International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 11, No. 2, July 2022, pp. 103~114
ISSN: 2089-4864, DOI: 10.11591/ijres.v11.i2.pp103-114

Automatic generation of user-defined test algorithm description file for memory BIST implementation

Aiman Zakwan Jidin[1,2], Razaidi Hussin[1], Lee Weng Fook[3], Mohd Syafiq Mispan[2], Loh Wan Ying[1]
[1]Faculty of Electronics Engineering Technology, Universiti Malaysia Perlis, Perlis, Malaysia
[2]Faculty of Electrical and Electronics Engineering Technology, Universiti Teknikal Malaysia Melaka, Melaka, Malaysia
[3]Emerald System Design Center, Penang, Malaysia

Article Info

Article history:

Received Jan 13, 2022
Revised Feb 15, 2022
Accepted Mar 11, 2022

Keywords:

Automation software
March algorithm
Memory BIST
Memory fault models
User-defined algorithm

ABSTRACT

Memory built-in self-test (BIST) is a widely used technique to allow the self-test and self-checking of the embedded memories on chips after the fabrication process. It can be used by implementing a standard testing algorithm available in the EDA tool library or a user-defined algorithm (UDA). This paper presents the development of software that automatically generates a description file of a UDA to be deployed for memory BIST circuit implementation using Tessent memory BIST software. It comprises the test setup and also the microprogram coding for each instruction to be executed when performing tests on embedded memories. The proposed automation software was tested by using March SR as the input algorithm and the results obtained from the simulations show that the output test patterns generated by the implemented memory BIST match the expected patterns and passed all the tests, which validated the correct functionality of the UDA description file generation. The proposed automation software also fast generation the UDA description file, which was completed in less than 500 ms.

Corresponding Author:

Razaidi Hussin
Faculty of Electronics Engineering Technology, Universiti Malaysia Perlis
06100 Arau, Perlis, Malaysia
Email: shidee@unimap.edu.my

1. INTRODUCTION

The process to test the embedded memories on a chip is becoming more challenging nowadays, since they are becoming more compact and more defects which may randomly happen since the introduction of the very deep submicron (VDSM) technologies [1]–[6]. Furthermore, it becomes more important than ever since the chips are now memory-dominant, where some studies show that up to 90% of the chip area is occupied by the memories [7]–[10]. Memory built-in self-test (BIST) is a technique that is very widely used for embedded memory testing. It offers several advantages such as the ability to perform self-test and self-check of the output responses without the use of an expensive external tester, and the ability to perform tests on multiple memories in parallel, which allow the reduction in overall test cost and test duration, respectively [8], [11]–[15]. Its efficiency in terms of the fault coverage and also the test duration depends on the test algorithm being used for its implementation [16].

A memory BIST can be implemented by using an electronic design automation (EDA) tool like Mentor Graphics Tessent software. It can be implemented by using either a standard test algorithm available in the EDA library or by using a user-defined algorithm (UDA) [17]. A UDA is an algorithm which is customized for a specific target, either to have a low test length or to have an optimized detection on a

specific set of faults. To use a UDA when implementing a memory BIST circuit, a description file is necessary to define a custom algorithm and to describe its behavior such as the test setup and the microprogram coding of each instruction to be executed during memory testing. In addition, a UDA can be either hard-coded or soft-coded in the memory BIST implementation. While the former offers design simplicity, the latter offers more flexibility where test algorithms can be changed during program execution [18].

This paper presents the development of automation software which generates a description file of an input UDA to be hard-coded for memory BIST implementation in Tessent memory BIST software, to reduce human effort in obtaining a correct description file of a UDA in a very brief delay. This was achieved by automatically extracting test operation sequences of the UDA and mapping the test sequences of each test element of the UDA to the corresponding operation name and the values of the related parameters to be written into the UDA description file.

Section 2 describes the test algorithm's test operation sequences. Then, Section 3 describes the contents of the UDA description file which is utilizable in the Tessent memory BIST software. Section 4 discusses the process flow of the proposed automation software. Finally, Section 5 observed and analyzed the outputs of the simulation performed on the implemented memory BIST circuit using the generated UDA description file. This paper is focusing only on the March-series test algorithm, with a test complexity lesser than 22*N*, where *N* is the size of the memory. March SR algorithm, with 14*N* test complexity, is used for elaboration and demonstration purposes since it consists of different test elements with different test sequences, test lengths, and test address directions, which is useful for testing the proposed automation software.

2. MARCH ALGORITHM AS UDA FOR MEMORY BIST IMPLEMENTATION

2.1. March algorithm description

Table 1 describes the symbol used in the test algorithm test operation sequence notation [3], [19]–[23]. In general, a March algorithm consists of *m* test elements, each of them separated by a semicolon. A test element consists of a set of test operations to be carried out on each cell, starting from the minimum address until the maximum address (in the case of ascending address order) or vice-versa (in the case of descending address direction), before proceeding to the next test element. The test operation can be either a read (r) or a write (w) operation, using only two possible test values (logic 0 or logic 1) called the data backgrounds [24], [25].

Table 1. The description of the symbols used in the memory testing algorithm notation

Symbol	Description
↑ or ⇑	address sequence changes in ascending order
↓ or ⇓	address sequence changes in descending order
↕ or ⇕	address sequence can change either way
R0 or r0	read operation (reading a 0 from a cell)
R1 or r1	read operation (reading a 1 from a cell)
W0 or w0	write operation (writing a 0 to a cell)
W1 or w1	write operation (writing a 1 to a cell)
;	test element separator

A March algorithm consists of multiple test elements, each of which is separated by a semicolon. Each test element will be executed sequentially, starting from the first test element until the final test element. In the example of March SR algorithm with the test operation sequences ⇑(w0); ⇑(r0, w1, r1, w0); ⇑(r0, r0); ⇑(w1); ⇓(r1, w0, r0, w1); ⇓(r1, r1) [26]. It consists of 6 test elements, notated as M(*i*) where *i* = {0, 1, 2, 3, 4, 5}. As can be seen, M(0) to M(3) have the ascending address order, where the test operations will be executed starting from the memory cell with the minimum address. While M(4) and M(5) have the descending address order, where the test operations will be executed starting from the memory cell with the maximum address. Since it has in total of 14 test operations, the test complexity of the March SR algorithm is 14*N*.

In the March SR algorithm, all cells will be initialized to 0 first in ascending address order during M(0). Then, in M(1), each cell will be read (expecting 0), written to 1, read (expecting 1), and rewritten to 0 starting from the cell with the minimum address. Next, each cell, starting from the cell with the minimum address, will be read twice (both expecting 0) in M(2). In M(3), all cells will be written to 1 in the ascending address order. After that, each cell, starting from the cell with the maximum address, will be read (expecting

1), written to 0, read (expecting 0), and rewritten to 1 in M(4). Finally, in M(5), each cell, starting from the cell with the minimum address, will be read twice (both expecting 1).

2.2. UDA Tessent Core Description (TCD) file

The TCD file is the configuration data syntax that is used to describe the modules in the Mentor Graphics Tessent software such as the memories, boundary scan segments, and fusebox. For memory BIST implementation purposes, a TCD file is necessary to specify the behavior of the memory e.g. the module name, the number of words, and the memory type (ROM, SRAM, or DRAM). Furthermore, in the case of the memory BIST implementation with UDA, an additional TCD file is needed to describe the memory test algorithm which will be hard-coded into the BIST controller. The TCD files for memory BIST are recognized with the *.tcd_mem_lib* extension [17].

For UDA description, the TCD file contains the test setup and the microprogram coding of each instruction to be executed during memory testing. The test setup consists of information such as the name of the UDA, the minimum and the maximum row and column addresses, and the selection of the test operation set. While the microprogram coding describes the test operation sequences of each test element separately, in terms of the address order (increment or decrement), the write data value, the expected read data value, and the operation name which are predefined in the operation set library specified in the test setup. The microprogram coding for each test element is written in the template:

```
Instruction (M<i>_<test operation sequences>){
        OperationSelect: <Operation name>;
        X1AddressCmd: <Address Order>;
        Y1AddressCmd: <Address Order>;
        ExpectDataCmd: <Expect data>;
        WriteDataCmd: <Write data>;
        NextConditions {
                //insert conditions
        }
}
```

For example, the coding for M(2) of March SR algorithm (⇑(r0, r0)) is written as:

```
Instruction (M2_r0r0){
        OperationSelect: ReadRead
        X1AddressCmd: Increment;
        Y1AddressCmd: Increment;
        ExpectDataCmd: DataReg;
        NextConditions {
                X1_EndCount : on;
                Y1_EndCount : on;
        }
}
```

In the case where a test element consists of more than 3 test operations, a special `BranchToInstruction` command will be added to the coding, so that it can be coded in two linked instructions. For example, M(1) of March SR algorithm (⇑(r0, w1, r1, w0)) is written as two linked instructions *M1_r0w1* and *M1_r1w0*, as:

```
Instruction (M1_r0w1){
        OperationSelect: ReadModifyWrite;
        ExpectDataCmd: DataReg;
        WriteDataCmd: InverseDataReg;
        NextConditions {
        }
}

Instruction (M1_r1w0){
        OperationSelect: ReadModifyWrite;
        X1AddressCmd : Increment;
        Y1AddressCmd : Increment;
        ExpectDataCmd: InverseDataReg;
        WriteDataCmd: DataReg;
        BranchToInstruction : M1_r0w1;
        NextConditions {
                X1_EndCount : on;
                Y1_EndCount : on;
        }
}
```

In addition, a specific instruction `InhibitLastAddressCount` needs to be added and set its value to `on` in the case where a transition between two test elements involves a change in the address direction, like in the case of the transition between M(3) (increase address order) and M(4) (decrease address order). This is necessary to prevent the address counter from wrapping around to the minimum address when the maximum address is reached at the end of M(3). Therefore, at the start of M(4), it will start to count down from the maximum address. Thus, the microprogram coding of M(3) is written as:

```
Instruction (M3_w1){
        OperationSelect : WriteWriteFastRow;
        X1AddressCmd : Increment;
        Y1AddressCmd : Increment;
        WriteDataCmd : InverseDataReg;
        InhibitLastAddressCount : on;
        NextConditions {
                X1_EndCount : on;
                Y1_EndCount : on;
        }
}
```

The description file is unique for each UDA since different algorithms are composed of different test operation sequences. It is then read by the memory BIST insertion tools, which will extract its test operation sequences based on the operations, the address directions, the values to be written into the memory cells, and the expected values to be read from the memory. During the memory BIST implementation process, this microprogram coding is converted into a memory BIST controller hardware, which is coded in Verilog HDL.

3. RESEARCH METHODOLOGY

Figure 1 shows the overall process flow of the proposed automation software, which is developed using the C++ programming language. Upon executing the software, the UDA is read from an input file and essential information is extracted from it, e.g. the test operation sequences and the number of test elements *m*. The input file reading process is done by using the functions available in the file streaming *fstream* library in C++. From here, *m* data structures are created, each of which is dedicated to store the information of each test element: the address order *ao*, the test operations *rw* which stores *r* or *w* for read or write operation, respectively, and the data background *db* associated to each test operation. In the case of the March SR algorithm, 6 data structures are created to store the *ao*, the *rw*, and the *db* of each of its test elements, as described in Table 2.

Table 2. The breakdowns of March SR algorithm test sequences into separated test elements

Test element	Address order *ao*	Test operations *rw*	Data backgrounds *db*
M(0)	⇑	w	0
M(1)	⇑	r, w, r, w	0, 1, 1, 0
M(2)	⇑	r, r	0, 0
M(3)	⇑	w	1
M(4)	⇓	r, w, r, w	1, 0, 0, 1
M(5)	⇓	r, r	1, 1

Next, it opens or creates a new UDA TCD file as the output, which is saved with the *.tcd_mem_lib* extension recognized by the Tessent memory BIST tools. Immediately after that, the name of the UDA and the test setup such as the starting address and the maximum memory address will be defined in the output file. Then, the automation software determines the operation name and the values of write data, and the expected read data of the test element to be written in the TCD file, by using the mapping provided in Table 3. The write data and the expected read data only have two possible values: *DataReg* (logic 0) and *InverseDataReg* (logic 1). While the test operations are mapped to the operation names that are available under the *TessentSyncRamOps* operation set library [17].

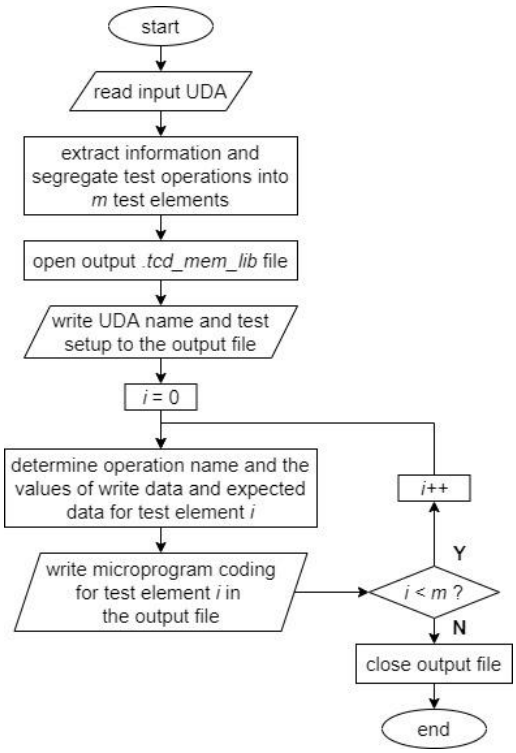

Figure 1. The process flowchart of the UDA description file generation

Table 3. The mapping of the extracted UDA to the parameters in the TCD file

Extracted data from input UDA		Data to be written into TCD file		
Test operations *rw*	Data backgrounds *db*	Operation name	Write data	Expected data
w	0	WriteWriteFastRow	DataReg	-
	1		InverseDataReg	-
r	0	ReadReadFastRow	-	DataReg
	1		-	InverseDataReg
rw	00	ReadModifyWrite	DataReg	DataReg
	01		InverseDataReg	DataReg
	10		DataReg	InverseDataReg
	11		InverseDataReg	InverseDataReg
rr	00	ReadRead	-	DataReg
	11		-	InverseDataReg
wr	00	WriteRead	DataReg	DataReg
	11		InverseDataReg	InverseDataReg
rwr	000	ReadWriteRead	DataReg	DataReg
	111		InverseDataReg	InverseDataReg
	011	ReadWriteReadInvert	InverseDataReg	DataReg
	100		DataReg	InverseDataReg

After determining these parameters, the microprogram coding of the test element is written to the output TCD file, by following the template previously discussed in Section 3. These processes are repeated for all test elements of the UDA. The process flow of determining the operation name, the value of the write data, and the expected read data for each test element is detailed in Figure 2. The provided flowchart also shows that if the address order of the current test element *ao*(*i*) is different from the one of the next test element *ao*(*i*+1), the value of *InhibitLastAddressCount* is set to *on*.

Once all the test elements have been coded and written to the TCD file, the output file is closed and the software execution ends. The generated TCD file is then copied into the Tessent memory BIST working directory. It will be read by the tools to be used as the algorithm for memory BIST implementation.

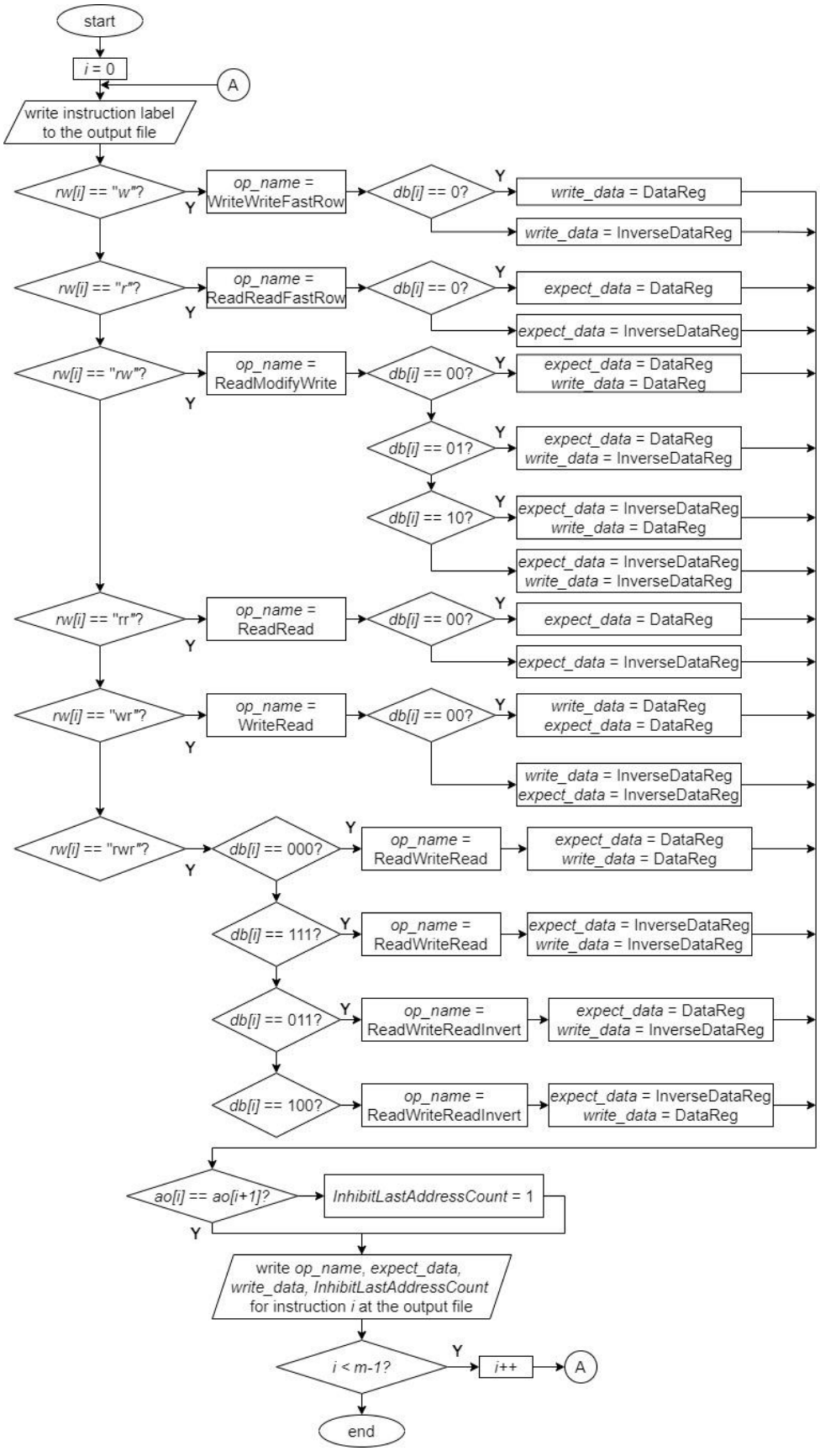

Figure 2. The process flowchart of determining the operation name and the values of write data and expected data of each test element

4. RESULTS AND DISCUSSION

To validate the functionality of the proposed automation software, firstly the test operation sequences of the March SR algorithm were stored in an input text file named *march_SR.txt*, to be used as the input UDA. The proposed automation software was executed by reading the input file, extracting the information, and producing the microprogram coding of the UDA in the output description file *march_SR.tcd_mem_lib*, as shown in Figure 3. It shows that both M(1) and M(4), which consist of 4 test operations each, require the branching command `BranchToInstruction` to link up two instructions together. In addition, the `InhibitLastAddressCount` command is also added and set to *on* inside the instruction M3_w1. By using the `gettimeofday()` function in the C++ programming language, the automation execution completion time to generate the UDA TCD file was measured in multiple attempts, which took less than 500 ms on a PC with a 2.40 GHz microprocessor and 8GB of RAM. However, no comparison of the completion time is to be made since no previous similar works were published.

```
Algorithm (march_SR) {
  TestRegisterSetup {
    OperationSetSelect : TessentSyncRamOps;
    AddressGenerator {
      AddressRegister (A) {
        LoadColumnAddress: MinColumn;
        LoadRowAddress: MinRow;
        X1CarryIn: None;
        Y1CarryIn: X1CarryOut;
      }
    }
    DataGenerator {
      LoadWriteData : 8'b00000000;
      LoadExpectData : 8'b00000000;
    }
  }
  MicroProgram {
    Instruction (M0_w0){
       OperationSelect : WriteWriteFastRow;
       X1AddressCmd : Increment;
       Y1AddressCmd : Increment;
       WriteDataCmd : DataReg;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
    Instruction (M1_r0w1){
       OperationSelect : ReadModifyWrite;
       ExpectDataCmd : DataReg;
       WriteDataCmd : InverseDataReg;
       NextConditions {
       }
     }
    Instruction (M1_r1w0){
       OperationSelect : ReadModifyWrite;
       X1AddressCmd : Increment;
       Y1AddressCmd : Increment;
       ExpectDataCmd : InverseDataReg;
       WriteDataCmd : DataReg;
       BranchToInstruction : M1_r0w1;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
    Instruction (M2_r0r0){
       OperationSelect : ReadRead;
       X1AddressCmd : Increment;
       Y1AddressCmd : Increment;
       ExpectDataCmd : DataReg;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
    Instruction (M3_w1){
       OperationSelect : WriteWriteFastRow;
       X1AddressCmd : Increment;
       Y1AddressCmd : Increment;
       WriteDataCmd : InverseDataReg;
       InhibitLastAddressCount : on;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
    Instruction (M4_r1w0){
       OperationSelect : ReadModifyWrite;
       ExpectDataCmd : InverseDataReg;
       WriteDataCmd : DataReg;
       NextConditions {
       }
     }
    Instruction (M4_r0w1){
       OperationSelect : ReadModifyWrite;
       X1AddressCmd : Decrement;
       Y1AddressCmd : Decrement;
       ExpectDataCmd : DataReg;
       WriteDataCmd : InverseDataReg;
       BranchToInstruction : M4_r1w0;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
    Instruction (M5_r1r1){
       OperationSelect : ReadRead;
       X1AddressCmd : Decrement;
       Y1AddressCmd : Decrement;
       ExpectDataCmd : InverseDataReg;
       NextConditions {
         X1_EndCount : on;
         Y1_EndCount : on;
       }
    }
  }
```

Figure 3. The generated description file *march_SR.tcd_mem_lib*

The generated TCD file was then read by the Tessent memory BIST tools, and the March SR algorithm was applied as the UDA for the memory BIST implementation. For this purpose, a simple arithmetic logic unit (ALU) which contains a 70-word SRAM as the memory instance is used. Figure 4 shows the schematic view of the generated memory BIST circuit. 5 additional modules are added and connected to the memory instances:

- **tessent_mbist_controller**, which generates the test addresses and the test inputs according to the UDA test sequences which are hard-coded inside this module
- **tessent_memory_interface**, which acts as the interface between the memory BIST controller and the memory instance
- **tessent_mbist_bap**, which is the BIST access port to configure the memory BIST controller and to monitor test pass/fail status
- two **tessent_sib instances**, the segment insertion bit blocks which act as the switches to include or to exclude the memory BIST from the IJTAG network on the chip

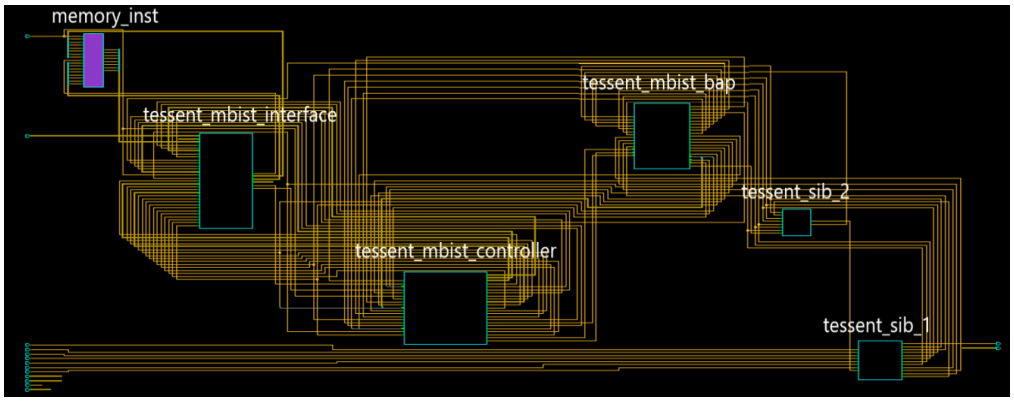

Figure 4. The generated memory BIST circuit

Once implemented, the memory BIST circuit is simulated in the QuestaSim simulator, by using the test patterns which are generated during its implementation process. Figure 5 shows the overall waveform of the simulation performed on the implemented memory BIST. It can be observed that the ERROR flag stays low throughout the simulation, which indicates that there is no mismatch occurring between the observed read outputs (*dout*) and the expected outputs. While the CMP_EN signal is toggling and is high whenever a comparison between the output read value and the expected value is necessary.

Besides, the simulation also shows that the overall test took 19.6 us to be completed, where the clock period used for this simulation is 20 ns. From here, the test complexity of the UDA can be derived using (1).

$$Test\ Complexity = \frac{Test\ Duration}{T_{clock} * N} \quad (1)$$

In this case, N = 70 which is the size of the memory model used for the test. Hence, the test complexity of the UDA used for this implementation is equal to 14, which is equal to the expected complexity of the March SR algorithm (14N) [26].

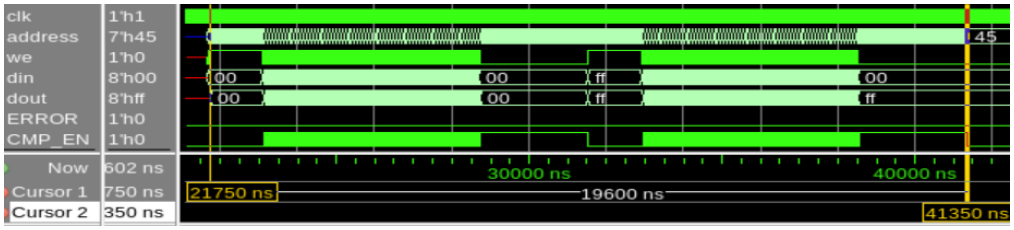

Figure 5. The overall waveform of the simulation performed on the implemented memory BIST

Figure 6 to Figure 11 shows the waveform of the simulation, representing the test patterns of each test element. The waveform in Figure 6 corresponds to the test operation of M(0): ⇑(w0), where all memory cells are written to 0, starting from address 0 to address 69 (or 45h in hexadecimal). No comparison is needed at this stage since it is a write-only operation (indicated by CMP_EN = 0). In Figure 7, the waveform demonstrates that each cell is read first (expecting 0 at the output), written to logic 1, reread (expecting 1 at the output), and finally written back to logic 0. These processes are executed in ascending address order. This translates the test operation sequences of M(1): (⇑(r0, w1, r1, w0). The patterns shown in Figure 8 correspond to the M(2): ⇑(r0, r0), where each cell is read twice in the ascending address order and both read operations are expecting logic 0 at the output.

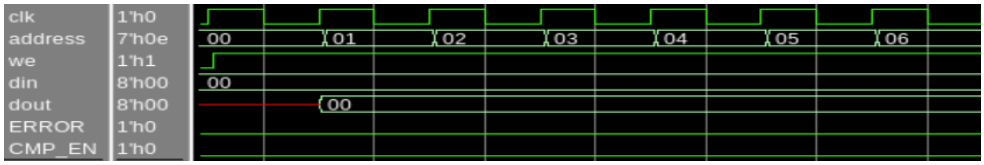

Figure 6. The simulation waveform represents the patterns of M(0)

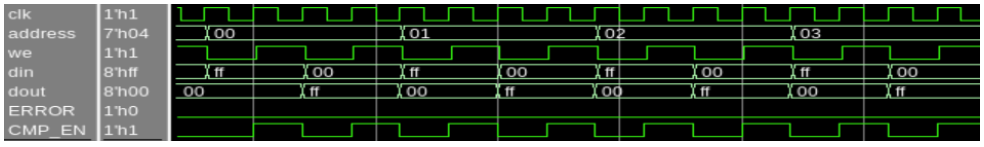

Figure 7. The simulation waveform represents the patterns of M(1)

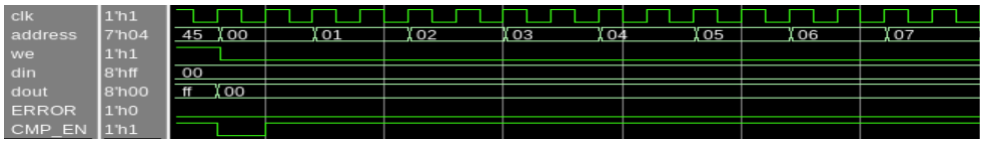

Figure 8. The simulation waveform represents the patterns of M(2)

While Figure 9 shows the patterns executed by M(3): ⇑(w1), where it has almost the same pattern as test element 0, but logic 1 is written to the cells instead of logic 0. Next, the waveform in Figure 10 corresponds to the patterns of M(4): ⇓(r1, w0, r0, w1), where each cell is read (expecting 1 at the output), written to logic 0, reread (expecting 0 at the output), and finally written back to logic 1, in the descending address direction. Finally, the patterns of M(5): ⇓(r1, r1) are translated by the waveform in Figure 11, where each cell is read twice (expecting logic 1 at the output) in the descending address order.

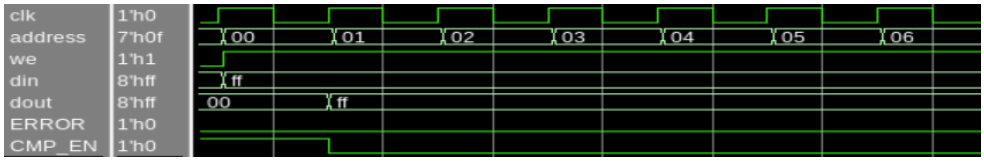

Figure 9. The simulation waveform represents the patterns of M(3)

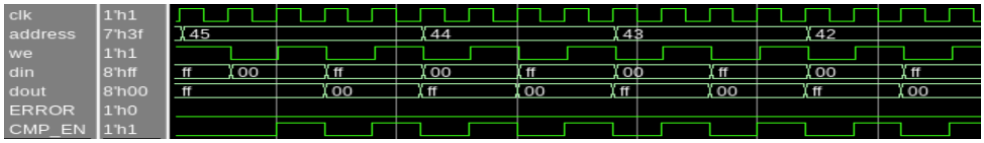

Figure 10. The simulation waveform represents the patterns of M(4)

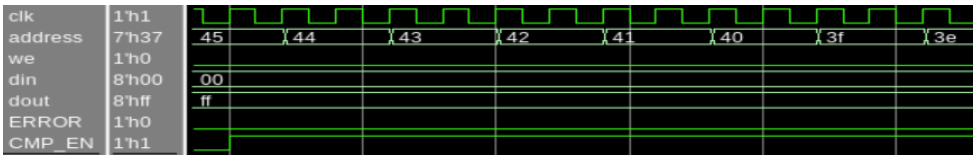

Figure 11. The simulation waveform represents the patterns of M(5)

Thus, the observed simulation waveforms met the expectations, where the observed test patterns correspond to the test sequences of the March SR algorithm and no mismatch occurred between the output values read from the memory and the expected output value, and they proved that the memory BIST circuit has been successfully implemented by using the UDA description file generated by the proposed automation software.

For future planning, the mapping provided in Table 3 will be improved by adding more possible combinations of test operations such as *wwr* or *rww* which may exist in test algorithms with higher test complexity than 22N, to ensure that it can work on as many algorithms as possible. Besides, the UDA description generation algorithm will be improved to allow the optimization of the UDA microprogram coding e.g. to reduce the line number or instructions by using the repetition technique. Furthermore, it will also be tested using various March algorithms with different test sequences and complexities to guarantee its accuracy and reliability.

5. CONCLUSION

This research paper has presented the development of automation software to automatically generate a UDA description file to be used for the memory BIST implementation in Tessent memory BIST software. The proposed automation software was developed by using the C++ programming language and consists of the reading and extracting information from the input UDA, segregation of test operation sequences into separated test elements, determination of operation names, write data values, and expected read values to be written into the output TCD file. The generated file is then read by Tessent memory BIST tools during the implementation process, and the simulation was performed on the implemented memory BIST circuit, which produced correct test patterns as per expectation and correspond to the intended test operation sequences. The proposed automation software allows the generation of the required UDA description file automatically with a completion time lesser than 500 ms, thus, reducing human effort and time in obtaining a working description file.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the Faculty of Electronic Engineering Technology, Universiti Malaysia Perlis (UniMAP), Universiti Teknikal Malaysia Melaka (UTeM), and the Ministry of Higher Education Malaysia, for their contribution, support to this research, and financial assistance under the SLAB scheme.

REFERENCES

[1] A. Bosio, S. Di Carlo, G. Di Natale, and P. Prinetto, "March AB, a state-of-the-art march test for realistic static linked faults and dynamic faults in SRAMs," *IET Computers and Digital Techniques*, vol. 1, no. 3, pp. 237–245, 2007, doi: 10.1049/iet-cdt:20060137.

[2] G. Harutyunyan, S. Shoukourian, and Y. Zorian, "Fault awareness for memory BIST architecture shaped by multidimensional prediction mechanism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 3, pp. 562–575, 2019, doi: 10.1109/TCAD.2018.2818688.

[3] A. Kumar, "Assertion based functional verification of March algorithm based MBIST controller," Indian Institute of Information Technology Allahabad, 2021.

[4] J. Kinseher, M. Richter, and I. Polian, "On the automated verification of user-defined MBIST algorithms," in *uE 2015; 8. GMM/ITG/GI-Symposium Reliability by Design*, 2015, pp. 50–55.

[5] P. E. Joseph and P. R. Antony, "VLSI design and comparative analysis of memory BIST controllers," in *2014 First International Conference on Computational Systems and Communications (ICCSC)*, 2003, pp. 272–276, doi: 10.1109/COMPSC.2014.7032661.

[6] M. Parvathi, N. Vasantha, and Ks. Parasad, "Modified March C - algorithm for embedded memory testing," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 2, no. 5, p. 571, Oct. 2012, doi: 10.11591/ijece.v2i5.1587.

[7] P. Ramakrishna, T. Vamshika, and M. Swathi, "FPGA implementation of memory BISTS using single interface," *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 9, no. 3, pp. 55–58, Sep. 2020, doi: 10.35940/ijrte.B3975.099320.

[8] T. S. N. Kong *et al.*, "An Efficient March (5n) FSM-Based Memory Built-In Self Test (MBIST) Architecture," in *2021 IEEE Regional Symposium on Micro and Nanoelectronics (RSM)*, Aug. 2021, pp. 76–79, doi: 10.1109/RSM52397.2021.9511602.

[9] P. S. N. Bhaskar, B. Sarada, and S. Kandregula, "Built-in self-repair techniques of embedded memories with BIST for improving reliability," *IOSR Journal of Computer Engineering*, vol. 21, no. 1, pp. 8–15, 2019, doi: 10.9790/0661-2101010815.
[10] V. S. Chakravarthi, *A practical approach to VLSI system on chip (SoC) design*. Cham: Springer International Publishing, 2020.
[11] A. A. Wojciechowski, K. Marcinek, and W. A. Pleskacz, "Configurable MBIST processor for embedded memories testing," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems,"* Jun. 2019, pp. 341–344, doi: 10.23919/MIXDES.2019.8787161.
[12] S. N. Bagewadi, S. Shadab, and J. Roopa, "Fast BIST mechanism for faster validation of memory array," in *2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, May 2019, pp. 61–65, doi: 10.1109/RTEICT46194.2019.9016882.
[13] R. Manasa, R. Verma, and D. Koppad, "Implementation of BIST technology using March-LR algorithm," in *2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, May 2019, pp. 1208–1212, doi: 10.1109/RTEICT46194.2019.9016784.
[14] A. K. S. Pundir, "Novel modified memory built in self-repair (MMBISR) for SRAM using hybrid redundancy-analysis technique," *IET Circuits, Devices & Systems*, vol. 13, no. 6, pp. 836–842, Sep. 2019, doi: 10.1049/iet-cds.2018.5218.
[15] N. I. Singh and P. V Joshi, "Performance analysis of BIST algorithms," *Indian Journal of Science and Technology*, vol. 12, no. 36, pp. 1–3, Sep. 2019, doi: 10.17485/ijst/2019/v12i36/147752.
[16] A. Z. Jidin, R. Hussin, L. W. Fook, and M. S. Mispan, "A review paper on memory fault models and test algorithms," *Bulletin of Electrical Engineering and Informatics*, vol. 10, no. 6, pp. 3083–3093, Dec. 2021, doi: 10.11591/eei.v10i6.3048.
[17] Siemens, "Tessent MemoryBIST User's manual for use with Tessent Shell." Siemens, 2020, [Online]. Available: https://eda.sw.siemens.com/en-US/ic/tessent/test/memorybist/.
[18] Y. Lu, "BIST implementation access through a reconfigurable network," Lund University, 2019.
[19] A. Singh, G. M. Kumar, and A. Aasti, "Controller architecture for memory BIST algorithms," in *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, 2020, pp. 1–5, doi: 10.1109/SCEECS48394.2020.43.
[20] S. B. Ghale and N. P, "Design and implementation of memory BIST for hybrid cache architecture," in *2021 6th International Conference on Communication and Electronics Systems (ICCES)*, Jul. 2021, pp. 26–31, doi: 10.1109/ICCES51350.2021.9489225.
[21] S. R. Patil and D. B. Musle, "Implementation of BIST technology for fault detection and repair of the multiported memory using FPGA," in *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, 2017, pp. 43–47, doi: 10.1109/ICECA.2017.8212849.
[22] N. A. Zakaria, W. Z. W. Hasan, I. A. Halin, R. M. Sidek, and X. Wen, "Fault detection with optimum March test algorithm," in *2012 Third International Conference on Intelligent Systems Modelling and Simulation*, 2012, pp. 700–704, doi: 10.1109/ISMS.2012.88.
[23] S. Hamdioui, R. Wadsworth, J. D. Reyes, and A. J. van de Goor, "Memory fault modeling trends: A case study," *Journal of Electronic Testing*, vol. 20, no. 3, pp. 245–255, Jun. 2004, doi: 10.1023/B:JETT.0000029458.57095.bb.
[24] N. A. Zakaria, "Multiple and solid data background scheme for testing static single cell faults on SRAM memories," Universiti Putra Malaysia, 2013.
[25] A. J. van de Goor, S. Hamdioui, and H. Kukner, "Generic, orthogonal and low-cost March element based memory BIST," in *2011 IEEE International Test Conference*, Sep. 2011, pp. 1–10, doi: 10.1109/TEST.2011.6139148.
[26] S. Hamdioui and A. J. Van De Goor, "An experimental analysis of spot defects in SRAMs: realistic fault models and tests," in *Proceedings of the Ninth Asian Test Symposium*, 2000, pp. 131–138, doi: 10.1109/ATS.2000.893615.

BIOGRAPHIES OF AUTHORS

Aiman Zakwan Jidin is currently a Ph.D. candidate at Universiti Malaysia Perlis, Malaysia. His research topic is focusing on optimizing memory testing algorithm efficiency for improving fault coverage. Previously, he obtained his MEng in Electronic and Microelectronic System from ESIEE Engineering Paris, France in 2011, before working as FPGA IP Core Design Engineer at Altera Corporation Malaysia (now part of Intel). He is a full-time lecturer and researcher at Universiti Teknikal Malaysia Melaka (UTeM), in Electronic and Computer Engineering. His research interests include DFT, VLSI, and FPGA system design. He can be contacted at email: aimanzakwan@utem.edu.my.

Razaidi Hussin received a Ph.D. degree in Electronic and Electrical Engineering from the University of Glasgow, the UK in 2017 with a focus on oxide-reliability issues in complementary metal-oxide-semiconductor nanoscale devices. He joined Universiti Malaysia Perlis (previously known as KUKUM) in 2002. He is currently a full-time Senior Lecturer at the Faculty of Electronic Engineering Technology, Universiti Malaysia Perlis. He can be contacted at email: shidee@unimap.edu.my.

Lee Weng Fook is a Technical Director at Emerald Systems Design Center with 26 years of IC Design experience. Lee has vast experience in designing with Verilog and VHDL, RTL coding, and logic synthesis for ASIC/FPGA/SOC. Lee's specialization is in synthesizing and tweaking synthesis for performance and low power, leading to enhanced methodology to address advanced DFT techniques for VDSM technology, development, and deployment of low power standard cell libraries. Lee has led the development of new architectures and micro-architectures for efficient PMSM motion control ASIC and has developed architectures for AI classification algorithms implementation in ASIC. Lee has published 4 IC Design books, "Learning from VLSI Design Experience" ISBN: 978-3030032371 with Springer Press, "VHDL Coding and Logic Synthesis with Synopsys" ISBN: 0-12-440651-3 with Academic Press Publication, "Verilog Coding for Logic Synthesis" ISBN: 0-471-42976-7 with John Wiley Publication and "VLIW Microprocessor Hardware Design for ASICs and FPGA" ISBN: 978-0071497022 with McGraw Hill Publication. Lee is also the inventor and co-inventor of 14 design patents granted by the US Patent and Trademark Office (US Patent # 7,057,949 7,010,736 6,891,752 6,771,093 6,665,214 6,654,349 6,622,274 6,587,982 6,549,477 6,546,410 6,532,175. He can be contacted at email: seanlee@emersysdesign.com.

Mohd Syafiq Mispan received B.Eng Electrical (Electronics) and M.Eng Electrical (Computer and Microelectronic System) from Universiti Teknologi Malaysia, Malaysia in 2007 and 2010 respectively. He had experience working in the semiconductor industry from 2007 until 2014 before pursuing his Ph.D. degree. He obtained his Ph.D. degree in Electronics and Electrical Engineering from the University of Southampton, the United Kingdom in 2018. He is currently a senior lecturer in the Faculty of Electrical and Electronics Engineering Technology, Universiti Teknikal Malaysia Melaka. His current research interests include hardware security, CMOS reliability, VLSI design, and Electronic Systems Design. He can be contacted at email: syafiq.mispan@utem.edu.my.

Loh Wan Ying received her B.Eng Electronic with Honours from University Tunku Abdul Rahman. Loh is currently a Ph.D. candidate at Universiti Malaysia Perlis, Malaysia. Her research interest is focusing on improving the efficiency of fault detection by optimizing memory testing algorithms. Loh has a few years of experience in designing with Verilog and VHDL, RTL ASIC/IP coding, and FPGA synthesis. Loh also has experience working with RISC V, AXI bus, and FPGA SoC. Loh has experience in Artificial Intelligence (AI) frontend design and full-chip verification. She can be contacted at email: wyloh@studentmail.unimap.edu.my.