# Implementation of an Algorithms for Improve the Quality of Memory Built-In Self-Test Tools

**C.Gangaiah Yadav[1] Dr. K.S. Vijula Grace[2]**
[2]Assistant Professor
[1,2]Department of Electronics and Communication Engineering
[1,2]Niu, Kumaracoil.Tamil Nadu, India

*Abstract*— In the paper, presented the two techniques ensuring the high quality of the memory built-in self-test. The described ideas illustrate general methods and are applicable to any commercial memory BIST tool. The first approach describes controller emulation in order to validate each step of the real controller's operations. The second solution presents a way to determine the test algorithms' fault coverage by means of the memory fault simulator. The experimental results shows functional benefits and effectiveness of the proposed methods.

***Key words:*** MBIST, TMB Validator, user-defined algorithms (UDA), Memory Fault Simulator, Controller Emulation, Fault Coverage

## I. INTRODUCTION

Nowadays, EDA industry is seeking maintenance methodologies to support its software, and to improve the overall quality of tools as they are crucial factors affecting customer satisfaction. Ensuring high quality is a recurring problem in now a days. Monitoring activities of tools and detecting post- development software errors cannot be overestimated. There are many methods to ensure product quality. This paper presents two effective techniques to solve the quality assurance problem for memory built-in self-test tools.

Commercial MBIST tools are state-of-the-art products that successfully tackle challenges of memory test by deploying many intricate components. One of them the MBIST controller is becoming more and more complicated with the advent of complex memory structures. Other essential parts of any MBIST scheme the actual test algorithms are customizable routines allowing one to shape the resultant fault coverage, especially in the presence of new and complicated fault models that require adequate test generation schemes. Their implementations, however, are error-prone, and hence they need appropriate quality assurance-based maintenance.

An effective and robust MBIST tool must comprise only reliable components. In particular, various components of an MBIST tool require different test techniques. Among them, controller emulation enables monitoring of its activities. As for test procedures, their exact fault coverage is always of primary interest, especially if one uses user-defined algorithms (UDA) that many MBIST tools are capable of running.

In the case of simple test algorithms, the design process is very simple and it tests the given modules simply.

.

In the case of simple test algorithms, UDA validation can be done manually. Typical memory test schemes, however, may require a non-trivial mathematical treatment. It turns out that memory fault simulators may serve this purpose. Existing contributions in this area present different techniques that depend primarily on the type of a modeled memory and applications of the simulator.

In this paper, we demonstrate how to employ a memory fault simulator in the quality assurance process. With no loss of generality and for the sake of clarity, the presented analysis is confined to a single memory design. It can be, however, easily adapted to other memory architectures, as well.

## II. MBIST CONTROLLER EMULATION

The overall objective of our emulator is to verify whether the MBIST controller works as expected. For this purpose, real controller activities are compared with a reference generated by the independent emulator. The idea of validating the MBIST controller through its emulation stems from the fact that test algorithms are independent of the controller. Since the emulator accepts a high-level description as its input, the probability of making the same mistake in the emulator and in the controller is relatively low, and thus the reliability of the proposed approach is increased. We have decided to describe test algorithms in Tcl the leading scripting language for EDA tools.

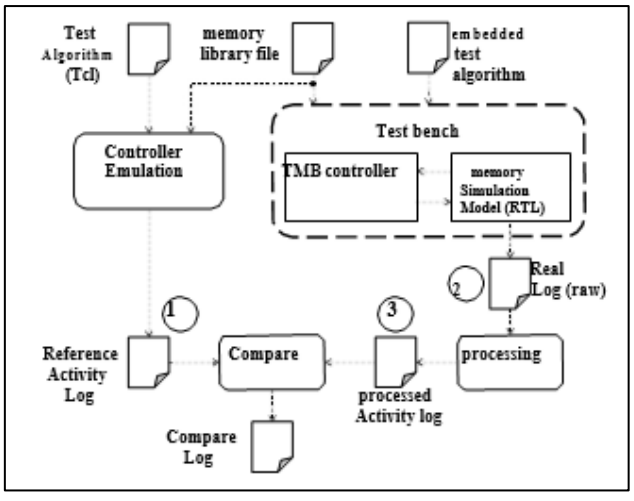


Fig. 1: TMB Validator workflow

In order to validate the MBIST controller operations, a TMB Validator has been developed. Given a set of parameters, it first instantiates the controller, and then tests the controller's features when performing memory read and write operations. In particular, the tool verifies whether the correct values are written at the right addresses through the corresponding memory input ports. Moreover, other signals, such as output enable, write enable, chip select, and write

 

mask, are checked. Similarly, the read ports and addresses are examined and verified in addition to the correct read data values. It is worth noting that "don't care" output bits are not taken into account, and only specified bits do matter when the read data pattern is evaluated.

Workflow. As shown in Fig. 1, both the controller emulator and the MBIST controller use the same memory library file as their input. On the other hand, test algorithm descriptions are customized. Implementation is independent of algorithms included in the MBIST controller library. It is a significant advantage because the same mistake in the MBIST controller and in the emulator is rather unlikely. The controller execution and emulation produce two stream-like activity logs. The real one (file (2) in the Fig. 1) is further processed to enable a log files comparison. The compare log exposes differences between the reference activity log (1) and the processed activity log (3). Supported modes of operation. The TMB Validator can run in various modes. One of them generates a simulation RTL model of a given memory architecture and the corresponding memory library file. Automated generation of memory models allows their application in other areas and it is useful in performing massive tests.

In another mode, the TMB Validator emulates the controller by working with the memory library file and the test algorithm script. In the first phase, it parses the memory library file to learn the memory parameters. Subsequently, the Tcl interpreter executes a test algorithm and creates file (1) comprising a complete list of expected read and write operations, as presented in Fig. 2. For every operation, the entire address is split into three parts corresponding to a bank, a row, and a column. This is followed by the infor- mation regarding the data pattern (binary), the current write mask (hexadecimal), and the activity type. Letter "x", repre- senting "don't care" bits in file (1), indicates values in the real log file that should be ignored. Every line of file (1) corresponds to a separate clock cycle. If several operations are performed in parallel with the primary controller activity, they occur in the same line ordered by the port number. In the case of the same port numbers, the operation type takes priority so that the order of operations within a line is always unique. As the same order is preserved in the processed activity log file, there is no risk of unambiguous comparisons.

```
port=0;bankA=0;rowA=3;colA=3;data=01010101;mask=ff;oper=write;port=1;bankA=0;rowA=2;colA=3;data=xxxxxxxx;oper=read;
port=0;bankA=0;rowA=4;colA=0;data=xxxxxxxx;oper=read;port=1;bankA=0;rowA=4;colA=0;data=xxxxxxxx;oper=read;
port=0;bankA=0;rowA=4;colA=0;data=10101010;mask=ff;oper=write;port=1;bankA=0;rowA=5;colA=0;data=xxxxxxxx;oper=read;
port=0;bankA=0;rowA=4;colA=1;data=xxxxxxxx;oper=read;port=1;bankA=0;rowA=4;colA=1;data=xxxxxxxx;oper=read;
port=0;bankA=0;rowA=4;colA=1;data=10101010;mask=ff;oper=write;port=1;bankA=0;rowA=5;colA=1;data=xxxxxxxx;oper=read;
```

Fig. 2: Example of the reference activity log file

Another mode of operation is the MBIST controller generation. Here, the only common file with the controller emulation mode is the memory library file. Thus, the controller emulator and the MBIST controller run independently. Based on the input information, the TMB Validator in-vokes the tool handling the MBIST controller, the test bench generation, and the final simulation. As a result of the test bench simulation, additional information is printed to the (raw) simulation log file (2) shown in Fig. 3. These lines originate from the memory Verilog simulation file. It is a special memory model reporting every read and write operation performed by the MBIST controller. In the case of parallel operations, each of them is recorded in a separate line with identical time stamps. All addresses are printed decimally with no breakdowns into the row, column, and bank segments.

```
TMBV:  521.3ns;port=0;addr=15;data=01010101;mask=ff;oper=write;
TMBV: 521.3ns;port=1;addr=11;data=10101010;oper=read;

TMBV:  526.3ns;port=0;addr=16;data=xxxxxxxx;oper=read;
TMBV: 526.3ns;port=1;addr=16;data=xxxxxxxx;oper=read;

TMBV:  531.3ns;port=0;addr=16;data=10101010;mask=ff;oper=write;
TMBV: 531.3ns;port=1;addr=20;data=xxxxxxxx;oper=read;

TMBV:  536.3ns;port=0;addr=17;data=xxxxxxxx;oper=read;
TMBV: 536.3ns;port=1;addr=17;data=xxxxxxxx;oper=read;
```

Fig. 3: Example of the (raw) simulation log file

The discrepancies between files (1) and (2) imply further processing of file (2) in order to compare it against the data produced by the controller emulator. The processing of file (2) is performed before execution of step Compare, which is detailed in the following.

Log files comparison. The comparison of two log files is the last mode supported by the TMB Validator. In order to compare the reference activity log against its real activity counterpart, file (2) is modified. First, based on the memory library model, the address is converted into row, column, and bank segments. Next, all parallel operations are determined (based on identical time stamps) and are put into a single line ordered by the port ID and the type of operation. As a result, the processed activity log (3) is obtained, as shown in Fig. 4.

Finally, the actual comparison of files (1) and (3) is per- formed in a line-by-line manner. First of all, the lengths of lines are compared to speed up the entire process. If they do not match, then further processing is void, the "Different line lengths" message is recorded, and the comparison of that line is aborted. If the lines are of the same length, they are com- pared on the character basis but positions with x's. In the case of mismatch, the lines are displayed in the compare log with the "Different line" alert. If one of the files is longer than the other, the "Different file sizes" message is included at the end of the compare log. An example of the compare log is illustrated in Fig. 5.

```
/*521.300ns*/port=0;bankA=0;rowA=3;colA=3;data=01010101;mask=ff;oper=write;port=1;bankA=0;rowA=2;colA=3;data=10101010;oper=read;
/*526.300ns*/port=0;bankA=0;rowA=4;colA=0;data=xxxxxxxx;oper=read;port=1;bankA=0;rowA=4;colA=0;data=xxxxxxxx;oper=read;
/*531.300ns*/port=0;bankA=0;rowA=4;colA=0;data=10101010;mask=ff;oper=write;port=1;bankA=0;rowA=5;colA=0;data=xxxxxxxx;oper=read;
/*536.300ns*/port=0;bankA=0;rowA=4;colA=1;data=xxxxxxxx;oper=read;port=1;bankA=0;rowA=4;colA=1;data=xxxxxxxx;oper=read;
/*541.300ns*/port=0;bankA=0;rowA=4;colA=1;data=10101010;mask=ff;oper=write;port=1;bankA=0;rowA=5;colA=1;data=xxxxxxxx;oper=read;
```

Fig. 4: Example of the processed real activity log file

 

```
Different line:
Emulation    (490):port=0;bankA=0;rowA=0;colA=0;data=01010101;mask=ff;oper=write;
Simulation(490):port=0;bankA=f;rowA=0;colA=0;data=10101010;mask=ff;oper=write;
 […]
Different file sizes.
File "real_activity.log" is longer.
```

Fig. 5: Example of the compare log file

## III. FAULT COVERAGE

The next solution presented in this paper aims at determining the fault coverage of test algorithms implemented in the MBIST tool. This information can be used in various ways, for example to cross-check it with the information included in the tool's documentation. The presented methods are based on a faulty memory simulation.

Our memory fault simulator mimics a word-oriented SRAM featuring one bank. Ports for the read and write operations are separated. We assume, however, that only one operation can be executed at a time. Simulations are run at the RTL level. The tool is written in Verilog, therefore to maintain consistency between the MBIST tool and the memory fault simulator, we have decided to develop a memory model and fault models in the same language.

Our objective was to implement a memory module similar to that of a real SRAM. We took into account both the memory structure and its modus operandi. As can be seen in Fig. 6

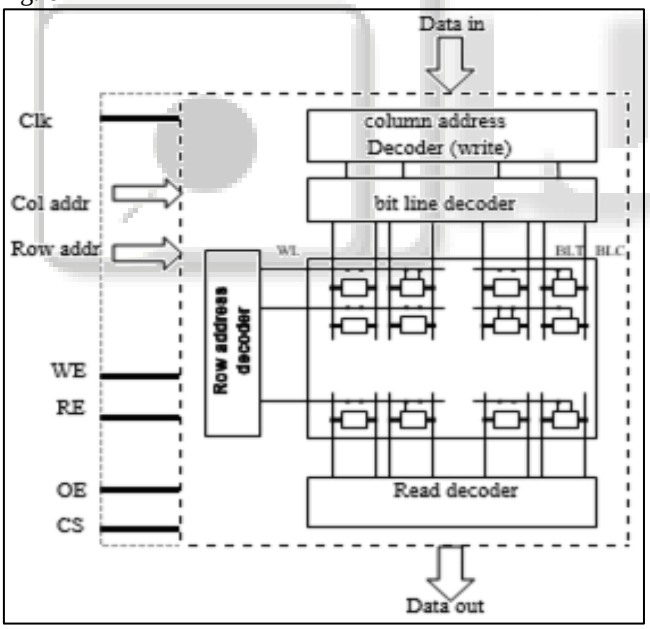


Fig. 6: Memory model

The column address decoder activates certain output line based on the column address (col addr). Accordingly, the bit line decoder assigns data to the selected pairs of bit lines (BLTs and BLCs, also referred jointly to as BLs). The row address (row addr) indicates the row activated by asserting the corresponding word line (WL). Intersections of active WL and BLs (with the data) determine where data is written. It consists of sense amplifiers and data multiplexers (not shown in Fig. 6). The sense amplifier checks whether BLT and BLC associated with the same cell have the opposite values. Based on the column address, the data multiplexers select signals from amplifiers and pass them to the memory output (data out) provided the output enable signal (OE) is active. Both the read and write operations need the asserted chip select signal (CS). Bit and word lines implementation.

The bit and word lines take part in both the read and write operations. The presence of bit and word lines within the memory model can be omitted in memory fault simulators because all standard memory operations from the functional point of view can be accomplished without these lines. On the other hand, their implementation extends significantly the range of fault models. It enables simulation of coupled bit lines or word lines, as well as various address decoder faults and other coupling faults within a memory array. It is a common issue in real circuits, so the ability to model this type of faults is a key feature.

The presence of bit and word lines within the memory model can be omitted in memory fault simulators because all standard memory operations from the functional point of view can be accomplished without these lines. On the other hand, their implementation extends significantly the range of fault models. It enables simulation of coupled bit lines or word lines, as well as various address decoder faults and other coupling faults within a memory array. It is a common issue in real circuits, so the ability to model this type of faults is a key feature.

The Verilog drivers are very useful in implementing the bit and word lines. The wire data type facilitates the model- ing of conflicts on a line. For example, when two or more cells in the same column try to put the opposite data onto bit lines, a conflict occurs. The resultant mismatch manifests itself as an unknown state. Such behavior makes fault model- ing relatively simple. Consider the bit line access transistor current leakage fault. Here, certain cells within the same column drive different values onto bit lines causing a desired mismatch on the lines.

Another advantage of our solution is its ability to model various address decoder faults. For example, in the fault-free memory only one word line can be high at a time. Never the less, our implementation of word lines enables simulation of several lines activated at the same time. The column decoder faults can be modeled in a similar fashion.

*Simulation model generator:*

Available MBIST tools are under continuous development, and new library algorithms can be introduced in the future. The ability to automatically generate faulty memory models of a given configuration is, therefore, of high importance. In particular, it can be beneficial for users checking their UDA against particular memory fault classes they want to cover. The automatic generation of simulation model significantly reduces time required to determine fault coverage for various memory sizes and other key parameters.

 

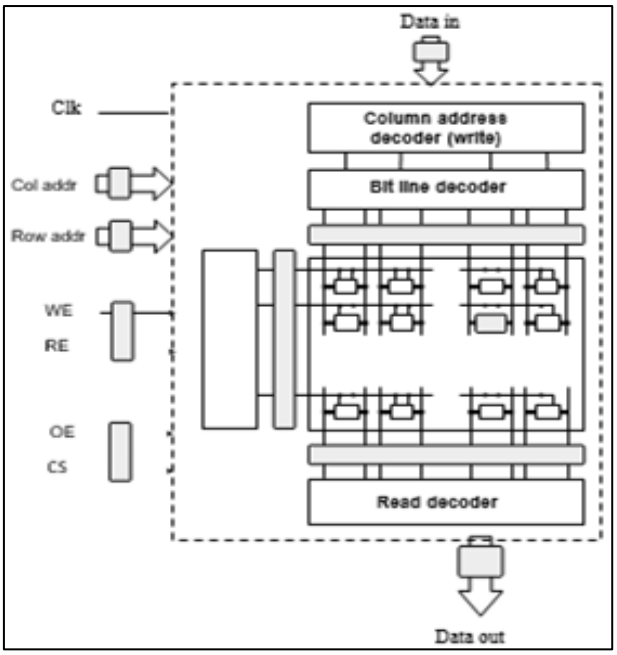


Fig. 7: Fault injection

Our simulation model architecture is kept in the input set- tings file. The proposed memory generator creates an RTL memory model and the corresponding library file based on this information. To enable fault injection with no changes to the memory model, additional modules are deployed, as shown in Fig. 7. These items are transparent for a fault-free memory, i.e., the unchanged input signals pass to the out- puts. For a faulty memory, the appropriate module is re- placed with the corresponding failing counterpart. Which module is to be replaced depends on the fault model. Injec- tion of faults affecting the memory cell relies on replacing the cell module on the faulty cell location(s).

## IV. Validation of the Tool

As the presented solutions are meant for MBIST tools, the best way to verify their capabilities and to examine their features is an application in a real DFT tool. We have ap- plied our techniques in the Tessent Memory BIST (TMB) line of products distributed by Mentor Graphics Corporation. While the described methods target different MBIST components, they all aim at maintaining high quality of a product in its entirety. Although they are tested within the same TMB framework, they need distinct simulation approaches. This section presents testing techniques for both the TMB Validator and the memory fault simulator.

### *A. Controller emulation:*

The TMB Validator emulates a controller operating on the fault-free memory. The modules are implemented using design patterns. In order to check the emulator's ability to detect any kind of TMB controller malfunctions, some mod- ifications in the tested controller are needed. These adjustments rely on the test targets. Each of them must be performed in a properly setup environment.

Simulation parameters depend on the tested tool variables. Each of them is set by a different software part, and thus requires its own preparations to the test. To perform exhaustive tests on the MBIST controller, various memory types are required. Fortunately, the automatic generation of the memory models and the corresponding library files facilitates a complete test of the controller.

Typically, MBIST tools undergo continuous yet error prone maintenance programs. Consequently, another objective of our solution is to provide the way to monitor the MBIST tools application. In particular, the TMB Validator is used in the regression tests so any malfunctions can be quickly identified.

### *B. Fault coverage:*

As can be easily observed, a single test does not suffice to determine the fault coverage of a given algorithm. This is because of too few fault locations examined. In fact, it may not be feasible to implement all fault configurations, which should be tackled by the test algorithm. The same problem is even more demanding for large memory arrays consisting of thousands of cells. Another drawback of a single simulation experiment run serially is its overall CPU time. To alleviate these problems, our test experiments were con- ducted in parallel by employing a grid computing.

The script workflow for the single simulation is shown in Fig. 8. The settings file and the faults file determine the memory architecture and type(s) of injected fault(s), respec- tively. The faults file is obtained by means of a fault loca- tion generator. The input variables, such as the number of rows, columns, and the size of a word, are essential to de- termine the boundaries of injected fault sites. The fault name and its parameters decide which function generating a fault location is invoked. Once all data are known, the generation of the fault location is performed and the result is passed to the faults file. When the simulation model and the library file are generated (based on the type of algorithm included in the library file), the faulty memory simulation is carried out, and the fault coverage numbers are stored in the output result file.

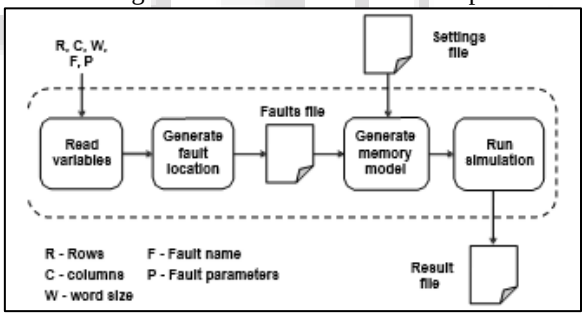


Fig. 8: Workflow for the single simulation

Given a single fault type, a test experiment comprises a number of simulations conducted for a particular test algorithm and a memory model with faults injected at random locations. These locations allow one to take care of peculiar scenarios where faults are not detectable by test algorithms. For instance, faults located in the first row, in a corner cell, and so on, are frequent test escapes due to lack of a respective detecting step in an algorithm implementation.

## V. Results

### *A. Controller emulation:*

The following variables were examined during the presented experiments.

- Test algorithms
- Single port versus multi-port
- Address segmentation

 

- Bit grouping
- Serial interface
- Shadow read/write operations
- Memory control inputs verification
- Interconnections in data path or address path

### *B. Fault coverage:*

For the series of experiments presented in this section, we have established a memory array consisting of 32 rows and 8 columns of 8-bit words in both interleaved and non-interleaved arrangements. The number of simulation runs for each type of fault depends on the number of possible fault configurations. Except for a few fault models (e.g., address decoder faults), a statistically significant number of 2,000 simulations were carried out for every combination of fault class and test algorithm.

As a result of the experiments, a few special cases were identified. It turns out that there are fault sites not covered by the test algorithms because of their implementation.

## VI. CONCLUSION

In this paper, we present two techniques for improve the quality of MBIST tools. The first solution tackles validation of the MBIST controller. In order to monitor the controller, its emulator was developed. The experiments show the ability of the TMB Validator to verify various controller features. The second method introduces the novel memory fault simulator. Tests performed with this tool demonstrate its versatility to determine reliably the test coverage when working with a variety of memory fault models. The experimental results confirm flexibility and efficacy of the described techniques. These two methods offer flexible ways to maintain high quality of MBIST products.

## REFERENCES

[1] Tessent® MemoryBIST Usage Guide and Reference, release 2013.1, Mentor Graphics Corp.
[2] K. Amirkhanyan et al., “Application of defect injection flow for fault validation in memories”, Proc. EWDTS, 2012, pp. 19-22.
[3] C.-F. Wu, C.-T. Huang, K.-L. Cheng and C.-W. Wu, “Fault simula- tion and test algorithm generation for random access memories”, IEEE Trans. CAD, vol. 21, pp. 480-490, Apr. 2002.
[4] R. D. Adams, High performance memory testing: design principles, fault modeling and self-test, Kluwer Academic Publ., 2003.
[5] M. Azimane and A. K. Majhi, “New test methodology for resistive open defect detection in memory address decoders”, Proc. VTS, 2004, pp. 123-128.
[6] A. Benso, S. Di Carlo, G. Di Natale and P. Prinetto, “Specification and design of a new memory fault simulator”, Proc. ATS, 2002, pp. 92-97.
[7] A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch and A. Virazel, Advanced test methods for SRAMs: effective solutions for dynamic fault detection in nanoscaled technologies, Springer, 2010.
[8] A. J. van de Goor, Testing semiconductor memories: theory and practice, ComTex Publishing, 1998.
[9] H.-H. Wu, J.-F. Li, C.-F. Wu, and C.-W. Wu, “CAMEL: An efficient fault simulator with coupling fault simulation enhancement for CAMs”, Proc. ATS, 2007, pp. 355-360.
[10] C.-F. Wu, C.-T. Huang, K.-L. Cheng, and C.-W. Wu, “Simulation- based test algorithm generation for random access memories”, Proc. VTS, 2000, pp. 291-296.

 