METHODS FOR SYNTHESIZING TESTABLE SEQUENTIAL CIRCUITS

Kwang-Ting Cheng and Vishwani D. Agrawal

Kwang-Ting Cheng *and* ***Vishwani D. Agrawal*** *are at AT&T Bell Laboratories, Murray Hill, New Jersey. Mr. Cheng is a member of technical staff in the Computing Systems Technology Research Department, where he conducts research on CAD for VLSI, with a special focus on testing and logic synthesis. He joined AT&T in 1988 with a B.S. in electrical engineering from the National Taiwan University, Taipei, Taiwan, and a Ph.D in electrical engineering and computer science from the University of California at Berkeley. Mr. Agrawal is a distinguished member of technical staff in the Computer Systems Research Department, where he researches new methods for VLSI testing, synthesis for testability, neural network applications, and parallel processing. He joined AT&T in 1978 with a B.Sc. from Allahabad University,* (continued on page 86)

We present three approaches to designing testable sequential machines. (Testability, in the present context, refers to the ability to generate tests. Testable synthesis guarantees high fault coverage by using an automatic test generator.) In the first approach, we develop a partial scan method in which scan flip-flops are selected to break up the cyclic structure of the sequential circuit. In the second approach, we present a novel state assignment method that results in reduced feedback or pipeline-like structure. The third approach, also applicable to finite state machines, embeds a suitably designed test machine in the given specification before synthesis.

Introduction

Most digital systems operate sequentially. This means the output of the system depends on the input and on the state of the internal memory elements. The difficulty of controlling and observing the memory elements can be so great that, in one design method, all memory elements are made directly accessible for testing.[1] The hardware needed here can be large, and is not acceptable in some very-large-scale integration (VLSI) applications. In this paper we advance several alternative approaches aimed at synthesizing circuits for testability with minimal overhead.

The first method introduces a novel partial scan procedure[2] where a subset of flip-flops is selected to break up the cyclic structure of the circuit. Thus, the present state of a flip-flop may not depend on its own state at any previous time, making test generation for the cycle-free structure relatively simple. The partial scan technique is suitable for large sequential circuits that normally combine datapath and control logic sections.

The second method applies to sequential machines normally encountered in the control logic. Again, emphasizing a cycle-free design, we develop a new state assignment method that produces highly testable designs.[3] And the third method presents a general architecture of testable design.[4] The test function is added to the required function and both functions are synthesized together.

Panel 1. Terms and Acronyms in This Paper

CAD	computer-aided design
CMOS	complementary metal-oxide semiconductor
DS	distinguishing sequence
FSM	finite state machine
I/O	input/output
VLSI	very-large-scale integration

Partial Scan

It has long been recognized that the cyclic structure of sequential machines is largely responsible for test generation complexity.[5] (For synchronous circuits, a *cycle* is defined as feedback among flip-flops.) Table I shows the test generation results obtained from a sequential logic test generation program STG3.[6] The first circuit is a traffic light controller (TLC); the other (CHIP-A) is a complementary metal-oxide semiconductor (CMOS) chip designed for a graphics terminal. Even though CHIP-A is three times larger than TLC, our test generator processed it in one-fifth the time, and produced a significantly higher fault coverage.

Table I. Test Generator for Two Sequential Circuits

	Circuit	
	TLC	CHIP-A
Number of gates/FFs	355/21	1112/39
Test generation CPU seconds	1246	269
Fault coverage	89.0%	98.8%
Sequential depth	14	14
Number of cycles	150317	39

The results in Table I show that gate count is not a key determinant of test generation complexity: and sequential depth, often cited as a circuit's *measure of sequentiality*, does not explain the result. The *sequential depth*, defined as the most flip-flops on a path between inputs and outputs, is 14 for both circuits. However, if we consider the *cycles* formed by flip-flops, CHIP-A contains only 39 cycles, and no cycles greater than length one. A length-one cycle (or loop) is formed when a flip-flop's output feeds back into its own input after passing only through combinational logic. TLC, on the other hand, has many more cycles (some of them long) that are responsible for the higher complexity of test generation.

Test Length and Cycles. Consider a directed graph $G = \{V,E\}$ of a synchronous sequential circuit. A vertex v_i in this graph represents a flip-flop (or state variable) i. A directed edge from vertex v_i to vertex v_j implies a combinational path from flip-flop i to flop-flop j. Only the memory elements and dependencies between memory elements are represented in this graph.

Figure 1 illustrates a circuit with six flip-flops. Signal flow through combinational elements is represented by dashed lines. The circuit is synchronous and all flip-flops are controlled by a single clock signal that is not shown. The graph of this circuit is shown in Figure 2.

We can classify the directed graph of a circuit into three categories:

- Acyclic directed graph
- Directed graph with only self-loops
- Directed graph with cycles of two or more vertices.

The effect of cycles on testing complexity was recognized by early workers.[7] Our analysis may be an oversimplification, but still shows some of the complexity of test generation. A more rigorous testability analysis of cycles may be found in a recent paper.[8]

We define the *distance* along a directed path between two vertices as the number of vertices on that path. The *sequential depth* of the circuit is the distance along the longest path in its graph. For example, the sequential depth of the circuit in Figure 1 is five. If the graph is acyclic (i.e., there is no feedback among sequential elements) then we can levelize the graph. For a circuit with depth D, any single fault can be tested by at most D test vectors.[5] The size of a complete test set also is bounded by $D \cdot 2^n$, where n is the number of primary inputs. Obviously, this type of sequential circuit will be easy to analyze with a sequential test generator.

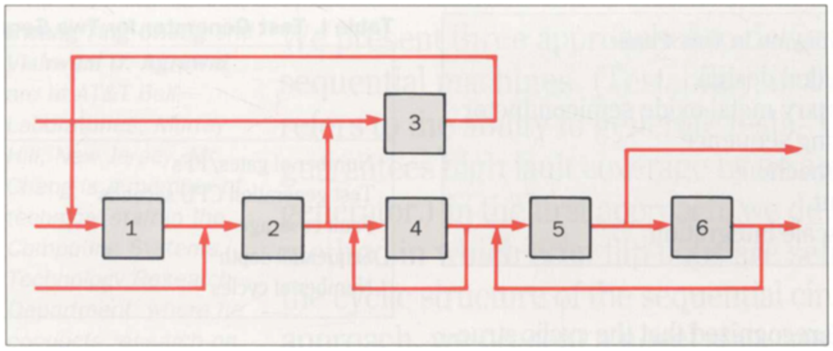


Figure 1. A circuit with six flip-flops.

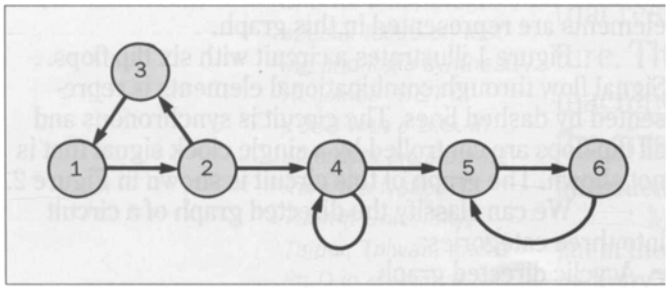


Figure 2. Graph of the circuit in Figure 1.

Consider the second type of structure, i.e., circuits with only self-loops. Here, a flip-flop's output may link to its input through combinational logic. For these flip-flops, we can collect all combinational logic between the output and input of the flip-flop to create a 2-state finite state machine (FSM), and use a single vertex to represent this FSM by a reconstructed graph. Thus, depending on whether the corresponding flip-flop has a self-loop, in the new representation a vertex can either represent a generalized 2-state machine or a single flip-flop. The new directed graph is acyclic again. However, two patterns at the inputs of the vertex may be needed to justify a logic value at the output of a vertex that represents a generalized 2-state machine. But only one pattern is needed for a vertex that represents a single flip-flop without self-loop. This follows from the assumption that the 2-state machine is initializable to a given state by one pattern. If this initial state is different from the required state, then one more pattern will be needed to move the machine to the other state. Using the pipeline argument, a test can be constructed with at most $2 \cdot D$ vectors.

For the third type of graph, with longer cycles, we again reconstruct an acyclic graph. A vertex in this new graph is a finite-state machine that can have up to 2^L states where L is the maximum length of any cycle. We assume that a vertex can be initialized to some state by a vector. To set the node in the desired state may take at most $2^L - 1$ vectors. In general, the depth of the reconstructed graph will be less than D. However, D is still an upper bound for depth. Thus, the length of a test sequence can be $D \cdot 2^L$. Most faults in the circuit can be tested by shorter sequences. We must remember that our derivation assumes a specific initialization procedure. Also, different test generators may pack vectors differently while creating test sequences. Therefore, this sequence length formula should be considered only an average measure of the test length. Our experience shows that in a few cases the length can be larger. A theoretical upper bound would be much higher.

The above analysis shows that test length and cycle length are related (cycle length is defined as the maximum number of flip-flops in a feedback cycle). If a sequential circuit test generator employing the time frame expansion technique is used, the number of processed time frames is equal to the test length. The longer the test length, the longer the test generation time, and the higher the probability the fault will be dropped because of a practical time limit imposed on the program.

To support this argument, we analyzed the structure of the two circuits in Table I. CHIP-A's cycle length is 1 and the TLC's is 4. Also, the longest test for a single fault in TLC has 243 vectors. The sequential depth for this circuit is 14; it has cycles of length 4. Thus, our formula gives a test length of 224. The longest sequence for CHIP-A has 102 vectors. Obviously, TLC presents a more difficult test generation problem than CHIP-A. The high fault coverage and low computation time for CHIP-A also show that self-loop structure can be handled successfully by a sequential test generator.

Scan Register Design. A scan flip-flop is assumed to break the circuit for test generation by making its I/O signals observable and controllable like primary outputs and inputs. This assumption is justified for a partial scan circuit if a separate control on the clock of scan flip-flops is available. To reduce the test generation complexity, we select a minimal set of flip-flops to eliminate all cycles. Self-loops or cycles of unit length are not broken to keep the scan overhead low because the number of self-loops in some circuits can be large.

A simple example of this design is shown in Figure 3. The original circuit (Figure 1) has only one clock, shown as CLOCK1. It is clear from Figure 2 that if flip-flops 2 and 5 are scanned, then the circuit will not have any cycles other than the self-loop of flip-flop 4. In Figure 3, flip-flops 2 and 5 have been replaced by scan flip-flops (shown as larger blocks) and SCANIN, SCANOUT, and MODE signals have been incorporated.[1] A new clock signal, CLOCK2, controls the scan flip-flops. In the normal mode of the circuit, CLOCK1 and CLOCK2 will be identical signals. However, in the scan mode CLOCK2 alone will be activated. Thus, while the states of scan flip-flops are being shifted in, all other flip-flops will retain their states.

Flip-Flop Selection Algorithm. A directed graph is first constructed from the circuit netlist. All self-loops are removed. The remaining graph contains either no cycle or cycles longer than 1. A program called `BreakLoop` is used to select a small subset of vertices to break all cycles. Because the problem of finding the vertex set that will break all cycles (feedback vertex set problem) is NP-complete,[9] heuristics must be used to bound the computation time. In `BreakLoop`, the process of finding cycles and selecting vertices is divided into several passes. In each pass, a time limit is set for the cycle-finding process. When the time limit is reached, the search for more cycles is suspended for the current pass.

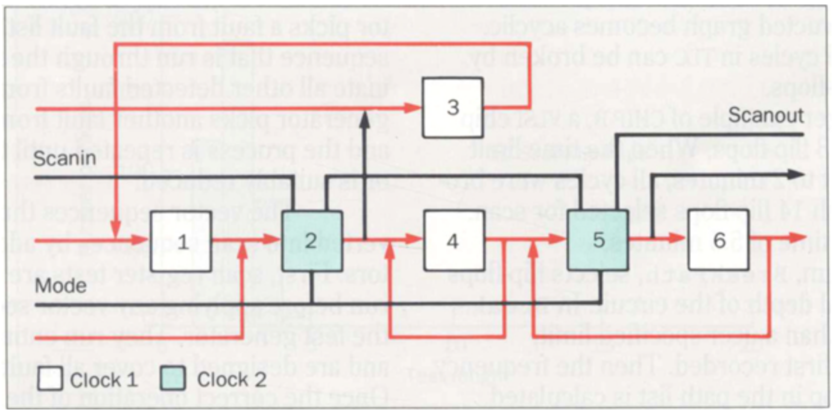


Figure 3. Design with flip-flops 2 and 5 scanned.

The selection of flip-flops here is based on the list of cycles found in the pass. Breaking a cycle will automatically break all bigger cycles where this cycle is embedded. Thus, the cycle list should only contain a set of representative cycles, so no cycle is embedded in any other cycle. To generate such a cycle list, when a new cycle is found, the current cycle list is first searched to check whether any cycle in the list covers or is covered by the new cycle. If any cycle in the list is embedded in the new cycle, the new one is discarded. If the new cycle is embedded in a cycle already in the list, the new one replaces the existing cycle. If neither case is found, the new cycle is added to the list. The algorithm for selecting vertices to break all cycles is as follows:

```
repeat
   for every vertex
      count the frequency of appearance
         in the cycle list.
   select the most frequently used vertex.
   remove all cycles containing the selected
         vertex from the cycle list.
until (cycle list is empty)
```

The selected vertices form a minimal set of vertices that break all cycles in the cycle list. These vertices are removed from the directed graph, and the graph is reconstructed. A new pass is then run. This process continues until the reconstructed graph becomes acyclic. With this procedure, all cycles in TLC can be broken by scanning nine of 21 flip-flops.

Consider a larger example of CHIP-B, a VLSI chip with 5,294 gates and 318 flip-flops. When the time limit for a single pass was set to 2 minutes, all cycles were broken in three passes, with 14 flip-flops selected for scan. It required a total CPU time of 5.5 minutes.

Another program, `BreakPath`, selects flip-flops to reduce the sequential depth of the circuit. In `BreakPath`, all paths longer than a user-specified limit, `PathThreshold`, are first recorded. Then the frequency of usage for each flip-flop in the path list is calculated. For a path of length D, if we scan the ith flip-flop, then the path will be broken into two shorter paths of length $i - 1$ and $D - i$, respectively. Clearly, selecting the flip-flop at the center of a path is the best choice for reducing the sequential depth of the circuit. Thus, in calculating the usage frequency, the flip-flops closer to the center of a path are weighted higher. The most frequent flip-flop is selected and the path list is updated. This procedure repeats until the lengths of all paths in the list are smaller than `PathThreshold`.

Test Generation. To effectively use the partial scan register for generating and applying tests, a separate scan clock is used. When shifting bits in or out of the scan register, the normal clock is held inactive to keep the states of the non-scanned flip-flops unchanged. An economical design of a scan flip-flop that combines the scan clock input with mode input is described later. With this design, the scan clock operation would not require any extra input pin over what is needed for the normal scan design.

After the scan flip-flops have been selected, the netlist is modified to create a test generation model. In this model, all scan flip-flops are removed, and their input and output signals are added to the primary output and primary input lists, respectively. A sequential circuit test generator and fault simulator are used to generate tests for faults in the modified netlist model. The test generator picks a fault from the fault list and produces a vector sequence that is run through the fault simulator to eliminate all other detected faults from the list. Next, the test generator picks another fault from the updated fault list, and the process is repeated until the list becomes empty or is suitably reduced.

The vector sequences thus generated are converted into scan sequences by adding two types of vectors. First, scan register tests are added. These tests are run before applying any vector sequences produced by the test generator. They run entirely in the scan mode and are designed to cover all faults in the scan register. Once the correct operation of the scan register is ensured,

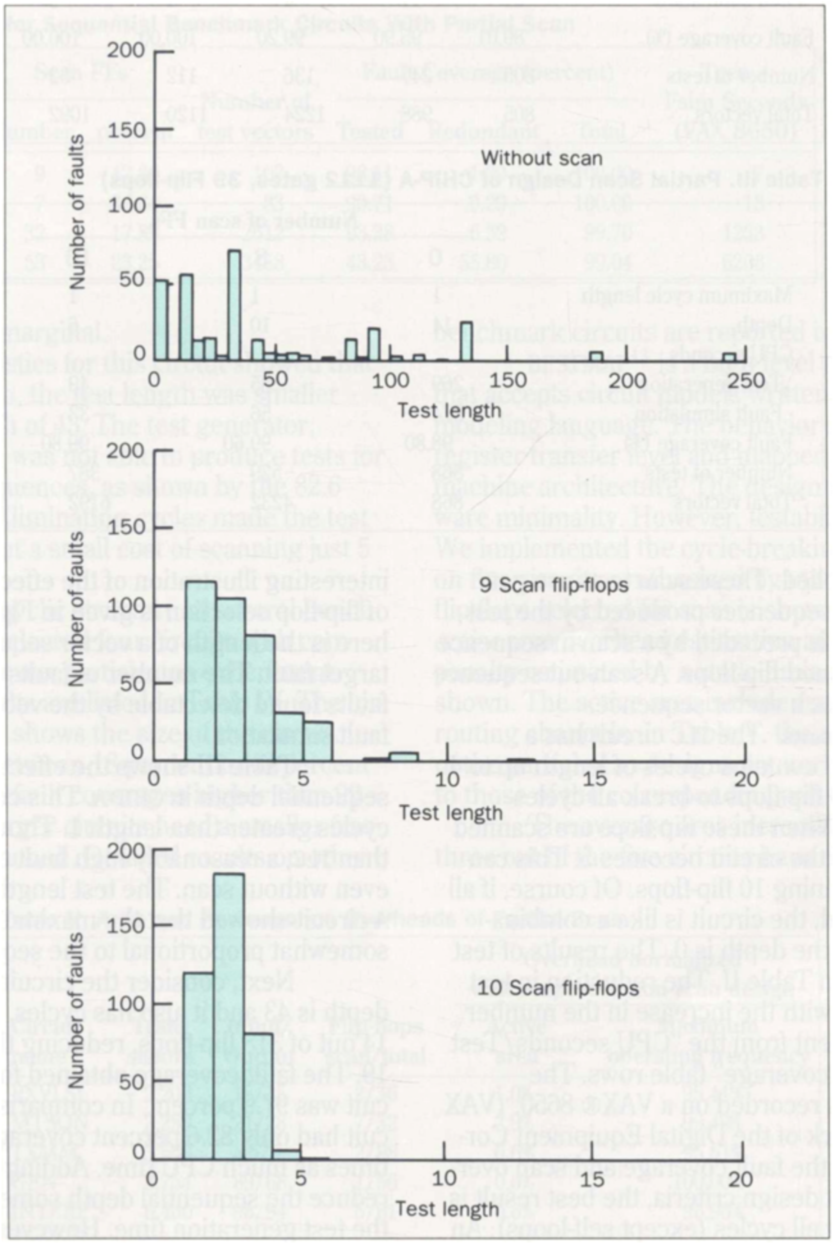


Figure 4. Test length statistics for the TLC circuit.

Table II. Partial Scan for TLC (355 gates, 21 Flip-flops)

	Number of scan FFs				
	0	4	9	10	21
Maximum cycle length	4	2	1	1	0
Depth	14	10	5	3	0
CPU seconds					
Test generation	1246	157	32	13	2
Fault simulation	61	11	4	4	2
Fault coverage (%)	89.01	95.90	99.20	100.00	100.00
Number of tests	805	247	136	112	52
Total vectors	805	988	1224	1120	1092

Table III. Partial Scan Design of CHIP-A (1112 gates, 39 Flip-flops)

	Number of scan FFs		
	0	8	16
Maximum cycle length	1	1	1
Depth	14	10	6
CPU seconds			
Test generation	269	85	49
Fault simulation	274	56	33
Fault coverage (%)	98.80	99.60	99.80
Number of tests	868	529	387
Total vectors	868	4232	6192

other scan tests are applied. These scan tests are generated from the vector sequences produced by the test generator. Each vector is preceded by a scan-in sequence to set the states of scanned flip-flops. A scan-out sequence is added at the end of each vector sequence.

Experimental Results. The TLC circuit has a sequential depth 14 and contains cycles of length up to 4. `BreakLoop` selected 9 flip-flops to break all cycles greater than length 1. When these flip-flops are scanned the sequential depth of the circuit becomes 5. This can be reduced to 3 by scanning 10 flip-flops. Of course, if all 21 flip-flops are scanned, the circuit is like a combinational circuit for which the depth is 0. The results of test generation are shown in Table II. The reduction in test generation complexity with the increase in the number of scan flip-flops is evident from the "CPU seconds/Test generation" and "Fault coverage" table rows. The reported CPU time was recorded on a VAX® 8650. (VAX is a registered trademark of the Digital Equipment Corporation.) Considering the fault coverage and scan overhead as most important design criteria, the best result is obtained by eliminating all cycles (except self-loops). An interesting illustration of the effectiveness of this method of flip-flop selection is given in Figure 4. The test length here is the length of a vector sequence generated for a target fault. The number of faults corresponds to the faults found detectable by the vector sequence through fault simulation.

Table III shows the effect of reducing the sequential depth in CHIP-A. This circuit does not have any cycles greater than length 1. Though the circuit is larger than TLC, a reasonably high fault coverage is obtained even without scan. The test length statistics for the CHIP-A circuit showed that the maximum test length was somewhat proportional to the sequential depth.

Next, consider the circuit CHIP-B. Its sequential depth is 43 and it also has cycles. `BreakLoop` identified 14 out of 318 flip-flops, reducing the sequential depth to 19. The fault coverage obtained for the partial scan circuit was 97.9 percent. In comparison, the unscanned circuit had only 82.6 percent coverage, and requires three times as much CPU time. Adding more scan flip-flops to reduce the sequential depth somewhat linearly reduced the test generation time. However, further increase in

Table IV. Test Generation for Sequential Benchmark Circuits With Partial Scan

Circuit Name	Total Number of FFs	Scan FFs Number	Scan FFs percent	Number of test vectors	Fault Coverage (percent) Tested	Redundant	Total	Tgen + Fsim Seconds (VAX 8650)
s400	21	9	42.86	107	98.11	1.89	100.00	7
s713	19	7	36.84	83	90.71	9.29	100.00	18
s5378	179	32	17.89	2612	93.38	6.32	99.70	1253
s9234	228	53	23.25	3458	43.23	55.80	99.04	6208

fault coverage was only marginal.

Test length statistics for this circuit showed that for the unscanned design, the test length was smaller than the sequential depth of 43. The test generator, because of its time limit, was not able to produce tests for faults requiring long sequences, as shown by the 82.6 percent fault coverage. Eliminating cycles made the test generation manageable at a small cost of scanning just 5 percent of flip-flops.

We studied the cyclic structure of several benchmark circuits.[10-11] We selected four circuits with complex cyclic structures for our partial scan experiment. The test generation results are listed in Table IV. The circuit name (first column) shows the size of the circuit. These circuits require between 18 percent to 43 percent scan flip-flops to achieve fault coverages higher than 99 percent. Interestingly, larger circuits need a smaller fraction of flip-flops to be scanned. Detailed results on other benchmark circuits are reported in a recent paper.[12]

BESTMAP[13] is a high level synthesis system that accepts circuit models written in a C-like behavior modeling language. The behavior is described at the register transfer level and mapped onto a finite state machine architecture. The design is optimized for hardware minimality. However, testability is not guaranteed. We implemented the cycle-breaking partial scan design on five circuits synthesized by BESTMAP. The number of flip-flops selected for scan is shown in Table V. The *active* area overhead estimation and the performance penalty estimated by a static timing analyzer[14] are also shown. The active area includes only the polycells and routing channels. In Table V, the area and performance of the partial scan designs are normalized with respect to those of the corresponding non-scan circuits.

The average area increase is 1.14 percent and three out of the five circuits have only a 0.6 percent

Table V. Area and Performance Overheads of Partial Scan

Circuit name	Tran-sistors	Input/ Output	Flip-flops scan/total	Overhead normalized with respect to non-scan design: Active area	Maximum operating frequency
RXVLSI	2,208	15/15	2/36	1.0%	95.41%
MTSQN	2,250	21/28	7/32	2.9%	99.49%
TXMM	3,560	45/9	2/98	0.6%	95.61%
GEN	3,754	89/68	1/56	0.6%	100.00%
INTERP	6,460	98/94	5/61	0.6%	96.96%

Table VI. Test Generation Results for the Partial Scan Circuits

Circuit Name	Total Faults	Detected Faults	Redundant Faults	Fault Coverage	CPU Time (SUN4/260)
TXMM	1803	1754	15	98.09%	2h30m
RXVLSI	1382	1132	234	98.61%	1h20m
MTSQN	1512	1381	108	98.36%	46m
INTERP	4790	4280	510	100.00%	22m
GEN	2381	2230	146	99.78%	51m

increase. The average performance degradation is 2.51 percent. The performance degradation can be further reduced by excluding the flip-flops on critical paths from the scan register. This can easily be done in our partial scan method.

Table VI shows the result of test generation by the STG3 program[6] for the partial scan circuits. Because of the reduced test complexity from the partial scan, STG3 was able to identify many redundant faults. In computing the fault coverage reported in Table III, the redundant faults were considered *detected.* This coverage exceeded 98 percent in all five cases.

Our experiment suggests that the partial scan method that attempts to break cycles, coupled with the sequential circuit test generator (STG3) program, offers an attractive design-for-testability alternative for BESTMAP users.

Table VII gives the results of partial scan design on three real designs. The test vectors given are the number of vectors produced by STG3. If a single scan register is used in the final implementation, the test length after adding the scan sequences will be equal to the number of test vectors times the number of scan flip-flops.

Flip-Flops for Partial Scan Design. Two types of flip-flops have been used in scan design. In one type, known as the level-sensitive design,[15] two non-overlapping clock signals are used in normal mode operation of the master and slave latches. A third clock signal, the *scan clock*, and the normal mode slave clock are used together in the scan operation. In a partial scan design, only a select subset of flip-flops will receive the scan clock signal. The scan operation, therefore, will not disturb the state of unscanned flip-flops. Compared to a complete scan design, the overhead in partial scan design is thus reduced in direct proportion to the fraction of scanned flip-flops.

An alternative design practice, however, uses a single clock signal. A master-slave operation is achieved either by using two latches that are sensitive to different levels of the clock signal, or by using edge-triggered latches. For scan design, flip-flops are preceded by multiplexers to switch between the normal and scan signals.[1] All multiplexers are controlled by a mode control signal added to the circuit. In this design, the same clock signal is used for both normal and scan functions. In a partial scan circuit, the unscanned latches must hold their normal states while data are being scanned in. One way to accomplish this is to use an additional clock pin that feeds only the scanned flip-flops. In the normal mode, both clock pins will carry the same signal. In the scan

Table VII. Partial Scan Results for Three Real Designs

	Circuit		
	DEVICE	CMULDEM*	SSBC
Gate count	5546	3754	7771
Flip-flops scan/total	50/256	50/229	178/482
Test vectors	7,366	6,521	4,865
Fault coverage (%)	98.52	97.14	99.45
CPU hours (SUN4/260)	4.8	9.0	3.8

* CMULDEM has 2 system clocks

mode, the clock signal is applied only to the second clock pin. Thus, besides the other signals needed for a complete scan design, the partial scan design may need one extra clock pin. The extra clock pin can be eliminated if the scan flip-flop, shown in Figure 5, is used. The two signals, MODE and CLOCK, are routed to all scanned flip-flops. The same CLOCK signal is routed to all unscanned flip-flops.

Signal waveforms for the two modes are easily derived. Assume the latches in Figure 5 are active when their clock input is high, and that the multiplexer selects DATA when its control signal is low; otherwise it selects SCANIN. In the normal mode, MODE is held *low* and CLOCK carries a periodic high (master active) and low (slave active) signal. Thus DATA inputs are clocked in all flip-flops. In the scan mode, CLOCK is held low. Thus, all unscanned flip-flops not receiving MODE are unaffected. For scanned flip-flops, MODE now carries a clock-like (periodic high and low) signal that activates their master and slave latches. Also, when the master latch is active, that is, MODE is high, SCANIN is selected by the multiplexer. In this design, the need for additional clock pins is eliminated.[16]

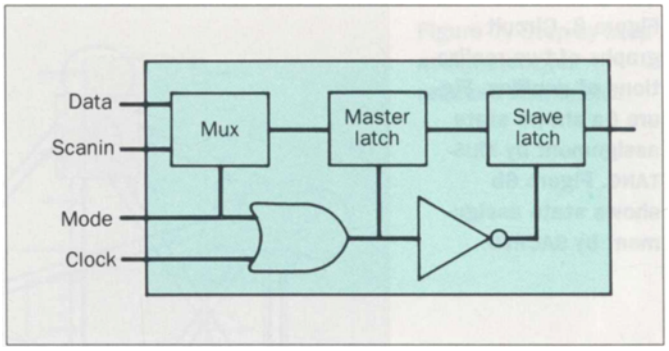


Figure 5. Scan flip-flops for single clock design.

Finite-State Machines With Reduced Feedback

A finite state machine (FSM) can be classified as a *finite-memory* or *infinite-memory* machine.[17] For a finite-memory machine with memory *m*, the response and the state depend on a finite number *m* of previous inputs. Such machines are also known as *definite machines.*[7] They can be implemented without cycles and thus are easier to test. The machines whose response and the state might depend on an infinite number of previous inputs are called the *infinite-memory machines.*

Implementing a finite state machine is strongly influenced by the state assignment. In our experience, even a finite-memory machine with an improper state assignment may be implemented with feedbacks, making test generation difficult.[5] Most previous work on state assignment is targeted at reducing hardware (i.e., the number of gates or the chip area). In this section, we describe a new method for state assignment aimed at reducing feedbacks in the implemented circuit. Ideally, for a finite-memory machine, a pipeline structure is produced. For other machines, our state assignment will still try to reduce feedbacks. Such reduced feedback design has advantages. For example, in our partial scan method, where the scan flip-flops are used only to break cycles,[2] only a few flip-flops will need scan modification.

The impact of our state assignment procedure on the amount of hardware is not obvious. However, several examples show no significant increase when compared to other state assignment methods. The examples show a significant effect on test generation through reduced run time and higher fault coverage.

Synthesis translates a given state table into a hardware implementation. In this process, the state assignment not only determines the complexity of combinational components but also is responsible for the sequential structure's complexity. After state assignment, the interdependency of state variables remains fixed through implementation. Therefore, test generation complexity often is also determined by the state assignment.

Example. Consider the finite 24-state machine *donfilex* from a recent synthesis benchmark collection.[18] We used a state assignment program, MUSTANG,[19] that incorporates a heuristic to produce an economical multi-

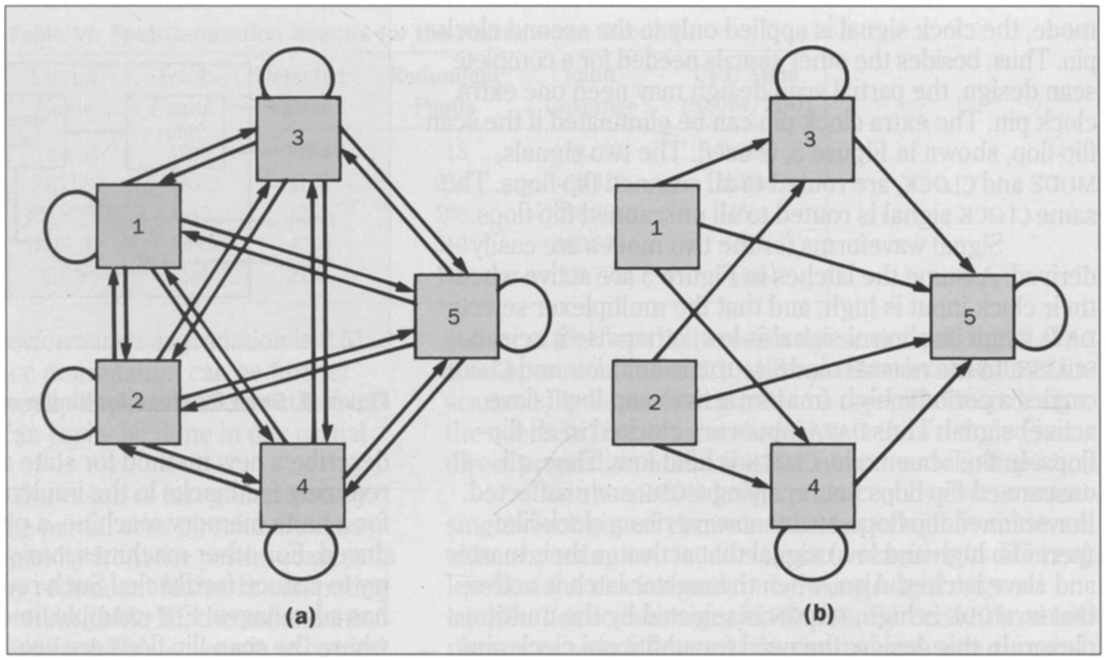


Figure 6. Circuit graphs of two realizations of *donflIex*. Figure 6a shows state assignment by MUSTANG. Figure 6b shows state assignment by SACRED.

level implementation. MIS,[20] a multilevel logic minimization program, was used to minimize combinational logic. And DAGON,[21] a technology mapping program, was used to translate the gate-level netlist into a standard cell implementation. Memory elements were finally added to complete the design of the state machine.

As explained before, the sequential complexity of this implementation can be analyzed using a circuit graph where vertices are the memory elements and the edges are the combinational signal dependencies. The graph for our example is shown in Figure 6a. To differentiate the state transition graph where the nodes (states) are normally shown as circles, we have used square vertices to represent state variables. It is a complete 5-vertex directed graph, i.e., every state variable depends on all five state variables. Notice that such a circuit graph is completely determined by the state assignment program. Logic minimization and technology mapping can only affect the combinational logic but not the sequential structure. An alternative state encoding devised to simplify the circuit graph (the encoding procedure will be described later) produced the much simpler (three self-loop) graph shown in Figure 6b. State variables 1 and 2 are completely controllable by primary inputs and do not depend on any other state variable.

Table VIII. Test Generator Results for Donflilex (24 States)

	method	
	MUSTANG	SACRED
Grid count	437	293
Test generation CPU seconds (SUN4/260)	1394	17
Test length	171	195
Fault coverage (%)	84.95	100.00

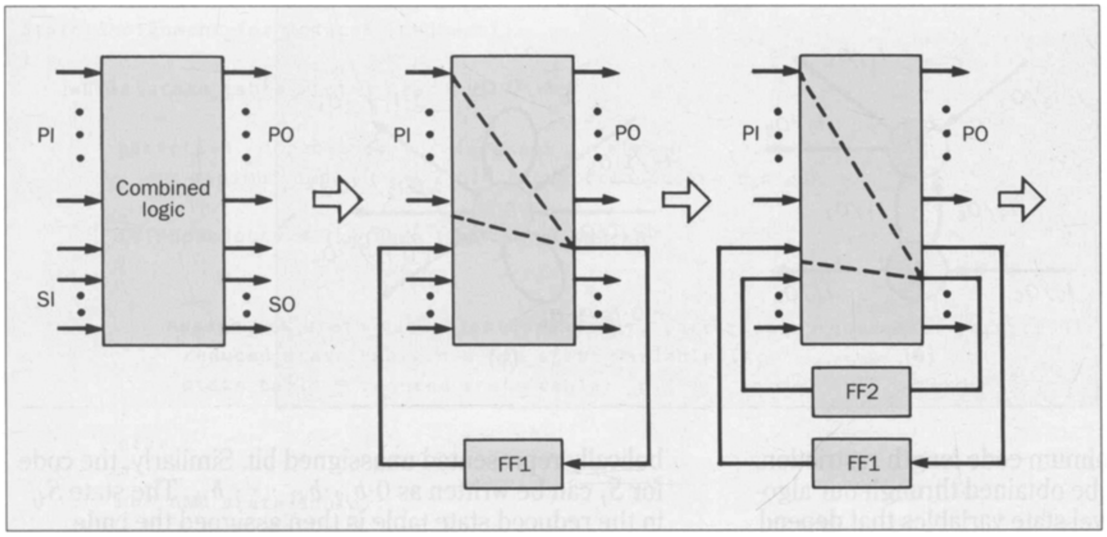


Figure 7. Step-by-step assignment for reduced feedback.

We used STG3[6] to generate test vectors for both implementations. The results appear in Table VIII, where our new state assignment is shown as SACRED (State Assignment for Cycle REDuction). The test generator spent 23 minutes of CPU time for the MUSTANG implementation and yet obtained less than 85 percent fault coverage. It took only 17 seconds to achieve 100 percent coverage for the SACRED implementation.

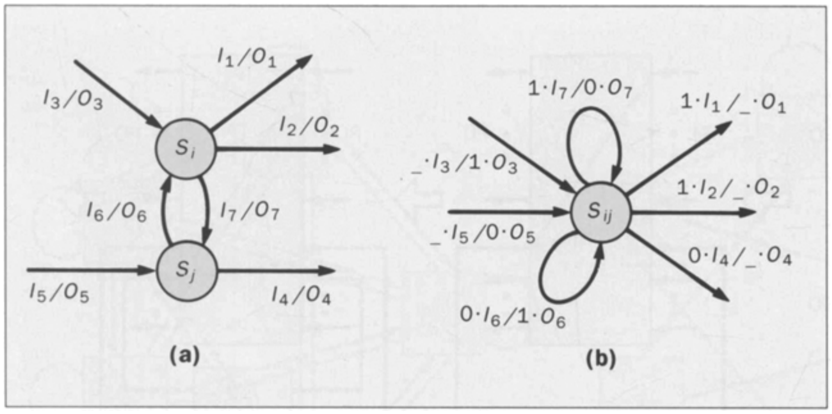
In Table VIII, we also give the grid count for both realizations. *Grid count* is an approximate measure of the circuit size in a standard cell design, and is closely related to the number of gates or transistors. While the new state assignment improved testability, it also used less hardware.

State Assignment Algorithm. The state assignment problem may be viewed as either an encoding or partitioning problem.[22-23] From the encoding viewpoint, a distinct code is assigned to each state in the state table. In the partitioning method used in SACRED, for each state variable Y_i, the states are partitioned into two sets. All states in one set are assigned 1 for Y_i while those in the other set are assigned 0. Our algorithm uses the minimum number of state variables, i.e., $\lceil \log n \rceil$ state variables for encoding n states. We can summarize the key ideas of the algorithm as:

- Use a multiple pass process assigning, in each pass, variables that depend only on primary inputs and other state variables assigned in previous passes. If no such variable can be found, then select those state variables for assignment that depend on the smallest number of other unassigned variables. Our state assignment idea differs from the reported work of Hartmanis and Stearns. However, we still use their algebraic theory of state partitioning.[24]
- After each pass, the state table is modified through a *state merging* technique where certain states are merged into a single state. This reduces the size of the state table for the next pass.

If m state variables are assigned in a pass and the state table has I inputs, O outputs, S states, and needs k bits to encode these states, i.e., $k = \lceil \log S \rceil$, then the reduced state table for the next pass will have $I + m$ inputs, $O + m$ outputs, and less than or equal to 2^{k-m} states.

If the machine is realizable as a feedback-free



I_1/O_1
I_3/O_3
S_i
I_2/O_2
I_6/O_6
I_7/O_7
S_j
I_5/O_5
I_4/O_4
(a)
$1 \cdot I_7/0 \cdot O_7$
$1 \cdot I_1/\_ \cdot O_1$
$\_ \cdot I_3/1 \cdot O_3$
$1 \cdot I_2/\_ \cdot O_2$
S_{ij}
$\_ \cdot I_5/0 \cdot O_5$
$0 \cdot I_4/\_ \cdot O_4$
$0 \cdot I_6/1 \cdot O_6$
(b)

Figure 8. Reduction of state diagram: (a) before merging, and (b) after merging.

structure under the minimum code length restriction, such a realization must be obtained through our algorithm. Here, the first-level state variables that depend only on primary inputs are assigned in the first pass. The second-level state variables that depend only on primary inputs and the first-level state variables are then assigned in the second pass. Subsequent passes are similar. The process is illustrated in Figure 7. PI and PO are primary inputs and outputs, respectively; and SI and SO are state inputs and outputs. In each pass, the newly assigned state variable only depends on PI and previously determined state variables.

When a feedback-free realization is not possible, a realization with minimum number of feedbacks (i.e., cycles) may be desired. Although our algorithm may not produce an optimal realization, the heuristics adopted usually produce a pipeline-like structure such that the number of cycles is minimized.

Example. Assume that one state variable Y_1 is assigned in a state table. This subdivides the states into two groups: the 1-set (ON-set) of Y_1 and the 0-set (OFF-set). Each state S_i in 1-set will be merged with some state S_j in the 0-set to produce a new state S_{ij} in the reduced state table. Thus, in the new state table, the number of states is reduced by a factor of two. Suppose S_i is encoded as $1 \cdot b_{i2} \cdot b_{i3} \cdots b_{ik}$, where b_i is a symbolically represented unassigned bit. Similarly, the code for S_j can be written as $0 \cdot b_{j2} \cdot b_{j3} \cdots b_{jk}$. The state S_{ij} in the reduced state table is then assigned the code $b_{ij2} \cdot b_{ij3} \cdots b_{ijk}$ where all bit positions are symbolic.

The construction of transition edges in the new graph is illustrated in Figure 8. Before merging, suppose the graph contains seven edges associated with the states S_i and S_j. Each edge has a label consisting of a binary input vector I and a binary output vector O. All seven edges appear in the reduced graph. Since one state variable is assigned, the width of input and output vectors is increased by one bit while the width of the state code is reduced by one bit. For an outgoing edge, the assigned state variable of the source state is added to the input vector. Thus, the edge label $I_1\ /O_1$ becomes $1 \cdot I_1\ /\_ \cdot O_1$. A 1 is appended to I_1 because the assigned state variable was 1 in S_i. The "_" appended to O_1 shows this value should be determined from the destination state of this edge not included in Figure 8. The state assignment of the destination state should be appended to the output vector in the label. For example, the label $I_3\ /O_3$ becomes $\_ \cdot I_3\ /1 \cdot O_3$. Again, the "_" in the input vector will depend on the source state of this edge. Since S_j belongs to the OFF-set of the assigned state variable, the edge $I_4\ /O_4$ transforms into $0 \cdot I_4\ /\_ \cdot O_4$ and $I_5\ /O_5$ transforms to $\_ \cdot I_5\ /0 \cdot O_5$. The

```
State_Assignment_for_Reduced_Feedback()
{
    while(state_table->total_states > 2)
    {
        partition, dependency, predecessor_partition
            = minimum_dependency_2_block_partition(state_table);
        if(dependency < $\lceil \log(\text{state\_table} \rightarrow \text{total\_states}) \rceil$)
        {
            assign_new_state_variables(dependency, partition, predecessor_partition);
            reduced_state_table = merge_state_variables();
            state_table = reduced_state_table;
        }
        else
        {
        MUSTANG(state_table);
            break;
        }
    }
}
```

Figure 9. The SACRED (State Assignment for Cycle REDuction) algorithm.

edges $I_6\ /O_6$ and $I_7\ /O_7$ that are between S_i and S_j appear as self-loops of S_{ij}.

The above illustration shows that for partially encoded states, the reduced state graph contains fewer states and longer edge labels. Also, the states in the reduced graph are completely uncoded. For further assignment of state variables, steps of variable assignment and state merging are repeated. This process will stop on two conditions: either only two states are left in the reduced state table, or each state variable in the reduced state table depends on all unassigned state variables. For the first case, either of the states is assigned 1 for the last state variable, and the remaining state is assigned 0. For the second case where no reduced feedback decision is possible, we use some alternative procedure, e.g., MUSTANG,[19] to find the encoding for the partially encoded table.

When a feedback-free realization is not possible, the following heuristics usually produce a pipeline-like structure:

- In each pass, select that state variable for assignment that depends on the smallest number of other unassigned variables.
- The state variables that have been assigned in the previous passes are usually more controllable. Thus, they may be treated as primary inputs for the following passes.

The detailed procedure for selecting the most controllable state variables and state mergings is discussed in a detailed presentation based on our research.[3]

Figure 9 shows a simplified pseudo-code of the SACRED algorithm. The term *dependency* is used here in an iterative sense. At any point in the procedure, a state variable is assigned to reduce its dependence on other unassigned state variables. Subroutine *minimum_dependency_2_block_partition* returns a least dependent 2-block partition (*partition*). It also returns the number of the state variable it depends on (dependency) and the corresponding predecessor partition of this 2-block partition (predecessor_partition). According to these returned values, if reduced dependency is possible, i.e., dependency is smaller than the number of unassigned state variables, a set of state variables is determined; thus codes are partially assigned. States are then merged according to the merging rules. The total number of states is now reduced. This process repeats with the reduced state table. The program stops either when only two states are left or when no reduced dependency is possible. As noted, in the latter case we may switch to an alternative state assignment procedure (e.g., MUSTANG) purely for logic minimization.

Experimental Results. In Table IX shows results for examples obtained from the MCNC 1987 Logic Synthesis Workshop[18], and some industrial examples. Our new algorithm, SACRED, was compared with two other algorithms, MUSTANG and MUSTANG.[19] After state encoding, MIS[20] was used for logic minimization and DAGON[21] was used to map the netlist onto a standard cell library. Memory elements were added to complete the design. We used a circuit structure analyzer program to calculate the total cycle length of the given circuit graph. The length of a cycle in the circuit graph is the number of vertices (i.e., flip-flops) contained in the cycle.[2] The total cycle length is calculated by totaling the length of all cycles in the circuit graph; it reflects the complexity of the circuit's cyclic structure. Finally, we generated tests using STG3.[6]

The third column in Table IX gives the grid count (not including the routing area) as a measure of the size of implementation. The test generation time shown is in CPU seconds on a SUN4/260™ workstation. (SUN4/260 is a trademark of Sun Microsystems, Inc.) The reported fault coverage includes the identified redundant faults. For most machines synthesized by SACRED, test generation is sped up tenfold. Surprisingly, most implementations synthesized by SACRED required less logic than those synthesized by MUSTANG.

A Testable Architecture For Synthesis

Finite state machine testing was addressed by Hennie,[29] who devised the *checking experiment*. In a checking experiment, an input sequence is applied to the machine so the output distinguishes the machine from all other machines with the same number of inputs and outputs, and the same or fewer states. Hennie's procedure of constructing an input sequence for the checking experiment is practical only when a distinguishing sequence (DS) exists. For a n-state machine, its distinguishing sequence will produce n different outputs depending on the initial state. Kohavi and Lavallee[30] identified a distinguishing sequence of bounded length as the requirement for diagnosability. Their *design for testability* added an extra output to make the DS shorter. Fujiwara et al.[31] further defined an *easily testable n*-state machine as one with synchronizing and transfer sequences of length $\log_2 n$. A *synchronizing sequence* is defined as an input sequence whose application is guaranteed to leave the FSM in a known final state regardless of its initial state. A *transfer sequence* is an input sequence that changes the state from a given initial state to a given final state.

For many FSMs with no distinguishing sequence, it is also possible to find *simple I/O sequences* as defined by Hsieh.[32] A simple I/O sequence distinguishes one specific state from all other machine states. Hsieh devised a reduced checking experiment by suitably ordering the simple I/O sequences of various states.

For most practical FSMs, these procedures produce long sequences. Though useful in simulation-based design verification, such tests are not commonly used in production testing. The testing problem, then, relates to finding a set of tests for modeled (usually stuck-type) faults. A commonly used design for testability is the scan method that converts the FSM testing problem to that of combinational logic testing.[1] In the scan method, and in many other methods, the circuit is modified after the design is completed. With tool-based synthesis, the advantage of logic optimization may be lost in such post-

Table IX. Experimental Results for SACRED

Circuit	Method	Grid count	Total cycle length	Test generation CPU seconds	Fault coverage (%)
train11	SACRED	149	12	4	100
11 states	MUSTANG-n	163	63	67	100
4 variables	MUSTANG-p	164	64	37	100
lion9	SACRED	118	4	0.7	100
9 states, 4 variables	MUSTANG-p	125	21	4.5	100
ex4	SACRED	221	38	15	100
14 states	MUSTANG-n	237	94	79	99.51
4 variables	MUSTANG-p	228	94	26	100
shift4	SACRED	72	0	0.1	100
16 states	MUSTANG-n	132	59	7.8	100
4 variables	MUSTANG-p	182	94	53.0	100
donfilex	SACRED	293	3	17	100
24 states	MUSTANG-n	437	625	1394	84.95
5 variables	MUSTANG-p	486	625	1924	85.82
red	SACRED	172	30	3.7	100
9 states	MUSTANG-n	191	94	19	100
4 variables	MUSTANG-p	170	64	16	100
malati	SACRED	338	469	32	100
24 states	MUSTANG-n	462	625	137	99.03
5 variables	MUSTANG-p	462	625	131	98.71

design modification.

Pradhan[33] and Bhattacharyya[34] proposed augmenting state tables to implement specific scan-like design. Reddy and Dandapani[35] described a procedure where the output and state equations of FSM are modified to incorporate the scan function in a programmable logic array. The main idea in our method is to economize on testability augmentation by reusing—wherever possible—the given function for testing. Though our methodology is general, the implemented test structure is customized for specific designs.

Figure 10 shows a high-level view of our proposed architecture.[4] It is recognized that the FSM *M1* to be synthesized may not be testable. We will call *M1* the ***object machine***. Testability refers here to the ability to generate tests. The machine is realized with memory elements and combinational logic. For the given object machine, we specify a test machine, *M2*, that has the same number of memory elements as the object machine. The physical memory elements, shown as flip-flops in Figure 10, are shared between the two machines. In an implementation of this concept, the logic, inputs, and outputs of these machines may also be either partially or fully shared. The test machine performs only two functions:

- It can be set to all its states by short predetermined sequences.
- Its state can be observed at the output by applying a short input sequence.

As the next subsection shows, these functions can easily be specified through a state diagram.

The first step in synthesis is merging the state diagrams of M1 and M2 to minimize the number of transitions added to M1. If M1 is fully or almost fully specified, such merging may not be possible and a test input can be added to the machine. Adding one input doubles the number of possible edges. The combined state diagram is then used for state assignment, logic

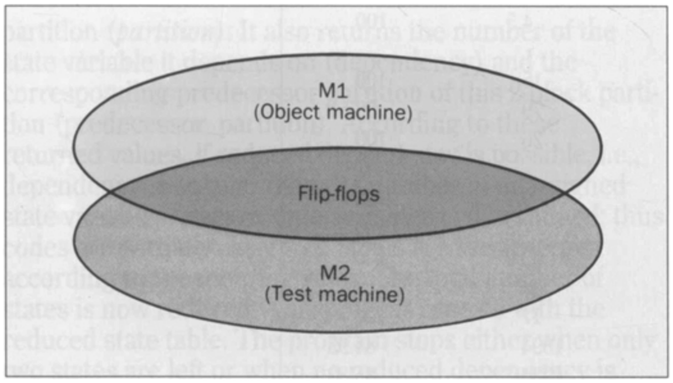


Figure 10. A testable design for finite state machines.

minimization, etc., to implement the function.

The implemented FSM can now perform both object and test functions. For testing, the test function is first executed to verify that all state variables (flip-flops) can be set and observed. The execution of this function will detect most faults in the flip-flops and some faults in the combinational logic that are common to both machines. For the remaining faults, a combinational circuit test generator determines the required inputs and states. Also, a fault may be detected either at a primary output or a state variable. These combinational tests are applied to the machine in much the same way as in a scan circuit. In place of scan-in and scan-out, M2's predetermined sequences are used.

Note that scan is a special case of our architecture where M2 is a shift register and the test mode input is the extra input added for merging the two state diagrams. Such presynthesis implementation of the test logic will allow using the functional logic for the test function where possible. The use of functional logic to provide the scan function has been shown by Reddy and Dandapani.[35] As a rule, for a given object function, a shift register may not be the best choice.

Designing a Test Machine. Though we have discussed the test function only in the classical sense of testability (distinguishing sequence, checking experiments, etc.), the concept in Figure 10 offers several possibilities. The test machine function can be specified to suit any practical test methodology. For example, M2 may provide a built-in self-test[36] or a self-checking[37] capability to M1. Thus, the following discussion should only be considered as one application of the concept.

Suppose the object machine has n states. Then the test machine must use $m = \lceil \log_2 n \rceil$ state variables. Thus, we specify the function of the test machine with $N = 2^m$ states, denoted as $T1, T2, \cdots, TN$. We require that, starting in any arbitrary state, the machine be brought to any given state by an input sequence of length $\log_k N$. Also, such an input sequence should produce an output sequence that depends only on the initial state. It may be necessary to select the design parameter k such that $\log_k N$ is an integer. We select k distinctly different inputs, $p1, p2, \cdots, pk$, and divide the N states of the test machine into k equal groups, $G1, G2, \cdots, Gk$. The state transitions are specified as follows:

- Each state has k outgoing edges labeled $p1$, $p2$, etc., such that the edge labeled pi terminates on a state belonging to the group Gi. Between any pair of states, only one edge is permitted in either direction. Thus, there can be at most two edges provided they are in opposite directions. Self-loops are permitted in the state diagram.
- Each state has exactly k incident edges. Clearly, all edges incident on state Tj will be labeled pi if Tj belongs to group Gi.

These conditions, while necessary, may not be enough.[38] The following procedure, however, will always result in a complete specification of the test machine.

Specifying State Transitions of Test Machines. We will illustrate the procedure with an example of 16 states, $T1$ through $T16$. Four inputs ($k = 4$), $p1$, $p2$, $p3$, and $p4$, are used. As Figure 11 shows, states are arranged in two rows; each row contains all states. The top row shows the *present* state and the bottom row contains the *next* states. A transition is represented by a directed edge

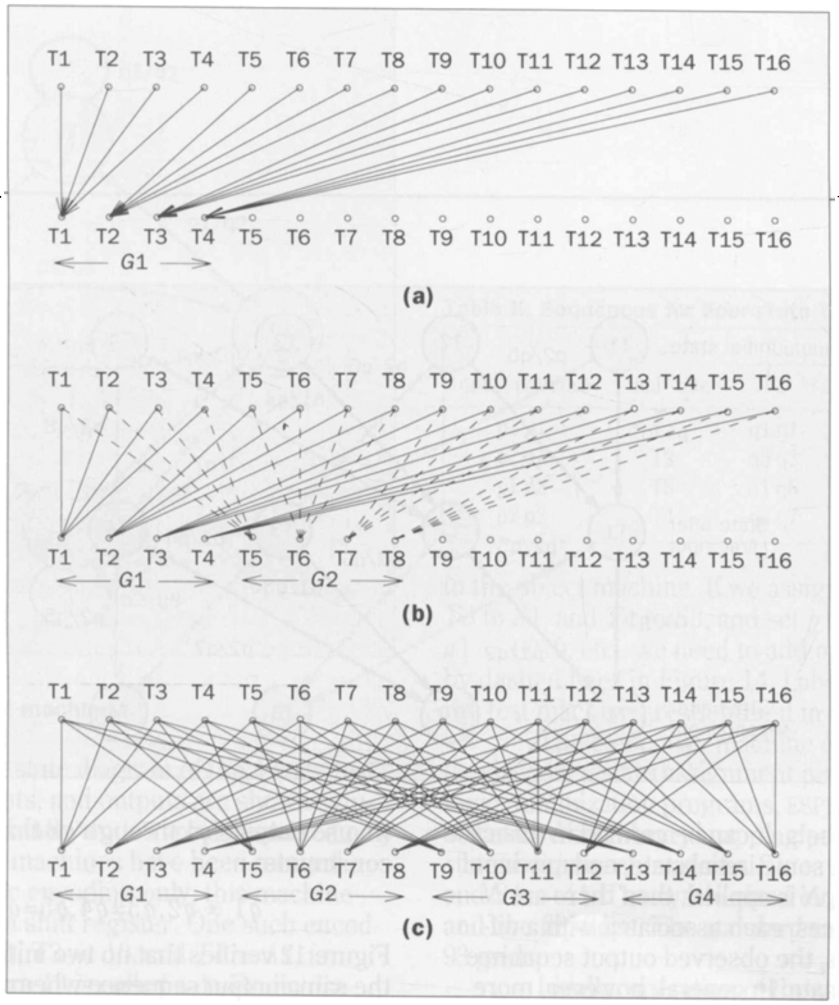


Figure 11. State transitions for test machine, $N = 16$, $k = 4$. 11a shows state transfers for input $p1$; 11b shows state transfers for inputs $p1$ and $p2$ (dashed); 11c shows all state transfers.

from a state in the top row to some state in the bottom row. The edge label (not shown in Figure 11 for clarity) shows the input and output. This diagram contains the same information as a conventional state diagram. In our case, however, any number of transitions can be represented by stacking up copies of the diagram.

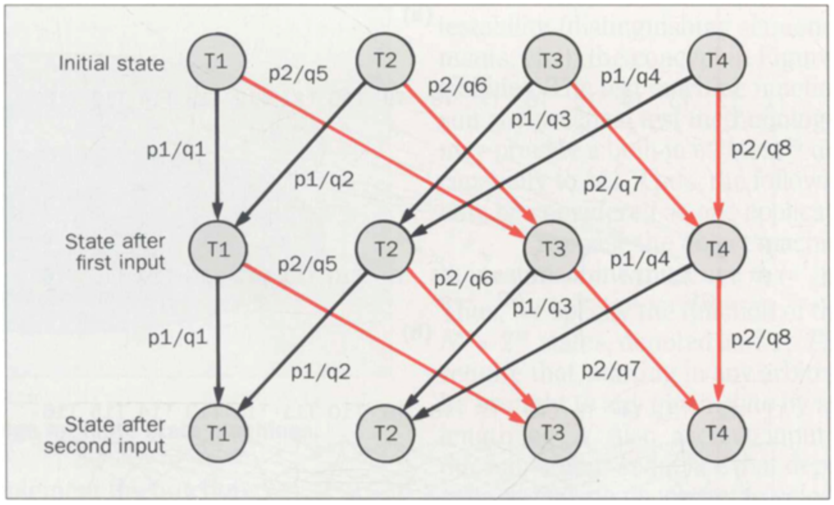
Figure 11a shows all edges with label $p1$. We draw edges from the first k states in the top row to the first state in the bottom row. Next, the states numbered from $k+1$ to $2k$ are connected to the second state in the bottom row. By the time this process is carried out for all states in the top row, all transitions to the first k states of the bottom row have been specified. These k states form the group $G1$ that is the target group for the input $p1$.

A similar procedure is carried out for the transitions of $p2$ and other inputs. The target states for $p2$ are the states numbered $k+1$ to $2k$ in the bottom row. These form the group $G2$. In Figure 11b, $p2$ transitions are shown by dashed lines to distinguish them from the $p1$ transitions. By the time this procedure is completed for all inputs up to pk, all target groups have been covered and the state diagram is complete. The result for the 16-state example with $k = 4$ is shown in Figure 11c. Arrows and edge labels in this diagram have been suppressed for clarity.

With the state diagram specified above, an input sequence of length $\log_k N$ will take the machine to a specific state. For k inputs used in the state diagram, there are N such sequences, each carrying the machine to one distinct state.

We can now specify the outputs for the test machine. Each of the k edges incident on a state must

Figure 12. Test machine sequences for a 4-state machine.



have different output labels. It can be verified in this case that if the machine is in some initial state, and any input sequence of length $\log_k N$ is applied, then there are N possible output sequences, each associated with a different initial state. Thus, the observed output sequence will identify the initial state. In general, however, more than k output patterns can be used as long as they satisfy certain constraints. We give an example below.

Notice that the value of k can be anywhere between 2 and N. However, the maximum value is bounded by the total number of possible input vectors. A larger value of k will require shorter sequences for setting and observing states. On the other hand, it will require a larger number of edges to be specified in the state diagram of the machine. With the addition of these edges to the object machine, the resulting design may need more hardware. Thus, a smaller value of k should be preferred. In the following example, we use $k = 2$.

Figure 12 shows an example of a four-state test machine. Because we have chosen $k = 2$, a series of two transitions will illustrate the complete operation. States $T1$ and $T2$ form the group $G1$. $T3$ and $T4$ belong to group $G2$. The two input patterns are $p1$ and $p2$. Suppose we use outputs $q1$ through $q8$ that satisfy the following constraints:

$$q1 \neq q2, q3 \neq q4, q5 \neq q6, \text{and } q7 \neq q8 \qquad (*)$$

Figure 12 verifies that no two initial states will produce the same output sequence when a sequence of two input patterns is applied. For example, if we apply the sequence $p2p2$ to reach the state $T4$, an initial state $T1$ will cause $q5q7$ output, while the initial state $T2$ will cause the output to be $q6q7$. Since $q5 \neq q6$, these two output sequences will be different. All four inequalities in (*) can be simultaneously satisfied by choosing $q1 = q3 = q5 = q7, q2 = q4 = q6 = q8$ and $q1 \neq q2$. However, the greater flexibility allowed by eight patterns is desirable when the test machine is superimposed on the object machine.

The operation of this machine is displayed in Table X. The state $T1$ has the synchronizing sequence $p1p1$. An application of this input sequence sets the machine in the state $T1$, regardless of the initial state. And because the output response identifies the initial state, all four synchronizing sequences are also distinguishing sequences. This is because of the output constraints (*).

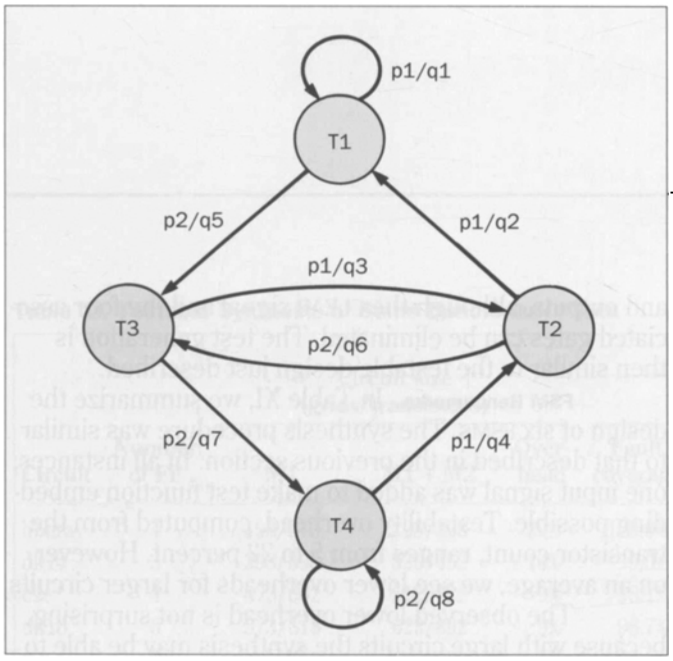


Figure 13. Four-state test machine.

Figure 13 is the state diagram of the 4-state test machine. All states, inputs, and outputs are shown symbolically. Their specific binary codes are determined only after the test and object machines have been merged. Interestingly, for specific encoding only, this machine can be implemented as a shift register. One such encoding is $T1 = 00$, $T2 = 01$, $T3 = 10$, and $T4 = 11$, for which the graph of Figure 13 is called a de Bruijn diagram[39] or Good diagram.[38] For other encodings, it may produce different structures. In all examples here, the state diagrams of the test and object machines were manually merged.

Graph embedding is a complex problem whose solution requires heuristics. For a good implementation, we seek a solution that adds the smallest number of edges to the object machine. If the two machines must have conflicting labels on the same edge, an input line is added. Finding good algorithms is a problem for further research.

A Simple Example. To illustrate our method, we consider a four-state machine. The function of our object machine is shown in Figure 14 by solid edges. The states are $S1$, $S2$, $S3$, and $S4$. We merge the 4-state test machine of Figure 13 with the object machine, using a merging criterion of adding a minimum number of edges

Table X. Sequences for Four-state Test Machine

		Output sequence for initial state			
Input sequence	Final state	T1	T2	T3	T4
p1 p1	T1	q1 q1	q2 q1	q3 q2	q4 q2
p2 p1	T2	q5 q3	q6 q3	q7 q4	q8 q4
p1 p2	T3	q1 q5	q2 q5	q3 q6	q4 q6
p2 p2	T4	q5 q7	q6 q7	q7 q8	q8 q8

to the object machine. If we assign $T1$ to $S2$, $T2$ to $S4$, $T3$ to $S1$, and $T4$ to $S3$, and set $p1 = 00$, $p2 = 01$, $q1 = 0010$, etc., we need to add only two edges as shown by dashed lines in Figure 14. Labels corresponding to the test machine are identified in bold-face.

The composite machine of Figure 14 was synthesized by a state assignment program, MUSTANG,[19] two logic minimization programs, ESPRESSO[40] and MIS,[20] and a standard cell library mapping program, DAGON.[21] The state assignment used was $S1 = 00$, $S2 = 10$, $S3 = 01$, and $S4 = 11$. The synthesized circuit contains 29 gates and two flip-flops. The standard cell design was sized at 93 grids.

Ideally, operation of the test machine should be verified by its complete checking experiment. For machines of any appreciable size, such an experiment will be too long to be practical. Therefore, we use a simple heuristic to test the test function by running a write and read test for the 0 and 1 states of each memory element. Indeed, such a test will not completely verify the test function, and coverage of the test sequence must be evaluated by a fault simulator. This heuristic is similar to that used for testing the scan register.[1]

For the present example, an input sequence, 00, 01, 01, 00, 01, 01, will set each flip-flop to 0 and 1, and observe their states at the primary outputs. The first two patterns set the machine to the 00 state, setting both flip-flops to 0. The next two patterns observe the state of flip-flops while also setting them to the 11 state. The last two patterns observe the 11 state. The gate-level coverage of single stuck faults for this sequence was 55 percent. The

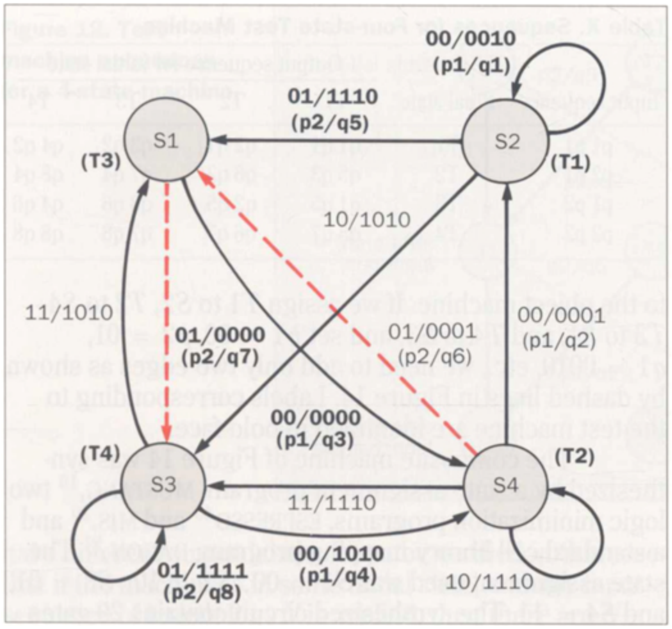


Figure 14. Object machine with embedded test machine.

combinational part of the circuit was isolated, and tests for the remaining faults were generated by a combinational circuit test generator. The test generator produced nine vectors specifying inputs $I1$, $I2$ and state variables $V1$, $V2$. Each test was preceded by two input vectors to set the required state and simultaneously distinguish the state of the previous test at the output. Thus, a set of 35 vectors detected all faults.

For comparison, the object machine was also synthesized without the dashed edges in Figure 14. This circuit contains 33 gates and two flip-flops. An additional CLEAR input and four gates for synchronous initialization were included. The standard cell design occupied 97 grids. Now, the tests must be generated either manually or by a sequential circuit test generator. Both methods require excessive effort. A conversion to scan design, however, takes at least 10 extra grids and extra inputs and outputs, although the CLEAR signal and the four associated gates can be eliminated. The test generation is then similar to the testable design just described.

FSM Benchmarks. In Table XI, we summarize the design of six FSMs. The synthesis procedure was similar to that described in the previous section. In all instances, one input signal was added to make test function embedding possible. Testability overhead, computed from the transistor count, ranges from 5 to 22 percent. However, on an average, we see lower overheads for larger circuits.

The observed lower overhead is not surprising, because with large circuits the synthesis may be able to reuse parts of the object function logic for the test function. In both cases (i.e., for M1 implemented alone, and for M1+M2) the tests were generated on a SUN4/260 workstation using an automatic test generator.[6]

Tests for M1+M2 consist of two parts. In the first part, to test the operation of M2, each flip-flop is successively set to 1 and 0, and its states are observed. These vectors are directly obtained from the test machine transitions in state diagram. The vectors are then run through a fault simulation of the entire circuit. For example, the 12 vectors for testing flip-flops of *bbara* covered 24.2 percent faults in the entire circuit. For the remaining faults, combinational tests were generated in 1.4 seconds on a SUN4/260 workstation. When these tests were converted into sequences and the 12 M2 vectors were added, the total vector count for this circuit became 156. Most circuits, in M1+M2 implementation, required much shorter time for test generation, as is typical of combinational circuits.

Another trend is clear from Table XI, i.e., that the achieved fault coverage for M1 starts dropping as the circuits become larger. This is typical of sequential circuits. Similar to scan design, M1+M2 require more test vectors.

Sometimes, as with the *sand* circuit, the M1+M2 design may turn out to be a scan register. The particular test function we have used can be implemented as one or more scan registers, and it should be synthesized that way if it is most economical. Therefore, we may not require more hardware than the scan design. In one

Table XI. Testable Synthesis of Some Benchmark FSMs

		Circuit size (grids/transistors)			Tests for M1+M2							
					Tests for M1			M2 tests		M1+M2 tests		
Circuit	Number of FF	M1	M1 + M2	Over-head	Fault coverage	CPU seconds	Test length	Test length	Fault coverage	Fault coverage	CPU seconds	Test length
bbara	4	193/270	225/328	22%	100.0%	46	133	12	24.2%	100%	1.4	156
dk14	3	293/398	329/452	14%	99.6%	13	64	9	31.5%	100%	2.0	132
cse	4	473/662	512/722	9%	98.1%	672	270	12	22.2%	100%	4.7	316
dk16	5	573/818	622/892	9%	98.7%	4570	276	15	25.4%	100%	9.9	445
tap	5	895/1290	937/1356	5%	98.1%	1652	329	15	29.1%	100%	13.2	510
sand	5	1062/1506	1195/1636	9%	96.4%	1848	316	15	10.2%	100%	23.3	735

example (i.e., *bbara* circuit), however, we found the overhead (22 percent) to be higher than scan. This could be because of the specific heuristics used in state assignment and in logic minimization. Other heuristics are worth investigating.

Conclusion

We have presented three methods of designing sequential circuits that consider testability before, during, and after logic synthesis. Our synthesis and post-synthesis approaches assume using a sequential circuit test generator. We have identified the cycles of flip-flops as the source of test generation difficulty. Our methods produce implementations either without cycles or with minimized cycles. In general, the state table specification of a typical FSM may contain unspecified ("don't care") transitions and inputs. Such incompleteness provides flexibility commonly used for optimization in logic synthesis. Our pre-synthesis approach of designing testable machines makes use of this flexibility.

References

1. V. D. Agrawal, S. K. Jain, and D. M. Singer, "Automation in Design for Testability," *Proceedings of the Custom Integrated Circuits Conference*, Rochester, New York, May 1984, pp. 159-163.
2. K.-T. Cheng and V. D. Agrawal, "A Partial Scan Method for Sequential Circuits with Feedback," *IEEE Transactions on Computers*, Vol. 39-4, April 1990, pp. 544-548.
3. K.-T. Cheng and V. D. Agrawal, "State Assignment for Testable Design," *International Journal of Computer Aided VLSI Design*, Vol. 3, March 1991.
4. V. D. Agrawal and K.-T. Cheng, "Finite State Machine Synthesis with Embedded Test Function," *Journal of Electronic Testing: Theory and Applications*, Vol. 1, No. 3, October 1990, pp. 221-228.
5. A. Miczo, *Digital Logic Testing and Simulation*, Harper & Row, New York, 1986.
6. W. T. Cheng, "The BACK Algorithm for Sequential Test Generation," *Proceedings of the International Conference on Computer Design (ICCD-88)*, October 1988, pp. 66-69.
7. A. D. Friedman and P. R. Menon, *Theory & Design of Switching Circuits*, Computer Science Press, Rockville, Maryland, 1975.
8. R. V. Hudli and S. C. Seth, "Testability Analysis of Synchronous Sequential Circuits Based on Structure Data," *Proceedings of the International Test Conference*, Washington, D.C., August 1989, pp. 364-372.
9. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.
10. F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of the International Symposium on Circuits and Systems*, May 1989, pp. 1929-1934.
11. K. T. Cheng and V. D. Agrawal, "Concurrent Test Generation and Design for Testability," *Proceedings of the International Symposium on Circuits and Systems*, May 1989, pp. 1935-1938.
12. S. Bhawmik, C.-J. Lin, K.-T. Cheng, and V. D. Agrawal, "PASCANT: A Partial Scan and Test Generation System," *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Diego, California,

May 12-15, 1991.
13. J.-Y. Jou et al., "BESTMAP: Behavioral Synthesis From C," *Proceedings of the International Workshop on Logic Synthesis*, May 1989, unpaged.
14. V. D. Agrawal, "Synchronous Path Analysis in MOS Circuit Simulator," *Proceedings of the 19th Design Automation Conference*, June 1982, pp. 629-635.
15. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Journal of Design Automation and Fault. Tol. Comp.*, Vol 2, May 1978, pp. 165-178.
16. M. R. Mercer and V. D. Agrawal, "A Novel Clocking Technique for VLSI Circuit Testability," *IEEE Journal of Solid-State Circuits*, Vol. SC-19, April 1984, pp. 207-212.
17. A. Gill, *Introduction to the Theory of Finite-State Machines*, Chapter 6, McGraw-Hill, New York, 1962.
18. R. Lisanke, "Finite-State Machine Benchmark Set," Preliminary benchmark collection, September 1987.
19. S. Devadas et al., "MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-7, December 1988, pp. 1290-1300.
20. R. K. Brayton et al., "MIS: A Multiple Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, November 1987, pp. 1062-1081.
21. K. Keutzer, "DAGON: Technology Binding and Local Optimation by DAG Matching," *Proceedings of the 24th Design Automation Conference*, June 1987, pp. 341-347.
22. D. B. Armstrong, "On the Efficient Assignment of Internal Codes to Sequential Machines," *IRE Transactions on Electronic Computers*, October 1962, pp. 611-622.
23. J. Hartmanis, "On the State Assignment Problem for Sequential Machines, I," *IRE Transactions on Electronic Computers*, Vol. EC-10, June 1961, pp. 157-165.
24. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1966.
25. Z. Kohavi, *Switching and Finite Automata Theory*, Second Edition, McGraw-Hill, New York, 1978.
26. H. A. Curtis, "Multiple Reduction of Variable Dependency of Sequential Machines," *Journal of the ACM*, Vol. 9, July 1962, pp. 324-344.
27. R. M. Karp, "Some Techniques of State Assignment for Synchronous Sequential Machines," *IEEE Transactions on Electronic Computers*, Vol. EC-13, October 1964, pp. 507-518.
28. N. N. Biswas, *Introduction to Logic and Switching Theory*, Gordon and Breach Science Publishers Ltd., New York, 1975.
29. F. C. Hennie, "Fault Detecting Experiments for Sequential Circuits," *Proceedings of 5th Annual Symposium on Switching Theory and Logical Design*, Princeton University, November 1964, pp. 95-110.
30. Z. Kohavi and P. Lavallee, "Design of Sequential Machines with Fault-Detection Capabilities," *IEEE Transactions on Electronic Computers*, Vol. EC-16, August 1967, pp. 473-484.
31. H. Fujiwara, Y. Nagao, T. Sasao, and K. Kinoshita," Easily Testable Sequential Machines with Extra Inputs," *IEEE Transactions on Computers*, Vol. C-24, August 1975, pp. 821-826.
32. E. P. Hsieh, "Checking Experiments for Sequential Machines," *IEEE Transactions on Computers*, Vol. C-20, October 1971, pp. 1152-1166.
33. D. K. Pradhan, "Sequential Network Design Using Extra Inputs for Fault Detection," *IEEE Transactions on Computers*, Vol. C-32, March 1983, pp. 319-323.
34. A. Bhattacharyya, "On a Novel Approach of Fault Detection in an Easily Testable Sequential Machine with Extra Inputs and Extra Outputs," *IEEE Transactions on Computers*, Vol. C-32, March 1983, pp. 323-325.
35. S. M. Reddy and R. Dandapani, "Scan Design Using Standard Flip-Flops," *IEEE Design and Test of Computers*, Vol. 4, February 1987, pp. 52-54.
36. B. Eschermann and H.-J. Wunderlich, "Optimized Synthesis of Self-Testable Finite State Machines," *Fault-Tolerant Computing Symposium (FTCS-20) Digest of Papers*, Newcastle-upon-Tyne, UK, June 1990, pp. 390-397.
37. R. Leveugle and G. Saucier, "Optimized Synthesis of Concurrently Checked Controllers," *IEEE Transactions on Computers*, Vol. 39, April 1990, pp. 419-425.
38. A. J. Nichols, "Minimal Shift-Register Realizations of Sequential Machines," *IEEE Transactions on Electronic Computers*, Vol. EC-14, October 1965, pp. 688-700.
39. S. W. Golomb, *Shift Register Sequences*, Aegean Park Press, Laguna Hills, California, 1982.
40. R. K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, Massachusetts, 1984.

Biographies (continued)

Allahabad, India; a B.E. in telecommunication engineering from the University of Roorkee, Roorkee, India; an M.E. in electrical communication engineering from the Indian Institute of Science, Bangalore, India; and a Ph.D. in electrical engineering from the University of Illinois, Urbana-Champaign.

(Manuscript received November 5, 1990)