Assertion Based Functional Verification of MBIST Controller Using Coverage Analysis

Ashwini Kumar, Akshay Mann

Abstract— For the functional validation of hardware designs, simulation with coverage analysis is still the primary means at RTL description of design. Here coverage analysis shows the way towards the optimal use of resource, validation or verification completeness and untested areas of HDL design. The complete coverage analysis of Memory Built in Self-Test (MBIST) controller using several code and functional coverage metrics is presented. Coverage metrics are used to keep the focus at assertions written to verify the functionality of MBIST controller. Use of assertions and coverage metrics gets rid-off massive test patterns to validate the MBIST controller design more strongly. This paper also presents the implementation of March algorithm based MBIST controller using System Verilog. With the help of assertions written in System Verilog and their coverage analysis, the test cases are reduced from 88 test cases to 25 test cases to achieve the approximately same functional coverage i.e. 97 % is also discussed in the paper.

Index Terms— *MBIST Controller, System Verilog Assertions (SVA), FSM, Coverage Analysis, Code Coverage, Functional Coverage.*

1. INTRODUCTION

A complex System on Chip (SoC) design comprises several embedded memories to increase the processing speed. The testing of memories is very crucial because of very low yield of memories during fabrication. They are very much affected by several memory faults. But in case of embedded memories it becomes a tedious task because of no primary inputs and outputs to external of SoC. For this purpose a MBIST circuitry is added with memory which make this testing job very comfortable while adding some area over head.

The main and important part of MBIST circuitry is MBIST controller which controls the sequence of all operations during testing [1]. The fault detection in the memory depends on the read and write operations sequence controlled by the MBIST controller. It is good to know that the design of MBIST controller does not depend on size of memory under test. For the reliability purpose of MBIST, the functional verification of the MBIST controller design is utmost necessary.

Functional verification of MBIST controller is the key to guarantee the quality of design and reliability for required functionality. Verification efforts has been put on verifying the correct functionality at initial register-transfer level (RTL) descriptions of MBIST controller written in hardware description language (HDL). Traditionally, the functional verification is done by simulation with large number of test patterns but at the same time formal verification techniques claim to verify the design at different design levels. The simulation based methods will continue to be an important part of verification process until the computing complexity of the formal verification methods reduces [2].

But during simulation, an important question always bothers to the designers and the verification engineers: Are we done yet? Or is the verification completed? In the typical design environments the verification is assumed to be completed if design engineer thinks that he has done complete and thorough simulation. But here, the quality of the test cases depends on the designer’s understanding of the design and that is not measurable.

A verification engineer can determines the level or depth of verification by coverage analysis of the design under verification (DUV). It can be analyzed that which part of HDL code of design is tested or not during simulation by monitoring the execution of code. The coverage analysis highlights the uncovered code portion[3] thus It provides the clear understanding where to put the effort to test untested the design functionality to achieve 100% coverage which is desirable for any design. Achieving 100% coverage cannot give 100% surety that design is error free. It provides the systematic approach to attain completeness of verification.

For this purpose, code coverage and functional coverage metrics are used to verify the design in HDL. The code coverage comprises the several coverage metrics and can vary based on used tool for coverage analysis [3]. The basic idea behind these coverage metrics is to cover up the design structure completely written in HDL. While the functional coverage focuses on the functionality of the design. Till now

IJSER

there is not such a metric which is accepted for complete and reliable verification coverage.

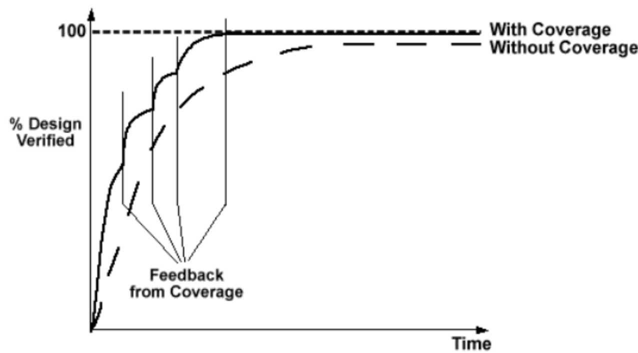


Fig. 1. Effect of applying coverage analysis.[3]

The improvement in the verification in less time is possible by using coverage analysis is shown in Fig. 1. Nevertheless the simulation based validation of design is partial complete.

This paper includes the FSM design of MBIST controller based on March algorithm with pause logic using system Verilog to detect different memory faults such as address faults (AFs), stuck at faults (SAFs), transition faults (TFs), and coupling faults (CFs) and retention faults in SRAM memory. For the complete validation of MBIST controller, simulation based and formal verification technique are combined for coverage validation of MBIST controller.

The rest of paper is organized as follows: section II shows the implementation of MBIST controller which is taken as DUV. Section III comprises assertion based functional verification of MBIST controller. Section IV comprises the different coverage metrics, coverage analysis and coverage results. Section V includes final results and test case analysis. In section VI paper is concluded.

2. FSM IMPLEMENTATION OF MBIST CONTROLLER

The controller generates the control signals for other components of MBIST circuitry such as patt_g for data/pattern generator, en to start the address counter inside the address generator, rw for Read/ writes generator and en for signature analyzer and other components too. The top level block diagram of MBIST controller is shown in Fig. 2. Mainly the controller handles the test sequence and its result corresponding to memory output. After assertion of T_mode signal from higher level processor or controller, MBIST controller generates the control signals corresponding to the pattern generator, address generator, read/write and signature analyzer. Controller and pattern generator control the up or down address sequence by generating the control signals for address generator. The generation of March pattern is based on the '0' or '1' value of control signal patt_g whether it is marching 0 or 1. Controller generates the control signal rw to the read/write controller for the reading or writing operations from and into the memory.

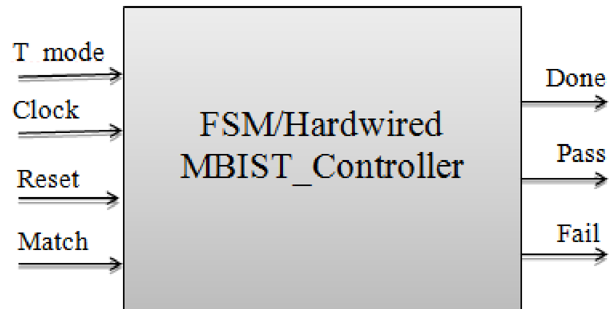


Fig. 2. Top level block diagram of MBIST controller

Match signal is the output of signature analyzer and input to the controller. Based on the value of match, controller asserts the pass or fails if match is '1' or '0' correspondingly. Only after completion of March test sequence, controller asserts the done signal.

Pause element is added just after the write element to check which memory cell is not capable to retain the same written data after a particular time. During pause the controller does not allows the execution of the original march test sequence of read/write operations on memory under test.

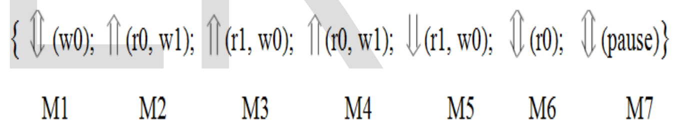


Fig. 3. March C algorithm with pause element.

The controller's implementation is based on March C algorithm [4] which is capable to detect SAFs, AFs, TFs, CFs (except linked CFs). The March C algorithm includes 6 march elements say M1, M2, M3, M4, M5 and M6 [5]. But here in Fig. 3, the implemented algorithm comprises an additional element pause say M7 to check the memory data retention time which generates memory retention faults.

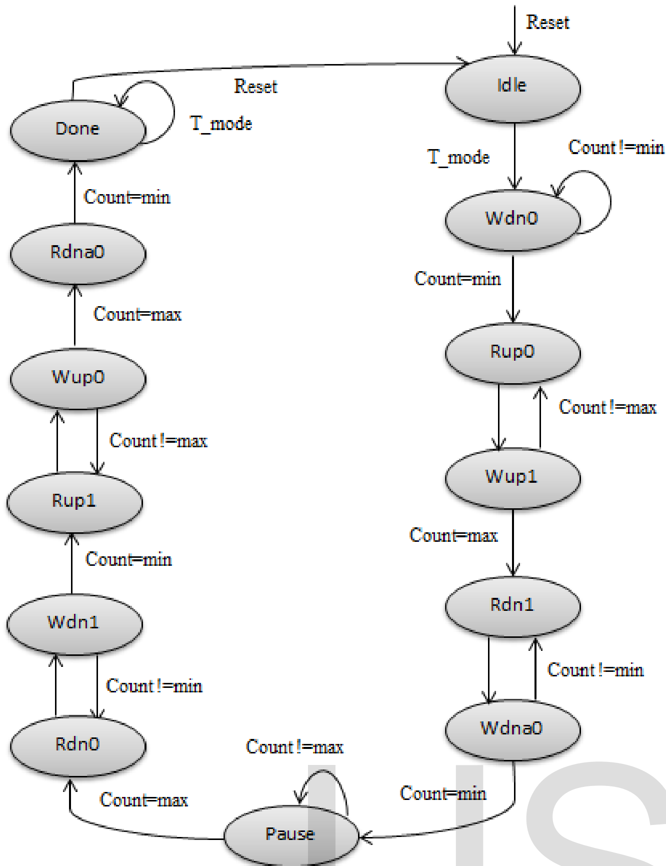
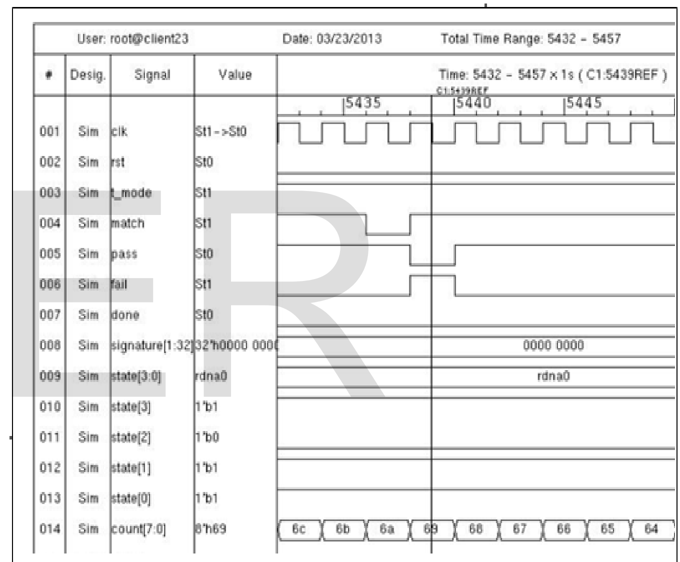


Fig. 4. Finite State Machine Diagram of MBIST Controller

The Fig. 4 shows the implemented FSM structure of MBIST controller which is easy to implement as a FSM and very feasible if no further improvement is required in MBIST controller design. The above FSM state diagram consists several states and every state represents a particular march element except state idle.

Initially controller rests at Idle state and keeps other the MBIST circuitry in idle until t_mode signals is asserted. But after getting value '1' at t_mode signal, controller starts traversing the next states to generate required pattern for read or write operations from and into the memory cells. Now in offline testing MBIST controller takes the controls of memory from higher level processor. At every state a read/write signal 'rw' gets a value '1' or '0'. State Wdn0 comes when t_mode signal is asserted high and at every maximum and minimum count value the present state changes to the next state as shown in Fig. 4. The maximum value of count depends on the address width of SRAM memory. In normal condition if a state transition happens, it means a March element gets complete. At first March state Wdn0, 'rw' signal is kept at '0' and read/write generator write data in memory until count value gets minimum. Here 'W' represents the write operation, 'dn' shows the down marching and '0' shows that marching pattern is 0. During up marching ( ), count starts from minimum address value to the maximum address value and state changes only when count gets its maximum value unless t_mode and reset signal change. In case of down marching, state transition occurs only when count gets its minimum value. Pause state is inserted between a write state (Wdn0) and a read state (Rdn0). During pause, controller just holds the marching operation on memory for the specified time. The March testing follows the read-test-write sequence that's why every read state in FSM design comprises the test result as a signal pass /fail. After completion of test controller enters into the Done state and controller asserts the done signal to show that the test is completed. The FSM shown in Fig. 4 is implemented using System Verilog in Synopsys VCS tool.



. Fig. 5. Simulation output

In the simulation output waveform only input and output of MBIST controller are shown in Fig. 5 except count and signature. The '1' value of pass or fail signal indicates the success and failure of corresponding applied data pattern on the memory. When the match signal gets '0' value in the next clock cycle, MBIST controller assert the fail high and low to the pass signal. Again when match goes high the controller assert pass signal high and low to fail. Whenever rst (reset) signal goes high, the state immediately switches to the idle state. After attaining idle state, there is no transition in the state of controller and state gets stable until rst gets low value again. Controller starts transitions state to next state only when rst is low and t_mode signal gets high value.

IJSER

Simulation output shown in Fig. 5 is only based on some test cases and provides the functionality check but not completely. With such functional checking, there is a possibility to leave some corner cases untested. And there is possibility of bugs in such untested corner cases. These corner cases are tested by using system Verilog assertion for design's functional properties and by adding different test cases after analyzing the coverage metrics for the MBIST controller.

3. Functional Verification of MBIST controller

Assertion Based Verification of MBIST Controller

Assertion Based Verification addresses all the challenges faced during only simulation based validation of design without the use of assertions. Here, assertions are defined for MBIST controller with complete assertion's ontology which includes the information about assertion under construction such as class, type, name, expression, message, severity level, condition and snippet. The SVAs are written for MBIST controller to check the required specifications and functional properties according to verification plan. Based on the verification requirements, assertions for MBIST controller design are categorized only on the basis of class and other ontological information would be covered with this.

Class: Two main types of classes to define assertions for MBIST controller are interface boundary and internal functional spots.

At Interface Boundary of MBIST Controller:

It is a block level based approach to write the assertions for MBIST controller. It is mentioned in section 2 that MBIST controller interacts with several blocks in MBIST architecture and other than MBIST too. SVAs are written here to check whether the controller interface/boundary signals are according to the required controller's specification or not. Here, assertions mainly check the timing requirements among the interface signals.

Table 1. Verification requirements for the interface signals.

Interface Features	Interface signals and their features	
	Signals	Expected features
1.	Reset, T_mode	T_mode=1 and reset=0 and its opposite must be synchronous
2.	T_mode, en	en=1 with in 1clock cycle after t_mode is asserted
3.	Match, Pass,	Pass=1 with in 1clock cycle after match is asserted
4.	Match, Fail,	Fail=0 with in 1clock cycle after match=0

Internal Functional Spots of MBIST Controller: MBIST controller is the control block of MBIST which is implemented as a FSM. The main verification requiremnets of FSM are defined as the transitions among the states, sequencing and timing requirements. For the every march operation shown in figure 2, there is a different state in controller's FSM design.

Now here, assertions are written according to verification plan and classified to verify the llegal state transitions, correct sequencing, timing requirements and functionality in each state.

4. COVERAGE ANALYSIS FOR MBIST CONTROLLER

Coverage analysis makes functional verification easy and less time consuming and it also helps to know whether functional verification is done enough or not. Thats why coverage analysis plays important role to check the verification completeness of MBIST controller.

The controller's coverage analysis depends on two main coverage, one is code coverage and another is functional coverage. Code coverage is independent of functional coverage but it helps to obtain verification completeness of MBIST controller.

A. Code Coverage Analysis of MBIST Controller

Here eight different very useful coverage metrics are introduced that can be classified as code coverage metrics for MBIST controller. Sometime different names are used for similar types of coverage metrics [2] depending upon the understanding and used coverage tool. Code coverage metrics helped to identify which structure in HDL code of controller is exercised during simulation.

1) Line Coverage*:* Line is one of the simplest structures of HDL code. Line coverage metric as Fig. 6 shows the result of exercised lines with respect to the total number of lines present in HDL code of MBIST controller. A line of system Verilog code is said to be covered if it has had transaction on it. An event may or may not occur during transaction at that line but line count as covered. [6]

IJSER

Module Name	Blks (%)	Blks	Stmnts (%)	Stmnts	Lines (%)	Lines
\$unit::nc_mbistcontroller.sv::tb3_mbist_controller.sv						
	--	0/0	--	0/0	--	0/0
tb_bist_cont	99.80	508/509	99.80	1018/1020	99.75	400/401
tb_bist_cont.uut	98.08	51/52	98.25	112/114	97.06	33/34

Fig. 6. Line coverage score for MBIST controller.

In Fig. 6, the total line coverage score is around 98.5% for system verilog HDL code of MBIST controller . At the same time the Fig. 7 shows the covered lines and uncovered line in HDL of controller. It helped to identify the cause of not covering the particular lines and shows the effectiveness of test suits.

2) Statement Coverage: In this coverage analysis of MBIST controller, it only counts the executable statements. A line may contains more than one statements .For a conditional statement at line 36 in Fig. 7, there one countable statement state<= s_idle considered for *if* statement and multiple countable statements are considered for *else* statement [2]. Fig. 6 shows the total statement coverage score is around 99%. In Fig. 8 at line 68, the red shaded portion shows the uncovered countable statements for else statement.

```
29    always_ff@( posedge clk or posedge rst)
30   begin
31   if(rst) begin state<=s_idle;en<=1'b0;done<=0;fail<=0;pass<=1; count<=8
32   else
33
34
35    case (state)
36    s_idle: if(~t_mode) begin state<=s_idle;end else begin en<=1'b1;state
37
38
39    wdn0:begin count<= count-1; if(count ==c_min) begin state =rup0;rw<=1
40                else state =wdn0; end
41
```

Fig. 7. Covered lines in the HDL code of MBIST controller.

2) Block Coverage: Block coverage is also called segment coverage of HDL code. Its measuring unit of code is a sequence of non-branching codes which is executed at the same simulation time [2]. Block coverage reduces the recording units for coverage analysis. Mostly system tool records as block coverage and display results in term of statement coverage but there is a slightly difference between them. In Fig. 6, the block coverage score is shown as around 99% with total number of blocks extracted from the MBIST controller HDL code by Synopsys-VCS tool.

3) Branch Coverage: Branch coverage metric is also called decision coverage matric. This metric involved the control flow through the MBIST controller HDL code during simulation and can be represented as control flow graph (CFG) to code [7]. As it measures the coverage of each branch present in *if* and *case* statements. It focused on the decision points or control statements that affect the control flow of the controller's HDL code execution. From a decision point, different branches originates in HDL such as form an *if* statement, there are two branches one is for true and another is for false case. Decision coverage will report the evaluation in both true and false cases during simulation. For the *case* statement present in FSM HDL code of MBIST controller, decision coverage verified that each branch of the *case* statement, including the *default*. But the *default* state in the designed MBIST controller is idle state which is controlled by reset signal which is at high value by default. It showed uncovered default branch and one other at line 68 shown in Fig. 8. The total branch coverage score for designed controller is 94.74 shown in Fig. 9. It is very much possible that at initial attempts line coverage is around 100% and branch coverage is much less than of it. It shows that there are still untested cases.

```
64        wdn1:begin  count<=count-1; if(count==c_min)begin count<=c_min;rw<=1;
65            state=rup1;end else state= wdn1;end
66
67         rup1:begin  count<=count+1; if(match)begin pass<=1'b1;fail<=1'b0; end
68                                     else begin fail<=1'b1; pass<=1'b0; end
69            if(count==c_max)begin state=wup0;g_patt<=0; rw<=0;count<=c_min;end
70            else state=rup1;end
71
72         wup0:begin count<=count+1; if(count==c_max)begin count<=c_max;rw<=1;
```

Fig. 8. Covered branches in HDL code of MBIST controller.

IJSER

The difference between line coverage and branch coverage of MBIST controller is due to untested default branch which is due to implied design of case statement in controller's HDL

Therefore, Branch coverage Metric is considered to be more complete than line coverage matric.

Module Name	Branches (%)	Branches
\$unit::nc_mbistcontroller.sv::tb3_mbist_controller.sv		
	--	0/0
tb_bist_cont	94.74	36/38
tb_bist_cont.uut	94.74	36/38

Fig. 9. Branch coverage score for MBIST controller

4) Path Coverage: Path coverage measures the coverage of all paths present in the HDL code. A path is defined as a unique sequence of branches or decisions from the starting of a code section defined in HDL to the end of it. This sequence must be corresponding to a path in the control flow of the HDL code. A path in HDL code must contain an edge which is not included in other paths [7]. In HDL code of controller, there are many if statements which generates the different branches and sequences. In other

International Journal of Scientific & Engineering Research, Volume 4, Issue 11, November-2013
ISSN 2229-5518

words, a control statement generates a different path in the HDL code. Path coverage is similar to decision coverage but it covers multiple sequential decisions.

```
Module Name                                  Path
                                             (%)
\$unit::nc_mbistcontroller.sv::tb3_mbist_controller.sv
                                             --
tb_bist_cont                                 81.58
tb_bist_cont.uut                             81.58

//*******************************************************

//            Total Module Instance Coverage Summary

//                              TOTAL     COVERED     PERCENT
//     paths                    38        31          81.58
```

Fig. 10. Path coverage score for MBIST controller

The branch of *case* statement (In the HDL code of MBIST controller) on line 67 followed by the *else* branch of *if* in Fig. 8 defines one path. The total path coverage score of MBIST controller is 81.58% as shown in Fig. 10. If a branch is not covered in HDL code then it may stop the coverage of several paths through this branch and generates the sequencing error in the design that's why path coverage metric is more complete than the branch or decision coverage metrics. It is a very cumbersome because of very large number of paths in a design which makes 100 % score impractical for path coverage [2].

IJSER

5) Conditional coverage*:* Conditional coverage or multiple condition coverage [8] is sometimes called expression coverage because the conditions are evaluating based on variable or expression in the conditional statements. It provides the coverage statistic for variables and expressions. It is shown in Fig. 11 as shaded portions (as evaluated expressions) how a conditional coverage metrics records the coverage by evaluating the variables or expressions. The total conditional coverage score is 100% shown in Fig. 12, it means every possible case of conditions has been evaluated during simulation of HDL code of MBIST controller. It is a very important and critical coverage metrics because it can find the errors in the conditional statements that cannot be easily found by any other coverage analysis [2].

```
39  wdn0:begin count<= count-1; if(count ==c_min) begin state =rup0;rw<
40              else state =wdn0; end
41
42        rup0:begin count<=count+1;if(match)begin fail<=0; pass<=1'b1
43                              else begin pass<=0;fail<=1'b1;
44           if(count == c_max) begin state= wup1;g_patt<=1;rw<=0; c
45              else state =rup0;end
46
47        wup1:begin count<=count+1; if(count==c_max)begin state=rdn1;
48           else  state= wup1; end
```

Fig. 11. Covered conditions in the HDL code of MBIST controller.

```
Module Name        Logical  Logical     Non-logical Non-logical    Events Events
                   (%)                  (%)                        (%)
\$unit::nc_mbistcontroller.sv::tb3_mbist_controller.sv
                   --       0/0         --          0/0            --     0/0
tb_bist_cont       100.00   22/22       --          0/0            100.00 2/2
tb_bist_cont.uut   100.00   22/22       --          0/0            100.00 2/2
```

Fig. 12. Conditional coverage scores for MBIST controller.

6) Event Coverage*:* Most HDL simulators are event-driven. Therefore, it is necessary to care about the possible events in a design. Events are associated with the change of a signal. For example, as shown in the line 29 of Fig. 7, there are two event *always @ (posedge clk or posedge rst)* which wait for the *clk* or *rst* signal changing from low to high. These two are the control events [9] of complete FSM design of MBIST controller. It is shown in Fig. 12 that both events have been covered completely and event coverage score for the controller's HDL code is 100%. This coverage metrics is very useful when there are too much control events in the design.

7) FSM Coverage*:* In code coverage point of view, the FSM coverage metric cover number of traversed states in FSM design during the simulation. Here FSM coverage is defined as language-based code coverage for the MBIST controller HDL code just to show whether the all design states are traversed or not. It is most important coverage metric for presented MBIST controller design because it found out most of the design bug due to its closeness to the behavior of design space. The transitions and sequencing coverage of MBSIST controller's FSM is defined in the functional coverage analysis in next subsection of paper.

```
Test Coverage Result: Total Coverage

Module Name        States  States     Trans.  Trans.     Seq.   Seq.
                   (%)                (%)                (%)
\$unit::nc_mbistcontroller.sv::tb3_mbist_controller.sv
                   --      0/0        --      0/0        --     0/0
tb_bist_cont       100.00  13/13      100.00  24/24      99.88  840/841
tb_bist_cont.uut   100.00  13/13      100.00  24/24      99.88  840/841
```

Fig. 13. FSM coverage score for MBIST controller.

The FSM state traversed coverage score is 100% as shown in Fig. 13. The all 13 states have been traversed in HDL of MBIST controller written in system Verilog during simulation shown as shaded portion in Fig. 14.

```
enum logic [3:0] {s_idle,wdn0,rup0,wup1,rdn1,wdna0,pause,rdn0,wdn1,rup1,wup0,rdna0,s_done} state;

 logic done,pass,fail,en ;
 logic [1:sign_size] signature;
 logic [c_size-1:0] c_max = 8'd255;
logic [c_size-1:0] count;
 logic [3:0] state;
logic [1:sign_size]mem_out;
 logic match ;
 logic [c_size-1:0] c_min =8'd0;
logic rw,g_patt;
logic pass,fail;
integer var_count =8'd0;
```

State	rdn1	Covered
Transition	wup1->rdn1	Covered
	rdn1->s_idle	Covered
	rdn1->wdna0	Covered
Sequence	wup1->rdn1	Covered
	rdn1->s_idle	Covered
	rdn1->wdna0	Covered
	rup0->wup1->rdn1	Covered
	wup1->rdn1->s_idle	Covered
	wup1->rdn1->wdna0	Covered

Show 10 lines only. see more in list pane.

/data/synopsys/test/labs/verilog/project1/nc_mbistcontroller.sv

Fig. 14. Covered states and transitions of MBIST controller FSM.

The defining of FSM coverage metric was relatively more beneficial than other metric because it is MBIST controller design dependent. The main problem was to write the coverage directed tests so that all states, transition and sequences gets covered.

B. Functional coverage Analysis of MBIST Controller

Here defined functional coverage metrics are corresponding to the designed FSM controller. Metrics defined in this category such as toggle coverage, Sequence and transition coverage of FSM and assertion coverage are related to computation performance of HDL code rather than its structure [7]. The main motive to define such metrics was to exercise each functional scenario of MBIST controller. During functional coverage, the coverage monitor looks for error in state transitions and event sequencing mainly in FSM designs. Assertions which interpret the MBIST controller hardware functionality in system Verilog language are considered the special case to monitor.

1) Toggle Coverage*:* Sometime toggle coverage is also called as variable coverage [2]. It measures that each bit in the nets and registers or bits of logic change their polarity during simulation and not stuck at one level [10]. Toggle coverage metrics is considered as first functional coverage metric because without tested a bit properly function coverage target cannot be complete. The functional behavior of MBIST controlled is different for the toggling from 0 to 1 and from 1 to 0 of some logic bits.

//	TOTAL	COVERED	PERCENT
regs	6	6	100.00
reg bits	68	68	100.00
reg bits(0->1)	68	68	100.00
reg bits(1->0)	68	68	100.00
nets	6	6	100.00
net bits	6	6	100.00
net bits(0->1)	6	6	100.00
net bits(1->0)	6	6	100.00
logics	13	11	84.62
logic bits	99	83	83.84
logic bits(0->1)	99	83	83.84
logic bits(1->0)	99	83	83.84

Fig. 15. Toggle coverage score for MBIST controller in detail.

In Fig. 15, detail toggle coverage report is shown with toggle score of net, register and logic bits. At same time the Fig. 16 shows that which logic signal bits did not toggled completely. In Fig. 16 the red shaded signals shows that their bits are not changing polarity that's why not covered by coverage metric during simulation. But it is understood here that signal c_max and c_min in MBIST controller design are the minimum and maximum value of address counter which are constants values.

Toggle coverage analysis is proved very useful for MBIST controller functional verification because involvement of 32 bit read and write for the 32 bit wide SRAM. It is applied for structural [10] and white box verification of controller to show that the control logic is working as intended.

```
2   output done,pass,fail);
3
4  enum logic [3:0] {s_idle,wdn0,rup0,
5
6   logic done,pass,fail,en ;
7   logic [1:sign_size] signature;
8   logic [c_size-1:0] c_max = 8'd255;
9  logic [c_size-1:0] count;
10  logic [3:0] state;
11 logic [1:sign_size]mem_out;
12  logic match ;
13  logic [c_size-1:0] c_min =8'd0;
14 logic rw,g_patt;
15 logic pass,fail;
16 integer var_count =8'd0;
```

Fig. 16. Toggle coverage details of logic bits in HDL code of MBIST controller.

2) Transition and Sequence Coverage as parts of FSM coverage: As functional coverage point of view of FSM MBIST controller design, the transition coverage metric and sequences coverage metric are considered. The transitions among the states of FSM depend on the functionality of MBIST and same with the possible sequences in the MBIST controller HDL code. There are 24 possible transitions in FSM of controller which are covered by transition coverage metric shown in Fig. 13 with also 100% state transition coverage. In the Fig. 13, it is shown

that total sequence coverage score is 99.88% and all sequences presented in HDL code of controller are covered except one.

	✓ 0	✓ 1	✓ 2	✓ 3	✓ 4	✓ 5	✓ 6	✓ 7	✓ 8	✓ 9	✓ 10	✓ 11	✓ 12
✓ 0 s_idle	12/12	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 1 wdn0	12/12	12/12	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 2 rup0	12/12	11/11	11/11	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 3 wup1	13/13	12/12	10/10	10/10	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 4 rdn1	14/14	13/13	11/11	9/9	9/9	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 5 wdna0	14/14	13/13	11/11	9/9	8/8	8/8	1/1	1/1	1/1	1/1	1/1	1/1	1/1
✓ 6 pause	14/14	13/13	11/11	9/9	8/8	7/7	7/7	1/1	1/1	1/1	1/1	1/1	1/1
✓ 7 rdn0	14/14	13/13	11/11	9/9	8/8	7/7	6/6	6/6	1/1	1/1	1/1	1/1	1/1
✓ 8 wdn1	14/14	13/13	11/11	9/9	8/8	7/7	6/6	5/5	5/5	1/1	1/1	1/1	1/1
✓ 9 rup1	15/15	14/14	12/12	10/10	9/9	8/8	7/7	6/6	4/4	4/4	1/1	1/1	1/1
✓ 10 wup0	15/15	14/14	12/12	10/10	9/9	8/8	7/7	6/6	4/4	3/3	3/3	1/1	1/1
✓ 11 rdna0	12/12	11/11	9/9	7/7	5/5	4/4	4/4	4/4	2/2	2/2	2/2	2/2	1/1
✓ 12 s_done	12/12	11/11	9/9	7/7	5/5	4/4	4/4	4/4	2/2	2/2	2/2	1/1	0/1

Fig. 17. No. of sequences with sequence coverage for controller' HDL code.

The first column and first row in the Fig. 17 shows the states present in MBIST controller. The Fig. 17 shows the total number of sequences with starting states of sequences while including how many times a particular state became part of sequences. For example, there are 14 sequences started with pause state in which these sequences include the s_idle state and these all sequences have been covered. The dark shaded cell in Fig. 17 shows the untested or uncovered sequence which start from s_done state and again include itself in the sequence.

3) Assertion Coverage: Assertions are embedded within the HDL code of MBIST controller annotate the functionality of design and their main purpose was to generate the assertion coverage metric. This assertion coverage metric covered the successful execution of assertions written for functionality of MBIST controller in system Verilog HVL [11]. As shown in Fig. 18, the successful execution of assertions is shown by shaded portion. And total coverage score for the assertion coverage is 100% shown in Fig. 19 with the total functional coverage report of MBIST controller.

```
328  property done_idle;
329  @(posedge clk) ((state==done)|=>((state ==s_idle)&&(rst))); endproperty
330  Ap33:assert property(done_idle);
331  Cp33:cover property(done_idle);
332
333  sequence s1;
334  state ==wdn0; endsequence
335  sequence s2;
336  !(rw||g_patt); endsequence
337  property f_wdn0;
338  @(posedge clk) s1 |->s2 ; endproperty
339  Ap34:assert property(f_wdn0);
340  Cp34:cover property(f_wdn0);
```

Fig. 18. Covered assertions.

It is a very tedious task to write the test cases for every functional specification of design to cover or verify it and it becomes impractical for large and complex designs. Assertions are included to validate the MBIST controller design by increasing the functional coverage.

5. RESULTS

According to verification plan for MBIST controller design all required internal and interfacial functional features are exercised using SVAs which helped to apply the efficient or directed test cases to the controller. Table 4 shows the directed test cases towards the desired functionality check with total functional coverage of MBIST controller. 37 out of 53 assertion got success with only 6 directed test ceses. Based on the failed assertions, the design bugs are removed and the test cases are improved to get the 100 percent success of the assertions. After applying 25 directed test cases 100 percent assertion coverage is obtained, as shown in Fig.19 and 96.50 percent functional coverage as shown in Table 4.

The simulation output with SVAs for the MBIST controller design is shown in figure 19 and the 96.50 percent functional coverage score is shown in Fig.20. All the results for implementation and verification of MBIST controller are generated using Synopsys-VCS®.

Table 4. Shows the improvements in functional coverage by using assertions with test cases.

Directed test cases	SVAs in verification plan	Successful assertions	Total functional coverage score
6	53	37	77.86 %
25	53	53	96.50 %

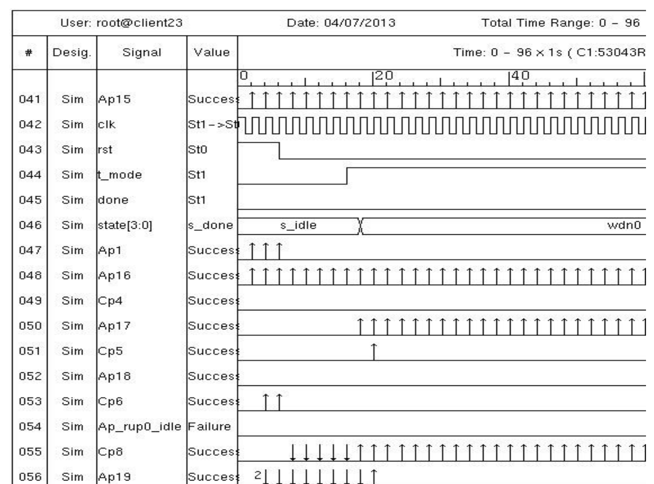


Fig 19. Simulation output during dynamic verification with assertions.

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	ASSERT
96.50	99.35	100.00	90.17	100.00	89.47	100.00

Hierarchical coverage data for top-level instances

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	ASSERT	NAME
96.50	99.35	100.00	90.17	100.00	89.47	100.00	tb_bist_cont

Fig 20. Total Functional coverage

6. CONCLUSION

In this paper, Imprementation and Verfication of MBIST controller using System Verilog is presented. The use of system verilog also increased the observability of the design. March C algorithm based MBIST controller is designed with an improvement in the algorithm by adding a pause element to test memory retention faults. To verify the designed MBIST controller against intended features, assertion based verification and coverage analysis approach is used. ABV and coverage analysis approach helped to make the verification and design process efficient and less time consuming by finding the bugs, exercising the corner cases in the design and by finding the directed test cases with less efforts. ABV helped to write directed and efficient or approx 32 % less test cases with 100% assertion coverage and approx equal total functional coverage i.e. 97 % approx., than the use of posssible random test cases for implemented MBIST controller design. In this way ABV using coverage analysis helped to fasten the design and verification process with better quality of the design.

ACKNOWLEDGEMENT

I would like to take this opportunity to express my gratitude to the people whose assistance has been invaluable in this paper. I would like to thank the Director of Indian Institute of Information Technology, Allahabad for providing financial and infrastructure support for making this work possible.

REFERENCES

[1] Nor Zaidi Haron1, Siti Aisah Mat Junos, Abdul Hadi Abdul Razak Mohd. Yamani Idna Idris4 *"Modeling and Simulation of Finite State Machine Memory Built-in Self- Test Architecture for Embedded Memories"* in proc. of IEEE, APC on applied electromagnetic, 2007,pp.1-4244-1435.

[2] Jing-Yang Jou and Chien-Nan Jimmy Liu "*Coverage Analysis Techniques for HDL Design Validation"* NOVAS Software Inc. under grant NSC89-2215-E-009-009.

[3] David Dempster and Micheal Stuart *"Techniques for Verifying HDL Designs"* Teamwork International and TransEDA Limited, june 2002.

[4] A.J. van de Goor and Yervant Zoriaii "Effective March Algorithms for Testing Single-Order Addressed Memories" IEEE,1993 1066-1409.

[5] Masnita M.I, Wan Zuha W.H., R.M. sidek and I.A Halin, *"The Data and Read/Write Controller for March Based SRAM Diagnostic Algorithm for MBIST"* in proc. of IEEE, SCROeD 2009, pp. 978-1-4244-5187-6.

[6] M. Zia Ullah Khan, and Sandra R. Clark *"Using Code coverage to Enhance Design Validation"*, Intel Corporation.

[7] Serdar Tasiran and Kurt Keutzer, *"Coverage Metrics for Functional Validation of Hardware Designs"*, in proc. of IEEE, Designs & Test of Computers'2001 , pp- 0740-7475.

[8] Graeme Cunningham, Paul B. Jackson and Julian A. B. Dines, *"Expression Coverability Analysis: Improving Code Coverage Analysis with Model Checking"*, Design and Verification Conference (DVCon), San Jose, March 2004.

[9] Richard C. Ho and Mark A. Horowitz, *"Validation Coverage Analysis of Complex Digital Design"*, in proc. of IEEE, ICCAD'1996, pp-1063-6757.

[10] System Verilog, *IEEE Standard for System Verilog*, IEEE Std. 1800-2012.

IJSER