generator TG is designed to meet the following goals: scalability, small test set size, full fault coverage, and very low hardware overhead. TG takes the form of a twisted ring counter with a small decoder array. We apply our technique to various datapath circuits including a carry-lookahead adder, an arithmetic-logic unit, and a multiplier-adder.

1. This research was supported by General Motors R&D Center.

1 INTRODUCTION

The widespread use of core-based designs makes built-in self test (BIST) an increasingly attractive design option [19]. BIST is a design-for-testability technique that places the testing functions physically with the circuit under test (CUT). BIST has several advantages over the alternative, external testing: (*i*) the ability to test in-system and at-speed, (*ii*) reduced test application time, (*iii*) less dependence on expensive test equipment, and (*iv*) the ability to automatically test devices on-line or in the field. On-line testing is especially important for high-integrity applications such as automotive systems, in which we are interested.

When BIST is employed, a VLSI system is partitioned into a number of CUTs. Each component CUT is logically configured as shown in Figure 1. In normal mode, the CUT receives its inputs *X* from other modules and performs the function for which it was designed. In test mode, a test pattern generator circuit (TG) applies a sequence of test patterns *S* to the CUT, and the test responses are evaluated by a response monitor (RM). This paper concentrates on the design of TG, although we also consider some relevant aspects of RM. In the most common type of BIST, test responses are compacted in RM to form signatures. The response signatures are compared with reference signatures generated or stored on-chip, and the error signal indicates any discrepancies detected. We assume this type of response processing in the following discussion.

Four primary parameters must be considered in developing a BIST methodology:

- *Fault coverage*: the fraction of faults of interest that can be exposed by the test patterns produced by TG and detected by RM. Most RMs produce the same signature for some

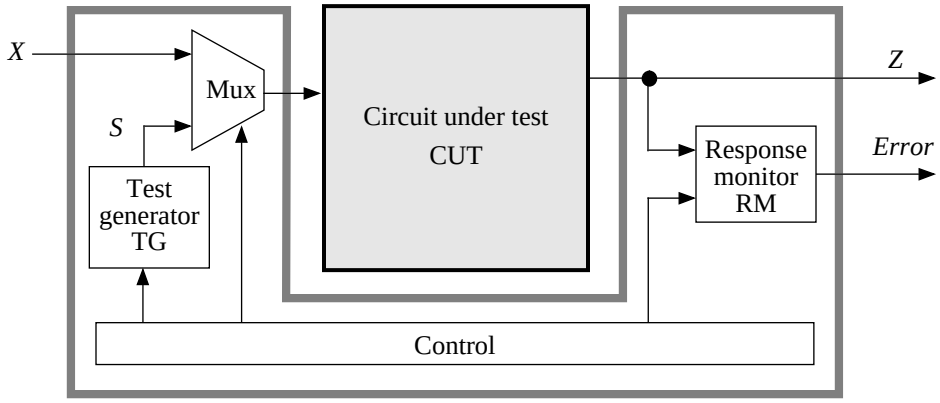


Figure 1 Generic BIST scheme.

faulty response sequences as for the correct response, a property called aliasing. This reduces fault coverage even if the tests produced by TG provide full fault coverage. Safety-critical applications require very high fault coverage, typically 100% of the modeled faults.

- *Test set size*: the number of test patterns produced by the TG. This parameter is linked to fault coverage: generally, large test sets imply high fault coverage. However, for on-line testing either at system start-up or periodically during normal operation, test set size must be kept small to minimize impact on system resources and reduce error latency, that is, the time elapsing before the effects of a fault are detected.
- *Hardware overhead*: the extra hardware needed for BIST. In most applications, high hardware overhead is not acceptable because of its impact on circuit size, packaging, power consumption, and cost.
- *Performance penalty*: the impact on performance of the normal circuit function, such as critical path delays, due to the inclusion of BIST hardware. This type of overhead is sometimes more important even than hardware overhead.

We have been investigating the design of TGs in the four-dimensional design space defined by the above parameters with the goals of 100% fault coverage, very small test sets, and low hardware overhead. The specific CUTs we are targeting are high-speed datapath circuits to which most existing BIST methods are not applicable. Our CUTs are N-input, scalable, combinational circuits with large values of N (64 or more). They also employ carry lookahead, a very common structure in high-performance datapaths. It is well known that such circuits have small deterministic test sets that can be computed fairly easily. For example, it is shown in [13] that the standard n-bit carry-lookahead adder (CLA) design, which has $N = 2n + 1$ inputs, has easily-derived and provably minimal test sets for all stuck-line faults; these test sets contain $N + 1$ test patterns. Some low-cost, scalable TG designs for datapath circuits based on C-testability (a constant number of test patterns independent of N) are known [12] [26], but they do not apply when CLA is used.

In this paper we describe a novel TG design methodology that addresses all the above issues, and illustrate it with several examples including an adder, an ALU and a multiplier-adder. The TG's structure is based on a twisted ring counter, and is tailored to generate a regular, deterministic test set of near-minimum size. Its hardware overhead is low enough to suggest that the TG can be incorporated into a standard cell or core design, as has been done for RAMs [20], adders [21]

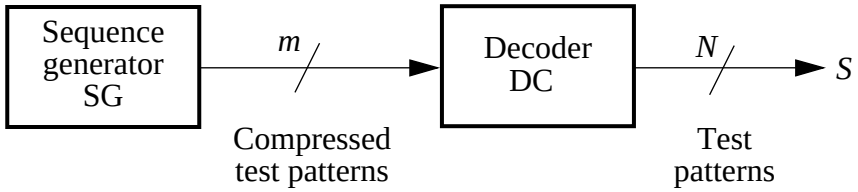


Figure 2 Basic structure of a test generation circuit.

and multipliers [12]. For a modest increase in hardware overhead and test set size, our method can also minimize the performance penalty. The proposed approach covers the major types of fast arithmetic circuits, and appears to be generalizable to other CUT types as well.

The paper is organized as follows. Section 2 reviews previous work on designing test generators. Section 3 describes the proposed approach to designing scalable test sets and test generators. In Section 4 we apply our approach to carry-lookahead adders, and apply it to several other examples in Section 5. We present some conclusions in Section 6.

2 TEST GENERATOR DESIGN

A generic TG structure applicable to most BIST styles is shown in Figure 2 [7]. The sequence generator SG produces an *m*-bit-wide sequence of patterns that can be regarded as compressed or encoded test patterns, and the decoder DC expands or decodes these patterns into *N*-bit-wide tests, where *N* is the number of inputs to the CUT. Generally, $m \leq N$, and the SG can be some type of counter that produces all 2^m *m*-bit patterns.

The most common TG design is a counter-like circuit that generates pseudorandom sequences, typically a maximal-length linear feedback shift register (LFSR) [6], a cellular automaton [5], or occasionally, a nonlinear feedback shift register [9]. These designs basically consist of a sequence generator only, and have $m = N$. The resulting TGs are extremely compact, but they must often generate excessively long test sequence to achieve acceptable fault coverage. Some CUTs, including the datapath circuits of interest, contain hard-to-detect faults that are detected by only a few test patterns T_{hard}. An *N*-bit LSFR can generate a sequence *S* that eventually includes $2^N - 1$ patterns (essentially all possibilities), however the probability that the tests in T_{hard} will appear early in *S* is low. Two general approaches are known to make *S* reasonably short. Test points can be

inserted in the CUT to improve controllability and observability; this, however, can result in a performance loss. Alternatively, some determinism can be introduced into *S*, for example, by inserting "seed" tests for the hard faults. Such methods aim to preserve the cost advantages of LFSRs while making *S* much shorter. However, these objectives are difficult to satisfy simultaneously. It can also be argued that pseudorandom approaches represent "overkill" for datapath CUTs, which, like RAMs [20], seem much better suited to directed deterministic approaches.

Weighted random testing adds logic to a basic LFSR to bias the pseudorandom sequence it generates so that patterns from the desired test set *T* appear near the start of *S* [6]. In a related method proposed by Dufaza and Cambon [11], an LFSR is designed so that *T* appears as a square block at the beginning of *S*. A test set must usually be partitioned into many square blocks, and the feedback function of the LFSR must be modified after the generation of each block, making this method complex and costly. The approach of Hellebrand et al. [14] [15] modifies the seeds used by the LFSR, as well as its feedback function. In other work, Touba and McCluskey [25] describe mapping circuits that transform pseudorandom patterns to make them cover hard faults.

Another large group of TG design methods, loosely called deterministic or nonrandom, attempt to embed a complete test *T* of size *P* in a generated sequence *S*. A straightforward way to do this is to store *T* in a ROM and address each stored test pattern using a counter. SG is then a $\lceil \log P \rceil$-bit address counter and the ROM serves as DC. Unfortunately, ROMs tend to be too expensive for storing entire test sequences. Alternatively, a $\lceil \log P \rceil$-state finite state machine (FSM) that directly generates *T* can be synthesized. However, the relatively large values of *P* and *N*, and the irregular structure of *T*, are usually more than current FSM synthesis programs can handle.

Several methods have been proposed that, like a ROM-based TG, use a simple counter for SG and design a low-cost combinational circuit for DC to convert the counter's output patterns into the members of *T* [3] [10]. Chen and Gupta [8] describe a test-width compression technique that leads to a DC that is primarily a wiring network. Chakrabarty et al. [7] explore the limits of test-pattern encoding, and develop a method for embedding *T* into test sequences of reasonable length.

Some TG design methods strive for balance between the straightforward generation of *T* using a ROM or FSM, and the hardware efficiency of an LFSR or counter. Perhaps the most straightfor-

ward way to do this was suggested by Agarwal and Cerny [1]. Their scheme directly combines the ROM and the pseudorandom methods. The ROM provides a small number of test patterns for hard-to-detect faults and the LFSR provides the rest of *T*.

None of the BIST methods discussed above explicitly addresses the scalability of the TG as the CUT is scaled. Scalable TGs based on C-testability have been described for iterative (bit-sliced) array circuits, such as ripple-carry adders [21] and array multipliers [12]. However, no technique has been proposed to design deterministic TGs that can be systematically rescaled as the size of a non-bit-sliced circuit, such as a CLA, is changed.

This paper introduces a class of TGs where SG is a compact (n + 1)-bit twisted ring counter and DC is CUT-specific. The output of SG can be efficiently decoded to produce a carefully crafted test sequence *S* that contains a complete test set for the CUT. As we will see, both SG and DC have a simple, scalable structure of the bit-sliced type. *S* is constructed heuristically to match a DC design of the desired type, so we can view this process as a kind of "co-design" of tests and their test generation hardware.

3 BASIC METHOD

We first examine the scalability of the target datapath circuits and their test sets. A circuit or module $M(n)$ with the structure shown in Figure 3 is loosely defined as *scalable* if its output function Z(n) is independent of the number n of its input data buses. Each such bus is w bits wide, and

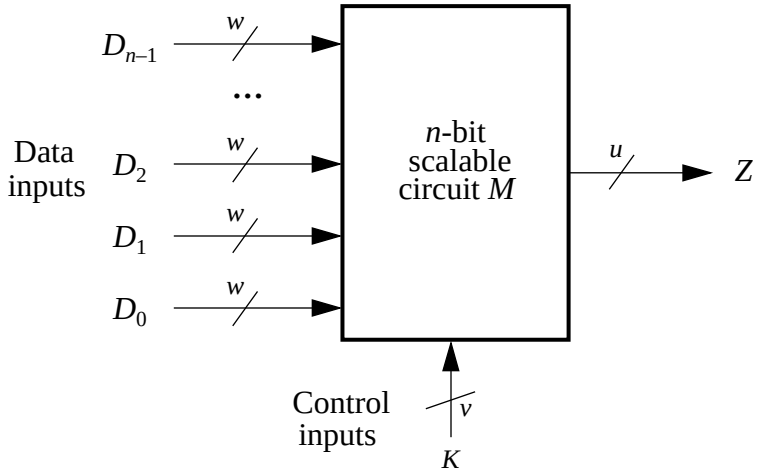


Figure 3 General scalable circuit.

there may also be a v-bit control bus, where w and v are constants independent of n. Bit-sliced arrays are special cases of scalable circuits in which each w-bit input data bus corresponds to a slice or stage. Most datapath circuits compute a function $Z(A(n), B(n))$, where $A(n) = A_{n-1}\ldots A_1A_0$ and $B(n) = B_{n-1}\ldots B_1B_0$, and are scalable in the preceding sense. They can be expressed in a recursive form such as

$$Z(A(n+1), B(n+1)) = z[Z(\mathrm{A}(n), B(n)), A_n, B_n]$$

For example, if Z is addition, we can write

$$Z_{\mathrm{add}}(A(n+1), B(n+1)) = Z_{\mathrm{add}}(A(n), B(n)) + 2^n \times A_n + 2^n \times B_n$$

where the 2^n factor accounts for the shifted position of the new operand $D_n = (A_n,B_n)$. Similarly, a test sequence $S(n)$ for a scalable circuit $M(n)$ can be represented in recursive form. $S(n)$ is considered to be scalable if

$$S(A(n+1), B(n+1)) = s[S(\mathrm{A}(n), B(n)), A_n, B_n]$$

As we will see, the test scaling functions s and S can take a few regular, shift-like forms for the CUTs of interest.

To introduce our method, we use the very simple example of a ripple-carry incrementer shown in Figure 4. Here the carry-in line C_0 is set to 1 in normal operation, but is treated as a variable during testing. The increment function Z_{inc} can be expressed as

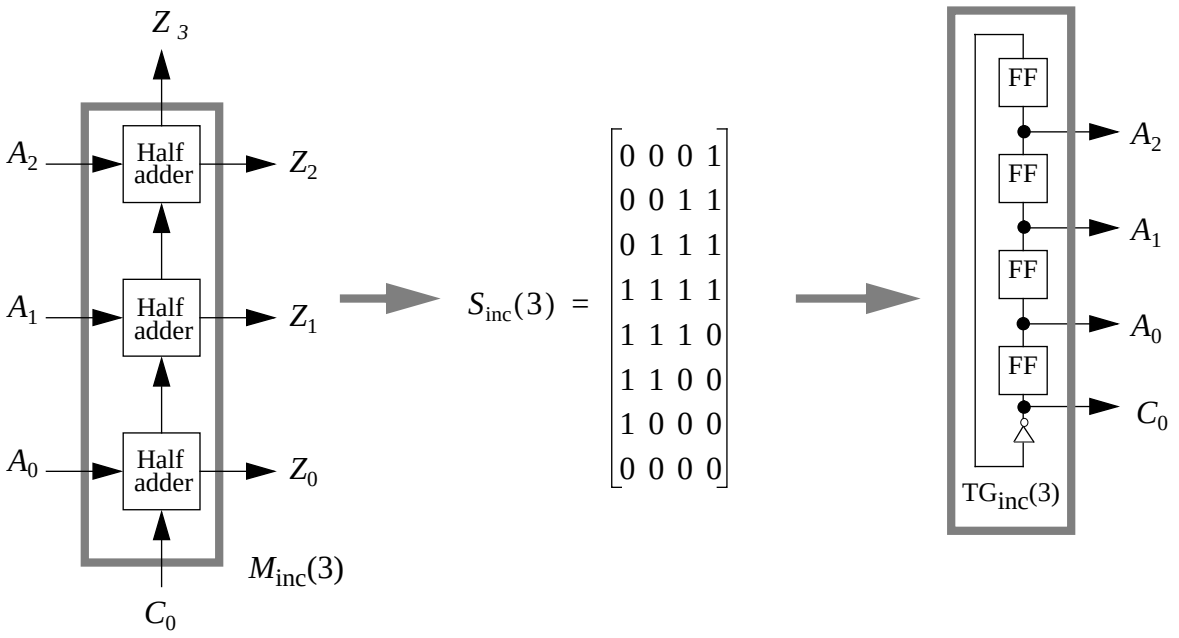
$$Z_{\mathrm{inc}}(A(n+1)) = Z_{\mathrm{inc}}(A(n)) + 2^n \times A_n + C_0 \qquad (1)$$

When $n = 1$, Equation (1) reduces to the half-adder equation

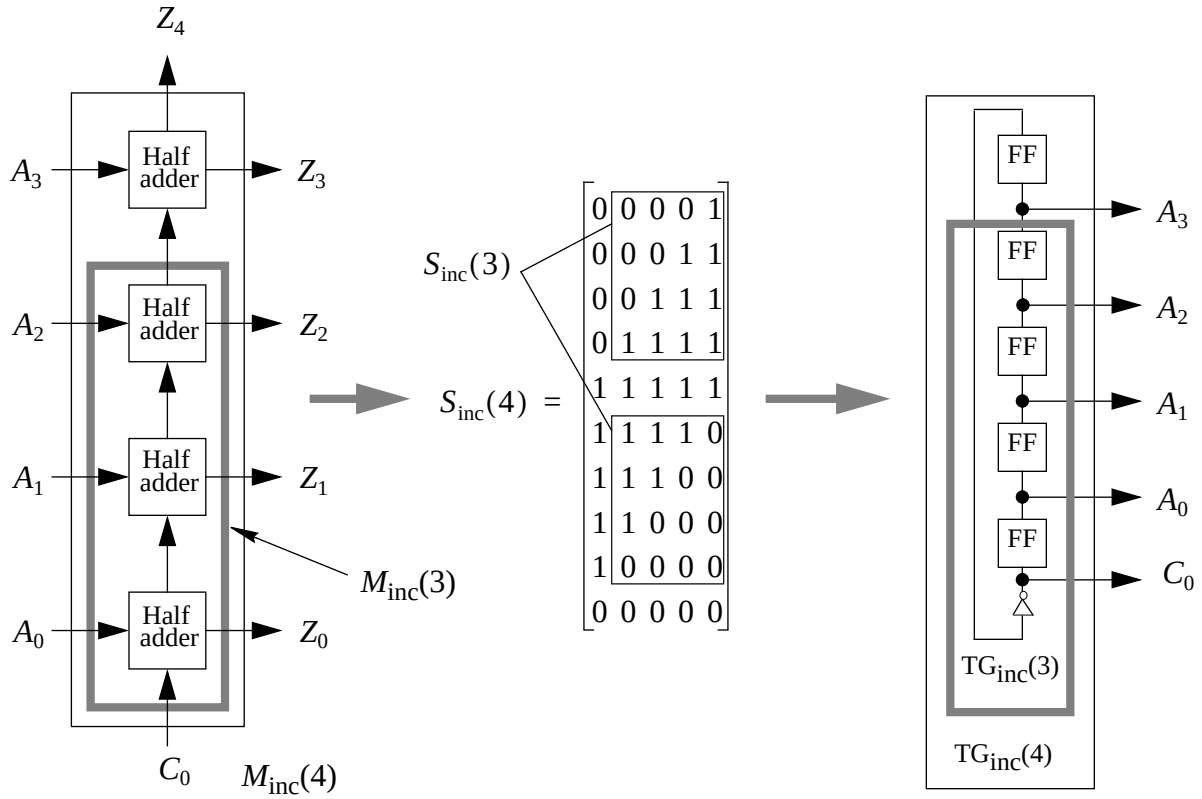
$$Z_{\mathrm{inc}}(A(1)) = A_0 + C_0 \qquad (2)$$

and (2) is realized by a single half-adder. An $(n + 1)$-bit incrementer $M_{\mathrm{inc}}(n)$ is obtained by appending a half-adder stage to $M_{\mathrm{inc}}(n - 1)$. Figure 4 shows how $M_{\mathrm{inc}}(3)$ is scaled up to implement $M_{\mathrm{inc}}(4)$.

A corresponding scaling of a test sequence $S_{\mathrm{inc}}(n)$ for $n = 3$ to 4 is also shown in the figure. $S_{\mathrm{inc}}(n)$ consists of $2n + 2$ test patterns of the form $A_{n-1}A_{n-2}\ldots A_0C_0$, each corresponding to a row in the binary matrices of Figure 4. These tests exhaustively test all half-adder slices of $M_{\mathrm{inc}}(n)$ by



(*a*)



(*b*)

Figure 4 Scalable incrementer and the corresponding test sequence and test generator (twisted ring counter) for (a) *n* = 3 and (*b*) *n* = 4.

applying the four patterns {00,01,10,11} to each half-adder and propagating any errors to the Z outputs. For example, the first test pattern $A_3A_2A_1A_0C_0$ = 00001 in $S_{inc}(4)$ applies 00 to the top three half-adders, and 01 to the bottom one. The next test 00011 applies 00 to the top two half-adders, 01 to the third half-adder from top, and 11 to the bottom one, and so on. If a fault is detected in, say, the bottom half-adder HA_0 by some pattern, an error bit appears either on Z_0, or on HA_0's carry-out line; in the latter case, the error will propagate to output Z_1, provided the fault is confined to HA_0. Thus $S_{inc}(n)$ detects 100% of all cell faults in the incrementer and, by extension, all single stuck-line (SSL) faults in $M_{inc}(n)$, independent of the internal implementation of the half-adder stages. The members of $S_{inc}(n)$ can easily be shown to constitute a minimal complete test with respect to cell or SSL faults. Note that, unlike a ripple-carry adder, a ripple-carry incrementer such as $M_{inc}(n)$ is *not* C-testable, and can easily be shown to require at least $2n + 2$ tests for 100% fault coverage. This linear testing requirement is unusual in bit-sliced circuits, but is typical of CLA designs.

Each test in the sequences $S_{inc}(n)$ shown in Figure 4 has been carefully chosen to be a shifted version of the test above it. Moreover, the first $n + 1$ tests have been chosen to be bitwise complements of the second $n + 1$ tests. (We will see later that these special properties of $S(n)$ can be satisfied in other, more general datapath circuits.) The sequence of the $2(n + 1)$ test patterns of S is exactly the state sequence of an $(n + 1)$-bit twisted ring (TR) counter[1]. This immediately suggests that a suitable test generator $TG_{inc}(n)$ for $M_{inc}(n)$ is an $(n + 1)$-bit TR counter, as shown in Figure 4. Clearly $TG_{inc}(n)$ is also a scalable circuit. Thus we have a TG design conforming to the general model of Figure 2, in which SG is a TR counter and DC is vacuous.

Although at first glance, a TG like $TG_{inc}(4)$ seems to embody a large amount of BIST overhead given the small size of $M_{inc}(4)$, we can argue that, in fact, $TG_{inc}(4)$ is of near-minimal (if not minimal) cost. Assuming 10 test patterns are required, any TG in the style of Figure 2 requires an SG of 10 states, implying $\lceil \log_2 10 \rceil = 4$ flip-flops, plus an indeterminate amount of logic to implement DC. Our design uses 5 flip-flops—one more than the minimum—plus a single inverter. The fact that DC is vacuous in this particular case is consistent with a basic property of the TR counter: it is almost fully decoded. In contrast, a comparable $(2n + 2)$-state ring counter has $2n + 2$ flip-flops and is fully decoded, whereas an ordinary (binary) counter has $\lceil \log_2(2n + 2) \rceil$ flip-

1. This well-known circuit is also called a switch-tail, Johnson or Moebius counter [17].

flops but is fully encoded. Thus we can hope to use TR counters in TGs that require little decoding logic.

We can now outline our general approach to designing TGs for scalable datapath circuits. We use high-level information about the CUT to explore in a systematic, but still heuristic, fashion a large number of its possible test sets to find one that has a regular, *shift-complement* (*SC)* structure resembling that illustrated by $S_{inc}(n)$ in Figure 4. The main steps involved are as follows:

1. Obtain a high-level, scalable model of the CUT $M(n)$.
2. Analyze this model using high-level functional analysis to derive a complete SSL-fault test set *T(n)* for $M(n)$ for some small value of n. Use don't cares in the test patterns wherever feasible.
3. Convert *T(n)* to an SC-style test sequence *S(n)* as far as possible.
4. Synthesize a test generator $TG(n)$ for $S(n)$ in the style of Figure 5.

The test generator $TG(n)$ adds to the TR counter of Figure 4 a decoding array *DC* of identical combinational cells $DC_0, DC_1, \ldots, DC_{n-1}$ that modify the counter's output as needed by a particular CUT. The array structure of *DC* ensures the scalability of TG. There is also a small mode-control FSM to allow *DC* to be modified for complex cases like multifunction circuits. The only inputs to the mode-control FSM are the signals *H* and *L*, which are active in the second half of states of the TR counter and the last state, respectively. The state behavior of the TR counter and the mode-control FSM are shown in Figure 5; they have $2n + 2$ and k states, respectively, where k is a fixed number independent of n. The total number of states for TG(n) is thus $k(2n + 2)$, which approximates the number of tests in the test set $T(n)$.

Our use of functional, high-level circuit models to derive test sets (Step 1 and 2 above) is based on the work of Hansen and Hayes [13], who show that test generation for datapath circuits can be done efficiently at the functional level while, at the same time, providing 100% coverage of low-level SSL faults for typical implementations. The model required for Step 1 is usually available for these types of circuits, since their scalable nature is exploited in their specification and carries through to high-level modeling during synthesis as illustrated by our incrementer example (Figure 4). Step 3 is perhaps the most difficult to formalize. It requires modifying and ordering the tests from Step 2 to obtain a sequence of shifting test patterns that resemble the output of the TR counter, but retain the full fault coverage of the original tests.

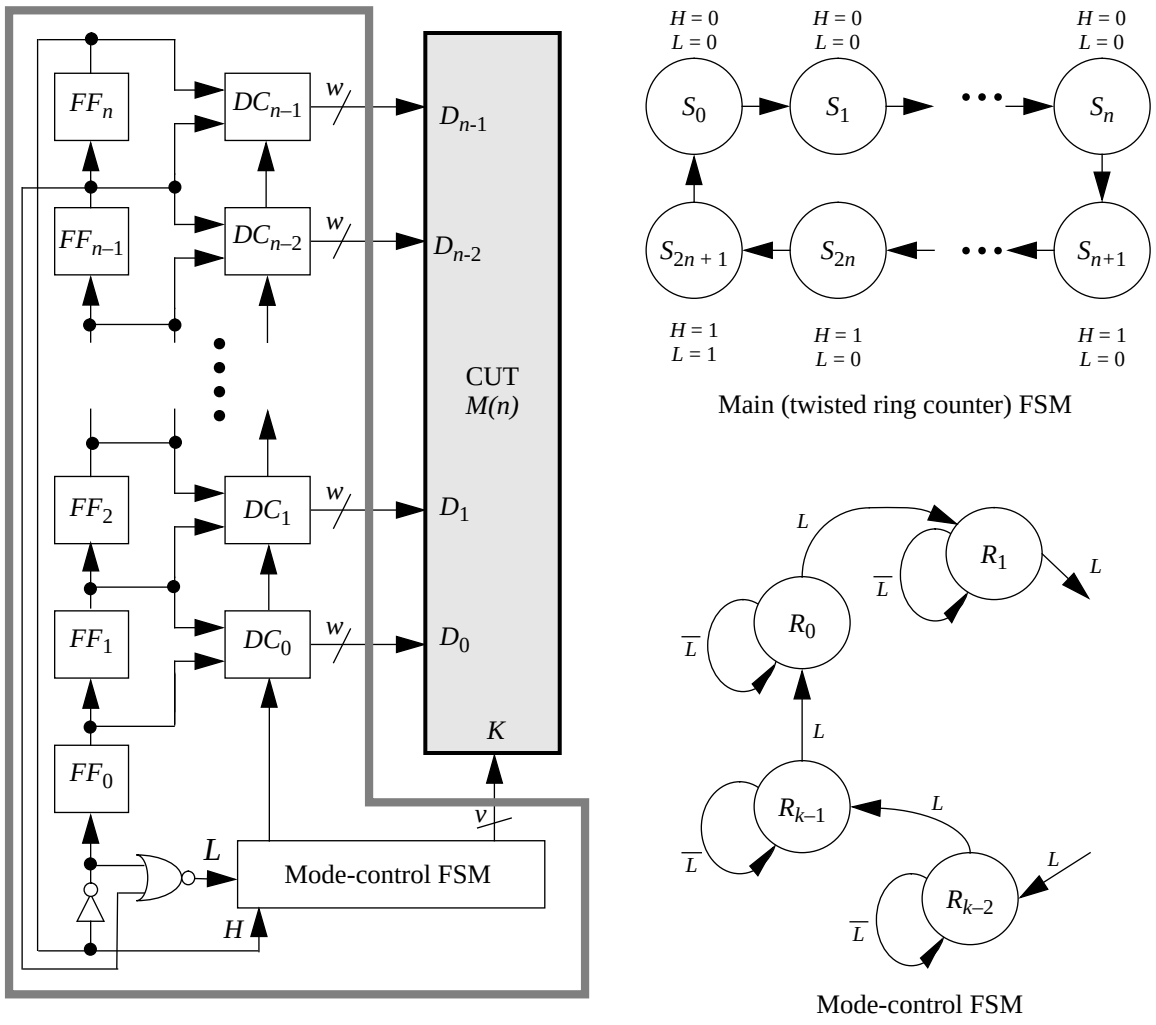


Figure 5 General structure of TG(*n*) and its state behavior.

In the remaining sections, we apply the preceding approach to derive similar, scalable test sets and test generators for the CLA and some other datapath circuits.

4 CARRY-LOOKAHEAD ADDER

The CLA is a key component of many high-speed datapath circuits, including arithmetic-logic units and multipliers. A high-level model of a generic n-bit CLA $M_{\mathrm{CLA}}(n)$, with the 4-bit 74283 [24] serving as a model, was derived in [13] and is shown in Figure 6. It is composed of (*i*) a module $M_{\mathrm{PGX}}(n)$ that realizes the functions $P_i = A_i + B_i$, $G_i = A_i B_i$, and $X_i = A_i \oplus B_i$, (*ii*) a carry-lookahead generator (CLG) module $M_{\mathrm{CLG}}(n)$ that computes all carry signals, and (*iii*) an

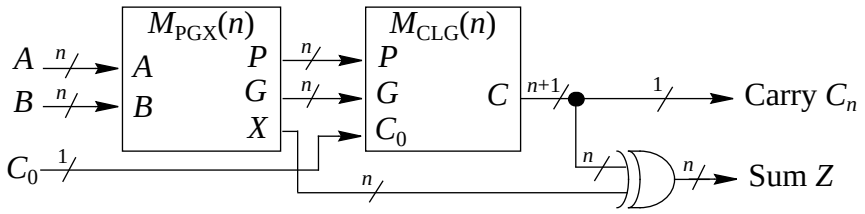


Figure 6 High-level model of the *n*-bit CLA [13].

XOR word gate that computes the sum outputs. The CLG module $M_{CLG}(n)$ contains the adder's hard-to-detect faults, and so is the focus of the test-generation process. Its testing requirements can be satisfied by generating tests for the SSL faults on the input lines of $M_{CLG}(n)$ that propagate the fault effects along the path to C_n, which is the longest and "hardest" fault-detection path. The resulting test set $T_{CLG}(n)$ contains $2n + 2$ tests and detects all faults in the CLG logic. For example, when $n = 2$, $T_{CLG}(2) = \{10101, 10110, 11000, 10100, 10001, 00111\}$, where the test patterns are in the form $P_1G_1P_0G_0C_0$. Hansen and Hayes [13] have proven that such a test set detects all SSL faults in typical implementations of $M_{CLG}(n)$. Their method induces high-level functional faults from the SSL faults, and generates $T_{CLG}(n)$ for a small set of functional faults that cover all SSL faults. Because the carry functions are unate, it can be shown that $T_{CLG}(n)$ is a "universal" test set in the sense of [2], hence it covers all SSL faults in any inverter-free AND/OR implementation of $M_{CLG}(n)$.

Once the tests for $M_{CLG}(n)$ are known, they are traced back to the primary inputs of the $M_{CLA}(n)$ through the module $M_{PGX}(n)$; the resulting test sets for $n = 2$, are shown in Table 1(a). The table gives a condensed representation of $M_{CLG}(2)$'s test requirements within $M_{CLA}(2)$, and specifies implicitly all possible sets of 6 tests (the minimum number) that cover all SSL faults in

Table 1 Condensed representation of complete test sets in (a) $M_{CLG}(2)$ and (b) $M_{PGX}(2)$. (c) Specific test sequence for the CLA that follow the SC style.

A_1 B_1	A_0 B_0	C_0
{10,01}	{10,01}	1
{10,01}	00	1
00	11	1
{10,01}	{10,01}	0
{10,01}	11	0
11	00	0

(a)

A_1 B_1	A_0 B_0	C_0
01	xx	x
10	xx	x
xx	01	x
xx	10	x

(b)

Test #	A_1 B_1	A_0 B_0	C_0
1	10	10	1
2	10	00	1
3	00	11	1
4	01	01	0
5	01	11	0
6	11	00	0

(c)

Table 2 Complete and minimal SC-style test sequence for the 74283 CLA and the corresponding responses.

Test #	Input pattern										Response				
	A_3	B_3	A_2	B_2	A_1	B_1	A_0	B_0	C_0	C_4	Z_3	Z_2	Z_1	Z_0
1	1	0	1	0	1	0	1	0	1	1	0	0	0	0
2	1	0	1	0	1	0	0	0	1	0	1	1	1	1
3	1	0	1	0	0	0	1	1	1	0	1	1	1	1
4	1	0	0	0	1	1	1	1	1	0	1	1	1	1
5	0	0	1	1	1	1	1	1	1	0	1	1	1	1
6	0	1	0	1	0	1	0	1	0	0	1	1	1	1
7	0	1	0	1	0	1	1	1	0	1	0	0	0	0
8	0	1	0	1	1	1	0	0	0	1	0	0	0	0
9	0	1	1	1	0	0	0	0	0	1	0	0	0	0
10	1	1	0	0	0	0	0	0	0	1	0	0	0	0

$M_{CLG}(2)$. For example, the first row in Table 1(a) defines the tests for the fault "C_0 fails to propagate 0 to C_2", which requires $C_0 = 1$ and $A_iB_i = 10$ or 01 for $i = 0$ and 1. Hence the potential tests for this fault are {10101, 10011, 01101, 01011}. The second row specifies the test for the faults "A_0 or B_0 fails to propagate 1 to C_2", which requires $A_0B_0 = 00$, but $A_iB_i = 10$ or 01 as before to ensure error propagation to C_2. To test for all SSL faults in module $M_{PGX}(n)$, each pair of bits A_iB_i must be exhaustively tested. The tests for $M_{CLG}(n)$ guarantee the application of 00 and 11 on each A_iB_i of $M_{PGX}(n)$, as we can see from Table 1(a), for the case of $n = 2$. Therefore, the remaining requirement for testing $M_{PGX}(n)$ is to apply 01 and 10 to each A_iB_i, as shown in Table 1(b). The n XOR gates that feed the sum output Z are automatically covered by the tests for $M_{CLG}(n)$ and $M_{PGX}(n)$, and also provide non-blocking error propagation paths for these modules.

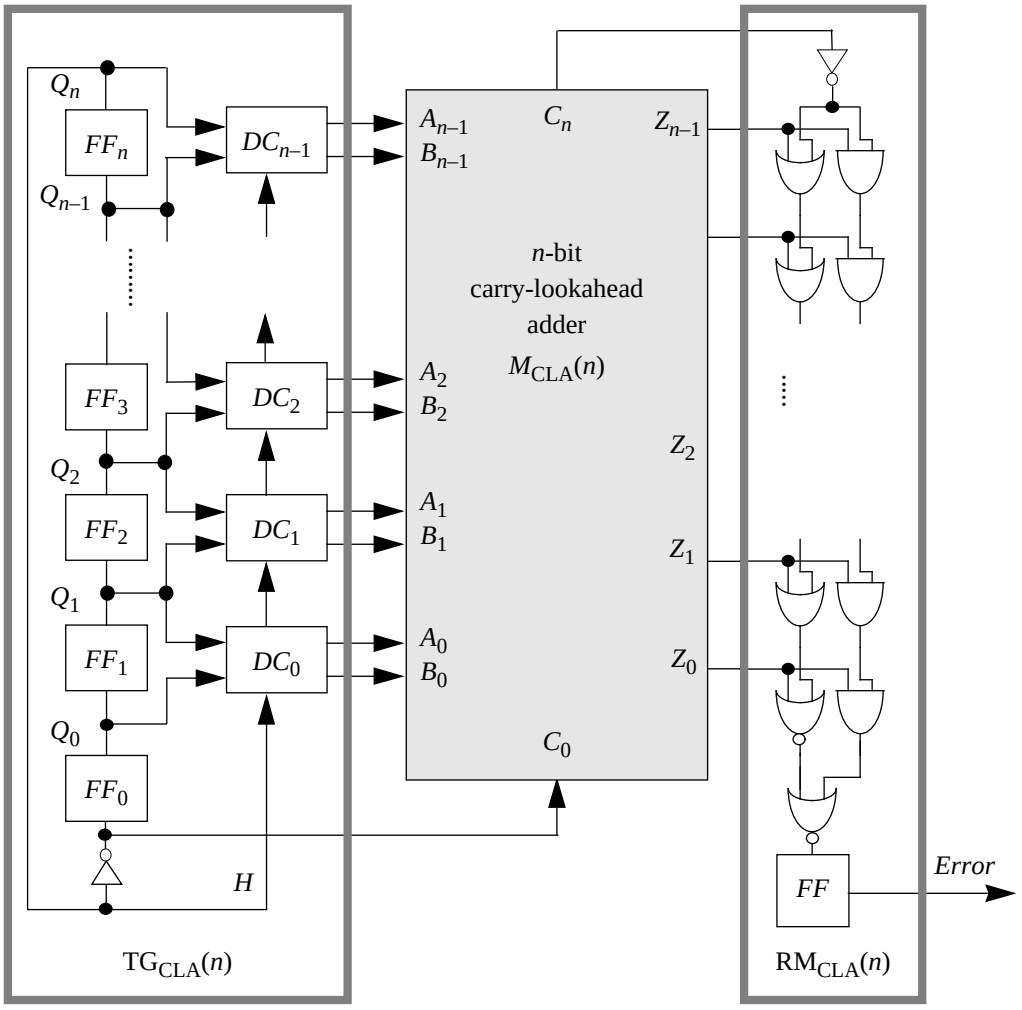
Once we know the possible test sets for $M_{CLA}(n)$, our next goal is to obtain a specific test sequence that follows the SC style. Such a test sequence of size 6 is extracted in Table 1(c). This sequence is minimal and complete for SSL faults in the CLA [13], as can be verified by simulation. Tests 1, 2, and 3 are selected to make the 00 pattern applied to A_iB_i shift from right to left, as the shading in the table shows. Tests 4, 5, and 6 are selected to be the complements of tests 1, 2, and 3 respectively. Hence these tests shift the pattern 11 on A_iB_i from right to left. The specific test sequence $S_{CLA}(2)$ in Table 1(c) can be easily extended to a complete test sequence $S_{CLA}(n)$ of size $2n + 2$ for any $n > 2$. For example, Table 2 shows how $S_{CLA}(2)$ is scaled up to $S_{CLA}(4)$ to obtain a complete SC-style test sequence for the 74283 CLA.

Table 3 Mapping of the CLA test sequence to the TR counter's output sequence.

Test #	TR counter outputs					TG outputs (CLA test sequence)				
	H	Q_4Q_3	Q_3Q_2	Q_2Q_1	Q_1Q_0	$A_3\,B_3$	$A_2\,B_2$	$A_1\,B_1$	$A_0\,B_0$	C_0
1	0	0 0	0 0	0 0	0 0	1 0	1 0	1 0	1 0	1
2	0	0 0	0 0	0 0	0 1	1 0	1 0	1 0	0 0	1
3	0	0 0	0 0	0 1	1 1	1 0	1 0	0 0	1 1	1
4	0	0 0	0 1	1 1	1 1	1 0	0 0	1 1	1 1	1
5	0	0 1	1 1	1 1	1 1	0 0	1 1	1 1	1 1	1
6	1	1 1	1 1	1 1	1 1	0 1	0 1	0 1	0 1	0
7	1	1 1	1 1	1 1	1 0	0 1	0 1	0 1	1 1	0
8	1	1 1	1 1	1 0	0 0	0 1	0 1	1 1	0 0	0
9	1	1 1	1 0	0 0	0 0	0 1	1 1	0 0	0 0	0
10	1	1 0	0 0	0 0	0 0	1 1	0 0	0 0	0 0	0

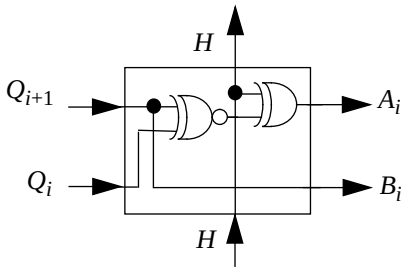
A test generator $TG_{\mathrm{CLA}}(n)$ for $M_{\mathrm{CLA}}(n)$ can now be synthesized from $S_{\mathrm{CLA}}(n)$ following the general structure in Figure 5. As in the incrementer example, the sequence generator is an $(n + 1)$-bit TR counter. Note, however, that the number of input lines has almost doubled from $N = n + 1$ to $N = 2n + 1$. The size of $S_{\mathrm{CLA}}(n)$ is $2n + 2$, which is the number of states of the TR counter, so no mode-control FSM is needed. Table 3 lists the CLA test sequence side by side with the TR counter's output sequence for the 4-bit case; the truth table of a decoder cell DC_i can be extracted directly, as shown in Figure 7. The combinations $(HQ_{i+1}Q_i) = \{010, 101\}$ never appear at the inputs of the decoder cells, hence the outputs of DC_i are considered don't care for these combinations. Furthermore, the patterns $(HQ_{i+1}Q_i) = \{011, 100\}$ never appear at the inputs of the high-order decoder cell DC_{n-1}, however, we choose not to take advantage of this, since our goal is to keep the decoder logic DC simple and regular. The carry-in signal C_0 can be seen from Table 3 to be $C_0 = \overline{H}$. The resulting design for $\mathrm{TG}_{\mathrm{CLA}}(n)$ shown in Figure 7 requires $n + 1$ flip-flops and n small logic cells that form DC. The hardware overhead of TG, as measured by transistor count in a standard CMOS implementation, amounts to 35.8% for a 32-bit CLA. This overhead decreases as the size of the CLA increases, a characteristic of all our TGs.

Our TGs, like the underlying TR counters, produce two sets of complementary test patterns. Such tests naturally tend to detect many faults because they toggle all primary inputs and outputs, as well as many internal signals. An n-bit adder also has the interesting property that A ***plus*** B ***plus*** $C_{in} = C_{out}S$ implies $\overline{A}$ ***plus*** $\overline{B}$ ***plus*** $\overline{C}_{in} = \overline{C}_{out}\overline{S}$, where ***plus*** denotes addition modulo 2^n. Hence the adder's outputs are complemented whenever a test is complemented, implying that there are only



H	$Q_{i+1}Q_i$	$A_i B_i$
0	0 0	1 0
0	0 1	0 0
0	1 1	1 1
1	1 1	0 1
1	1 0	1 1
1	0 0	0 0

Truth table of DC_i



DC_i circuit

Figure 7 Scalable hardware test generator and response monitor for an *n*-bit CLA.

two distinct responses, 100...0 and 011...1, to all the tests in $TG_{CLA}(n)$, as can be seen from Table 2. Consequently, a simple, low-cost and scalable RM can be easily designed for the CLA adder as depicted in Figure 7. This example shows that some of the benefits of scalable, regular tests carry over to RM design.

5 OTHER EXAMPLES

In this section, we extend the approach developed in the preceding sections to the design of a TR-counter-based TG for an arithmetic logic unit and two circuits involving multiplication.

Arithmetic Logic Unit. We first consider an n-bit ALU $M_{ALU}(n)$ that employs the standard design represented by the 4-bit 74181 [24]. This ALU is basically a CLA with additional circuits that implement all 16 possible logic functions of the form $f(A, B)$. A high-level model for the 74181 is shown in Figure 8 [13], and consists of a CLG module M_2, a function select module M_1, and several word gates. Following the approach of the previous section, the tests needed for the CLG module M_2 are traced back to the ALU's primary inputs. During this process, the signal values applied to the function-select control bus S are chosen to satisfy the testing needs for M_1 as well. An obvious choice is to make S select the add ($S_3S_2S_1S_0$ = 1001) and subtract ($S_3S_2S_1S_0$ = 0110) modes of the ALU. However, we found by trial and error that the assignments $S_3S_2S_1S_0$ = 1010 and 0101 lead to a TG design with less overhead. The testing needs for the word gates in the high-level model of the ALU must be also considered. The final test sequence $S_{ALU}(n)$ has an SC structure that closely resembles that of the CLA. Table 4 shows $S_{ALU}(4)$; note how the tests exhibit the same shifting property as before for the patterns $A_iB_i = 11$ and $A_iB_i = 00$. Moreover, tests 1:20 are the complements of tests 21:40. The test sequence $S_{ALU}(4)$ is not minimal, however, since 12 tests are sufficient to detect all SSL faults in the 74181 [13]. $S_{ALU}(4)$ can be easily extended to $S_{ALU}(n)$ with a near-minimal size of $8n + 8$.

A test generator $TG_{ALU}(n)$ for $M_{ALU}(n)$ is shown in Figure 9, which again follows the general test generator model of Figure 5. Since the test sequence size is $8n + 8$ and the general test gener-

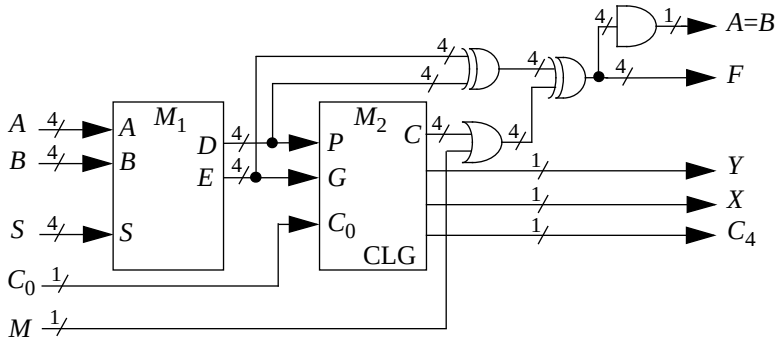
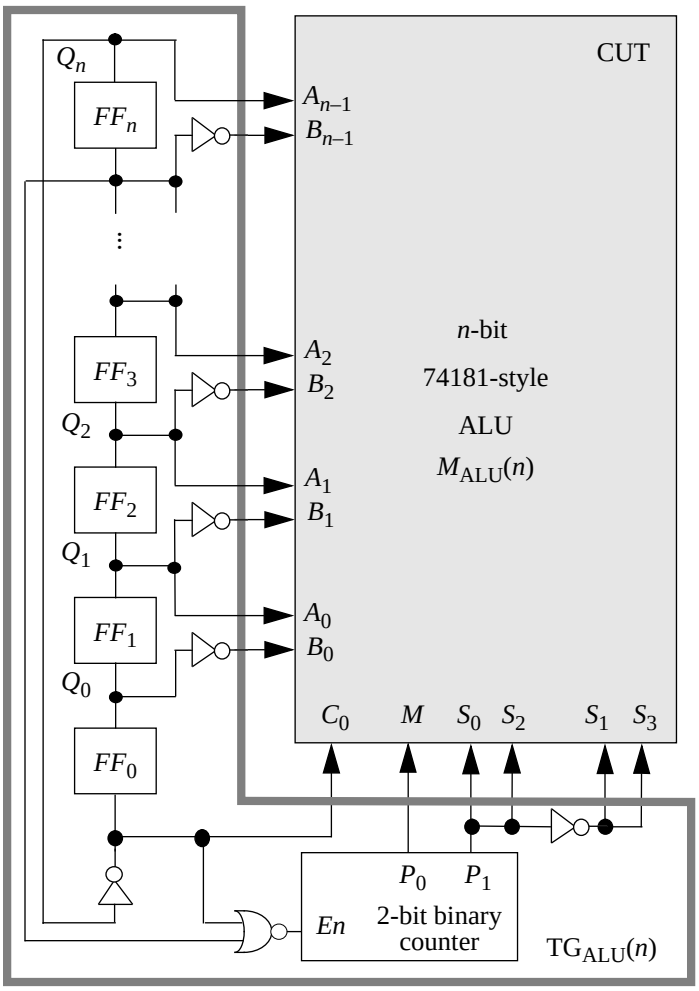


Figure 8 High-level model for the 74181 ALU [13].

Table 4 Complete and near-minimal SC-style test sequence for the 74181 ALU.

Test #	A_3	B_3	A_2	B_2	A_1	B_1	A_0	B_0	C_0	M	S_3	S_2	S_1	S_0
1	0	1	0	1	0	1	0	1	1	0	1	0	1	0
2	0	1	0	1	0	1	0	0	1	0	1	0	1	0
3	0	1	0	1	0	0	1	0	1	0	1	0	1	0
4	0	1	0	0	1	0	1	0	1	0	1	0	1	0
5	0	0	1	0	1	0	1	0	1	0	1	0	1	0
6	1	0	1	0	1	0	1	0	0	0	1	0	1	0
7	1	0	1	0	1	0	1	1	0	0	1	0	1	0
8	1	0	1	0	1	1	0	1	0	0	1	0	1	0
9	1	0	1	1	0	1	0	1	0	0	1	0	1	0
10	1	1	0	1	0	1	0	1	0	0	1	0	1	0
11	0	1	0	1	0	1	0	1	1	1	1	0	1	0
12	0	1	0	1	0	1	0	0	1	1	1	0	1	0
13	0	1	0	1	0	0	1	0	1	1	1	0	1	0
14	0	1	0	0	1	0	1	0	1	1	1	0	1	0
15	0	0	1	0	1	0	1	0	1	1	1	0	1	0
16	1	0	1	0	1	0	1	0	0	1	1	0	1	0
17	1	0	1	0	1	0	1	1	0	1	1	0	1	0
18	1	0	1	0	1	1	0	1	0	1	1	0	1	0
19	1	0	1	1	0	1	0	1	0	1	1	0	1	0
20	1	1	0	1	0	1	0	1	0	1	1	0	1	0
21	0	1	0	1	0	1	0	1	1	0	0	1	0	1
22	0	1	0	1	0	1	0	0	1	0	0	1	0	1
23	0	1	0	1	0	0	1	0	1	0	0	1	0	1
24	0	1	0	0	1	0	1	0	1	0	0	1	0	1
25	0	0	1	0	1	0	1	0	1	0	0	1	0	1
26	1	0	1	0	1	0	1	0	0	0	0	1	0	1
27	1	0	1	0	1	0	1	1	0	0	0	1	0	1
28	1	0	1	0	1	1	0	1	0	0	0	1	0	1
29	1	0	1	1	0	1	0	1	0	0	0	1	0	1
30	1	1	0	1	0	1	0	1	0	0	0	1	0	1
31	0	1	0	1	0	1	0	1	1	1	0	1	0	1
32	0	1	0	1	0	1	0	0	1	1	0	1	0	1
33	0	1	0	1	0	0	1	0	1	1	0	1	0	1
34	0	1	0	0	1	0	1	0	1	1	0	1	0	1
35	0	0	1	0	1	0	1	0	1	1	0	1	0	1
36	1	0	1	0	1	0	1	0	0	1	0	1	0	1
37	1	0	1	0	1	0	1	1	0	1	0	1	0	1
38	1	0	1	0	1	1	0	1	0	1	0	1	0	1
39	1	0	1	1	0	1	0	1	0	1	0	1	0	1
40	1	1	0	1	0	1	0	1	0	1	0	1	0	1

ator has $k(2n + 2)$ states, the mode-select FSM of $TG_{ALU}(n)$ has $k = 4$ states. The state table of the mode-select FSM and the truth table of the decoder cell are shown in Figure 9. The decoder cell DC_i turns to be extremely simple in this case—a single inverter. The overall test generator $TG_{ALU}(n)$ requires $n + 3$ flip-flops, n inverters, and a small amount of combinational logic whose size is independent of n. The hardware overhead decreases as the number of inputs n of the ALU



Present state	Next state	Present outputs	
		M	$S_3\ S_2\ S_1\ S_0$
R_0	R_1	0	1 0 1 0
R_1	R_2	1	1 0 1 0
R_2	R_3	0	0 1 0 1
R_3	R_0	1	0 1 0 1

State table of the mode-select FSM

H	$Q_{i+1}Q_i$	$A_i\ B_i$
0	0 0	0 1
0	0 1	0 0
0	1 1	1 0
1	1 1	1 0
1	1 0	1 1
1	0 0	0 1

Truth table of DC_i

Figure 9 Test generator for an *n*-bit 74181-style ALU.

increases, and it amounts to 11.4% for a 32-bit ALU.

Multiply-Add Unit. Our next example introduces another important arithmetic operation, multiplication. The high-level model and some implementation details of the target $n \times n$-bit multiply-add unit (MAU) $M_{MAU}(n)$ are shown in Figure 10. The MAU composed of a cascaded sequence of carry-save adders followed by a CLA in the last stage. This design is faster than a normal multiply-add unit where the last stage is a ripple-carry adder [4] [16].

Following our general methodology, we first analyze a small version of MAU, in this case, the 4-bit case. Again the tests for the CLA (Table 2) are traced back to the primary inputs through the cell array. The primary input signals are selected to preserve the shifting structure of the CLA tests. The resulting MAU tests do not test the cell array completely—two SSL faults per cell

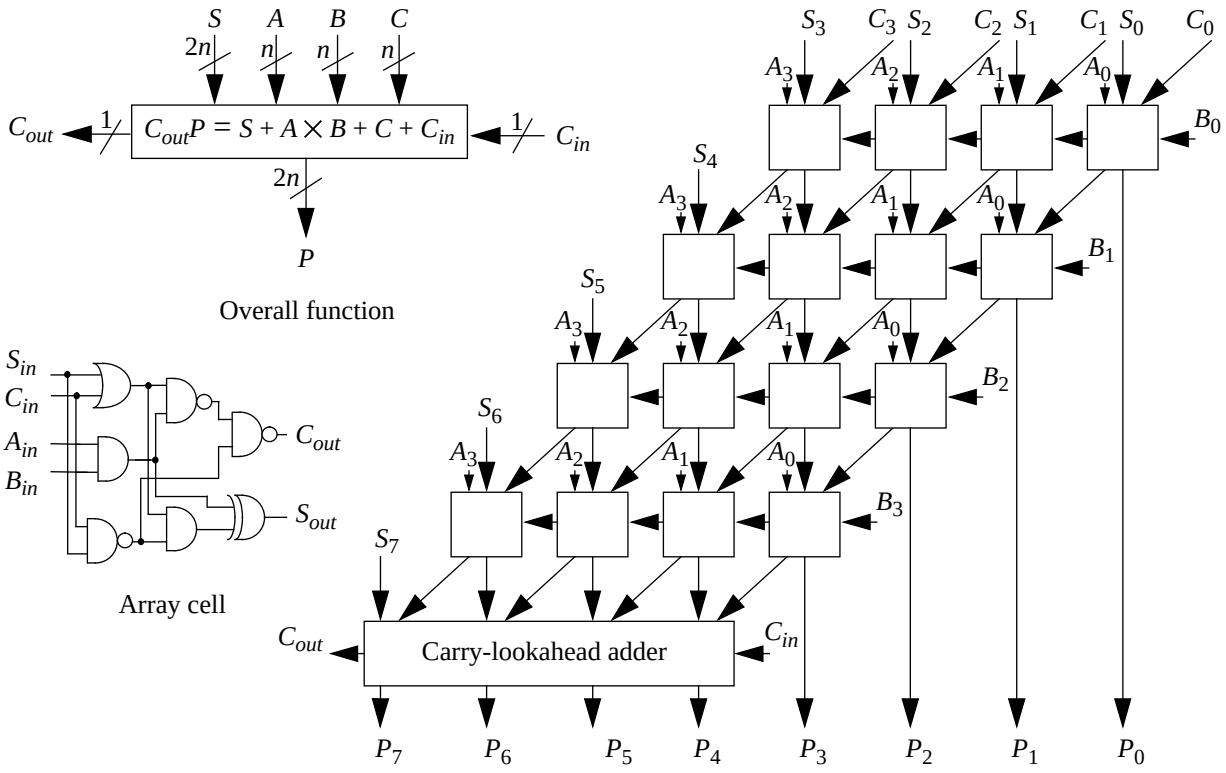


Figure 10 High-level model for the multiply-add unit.

remain undetected. These undetected faults require two extra tests, leading to a complete test set of size 12. Once the possible test sets are determined, a sequence that has the desired SC structure is constructed. Table 5 shows a possible test sequence $S_{\mathrm{MAU}}(4)$ of size 20 for $M_{\mathrm{MAU}}(4)$. This test sequence can be easily extended to $M_{\mathrm{MAU}}(n)$ with a resultant test set of size $4n + 4$.

A test generator $\mathrm{TG}_{\mathrm{MAU}}(n)$ for $M_{\mathrm{MAU}}(n)$ in the target style is shown in Figure 11. Since the test sequence size is $4n + 4$ and the general test generator TG(n) has $k(2n + 2)$ states, the mode-select FSM has $k = 2$ states (one flip-flop). The state table of the mode-select FSM and the truth table for DC_i are shown in Figure 11. The hardware overhead of $\mathrm{TG}_{\mathrm{MAU}}(n)$ is estimated to be only 0.8% for a 32×32-bit multiply-add unit.

Booth multiplier. Our technique can be applied with some minor modifications, to a fast Booth multiplier that is composed of a cascaded sequence of carry-save adders followed by a final stage consisting of a 2n-bit CLA [4]. Our design is faster than the usual Booth multiplier where the last stage is a ripple-carry adder; test generation has been studied before only for the slower, ripple-carry design [12]. We have been able to derive a complete scalable test sequence of size $4n + 14$

Table 5 Complete and near-minimal SC-style test sequence for the multiply-add unit.

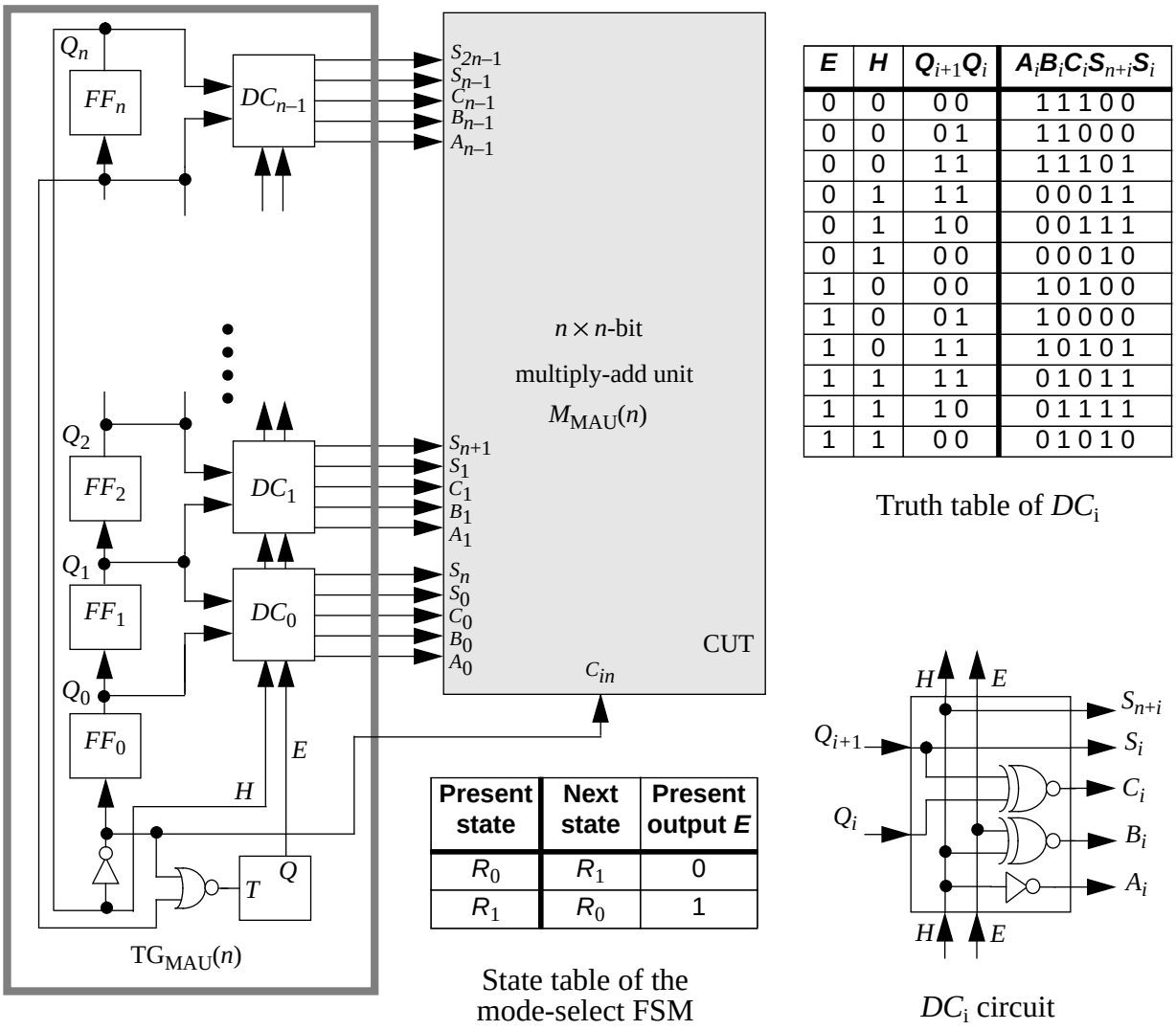
Test #	$A_3B_3C_3S_7S_3$	$A_2B_2C_2S_6S_2$	$A_1B_1C_1S_5S_1$	$A_0B_0C_0S_4S_0$	C_{in}
1	11100	11100	11100	11100	1
2	11100	11100	11100	11000	1
3	11100	11100	11000	11101	1
4	11100	11000	11101	11101	1
5	11000	11101	11101	11101	1
6	00011	00011	00011	00011	0
7	00011	00011	00011	00111	0
8	00011	00011	00111	00010	0
9	00011	00111	00010	00010	0
10	00111	00010	00010	00010	0
11	10100	10100	10100	10100	1
12	10100	10100	10100	10000	1
13	10100	10100	10000	10101	1
14	10100	10000	10101	10101	1
15	10000	10101	10101	10101	1
16	01011	01011	01011	01011	0
17	01011	01011	01011	01111	0
18	01011	01011	01111	01010	0
19	01011	01111	01010	01010	0
20	01111	01010	01010	01010	0

for the CLA-based Booth multiplier. The corresponding test generator TG(n) contains a TR counter with $n + 1$ flip-flops and a 10-state mode-control FSM with 5 flip-flops. The hardware overhead is estimated to be 5.3% for a 32×32-bit multiplier.

6 DISCUSSION

We have presented a new approach to the design of scalable hardware test generators for BIST, and illustrated it for several practical datapath circuits. The resulting test generators produce complete and extremely small test sets; they are of minimal or near-minimal size for all examples covered. Small test sets of this kind are essential for the on-line use of BIST, especially in applications requiring fast arithmetic techniques like carry-lookahead, for which previously proposed BIST schemes are not well suited. The TGs proposed here also have low hardware overhead, and are easily expandable to test much larger versions of the same target CUT.

Table 6 summarizes the results obtained for the scalable TGs we have designed so far. The first part of the table contains the results for the circuits discussed in Sections 4 and 5. The average hardware overhead for the ALU, MAU, and Booth multiplier with n = 32 is around 6%. The table



E	H	$Q_{i+1}Q_i$	$A_iB_iC_iS_{n+i}S_i$
0	0	0 0	1 1 1 0 0
0	0	0 1	1 1 0 0 0
0	0	1 1	1 1 1 0 1
0	1	1 1	0 0 0 1 1
0	1	1 0	0 0 1 1 1
0	1	0 0	0 0 0 1 0
1	0	0 0	1 0 1 0 0
1	0	0 1	1 0 0 0 0
1	0	1 1	1 0 1 0 1
1	1	1 1	0 1 0 1 1
1	1	1 0	0 1 1 1 1
1	1	0 0	0 1 0 1 0

Truth table of DC_i

Present state	Next state	Present output E
R_0	R_1	0
R_1	R_0	1

State table of the mode-select FSM

DC_i circuit

Figure 11 Test generator for an $n \times n$-bit multiply-add unit.

also indicates how the overhead decreases as n increases from 4 to 32. The overhead for the MAU shrinks by 90%, and the average decrease for all the circuits is 61%.

When applying BIST in a system, designers usually try to take advantage of existing flip-flops and logic already present in or around the CUT. For a typical datapath in, say, a digital signal processing circuit, all the data inputs to ALUs or multipliers come from a small register file. These registers can be designed to be reconfigured into TR counters like that in Figure 5, thus eliminating the need for special flip-flops in SG. Similar schemes have been proposed in prior techniques such as BILBO [6]. Moreover, it may be possible to share the resulting SGs among several CUTs. Multiplexing logic will then be needed to select the DCs for individual CUTs during test mode but

Table 6 Summary of the scalable test generator examples.

Circuit(s)		SSL fault coverage	Regular test set size	Hardware overhead %			
				$n = 4$	$n = 8$	$n = 16$	$n = 32$
Carry-lookahead adder (CLA)		100%	$2n + 2$	45.5	40.1	36.9	35.8
Arithmetic-logic unit (ALU)		100%	$8n + 8$	23.2	16.1	12.9	11.4
Multiply-add unit (MAU)		100%	$4n + 4$	7.8	3.5	1.6	0.8
Booth multiplier		100%	$4n + 14$	32.9	18.0	9.9	5.3
A combination of ALU, MAU, and registers	Separate TGs	100%	$8n + 8$	9.8	5.7	3.3	1.8
	Combined TG			6.2	3.6	2.1	1.1

circumvent them during normal operation. For a small additional increase in circuit complexity, time-multiplexing can be used to select the DCs in test mode, while avoiding the performance penalty associated with multiplexers.

In some cases, it may be feasible to share the entire TG. To illustrate this possibility, consider an n-bit ALU, an $n \times n$-bit MAU, and a register file connected to a common bus. A single, reconfigurable TG attached to the bus can test both arithmetic units. The results of this approach are summarized in Table 6 for various values of n, and suggest that replacing separate TGs for the ALU and MAU by a single combined TG reduces overhead by about a third.

Our TG designs shed some light on the following interesting, but difficult question: How much overhead is necessary for built-in test generation? As we noted in the incrementer case, the size of the $TG_{inc}(4)$ must be close to minimal for any TG that is required to produce a complete test sequence of near-minimal length. The same argument applies to $TG_{CLA}(4)$, since it has 5 flip-flops in SG and a small amount of combinational logic in DC; any test generator G(4) producing the same number of tests (12) must contain at least 4 flip-flops in its SG. In general, the overhead of a TR-counter-based design TG(n) scales up linearly and slowly with n. The number of flip-flops in some other test generator G(n) may increase logarithmically with n, but the combinational part of G(n) is likely to scale up at a faster rate than that of TG(n). This suggest that even if the overhead of TG(n) is considered high, it may not be possible to do better using other BIST techniques under similar overall assumptions. If the constraints on test sequence length are relaxed, simpler TGs for datapath circuits may be possible, but such designs have yet to be demonstrated.

REFERENCES

[1] V. D. Agarwal and E. Cerny, "Store and generate built-in testing approach", *Proc. International Symposium on Fault-Tolerant Computing*, 1981, pp. 35–40.

[2] S. B. Akers, "Universal test sets for logic networks", *IEEE Transactions on Computers*, Vol. C-22, September 1973.

[3] S. B. Akers and W. Jansz, "Test set embedding in a built-in self-test environment", *Proc. International Test Conference*, 1989, pp. 257-263.

[4] M. Annaratone, *Digital CMOS Circuit design*, Kluwer Academic Publishers, Boston, 1986.

[5] S. Boubezari and B. Kaminska, "A deterministic built-in self-test generator based on cellular automata structures", *IEEE Transactions on Computers*, Vol. 44, pp. 805-816, June 1995.

[6] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Self-Test for VLSI: Pseudorandom Techniques*, Wiley, 1987.

[7] K. Chakrabarty, B. T. Murray, J. Liu, and M. Zhu, "Test width compression for built-in self testing", *Proc. International Test Conference*, 1997, to appear.

[8] C.-A. Chen and S. K. Gupta, "A methodology to design efficient BIST test pattern generators", *Proc. International Test Conference*, 1995, pp. 814-823.

[9] W. Daehn and J. Mucha, "Hardware test pattern generation for built-in testing", *IEEE Test Conference*, 1981, pp. 110-113.

[10] R. Dandapani, J. H. Patel, and J. A. Abraham, "Design of test pattern generator for built-in self-test", *Proc. International Test Conference*, 1984, pp. 315-319.

[11] C. Dufaza and G. Cambon, "LFSR based deterministic and pseudo-random test pattern generator structures", *Proc. European Test Conference*, 1991, pp. 27-34.

[12] D. Gizopoulos, A. Paschalis, and Y. Zorian, "An effective BIST scheme for Booth multipliers", *Proc. International Test Conference*, 1995, pp. 824-833.

[13] M. C. Hansen and J. P. Hayes, "High-level test generation using physically-induced faults", *Proc. VLSI Test Symposium*, 1995, pp. 20-28.

[14] S. Hellebrand et al., "Pattern generation for deterministic BIST scheme", *Proc. International Conference on Computer-Aided Design*, 1995, pp. 88-94.

[15] S. Hellebrand, S. Tarnick, and J. Rajski, “Generation of vector patterns through reseeding of multiple-polynomial linear feedback shift registers”, *Proc. International Test Conference*, 1992, pp. 120–128.

[16] I. Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, Englewood Cliffs, N. J., 1993.

[17] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall, Englewood Cliffs, N. J., 1986.

[18] F. Muradali, V. K. Agarwal, and B. Nadeau-Dostie, “A new procedure for weighted random built-in self-test”, *Proc. International Test Conference*, 1990, pp. 660–669.

[19] B. T. Murray and J. P. Hayes, “Testing ICs: Getting to the core of the problem”, *IEEE Computer*, Vol. 29, pp. 32-45, November 1996.

[20] B. Nadeau-Dostie, A. Silburt, and V. K. Agarwal, “Serial interfacing for embedded memory testing”, *IEEE Design and Test*, vol. 7, no. 2, pp. 52–63, April 1990.

[21] M. Nicolaidis, “Test pattern generators for arithmetic units and arithmetic and logic units”, *Proc. European Test Conference*, 1991, pp. 61-71.

[22] K. K. Saluja, R. Sharma, and C. R. Kime, “A concurrent testing technique for digital circuits”, *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 1250-1259, December 1988.

[23] N. R. Saxena and J. P. Robinson, “Accumulator compression testing”, *IEEE Transactions on Computers*, Vol. C-35, pp. 317-321, April 1986.

[24] Texas Instruments, *The TTL Logic Data Book*, Dallas, 1988.

[25] N. A. Touba and E. J. McCluskey, “Synthesis of mapped logic for generating pseudorandom patterns for BIST”, *Proc. International Test Conference*, 1995, pp. 674–682.

[26] B. Vasudevan et al., “LFSR-based deterministic hardware for at-speed BIST”, *Proc. VLSI Test Symposium*, 1996, pp. 201-207.