A Generic Configurable Error Injection Agent for On-Chip Memories

Niharika Sachdeva, Arjun Suresh Kumar, Damandeep Saini, Anil Deshpande, Ravi Teja Gopagiri, Somasunder KS, Samsung Semiconductor India Research (SSIR), Bengaluru, India
niharika.s@samsung.com, arjun.sk@samsung.com, daman.saini@samsung.com,anil.pande@samsung.com, raviteja.g@samsung.com, soma.ks@samsung.com,
Jaechul Park, Principal Engineer, Samsung Electronics, Korea (*jejepark@samsung.com*)

***Abstract*—Memory reliability testing is indispensable today as errors in memory can cause not only serious hardware system failures but also compromises the safety aspect. System chips contain error detection and correction modules to recover from these soft and hard errors, hence verifying such ECC circuits becomes imperative. All SoCs and memory controllers deal with various types of memory like SRAM, MRAM and Flash Memory. In this paper, we propose a generic and configurable UVM agent, which can simulate error injection on any type of memory. It can scale up with different address and data widths. It is capable of doing single, double, multi bit error injection using march, and scan test patterns for the entire address space. This standardizes an approach for verification of ECC blocks, and saves time on individual implementation of scenarios.**

Keywords—memory, ecc, uvm, error correction, detection, error injection

I. Introduction

Memories are used everywhere ranging from satellites and planes in the sky, automotive and electronic equipment's in the ground to the submarines below sea. As the size of memories are constantly becoming smaller, errors in memories are becoming more frequent. With small memories operating in low power, minuscule electrical disturbance is enough to create a bit flip to start a cascade of catastrophic events. Errors or bit flips are usually classified into soft errors and hard errors. Soft errors are transient and are usually correctable. They are created by temporary environmental factors such as particle strikes from radioactive decay and cosmic ray-induced neutrons. Hard errors, are caused by inherent manufacturing defect, insufficient burn-in, or device aging [5]. Once these manifest, they tend to cause more predictable errors as the deterioration is mostly irreversible. However, before transitioning into permanent hard errors, they may put the device into a marginal state causing apparently intermittent errors. If memories are used in high altitude, harsh weather or extreme temperature environments, they need extra reliability and safety components to keep the data error-free from both soft and hard errors. So all mission critical memory components are protected by Error Correcting Code (ECC) circuitry to detect and correct such bit flips. Radiolab did an episode on the case of a cosmic bit flip changing the vote tally in a Belgian election in 2003. The error was caught because one candidate got more votes than was logically possible. A recount showed that the person in question got 4096 more votes in the first count than the second count. The difference of exactly 2^{12} votes was a clue that there had been a bit flip. All the other counts remained unchanged when they reran the tally. There are many such incidents which occur due to bit flip in memories. Hence it is crucial to verify the ECC component of the design, which is going to recover the system in case of memory errors during operation. In this paper, we present a generic simulation-based verification method which will effectively find issues with ECC design in a faster way for different types of memories like SRAM, MRAM and Flash memory. This will reduce the verification time required for complete testing of all possible bit errors and combinations.

The remaining parts of this paper is organized as follows: Section II describes about on-chip memories and organization. Section III briefly touches upon components of an ECC implementation. Section IV presents some of the related work in formal-based verification and simulation-based verification. Section V discusses about the implementation of generic error injection agent. Section VI contains the results and Section VII concludes the paper with ideas for future enhancements.

II. ON-CHIP MEMORIES

Memories can be classified based on the way it operates, RAM (Random Access Memory), ROM (Read Only Memory). In these 2 categories, there are many types of memories based on technology, volatility, programmability and system use-case like MRAM, Flash, SRAM, DRAM etc. as shown below in Figure 1

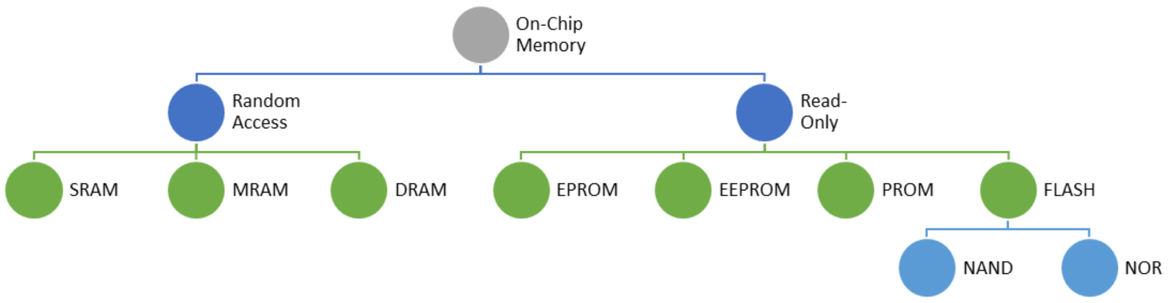

Figure 1 Types of On-Chip Memory

These memories are used in CPU, Caches, Primary Memory and Secondary Memory. Different memory types are used considering performance, volume and latency requirements of IP's and SOC's as shown in Figure 2.

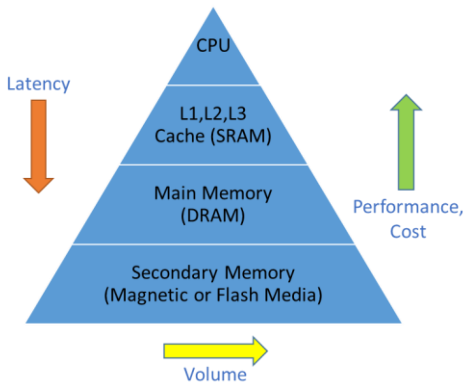

Figure 2 Memory Hierarchy

DRAM is a volatile memory, which means it will lose the contents of its memory as the capacitors that store the bits discharge, commonly it will discharge within a few milliseconds. As a result, DRAM requires refresh cycles that read the data bits and then re-write the data back to the chip to re-enforce the stored data. DRAM also destructively reads. This means that when a bit is read from DRAM, the contents of the memory bit that was accessed are forgotten and therefore require a write-back operation.

Another memory technology that exists, called SRAM, is a volatile memory technology that does not use capacitors to store bit. Instead, it includes a simple latch made of six transistors. While SRAM also loses its stored information when turned off, it does not require refresh cycles because its feedback loop design latches data when it is written to. Since it does not require a write-back operation to retain the data; this makes SRAM faster than DRAM.

FLASH is a memory technology that is both similar to and distinct from DRAM. First, each bit in FLASH memory is made up of a single transistor, but these transistors have a special layer called a *floating gate*. Bits are stored in FLASH memory by using quantum tunneling to trap electrons in the floating gate layer, which makes the transistor more or less conductive. When a voltage is applied across the transistor bit, the conductive capability of that transistor will depend on whether there are electrons trapped in the floating gate. Unlike DRAM, FLASH memory is non-volatile, which means that FLASH memory will retain any data stored to it when turned off.

The memories discussed so far all rely on storing electrical charge in some fashion. Magneto resistive memory (MRAM), by contrast, uses a magnetic layer to store a bit. A single large version of such a cell has been in use for a long time – in the read heads of hard disk drives. The bit cell consists of two magnetic layers. One has a fixed polarity; the other can have its polarity set either parallel or anti-parallel to the fixed layer. If both layers are polarized in the same direction, then the current tunneling across a barrier will flow relatively easily. If the layers are in opposite directions, then the tunneling current will have a harder time flowing. The cell value is read by measuring that current. MRAM technology has evolved for denser embedding in SoCs. The magnetic layers, however, require CMOS-friendly materials that aren't used for standard CMOS. Those extra steps can increase the cost of the finished wafer. But because of the small cell size, an MRAM-based SoC will be smaller than an equivalent SRAM-based SoC would be.

III. Error Detection And Correction Codes

ECC is implemented by adding some extra bits called parity or check bits to data bits of the memory. The logic used to generate those bits is based on popular algorithms like Hamming and BCH. Encoding involves generating parity bits from the data bits and storing them alongside in memory, this is generally done during a memory write. When memory read occurs, the decoding process will read the data and parity bits and detect if any errors. Depending upon the capability of the ECC algorithm implemented, it will be able to detect and correct errors or raise uncorrectable flags to let the design and user know the status of data corruption. Figure 3 shows the general flow below for any type of on chip memory we discussed in previous section:

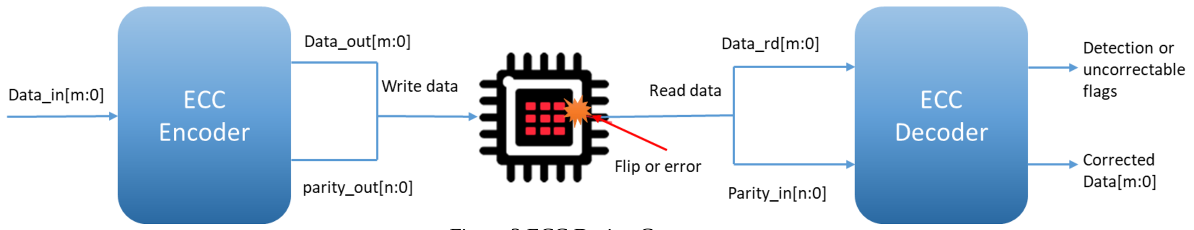

Figure 3 ECC Design Components

The verification of ECC hardware design usually focuses on the decoding process that it is able to detect and correct single/multiple bit errors as per specification. The ECC Circuit may fail to detect errors if more errors are injected than supported by the ECC algorithm. For Example: In a 3-bit Detection and 2-bit Correction ECC design, if more than 3 bit errors are injected, there is a possibility that error might not be detected or it might wrongly show as single bit error.

IV. Related Work

There are two main categories of ECC block verification methodologies namely

1) Formal Based
2) Simulation Based

A. *Formal Based*

Formal verification involves two aspects, one is formal reasoning and other is formal modelling. Formal modelling involves developing a system model, which covers the complete specification of ECC logic and equations implementation. Formal reasoning involves mathematical reasoning to verify model and its associated properties, properties will cover both no error and presence of error behavior. The required characteristics of such an approach is that it is reusable in different implementations and converges in limited amount of time.
However, it has been observed that it is generally not scalable and hence limited in actual real case of SOC memories where we might have different types of memories with varied capabilities of ECC [1]. Moreover, to converge such models faster, optimizations are needed which in turn require knowledge about internal circuitry. Such knowledge about hardware design may not be available always if vendor IP's are being used. We have not explored formal for this paper; it can be a future enhancement for more comparison parameters.

B. Simulation Based

Second category of simulation deals with error injection to test how the memory or system recovers. This is the approach we take in our reactive agent as well. Error injection is mainly on the read/decoding process. We use UVM as a platform for the agent as it offers reusability, sufficient randomness and coverage to verify the design effectively. Most of the simulation-based verification targets for one specific type of memory or controller design [2]. The approaches mostly test some limited error combinations to verify the design, which affects the reliability of the simulation results. Standardization of approach is required to ensure that we are achieving confidence in terms of combinations testing and coverage of design.

The proposed agent aims to overcome the limitations listed. It can be configured for any type of memory and can be easily plugged in and run in parallel with existing memory read/write verification scenarios without knowing the ECC algorithm details.

V. IMPLEMENTATION

The agent can be instantiated in an existing target VE or used standalone with a memory module.

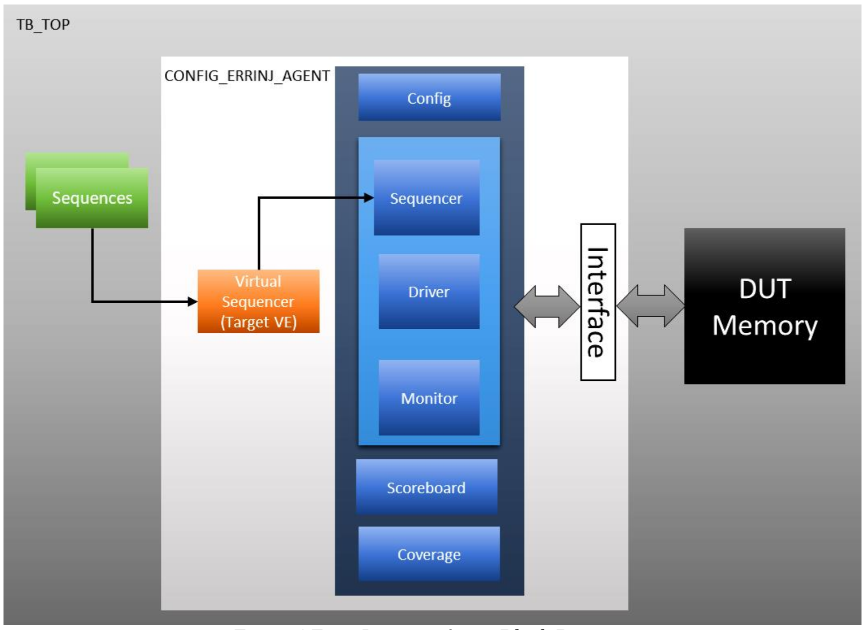

Figure 4 Error Injection Agent Block Diagram

Main components of the agent are:

A. *Sequences:* This class contains code for scan and march patterns. For example, 4N operations of WR→RD→WR→RD with random, all 1's, all 0's, all A's, all 5's data pattern. The errors are injected randomly or incrementally on the entire address space. It also has sequence to simulate stuck at faults (SF) and transition faults (TF).

B. *Sequencer:* This module receives stimulus' data from the sequence and it sends this information into the driver.

C. *Driver:* This unit drives the error information on the memory bus, it will corrupt the data during read.

D. *Monitor:* This entity monitors all pin toggles on the memory bus interface. The monitor generates a transaction item of read/write data, address and error injection position.

E. *Coverage:* The coverage class will continuously track the error injected positions and error address and this will give a measure of the quality of stimulus to the verification engineer on how much the ECC block is tested. Since state space of verifying an ECC block can quickly explode, this coverage class help to do closure faster. The crosses in the coverage class help to identify the address and data position range covered in different cases of error injection. Cross coverage between normal and parity bits are also available.

F. *Scoreboard:* The transaction item from monitor is sent to an existing scoreboard of a verification environment to implement customized checkers or to the scoreboard of our agent to check if correction and detection are working as expected. This class compares predicted (from reference model) and observed (from monitor) ECC status.

G. *Configuration class:* The agent will identify reads from the interface connected to memory and points of error injection from the configuration class to do the actual forcing. This class will also contain randomized variables, which a user can customize as per requirement. Some of them are:

1) Address range,
2) Data width ranges
3) Error count Type (single, double, triple or multiple)
4) Continuous error insertion for all operations, this can help to model stuck at faults as well.
5) Error insertion on fixed positions
6) Error insertion based on Ratio

It is independent of the type of algorithm used to do error correction/detection. It does random insertion and monitors the received data. Table 1 lists the error injection features supported by the agent.

Table 1 Error Injection Agent Configuration Categories

Feature	**Description**
Type of error	It supports single, double, triple, and multiple (more than 3) error injection
Error injection mode	It supports injecting on random address, fixed address, on all addresses starting from an initial address (incremental mode)
Ratio mode	In case of multiple error injection, error can be injected in ratios like 80% are flipped, or all bits are injected. Random ratio can be also be selected.
Non-stop mode	Non-stop error injection is supported, continuously inject Error for all reads to model a completely faulty memory
Fixed error position	User can use this to only inject error on a particular data bit position, it can help to model stuck at faults and transition faults.
Address range	Address range can be configured to only insert randomly on a set of address
Position range	Position range can be configured to only insert in a particular data width range
Error Count	If in incremental mode, user can configure number of errors injected.
Number of ECC's	If a particular data width is covered by multiple ECC blocks, then error injection can be configured to have parallel testing for those blocks.
Coverage class	The coverage class will contain cover points related to features mentioned and have crosses of them as well. To cover data width fully, a cover point has been added to see all bits are injected at least once. For double and triple error injection cases, data width is divided into 8 sections and cross combinations of those sections are added as bins.

For standalone case, the agent offers a collection of march/scan patterns to test, which can be randomly configured for any address range, data pattern and order of operations.

Table 2 Test Scenario Annoatations

r	Read on Memory
w	Write on Memory
r0	Read 0 on Memory
r1	Read 1 on Memory
w0	Write 0 on Memory

w1	Write 1 on Memory
w0/1	Write 0 in even address and write 1 in odd address
r0/1	Read for 0 in even address and Read for 1 in odd address
w1/0	Write 1 in even address and write 0 in odd address
r1/0	Read for 1 in even address and Read for 0 in odd address
↑	Increasing Memory Address Order
↓	Decreasing Memory Address Order
↨	Random Memory Address Order

Different types of Patterns are available to test memory, but we found the below patterns to have maximum coverage and effectiveness in testing the ECC circuitry for on-chip memories. The patterns are as follows:

A. *MARCH C* : {↨ w0, ↑ (r0, w1), ↑ (r1, w0), ↓ (r0, w1), ↓ (r1, w0), ↨ r0}
   1. Write 0s in all address in any order
   2. Read from the lowest address and check for zero data. Write 1 at this Address and repeat r0, w1 until the last address
   3. Read from the lowest address and check for all one data. Write 0 at this address and repeat r1, w0 until the last address
   4. Read from the last address and check for zero data. Write 1 at this address and repeat r0, w1 until the first address
   5. Read from the last address and check for all one data. Write 0 at this address. Repeat r1, w0 until the first address.
   6. Read all address in any order and check for zero data.

B. *SCAN* : {↑ w0, ↑ r0, ↑ w1, ↑ r1}
   1. Write 0s to all address in ascending order of address
   2. Read and check for zero in the ascending order of address
   3. Write 1s to all address in ascending order
   4. Read and check for all 1s in the same order

C. *CHECKERBOARD* : {↑ w0/1, ↑ r0/1; ↑ w1/0, ↑ r1/0}
   1. Write 0s and 1s in alternating order while the address increases in ascending order till the last address
   2. Read the address in the same order and check for 0 and 1 in the same alternating order it was written.
   3. Write 1s and 0s in alternating order while the address increases in ascending order till the last address
   4. Read the address in the same order and check for 1s and 0s in the same alternating order it was written.

D. *RANDOM* : {↑ w, ↑ r, ↓ w, ↓ r, ↨ w, ↨ r}
   1. Write Random Value in ascending order of address
   2. Read back the same random value in the same order
   3. Repeat step 1 and step 2 but in descending and random order of address subsequently

Below is the snapshot showing agent in action, injecting errors continuously for reads to memory:

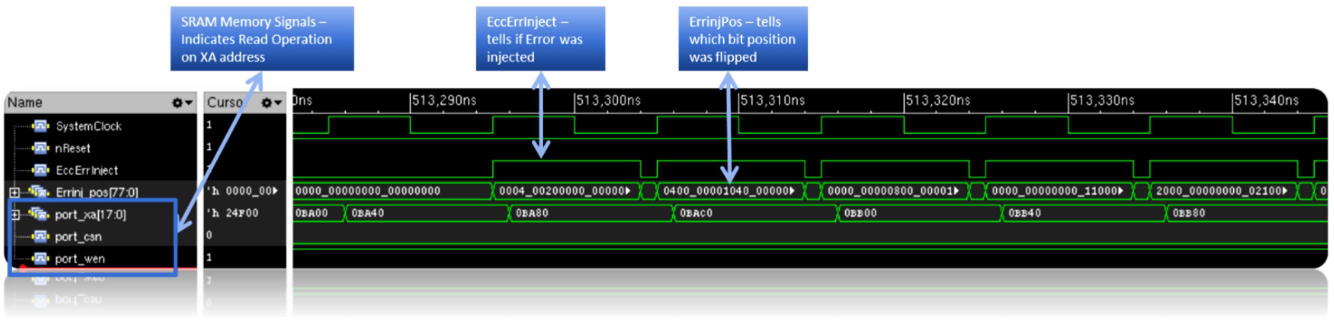

Figure 5 Error Injection Waveform

VI. RESULTS

Using the combination of different patterns as mentioned above, we were able to cover both encoder and decoder logic of the ECC blocks for 2 configurations of MRAM using our agent as shown in Table 3. Each bit in the data width was injected with error at least once as well (part of the functional coverage). In a tested SOC, with multiple SRAM's present, this common agent could be used for all memory instances avoiding repetition of code. The saving in testbench development time shows how our agent is easy to integrate and scalable. Since it can be run in parallel with patterns, it does not pose an overhead in simulation time, but instead helps in saving time to achieve coverage faster.

Table 3 Tested Memory Configurations

Memory Type	Data width	No of ECC Blocks	Number of coverage bins (RTL design code)	Simulation time saved to achieve 100% coverage	Test bench development time saved
MRAM	64 data bits + parity bits	1	7100	25%	40%
MRAM	2*(128 data bits + parity bits)	2	18400*2	35%	45%
SRAM	32 bits + parity bits	1	400	20%	50%
FLASH	32 bits + parity bits	1	1589	25%	35%

Figure 6, Figure 7 and Figure 8 illustrate the difference in amount of time taken to achieve 100% coverage with different patterns for MRAM (64-bit and 128-bit configuration), Flash and SRAM respectively. Since 128-bit configuration had 2 ECC block, multiple thread option was used in simulation.

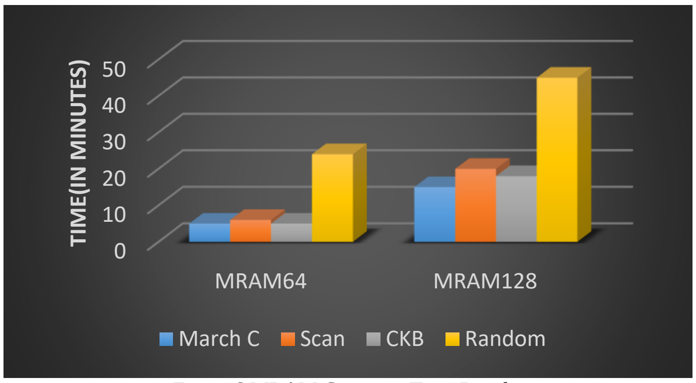

Figure 6 MRAM Coverage Time Results

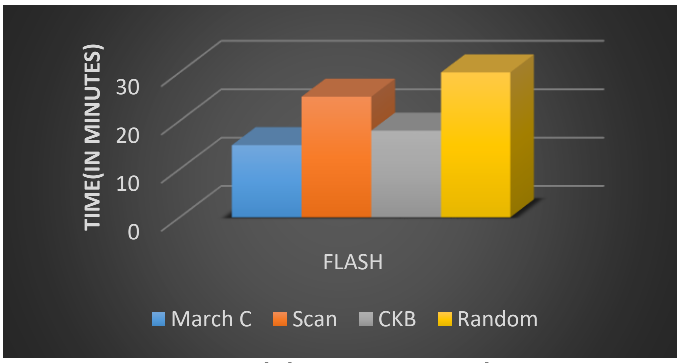

Figure 7 Flash Coverage Time Results

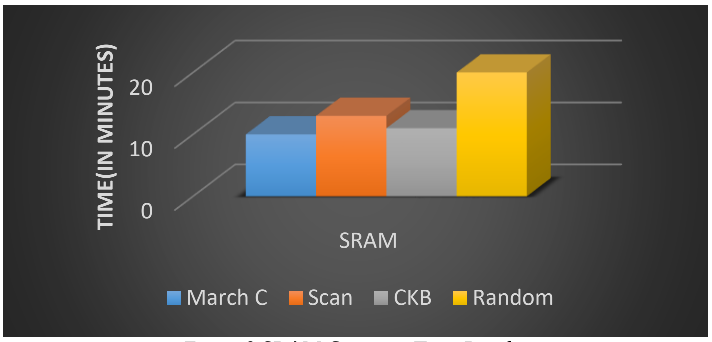

Figure 8 SRAM Coverage Time Results

Utilizing this generic and configurable agent, error resiliency of the design has improved. We found critical and corner case bugs in a memory BIST controller module within a week, which otherwise would have required directed error injection scenarios to catch them.

VII. Conclusion And Future Work

The impact of using this agent in early stages of system verification in as follows:

1. A week of effort reduced to a day of work and verification engineer can save almost 50% of the time spent to design error injection scenario for changed specification or new addition of memory.
2. Early detection of bugs in embedded ECC logic in memory designs and controllers
3. Coverage class of the agent gives confidence that ECC circuit is good for sign off.

As part of future enhancements, we are working on optimizing on our memory test patterns to close verification faster and continuously testing for different controllers and memories; Also, we would like to test our agent on actual memory netlists (non-behavioral model) and compare the performance metrics.

References

[1] E. Arbel, S. Koyfman, P. Kudva and S. Moran, "Automated detection and verification of parity-protected memory elements," *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014

[2] G. Visalli, "UVM-based verification of ECC module for flash memories," *2017 European Conference on Circuit Theory and Design (ECCTD)*, 2017, pp. 1-4

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

[4] S. El-Ashry, M. Khamis, H. Ibrahim, A. Shalaby, M. Abdelsalam and M. W. El-Kharashi, "On Error Injection for NoC Platforms: A UVM-Based Generic Verification Environment," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 5, pp. 1137-1150, May 2020.

[5] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro, 25(6):10–16, Nov.–Dec. 2005.

[6] Reeja J, Anusree L. S., “Design of Built-in Self-Test Core for SRAM”, International Journal of Engineering Research & Technology (IJERT), ISSN: 2278-0181, Vol. 3 Issue 3, March – 2014

[7] Twinkle Koshy, Manjusree S, “Fusion of March Algorithms in Counter based BIST for the Detection of Faults in RAM”, International Journal of Science and Research (IJSR) ISSN (Online): 2319-7064 Index Copernicus Value (2013): 6.14 | Impact Factor (2015): 6.391