VLSI DESIGN
2001, Vol. 12, No. 4, pp. 487–500
Reprints available directly from the publisher
Photocopying permitted by license only

BIST-Based Fault Diagnosis in the Presence of Embedded Memories*

JACOB SAVIR

ECE Dept., New Jersey Institute of Technology, University Heights, Newark, New Jersey 07102-1982

(Received 15 August 1999; In final form 11 September 2000)

An efficient method is described for using fault simulation as a solution to the diagnostic problem created by the presence of embedded memories in BIST designs. The simulation is event-table-driven. Special techniques are described to cope with the faults in the Prelogic, Postlogic, and the logic embedding the memory control or address inputs. It is presumed that the memory itself has been previously tested, using automatic test pattern generation (ATPG) techniques via the correspondence inputs, and has been found to be fault-free.

Keywords: Fault detection; Fault simulation; Fault diagnosis; Built-in Self-Test (BIST); Multiple-Input Shift Register (MISR); Signature

1. INTRODUCTION

Built-in self-test (BIST) has seen increased use by major computer and chip manufacturing companies over the last decade. Even though BIST methodology as a test vehicle has reached its maturity, more is required in advancing and defining an efficient diagnostic process. Without such a process, a BIST-based diagnostic becomes a complete nightmare.

Since BIST-based diagnostic is a very complicated, costly, and time consuming task, it is necessary to have several diagnostic aids to guide the process. First degree of complication stems from the fact that BIST uses data compression, and the only thing available at the end of the test is a short signature. The second degree of complication (and not less serious one) is due to the presence of embedded memories. Errors (due to some faults) penetrate the memories during BIST only to show up at a later time as faulty signature. This error latency is one of the primary causes as to why the diagnostic success is quite poor. Still, a third degree of complication is due to a lack of "design for diagnosability" in BIST structures. Most BIST designs concentrate on efficient fault detection, but lack the necessary "hooks" to ease the diagnostic task.[1] As this paper will show,

* This work was supported by NJCST for the Center on Embedded System on a Chip Design.
[1] It is assumed here that no boundary scan exists around the memories because of strict performance requirements.

proper distribution of the MISRs in the design can highly increase the diagnostic resolution, as well as decrease the amount of time spent on isolating the faults.

Diagnostic aids constitute a "bag of tricks". The task of these diagnostic aids is to provide "short cuts" in pinning down the failing pattern (out of several million). Once the failing pattern is determined, deterministic diagnostic practices can be used to isolate the fault (or fault equivalence class). In [7, 12] two such diagnostic aids have been described to identify the failing test in combinational logic. These methods may be used quite successfully to diagnose failures in the combinational logic between shift register latches, provided none of the memories have been corrupted.

Other prior art is relevant to this paper. In [2] general discussion of BIST diagnostic is provided. In [9] an efficient BIST scheme for naked memories is described for which fault diagnosis comes for free. Still another BIST scheme with self diagnostic capabilities are described in [8]. A memory BIST that is based on error detecting codes is described in [6]. BIST and diagnosis of analog circuits is described in [5]. BIST for static RAMs is described in [13]. In [3] a description is given for BIST for embedded RAMs.

The diagnostic objectives of this paper are several fold: isolate failures during chip bring-up (1st silicon); deal with design and process changes; rectify fabrication problems, *etc.*

The test environment can simply be described as follows. A logic structure which conforms to the level sensitive scan design (LSSD) [4] ground rules, and which contains embedded random access memories (RAMs) is being tested with pseudo-random patterns. The responses of the structure are being compressed in a single-input or multiple-input signature register (SISR or MISR respectively). A faulty signature is observed after test completion. It is necessary to identify the fault location in order to permit a repair action. It is assumed that the memory itself has been previously tested using known memory test techniques and found to be fault-free. It is further assumed that a single fault exists in the faulty circuit that undergoes diagnostics.

This paper first describes the simulation-based diagnostic methodology as applied to a simple, single-port, Read/Write memory. This simple case is then generalized to BIST designs having multiple single-port, Read/Write memories. The diagnostic bottlenecks that exist in this, more general case, are then identified. It is shown that there is a need to distribute the MISRs throughout the design, rather than have a single MISR collect a single signature, in order to ease the diagnostic task. The distributed MISRs (ideally one for each embedded memory) add a degree of observability that is very useful during diagnosis (this does not necessary add any more power to the fault detection capability, though). MISRs distribution should be considered as one dimension in the design for diagnosability. Obviously, there is an added cost to this design.

Section 2 describes the main diagnostic procedure. Sections 3–6 describe the core properties of diagnosing failures in BIST circuits having a single memory. More specifically, Section 3 describes the memory and the techniques for fault identification in the memory Prelogic (the logic feeding the data-in lines) and Postlogic (the logic fed by the data-out lines). In Section 4 we describe a mechanism for locating the fault in the address or control logic, and in Sections 5 and 6 we show special techniques for isolating address and control logic faults. Section 7 discusses fault diagnosis in circuits having multiple embedded memories. Section 8 outlines the enhanced diagnostic procedure. Section 9 shows some experimental results. Finally, Section 10 concludes with a listing of the special control functions that our methodology requires from the memory and embedding logic design, and summarizes the important findings of this paper.

In the sequel it is assumed that the reader is familiar with deterministic diagnostic practices and basic BIST diagnostic methodology for LSSD-based designs [1, 2].

2. MAIN DIAGNOSTIC PROCEDURE

As a framework for discussion we first introduce the main diagnostic routine as practiced with BIST-based products which also conform with the LSSD design rules.

Figure 1 shows the circuit (chip, card, *etc.*) view in BIST mode. The BIST hardware is shown separate from the functional circuit. The BIST hardware includes two linear feedback shift registers (LFSRs) driving random inputs into the primary inputs (PIs) and shift register input (SRI). The MISR and the SISR collect the test responses from the primary outputs (POs) and shift register output (SRO). It is inherent that the entire circuit conforms with the LSSD design rules. Figure 1 shows a single scan chain whose input is SRI and whose output is SRO. In case there are multiple scan chains, multiple SRIs and SROs will exist connecting to their separate LFSR and SISR. The A/B inputs are the scan chain shift clocks, and the MCs are the functional machine clocks.

A large number of tests are to be applied to the functional unit. For each test the controller must do the following:

(1) Load a random (actually, pseudo-random) test vector into the SR string of the functional circuit by simultaneously clocking the SRI LFSR and the A/B shift clocks. On each clock cycle, one new pseudo-random bit is generated by the LFSR and shifted into the SRI. Enough shift clock cycles are applied to fill the scan string completely.
(2) Apply one clock cycle to the PI LFSR to apply a new pseudo-random vector to the PIs.
(3) Cycle through the machine clocks (MCs) to capture the responses to the random stimuli in the scan string latches.
(4) Unload the responses on the POs to the MISR by applying one clock cycle to the MISR shift clocks.
(5) Unload the captured responses in the internal latches via the scan path to the SISR by simultaneously clocking the A/B clocks and the SISR shift clocks.

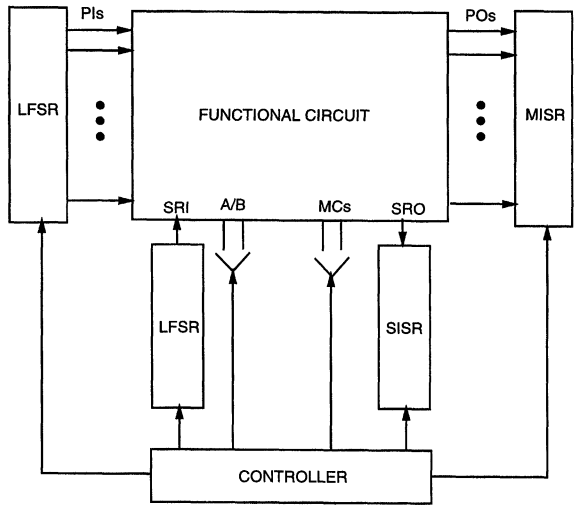

FIGURE 1 Circuit view in BIST mode.

Steps 1 and 5 of the process can be overlapped since it is possible to unload the SR string into the SISR at the same time as the next pseudo-random pattern is loaded into the SRI. A pass/fail indication is obtained after the last test by comparing the binary values remaining in the MISR and SISR with the expected values.

In order to ease the diagnostic process the entire test is divided into smaller *test windows* for which the expected signatures are precomputed. In normal BIST mode the entire test is run and signatures are compared at the end of the test. If the signatures at the end of the test are correct, the circuit is declared fault-free. If the signatures are incorrect the diagnostic process is invoked to suggest a repair action. To do this, the test is rerun stopping at the end of each test window to compare intermediate signatures against their precomputed values. The test continues till a test window is reached that captures a wrong signature. It is, then, most likely that the first failing test has occurred during this last test window. This is

true in most, but not all, cases. In some cases the diagnostic procedure may experience a latent fault, where wrong data is written to some memory location, but being read out at a much later time. Thus, a latent fault may be captured in one test window and show up as an error at a later one. We will say more about this case later in the paper.

The diagnostic process begins by restoring the state of the machine to what it was at the beginning of the failing test window. The test mode is then switched from BIST to LSSD/DIAGNOSIS mode. In this mode the MISR and the SISR act as pure shift registers, in order to be able to investigate each and every test pattern deterministically. The last test window is then repeated, pattern by pattern, till the failing pattern is encountered. Deterministic diagnostic practices are then invoked to isolate the fault, or fault equivalence class [1].

3. PRELOGIC AND POSTLOGIC FAULTS IN A SIMPLE R/W MEMORY

To simplify the discussion we assume a single port Read/Write memory within combinational logic as shown in Figure 2. We will address circuits with multiple embedded memories later in the paper.

Figure 2 shows the circuit as seen at the time the diagnostic process begins. Notice that the BIST hardware is not shown. Inputs to the structure are the primary input groups marked D1, D2, D3, and D4. The only observation points are the Primary Outputs (POs). The memory contains 2^n words of n bits each (this is a special case in which the memory word is exactly as wide as the address space, and will later be generalized). The number of data-out lines is equal to the number of data-in lines. When the R/W Control Line is set to Write, the word on the memory data-in lines is written to the selected address in a write-through mode (so that the memory data-out lines take on the value of the word written). When the R/W Control Line is Set to Read, the word stored at the selected address appears at the memory data-out lines. All

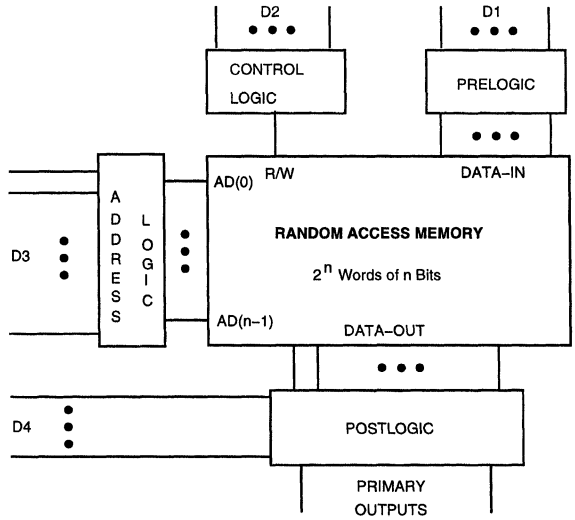

FIGURE 2 A simple R/W embedded memory.

primary input and primary output values are saved for each test in the sequence.

The first step in the method is to perform good-machine simulation on just the Control Logic and the Address Logic to determine both the memory address referenced and the type of operation, Read or Write, for each of the K tests in the test sequence. This information is used to construct an Event Table, a self-explanatory example of which is shown in Table I.

The next step is to use this event table to construct a Simulation Event Table which shows, for each test (now called an event), the test number

TABLE I Event Table for the memory of Figure 1

Test	Address	R/W
1	9	Read
2	61	Write
3	37	Write
4	43	Read
5	10	Write
6	37	Write
7	10	Read
8	43	Read
9	10	Read
.	.	.
.	.	.
.	.	.
K	38	Read

at which the word on the memory data-out lines was written. An example, derived from event Table I, is given in Table II. In Table II, the entry 0 in the column "Data Written at Test No". indicates that the data-out lines take the initialization word for the selected address (since this particular memory address has not been written since initialization).

The last step is to use the Simulation Event Table to move the primary input values which generated a word to the test at which that word was read out. By doing this, the memory is effectively removed from the structure and simulation can proceed on a tester-loop-independent basis. Figure 3 shows the simulation model to be used for Test 7 of Table II. The D1 inputs are those that existed at Test 5 since from the Simulation Event Table the word read at Test 7 had been written during Test 5. The D4 inputs are those that existed at Test 7. It is apparent that the PO values in Figure 3 are the values read out of the structure at Test 7.

There is a second simulation model that has to be used if the data-out word for a particular test is the initialization word. It is presumed that the words initially in each memory location are known. As a matter of fact, these are the contents of the memory words at the start of the test window. (In order to ease the diagnostic process the entire test is divided into smaller *test windows*

TABLE II Simulation Event Table constructed from Table I

Event	Address	Data written at Test No.
1	9	0
2	61	2
3	37	3
4	43	0
5	10	5
6	37	6
7	10	5
8	43	0
9	10	5
.	.	.
.	.	.
.	.	.
K	38	27

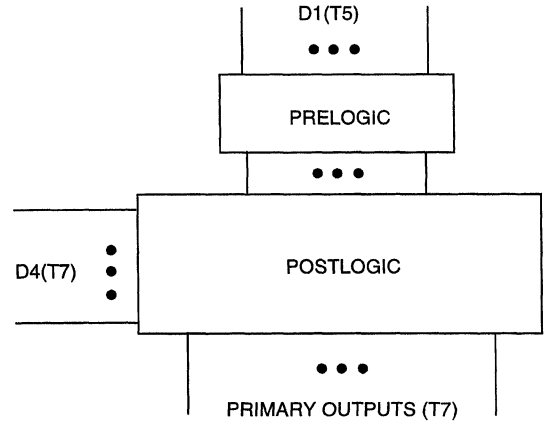

FIGURE 3 Tester-loop-independent simulation of Test 7.

for which the expected signatures are precomputed.) Figure 4 shows this alternative model using as an example Event 8 from Table II. Again it should be evident that the PO values from Figure 4 are the same as those captured at the actual Test 8.

The Simulation Event Table is used to construct simulation models of the type shown in either Figure 3 or Figure 4, and good machine simulation is used to compute the expected PO values for each test. These expected values are compared with the actual test values, received during test, to identify the failing tests.

The first diagnostic assumption is that a single fault exists in the structure and that the fault is in either the Prelogic or the Postlogic. Simulation models of the types shown in Figures 3 and 4 are fault simulated to identify a fault (or fault equivalence class) which satisfies the failing output

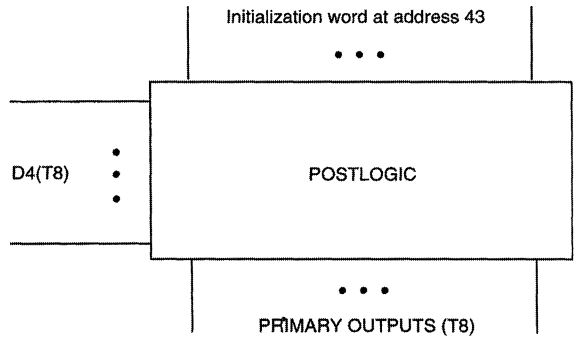

FIGURE 4 Tester-loop-independent simulation of Test 8.

values. If such a fault is found, either in the Prelogic or Postlogic, the job is done.[2] However, when multiple failing tests occur, it may turn out that no single fault can explain all failing tests. With incompatible fault equivalence classes in Prelogic and Postlogic, one could postulate multiple faults, but it seems more appropriate to maintain the assumption of a single fault and to suspect a fault in either the Control or the Address logic.

4. SEPARATING ADDRESS FROM CONTROL LOGIC FAULTS

A fault in the Control Logic will either cause an unintended write or inhibit an expected write. A fault in the Address Logic will access (either a read or a write operation) an incorrect address. To identify which of the two logics contains the fault it seems necessary to postulate an intelligent diagnostic program working from a list of heuristics, the Event Table, the Simulation Event Table, and the set of failing tests. The heuristics might include the following:

(1) Suppose all (or most of) the write operations fail. Since writing is in a "write-through" mode regardless of the addresses selected, weight is added to the hypothesis of a Control Logic fault.
(2) Suppose all (or most of) the reads of some address A fail, that writes to A pass, and that reads of at least some other addresses pass. Weight is added to the hypothesis of an Address Logic fault.
(3) Suppose a failing test is a write operation to some address A. There may be a subsequent read of A (without any intervening writes). If this read also fails, a Control Logic fault is suggested.
(4) Suppose a read of address A fails. If there is a preceding write to A that passes, and if there are earlier reads which also pass, an Address Logic fault is suggested.

It is important to note, of course, that the heuristics must be tailored to the type of memory being tested. However, the purpose of the intelligent program is to identify which logic *probably* contains the fault for the purpose of reducing subsequent computation, and avoiding unexplainable error symptoms.

5. ADDRESS LOGIC FAULTS IN A SIMPLE R/W MEMORY

Suppose now that the program suggests the fault lies in the Address Logic. We want to perform fault simulation on the address logic to identify the fault. We start by noticing that the memory word is exactly as wide as the address space. (We will describe later how to cope with memories in which the word is narrower or wider than the address space). Now by using the correspondence inputs[3] and the address cycling register,[4] we deterministically load each memory word with its address as illustrated in Table III. This capability is assured since the memory itself was previously tested good using the correspondence inputs.

The next step is to rerun part of the test sequence on the structure with the memory control logic (which is assumed to be fault-free) held to a constant read. To do this requires generating a deterministic vector for inputs D2 of Figure 2 which will hold the R/W line at a Read value. (Alternatively, since we determined which tests caused read operations in the process of generating Tab. I, the Event Table, we could use one of these read-invoking sets of D2 values).

[2] Because of the single-fault assumption.
[3] These are inputs that feed directly, or indirectly, the data-in lines, R/W control, and address lines, so that when a proper vector is set, it is possible to write into the memory arbitrary data.
[4] Normally added to BIST designs in order to ease cycling through the address space; implemented by a counter.

TABLE III Data written to addresses prior to address logic simulation

Address	Data written to the address
0	000...00
1	000...01
2	000...10
3	000...11
.	.
.	.
.	.
2^n-1	111...11

During this rerun of the tests, the Prelogic is out of the circuit, since we are holding a constant read. Because each word in the memory contains its address, whenever an address A is applied to the memory decoder inputs the address A appears on the memory data-out lines. Effectively the memory has been removed and a structure created which behaves exactly like the circuit shown in Figure 5. In Figure 5, test T7 is simulated. The D3 inputs to the Address Logic and the D4 inputs to the Postlogic are those from T7. Notice, however, that the PO values T*7 are not those that were recorded from the original Test 7 since the memory contents are not the same. We must do good-machine simulation of the structure of Figure 5 to obtain the expected T* output values for comparison with the values obtained from the test rerun. Given the expected values and the actual values from rerunning the tests, we can do fault simulation on the structure of Figure 5 to identify the fault (or equivalence class) within the Address Logic.

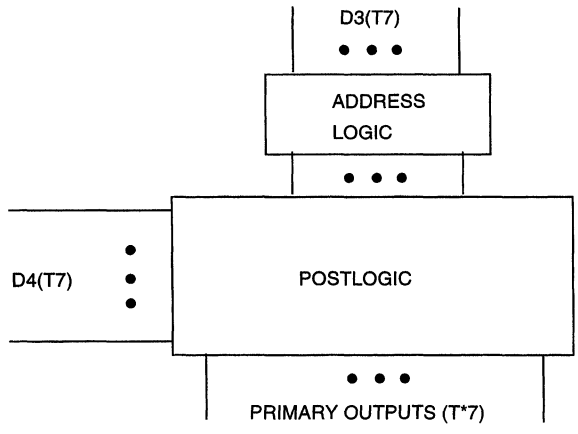

FIGURE 5 Structure for fault simulation of Address Logic.

The question now is how to identify the tests which must be repeated during simulation of the Address Logic. We earlier used good machine simulation on models of the types shown in Figures 3 and 4 to compute the expected PO values for comparison with the actual values to identify the failing tests. Let T_j be the first failing test. Now suppose that a prior test, say test T_i, was a write operation to some address A. During T_i the memory is in write-through mode, and since the Prelogic and Postlogic are fault-free the test passes, regardless of the Address Logic fault. If during T_i the fault affects the memory decoder inputs, then instead of writing address A, the fault causes a write to an incorrect address B. A subsequent read operation on address A (or address B) retrieves an incorrect word, and the read operation fails. Hence we find it impossible to identify the precise "failing tests" other than to say that there was at least one failing test prior to test T_j. It is, therefore, necessary to repeat all of the tests up to and including T_j. For each of these tests, the models of Figure 5 are good-machine-simulated to obtain PO values for comparison with the actual values obtained from the tests. This comparison will identify the actual "failing tests" which are then used in fault simulation (again with models like Fig. 5) to identify the fault.

If the memory word is narrower than the address space, we are forced into a multiple-pass test of the Address Logic. On each pass a portion of its address is stored in each memory word. For example, suppose the address space requires 8 bits but the memory is only 4 bits wide. On the first pass we load the low-order 4 bits of the address to its word, and on the second pass load the remaining address bits. The appropriate portion of the test sequence then is repeated twice, once for each pass, again holding a constant read. For each pass we record the PO values. We, then, combine the PO values for each test and compare these values against a simulation model of the type

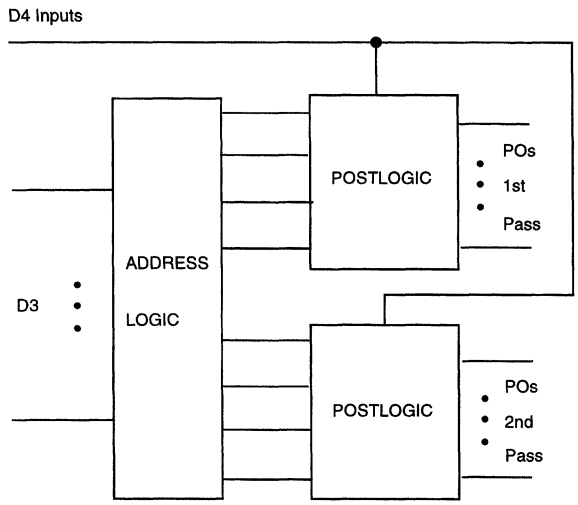

FIGURE 6 Address Logic simulation when word is narrow.

shown in Figure 6. By comparing the values on the combined POs during good machine simulation with the actual values recorded during the two test passes, we can identify the failing tests. These failing tests can in turn be fault simulated, again using models like Figure 6, to identify the fault. Note that the fault sites in the two postlogic replicas need not be simulated since it is known, at this point, that the Postlogic is fault-free.

If the memory word is wider than the address space, the surplus bits in each word may be loaded with 0s, for example.

6. CONTROL LOGIC FAULTS IN A SIMPLE R/W MEMORY

Suppose that the intelligent program suggests the fault lies in the Control Logic. We are quite sure that the Prelogic, the Postlogic, and the Address Logic are fault-free, and will proceed on the assumption that they are indeed fault-free.

Earlier we have identified all the failing tests. From the event table we know the address and the expected operation (read or write) for these failing tests. We first consider those failing tests for which the expected operation is a write, since it is the most straight forward case, and will later consider expected reads.

6.1. When an Expected Write Operation Fails

Consider a failing test for which the expected operation is a write to some arbitrary address A. Since the memory is write-through, the expected word on the memory data-out lines is the same as the word on the data-in lines. Furthermore, the Prelogic and the Postlogic are fault-free, so that the test should have passed if the operation was actually a write. Since the test failed, we conclude that the actual operation was a read of address A.

The foregoing argument applies to any failing test in which the expected operation is a write. We then have the following: if we are sure that the fault lies in the Control Logic, then all failing tests which are good-machine writes are actually faulty reads. This means that during such tests we are certain that the value on the R/W control line is inverted from write to read. We can then do fault simulation on the Control Logic for these tests to identify a fault equivalence class which explains the failing tests. The simulation structure is shown in Figure 7 for a failing test t which is an expected write. In Figure 7 the D2 inputs to the Control Logic are those that existed at test t and the output R/W(t) is incorrectly inverted from Write to Read.

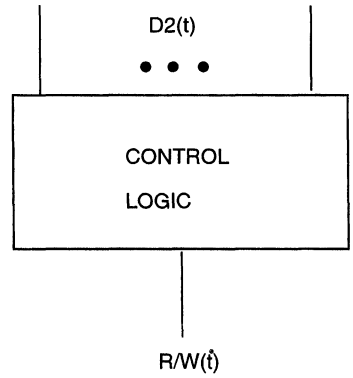

FIGURE 7 Structure for Control Logic fault simulation.

6.2. When an Expected Read Operation Fails

Consider those cases where the expected operation during a failing test is a read of some address A. In these cases we have an uncertainty between two possibilities:

(1) The read actually occurred correctly but the test failed because the word stored at the address was incorrect, either because some earlier expected read was changed to a faulty write, or an earlier write was changed to a faulty read.
(2) The read operation during this test was changed by the fault to a write.

The first step in sorting out these possibilities is to search the event table for all tests which accessed address A prior to the failing test t. We use these tests to set up an activity table, an example of which is shown in Table IV.

The test numbers in Table IV are listed in decreasing value and are those tests prior to and including failing test t which accessed address A. The Expected Operation column entries are from the event table that was constructed earlier.

The Expected PO Values are obtained by a tester-loop-independent good machine simulation of structures like that shown in Figure 8. In these simulations, the D1 inputs are those from the most recent write operation to address A. These are exactly the values which were listed in the simulation event table constructed in Section 3 to obtain the PO values for each test. However,

TABLE IV Prior accesses to A and their actual PO values with D4(t)

Test #	Expected operation	Expected PO values	Actual PO values	Mock-Write PO values
t	Read	PO(Y)	PO(X)	PO(T)
$t-t_1$	Read	PO(Y)	PO(X)	PO(U)
$t-t_2$	Write	PO(Y)	PO(X)	PO(Y)
$t-t_3$	Read	PO(X)	PO(X)	PO(V)
$t-t_4$	Write	PO(X)	PO(X)	PO(X)
$t-t_5$	Read	PO(Z)	PO(Z)	PO(W)
$t-t_6$	Write	PO(Z)	PO(Z)	PO(Z)

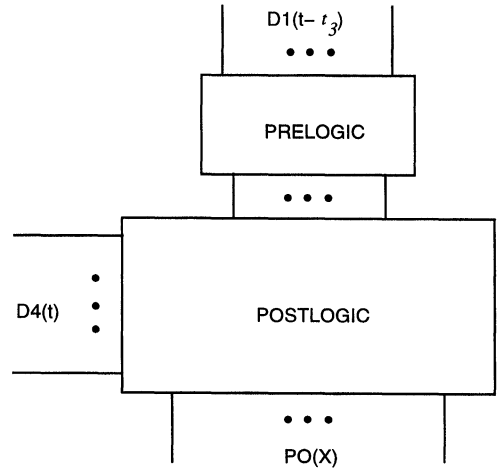

FIGURE 8 Expected PO values obtained by holding D4 at D4(t).

during each simulation, *the* D4 *inputs to the Postlogic are held at their values during the failing test t*. Each of the prior accesses is simulated in this fashion to complete the Expected PO Values entries in Table IV. (Note that these Expected PO Values are *not* those obtained earlier in Section 3).

The column marked Actual PO Values is obtained from a rerun of the listed tests on the actual structure when *during each test the* D4 *inputs to the Postlogic are held at their values during the failing test t*. All other structure inputs (D1, D2, and D3) are driven with their original values during each test. We record the Primary Output values for each of these test reruns and enter these in the Actual PO Values column.

The column marked Mock-Write PO Values is obtained by doing a mock simulation of a write operation. Let test T9 be a test which is a read operation in the good-machine. Now do a good-machine simulation of the Prelogic as shown in Figure 9 to obtain the PO values T*9. Because the memory is write-through, this simulation reflects a presumed write operation at T9. It is these T* values for each of the listed reads which are entered into the Mock-Write PO Values column.

What is the activity table? At test t, some incorrect word X appeared on the memory

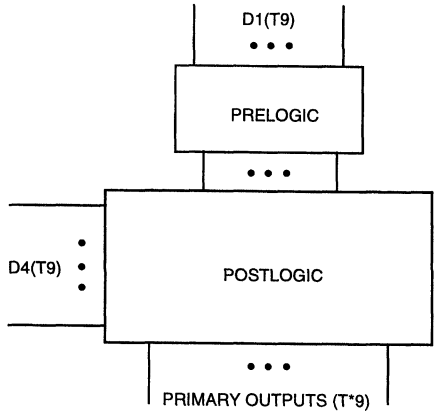

FIGURE 9 Mock good-machine simulation of a Write operation at Test 9.

data-out lines instead of the correct word Y. This incorrect word X caused a set of incorrect PO values PO(X) instead of the correct set PO(Y). We want to know how this incorrect word X got into the memory. Since the Postlogic primary inputs D4(t) at test t exposed the incorrect word, we use these primary inputs to expose the test at which X was loaded into the memory. This should become clearer as we describe how to use the table.

We compare, starting up from the bottom of the activity table, the expected, the actual, and the mock-write values. An inconsistency between expected and actual values, if supported by the mock-write values, indicates a test on which the R/W control line failed.

As an example, in Table IV it is obvious that the Write operation at test $t-t_2$ the R/W control line was inverted from Write to Read. Now we can use fault simulation on the Control Logic as shown in Figure 7, using the appropriate D2 values, to identify the actual fault.

The Control Logic fault may cause multiple R/W line failures before a failing test occurs. Consider, as an example, the sequence shown in Table V where test t is a failing test, and there are 3 prior accesses to the address. Beginning at the bottom of the table (the earliest access) and working up:

(1) Test $t-\gamma$ wrote word Z correctly.
(2) Test $t-\beta$ failed. The PO values should have been PO(Z), but the actual values were PO(X) which matches the Mock-Write values for this test. Hence, the fault caused the R/W line to invert from Read to Write.
(3) Test $t-\alpha$ also failed. It should have written Y, but instead the fault inverted the R/W line to a Read.

As a final example, Table VI shows 3 R/W line failures at test t, $t-\alpha$, and $t-\beta$.

TABLE V A multiple-failure activity table

Test #	Expected operation	Expected PO values	Actual PO values	Mock-Write PO values
t	Read	PO(Y)	PO(X)	PO(W)
$t-\alpha$	Write	PO(Y)	PO(X)	PO(Y)
$t-\beta$	Read	PO(Z)	PO(X)	PO(X)
$t-\gamma$	Write	PO(Z)	PO(Z)	PO(Z)

TABLE VI R/W line failures at Test t, $t-\alpha$, and $t-\beta$

Test #	Expected operation	Expected PO values	Actual PO values	Mock-Write PO values
t	Read	PO(Y)	PO(X)	PO(X)
$t-\alpha$	Write	PO(Y)	PO(Z)	PO(Y)
$t-\beta$	Read	PO(X)	PO(Z)	PO(Z)
$t-\gamma$	Write	PO(X)	PO(X)	PO(X)

7. DIAGNOSTIC OF CIRCUITS WITH MULTIPLE MEMORIES

When the circuit has multiple memories fault isolation is even more complex. To see this, consider a case of a circuit having two embedded memories M1 and M2. Assume further that only one MISR exists to collect test responses at the circuit POs. Let the Simulation Event table be constructed as was previously suggested. Consider the case where the table indicates that many reads to some address A in memory M1 fail, and many writes in memory M2 fail. Is this due to a likely fault in the Address Logic of memory M1, or a fault in the Control Logic of memory M2? The

answer is that it could be either. Thus, the "intelligent program" is unable to guide the diagnostic process in a manner similar to what used to be in a case where only a single memory exists. The diagnosis, then, has to proceed by carefully examining both cases. This increases the diagnostic time tremendously.

What can be done to ease this case? Unfortunately, not much can be done unless special attention is given to the BIST circuitry at the design stage. To see this, consider the same case as before, except for the fact that two separate MISRs are used to collect signatures from memory Postlogics. Under the single fault assumption, only one MISR will show the evidence of the fault, and therefore, only one Simulation Event table will be needed. The intelligent program will be able to guide the diagnostic process as if the circuit had a single embedded memory.

This argument suggests quite clearly that *distributed* BIST *hardware is preferable to the traditional design that uses one* MISR *to collect a global test signature*. Ideally there should be a separate MISR for each memory Postlogic. This can be done without degrading the fault detection capability. It may require, though, a somewhat higher BIST hardware cost. As an alternative to distributed MISRs it is possible to have a single MISR with gated inputs, so that a signature reading may be obtained from separate memories. This, however, will increase the diagnostic time, since several reruns with gated MISR inputs would be necessary to isolate the fault. Moreover, this may require computing more signatures for reference.

There may be cases where a single Postlogic is common to more than one memory. In these cases additional control inputs may be added to make this shared Postlogic look as two separate ones in BIST mode.

There may be cases where a single Prelogic is shared by more than one memory. In these case it is possible for a Prelogic fault to corrupt multiple memories. This may show up in more than one faulty MISR signature. As a matter of fact, the intelligent program should add weight to a probable Prelogic fault, so that the diagnostic procedure investigates it first.

A decision on how many MISRs to use, and how to distribute them inside the circuit should be based on a compromise between the need for diagnostic efficiency, low hardware overhead and low performance impact.

8. THE ENHANCED DIAGNOSTIC PROCEDURE

The intelligent program has been written and added to the existing main diagnostic procedure. The intelligent program provides "advice" as to where to look for the fault based on the "symptoms" that surface in the Simulation Event Table. The enhanced diagnostic procedure, DIAGNOSE, follows the following steps. It is assumed here that the test has failed at test window I. Let i be a pointer to the current test window that was added to the diagnostic process.

(1) Let $i = I$.
(2) If $i = 0$ no single fault can explain the observed failing signature. The fault is unexplainable. Stop.
(3) Restart all test windows from i to I and record each pattern activity. Complete the Event Table and Simulation Event table for these test windows.
(4) Perform fault simulation according the advice provided by the intelligent program; use the appropriate simulation model as the case warrants (the different simulation models have been discussed earlier).
(5) If a single fault (or fault equivalence class) explains all fails – initiate a repair action accordingly – Stop. If no such fault (or fault equivalence class) is found – continue.
(6) Set $i = i - 1$ and go to Step 2.

If a single pass through DIAGNOSE does not reveal a repair action, it backtracks to a previous

test window to try to explain the symptoms by a latent fault that might span more than one test window. DIAGNOSE, unless otherwise directed, will keep trying to explain the symptoms by going as far back as the initial test window. The test engineer, however, may want to limit the backtracks to save on diagnostic time. If backtracking does not help isolate the fault, multiple faults are then probable. Since the cost of going after multiple faults is by far higher, DIAGNOSE gives up on those.

9. EXPERIMENTAL RESULTS

In order to investigate the efficiency of DIAGNOSE and the effect of single *vs.* distributed MISRs on the diagnostic resolution the following experiment has been conducted. Circuit Ckt1 is a portion of a chip that includes a single simple memory. Circuit Ckt2 is a chip that has been designed for BIST applications, has 8 memories, but has only a single MISR. Circuit Ckt3 has the same function as Ckt2, but instead of having a single MISR it has 8 (shorter) MISRs distributed over the chip to achieve separate observability from each memory. Circuit Ckt1 had approximately 30,000 equivalent gates, and circuits Ckt2 and Ckt3 had approximately 250,000 equivalent gates.

The total test length was 100 K vectors divided into 100 windows of 1 K each. We have injected 10 random stuck-faults into the circuit models and invoked DIAGNOSE. Circuits Ckt2 and Ckt3 have been injected with the same set of random faults. The results are reported in Table VII.

TABLE VII Experimental results from DIAGNOSE on 3 chip models

Circuit	No. of explained faults	Avg. diag. time in hrs.	Max. no. of test windows backtracked
Ckt1	10	0.8	3
Ckt2	5	10.2	8
Ckt3	9	1.2	15

The first thing to note from Table VII is that it took on the average 0.8 hours to diagnose the faults in Ckt1, and that DIAGNOSE was able to isolate all faults (up to their equivalence class). This is not much of a surprise since Ckt1 has only one memory, and this was the main vehicle by which DIAGNOSE was designed.

The interesting part is the comparison between DIAGNOSE performance on circuits Ckt2 and Ckt3. Besides the fact that DIAGNOSE missed many more faults in circuit Ckt2, it is obvious that DIAGNOSE has to work harder in this case as evidenced by the average diagnostic time. This is the case where a single MISR was used to compress all test data, and obviously DIAGNOSE was incapable of giving "good" advice as to where to look for the faults. So, in this case DIAGNOSE was performing an exhaustive "walk" through the set of possibilities, postulating a fault location, and then trying to prove or disprove it. Since the total diagnostic time given to an experiment was limited, DIAGNOSE did not have enough time to backtrack far enough to catch 4 of the latent faults. In circuit Ckt2 DIAGNOSE was only able to backtrack a maximum of eight test windows, compared to 15 for circuit Ckt3. Thus, DIAGNOSE was more efficient in chasing the faults in circuit Ckt3, and successfully explained 9 out of the total 10. This higher efficiency was due to the separate MISRs that existed in Ckt3.

Another point to notice is that the average diagnostic time for circuit Ckt3 is only slightly higher than that for Ckt1. This slight increase is due to Ckt3 having some shared Prelogics between memories, and being 8 times larger than Ckt1. This definitely proves that DIAGNOSE is quite suited to handle BIST structure with multiple memories *provided attention is given to the proper distribution of the* MISRs.

What happened to the 10th fault in the case of circuit Ckt3? As it turned out, the pseudo-random test of length 100 K was insufficient to detect this fault. So, since the fault was not detected, DIAGNOSE did not have a chance to go after it.

10. CONCLUSIONS

A method has been described for using tester-loop-independent fault simulation to diagnose faults in the Prelogic, Postlogic, the Address Logic, and the Control Logic of an embedded memory. A simulation event table is generated which shows the data source of each test operation. Each event in the table can then be simulated using the source primary input values with the memory removed to identify the failing tests. Fault simulation of the failing tests with the appropriate simulation models will identify the fault or fault equivalence classes. When fault simulating with replicated copies of the Prelogic, as required by some memories, the same fault site in each of the copies must be provoked.

The program DIAGNOSE has been written to efficiently isolate faults in BIST designs having embedded memories. DIAGNOSE includes an "intelligent program" running on heuristics to reduce the simulation load by suggesting either a control or an address logic fault. Special techniques for isolating these type of faults were described. It was noted that these techniques must be tailored to the specific design of the memory under test.

The heuristics used here require that certain control capabilities exist over the embedded memories. We list these capabilities here for ease of reference.

(1) It must be possible to load any desired word into any address of a memory using the correspondence inputs. (This makes it possible to load complete or partial addresses to memory before Address Logic fault isolation).

(2) It must be possible to hold the memory control lines at a constant read operation, by holding the D2 inputs at a fixed value, while the other inputs (D1, D3, and D4) are at their normal pseudo-random test values. This allows a test rerun with only read operations to isolate Address Logic faults.

(3) It must be possible to hold the D4 inputs to the Postlogic at a fixed value while all other inputs (D1, D2, and D3) are at their normal pseudo-random test values. This allows the test rerun to identify the Actual PO Values for the activity table, and permits isolating tests at which the memory control line failed.

Our experiments have shown that by allowing separate MISRs collect test responses from the different memories, the diagnostic efficiency and resolution greatly increases. This feature should be an integral part of any design for diagnosability. A cost effective way of distributing the MISRs inside the circuit is still an open question. In doing so many parameters need to be considered: hardware overhead, performance degradation, diagnostic efficiency, wireability, *etc.*

This paper concentrated on the diagnostic effort that is initiated at the time the first test window fails. There is a lot of merit, however, to continue the test even after the first failing test window in order to collect more diagnostic information. This will help narrow down the set explaining the fault to a smaller area. There are several problems associated with this approach. The first one being: how do you determine which future test window has failed, since these test windows will already be tainted by an incorrect initial seed in the MISR? Solution to this problem has been reported in [10, 11].

References

[1] Abramovici, M., Breuer, M. A. and Friedman, A. D., *Digital Systems Testing and Testable Design*, IEEE Press, Piscataway, New Jersey, 1994.

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

[3] Camurati, P., Prinetto, P., Reorda, P., Barbagallo, M. S., Burri, S. and Medina, A. (1995). Industrial BIST of embedded RAMs. *Design and Test of Computers*, **12**, 86–95.

[4] Eichelberger, E. B. and Williams, T. W. (1978). A Logic Design Structure for LSI Testability. *J. Design Automation and Fault-Tolerant Computing*, **2**, 165–178.

[5] Hatzopoulos, A. A., Siskos, S. and Kontoleon, J. M., A complete scheme of built in self tests (BIST) structure for diagnosis in analog circuits and systems. *IEEE Transactions on Instrumentation and Measurement*, **42**(3), 689–694, June, 1993.

[6] Karpovsky, M. G. and Yarmolik, V. N., Transparent memory BIST. In: *Proc. of the IEEE International Workshop on Memory Technology, Design, and Test*, pp. 106–111, August, 1994.
[7] McAnney, W. H. and Savir, J., There is information in faulty signatures. In: *Proc. Int. Test Conf.*, pp. 630–636, September, 1987.
[8] Chin Tsung Mo, Chung Len Lee and Wen Ching Wu, A self diagnostic BIST memory design scheme. In: *Proc. of the IEEE International Workshop on Memory Technology, Design, and Test*, pp. 7–9, August, 1994.
[9] Savir, J., Fast RAM test and diagnosis. In: *IASTED Int'l. Conf. on Modelling, Simulation and Optimization, on CD ROM*, May, 1996.
[10] Savir, J., Salvaging test windows in BIST diagnostics. In: *Proc. VLSI Test Symp.*, pp. 416–425, April, 1997.
[11] Savir, J., Salvaging test windows in BIST diagnostics. *IEEE Trans. Computers*, **47**(4), 486–491, Apr., 1998.
[12] Savir, J. and McAnney, W. H., Identification of failing tests with cycling registers. In: *Proc. Int. Test Conf.*, pp. 322–328, September, 1988.
[13] van Sas, J., Van Wauwe, G., Huyskens, E. and Rabaey, D., BIST for embedded static RAMs with coverage calculation. In: *Proc. Int. Test Conf.*, pp. 339–348, October, 1993.

Authors' Biography

Dr. Savir holds a B.Sc. and an M.Sc. degree in Electrical Engineering from the Technion, Israel Institute of Technology, and an M.S. in Statistics and a Ph.D. in Electrical Engineering from Stanford University. He is currently a Distinguished Professor at New Jersey Institute of Technology, where he held the position of Director of computer engineering (1996–2000), and Newark College of Engineering Associate Dean for research (1999–2000). Previously with IBM, Dr. Savir was a Senior Engineer/Scientist at the IBM PowerPC Development Center in Austin, TX; at IBM Micro electronics Division in Hudson Valley Research Park; at IBM Enterprise Systems in Poughkeepsie, NY, and a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, N.Y. He was also an Adjunct Professor of Computer Science and Information Systems at Pace University, N.Y, and SUNY Purchase, N.Y.

Dr. Savir's research interests lie primarily in the testing field, where he has published numerous papers and coauthored the text "Built-In Test for VLSI: Pseudorandom Techniques" (Wiley, 1987). Other research interests include design automation, design verification, design for testability, statistical methods in design and test, fault simulation, fault diagnosis, and manufacturing quality.

Dr. Savir has received four IBM Invention Achievement Awards, six IBM Publication Achievement Awards, and four IBM Patent Application Awards. He is a member of Sigma Xi, and a fellow of the Institute of Electrical and Electronics Engineers.