ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

Հայրապետյան Դավիթ Լևոնի

ԲՅՈՒՐԵՂԻ ՎՐԱ ՍԽԱԼՆԵՐԻ ԱՌԿԱՅՈՒԹՅԱՄԲ ՀԻՇՈՂ ՀԱՆԳՈՒՅՑՆԵՐԻ ՆԵՐԿԱՌՈՒՑՎԱԾ ԹԵՍՏԱՎՈՐՄԱՆ ԳՈՐԾԸՆԹԱՑԸ ՄՈԴԵԼԱՎՈՐՈՂ ԾՐԱԳՐԱՅԻՆ ԳՈՐԾԻՔԻ ՀԻՄՆԱՎՈՐՈՒՄ ԵՎ ՄՇԱԿՈՒՄ

Ատենախոսություն

Ե.13.04 «Հաշվողական մեքենաների, համալիրների, համակարգերի և ցանցերի մաթեմատիկական և ծրագրային ապահովում» մասնագիտությամբ տեխնիկական գիտությունների թեկնածուի գիտական աստիճանի համար

Գիտական ղեկավար՝

ՀՀ ԳԱԱ ակադեմիկոս, ֆ.մ.գ.դ.

Ս. Կ. Շուքուրյան

ԵՐԵՎԱՆ-2019

YEREVAN STATE UNIVERSITY

Davit Levon Hayrapetyan

JUSTIFICATION AND DEVELOPMENT OF A SOFTWARE TOOL MODELING FAULT-INCLUSIVE SILICON BUILT-IN TEST FLOW FOR EMBEDDED MEMORY COMPONENTS

Thesis
for candidate degree in technical sciences in specialty 05.13.04
“Mathematics and software of computers, complexes, systems and networks”

Supervisor:
Academician of NAS RA, Doctor of Science in Physics and Mathematics
S. K. Shoukourian

YEREVAN-2019

TABLE OF CONTENTS

INTRODUCTION

The relevance of the topic

A rapid increase of density and capacity in memory IP cores embedded in modern system-on-chip (SoC) creates new challenges of preserving test and repair cost while also minimizing time-to-market [1]. On-chip infrastructure IP was suggested to maximize test and repair efficiency utilizing the memory design knowledge and providing analysis on failure data [2]. Considering the increasing complexity of SoC design, it becomes crucial for silicon embedded memory test and repair solutions to keep up with the technology advances and to provide adequate chip quality and yield consistently [2].

With technology shrinking new types of memory defects and corresponding memory fault models for memory test have been observed during post-silicon analysis. That posed new challenges in test and diagnosis of embedded memories in an SoC using all-in-one solutions. We follow the approach of task distribution between hardware (HW) memory built-in self-test (MBIST) network and software (SW) automation tools [3], where the management and control of test and diagnosis flow are implemented via SW, while actual at-speed basic test and diagnostic procedures are performed by the components of MBIST network. This approach is implemented in various solutions that are broadly used in the semiconductor industry [4].

The interaction between SW and HW sides of this mature solution is managed via the creation of test patterns at the SW side, their application to MBIST network via standard interfaces and analysis of obtained results/chains from MBIST network at the SW side. Our considerations are based on a mature test and diagnosis flow [5] that covers three main phases expected from such flows: fault detection, localization, and classification. Each phase of the flow requires special test patterns to be created and analyzed so that results can be propagated to the next phase preparation step. Specific march test algorithms and march-like test algorithms should be developed and either embedded into the MBIST or used for each phase of test and diagnosis flow for generating the corresponding test patterns, which in their turn will be passed to MBIST engine for further at-speed execution.

With the introduction of FinFET technology new types of memory defects have been observed. Test and diagnosis flows designed for faults present in previous designs were not applicable as they were not able to provide necessary coverage and required modification of detection, classification and localization phases [6]. At the same time, solutions elaborated for current designs will face the same issue in the future because of continuous changes in memory designs due to technology shrinkage. A natural demand for prediction of new fault types and modifications of solutions required for test and diagnosis arose. The issue was addressed with the introduction of multidimensional prediction mechanism for memory fault classification [7], that systemizes all known memory faults in periodic manner and gives a view on impending new faults that may appear in memories with new technology nodes. Furthermore, the mechanism offers a general flow for efficient new march test algorithm generation for the new faults based on a test algorithm template. This mechanism will be used in our considerations too for increasing the effectiveness of patterns and fault model verification.

Although the mentioned above patterns for each phase of the considered flow are thoroughly verified before applying to the MBISTs for excluding essential time and quality losses, to the best of our knowledge there are few publications on methods of verification used for MBIST networks post-silicon analysis automation tools. Particularly in [4] the verification approach using MBIST engine based on specifically designed customizable memory device is proposed. Nevertheless, the verification problem of the post-silicon analysis tools on complete MBIST networks remains open. Since it is crucial to ensure the correctness of test and diagnosis flow implementation before it is applied to a manufactured SoC, a justified fault-inclusive and adequate to MBIST network implementation in SoC environment for patterns verification is required for modeling test pattern execution on the MBIST network. Accurate models of memory faults corresponding to defects that might be present in the memory cores should be described and used within the environment. The considered environment should cover necessarily the following 2 requirements:

- fault model generation and injection
- verification of test pattern generation and output chain analysis

As usually the Register Transfer Level (RTL) representation of the MBIST network exists, it is natural to build the environment over and existing HDL (Verilog [7] or VHDL [8]) implementation of MBIST network.

The aim of the thesis

The thesis aims to justify and develop a verification environment for memory test and diagnosis flow basing on extension of memory built-in self-test network RTL representation with fault models. The environment will allow to analyze the received output chains of test patterns applied to the fault-inclusive RTL representation via generation, injection and modelling the memory faults.

Objects of the research

The objects of the research are memory fault models, test algorithms and patterns for memory built-in self-test systems and methods for fault model generation, injection and test pattern verification.

The methods of the research

The methods of theory of automata, test and diagnosis of memory devices, and object-oriented design are used.

Scientific novelty

- Extendable fault model for memory internal faults and its implementation in the RTL HDL simulation environment.
- Automated generation flow for memory internal fault models based on fault periodicity table (FPT).
- Method of test pattern verification in fault-inclusive environment via resulting chain analysis and traversed fault model graph comparison.
- Approach for verification of test and diagnosis flow implementation.

Practical value

The proposed approach allows to increase reliability of test and diagnosis flow implementation in post-silicon analysis test automation tools. The suggested fault modeling mechanism can be effectively extended for new types of faults.

Implementation

The results of the thesis were implemented in Synopsys Inc. software post-silicon analysis automation tool which is currently used by multiple customers.

The following topics are presented for defence

- Extendable fault model for memory internal faults at the Register Transfer Level.
- Approach for fault model injection in the RTL simulation environment.
- A generation flow for memory internal fault models via fault periodicity table.
- Method of generated test patterns verification.
- Approach for verification of test and diagnosis flow implementation.

Presentations

The results of the thesis have been presented and discussed at:

- 11th International Conference on Computer Science and Information Technologies (CSIT), Yerevan, Armenia, 2017
- IEEE East-West Design & Test Symposium (EWDTS), Novi Sad, Serbia, 2017
- IEEE East-West Design & Test Symposium (EWDTS), Kazan, Russia, 2018
- A common seminar of IT Educational and Research Center of YSU

Publications

The results of the study were presented in 4 scientific papers. The papers are listed in the References section.

The structure and size of the thesis

The work comprises of introduction, 3 chapters and an appendix. The content of the work without appendix is represented in 103 pages. It includes 46 figures, 19 tables and the list of references. Total size of the work is 104 pages.

CHAPTER 1. VERIFICATION OF PATTERNS FOR TEST AND DIAGNOSIS FLOW IMPLEMENTATION IN POST-SILICON ANALYSIS AUTOMATION TOOLS

1.1. Problems of implementation of test and diagnosis flows in post-silicon analysis software tools

The main aim of post-silicon test and diagnosis flows for embedded memories is to ensure that they possess a high quality and yield. Diverse approaches were proposed, the objectives of which are to suggest solutions to the three main essential issues of fault detection, localization, and classification that are usually considered after silicon manufacturing. The abovementioned approaches are traditionally used for dislocation of the appeared physical problems and corresponding memory faults by observing their logical behavior [9].

Authors in [10] proposed memory diagnostic [1] tests that allow single-cell faults to be distinguished from multiple-cell faults, while also detecting and localizing them. In [11] researchers used March and March-like test algorithms to obtain test syndromes from random-access memories (RAMs). Syndromes were used to classify the known traditional faults.

In [12], an approach on the physical shape analysis of failures is proposed. The tool also considers incomplete data that can be a consequence of intermittent effects of faults or test noise. Moreover, a software analysis instrument is used for post-processing the reported failure data and providing a hypothesis on memory faults.

The tool essentially leverages memory topological information. The results of the conducted experiments have been provided for the 90nm technology automotive-oriented SoC.

Proposed test and diagnosis solutions are based on memory built-in infrastructures (MBISTs) that are operated via a variety of test patterns[13][14][15], which have an ability to cover maturely the mentioned above three problems. It is a well-known fact that semiconductor companies put tremendous efforts to make manufacturing tests both cost and time effective. This includes test patterns that are mostly designed via post-silicon analysis tools during pre-silicon development of SoCs with the objective to cover multiple

scenarios [16][17]. They essentially depend on a chosen test and diagnosis flow. To the best of our knowledge the most complete flow at the moment is suggested in [5] which we are considering further in our study.

The test and diagnosis flow that we have chosen for the examination consists of the following 7 important phases:

1. Identification of the failed memory instances via running test algorithms on MBIST network.
2. Obtaining logical addresses of memory faults, by instructing MBISTs to stop test execution on N-th error and output diagnostic information.
3. Recognition of physical addresses of faults, based on logical addresses and memory scrambling [3].
4. Identification of physical X and Y coordinates of failing cells based on their physical addresses and memory scrambling.
5. Categorization of the defects based on their placement.
6. Classification of the faults, with the execution of individual tests, and analysis of test syndromes, for faults reported at step 2.
7. Localization of the faulty memory cells (applicable for coupling faults) which also depend on the scrambling information.

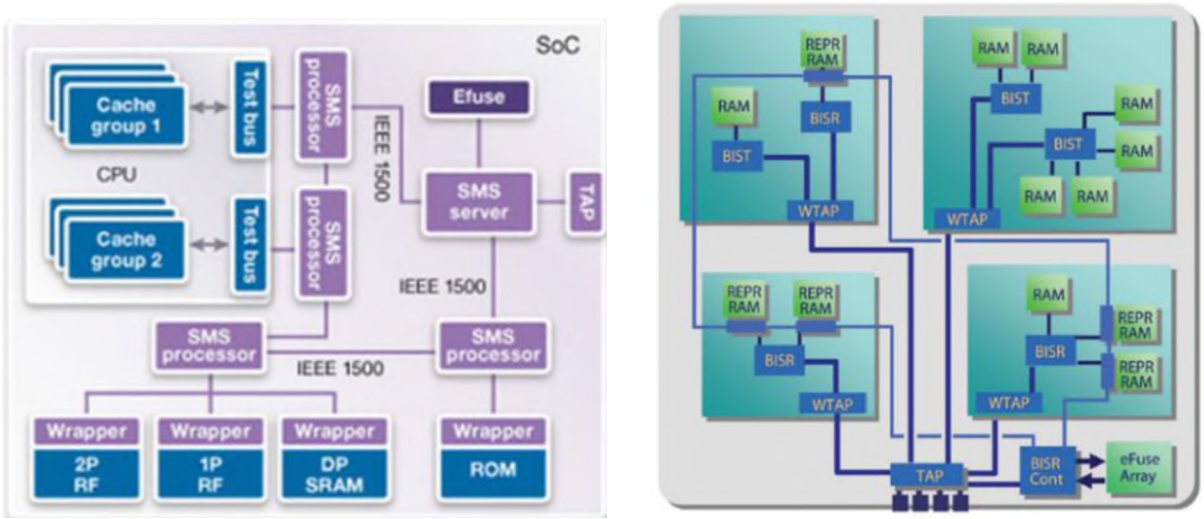
The enactment procedure of each phase of the flow in post-silicon analysis software tools utilizes the data provided by the previous step. Stage 6 though relies only on the data that has been gained in step 2.

Therefore, steps 1, 2, 6, 7 and 1, 2, 3, 4, 5 can be executed separately during the implementation. Steps 1, 2, 5, 6, 7 require a generation of test patterns, their application on MBIST network and analysis of MBIST network outputs.

Test pattern generation and analysis procedures implemented in post-silicon analysis tools require thorough verification at pre-silicon step, since incorrect design of the patterns or software related errors can lead to malexecution of test patterns or misinterpretation of received output chains on manufactured SoC. Considering the abovementioned test and diagnosis flow, an issue present in implementation of one of the phases may lead to an inevitable misbehavior of the whole flow.

1.2. Memory built-in self-test networks

Memory built-in self-test systems, which are also known as MBIST systems represent special infrastructures embedded into a given SoC and are used for at-speed testing of memory devices[2][4]. Modern processor-based MBIST systems (Figure 1) [18][19] comprise memory cores in hierarchical networks referred further as MBIST networks, and are capable of parallel at-speed testing of multiple memory cores via MBIST sub-components. Complexity of the hierarchy is continuously increasing based on increasing number of memory components assembled in SoC. This results in involvement of new supplementary sub-components and more levels of hierarchy. Interactions with MBIST networks are made via binary chains through test access port (TAP)[13][14][15] and the usage of boundary scan. One of the initial stages of MBIST network development is their representation in RTL which is performed using Verilog or VHDL hardware description languages [21]-[24].



a. b.

Figure 1. Examples of commercial MBIST network solutions a. DesignWare STAR Memory System[18] b. Tessent Memory Test[19]

1.2.1. Operating with MBIST network

Instructions are loaded and the information is obtained from MBIST network employing the boundary scan technique: "To help ensure that built-in test facilities can be used or that preexisting test patterns can be applied, a framework is needed that can be used to convey test data to or from the boundaries of individual components so that they can be tested as if they were freestanding. This framework will also allow access to and control of built-in test

facilities of components. A boundary scan coupled with a test access bus provides such a framework."[20]. This enables to accessing all the sub-components of the network via single input and output.

The example of usage of the boundary scan in IJTAG P1687 standard connected to test-access port (TAP) is demonstrated in Figure 2 [25]. The components of SoC in the example are attached to Segment Insertion Bits (SIBs) which control the inclusion of corresponding component in the boundary scan. SIBs are also directed using boundary scan. Consequently, it is possible to state that input with corresponding output chain sizes may diverse from execution to execution. This phenomenon can depend on the specific components, which have been enabled for scan.

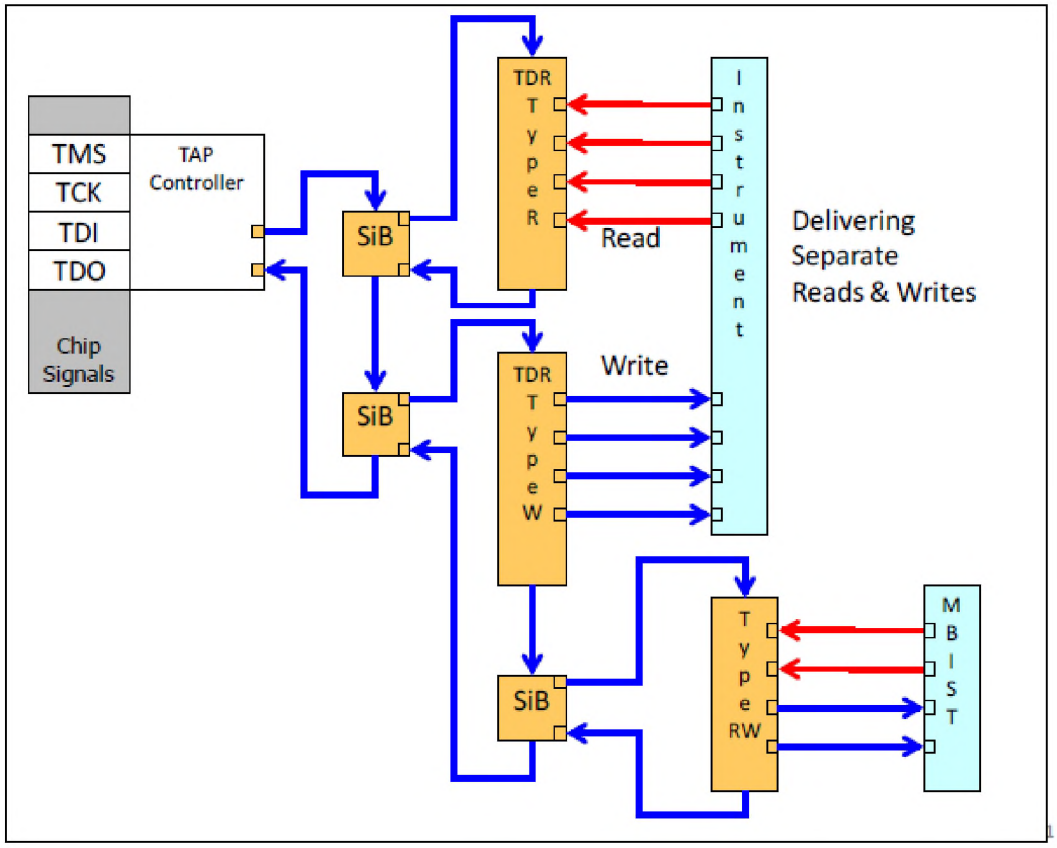


Figure 2 Example of on-chip IEEE P1687 architecture with 3 SIB bits [25]

MBIST networks are comprised of MBISTs for testing memory cores. The main objective of the network is to ensure the propagation of the input chains to corresponding MBISTs, for starting the memory test, and output chains with test results from MBISTs. As chain sequences become more complex it becomes essential to verify these sets before applying them on the real MBIST network, i.e. if the chains are correctly constructed to propagate

commands to MBIST network components. These components are further referred as MBIST. Furthermore, the complexity of MBIST networks is necessarily considered during the analysis of the output chains in post-silicon software tools.

1.2.2. MBIST types

It is a well-known fact that MBISTs use test algorithms for memory testing, generally, thus, it is possible to classify MBISTS into two general categories: FSM-based and microcode-based [26]-[28].

FSM-based MBIST uses hardwired finite state machine representation of test algorithms. This results in the optimum overhead of occupied silicon area, with drawback of not being flexible. Thus, any minor change to test algorithm requires the re-design of MBIST. Even though it can cause inconvenience and be time-consuming in case if minor changes will be needed, these types of MBISTs are still widely spread and used.

In contrast to FSM-based MBISTs, microcode-based MBISTs are more flexible in terms of test algorithms. Having a reasonable area overhead for hardware logic they allow the representation of test algorithms in the form of a microcode. MBISTs can be further reprogrammed with new test algorithms and/or it is also possible to modify the existing test algorithms.

Memory test is not only conducted at the foundry. It is sometimes required for detection of faults that were initially skipped during production and were observed in the field. Therefore, designing programmable MBISTs is essential [29].

Various MBIST solutions are proposed that combine FSM-based and microcode-based approaches while also enabling the programmability of MBISTs through the usage of a test pattern generator [30][31][32]. Nevertheless, these MBISTs can be programmed to generate a test algorithm only from a list of predefined test algorithms.

With the development of new approaches for MBIST, a design is proposed to provide coverage of new types of memory faults [33]. Additional logic is added to MBIST to consider faulty behavior of memory cells based on background patterns, which have been experimentally proved that provides variation in fault coverage by 35% [34]. The types of background patterns are shown in Figure 34. The designed MBIST requires to consider

memory scrambling to correctly distribute values in the memory array to match the background pattern.

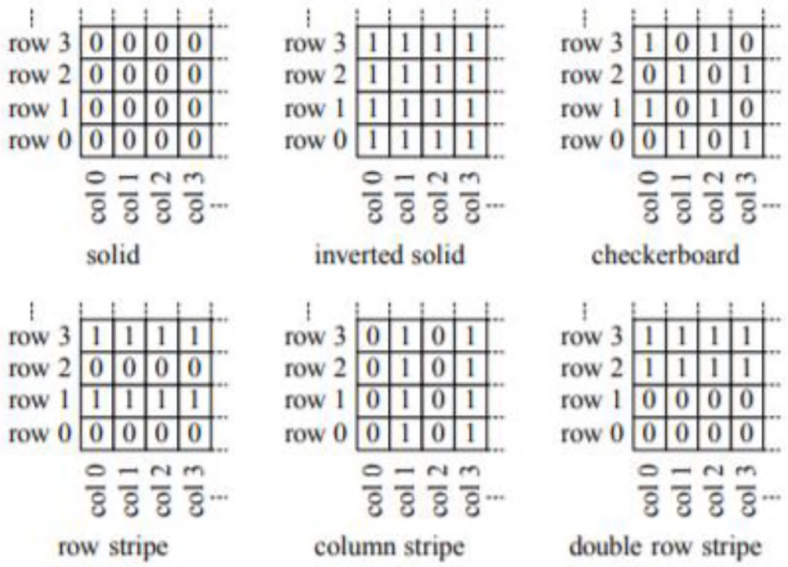


Figure 3. Types of background patterns. [34]

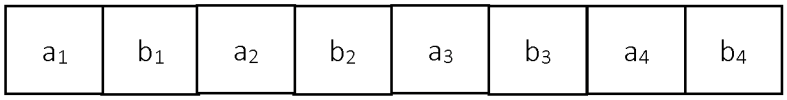
Another approach is also proposed for automated generation of test patterns by MBIST which requires additional hardware logic [6]. MBIST uses the information on memory faults provided on its input. Those faults should be detectable by the resulting generated test algorithm. The development of such architecture is currently in progress and the research is still being conducted. The solution though shows good perspectives to be implemented in future memory built-in self-test systems.

The evolution of MBIST essentially leads to the development of new test algorithms that consider additional features of MBISTs. Therefore, the number of feasible test algorithms drastically increases. Modern MBIST solutions [35], provide an enhanced level of programmability while trying to keep area overhead low. Test algorithms are designed and passed to the MBISTs externally via the abovementioned input chains.

1.2.3. Memory scrambling

Scrambling denotes to the way memory addresses and data patterns are observed outside of the memory device [34]. Our research is mainly concentrating on two types of scrambling which are related to memory array folding and address decoding.

As data in modern memories are stored as words that are accessed with bit-lines and word-lines, the need to adjust the ratios of memory array arises to reduce delay times as well as capacitances for accessing the array cells. Hence, multiple words may be present in a single row of the memory array. The data scrambling, which is generally caused by the alternation of the bits of multiple words with the same index is determined by column multiplexing (CM). Example CM = 2:



First word: a_1, a_2, a_3, a_4
Second word: b_1, b_2, b_3, b_4

Figure 4. Data scrambling

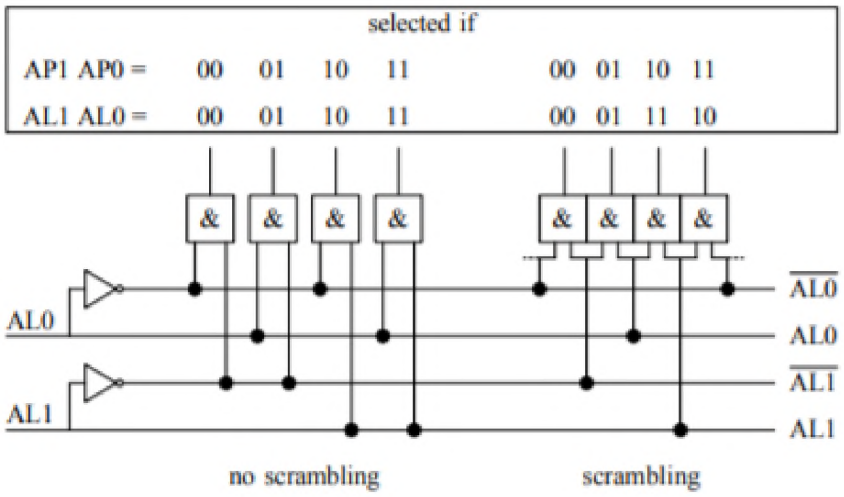


Figure 5. Example of scrambling in address decoder. [34]

Memory arrays are made in the form of a rectangle, and the memory address traditionally consists of two parts, where the first part is used for pointing at a row, while the other points at a column (furthermore, memories can be represented in the shape of multiple banks). Memory address for memories without scrambling is interpreted as row bits followed by column bits. A was mentioned the memory array cells are accessed via word-lines and bit-lines. Mapping logical addresses of words on bit-lines and word-lines for each word separately leads to an overhead of design and additional connection. Reuse of links to the signals in

decoder decreases the number of required connections, while allowing the same level of access to the memory cells.

The memory address scrambling information describes the intermixed positions of the bits from these two parts (Figure 6). Furthermore, the address can also involve bits for memory banks, which in their turn add more complexity to the scrambling.

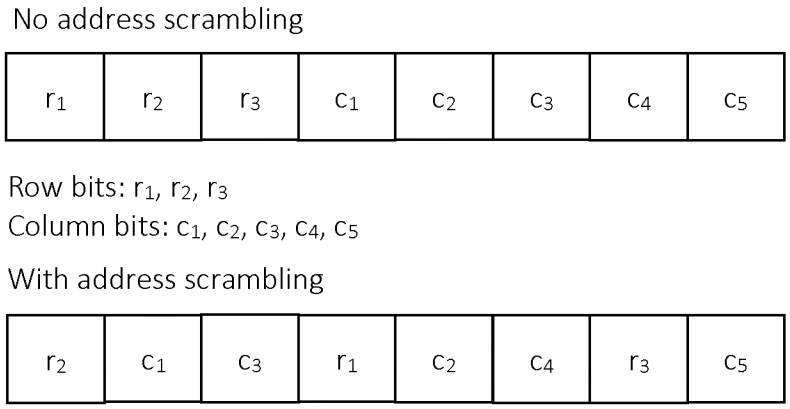


Figure 6. Address scrambling

Scrambling information is required to determine physical address of the memory cells during the chain analysis. The knowledge of memory scrambling is essential during the application of background patterns on memory array or accessing the neighborhood of a targeted memory cell.

1.3. Test algorithms

Embedded memories are tested through the application of sequences of "write"/"read" operations (test algorithms) towards memory cells. These test algorithms require iteration over each memory cell at least once to ensure the coverage. Therefore, the complexity of test algorithms is defined via number of memory cells in the memory array **n** and the number of operations applied to each memory cell. The complexity of some traditional test algorithms used in memory testing is provided in Table 1. The algorithms provided different coverage for memory faults, and thus are used to detect different faulty behavior. Some of them are used only for detection of the faults the others may also localize the cells involved in the faulty behavior.

Table 1. Some traditional test algorithms and their complexity

Test Algorithm	Number of Operations	Complexity
Zero-One	4n	O(n)
Checkerboard	4n	O(n)
MATS++	5n	O(n)
March A	15n	O(n)
March B	17n	O(n)
GALPAT	$2(n+2n^2)$	$O(n^2)$
Walking 1/0	$2(3n + n^2)$	$O(n^2)$
Sliding Diagonal	$6n + 2n\sqrt{n}$	$O(n\sqrt{n})$
Butterfly	-	$O(n\log_2 n)$

The complexity of test algorithms can drastically affect the test time and therefore usage of more complex test algorithms is not always reasonable on large memories. For instance, test times for 1GHz memory are shown in Table 2.

Table 2. Approximate testing times of 1GHz memories based on test algorithm complexity

Memory Size in bits	O(n)	O(nlog$_2$n)	O(n√n)	O(n^2)
1Kb	1000ns	~9965ns	~31622ns	0.001s
1Mb	0.001s	~0.019s	1s	~16.7min
100Mb	0.1s	~2.657s	~16.7min	~115days
1Gb	1s	~29.8s	~8.7hours	~31.6years

Since memory testing is very costly [16] test algorithms of linear complexity are generally being used in production. Nevertheless, some modern programmable MBISTs enable usage of non-linear test algorithms for diagnosis. An example representation of such test algorithm instructions is provided in (Figure 7).

[8]	[7]	[6]	[5]	[4:3]				[2]	[1]	[0]
EBL	BBL	ELL	BLL	B	L	B+1	B-1	0/1	R/W	A
End of Base Loop	Begin of Base Loop	End of Local Loop	Begin of Local Loop	Base	Local	Base + 1	Base - 1	Data or Data Inverse	Read or Write	Algorithm Instruction (=1)

Figure 7. Example of test algorithm instruction representation in programmable MBIST [35]

1.3.1. March test algorithms

Test algorithms of linear complexity were used since the early 1970s when the first embedded memories were launched in mass production. March test algorithms also known as March tests were introduced as well during that period of time [36].

March test are represented as a finite sequence of March elements M = {M_1, M_2, ..., M_k}. March elements in their turn consist of finite number of test operations $M_i = A_i$ (O_1, O_2, ..., O_m).

$A_i \in \{\uparrow, \downarrow, \updownarrow\}$ is addressing direction where:

↑- denotes to increasing address order (from address 0 to n – 1),

↓- denotes to decreasing address order (from address n – 1 to 1),

↕ - means the addressing direction is irrelevant (either ↑ or ↓ can be used).

$O_j \in$ {W(0), W(1), R(0), R(1) } is test operation where:

W(0), W(1) - denote to "write" operation with correspondingly 0 and 1 values applied to a memory cell,

R(0), R(1) - denote to “read” operation applied to a memory cell with expected values correspondingly 0 and 1.

Each operation of March element is sequentially applied to a memory cell, execution of March element is propagated to the next cell, and this action is continuous until all memory cells are tested. The execution can be further passed to the next March element, until all march elements have been considered. An example of March algorithm (MATS+): {⇕{W(0)}, ↑{R(0),W(1)}, ↓{R(1),W(0)}}.

It was shown that March test notation could be extended to operate with multiple bits rather than only single one [37]. For example “write” operation may be represented as $W(D_1, \ldots, D_n)$ where n is the variable that determines the length of memory words.

MBISTs may be designed to operate with multiple “read-write” port memories[38]. The access to each port is implemented separately in MBIST (Figure 8). This leads to modification on test algorithm to consider separate “write” and “read” operations e.g. $W_A(0)$ and $W_B(0)$ correspondingly for A and B ports [39].

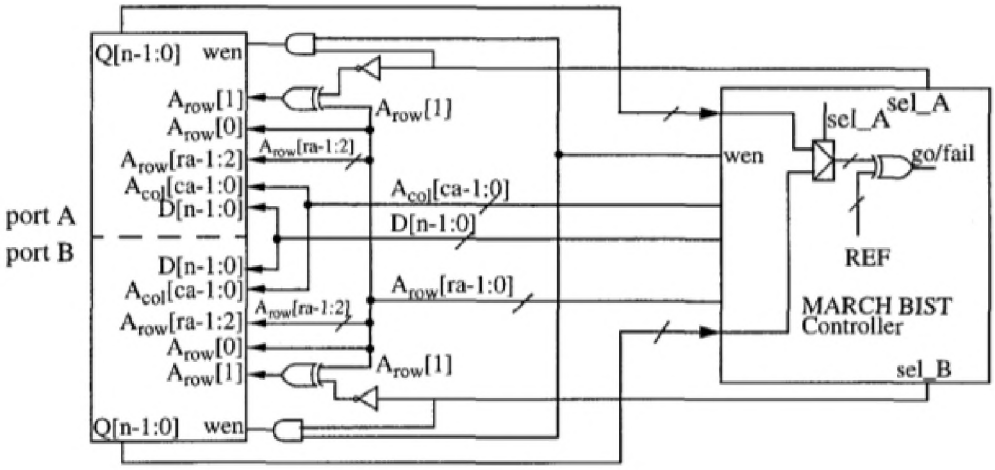


Figure 8. Multi-port memory BIST example. [38]

1.3.2. March-like test algorithms

Faulty behavior in memories occasionally involves more than one memory cell. Some of the memory cells may affect the function of other memory cells while behaving as normal cells. In order to come to the correct conclusion and to achieve accurate diagnosis these cells must be also located. This procedure can be applied after the faulty cell is located using March test algorithms. The faulty cell will be further referred as a victim cell.

Extended notation for March tests is used to describe March-like test algorithms [40][41][42].

$\uparrow_{L}^{H}$ and $\downarrow_{L}^{H}$ addressing is used, where [L-H] describes the range of addressing. W_v and R_v operations are defined which are applied only on victim cell. An example of March-like test algorithm: $\{\uparrow_{0}^{y-1}\{W(\sim A)\}, \{W_v(V)\}, \uparrow_{0}^{y-1}\{W(A), R_v(V)\}\}$[41].

March-like algorithms though require additional hardware infrastructure components to allow the usage of extended March test notation. Particularly, the physical address of the victim cell needs to be stored as well as an additional logic to stop the execution of the March element when it reaches the preceding/succeeding memory physical address. In addition, the infrastructure must be designed to be aware of memory scrambling information since it is operating with memory input/outputs generally via logical addresses.

In sum, modern March and March-like test algorithm have a more complex representation. Programmable MBIST use binary representation for test algorithms which require verification on being correctly generated before passing them to real MBIST networks with input chains.

1.4. Test patterns and output chain analysis

As long as MBIST networks are operated via input chains (p. 12) post-silicon analysis software tools require a convenient representation of these chains. Each input chain has a corresponding output chain of the same length.

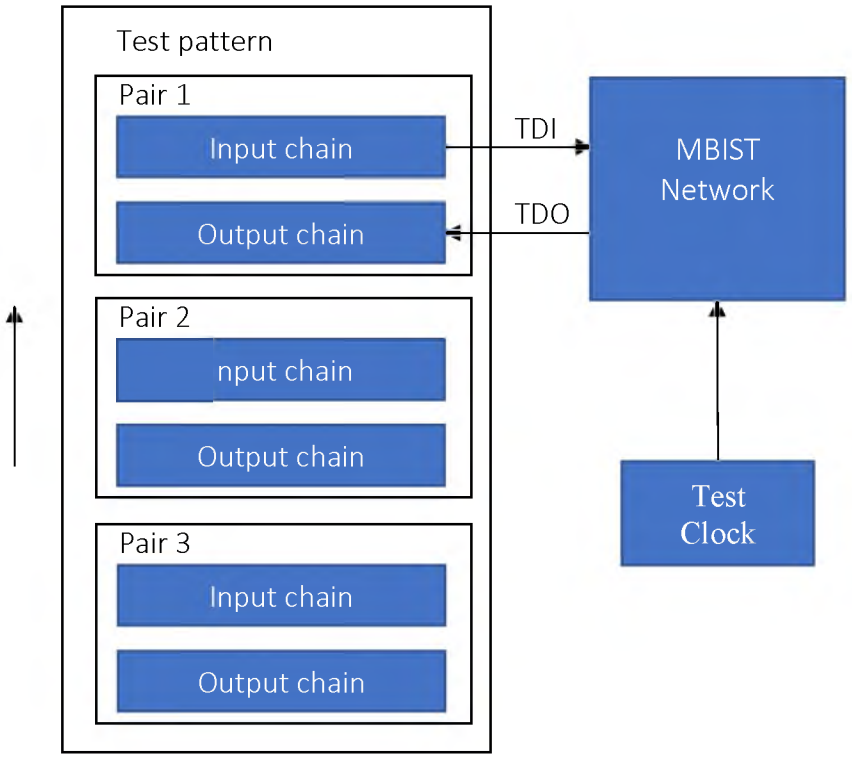


Figure 9. Execution of test pattern on MBIST network

For that reason, test patterns are typically representations of MBIST network input and expected output chains in a sequential form. Their main purpose is to instruct the sub-components of the MBIST network to start the memory testing and propagate test results to the output.

Modern standards for memory testing are based on boundary scan technique, and this is the main reason why test pattern represented in the shape of a list of instructions is converted to binary chains while operating with MBIST. Thus, it is possible to conclude that, verification of the test pattern is the same as a verification of corresponding binary chains. Moreover, the verification of analysis of outputs is the same as of the one of output binary chains.

Considering that decreasing post-silicon test time is essential, accurate test patterns usage on MBIST network allow parallel testing of multiple components. Nevertheless, parallel test

of all the memory cores at the same time is not always possible based on power consumption limitations [44]. Therefore, test patterns are initially enabled and disabling the MBIST network components that will be used in further testing.

The chains are being shifted into MBIST network through the test data input pin, while the output chain is retrieved from the test data output pin. All these operations are made with test clock, it generally takes one test clock cycle to shift in and out 1 bit of data.

Test patterns may additionally be used for propagating the user defined test algorithms through the hierarchy for further execution. This leads to development of software tools that not only enable designing complex user defined algorithms and converting them into binary format, but also embedding them into the input chains passed to the MBIST networks.

Considering the abovementioned diagnosis flow analyzed information is displayed in a convenient format for steps 1, 2 and is processed for more details for steps 3, 4, 5, 6, 7.

Software tools are continuously modified to support new test patterns based on contemporary test algorithms and a rise in complexity of MBIST networks. Furthermore, the examination and reporting flows of those tools are also enhanced to support new types of memory faults.

The input chain generation and output chain analysis flows for test patterns must be verified before the execution on real silicon. The verification requires an adequate MBIST network model to interact, with an ability to define memory defects/faults within.

1.5. Modeling memory internal defects and faults

Embedded memory devices are widely used in modern integrated circuits. Particularly random-access memories (RAMs) are the primary concern of memory testing since they are the most common type of memories used in modern SoCs [1]. There are different types of embedded RAM memories: static RAMs (SRAMs), dynamic RAMs (DRAMs), magnetoresistive RAMs (MRAMs), etc. The RAM devices are comprised of various components such as address latches, column and row decoders, write drivers, sense amplifiers, data registers, refresh logic (DRAMs).The example is the functional model for SRAM that is demonstrated in Figure 10.

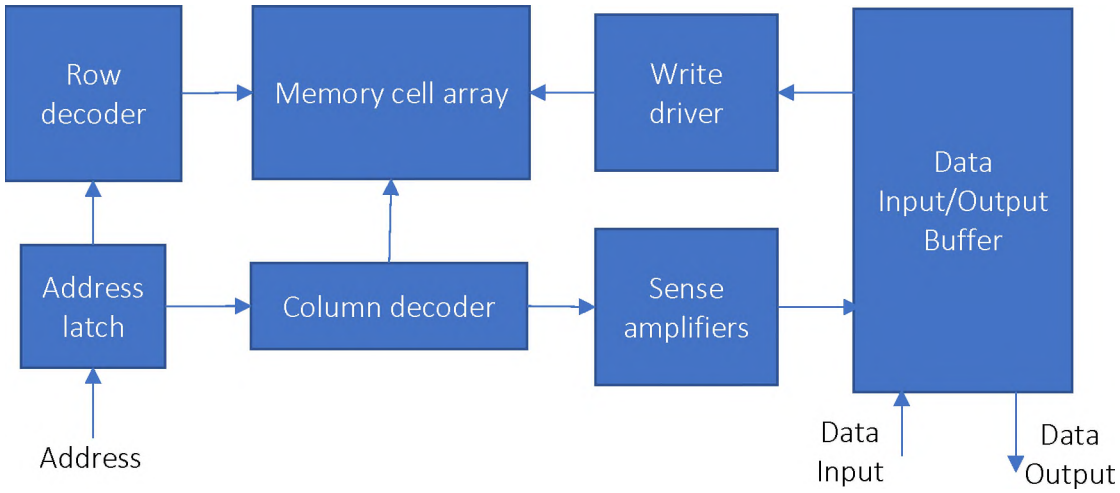


Figure 10. Functional model of SRAM memory

The embedded memory testing considers defects and faults that may occur in different components of the memory device. In addition, we will discuss functional behavior of the defect and error affected memory cells further referred as memory internal faults.

1.5.1. Memory defects, faults and their classes

RAM Memory cells are in their turns comprised of various components. Although having different representation their aim is to maintain/store 0 and 1 values, that can be accessed by reading them or modified by set/reset process.

For example, conventional SRAM memory cells are represented in the form of 6 transistors (Figure 11), though other types of SRAM memory cell representations with 7, 8, 9, 10 transistors are also used [43].

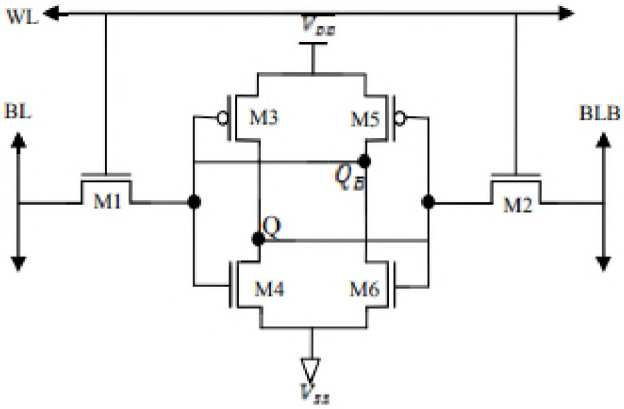


Figure 11. Conventional 6T SRAM cell

DRAM cells use capacitors in their representation which requires to be periodically charged.

Defects occasionally occur in memory cells during silicon manufacturing, which alters the behavior of the cell. Shortcomings can be present in terms of opens and shorts in memory cells [46]100. The resulting faulty behavior of memory cells varies based on the type and location of the defect. The mathematical abstraction of such behavior of memory cells with fault primitives (FPs) had been proposed in [37] which was in future extended in [45]. Special <S/F/R> notation is used to describe fault primitives, where S is a sequence of fault sensitizing operations applied on memory cells, F – state value of the cell after the fault is sensitized, R – value observed on memory cell if last operation of S was "read". S may be split into subsets $\{S_1, ..., S_i, ..., S_n\}$, where each S_i is sequence of operations applied on i-th cell. S_i on its turn may be represented as $\{I, O_1(V_1), ..., O_m(V_m)\}$, where:

I - is the initial state of the cell.

O_j – operation from {W,R} applied on the cell.

V_m – if O_j is "write" operation, then it is the set value. If O_j is "read" operation, then it is the expected value.

It is worth noticing, that V_j value of O_j if it is "read" operation should be equal to the last "write" operation value, or I if no "write" operation was made, i.e. <0R1/1/0> is incorrect notation since if R(1) operation would fail on any fault-free memory cell with 0 value.

First observed memory faults were results of memory incorrect permanent state value. Further evolution of faulty behavior of memory cells ranges from one operation activated

(static fault) to multiple operations enabled (dynamic). List of single-cell static faults is shown in Figure 12.

#	S	F	R	FP	Fault model
1	0	1	–	$< 0/1/- >$	SF_0
2	1	0	–	$< 1/0/- >$	SF_1
3	$0w0$	1	–	$< 0w0/1/- >$	WDF_0
4	$0w1$	0	–	$< 0w1/0/- >$	$TF\uparrow$
5	$1w0$	1	–	$< 1w0/1/- >$	$TF\downarrow$
6	$1w1$	0	–	$< 1w1/0/- >$	WDF_1
7	$0r0$	0	1	$< 0r0/0/1 >$	IRF_0
8	$0r0$	1	0	$< 0r0/1/0 >$	$DRDF_0$
9	$0r0$	1	1	$< 0r0/1/1 >$	RDF_0
10	$1r1$	0	0	$< 1r1/0/0 >$	RDF_1
11	$1r1$	0	1	$< 1r1/0/1 >$	$DRDF_1$
12	$1r1$	1	0	$< 1r1/1/0 >$	IRF_1

Figure 12. Single-cell static faults [45]

Single-cell static faults are also referred by their names rather than notation only, i.e.:

SF – stack-at fault. The memory cell is in a permanent faulty state.

WDF – write-destructive fault. When a "write" operation is applied to a memory cell with a value corresponding to the current memory state, it forces the state of the memory cell to invert.

TF – transition fault. Denotes the case when transition on the memory cell from the given state to the opposite cannot be made.

IRF – incorrect read fault. "Read" operation on the memory cell returns value opposite to the state stored in memory cell [47].

DRDF – deceptive read-destructive fault. "Read" operation inverts the value stored in the memory cell at the same time expected "read" value is returned.

RDF – read-destructive fault. "Read" procedure inverts the value stored in the memory cell and the inverted value can be observed with the operation.

Faulty behavior may also involve two memory cells (coupling faults). In this case one memory cell (aggressor) affects the functioning of the second memory cell (victim), while

acting like fault-free in its turn. Coupling faults are conventionally denoted with <S_a; S_v/F/R> notation, where S_a is the sequence of operations applied on aggressor cell and S_v is the sequence of operations used on a victim cell. As shown in Figure 13 there are overall 36 types of static coupling faults. At the best of our knowledge <S_a; S_v/F/R> notations where both S_a and S_v are operation sequences are not considered as observable in current memory designs.

#	S_a	S_v	F	R	$<S_a;S_v/F/R>$	#	S_a	S_v	F	R	$<S_a;S_v/F/R>$
1	0	0	1	–	<0;0/1/–>	2	0	1	0	–	<0;1/0/–>
3	1	0	1	–	<1;0/1/–>	4	1	1	0	–	<1;1/0/–>
5	0w0	0	1	–	<0w0;0/1/–>	6	0w0	1	0	–	<0w0;1/0/–>
7	0w1	0	1	–	<0w1;0/1/–>	8	0w1	1	0	–	<0w1;1/0/–>
9	1w0	0	1	–	<1w0;0/1/–>	10	1w0	1	0	–	<1w0;1/0/–>
11	1w1	0	1	–	<1w1;0/1/–>	12	1w1	1	0	–	<1w1;1/0/–>
13	0r0	0	1	–	<0r0;0/1/–>	14	0r0	1	0	–	<0r0;1/0/–>
15	1r1	0	1	–	<1r1;0/1/–>	16	1r1	1	0	–	<1r1;1/0/–>
17	0	0w0	1	–	<0;0w0/1/–>	18	1	0w0	1	–	<1;0w0/1/–>
19	0	0w1	0	–	<0;0w1/0/–>	20	1	0w1	0	–	<1;0w1/0/–>
21	0	1w0	1	–	<0;1w0/1/–>	22	1	1w0	1	–	<1;1w0/1/–>
23	0	1w1	0	–	<0;1w1/0/–>	24	1	1w1	0	–	<1;1w1/0/–>
25	0	0r0	0	1	<0;0r0/0/1>	26	1	0r0	0	1	<1;0r0/0/1>
27	0	0r0	1	0	<0;0r0/1/0>	28	1	0r0	1	0	<1;0r0/1/0>
29	0	0r0	1	1	<0;0r0/1/1>	30	1	0r0	1	1	<1;0r0/1/1>
31	0	1r1	0	0	<0;1r1/0/0>	32	1	1r1	0	0	<1;1r1/0/0>
33	0	1r1	0	1	<0;1r1/0/1>	34	1	1r1	0	1	<1;1r1/0/1>
35	0	1r1	1	0	<0;1r1/1/0>	36	1	1r1	1	0	<1;1r1/1/0>

Figure 13. List of static coupling faults [45]

Complex defective behavior that involves multiple FPs and affects one memory cell while masking the faulty behavior of each other (linked faults) has also been observed [48].

Linked faults model denotes the condition when multiple defects corresponding to different single-cell and/or coupling faults are simultaneously present on the overlapping sets of memory cells. These faults are defined in terms of single-cell and coupling faults. The notation for these faults is LF = $FP_1 \rightarrow FP_2$, where FP_1, FP_2 are single-cell and/or coupling faults. Here FP_1 and FP_2 affect the same victim cell, and if the fault sensitizing operation sequence of FP_2 is applied after sensitizing operation sequence of FP_1 the fault will be masked. In other words, it makes sense to consider linked faults, when the following conditions are met:

1. "Read" operations of FP_1 and FP_2 do not detect the faults: if any of the two faults would be detectable by "read" operation the fault can be viewed as stand-alone out of linked faults context.

2. FP_2 masks FP_1: $F_2 = \sim F_1$.
3. FP_2 is compatible with FP_1: sequence of operations S_2 can be applied after S_1, that is, the final states of aggressor and/or victim of FP_1 should be the same as initial states of FP_2. The presence of linked faults may reduce the fault coverage of tests that are not designed to cope with LFs.

In general, there is a classification of 5 types of linked faults which are envisaged in Figure 14.

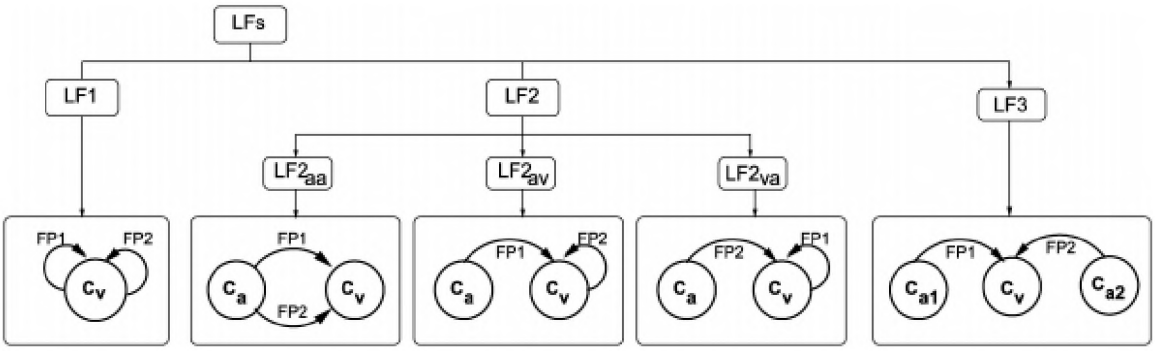


Figure 14. Classification of linked faults [48]

A probability of existence for memory faults that involve more than three cells was also studied [37],[49],[50]. These faults are generally referred as neighborhood pattern sensitive faults (NPSF).

Two types of NPSF faults are considered. They consist of 1 victim cell and 4 or 8 aggressor cells correspondingly (Figure 15). Fault primitives for NPSFs have the following notation:

Type-1 – FP = $<S_N; S_W; S_E; S_S; S_B/F/R>$.

Type-2 – FP = $<S_N; S_W; S_E; S_S; S_{NW}; S_{NE}; S_{SW}; S_{SE}; S_B/F/R>$.

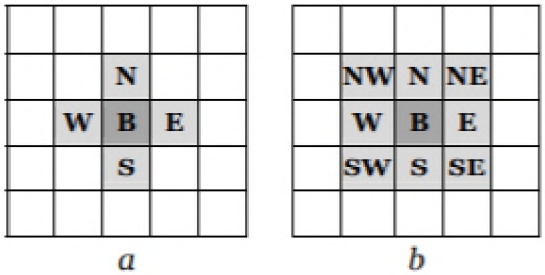


Figure 15. a.type-1, b.type-2 NPSF

S_B denotes the sequence of sensitizing operations applicable on a victim cell, while S_N, S_W, S_E, S_S, S_{NW}, S_{NE}, S_{SW}, S_{SE} are the sequences of operations applicable on aggressor cells. Only one of S_i-s for aggressors can be presented utilizing operation sequence though.

NPSF faults are difficult to detect and require time-consuming test algorithms to be used. Due to the mentioned factors, they are usually not considered during memory testing.

The range of probable faults described in terms of FPs and LFs can be infinite if also considering dynamic faults. Various researches were conducted to determine the realistic dynamic faults and efficient test mechanisms for their detection [51]-[59].

1.5.2. Defect and fault modeling techniques

Fault modeling is an integral part of embedded memory test and diagnosis tools development. It is used almost in all the phases.

With the introduction of Fin Field-effect transistor (FinFET) technology new types of faults were observed since some of the defects may be present in the transistor (not only in memory cell interconnections) and affect its behavior. In today's modern world, major semiconductor corporations use the technology [60]-[62], thus, for the further development of the field, it is essential to have a complete understanding of the defects and precisely determine the faulty behavior that they can cause.

Some latest researches are dedicated to the modeling of memory behavior at transistor level while considering transistor defects [63]-[65]. The defect modeling technique is based on the usage of Simulation Program with Integrated Circuit Emphasis (SPICE) [66]. Two separate simulations are made on memory layout: defect-free and defect-injected. Defect-injected branch uses defects from a library for injection in GDS and SPICE netlist. After the simulation is made resulting waveforms are compared with the results of the defect-free branch.

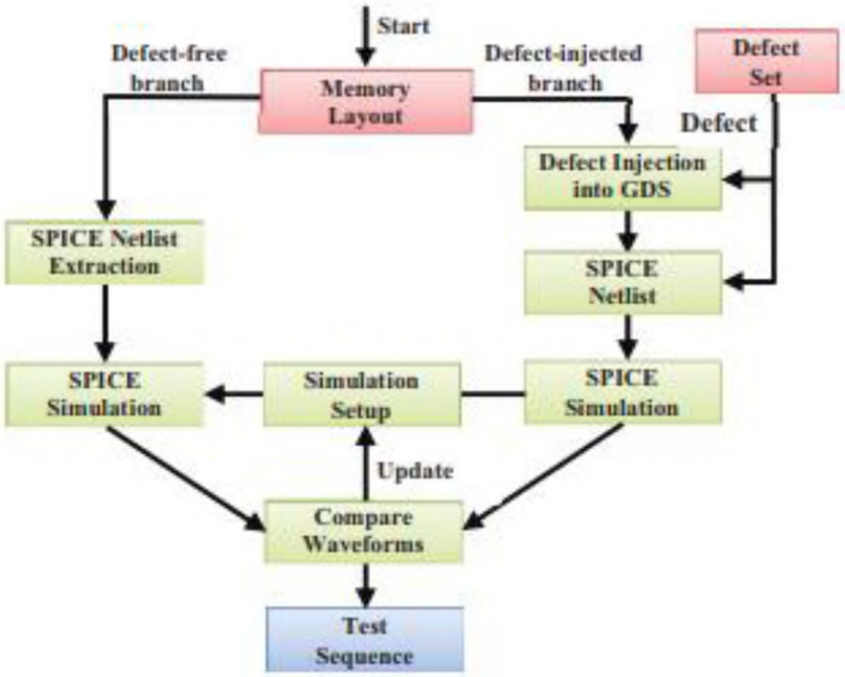


Figure 16. Defect modeling during transistor level simulation [63]

New types of dynamic faults were introduced via usage of the abovementioned technique, e.g. <0R0^{7}/1/0> fault model was determined after resistive fin open defect was injected on one of the memory cell transistors (Figure 17), a sequence of "read" and "write" operations was applied on the memory cell, and the resulting waveform was analyzed.

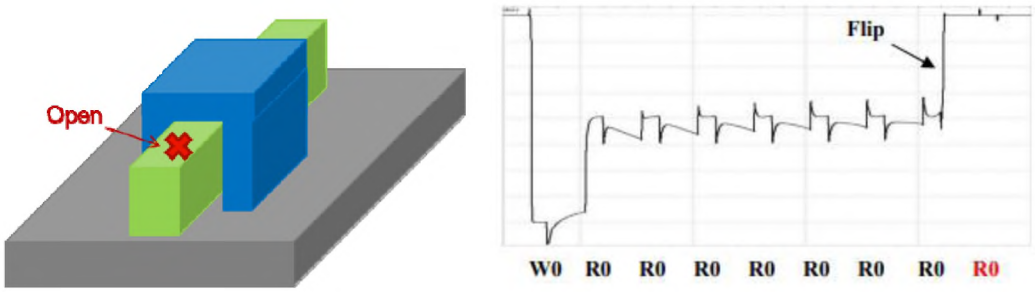


Figure 17. Resistive fin open defect and corresponding waveform [63]

It is evident though that SPICE modeling technique is not applicable for modeling MBIST network behavior in view of the fact that it requires modeling of all the contained components at the transistor level. Even modeling memory behavior is 10-15 times slower in SPICE if compared to Verilog.

Another fault modeling technique for automatic march test algorithms generation was proposed [67]. It is based on the Mealy state machine representation of the memory array.

FPs are represented as additional arcs added to the memory model. Each of the inserted arcs denotes an Addressed Fault Primitive (AFP), which is represented in a shape of a triple:

$$AFP = (I, E, O)$$

- I – is the initialization state.
- E – is the list of operations needed to excite the fault.
- O – is the operation needed to observe the fault.

The number of states of the memory model is determined by the maximum number of cells involved in used FPs: 2^n where *n* is the maximum number of cells. Therefore, the memory model for single-cell and coupling faults is constructed using 4 states. An example of the memory model with FP_1 = <0W1; 0/1/-> and FP_2 = <0W1; 1/0/-> is shown in Figure 18. The model is used for automatic generation of March test algorithms through the medium of solving graph iteration problem, where each "faulty" arc must be traversed only once.

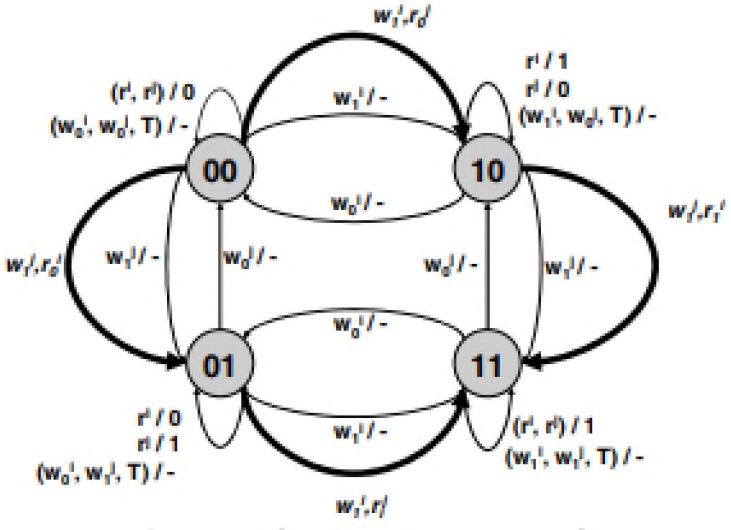


Figure 18. Memory model with coupling faults [67]

Authors in [68] proposed a fault model based on Mealy state machine. An example of coupling fault <0W1; 0/1/-> is shown in Figure 19.

Representation of fault as state machines allows higher level of abstraction. The model can be extended to cover new fault types and be applicable during verification of post-silicon analysis tools.

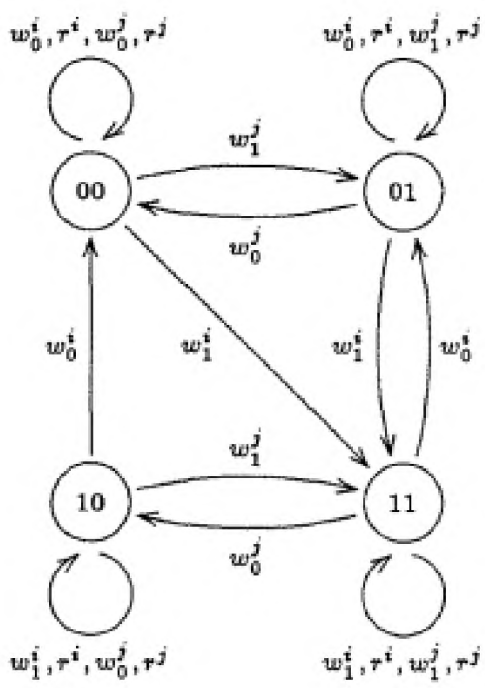


Figure 19. Mealy state machine for coupling fault [68]

1.5.3. Fault prediction mechanism

Authors [69]-[71] have conducted continuous research on the classification of realistic memory faults, determining rules of fault periodicity and developing an approach of test algorithm generation based on the accumulated knowledge which resulted in the introduction of the fault prediction mechanism [6]. The mechanism is based on two constructs: fault periodicity table (FPT) and test algorithm template (TAT).

Fault periodicity table (Figure 20) cumulates FPs of realistic faults in fault groups based on the number of affected cells by FPs and sequences of operations required for detection of defects with the help of test algorithms. Columns in FPT denote the numbers of memory cells affected by FP. Fault groups in each column are combined into fault families based on the number of operations required for fault sensitization. $FG_1(x,S)$ and $FG_2(x, S)$ are referred as FG(x, S). FG(x, S) and FG(~x,~S) are positioned in neighboring rows in FPT.

C0 column denotes memory external faults, C1 – single-cell faults, C2 - coupling faults, C3 – three-cell faults.

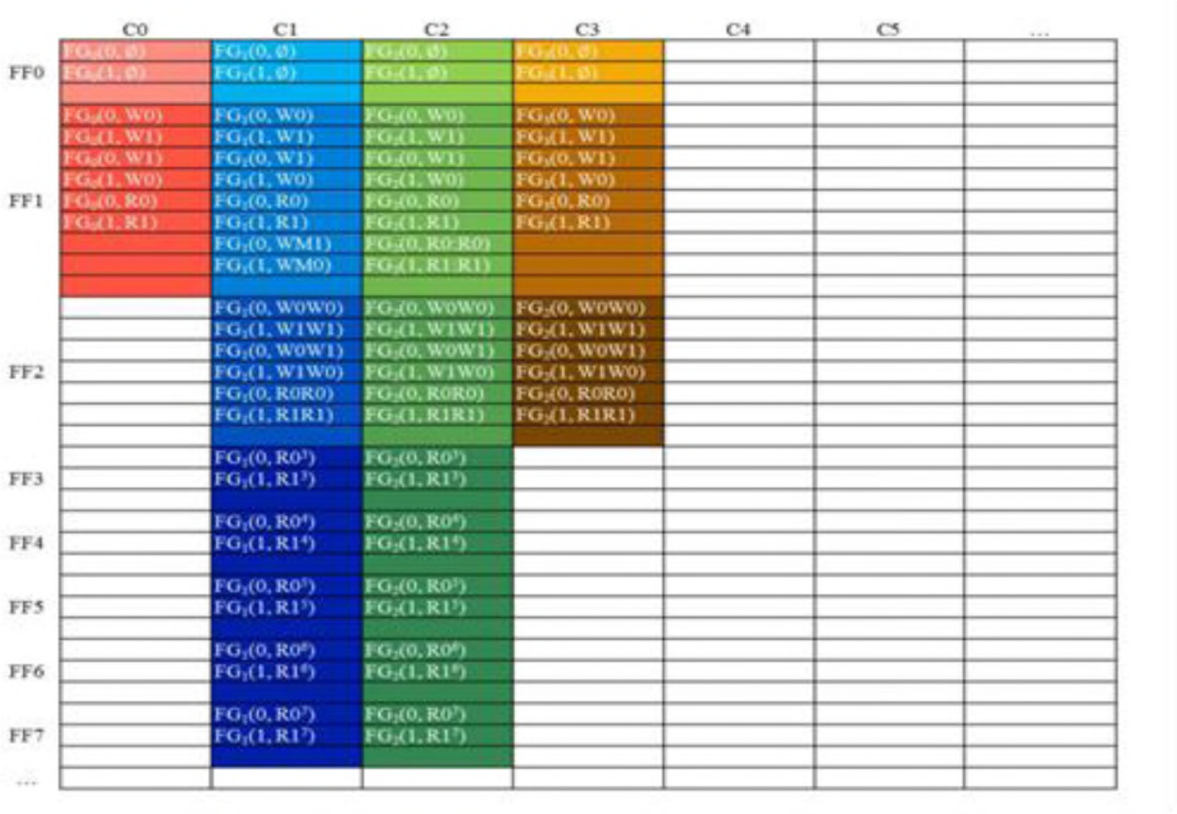


Figure 20. Fault periodicity table

Test algorithm template TAT(x, S) is a function that receives FG(x, S) fault group and constructs a test algorithm is responsible for the detection of the present faults in that group.

Assuming $S = OP_1(D_1), ..., OP_k(D_k)$, $k \geq 0$ and $x \in \{0, 1\}$. Test algorithm template TAT(x, S) has the following structure:

$\updownarrow (W(\sim D_k))$;

$\uparrow ([R(\sim D_k)],[W(x)], S)$;

$\uparrow ([R(D_k)],[W(\sim x)], \sim S)$;

$\downarrow ([R(\sim D_k)],[W(x)], S)$;

$\downarrow ([R(D_k)],[W(\sim x)], \sim S)$;

$\updownarrow (R(\sim D_k))$

where:

1) $\sim S = OP_1(\sim D_1), ..., OP_k(\sim D_k)$, if $k \geq 1$. $\sim S = \emptyset$, if $S = \emptyset$.

2) [W(x)] and [W(~x)] are absent, if $S \neq \emptyset$ and $x = \sim D_k$, otherwise they are present.

3) $[R(D_k)]$ and $[R(\sim D_k)]$ are absent, if $S \neq \emptyset$ and $x = \sim D_k$ and $OP_1 = R$, otherwise they are present.

4) if S = Ø, then consider D_k = x.

The TAT can also be used to construct a combined test algorithm for multiple fault groups.

It will be shown further through our work how the realistic linked faults can also be grouped in the fault periodicity table. This will finally cumulate all currently known realistic memory internal faults that can be described via terms of FP or derived notations in the fault periodicity table.

Fault prediction mechanism can be leveraged during both the verification and test pattern creation of post-silicon analysis automation tools to decrease exhaustion in the fault model through consideration of only realistic memory faults and automated test algorithm generation.

1.6. Problem statement

Post-silicon validation effort consumes more than a half of an SoC's overall design effort and therefore different approaches are probed to decrease the gap between pre-silicon and post-silicon validation techniques [72]. Verification environments are used for functional testing of SoCs, based on either RTL simulation or using accelerators (FPGA boards) that follow the unified verification methodology [73] (Figure 21).

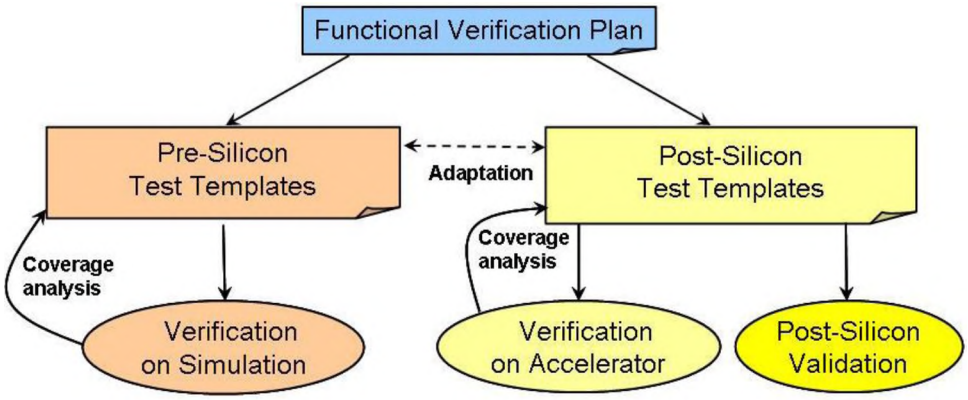


Figure 21. A unified verification methodology [73]

Verification flows for tools operating with MBIST networks, though, need to consider not only the functional behavior of the SoC, but also probable faulty behavior of memory cells. At best of our knowledge there are just a few works dedicated to that topic. Particularly authors in [4] proposed an environment for testing MBIST controller and tools that operate with it, while using specially designed memory Verilog model (Figure 22). The primary purpose of the model is to depict the real structure of the memory device, by individually modeling each memory cell along with their interconnections within the device. This approach though might be inefficient for verification of MBIST networks with multiple memory devices of different size and configuration. Besides, the model should be able to depict the complexity of the inner structure of modern memory designs.

In another research, authors used a fault injection framework for exploiting error correction code (ECC) functionality of some SRAM memories [74]. The verification environment has been implemented in VHDL and currently covers only a small range of faults.

Since memory internal faults cannot be modeled with transistor defects, opens and shorts during MBIST verification in RTL or on FPGA, due to a higher level of abstraction used in

corresponding tools, a fault model based on a special fault FP notation of the faults may be considered, instead.

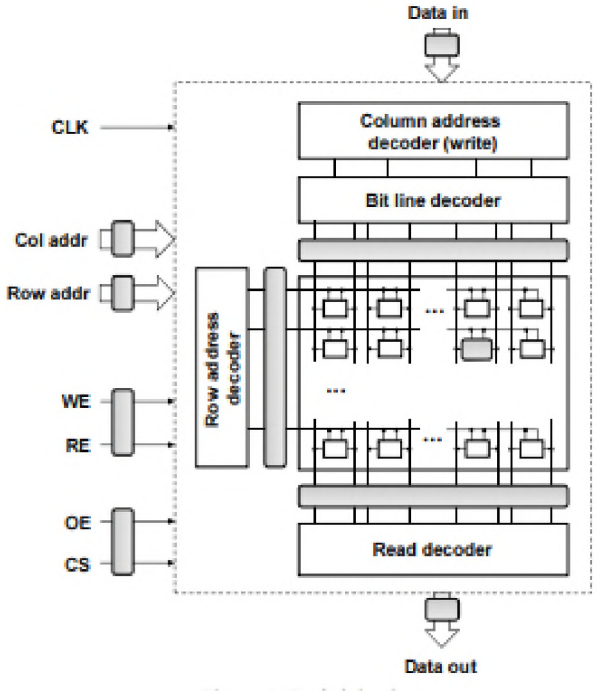


Figure 22. Memory model with fault injection [4]

Summarizing the study presented in this chapter: the need of development of new test patterns and corresponding test algorithms, new reporting and analysis flows, rising complexity of MBISTs and new types of memory faults, a verification environment must be built which will:

- Allow injection of a given fault model or set of fault models into the RTL model describing a considered SoC.
- Enable verification of test pattern generation.
- Enable verification of output chain analysis.

Additionally, the environment may consider fault prediction mechanism to decrease the exhaustion and automate the process.

The following chapters are dedicated to identification and proposition of solutions for the listed above problems.

Conclusions

1. Test pattern verification problem was outlined as one of the most important verification problems in post-silicon analysis of MBIST networks.
2. It is necessary to build a fault-inclusive environment for effective modeling test pattern execution on an MBIST network.
3. The fault-inclusive environment should be capable to build extendable fault models.
4. Approaches on verification of test pattern input chain generation and output chain analysis must be investigated. It is shown that verification problem is reduced to verification of test pattern generation and output chain analysis.

CHAPTER 2. MEMORY FAULT MODELS AND THEIR GENERATION FLOW

In this chapter we propose a technique for modeling memory internal faults. It uses a deterministic finite automata (DFA) based models. An extendable automata fault model based on [68] will be developed for RTL HDL representation of MBIST networks, as well as provide capabilities on modeling the newly introduced faults based on memory fault periodicity.

Formal representation and fault model DFA generation procedures for single-cell and coupling faults will be included along with the supporting examples [75],[76]. Linked faults model and procedure of its generation will be derived via the mentioned fault models for FPs [77].

Finally, an automated FDT generation flow based on fault prediction mechanism will be adduced [85].

Requirements for fault model implementation in the HDL representation of the MBIST network will be identified as well [78].

2.1. Automata model of memory faults

In this paragraph we introduce a fault-modeling technique for FPs based on deterministic finite automata. The structure of the model, as well as its features, are discussed further.

2.1.1. Deterministic finite automaton

We use the formal definition for deterministic finite automata (further DFA) given in [75], where DFA is defined as 5-tuple:

$$A = (Q, \sum, \delta, q_0, F)$$

where:

1. Q is a finite set of DFA states,
2. $\sum$ is a finite set of input symbols,
3. δ or $\delta\ (q, a) = q'$ is a transition function that takes a state q and an input symbol a as arguments and returns a state q'. In DFA δ must be determined for each $q \in Q$ and each $a \in \sum$, resulting in a transition to a single $q' \in Q$.

4. q_0 is the start state,
5. F is set of final or accepting states and $F \in Q$.

The elements of 5-tuple will be defined for fault model in the following manner:

1. Since FP is determined for n memory cells, with $n-1$ aggressor cells and 1 victim cell, each $q \in Q$ will be presented as $(n+1)$-tuple $q:=(R, S_1, ..., S_n)$, where n determinant is the number of cells affected by FP and R is the value to be returned by "read" operation on the victim cell if applied at DFA state q,
2. Each $a \in \sum$ is an operation that may be applied to one of the n cells. Importantly in case if two or more cells are affected by FP, then, they must be distinguishable despite the fact that the same operation is being applied to either one of them. For instance, "read" operation R can be numbered for each of n cells in $\sum$ in the following manner $R_1, R_2, ..., R_n$,
3. $\delta (q, a) = q'$ is a transition from $(n+1)$-tuple q to q' in a result of operation a applied on one of n cells.
4. q_0 denotes initial state of DFA where all n cells are uninitialized.
5. F set consists of one final state which denotes the activation of the fault.

DFA model can also be depicted in a form of a finite graph or a finite state-machine. This ability will be used during the implementation of the model.

2.1.2. The structure of the model

The general structure of the model can be divided into four parts and is presented in the following illustration (Figure 23). The four integral segments include: initial state, initialization block, fault behavioral block and fault activation state.

2.1.3. Initial state and initialization block

The initial state of fault model, as it has beed already mentioned, represents uninitialized memory cells affected by FP. The cell values of $(n+1)$-tuple are assigned with X value. X values

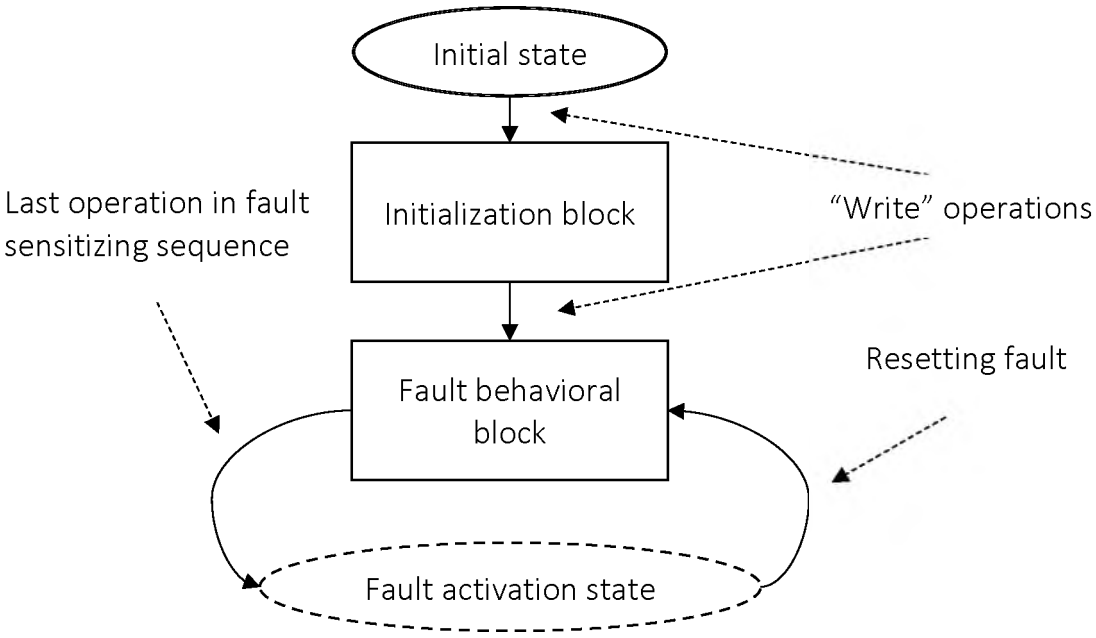


Figure 23. Fault model structure

of *(n+1)*-tuple cells are replaced with 0s and 1s during transitions in initialization block. Taking into consideration that there is no "write X" operation, at some point X value is eliminated from DFA state *n*-tuples and the transition is made to fault behavioral block.

The purpose of the initial state and initialization block are to model behavior of memory cells ar the beginning of the test, when the actual values of memory cells are unknown (can be either 0 or 1).

X value is also commonly used to represent unknown value in HDL languages.

2.1.4. Fault behavioral block and fault activation state

Fault behavioral block models the behavior of fault-free memory cells till conditions defined by FP are not met. Otherwise, the transition to fault activation state is made. The back and forth conversion from fault activation state to fault behavioral block will be further supported in case if there is no contradiction and conflict with FP.

Taking into account that some faults may be initiated without being noticed, due to the missing "read" operation, it is essential to use the fault activation state as a final stage, in order to avoid the possible lacks. The triggering final state is helpful in the additional analysis of the test approach for detection of the modeled fault.

2.1.5. Reset operation

Since dynamic faults are sensitized after the sequence of operations is uninterruptedly [79] applied on a memory cell, "reset" operation is added to the model for purpose of being used for transitions in cases when the sequence of operations was interrupted. "Reset" transition should be explicitly called on the DFA of faulty memory cell each time a switch to another cell is made.

2.1.6. Model interpretation as Mealy state machine

Introduced DFA model can be easily converted to Mealy state machine by adding $\Omega = \{-, 0, 1\}$ output alphabet, and λ output function $\lambda: Q \times \sum \rightarrow \Omega$, for $\forall\, q \in Q$ and $\forall\, a \in \sum$, such as for given $\delta\,(q, a) = q'$, λ on a operations different from "read" returns "-", and returns "0" or "1" on "read" operation based on value of:

1. Memory cell state of $(n+1)$-tuple q' if it is an aggressor cell,
2. "Read" value of $(n+1)$-tuple q' if it is a victim cell.

Mealy state machine representation will be used during the application of fault model.

2.1.7. Fault description table

Throughout this work we will leverage the core ability of DFA to be represented in a form of a table (Table 3).

Table 3. Table representation of DFA

State	a_1	...	a_k
q_0	$\delta\,(q_0, a_1)$	...	$\delta\,(q_0, a_k)$
...	...	...	...
q_m	$\delta\,(q_m, a_1)$	...	$\delta\,(q_m, a_k)$

The binary format for the table will be a useful tool during the fault model implementation. The illustration is further referred as fault description table (FDT) (Table 4). As long as each row of the table corresponds to a DFA state, its index in FDT, counting from the top, is converted to a binary address and is bound to that state, $\delta\,(q_0, a_1)_{bin} := binary\ (index$

(δ (q0, a1))). The first column is split into (n+1) parts for storing *(n+1)*-tuple values. Each *δ (qi, aj)* state is replaced with its binary address.

Table 4. Fault Description Table (FDT)

R	S_1	...	S_n	a_1	...	a_k
R^0	S_1^0	...	S_n^0	$\delta (q_0, a_1)_{bin}$	...	$\delta (q_0, a_k)_{bin}$
...	...	...	...	...	...	...
R^m	S_1^m	...	S_n^m	$\delta (q_m, a_1)_{bin}$	...	$\delta (q_m, a_k)_{bin}$

2.1.8. Visualization of fault model

Graphviz open source code Graph Visualization Software [80] will be used further for visualization of generated fault FDTs. Illustration of fault FDTs is useful to fix appeared minor issues in the implementation of the corresponding generation procedures. It also allows observing the behavior of fault model in a convenient format, rather than manually iterating through FDT, thus making the implementation of the process faster and more efficient.

Generation of FDT representation in DOT language [81] was implemented in FDT generation procedures. An example of <0/1/-> single-cell fault FDT in DOT is provided below:

```
digraph G {
    size = "7, 10!" ratio = fill;
    label = " (<0/1/->)"
    00000000 -> 00000010 [label = R]
    00000000 -> 00000010 [label = W0]
    00000000 -> 00000010 [label = W1]
    00000000 -> 00000010 [label = Reset]
    00000000 [label = "Rx\nSx"]
    00000001 -> 00000001 [label = R]
    00000001 -> 00000001 [label = W0]
    00000001 -> 00000010 [label = W1]
    00000001 -> 00000001 [label = Reset]
    00000001 [label = "R0\nS0"]
    00000010 -> 00000010 [label = R]
```

```
        00000010 -> 00000010 [label = W0]
        00000010 -> 00000010 [label = W1]
        00000010 -> 00000010 [label = Reset]
        00000010 [label = "R1\nS1\nF"]
        00000000 [style = dotted]
        00000010 [peripheries = 2]
    }
```

The representation was further depicted in Graphviz using dot.exe.

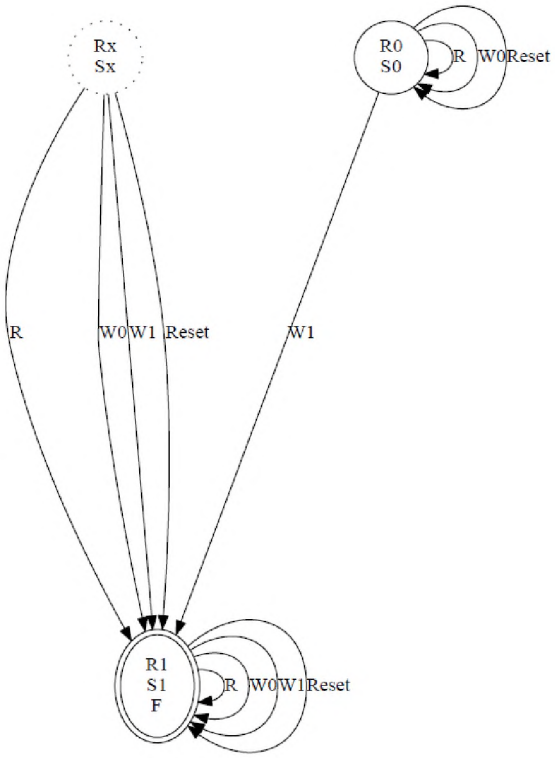


Figure 24. <0/1/-> stack-at 1 fault DFA visualized with Graphviz

2.2. Generation of fault models for single-cell, coupling and linked faults

In this passage we will discuss and concentrate more on the technique of modeling single-cell, coupling [76] and linked [77] faults with DFA and generation procedures for corresponding FDTs [78].

2.2.1. Automata model for single-cell faults

The fault model template is based on <S/F/R> notation for single-cell faults. If we assume that the number of operations in S is *n,* it can be represented as follows:

$$DFA\ A_{single\text{-}cell} = (Q, \sum, \delta, q_0, F)$$

where:

1. $Q = q_0 \cup \{q_i := (R\alpha, S\alpha) : \alpha \in \{0, 1\}, i = 1, ..., n+1\}_{fault\text{-}behavioral\text{-}block} \cup \{q_{fault\text{-}activation\text{-}state} := (R\alpha, S\beta): \alpha \in \{0, 1\}, \beta \in \{0, 1\}\}$, where $\{...\}_{fault\text{-}behavioral\text{-}block}$ is absent in case of stack-at faults,
2. $\sum = \{W(0), W(1), R, Reset\}$,
3. $\lambda: Q \times \sum \rightarrow Q$, for $\forall\ q \in Q$ and $\forall\ a \in \sum$,
4. $q_0 := (RX, SX)$,
5. $F = \{q_{fault\text{-}activation\text{-}state}\}$.

Each state of DFA in the model is a pair (R, S), where R is the value returned by "read" operation if applied on a memory cell, while S is actual value of memory cell at that state.

2.2.2. FDT generation procedure of for single-cell fault model

FDT for single-cell FP with <S/F/R> notation is constructed on the basis of the predefined FDT of the normal, non-faulty memory cell (Table 5). 4 bits is sufficient memory for addressing states of single-cell fault DFA. Indeed, 2^4=16, and since the number of DFA states is at most (n+3) where n is the number of operations in S, n = 16 – 3 = 13. No realistic single-cell FP with 13 operations is currently observed.

Table 5. Predefined FDT for single-cell faults

R	S	R	W(0)	W(1)	Reset
X	X	0000	0001	0010	0000
0	0	0001	0001	0010	0001
1	1	0010	0001	0010	0010

The second and the third rows will be referred as state-0 and state-1 for convenience.

The flow consists of the following steps:

1. Parsing <S/F/R> notation and storing the results in dedicated constructs. Particularly S is represented as a sequence of sensitizing operations in the following format:

$$S := IO_1(V_1)...O_n(V_n))$$

where $O_i \in \{W, R\}$, $V_i \in \{0, 1\}$, $I \in \{0, 1\}$.

2. In case if n = 0, then it is "stack-at" fault case. No additional state is required to be added to the predefined FDT. Instead, based on the *I* value transitions from initial state are set to either state-0 or state-1. All transitions from the last one, in their turn, are modified to point the current row.

3. If n > 0, then for each O_i operation add a new row to FDT as follows:

   a. V_i_V_i_XXXX_XXXX_XXXX_XXXX, if i ≠ n,

   b. R_F_XXXX_XXXX_XXXX_XXXX, if i = n and R is present in <S/F/R> notation,

   c. F_F_XXXX_XXXX_XXXX_XXXX, if i = n and R is not present in <S/F/-> notation.

Simultaneously transitions at the preceding row in FDT are updated for O_i if i > 1 or state-0/state-1 if i = 1 to point to the newly added row. Transitions in the recently inserted row are set to point the state-0 and state-1 for "write" operations and state-0/state-1 based on V_i value for "read" and "reset" operations.

4. The changes in the final row are set in the following way if n > 1, "reset" points to state-0/state-1 based on F, "read" transition points to the final row if O_n is "read" operation or to state-0/state-1 based on R of <S/F/R>, "write" transitions point to state-0 and state-1 correspondingly. If n = 1, then:

   a. if O_n is "read" operation, then "read" and "reset" transitions are set to state-0/state-1 if V_i = R and R ≠ F, and set to point the final row if V_i = F and R ≠ F. "Write" transitions are set to state-0 and state-1 correspondingly.

b. if O_n is “write” operation and $V_n \neq I$, then $W(V_n)$ transition is set to point to the final row of FDT, while $W(\sim V_n)$ points to state-0/state-1 correspondingly. “Read” and “reset” transitions are set to point to the last row.

All added states, besides the final state will be referred further, as fault model intermediate states.

Some examples of single-cell faults generated with described flow and visualized through Graphviz are presented below (Table 6, Figure 25, Table 7Figure 25.<R0R0R0R0R0/1/1> fault visualized with Graphviz.). Notably, the rise in the complexity of representation can be observed due to the increase of operations in fault sensitizing sequences in FPs.

Table 6. FDT for <0/1/-> fault

R^S	S^S	R	W(0)	W(1)	Reset
X	X	0010	0010	0010	0010
0	0	0001	0001	0010	0001
1	1	0010	0001	0010	0010

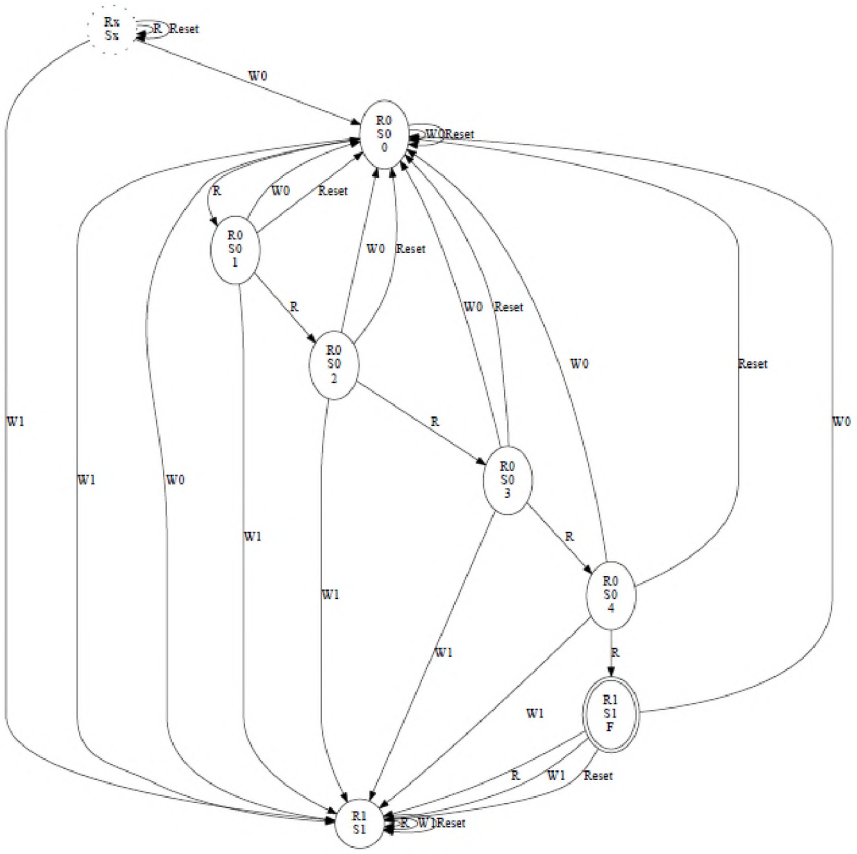


Figure 25.<R0R0R0R0R0/1/1> fault visualized with Graphviz.

Table 7. FDT for <R0R0R0R0R0/1/1> fault

R^s	S^s	R	W(0)	W(1)	Reset
X	X	0000	0001	0010	0000
0	0	0011	0001	0010	0001
1	1	0010	0001	0010	0010
0	0	0100	0001	0010	0001
0	0	0101	0001	0010	0001
0	0	0110	0001	0010	0001
0	0	0111	0001	0010	0001
1	1	0010	0001	0010	0010

2.2.4. Automata model for coupling faults

Coupling fault model template is based on $<S_a; S_v/F/R>$ notation for coupling faults, assuming the total number of operations in S_a and S_v is *n,* it can be represented as follows:

$$DFA\ A_{single\text{-}cell} = (Q, \sum, \delta, q_0, F)$$

where:

1. $Q = q_0 \cup \{(RX, S_a0, S_vX), (RX, S_a1, S_vX), (R1, S_aX, S_v1), (R0, S_aX, S_v0)\}_{initialization\text{-}block} \cup \{q_i := (R\alpha, S_a\beta, S_v\alpha) : \alpha \in \{0, 1\}, \beta \in \{0, 1\}, i = 1, ..., n+4\}_{fault\text{-}behavioral\text{-}block} \cup \{q_{fault\text{-}activation\text{-}state} := (R\alpha, S_a\beta, S_v\gamma): \alpha \in \{0, 1\}, \beta \in \{0, 1\}, \gamma \in \{0, 1\}\}$,
2. $\sum = \{W_a(0), W_a(1), R_a, W_v(0), W_v(1), R_v, Reset\}$, where $W_a(0), W_a(1), R_a$ are operations that can be applied on the aggressor cell with $W_v(0), W_v(1), R_v$ operations for the victim cell correspondingly,
3. $\lambda: Q \times \sum \rightarrow Q$, for $\forall\ q \in Q$ and $\forall\ a \in \sum$,
4. $q_0 := (RX, S_aX, S_vX)$,
5. $F = \{q_{fault\text{-}activation\text{-}state}\}$.

Each state of DFA is represented in the form of a triplet (R, S_a, S_v), where R is the value returned by "read" operation if applied on a victim cell, S_a is the value of aggressor cell and S_v

is the actual value of the victim cell at that state of DFA. "Read" value of aggressor cell is not included, due to the fact that it is the same as S_a.

2.2.5. FDT generation procedure for coupling faults

FDT for coupling fault FP with <S_a; S_v/F/R> notation is constructed similarly to single-cell faults, yet, with some exceptions. Predefined FDT of normal, non-faulty behavior of two memory cells is used as a basis for construction of DFA model (Table 8). As long as the number of initial states has increased due to the involvement of an additional cell we use 6 bits for transition addressing in FDT for coupling faults.

Table 8. Predefined FDT for coupling faults

R^s	S^s_a	S^s_v	R_a	$W_a(0)$	$W_a(1)$	R_v	$W_v(0)$	$W_v(1)$	Reset
X	X	X	000000	000001	000010	000000	000011	000100	000000
X	0	X	000001	000001	000010	000001	000101	000111	000001
X	1	X	000010	000001	000010	000010	000110	001000	000010
0	X	0	000011	000101	000110	000011	000011	000100	000011
1	X	1	000100	000111	001000	000100	000011	000100	000100
0	0	0	000101	000101	000110	000101	000101	000111	000101
0	1	0	000110	000101	000110	000110	000110	001000	000110
1	0	1	000111	000111	001000	000111	000101	000111	000111
1	1	1	001000	000111	001000	001000	000110	001000	001000

We will use state-S_aS_v for reference of the rows from 1 to 9.

The flow consists of the following steps:

1. Parsing <S_a; S_v/F/R> notation and storing the results in dedicated construct. The S_a and S_v are parsed in the following manner:

$$S_a := I_aO_1(V_1)...O_n(V_n)),$$

$$S_v := I_vO_1(V_1)...O_n(V_n))$$

Since only either S_a or S_v can represent the sequence of operations, I_a/I_v will be considered for the second memory cell behavior.

2. If n = 0, then it is state coupling fault case. All $W_a(I_a)$ transitions set from initial state and initialization block states are updated to point the faulty state-I_aF. Now additional rows are required to be added to the FDT.

The resulting DFA model for <0;1/0/-> fault is shown in Figure 26.

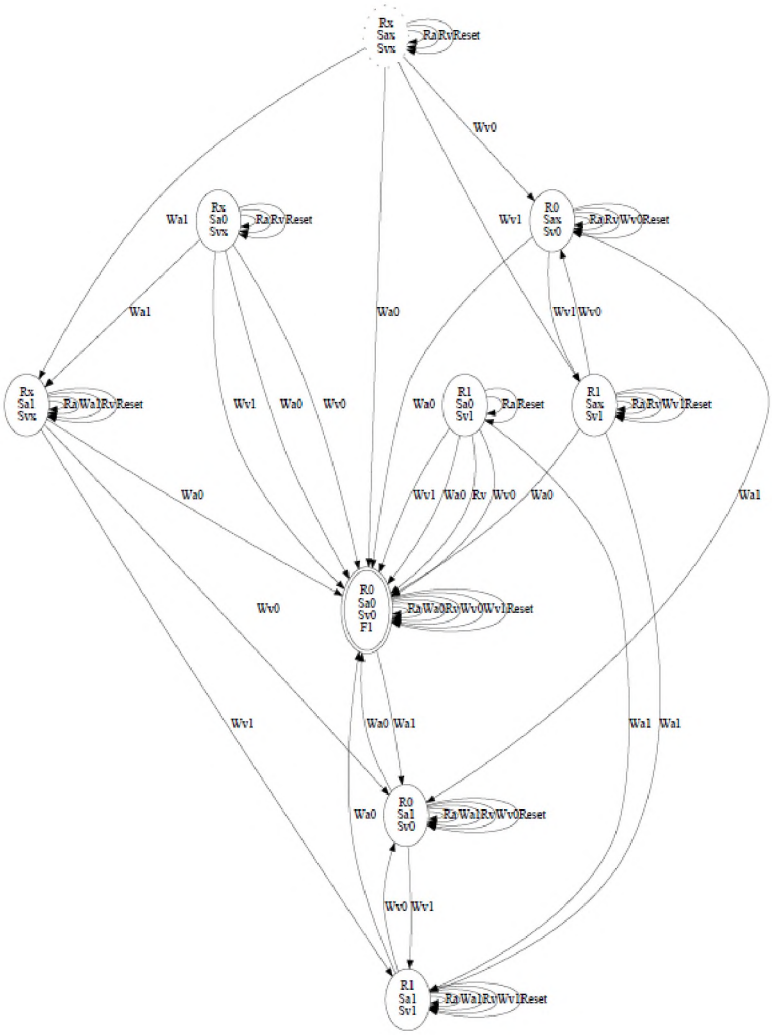


Figure 26. <0;1/0/-> coupling fault visualized with Graphviz

3. If n > 0 and the sequence of fault activating operations is defined on aggressor, then for each O_i operation add a row to FDT as follows:
    a. I_v_V_i_I_v_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX, if i ≠ n,
    b. R_V_i_F_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX if i = n and R is present in <S_a; S_v/F/R> notation,
    c. F_V_i_F_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX if i = n and R is not present in <S_a; S_v/F/-> notation,

Simultaneously transitions at preceding row in FDT are updated for O_i if i > 1 or state-I_aI_v if i = 1 to point the newly added row. Transitions in the newly added row are set to point to the same states as if it was state-V_iI_v for "write", "read", "reset" operations.

4. If n > 0 and the sequence of fault activating operations is defined on victim, then for each O_i operation add a row to FDT as follows:
    d. V_i_I_a_V_i_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX, if i ≠ n,
    e. R_I_a_F_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX if i = n and R is present in <S_a; S_v/F/R> notation,
    f. F_I_a_F_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX_XXXXXX if i = n and R is not present in <S_a; S_v/F/-> notation,

Similarly, simultaneously transitions at preceding row in FDT are updated for O_i if i > 1 or state-I_aI_v if i = 1 to point to the newly added row. Transitions in the newly added row are set to point to the same states as if it was state-I_aV_i for "write", "read", "reset" operations.

5. The transitions in the final row are set in the following way if n > 1, "reset| points to state-S_aF, "read" transition points to the final row if O_n is "read" operation or to state-S_aR based on R of <S_a; S_v/F/R>, "write" transitions point to the same states as if from state-S_aF. If n = 1, then if O_n is "read" operation, then "read" and "reset" transitions are set to state-S_aV_i if V_i = R and R ≠ F, and set to point the final row if V_i = F and R ≠ F.

All the added states, except the final one, are also referred as intermediate states similarly as for single-cell fault model.

2.2.6. Automata model for linked faults

Observed realistic linked faults [82]-[83] present in current memory designs involve at most 3 cells, where FP_1 and FP_2 are either single-cell or coupling faults. Since both FP_1 and FP_2 are present in memory, both faults may be sensitized. With given DFA fault model for FP_1 and FP_2 we may reflect on the idea of unified representation for linked fault.

Such unified representation may be obtained with the application of the union operation [84] on two DFAs A_1 and A_2 defined in the same alphabet:

for given

$$DFA\ A_1 = (Q^1, \sum, \delta^1, q_0^1, F^1),$$

$$DFA\ A_2 = (Q^2, \sum, \delta^2, q_0^2, F^2)$$

union operation

$$A_U = A_1 \cup A_2$$

results in

$$DFA\ A_U = (Q, \sum, \delta, q_0, F)$$

where:

1. $Q = Q^1 \times Q^2$ is a Cartesian product,
2. $\sum$ alphabet is the same as for A_1 and A_2,
3. $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$, where $q^1 \in Q^1$, $q^2 \in Q^2$, $(q^1, q^2) \in Q$, $a \in \sum$,
4. $q_0 = (q_0^1, q_0^2)$,
5. $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

The DFA models A_{FP1} and A_{FP2} proposed for FP_1 and FP_2 are not necessarily defined on the same alphabets. Therefore, some modifications must be made to A_{FP1} and A_{FP2} to comply with the preconditions of DFA union operation.

Without any loss of generality, we can take into consideration the example where FP_1 and FP_2 are coupling faults, which contain different aggressor cells, since DFA model for coupling faults may be viewed as an extension of DFA model for single-cell faults.

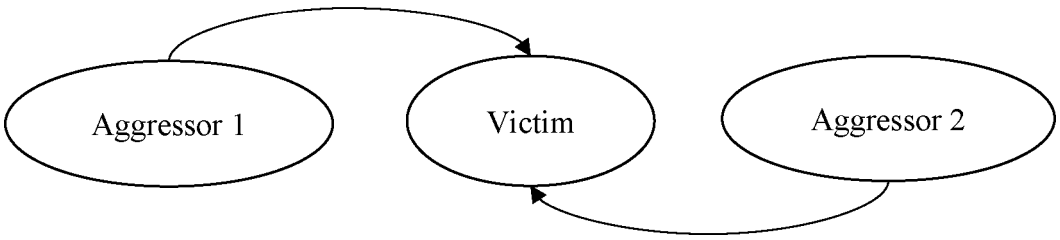


Figure 27. Example a linked fault

The operations on cells should be renamed to include cell identifier, then the following assignment should be done: $\sum_{FP1}, \sum_{FP2} := \sum_{FP1} \cup \sum_{FP2}$. The resulting alphabets will be: $\sum_{FP1} = \sum_{FP2} = \{W_{a1}0, W_{a1}(1), W_{a2}(0), W_{a2}(1), W_v(0), W_v(1), R_{a1}, R_{a2}, R_v, \text{Reset}\}$.

The δ function must be redefined to include newly added operations for each DFA. Let's consider A_{FP1} (the same is true for A_{FP2}):

1. {∀q ∈ Q_{FP1}: q is not intermediate state}, operations {$W_{a2}(0)$, $W_{a2}(1)$, R_{a2}} point to current state. Consequently, it is possible to state, that while FP_1 is not being sensitized, it does not care what operation is applied on the other memory cells.

2. {∀q ∈ Q_{FP1}: q is intermediate state}, operations {$W_{a2}(0)$, $W_{a2}(1)$, R_{a2}} lead to the same state as Reset operation.

After these changes DFA $A_{LF} = A_{FP1} \cup A_{FP2}$ may be constructed using the recently described algorithm.

Furthermore, the state parameters of DFAs can also be modified for further convenience:

1. $Q_{FP1} = q_0 \cup \{(RX, S_{a1}0, S_vX), (RX, S_{a1}1, S_vX), (R1, S_{a1}X, S_v1), (R0, S_{a1}X, S_v0)\}_{initialization\text{-}block} \cup \{q_i := (R\alpha, S_{a1}\beta, S_v\alpha) : \alpha \in \{0, 1\}, \beta \in \{0, 1\}, i = 1, \ldots, n+5\}_{fault\text{-}behavioral\text{-}block} \cup \{q_{fault\text{-}activation\text{-}state} := (R\alpha, S_a\beta, S_v\gamma): \alpha \in \{0, 1\}, \beta \in \{0, 1\}, \gamma \in \{0, 1\}\}$,
2. $Q_{FP2} = q_0 \cup \{(RX, S_{a2}0, S_vX), (RX, S_{a2}1, S_vX), (R1, S_{a2}X, S_v1), (R0, S_{a2}X, S_v0)\}_{initialization\text{-}block} \cup \{q_i := (R\delta, S_{a2}\varepsilon, S_v\delta) : \delta \in \{0, 1\}, \varepsilon \in \{0, 1\}, i = 1, \ldots, n+5\}_{fault\text{-}behavioral\text{-}block} \cup \{q_{fault\text{-}activation\text{-}state} := (R\delta, S_{a2}\varepsilon, S_v\zeta): \delta \in \{0, 1\}, \varepsilon \in \{0, 1\}, \zeta \in \{0, 1\}\}$,

Renaming the parameters will allow us to clarify which victim "read" operation value and victim state should be considered when converted to Mealy SM. An interpretation of the resulting DFA A_{LF} state (Figure 28) should be done as follows:

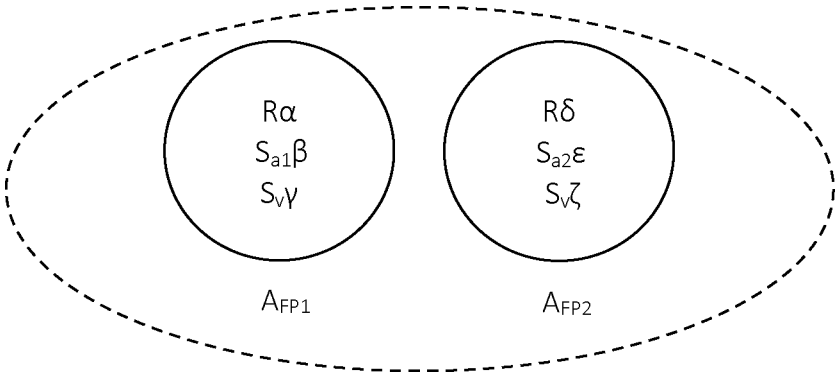


Figure 28. State of DFA model for linked coupling faults

1. Before sensitizing of any of two faults:

a. For non-final states of the A_{LF} the following statement is true based on pre-requirements for linked faults:

$$\alpha = \gamma = \delta = \zeta$$

b. For final states of the A_{LF} three options are possible:

i. A_{FP1} state is final and A_{FP2} is not. S_v and R of A_{FP1} state should be considered.

ii. A_{FP2} state is final and A_{FP1} is not. S_v and R of A_{FP2} state should be considered.

iii. Both A_{FP1} and A_{FP2} states are final. If $\alpha \neq \gamma$, then value of R is X. If $\delta \neq \zeta$, then the value of S_v is X.

2. After either FP_1 or FP_2 has been sensitized, remember the corresponding FP for further transitions:

a. For non-final states of the A_{LF}:

i. If FP_1 was sensitized consider, S_v and R of A_{FP1} state should be considered.

ii. If FP_2 was sensitized consider, S_v and R of A_{FP2} state should be considered.

b. For final states of the A_{LF}, the same case as for 1.b

c. After both FP_1 and FP_2 are sensitized simultaneously than case 1.b.iii applies to current A_{LF} state.

2.2.7. FDT generation procedure for linked faults

FDT for linked faults is generated based on the result of union $A_{LF} = A_{FP1} \cup A_{FP2}$. Besides the victim cell, aggressor cells of A_{LF1} and A_{LF2} may be the same. Therefore, the languages in $\sum_{FP1}$ and $\sum_{FP2}$ intersect. Table 9 and Table 10 demonstrate the example of the FDT for linked coupling faults FP_1 and FP_2, with different aggressor cells, which is currently the case with the most of involved cells. In the example transitions for victim cell and "reset" operations are resulting in one column for both FPs.

Table 9. FDT for two coupling linked faults with different aggressor cells (1)

R_v^1	S_a^1	S_v^1	R_v^2	S_a^2	S_v^2	R_a^1	$W_a^1(0)$	$W_a^1(1)$	R_v	Continued
R^0_1	S_a^0 1	S_v^0 1	R^0 2	S_a^0 2	S_v^0 2	$(\delta^1(q_0^1,$ $R_a^1),$ $\delta^2(q_0^2,$ $R_a^1))$ bin	$(\delta^1(q_0^1,$ $W_a^1(0)),$ $\delta^2(q_0^2,$ $W_a^1(0)))$ bin	$(\delta^1(q_0^1,$ $W_a^1(1)),$ $\delta^2(q_0^2,$ $W_a^1(1)))$ bin	$(\delta^1(q_0^1,$ $R_v),$ $\delta^2(q_0^2,$ $R_v))$ bin	
...	...	...	...	...	...	...	...	...	...	
$R^{i\,1}$	S_a^i 1	S_v^i 1	$R^{j\,2}$	S_a^j 2	S_v^j 2	$(\delta^1(q_0^1,$ $R_a^1),$ $\delta^2(q_0^2,$ $R_a^1))$ bin	$(\delta^1(q_i^1,$ $W_a^1(0)),$ $\delta^2(q_j^2,$ $W_a^1(0)))$ bin	$(\delta^1(q_i^1,$ $W_a^1(1)),$ $\delta^2(q_j^2,$ $W_a^1(1)))$ bin	$(\delta^1(q_i^1,$ $R_v),$ $\delta^2(q_j^2,$ $R_v))$ bin	

Table 10. FDT for two coupling linked faults with different aggressor cells (2)

Continued	$W_v(0)$	$W_v(1)$	R_a^2	$W_a^2(0)$	$W_a^2(1)$	Reset
	$(\delta^1(q_0^1,$ $W_v(0)),$ $\delta^2(q_0^2,$ $W_v(0)))$ bin	$(\delta^1(q_0^1,$ $W_v(1)),$ $\delta^2(q_0^2,$ $W_v(1)))$ bin	$(\delta^1(q_0^1,$ $R_a^2),$ $\delta^2(q_0^2,$ $R_a^2))$ bin	$(\delta^1(q_0^1,$ $W_a^2(0)),$ $\delta^2(q_0^2,$ $W_a^2(0)))$ bin	$(\delta^1(q_0^1,$ $W_a^2(1)),$ $\delta^2(q_0^2,$ $W_a^2(1)))$ bin	$(\delta^1(q_0^1,$ Reset), $\delta^2(q_0^2,$ Reset)) bin
	...	...	...	...	...	...
	$(\delta^1(q_i^1,$ $W_v(0)),$ $\delta^2(q_j^2,$ $W_v(0)))$ bin	$(\delta^1(q_i^1,$ $W_v(1)),$ $\delta^2(q_j^2,$ $W_v(1)))$ bin	$(\delta^1(q_i^1,$ $R_a^2),$ $\delta^2(q_j^2,$ $R_a^2))$ bin	$(\delta^1(q_i^1,$ $W_a^2(0)),$ $\delta^2(q_j^2,$ $W_a^2(0)))$ bin	$(\delta^1(q_i^1,$ $W_a^2(1)),$ $\delta^2(q_j^2,$ $W_a^2(1)))$ bin	$(\delta^1(q_i^1,$ Reset), $\delta^2(q_j^2,$ Reset)) bin

After the application of $A_{LF} = A_{FP1} \cup A_{FP2}$ union operation, when Q_{LF} is obtained it could be sorted considering the final states of A_{FP1} and A_{FP2} that are present in A_{LF}, which results in structured FDT.

Table 11. Structured FDT for linked faults

$q_0 \in Q_{LF}$ initial state
$q \in Q_{LF}$, where $q = (q_{FP1}, q_{FP2})$ $q_{FP1} \notin F_{FP1}$, $q_{FP2} \notin F_{FP2}$
$q \in F_{LF}$, where $q = (q_{FP1}, q_{FP2})$ $q_{FP1} \in F_{FP1}$, $q_{FP2} \notin F_{FP2}$
$q \in F_{LF}$, where $q = (q_{FP1}, q_{FP2})$ $q_{FP1} \notin F_{FP1}$, $q_{FP2} \in F_{FP2}$
$q \in F_{LF}$, where $q = (q_{FP1}, q_{FP2})$ $q_{FP1} \in F_{FP1}$, $q_{FP2} \in F_{FP2}$

The structuration of FDT is made to support further implementation.

2.2.8. Parametrized FDTs for symmetric notations

Since DFA fault model lays on the bases of on <S/F/R> notation it is possible to apply the symmetric property that is outlined via fault notation to the DFA model. FDT generation procedures are modified to work with symmetric notations to generate DFA templates. A template only needs to be filled with exact values during the memory fault injection phase. This set of actions eliminates the need to generate DFA model multiple times for each member of the symmetric group. The FDT can be generated once and values corresponding to each fault may be filled in a separate copy. The FDT generation procedure for single-cell faults can be modified to consider {t,T} variables instead of {0,1} values correspondingly on the basis of the FDT (Table 12).

Table 12. Predefined parametrized FDT for single-cell faults

R^s	S^s	R	W(t)	W(T)	Reset
X	X	0000	0001	0010	0000
t	t	0001	0001	0010	0001
T	T	0010	0001	0010	0010

An example of parametrized FDT for <x/~x/-> symmetric notation is provided below (Table 13).

Table 13. Parametrized FDT for <x/~x/->

R^s	S^s	R	W(t)	W(T)	Reset
X	X	0010	0010	0010	0010
t	t	0001	0001	0010	0001
T	T	0010	0001	0010	0010

Similar to the case of single-cell faults, coupling faults also have the symmetric property. FDT generation procedure can also be modified to work with symmetric notations to generate a DFA templates. The generation procedure can be modified to consider $\{t_1,T_1\}$ and $\{t_2,T_2\}$ variables instead of {0,1} values correspondingly for the aggressor and victim cells on the basis of the FDT, since 0 or 1 value are not used in FDT generation. The predefined FDT for modified flow is presented in Table 14.

Table 14. Predefined parametrized FDT for coupling faults

R^s	S^s_a	S^s_v	R_a	$W_a(t_1)$	$W_a(T_1)$	R_v	$W_v(t_2)$	$W_v(T_2)$	Reset
X	X	X	000000	000001	000010	000000	000011	000100	000000
X	t_1	X	000001	000001	000010	000001	000101	000111	000001
X	T_1	X	000010	000001	000010	000010	000110	001000	000010
t_2	X	t_2	000011	000101	000110	000011	000011	000100	000011
T_2	X	T_2	000100	000111	001000	000100	000011	000100	000100
t_2	t_1	t_2	000101	000101	000110	000101	000101	000111	000101
t_2	T_1	t_2	000110	000101	000110	000110	000110	001000	000110
T_2	t_1	T_2	000111	000111	001000	000111	000101	000111	000111
T_2	T_1	T_2	001000	000111	001000	001000	000110	001000	001000

Consequently, it becomes obvious that the usage of parametrized FDT generation procedure is twice efficient for coupling faults if compared with the one for single-cell faults, due to the reason that one parametrized FDT covers twice more FPs (based on values of aggressor and victim cell).

2.5.1. Extending model for NPSF

Although considered theoretical for now, neighborhood pattern sensitive faults (NPSF) can also be modeled via proposed approach. This paragraph will outline the main aspects of modeling of NPSF DFA model.

FDT generation procedure can be implemented for NPSF with given N (N=5 for type-1, N=9 for type-2):

1. The rows are extended to contain information of N states, and transitions to other states for "read", "write" and "reset" operations,
2. Number of rows of FDT blocks are calculated in the following way 2^N for initial states with all cell values different from X and $(3^N - 2^N)$ for initialization,
3. During parsing phase consider S_N; S_W; S_E; S_S; S_B states or operation sequences for type-1 and four more for type-2 NPSF.
4. Victim cell S_B should be triggerable only when S_N; S_W; S_E; S_S conditions (4 more for type-2 NPSF) are met.

The resulting model has a drawback, which is the increased memory usage with roughly $\approx 3^N$ rows of FDT for given N number of cells during implementation. This is because the initialization block of FDT contains cells with X state, which is further set to 0 or 1, and can't be X again. Generation of such fault models can be time and resource consuming in terms of required memory for storage of the model.

Since the increased memory usage is mainly affected by initialization block, which has a regular representation from fault to fault, and no transitions back to the X states may be made, this block can be replaced with a so called "rewritable state" with some changes made to FDT. The resulting changes will result in FDT which will only partially be DFA state transition table, and requires additional logic to be implemented to be operated.

Let's consider the case of N = 5 for convenience. We can address the states of the corresponding DFA with 8 bits (in case of bigger Ns we should consider more bits). For FDT to comply the symmetry property $\{t_1,T_1\}$, $\{t_2,T_2\}$, $\{t_3,T_3\}$, $\{t_4,T_4\}$ and $\{t_5,T_5\}$ values will be used in template. The first row of our FDT table numbered as 00000000 will be rewritable, therefore, while at least one state from N states is X, the state of DFA stays at 00000000 changing initial part of the row corresponding to the applied operation if necessary, for example if $W_W(T_1)$ operation is applied on W cell, the DFA does not change the state, but XXXXX value in the row is rewritten

with XT_1XXXX. Other rows in predefined parametrized FDT should be modified as shown in table 5. If the first state contains only one X value and "write" operation is applied towards the corresponding cell the appropriate row for transition is determined by the following algorithm:

1. For given $RS_NS_WS_ES_SS_B$ apply the value of the operation to the state, i.e. change corresponding bit for the cell in first row of FDT,
2. Remove first R and add 000: $000S_NS_WS_ES_SS_B$,
3. If $T_1 = 1$, $T_2 = 1$, $T_3 = 1$, $T_4 = 1$, $T_5 = 1$, make transition to $000S_NS_WS_ES_SS_B + 1$ state of FDT,
4. If any of T_1, T_2, T_3, T_4, $T_5 = 0$, reverse corresponding S_N, S_W, S_E, S_S or S_B value and make transition to $S_NS_WS_ES_SS_B + 1$. For example:

   $T_1 = 1$, $T_2 = 0$, $T_3 = 0$, $T_4 = 1$, $T_5 = 1$ and $S_NS_WS_ES_SS_B = 01110$. Reversing S_N and S_S bits corresponding to T_2 and T_3, $S_NS_WS_ES_SS_B = 00010$. Transition should be made to 00000010 + 1 = 00000011 row address. Indeed, if DFA state corresponding to 00000011 is observed as $t_5t_1t_2t_3T_4t_5$ with 0 and 1 values set, the 001110 value is obtained.

To eliminate the issue with state NPSF faults, where the transition to fault activation state is done directly from initialization block, a separate construct can be added that will keep information on transition to fault activation state. Each time operation is applied to "rewritable state" the value of the DFA state should be verified with the added construct. For example, if fault is <0; 0; 1; 0; 0/1/->: than transition to the DFA state with value 00101 must be done, the address of the corresponding FDT row should be calculated as shown in point 4.

NPSF fault types other than state faults, should be modeled by simply adding new rows to FDT, and modifying the transitions from the preceding DFA states as mentioned in the previous fault FDT generation procedures.

Table 15. Predefined parametrized FDT for type-1 NPSF

State numbering	$RS_NS_WS_ES_SS_B$	...
00000000	XXXXXX	...
00000001	$t_1t_2t_3t_4t_5t_1$	...
00000010	$T_1t_2t_3t_4t_5T_1$	...
00000011	$t_1t_2t_3t_4T_5t_1$	...
00000100	$T_1t_2t_3t_4T_5T_1$	...
00000101	$t_1t_2t_3T_4t_5t_1$	...
00000110	$T_1t_2t_3T_4t_5T_1$	...
00000111	$t_1t_2t_3T_4T_5t_1$	...
00001000	$T_1t_2t_3T_4T_5T_1$	...
Continued		

Continued		
00001001	$t_1t_2T_3t_4t_5t_1$	...
00001010	$T_1t_2T_3t_4t_5T_1$	...
00001011	$t_1t_2T_3t_4T_5t_1$	...
00001100	$T_1t_2T_3t_4T_5T_1$	...
00001101	$t_1t_2T_3T_4t_5t_1$	...
00001110	$T_1t_2T_3T_4t_5T_1$	...
00001111	$t_1t_2T_3T_4T_5t_1$	...
00010000	$T_1t_2T_3T_4T_5T_1$	...
00010001	$t_1T_2t_3t_4t_5t_1$	...
00010010	$T_1T_2t_3t_4t_5T_1$	...
00010011	$t_1T_2t_3t_4T_5t_1$	...
00010100	$T_1T_2t_3t_4T_5T_1$	...
00010101	$t_1T_2t_3T_4t_5t_1$	...
00010110	$T_1T_2t_3T_4t_5T_1$	...
00010111	$t_1T_2t_3T_4T_5t_1$	...
00011000	$T_1T_2t_3T_4T_5T_1$	...
00011001	$t_1T_2T_3t_4t_5t_1$	...
00011010	$T_1T_2T_3t_4t_5T_1$	...
00011011	$t_1T_2T_3t_4T_5t_1$	...
00011100	$T_1T_2T_3t_4T_5T_1$	...
00011101	$t_1T_2T_3T_4t_5t_1$	...
00011110	$T_1T_2T_3T_4t_5T_1$	...
00011111	$t_1T_2T_3T_4T_5t_1$	...
00100000	$T_1T_2T_3T_4T_5T_1$	...

2.3. Automated FDT generation flow

This paragraph focuses on the automated FDT generation flow that aims to decrease exhaustion for fault modeling. The main argument is that the flow is based on FPT, thus, considers only known realistic faults.

The flow uses parametrized FDT, which is generated only once for symmetric faults. Since the number of faults cumulated by symmetric notation increases along whit the rise of the number of memory cells affected by the fault, the parametrized FDT can be efficiently used in generation of linked faults. Originated FDTs may be accumulated for further reuse.

The authors proposed simplified notation for single-cell and coupling faults in FPT:

- Simplified notation for single-cell fault <S/F/R> FPs included in $FG_1(x, S)$ sub-group: (x, S)
- Simplified notation for coupling fault $<S_a; S_v/F/R>$ FPs in $FG_2(x, S)$ sub-group: (x, S, y), (y, x, S), where $y \in \{0,1\}$

We extended the simplified notation of single-cell and coupling faults for N-cell faults in order to enable the automated fault injection flow to be extendable. We assumed that only one memory could have a sequence of sensitizing operations since that the mentioned observation is valid for single-cell and coupling faults, and for the discussed theoretical NPSF faults as well:

Simplified notation for n-cell fault $<S_1, \dots S_{n-1}; S_n/F/R>$ FPs in $FG_n(x, S)$ sub-group: $(x, S, y_1, \dots, y_{n-1})$, $(y_1, x, S, \dots, y_{n-1})$,..., $(y_1, \dots, y_{n-1}, x, S)$, where $y_1, \dots, y_n \in \{0,1\}$.

Linked faults $LF = FP_1 \rightarrow FP_2$ may be cumulated in FPT in the following manner:

1. $FG_1(x, S)$ contains $LF = FP_1 \rightarrow FP_2$:
    a. $FP_1 \in FG_1(x, S)$, $FP_2 \in C1$,
2. $FG_2(x, S)$ contains $LF = FP_1 \rightarrow FP_2$:
    a. $FP_1 \in FG_2(x, S)$, $FP_2 \in C1$,
    b. $FP_1 \in FG_2(x, S)$, $FP_2 \in C2$, where aggressor and victim cells of FP_1 and FP_2 are defined to be as the same memory cells,
3. $FG_3(x, S)$ contains $LF = FP_1 \rightarrow FP_2$:
    a. $FP_1 \in FG_2(x, S)$, $FP_2 \in C2$, where only victim cells of FP_1 and FP_2 are defined on the same memory cells,

The resulting groups do not intersect.

Two generation sub-flows are proposed to guide through the generation flow of fault models for linked and unlinked faults while using FPT. Both sub-flows use the interactive procedure of symmetric FP parsing.

The interactive procedure of FP parsing:

1. Input: index j of a cell with sensitizing operation sequence and simplified group (y_1, y_{j-1}, x, S, y_{j+1}, ..., y_{i-1}),
2. Extract FP template < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /F/R >
    a. If the last operation of S is "write":
        i. Resulting FP = < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /~x/- >
    b. If the last operation of S is "read", suggest the following cases:
        i. Is RDF, FP = < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /~x/~x >
        ii. Is DRDF, FP = < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /~x/x >
        iii. Is IRF, FP = < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /x/~x >

The two sub-flows for linked and unlinked faults are presented below:

1. For unlinked faults:
    a. Select a cell FGi(x,S) from one of the {C1, ...,Cn} columns of FPT.
    b. Call interactive procedure of symmetric FP parsing.
    c. Check if already generated parametrized FDT for FP1 → FP¬2 exists.
    d. If parametrized FDT already exists, reuse.
    e. If parametrized FDT does not exist.
        i. Generate FDT and store it.
    f. Initialize parametrized FDT with values.
2. For linked faults:
    a. Select a cell FG_i(x,S) from one of the {C1, ...,Cn} columns of FPT.
    b. Call interactive procedure of symmetric FP_1 parsing.
    c. Select a cell FG_j(x,S) from the suggested list of columns of FPT. The list also excludes non-realistic linked faults.
    d. Call interactive procedure of symmetric FP_2 parsing.
    e. Check if already generated parametrized FDT for FP_1 → FP_2 exists.

f. If parametrized FDT already exists, reuse.

g. If parametrized FDT does not exist.

   i. Apply union operation LF = $FP_1 \cup FP_2$.

   ii. Generate FDT and store it.

h. Initialize parametrized FDT with values.

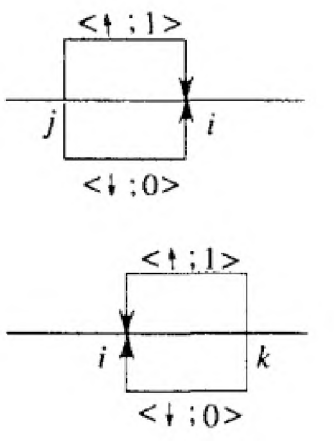


Figure 29. Two idempotent linked faults [82]

The following linked faults are considered non-realistic:

1. Linked faults which include CFins will not be considered since there is no practical evidence of their existence [82]. CFins are theoretical coupling inverse faults that are represented as <↑,↕> or <↓,↕>. If represented with <S/F/R> notation <↑,↕> = <0W1; 0/1/->&<0W1; 1/0/-> and <↓,↕> = <1W0; 0/1/->&<1W0; 1/0/->
2. Linked faults that include two CFids idempotent faults are also considered non-realistic, based on inconsistent physical causes of the faults. Example of such linked faults $<\downarrow,\downarrow>_{ji}$#$<\uparrow,\uparrow>_{ji}$, $<\downarrow,\downarrow>_{ik}$#$<\uparrow,\uparrow>_{ik}$ is shown in Figure 29.
3. Linked faults that include two CFdsts destructive faults are also considered non-realistic, based on the same reasons as for CFins. $<x,\downarrow>_{ji}$#$<y,\uparrow>_{ji}$, $<x,\downarrow>_{ik}$#$<y,\uparrow>_{ik}$ where x,y ∈ {R0, R1, W0, W1}.
4. Some dynamic linked faults are also excluded [83] (Figure 30).
   a. The LF1 are comprised of two single-cell faults.
   b. The $LF2_{av}$ are comprised of coupling and single-cell faults.
   c. The $LF3_{aa}$ are comprised of two coupling faults with the same aggressor cell.
   d. The LF4 are comprised of two coupling faults with different aggressor cells.

The knowledge of non-realistic linked faults can help us further to decrease the space of considered faults.

Subclass	Nonrealistic Faults
LF1	<xWyRy/~y/~y>*<xWyRy/y/~y> <xWyRy/~y/~y>*<xWyRy/~y/y> <xWyRy/y/~y>*<xWyRy/~y/y> <xRxRx/~x/~x>*<xRxRx/x/~x> <xRxRx/~x/~x>*<xRxRx/~x/x> <xRxRx/x/~x>*<xRxRx/~x/x>
$LF2_{av}$	<x; yWzRz/~z/~z>*<yWzRz/z/~z> <x; yWzRz/~z/~z>*<yWzRz/~z/z> <x; yWzRz/z/~z>*<yWzRz/~z/~z> <x; yWzRz/z/~z>*<yWzRz/~z/z> <x; yWzRz/~z/z>*<yWzRz/~z/~z> <x; yWzRz/~z/z>*<yWzRz/z/~z> <x; zRzRz/~z/~z>*<zRzRz/z/~z> <x; zRzRz/~z/~z>*<zRzRz/~z/z> <x; zRzRz/z/~z>*<zRzRz/~z/~z> <x; zRzRz/z/~z>*<zRzRz/~z/z> <x; zRzRz/~z/z>*<zRzRz/~z/~z> <x; zRzRz/~z/z>*<zRzRz/z/~z>
$LF2_{aa}$	<x; yWzRz/~z/~z>*<x; yWzRz/z/~z> <x; yWzRz/~z/~z>*<x; yWzRz/~z/z> <x; yWzRz/z/~z>*<x; yWzRz/~z/z> <x; zRzRz/~z/~z>*<x; zRzRz/z/~z> <x; zRzRz/~z/~z>*<x; zRzRz/~z/z> <x; zRzRz/z/~z>*<x; zRzRz/~z/z>
LF3	<x; yWzRz/~z/~z>*<t; yWzRz/z/~z> <x; yWzRz/~z/~z>*<t; yWzRz/~z/z> <x; yWzRz/z/~z>*<t; yWzRz/~z/z> <x; zRzRz/~z/~z>*<t; zRzRz/z/~z> <x; zRzRz/~z/~z>*<t; zRzRz/~z/z> <x; zRzRz/z/~z>*<t; zRzRz/~z/z>

Figure 30. Non-realistic dynamic linked faults [83]

2.4. Identification of requirements for fault model implementation in HDL representation of MBIST network

Since developers of post-silicon analysis and MBIST solutions generally do not have access to real memory design used by customers. Therefore, the RTL representation of real memory is not always available during the design of MBIST. Instead, an alternative a virtual memory device is used, the behavior of which is almost identical to the original memory design.

Since the fault model cannot be used to replace the memory cell directly, it can be placed outside the memory device while tracking its inputs and altering the outputs.

2.4.1. Fault model placement in RTL

The real RTL for memory is not always available during design of BIST for that memory. A virtual memory device, that behaves close to the original memory is used instead. Therefore, since the fault model cannot be used to replace the memory cell directly, it can be placed near the memory device in the BIST network (Figure 31). Forcing the fault model directly on memory device pins ensures that all components of memory BIST hierarchy have completed their part in memory testing flow before accessing it.

It must be bound to the address of memory cell and alter the memory device outputs if that specific cell is being accessed. For that purpose, the fault model requires a controller that will be responsible for the following functions::

1. Listen to memory pins and ports.
2. Track if changes affect the faulty cell(s).
3. Iterate through FDT.
4. Alter memory device outputs.

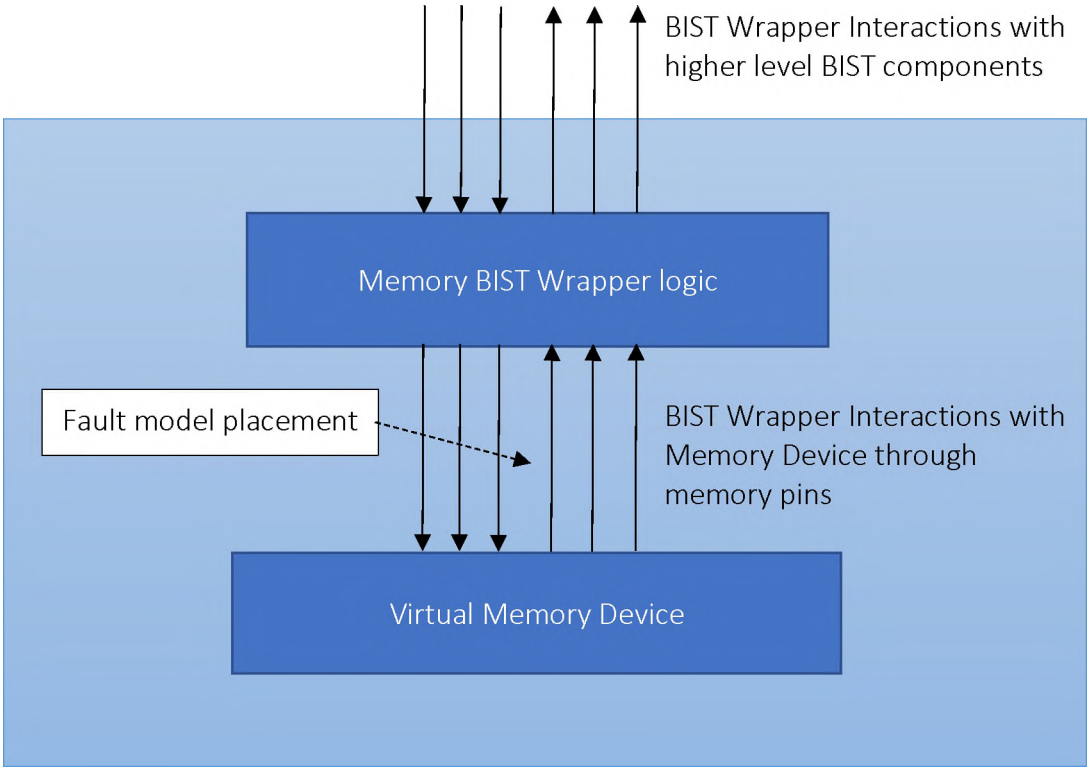


Figure 31. Memory BIST wrapper

The vital component of the fault model injection step includes the precise knowledge of memory pins along with their description. The information can be obtained from memory description file that is also used during the design of memory BIST system when the Virtual Memory Device is created, since the BIST system must also possess the information on memory device pins.

2.4.2. Memory configuration information

The memory configuration information may comprise plenty of data on the memory device that are being utilized by BIST compilers. Nevertheless, the necessary information regarding fault modeling is:

1. Address port.
2. Data input port.
3. Data output port.
4. "Read" operation pin.
5. "Write" operation pin.
6. Information on data scrambling.

7. Information on address scrambling.
8. Memory bank port.
9. Memory operating clock.

An example of memory configuration information without scrambling:

```
MemConfig {
    addr.Direction = "Input"
    addr.Range = "[9:0]"
    addr.Tag = "Address"
    clk.Direction = "Input"
    clk.Tag = "Clock"
    re.ActiveLevel = true
    re.Direction = "Input"
    re.Tag = "ReadEnable"
    din.Direction = "Input"
    din.Range = "[132:0]"
    din.Tag = "Data"
    dout.Direction = "Output"
    dout.Range = "[132:0]"
    dout.Tag = "Data"
}
```

After memory information is obtained fault model controller can be created.

Conclusions

1. An extendable fault model for memory internal faults was proposed.
2. Generation procedures for single-cell, coupling and linked fault models were adduced.
3. Parametrized fault model representation and an automated generation flow for realistic faults were derived based on fault prediction mechanism.
4. Requirements for fault model implementation in RTL HDL representations of MBIST networks are identified.

CHAPTER 3. TEST PATTERN VERIFICATION ENVIRONMENT FOR POST-SILICON ANALYSIS AUTOMATION TOOLS

This chapter will mainly focus on verification of test patterns generated and analyzed via post-silicon analysis automation tools. The test patterns will be considered in terms of test and diagnosis flow implementation. A verification environment will be constructed based on the fault model described in the previous chapter[85].

Firstly, the process of fault injection into memory BIST RTL will be described. Verification environment will be constructed, and its components and flows will be reported. Moreover, the chapter will provide a verification of each phase of the test and diagnosis flow also It will justify the constructed verification environment

Finally, some modifications on test and diagnosis flow implementation will be adduced based on fault prediction mechanism.

All the activity diagrams in this chapter were described in Unified Modeling Language [86] via StarUML [87] software.

3.1. Fault injection

This paragraph describes the process of inclusion of DFA fault model in RTL code. The RTL in SystemVerilog was used since the memory BIST system compilers use Verilog for the generation of components.

3.1.1. Making the fault model traversable

Here FDT generation flow was slightly modified to make the fault model DFA traversable. Transition flag bits were added after each operation transition in FDT for that purpose as shown in (Table 16 and Table 17) that are initialized with 0.

Table 16. Modified coupling fault FDT row

R^s	S^s_a	S^s_v	R_a	F	$W_a(t_1)$	F	$W_a(T_1)$	F	R_v	F	$W_v(t_2)$	F	$W_v(T_2)$	F	Reset	F

Table 17. Modified single-cell fault FDT row

R^s	S^s	R	F	W(t)	F	W(T)	F	Reset	F

The FDT table is being updated during the simulation each time a transition in DFA occurs as described. The transition flag values of FDT may be displayed in the end of the simulation for further analysis.

3.1.2. Fault model controller

Since, the fault model is represented in a form of table FDT, it requires a controller to traverse it (Figure 32). The fault model controller is wired to memory pins. It contains a pointer to the current fault DFA state (FDT row), information on the final states of DFA, information on the memory bank, memory word address, and the bit as the data in memory device is represented by words stored in memory banks.

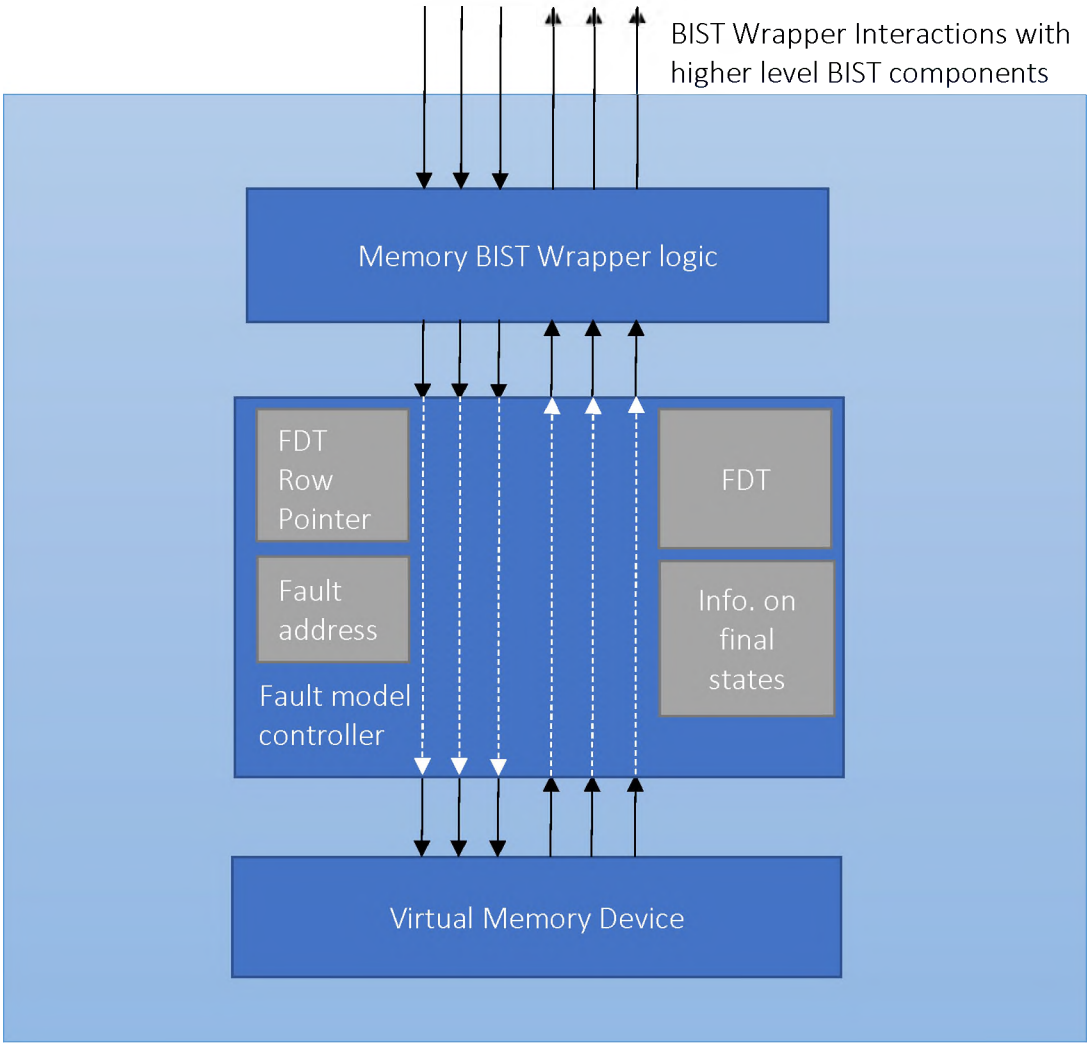


Figure 32. Fault model controller injection

These are the functions of the fault model controller for single-cell and coupling faults:

1. The controller listens to the clock and activates with every clock cycle.

2. The controller traces the value of memory address and bank ports, “write” and “read” pins.
3. If an operation is applied on specified memory fault address(es) in specified banks, fault model controller determines the transition index in FDT table from which is used to allow symmetric FDT generation.
4. Fault model controller makes a transition determined in the previous step to the corresponding FDT state by assigning the value, corresponding to the applied operation obtained from the current row, to the current state pointer, while also setting analogous transition flag to 1.
5. If “read” operation is applied on the memory, then forcing the R value of the current row of FDT on corresponding bit memory data output port.
6. If memory address or bank ports change their value and do not match the fault address(es), then “reset” transition is made on FDT.

Fault model controller for linked faults additionally tracks the final state of which FP was recently sensitized as described in (p. 41). For that purpose, two additional flags are needed, as well as the pointers to edges of final state sections (p. 50) in FDT.

3.1.3. Fault injection information and its processing

Fault injection information file (Figure 33) is provided on the input of fault injection flow. It includes data on memory fault type, memory cell address and the hierarchical path to the memory instance. The gained information is further parsed and processed.

Although other methods for providing the fault injection information may also be considered, the points outlined below are mandatory.

Memory data and address must be considered during fault injection, since memory logical address used during fault injection is not always the same physical address in the memory [11]. This is particularly crucial during the injection of coupling faults, where the two cells require to be placed near each other in a specified order.

```
server{
[processor1 [1]]
  MEMORY [1]
    //   Address in range: [1023 : 0], Virtual banks = 1, Number of Bits = 133

    single cell faults:
    ADDR = 3, BIT = 2, VBK = 1, FTYPE = <R0/0/1>
    ADDR = 2, BIT = 1, VBK = 1, FTYPE = <0/1/->

    coupling faults:
    ADDR_A = 836, BIT_A = 5, VBK_A = 1, ADDR_V = 837, BIT_V = 5, VBK_V = 1, FTYPE = <1W1;0/1/->

[processor2 [2]]
  MEMORY2 [1]
    //   Address in range: [4991 : 0], Virtual banks = 1, Number of Bits = 61

    linked faults:
    ADDR_A = 843, BIT_A = 3, VBK_A = 1, ADDR_V = 840, BIT_V = 4, VBK_V = 1, FTYPE_L1 = <0R0R0;1/0/->
    ADDR_A = 842, BIT_A = 3, VBK_A = 1, ADDR_V = 840, BIT_V = 4, VBK_V = 1, FTYPE_L2 = <1;1R1R1/0/1>

  MEMORY2 [2]
    //   Address in range: [4991 : 0], Virtual banks = 1, Number of Bits = 61
}
```

Figure 33. Fault injection information file example

3.1.4. Implementation in SystemVerilog

Usage of SystemVerilog [88] provides better flexibility with multidimensional arrays. Since FDT is organized as two-dimensional array, which needs to be dynamically accessed during runtime, and MBIST network RTL is presented in Verilog compatible with SystemVerilog, this choice looks reasonable.

Fault model controller either for single or coupling faults in SystemVerilog consists of the following elements:

1. FDT.
2. Pointer to current state (an index for FDT row).
3. “Read” flag that activates if “read” operation was applied, this flag is necessary for modifying memory pins on the next clock cycle, when the corresponding data output is normally expected.
4. @always block for fault state transition, listening to memory control pins, determining the transition based on operation and making transitions in FDT, setting “read” flag on “read” operation, and setting a transition flag for the corresponding transition in the current row to 1. If an operation is applied on a different address “reset” transition should be called with corresponding transition flag set.

5. @always block for handling “read” operation if the “read” flag is set. The memory data output pin corresponding to the faulty bit is assigned with the “read” value of the active state.

The fault model controller for linked faults additionally requires the following components:

1. Two flags with the purpose of tracking final state of which FP was activated.
2. Three pairs of pointers to the edges of the final state sections of structured FDT.
3. @always block for fault state transition should also track if one or both final states of two FPs of LF were reached. If this is the case, corresponding flags should be set to zero; otherwise, the flags would be set to 0.

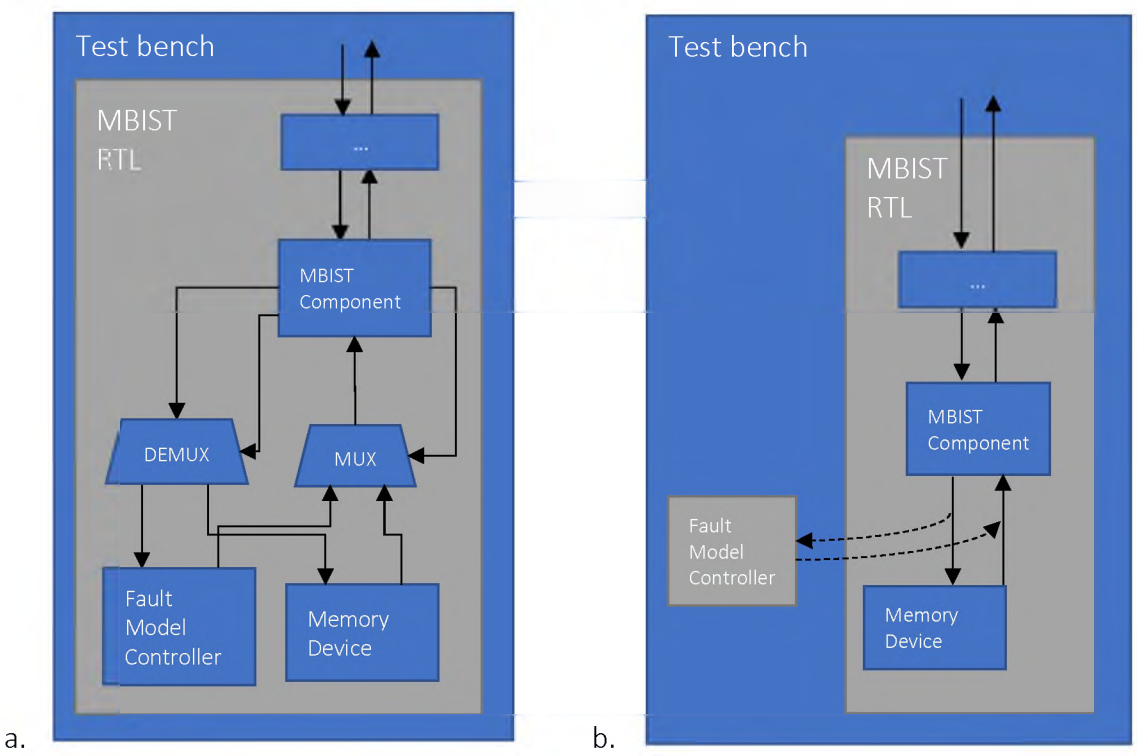


Figure 34. Considered two methods of controller injection in Verilog a) inside the MBIST network RTL b) outside MBIST network RTL

4. @always block for handling “read” operation should consider final state flags and alter the memory data output as described during the FDT generation procedure for linked faults (p. 53).

These two options were considered for fault model controller inclusion in MBIST network RTL:

1. Placing the fault model controller between the memory device and connected the MBIST network component using multiplexer and demultiplexer (Figure 34 a).
2. Placing the fault model controller outside the MBIST network and forcing the value on memory outputs externally using "force" and "release" operations (Figure 34 b).

The first method requires modification of MBIST network RTL HDL files for injection of fault model controller with rewiring of the memory and MBIST network component inputs and outputs.

The second method though injects the fault controller into the test bench and does not alter the existing RTL HDL files. On each positive edge of the test clock "release" operation is applied to the memory output. If the "read" flag was set "force" operation alters the output data of memory with a short delay and unsets the "read" flag. The delay may be calculated based on the test clock cycle period that can be obtained from the MBIST network configuration data.

In this research we followed the second approach that has been described above.

3.1.5. Implementation of fault injection flow

Finally, as all the components of fault injection have been defined, the fault injection flow may be implemented (Figure 35).

The flow consists of 10 steps:

1. Firstly, FPT is provided at input.
2. Automated parametrized FDT generation flow is used to create and store parametrized FDTs on the basis, of FPT for further reuse.
3. Parametrized FDTs may be stored in file system or any other convenient database.
4. Fault injection information is provided in a form of fault injection data file that contains information on fault types, memory instances and their hierarchical paths, fault injection addresses.
5. An input data list is constructed based on the provided data.

For example:

({(0,3,2)}, <R0/0/1>, ”server.processor1 [1].MEMORY [1]”),

({(0,2,1)}, <0/1/->, ”server.processor1 [1].MEMORY [1]”),

({(0,836,5), (0,840,5)}, <1W1;0/1/-> ,”server.processor1 [1].MEMORY [1]”),

({(0,843,3), (0,840,4), (0,842,3), (0,840,4)}, {<1W1;0/1/->, <1;1R1R1/0/1>},

”server.processor2 [2].MEMORY2 [1]”).

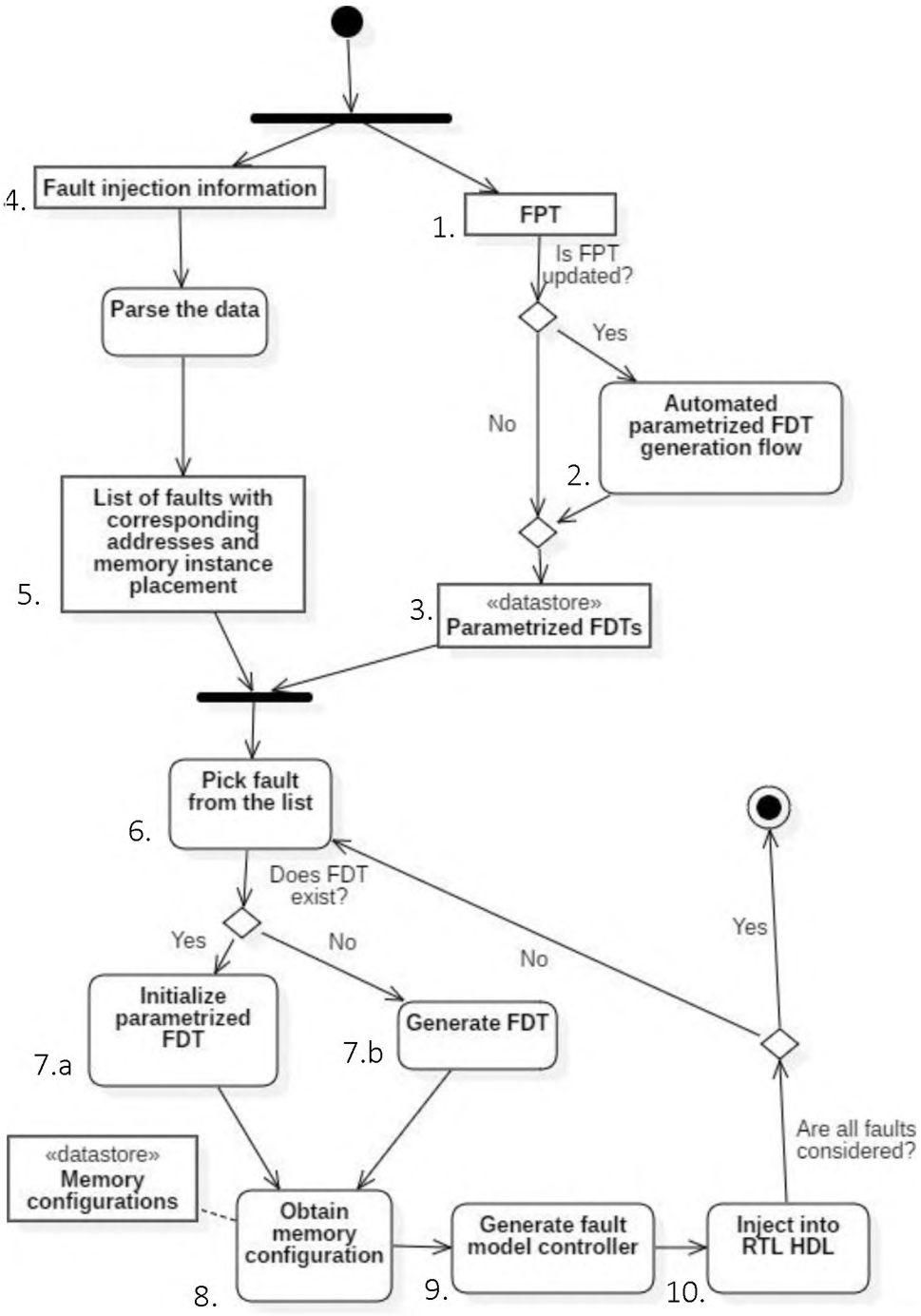


Figure 35. Fault injection flow implementation

6. For each fault from the input data list its symmetric notation is obtained. Since each fault may be described with various forms of symmetric notation (e.g. <0/1/-> may be described as <x/~x/-> or <~x/x/->) a standardized approach should be used in this step. Symmetric notation for FPs may be represented via either of the following formats:
   a. FP = < y_1; y_{j-1}; xS; y_{j+1}; ...; y_{i-1} /F/R >
   b. FP = < y_1; ...; y_{i-1}; xS /F/R >

   In case a. values of F and R are represented via y_{i-1}. In case b. values of F and R are represented via x.

   Additionally, the fault injection addresses should also be considered for linked faults, as they may share the same aggressor cell or be defined on different ones. Therefore, they are described via different FDTs.
7. After symmetric notation is obtained it is searched among parametrized FDTs. Based on the data cumulated as a result of the search two cases are possible:
   a. If there is an existing parametrized FDT, values should be parsed from fault notation and the FDT should be initialized. For instance, for the given fault notation <0W1; 0/1/-> the {t_1,T_1} and {t_2,T_2} variables should be initialized in the following manner: T_1 = 0, t_1 = 1, T_2 = 0, t_2 = 1. A mapping should be constructed to be passed to the fault controller generation phase (Figure 36). Moreover, it should indicate which FDT columns are used to describe the transition operation.

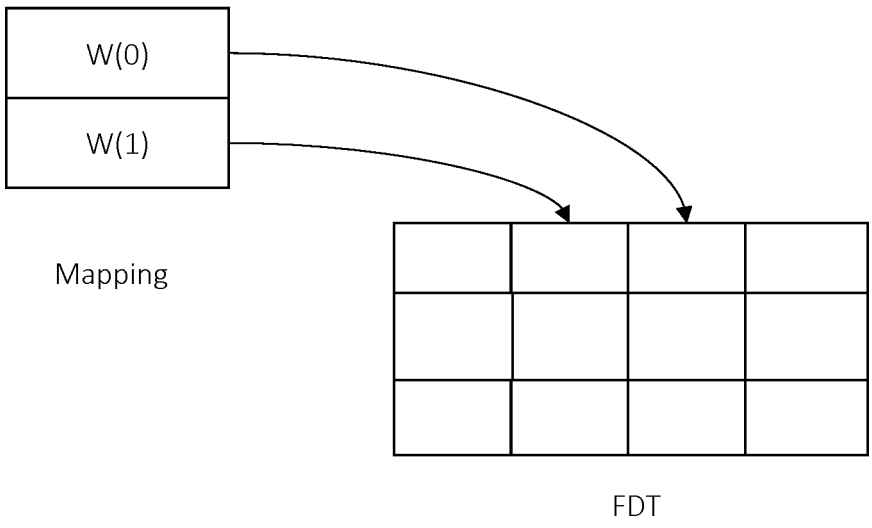


Figure 36. Transition mapping used for fault controller generation

b. If a parametrized FDT does not exist than FDT for the provided fault is generated. One of the adduced generation procedures is used for single-cell, coupling or linked faults FDT.

8. Memory configuration information is provided for each memory in the MBIST network. The memory port and pin names are determined for further wiring with the fault model controller. The corresponding memory configuration information can be obtained for each memory instance and used for further generation of fault model controller.
9. Create an empty file for inclusion in RTL. For each fault:
   a. Generate a fault controller using fault injection and memory configuration information, the mapping for operation transitions and the FDT.
   b. Generate System Verilog representation of a fault model controller and include it in the file.
10. Incorporate the data into the test bench.

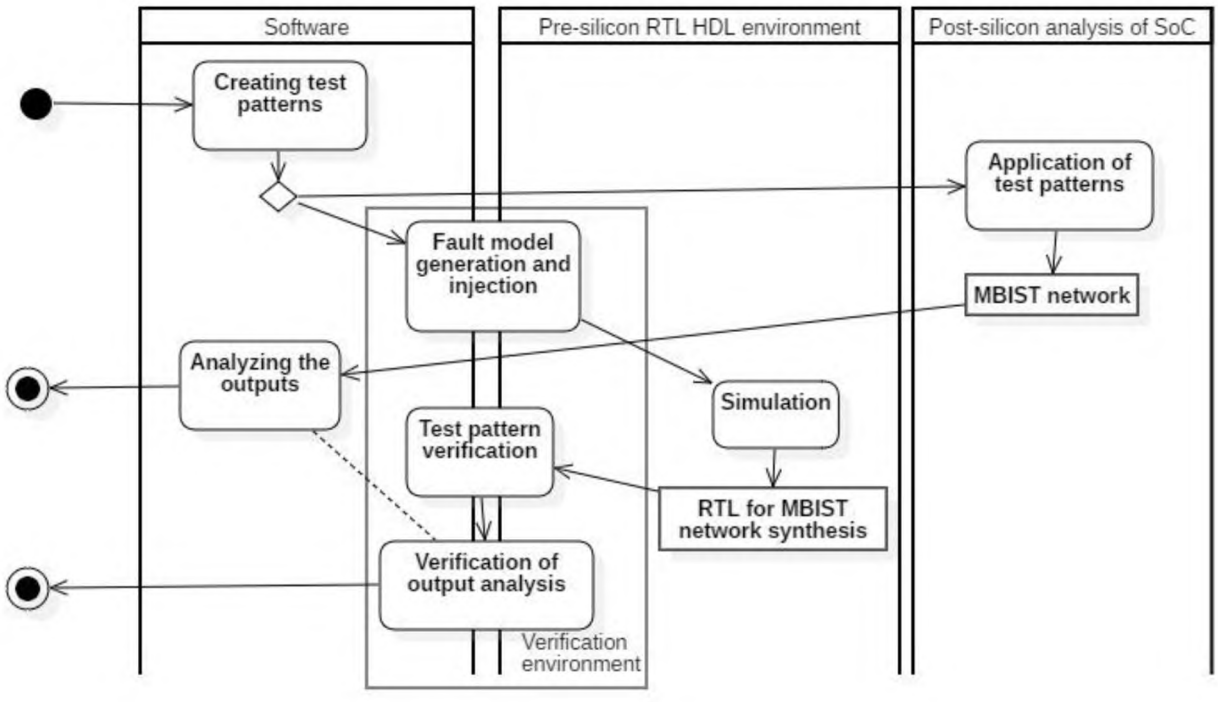


Figure 37. Verification flow for software post-silicon analysis automation tool.

3.2. Verification environment for test patterns in manufacturing tools.

Using the fault injection flow described in the previous paragraph and MBSIT RTL a verification environment can be constructed and used for the verification of software post-silicon analysis automation tool (Figure 37).

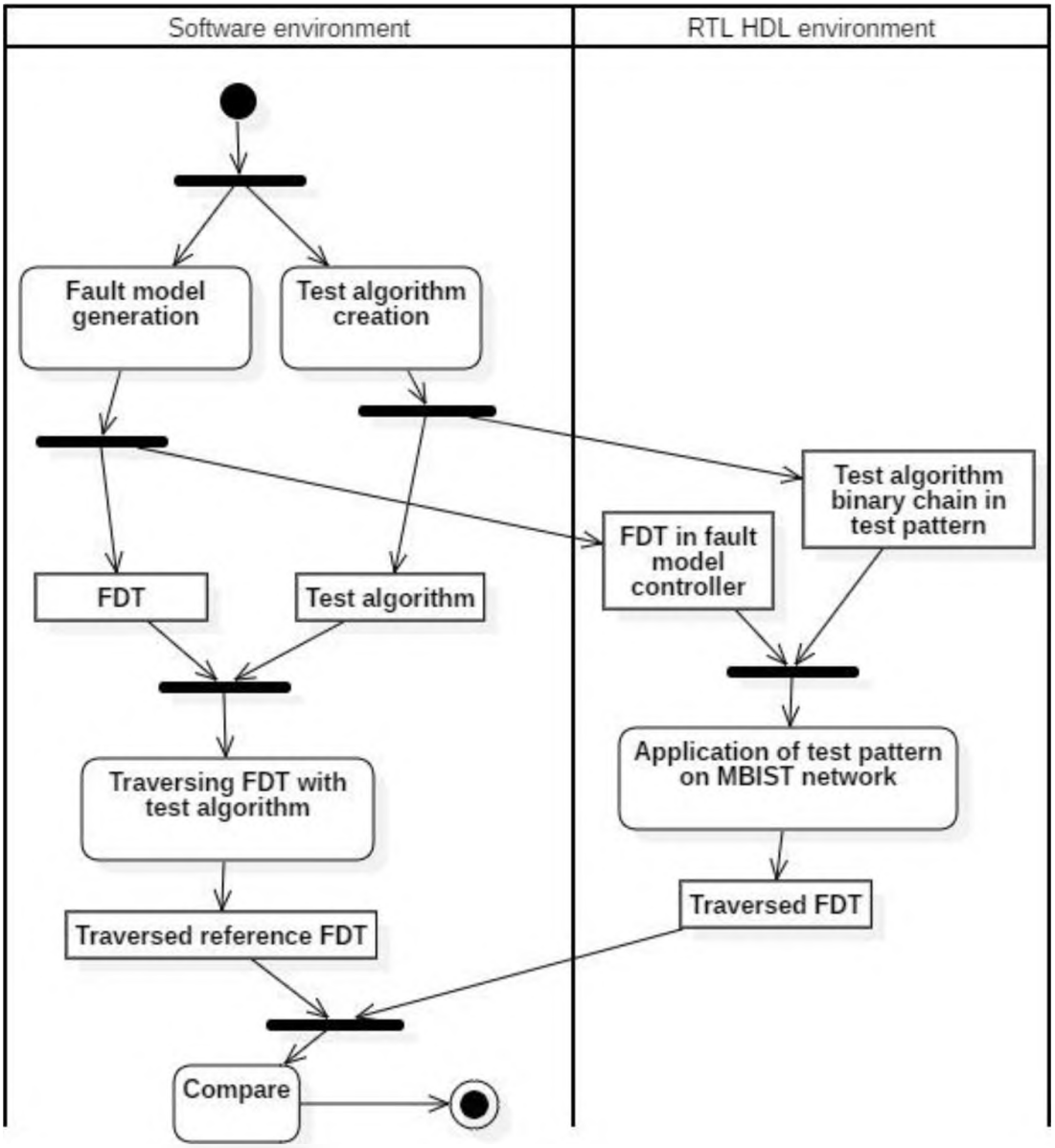


Figure 38. Verification of test patterns for user defined test algorithms

3.2.1. Verification of test patterns for user defined test algorithms

Considering the fact that test patterns can be used for instructing MBISTs to run user defined March test algorithms (March-like test algorithms) on the memory devices, an approach of determining whether the test pattern successfully executed or not is to observe the behavior of fault model injected into the MBIST network.

The fault model is traversable through the transition flags in FDT representation. Each time a transition from a state occurs the transition flag of current FDT row for corresponding operation is set to 1. After the simulation is over the resulting FDT can be compared to a reference FDT (Figure 38).

Comparison of two separately traversed FDTs aims to verify two things:

a. If the test algorithm chain was correctly generated in the software tool.
b. If the test algorithm was applied to the intended memory core.

Reference FDT is created on the basis of generated fault model FDT, traversed separately via the March test algorithm that is used in the test pattern.

3.2.2. FDT tracing software tool

For simulating the execution of the March test algorithms on FDT a simple software tool is implemented as part of the mature tool, that takes FDT, March test algorithm and traverses the FDT while applying the algorithm. The logical representation of test algorithm is used in the tool, i.e. operations are considered instead of their binary codes. Traversed FDT was used as a reference FDT in the previous paragraph. The tracing of FDT for single-cell and coupling faults in the software is done in the following way:

1. If FDT corresponds to single-cell fault model, typically make a transition on each operation of March element and mark the transition flag, if March element changes Reset transition must be made.
2. If FDT corresponds to coupling fault model, the relative placement of aggressor *a* and victim *v* cells needs to be considered. This information is based on fault injection addresses. On each March element consider two cases:
    a. $a < v$, apply the operations first to the aggressor than to victim, if addressing is ascending, in the opposite sequence if descending,
    b. $a > v$, apply the operations first to aggressor than to victim, if addressing is decreasing, in reverse sequence if ascending.

Consider Reset transition for each cell similarly to a single-cell fault.

The traversed FDTs are stored for each fault injected into the MBIST network RTL HDL and are compared on conformity with the FDTs traversed via test algorithms by MBISTs.

Example of traversed FinFET-specific fault graph is provided in Figure 39.

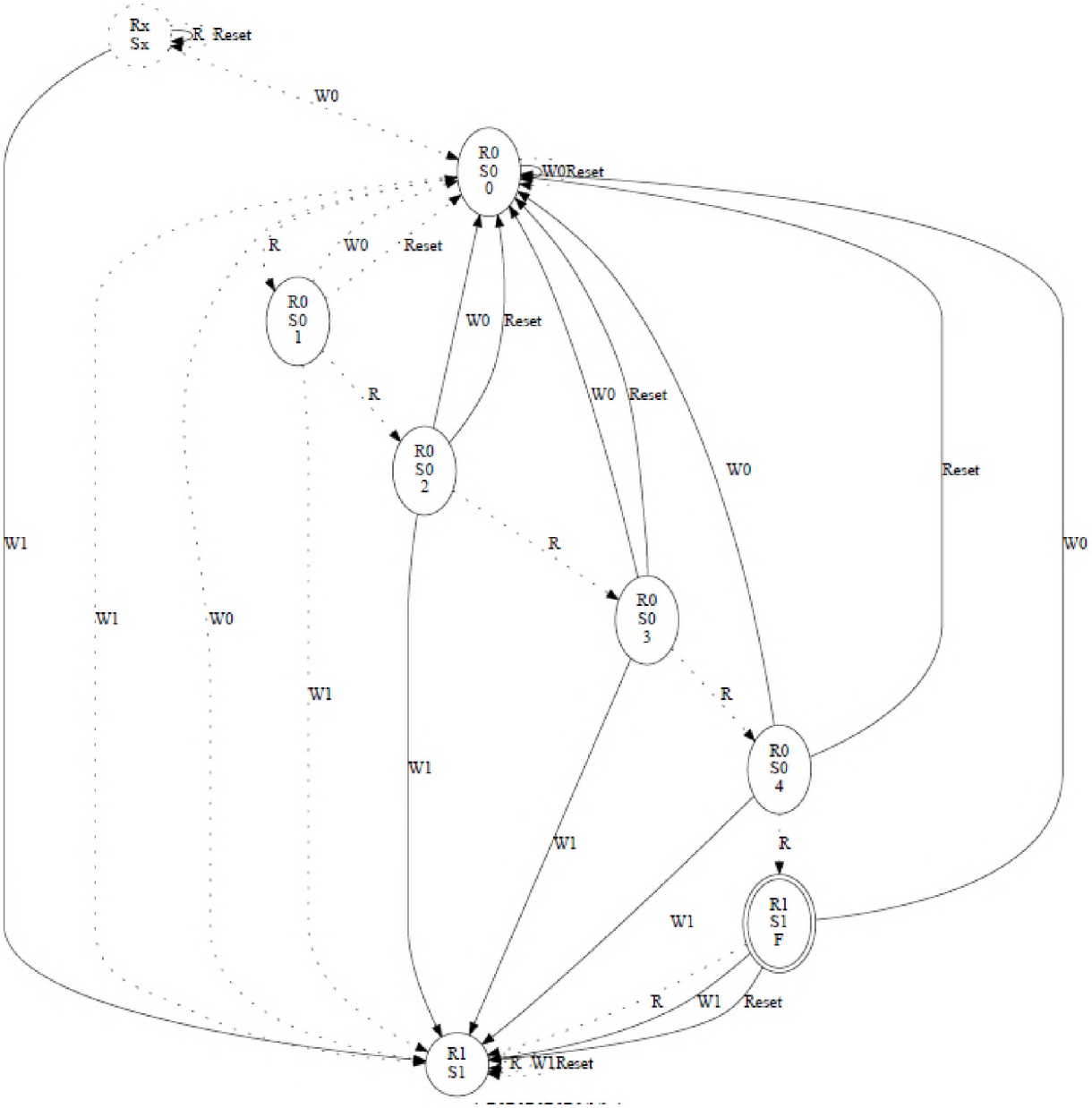


Figure 39. <R0R0R0R0R0/1/1> fault traversed by FFDD and visualized with Graphviz

3.2.3. Verification of chain analysis

As each test is used for a specific purpose, the chain analysis is also unique for each of them. Nevertheless, if the chain analysis is related to test failure, a reference chain may be constructed on the base of the fault injection information file. The reference chain is compared to the resulting output chain after test pattern input chain is applied on MBIST network RTL (Figure 40).

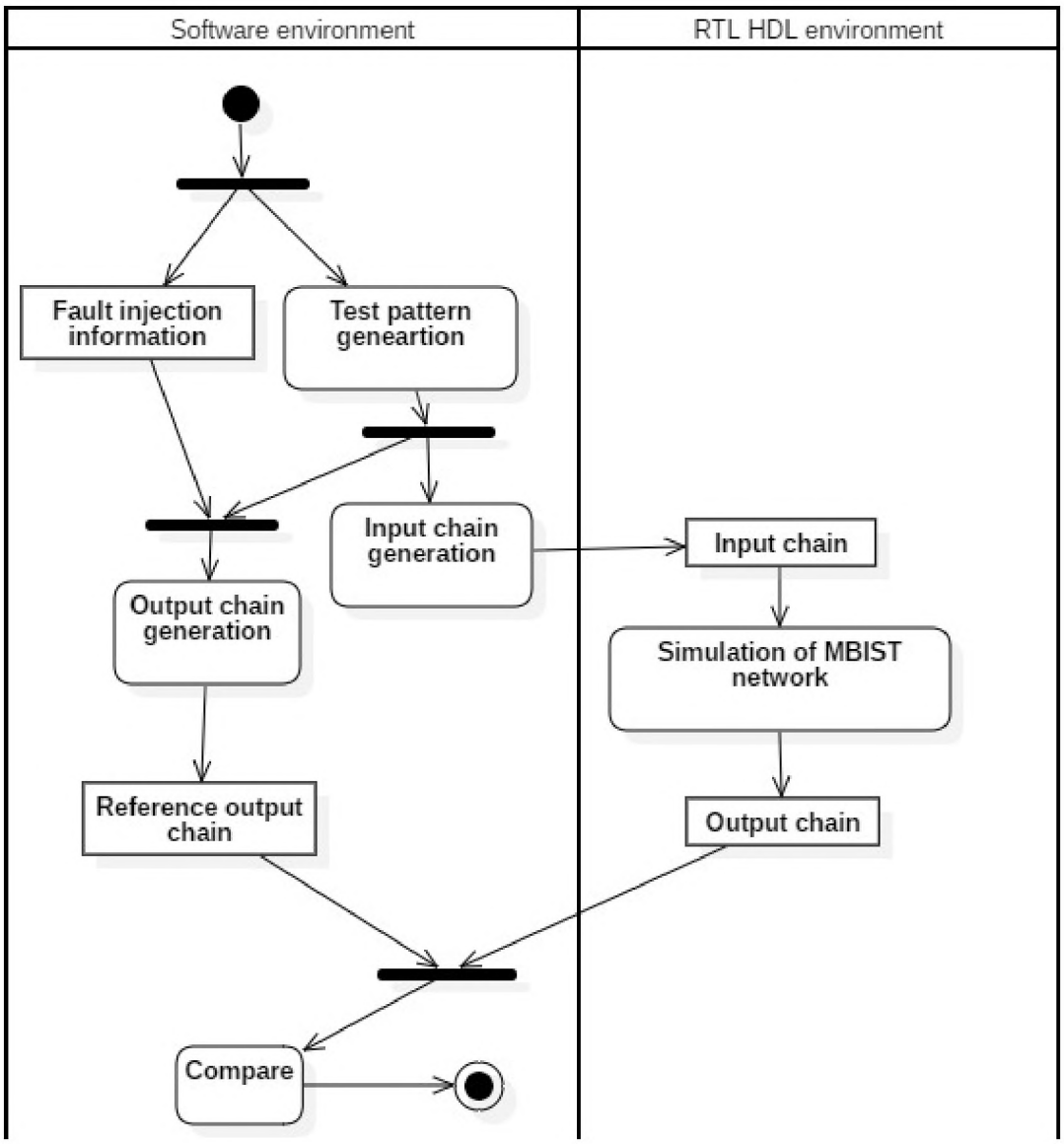


Figure 40. Verification of chain analysis

This verification procedure is individual for each phase of test and diagnosis flow, since each phase is specific, i.e. in the detection phase output chain contains only information on memories that failed/passed the test, chains for diagnosis phase additionally contain binary information on the failed memory address, bit and march test operation.

3.3. Verification of test and diagnosis flow implementation

Verification of test and diagnosis flow [5] was considered in this work. Since this multiphase flow is used in post-silicon analysis and the creation of a test pattern in each phase depends on the results of test pattern execution in the previous phase, verification of this flow implementation is essential.

3.3.1. Verification of detection phase

Test pattern template generally used in this phase looks like:

SELECT_MEMORY_GROUP

LOAD_TEST_ALGORITHM [TEST_ALGORITHM]

RUN_BIST

READ_FAILED_MEMORY_INFO

Here the first input chain selects a group of memories that are going to be tested. With the second input chain a test algorithm is loaded into MBISTs for further execution. The third chain commands MBISTs to run the tests. Information is obtained from MBIST network with the last chain and is further analyzed by the manufacturing tool on which memories have failed.

Verification of the created test pattern is done in the following way:

a. Choosing the faults that are covered by the test algorithm used in the sample,
b. Injecting the faults on various memories,
c. Traversing the faults as shown at p. 78.

Verification of the output chain for the reported failed memories is based on the conformity with memories used for fault injection in fault injection information file (Figure 41). Furthermore, to verify that test algorithms binary code used in the input chain has been correctly generated and if the test algorithms have reached the intended memory cores, the FDTs of faults provided with fault injection information are separately traversed via software tool and compared to the traversed FDTs reported in the end of MBIST network RTL simulation.

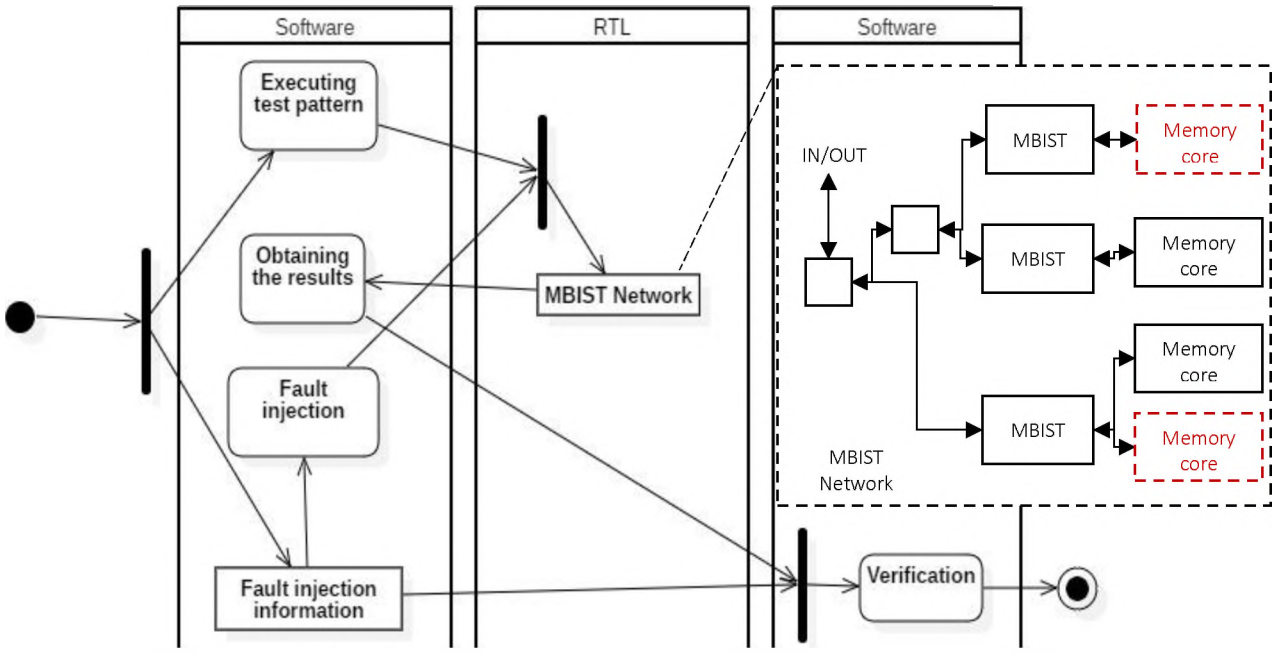


Figure 41. Detection of failed memory cores

3.3.2. Verification of fault localization phase

Test pattern template used in this phase is similar to the one used in the detection phase:

SELECT_MEMORY_GROUP

LOAD_TEST_ALGORITHM [TEST_ALGORITHM]

SET_SONE [SONE_VALUE]

RUN_BIST

READ_DIAGNOSTIC_INFORMATION

The difference is in the additional input chain that sets the value of "stop on n-th error" register. Based on that value MBIST will stop the execution of the March test algorithm if the n-th error occurs. Finally, the information on memory failure is obtained and reported in manufacturing tool.

The reported information contains: memory failed bank, memory failed address, memory failed bit, failed March test element and operation.

SONE_VALUE is set to 1 to report information on the first encountered incorrect "read" operation of the test algorithm. If multiple faults are injected in a single memory core, then the SONE_VALUE can be incremented till all the faults have been located.

Verification of the reported diagnosis information output chain is based on the conformity with banks, addresses and bits provided for the fault injection in corresponding special information file. The approach is alike the one used in the detection phase.

3.3.3. Verification of fault classification phase

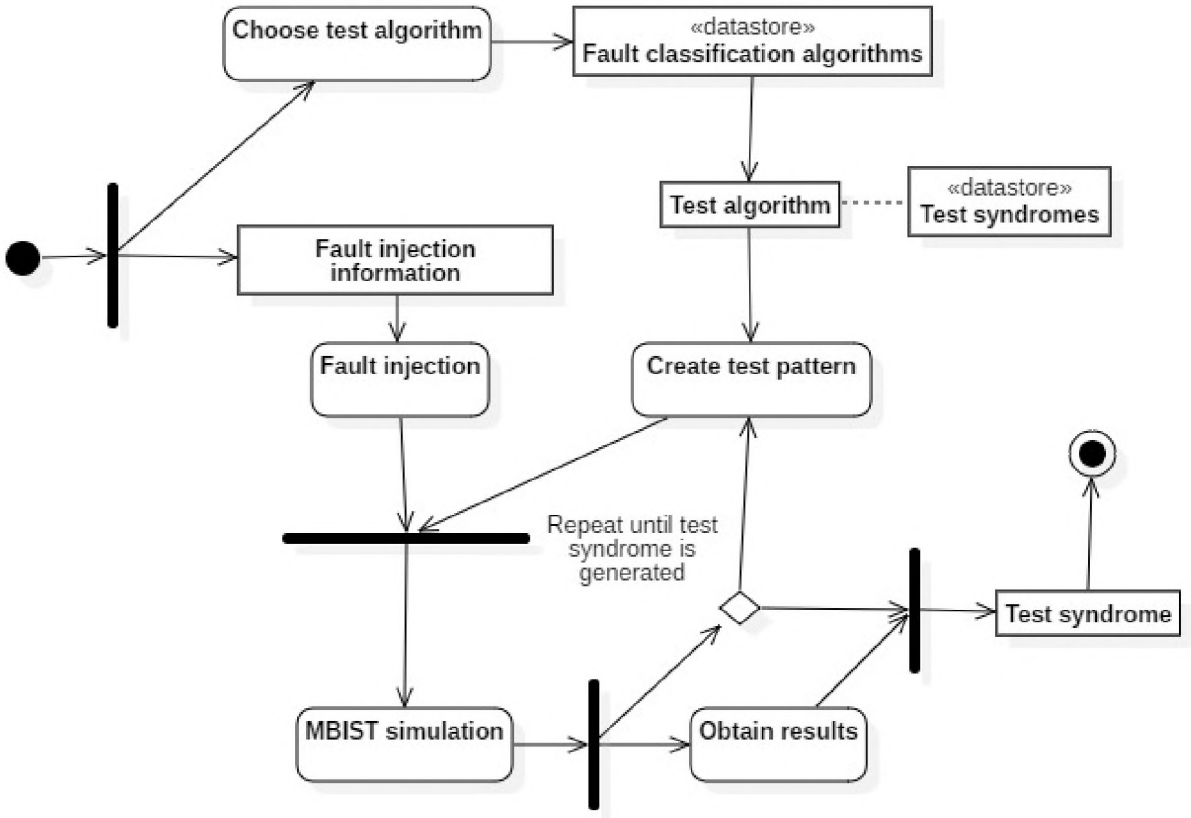


Figure 42. Fault classification flow

Assuming the pattern described in the previous phase was verified, it may be used in classification phase. At this point, we also presume that a fault was detected, localized and needs to be classified.

Test patterns are created, while using fault classification March test algorithms respectively designed for this diagnosis phase. Furthermore, fault classification phase can leverage the SONE register. n number of pattern executions need to be made while modifying the value of the SONE register to ensure that n "read" operations have been applied on the

faulty cell, where n is the number of "read" operations present in test algorithm. Test syndrome is being generated as a result, which is the n-bit signature, where the order of bits corresponds to the sequence of "read" operations in the test algorithm. The faults are classified based on the obtained signature using the signature dictionary which must be provided with classification test algorithms.

Classification algorithms aim to cover a wide range of faults and distinguish them. As a result, unique signature (test syndrome) should be generated for each failure covered by the algorithm at the end of the classification phase.

Verification of this phase is done by checking the conformity of classified fault type with FP provided in fault injection information file. The flow is described in Figure 42.

3.4.4. Verification of fault localization for aggressor cells

The test pattern template for this phase is as follows:

SELECT_MEMORY_GROUP

APPLY_MARCH_LIKE_TEST_ALGORITHM [TEST_ALGORITHM]

READ_DIAGNOSTIC_INFORMATION

The test pattern in this phase is generally used with coupling faults for locating the aggressor cell. We assume that the victim cell was ascertained in the detection phase, while the fault type was determined at the classification phase. An adaptive March-like algorithm is being applied to the neighborhood of a memory victim cell (Figure 43). The idea of the algorithm is to:

1. Set potential aggressor cells to the state opposite to the activation state of aggressor for the coupling fault.
2. Apply sequence of fault activating operations to the potential aggressor cell or victim cell depending on the FP.
3. Read the value of the victim cell.
4. If the fault was not triggered then repeat steps 1,2,3 for the next potential aggressor.
5. The algorithm stops when the faulty value is finally observed on the victim cell.

2	3	4
1	V	5
8	7	6

Figure 43. Victim cell V with potential aggressors 1, 2, 3, 4, 5, 6, 7, 8

Post-silicon analysis automation tool further generates a report on failed memory aggressor cell based on information on failed March test element and test operation.

The reported aggressor cell address, bit and bank are verified in compliance with fault injection information for coupling fault.

In some cases, if application of March-like test algorithms is not feasible, but the memory scrambling information is present, another approach of aggressor cell allocation may be applied. The approach is based on serial execution of each operation on potential aggressor and the victim memory cells in the same way as describe above. As long as the execution of operations is done manually via memory logical addresses, an additional initial step is required for determination of all the potential aggressor cell logical addresses for the given victim cell, which logical address was obtained during the previous phases of diagnosis flow. This approach cannot be applied though on dynamic faults, since they require at-speed execution and the serial access to memory cells is made through SoC inputs/outputs usually at much slower clock speed.

3.5. Justification of verification environment

In this paragraph we will show how the verification environment was justified. It was proven on 600 randomly generated MBIST networks created via Synopsys DesignWare STAR Memory System line of products. The networks were generated based on 760 different memory configurations. Two types of network hierarchies were considered as shown in Figure 44, conditionally referred as a. One-level and b. Two-level hierarchies. Components of the network are connected in terms of IEEE1500 standard.

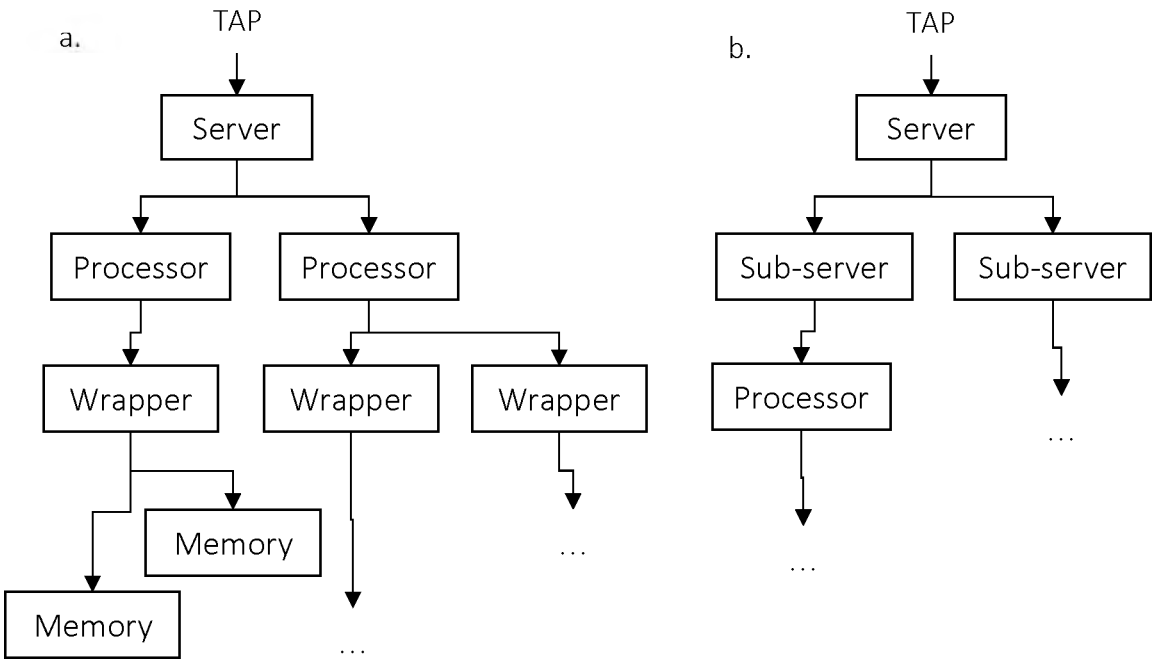


Figure 44. Considered types of MBIST network hierarchies: a. one-level b. two-level

Parameters of the networks were randomly picked from the range shown in Table 18. Networks with 0 sub-servers correspond to one-level network hierarchies, while the ones with 1+ sub-servers correspond to two-level hierarchies. The upper bound of the total number of components used in the system during the simulations was set to 500. Different memory configurations were also considered Table 19.

MBIST networks were generated once. Further simulations with or without fault injection were made on the same projects. Generation of the projects took approximately 12 hours.

Table 18. MBIST network configurations

Network Components	Minimal configuration	Maximal configuration
Sub-servers	0	7
Processors under each (sub-)server	1	30
Wrappers under each processor	1	30
Memories under each wrapper	1	30

Table 19. Memory configurations

Parameter	Minimal configuration	Maximal configuration
Address scrambling	No	Yes
Number of words	16	32000
Number of bits in word	4	256
Number of banks	1	4
Column multiplexing	1	64

March FD(35N), VLP1(26N), VLP2(26N), VLP3(22N), FFDD(42N) test algorithms [5] were chosen, since the first algorithm covers all simple static faults, VLP1-VLP3 cover all static and two-operation faults, FFDD covers all FinFET-specific dynamic faults.

For each network, three test patterns were created:

1. SELECT_MEMORY

   LOAD_TEST_ALGORITHM FD

   RUN_BIST

   READ_FAILED_MEMORY_INFO,
2. SELECT_MEMORY

   LOAD_TEST_ALGORITHM VLP1

   RUN_BIST

   READ_FAILED_MEMORY_INFO

   LOAD_TEST_ALGORITHM VLP2

   RUN_BIST

   READ_FAILED_MEMORY_INFO

LOAD_TEST_ALGORITHM VLP3

RUN_BIST

READ_FAILED_MEMORY_INFO,

3. SELECT_MEMORY

LOAD_TEST_ALGORITHM FFDD

RUN_BIST

READ_FAILED_MEMORY_INFO

For construction of the following test pattern for each network a March LSD (75N) [83] test algorithm was used to justify the environment for linked fault models:

SELECT_MEMORY

LOAD_TEST_ALGORITHM LSD

RUN_BIST

READ_FAILED_MEMORY_INFO

Corresponding single-cell, coupling and linked faults from FPT (C1, C2, C3 columns) were automatically modeled for each test pattern and injected into the test benches. All injected faults were detected, traversed and compared with separately traversed FDTs in software tool (p. 78).

Furthermore, the abovementioned test algorithms for single-cell and coupling faults were used for verification of fault localization and classification step in the matching test pattern (p. 82). Simulations were done with one fault injected into each memory core. The faults were successfully classified.

Aggressor cell localization flow was verified using simple static faults using March-like LD1(2N+O(1)), LD2(41N+O(1)) test algorithms [41].

Simulations on the projects were run with and without fault injection on 350 dedicated machines and took approximately 48 hours in both cases; consequently, the fault model has not affected the simulation time in any noticeable way.

3.6. Modifications of test and diagnosis flow implementation and introduction of interactive fault injection flow

Particular faults require corresponding test algorithms to be used in test patters during test and diagnosis flow. Furthermore, peculiar test algorithms/groups of test algorithms may be developed for coverage of a group of faults as seen in paragraph 3.4. To eliminate the exhaustion during test algorithm development while considering only realistic faults, a modification to test and diagnosis flow is applied to leverage the fault prediction mechanism.

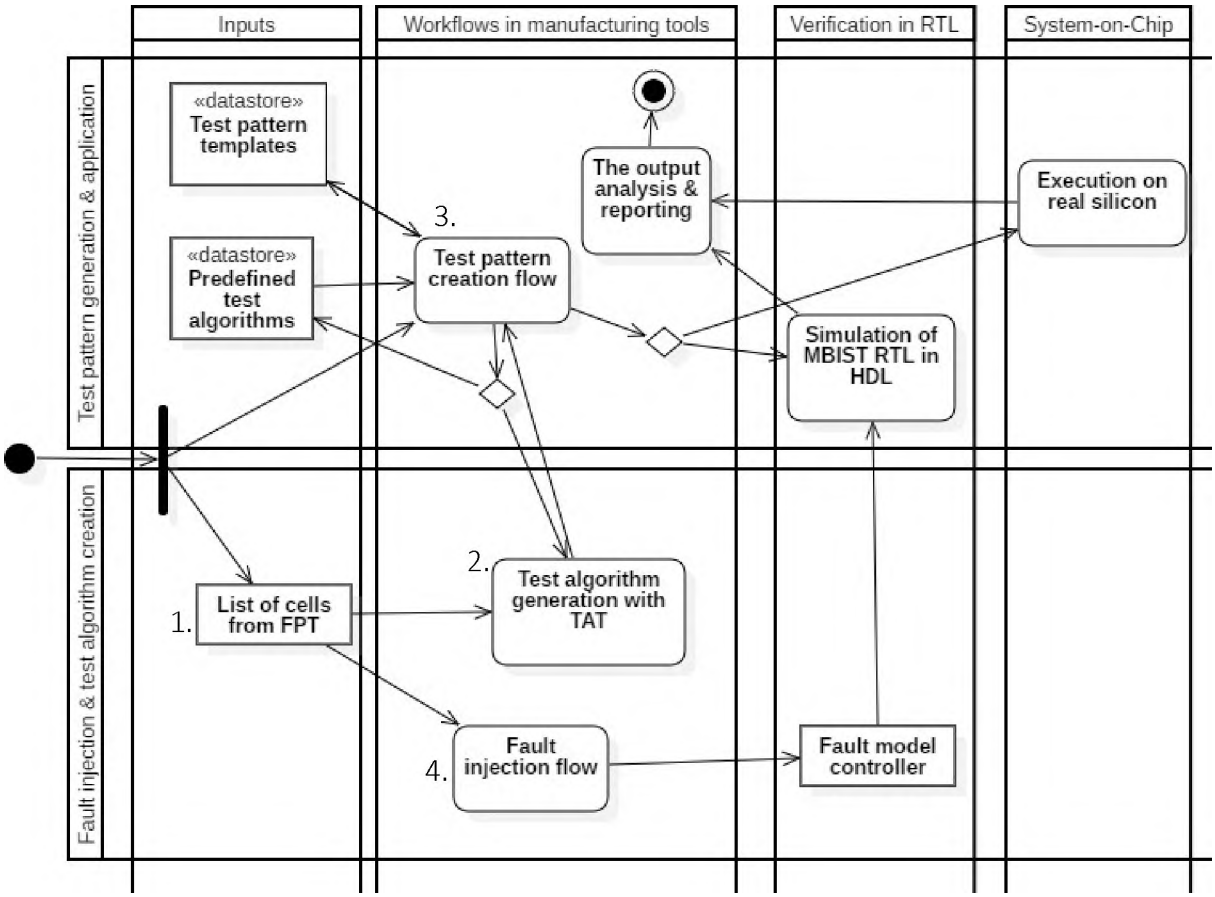


Figure 45. Modification to test pattern creation and verification flow

The modification of test pattern creation flow consists of the following 4 steps:

1. The list of fault groups is picked from FPT.
2. Test algorithm is created based on selected fault groups.
3. Test algorithm may be used in created test patterns instead of a predefined one.

4. The list of FPs is extracted from inputs and used in fault injection flow for resulting test pattern verification.

This allows automatic creation of test algorithms for detection of particular faults and their verification.

3.6.1. Using TAT in test pattern templates for memory test and fault detection

Test algorithm template was proposed for implementation in hardware [5] and the benefits of such approach were outlined. Nevertheless, the proposed test algorithm template can also be used in software tools to support automated test algorithm and test pattern generation flows.

Test algorithm for coverage of the particular group of faults is constructed based on test algorithm template with the following flow:

1. Based on the list of $\{FG_1(x_1, S_1), ..., FG_1(x_i, S_i), FG_2(x_{i+1}, S_{i+1}), ..., FG_2(x_n, S_n)\}$ provided at input, list of $\{FG(x_1,S_1), ..., FG(x_n,S_n)\}$ is constructed,
2. A sequence of operations S is constructed based on the resulting list of $\{FG(x_1,S_1), ..., FG(x_n,S_n)\}$ in the following manner $S = S_1W(x_2)S_2...W(x_n)S_n$ [5],
3. Test algorithm is formed using $TAT(x_1, S)$ for the list of given SMS processors,
4. It can now be used in test patterns for test and detection.

Finally, based on the list of provided FPT sub-groups the list of corresponding FPs can be extracted and randomly injected in MBIST networks. Fault detection and localization test patterns can be generated with the help of acquired test algorithm and further verified using the proposed approaches.

3.6.2. Using TAT for fault partial classification

March test algorithms generated with TAT can also be used for fault partial classification. During partial classification there is no guarantee that all the faults are uniquely identified as some test syndromes may correspond to more than one fault. For example, abovementioned VLP1(26N), VLP2(26N), VLP3(22N) are among those algorithms used for partial classification.

A generated algorithm can be applied on the FDT software model of each fault of FPT groups used as input for TAT. The process can benefit from parametrized FDTs that have been already generated and stored (p. 73). The resulting test syndromes may be stored along with the test algorithm for further use on the MBIST network. The flow is depicted in Figure 46.

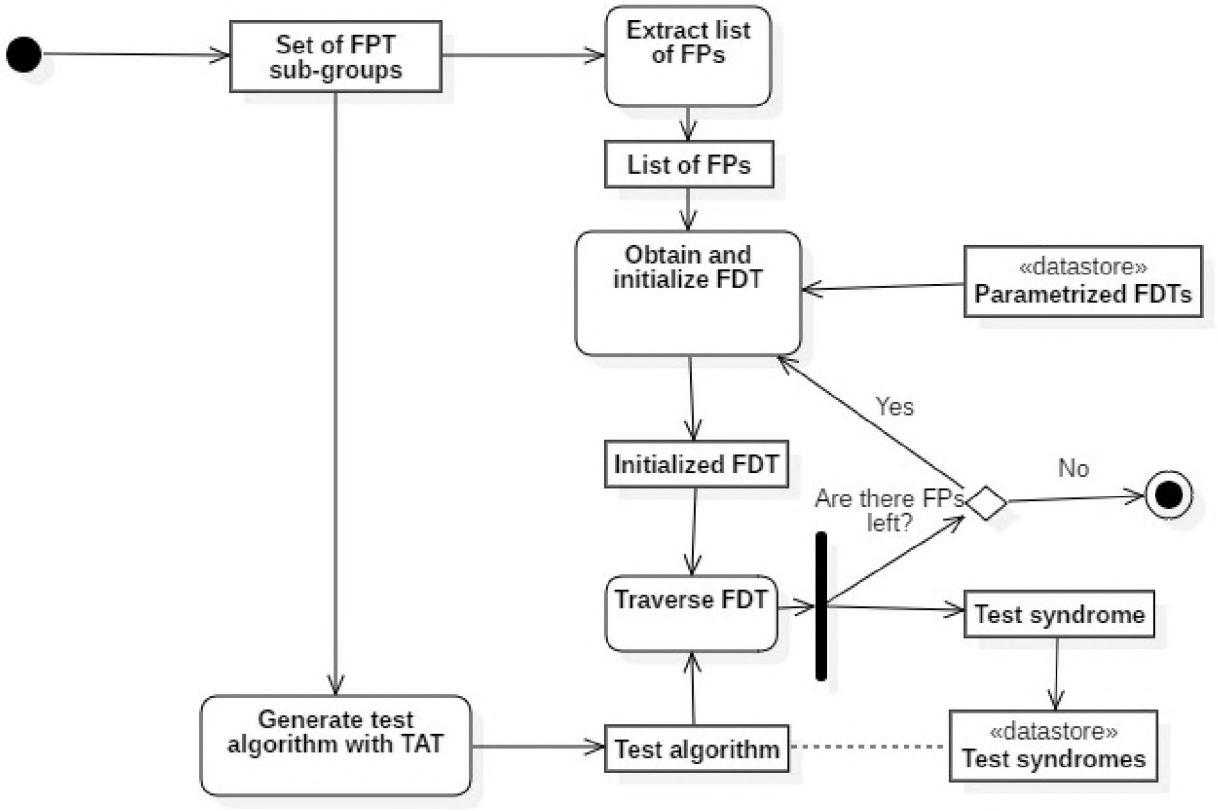


Figure 46.Test syndromes for partial fault classification

Conclusions

1. The fault injection flow is described for an arbitrary MBIST network configuration.
2. An environment for test pattern verification is built including input chain generation and output chain analysis.
3. Over 600 experiments for various MBIST network configurations are conducted to justify correctness of the proposed solution.
4. A modification of test and diagnosis implementation is suggested based on fault prediction mechanism.

SUMMARY

During the study:

1. Test pattern verification environment for test and diagnosis flow implementation for post-silicon analysis tools is built and experimentally justified.
2. An extendable fault model and fault generation flow are developed.
3. Fault model injection mechanism for memory internal faults is developed.
4. Approach on verification of test pattern input chain generation and output chain analysis is suggested.

TABLE OF FIGURES

REFERENCES

[1] "International Technology Roadmap for Semiconductors", www.itrs2.net, 2015

[2] Shoukourian S., Vardanian V. and Zorian Y., "SoC yield optimization via an embedded-memory test and repair infrastructure," IEEE Design & Test of Computers, vol. 21, no. 3, 2004, pp. 200-207

[3] Zorian Y., Shoukourian S. "Test Solutions for Nanoscale Systems-on-Chip: Algorithms, Methods and Test Infrastructure", Selected papers of Ninth International Conference on Computer Science and Information Technologies, IEEE, 2013, pp. 1-3

[4] Au A., Pogiel A., Rajski J., Sydow P., Tyszer J., Zawada J. "Quality assurance in memory built-in self-test tools", 17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, 2014, pp. 39-44

[5] Harutyunyan G., Martirosyan S., Shoukourian S., Zorian Y. "Memory Physical Aware Multi-Level Fault Diagnosis Flow", IEEE Transactions on Emerging Topics in Computing, 2018

[6] Harutyunyan G., Shoukourian S., Zorian Y. "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, 2019, pp. 562-575

[7] "IEEE Standard for Verilog Hardware Description Language" in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006, pp.1-590

[8] "IEEE Standard VHDL Language Reference Manual" in IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002), 26 Jan. 2009, pp.1-626

[9] Baltagi Y. et al., "Embedded memory fail analysis for production yield enhancement" 2011 IEEE/SEMI Advanced Semiconductor Manufacturing Conference, Saratoga Springs, NY, 2011, pp. 1-5

[10] Bergfeld T. J., Niggemeyer D. and Rudnick E. M., "Diagnostic testing of embedded memories using BIST" Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537), Paris, France, 2000, pp. 305-309

[11] Li J.-F., Cheng K.-L., Huang C.-T. and Wu C.-W., "March-based RAM diagnosis algorithms for stuck-at and coupling faults" Proceedings International Test Conference 2001 (Cat. No.01CH37260), Baltimore, MD, USA, 2001, pp. 758-767

[12] Carvalho M. et al., "Optimized embedded memory diagnosis" 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Cottbus, 2011, pp. 347-352

[13] IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device" in IEEE Std 1687-2014, 5 Dec. 2014, pp. 1-283

[14] IEEE Standard Testability Method for Embedded Core-based Integrated Circuits" in IEEE Std 1500-2005, 29 Aug. 2005, pp. 1-136

[15] IEEE Standard for Test Access Port and Boundary-Scan Architecture" in IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001), 13 May 2013, pp. 1-444

[16] Arslan B. and Orailoglu A., "Aggressive Test Cost Reductions Through Continuous Test Effectiveness Assessment", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 12, 2016, pp. 2093-2103

[17] Lee Y., Choi I., Oh K., Ko J. J. and Kang S., "Test item priority estimation for high parallel test efficiency under ATE debug time constraints", 2017 International Test Conference in Asia (ITC-Asia), Taipei, 2017, pp. 150-154.

[18] DesignWare STAR Memory System, Synopsys, Inc., https://www.synopsys.com/designware-ip/technical-bulletin/embedded-memory-test.html

[19] Tessent Memory Test, Mentor, a Siemens Business, http://s3.mentor.com/public_documents/datasheet/products/silicon-yield/products/memorybist-ds.pdf

[20] IEEE Standard for Test Access Port and Boundary-Scan Architecture" in IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001), 13 May 2013, p. 3

[21] Cheng C., Huang C.-T., Huang J.-R., Wu C.-W., Wey C.-J. and Tsai M.-C., "BRAINS: a BIST compiler for embedded memories," Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Yamanashi, Japan, 2000, pp. 299-307

[22] Haron N. Z., Junos S. A. M. and Aziz A. S. A., "Modeling and simulation of microcode Memory Built In Self Test architecture for embedded memories," 2007 International Symposium on Communications and Information Technologies, Sydney, NSW, 2007, pp. 136-139

[23] Haron N. Z., Junos S. A. M., Razak A. H. A. and Idris M. Y. I., "Modeling and simulation of finite state machine Memory Built-in Self Test architecture for embedded memories," 2007 Asia-Pacific Conference on Applied Electromagnetics, *Melaka, 2007, p*p. 1-5.

[24] Joseph P. E. and Antony P. R., "VLSI design and comparative analysis of memory BIST controllers," 2014 First International Conference on Computational Systems and Communications (ICCSC), *Trivandrum, 2014*, pp. 372-376

[25] Crouch A., "IEEE P1687 Internal JTAG (IJTAG) taps into embedded instrumentation", ASSET InterTech, 2011

[26] Cheng A., "Comprehensive Study on Designing Memory BIST: Algorithms, Implementations and Trade-offs", Advanced Computer Architecture Lab Department of Electrical Engineering and Computer Science, The University of Michigan, 2002, pp. 5-13

[27] Noor N. Q. M., Yusof Y. and Saparon A., "Low area FSM-based memory BIST for synchronous SRAM," 2009 5th International Colloquium on Signal Processing & Its Applications, Kuala Lumpur, 2009, pp. 409-412

[28] Dongkyu Y., Taehyung K. and Sungju P., "A microcode-based memory BIST implementing modified march algorithm," Proceedings 10th Asian Test Symposium, Kyoto, Japan, 2001, pp. 391-395.

[29] Boutobza S., Nicolaidis M., Lamara K. M. and Costa A., "Programmable memory BIST," IEEE International Conference on Test, 2005., Austin, TX, 2005, pp. 1-10

[30] Park K., Lee J. and Kang S., "An area efficient programmable built-in self-test for embedded memories using an extended address counter," 2010 International SoC Design Conference, Seoul, 2010, pp. 59-62

[31] Tsai P.-C., Wang S.-J. and Chang F.-M., "FSM-based programmable memory BIST with macro command," 2005 IEEE International Workshop on Memory Technology, Design, and Testing (MTDT'05), Taipei, 2005, pp. 72-77

[32] Noor N. M., Saparon A. and Yusof Y., "Programmable MBIST Merging FSM and Microcode Techniques Using Macro Commands," 2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems, Kyoto, 2010, pp. 115-121

[33] Aleksanyan K., Amirkhanyan K., Shoukourian S., Zorian Y., "Memory modeling using an intermediate level structural description", US Patent 07768840, issued on August 3, 2010

[34] van de Goor A. J. and Schanstra I., "Address and data scrambling: causes and impact on memory tests," Proceedings First IEEE International Workshop on Electronic Design, Test and Applications '2002, Christchurch, New Zealand, 2002, pp. 128-136

[35] Du X., Mukherjee N., Cheng W.-T. and Reddy S. M., "Full-speed field-programmable memory BIST architecture," IEEE International Conference on Test, 2005., Austin, TX, 2005, pp. 1-9

[36] Suk and Reddy, "A March Test for Functional Faults in Semiconductor Random Access Memories," in IEEE Transactions on Computers, vol. C-30, no. 12, pp. 982-985, Dec. 1981

[37] van de Goor A. J. "Testing Semiconductor Memories: Theory and Practice", John Wiley & Sons, 1991

[38] Yuejian Wu and S. Gupta, "Built-in self-test for multi-port RAMs," Proceedings Sixth Asian Test Symposium (ATS'97), Akita, Japan, 1997, pp. 398-403

[39] Zhao J., Irrinki S., Puri M. and Lombardi F. "Detection of inter-port faults in multi-port static RAMs," Proceedings 18th IEEE VLSI Test Symposium, Montreal, Quebec, Canada, 2000, pp. 297-302

[40] Li J.-F., Cheng K.-L., Huang C.-T. and Wu C.-W., "March-based RAM diagnosis algorithms for stuck-at and coupling faults," Proceedings International Test Conference 2001 (Cat. No.01CH37260), Baltimore, MD, USA, 2001, pp. 758-767

[41] Vardanian V. A. and Zorian Y., "A March-based fault location algorithm for static random access memories," Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002), Isle of Bendor, France, 2002, pp. 62-67

[42] Harutunyan G., Vardanian V. A. and Zorian Y., "A March-Based Fault Location Algorithm with Partial and Full Diagnosis for All Simple Static Faults in Random Access Memories," 2007 IEEE Design and Diagnostics of Electronic Circuits and Systems, Krakow, 2007, pp. 1-4

[43] Kiran P. N. V. and Saxena N., "Design and analysis of different types SRAM cell topologies," 2015 2nd International Conference on Electronics and Communication Systems (ICECS), Coimbatore, 2015, pp. 1060-1065

[44] Martirosyan L., Harutyunyan G., Shoukourian S. and Zorian Y., "A power based memory BIST grouping methodology," 2015 IEEE East-West Design & Test Symposium (EWDTS), Batumi, 2015, pp. 1-4

[45] van de Goor A. J. and Al-Ars Z., "Functional memory faults: a formal notation and a taxonomy," Proceedings 18th IEEE VLSI Test Symposium, Montreal, Quebec, Canada, 2000, pp. 281-289.

[46] Huang R.-F., Chou Y.-F. and Wu C.-W., "Defect oriented fault analysis for SRAM," 2003 Test Symposium, Xi'an, China, 2003, pp. 256-261

[47] Haron N. Z. and Hamdioui S., "On Defect Oriented Testing for Hybrid CMOS/Memristor Memory," 2011 Asian Test Symposium, New Delhi, 2011, pp. 353-358

[48] Hamdioui S., Al-Ars Z., van de Goor A. J. and Rodgers M., "Linked faults in random access memories: concept, fault models, test algorithms, and industrial results," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 5, May 2004, pp. 737-757

[49] Huzum C. and Cascaval P., "A multibackground march test for all static simple neighborhood pattern-sensitive faults in RAMs," 15th International Conference on System Theory, Control and Computing, Sinaia, 2011, pp. 1-6.

[50] Sfikas Y., Tsiatouhas Y. E. and Hamdioui S., "Layout-Based Refined NPSF Model for DRAM Characterization and Testing," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 6, June 2014, pp. 1446-1450

[51] Hamdioui S., Al-Ars Z. and van de Goor A.J., "Testing Static and Dynamic Faults in Random Access Memories", In Proceedings of IEEE VLSI Test Symposium (VTS), 2002, pp. 395-400

[52] Hamdioui S., Gaydadjiev G. N. and van de Goor A.J., "A fault primitive based analysis of dynamic memory faults", IEEE Workshop on Circuits, Systems and Signal Processing, 2003, pp. 84-89.

[53] Dilillo L., P. Girard, S. Pravossoudovitch, A. Virazel, "Dynamic read destructive fault in embedded-SRAMs: analysis and march test solution" IEEE European Test Symposium, 2004, pp. 140–145.

[54] Hamdioui S., Wadsworth R., Reyes J. D., van de Goor A.J., "Importance of Dynamic Faults for New SRAM Technologies", IEEE European Test Workshop, 2003, pp. 29-34.

[55] Azimane M., Majhi A. K., Gronthoud G., Lousberg M., et al, "A New Algorithm for Dynamic Faults Detection in RAMs", IEEE VLSI Test Symposium, 2005, pp. 177-182.

[56] Ney A., Girard P., Landrault C., Pravossoudovitch S., Virazel A., Bastian M., "Slow Write Driver Faults in 65nm SRAM Technology: Analysis and March Test Solution", IEEE Design, Automation and Test in Europe, 2007, pp. 528-533.

[57] Ney A., Girard P., Landrault C., Pravossoudovitch S., Virazel A. and Bastian M., "Dynamic Two-Cell Incorrect Read Fault due to Resistive-Open Defects in the Sense Amplifiers of SRAMs", IEEE European Test Symposium, 2007, pp. 97-104.

[58] Al-Ars Z., Hamdioui S., Gaydadjiev G., "Manifestation of Precharge Faults in High Speed DRAM Devices", IEEE Design and Diagnostics of Electronic Circuits and Systems, 2007, pp. 179-184.

[59] Bosio A., Dilillo L., Girard P., Pravossoudovitch S., Virazel A., "Advanced Test Methods for SRAMs: Effective Solutions for Dynamic Fault Detection in Nanoscaled Technologies", Springer, 2009.

[60] http://www.samsung.com/global/business/semiconductor/file/media/Samsung_Foundry_14nm_FinFET-0.pdf [Online]

[61] http://www.intel.com/content/www/us/en/siliconinnovations/intel-22nm-technology.html [Online]

[62] http://www.tsmc.com/english/dedicatedFoundry/services/reference_flow.htm [Online]

[63] Harutyunyan G., Tshagharyan G., Vardanian V., Zorian Y., "Fault modeling and test algorithm creation strategy for FinFET-based memories", VLSI Test Symposium, 2014, pp. 1-6.

[64] Harutyunyan G., Tshagharyan G., Zorian Y., "Test & Repair Methodology for FinFET-Based Memories", IEEE Transactions on Device and Materials Reliability, Vol. 15, No. 1, March 2015, pp. 3-9.

[65] Tshagharyan G., Harutyunyan G., Shoukourian S., Zorian Y., "Overview Study on Fault Modeling and Test Methodology Development for FinFET-Based Memories", IEEE East-West Design and Test Symposium, 2015, pp. 19-22.

[66] Nagel L. W., Pederson D. O., "SPICE (Simulation Program with Integrated Circuit Emphasis)", Memorandum No. ERL-M382, University of California, Berkeley, Apr. 1973.

[67] Benso A., Bosio A., Di Carlo S., Di Natale G. and Prinetto P., "Automatic March tests generation for static and dynamic faults in SRAMs," European Test Symposium (ETS'05), Tallinn, Estonia, 2005, pp. 122-127

[68] Brzozowski J. A. and Jurgensen H., "Composition of multiple faults in RAMs," Records of the 1995 IEEE International Workshop on Memory Technology, Design and Testing, San Jose, CA, USA, 1995, pp. 123-128

[69] Harutyunyan G., Shoukourian S., Vardanian V. and Zorian Y., "An effective solution for building memory BIST infrastructure based on fault periodicity," 2013 IEEE 31st VLSI Test Symposium (VTS), Berkeley, CA, 2013, pp. 1-6.

[70] Harutyunyan G., Shoukourian S., Vardanian V. and Zorian Y., "Extending fault periodicity table for testing faults in memories under 20nm," Proceedings of IEEE East-West Design & Test Symposium (EWDTS 2014), Kiev, 2014, pp. 1-4

[71] Harutyunyan G., "Extending fault periodicity table for testing external memory faults," 2016 IEEE East-West Design & Test Symposium (EWDTS), Yerevan, 2016, pp. 1-4

[72] Nahir A. et al., "Bridging pre-silicon verification and post-silicon validation," Design Automation Conference, Anaheim, CA, 2010, pp. 94-95

[73] Mishra P., Morad R., Ziv A. and Ray S., "Post-Silicon Validation in the SoC Era: A Tutorial Introduction," in IEEE Design & Test, vol. 34, no. 3, June 2017, pp. 68-92

[74] Harcha G., Bosio A., Girard P., Virazel A. and Bernardi P., "An effective fault-injection framework for memory reliability enhancement perspectives," 2017 12th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), Palma de Mallorca, 2017, pp. 1-6

[75] Hopcroft J., Motwani R., Ullman J. "Introduction to Automata Theory, Languages, and Computation" 3rd edition, Addison-Wesley, 2013

[76] Hayrapetyan D., Manukyan A. “Modeling dynamic single-cell and coupling faults via automata models”, Computer Science and Information Technologies (CSIT), Armenia, 2017, pp. 65-68

[77] Hayrapetyan D., “Modeling linked faults via automata models”, IEEE East-West Design and Test Symposium (EWDTS), Serbia, 2017, pp. 237-241

[78] Hayrapetyan D., Manukyan A., Tshagharyan G. “Implementation of Memory Static, Coupling and Dynamic Fault Models at the Register Transfer Level”, IEEE East-West Design and Test Symposium (EWDTS), Russia, 2018, pp. 744-748

[79] Dilillo L., Girard P., Pravossoudovitch S., Virazzel A., “Efficient test of dynamic read destructive faults in sram memories”, 6th IEEE Latin American Test Workshop, Salvador, Bahia, Brazil, 2005, pp. 40-45

[80] Graphviz - Graph Visualization Software, https://www.graphviz.org/ [Online]

[81] The DOT language, https://www.graphviz.org/doc/info/lang.html [Online]

[82] van de Goor A. J., Gaydadjiev G. N., Mikitjuk V. G. and Yarmolik V. N., "March LR: a test for realistic linked faults," Proceedings of 14th VLSI Test Symposium, Princeton, NJ, USA, 1996, pp. 272-280

[83] Harutyunyan G., Shoukourian S., Vardanian V. and Zorian Y., "A New Method for March Test Algorithm Generation and Its Application for Fault Detection in RAMs," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, no. 6, 2012, pp. 941-949

[84] Sipser M. “Introduction to the Theory of Computation”, 2nd edition, Thomson, 2006

[85] Hayrapetyan D. “Verification of Test and Diagnosis Flow Implementation in Software Post-Silicon Analysis Automation Tools”, Reports of the National Academy of Sciences and the State Engineering University of Armenia. Series of Technical Sciences, Yerevan, 2019

[86] Seidl M., Scholz M., Huemer C., Kappel G. “UML @ Classroom: An Introduction to Object-Oriented Modeling (Undergraduate Topics in Computer Science)” Springer, 2015

[87] StarUML, http://staruml.io/ [Online]

[88] “IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language – Redline”, in IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline, 2009, pp.1-1346

APPENDIX: CERTIFICATE OF IMPLEMENTATION

SYNOPSYS®

№ 285/19 "22" 04 2019

Հաստատում եմ՝
«ՍԻՆՈՓՍԻՍ ԱՐՄԵՆԻԱ» ՓԲԸ
Գլխավոր տնօրեն՝
Հ. Մուսայելյան

«Բյուրեղի վրա սխալների առկայությամբ հիշող հանգույցների ներկառուցված թեստավորման գործընթացը մոդելավորող ծրագրային գործիքի հիմնավորում և մշակում» թեմայով Դավիթ Լևոնի Հայրապետյանի թեկնածուական ատենախոսության արդյունքների

ՆԵՐԴՐՄԱՆ ԱԿՏ

Դ.Լ. Հայրապետյանի տեխնիկական գիտությունների թեկնածուի գիտական աստիճանի ատենախոսության կատարման ընթացքում ստացված հետևյալ արդյունքները՝

- հիշող սարքերում RTL մակարդակում անսարքությունների մոդելավորման ծրագրային համակարգը,
- հիշող սարքերի ներկառուցված ինքնաթեստավորման համակարգում թեստավորման և ախտորոշման գործընթացների իրականացումը ստուգող միջավայր,

ներդրվել են «Սինոփսիս» ընկերության DesignWare STAR Memory System (SMS) արտադրանքի հիշող սարքերի ներկառուցված ինքնաթեստավորման համակարգի համար նախագծված Yield Accelerator վերլուծական գործիքում և այժմ փորձարկումներ է անցնում:

Արդյունքների օգտագործումը բերել է նշված գործիքի արդյունավետության բարձրացման, որի արդյունքում՝

- հնարավոր է դարձել նորագույն տեխնոլոգիաներին բնորոշ անսարքությունների մոդելավորումը RTL մակարդակում և դրանց հայտնաբերման ու ախտորոշման համար նախատեսված թեստային նմուշների ստուգումը,
- բարելավվել է հիշող սարքերի ներկառուցված ինքնաթեստավորման համակարգի համար թեստավորման և ախտորոշման գործընթացների իրականացումը նորագույն տեխնոլոգիաների ազդեցությունը հաշվի առնելու համար:

Սինոփսիս ընկերության ավագ կառավարիչ,
ՀՀ ԳԱԱ ակադեմիկոս, ֆ.մ.գ.դ., պրոֆեսոր Ս. Շուքուրյան

"ՍԻՆՈՓՍԻՍ ԱՐՄԵՆԻԱ" ՓԲԸ
0026, ՀՀ, ԵՐԵՎԱՆ, ԱՐՇԱԿՈՒՆՅԱՑ 41
ՀԵՌ. (+374 10) 49 21 00, ՖԱՔՍ (+374 10) 49 26 96
ՀՎՀՀ 02236362

"SYNOPSYS ARMENIA" CJSC
41 ARSHAKUNYATS AVE., YEREVAN, ARMENIA, 0026
TEL.: (+374 10) 49 21 00, FAX: (+374 10) 49 26 96
TAX PAYER'S ID 02236362