

Enabling ECC and Repair Features in an eFuse Box for Memory Repair Applications

Miguel Costa, Srikanth Beerla
Intel Corporation, Intel Technology Pvt Ltd

ABSTRACT

To manage yield on the ever-growing memories in SoCs it is critical not to compromise on the reliability of the eFuse box storing the memory repair data. To this end, the latest eFuse memories support Error Correction Code (ECC) and repair features to increase reliability and enhance yield. However, conventional Design-For-Test (DFT) methods do not provide solutions to leverage those features. This paper proposes a methodology to enable eFuse repair and presents a custom ECC logic module that allows an eFuse box to seamlessly interface with a near-industry-standard Built-In Self-Repair (BISR) controller.

KEYWORDS

DFT, BIST, MBIST, BIRA, BISR, eFuse Box, Self-Repair, Memory Repair, Yield, Reliability

1. INTRODUCTION

Implementation of memory repair schemes has become an essential part of modern SoC development in order to achieve a high manufacturing yield. BISR is the most widely used technique to perform on-chip repair. It operates in unison with Built-In Self-Test (BIST) and Built-In Repair Analysis (BIRA) circuitry to come up with repair signatures for faulty memories and efficiently store them in an eFuse box. The repair information from the fuse box is shifted into memory redundancy registers every time the chip is powered on. Thus, the fuse box plays a critical role in memory repair flows.

The trend of an increasing number of memories on SoCs has resulted in a need for more fuses on the die to repair them. However, with contemporary technology scaling, the fuses that store memory repair information have increasingly serious problems with yield and reliability. To mitigate this problem, chip designers across the industry are implementing fuse boxes with at least 2x redundancy. That is, every fuse bit cell will have a copy that stores identical information. The increased number of eFuses on the chip results in a substantial area overhead. A typical eFuse bit cell is physically large and does not scale well at lower technologies, causing floorplanning issues. Additionally, eFuses are programmed by applying a large current to the fuse body. As the operating voltages have scaled down, eFuses require additional power supply rails to supply the relatively massive power required for write operations.

Considering all these factors, a more optimal solution is to minimize the number of eFuses on the die by utilizing the ECC and repair functions provided in the latest eFuse boxes. However, existing commercial MBIST tools do not provide a mechanism to enable those functions. This paper will describe a methodology to enable them for a generic fuse box using the Tessent Shell MBIST flow. The approach presented here can be extended to vendor-specific fuse boxes and other EDA tools with appropriate modifications.

2. EFUSE BOX OVERVIEW

An eFuse is a type of non-volatile memory that can only be programmed once to permanently store any kind of information. It has become a popular choice to enable such features as memory redundancy, chip identification, security keys, device trim settings and more.

The fuse box for this test is designed by instantiating an eFuse box along with some custom logic. Figure 1 shows the block diagram of a typical eFuse box of size 64x32 (2K), which is built from two 1K eFuse arrays.

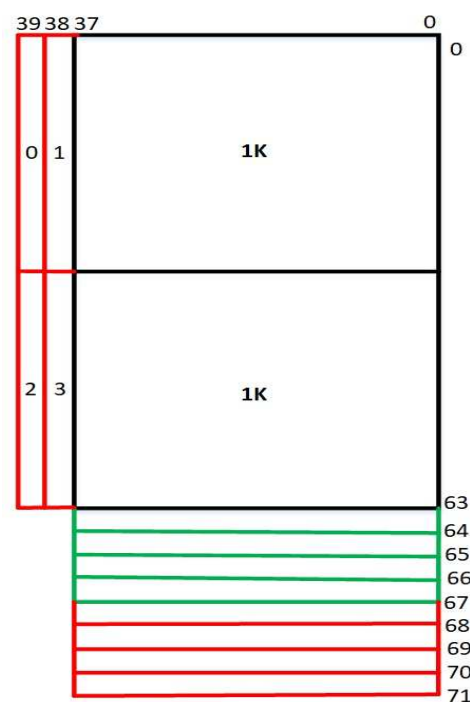


Fig. 1. eFuse Box Block Diagram

It consists of 72 rows and 40 columns that include 64 rows and 32 columns intended for data, 6 additional bits per row (37:32) for ECC to increase reliability, and 8 rows and 2 columns (highlighted in color) for repair to enhance yield. The 8 repair rows (71:64) and 2 redundancy columns (39:38) included in the fuse box provide space to store the data for 4 32-bit row repairs, 4 32-bit column repairs, and/or 16 single bit repairs in the red portion of Figure 1.

The data assignment in the top 4 repair rows (67:64) shown by the green region dictates the repair scheme of the fuse box. Due to flexible assignments in the repair scheme, only a subset of all possible repairs can be implemented depending on the failures observed. After every power on and reset operation, the fuse controller must read out the content of the repair scheme rows, before resuming any operations. The accompanying logic within the eFuse box will automatically apply any repairs specified in those rows.

3. METHODOLOGY TO ENABLE eFUSE REPAIR

3.1. Methodology Overview

The Tessent BISR controller operates in three different modes: autonomous mode, BISR chain access mode and fuse box access mode. In autonomous mode, the BISR controller has the capability to gather repair information from the BIRA registers, compress the content, and write it to the fuse box. In this mode it does not require any external data patterns from the tester. Thus, the built-in self-repair capability for memories is fully enabled by the autonomous mode. The other two modes can also be utilized to perform memory repair, but they are highly tester-interactive and predominantly used for diagnostic purposes. The BISR controller provides read and write access to the fuse box via the JTAG network when running in fuse box access mode and similarly provides access to the BISR chain in BISR chain access mode. These two modes can be leveraged to repair the fuse box.

For the BISR controller to repair the fuse box, its repair row addresses must be accessible for write and read. An additional address bit must be controlled to map to the repair rows (71:64). Helpfully, the Tessent Shell flow has the provision to define a fuse box size that exceeds the maximum number of fuses available for memory repair. The BISR controller always writes memory repair information to the maximum available fuse box address for memory repair. The remaining fuses can be accessed through the TAP in fuse box access mode. This feature can be used to enable access to the fuse box repair space. The example shown below is the Test Core Description (TCD) file for the eFuse interface that instantiates a 64x32 logical eFuse for data values as described in the previous section. The `number_of_fuses` variable is deliberately set as 4096 so that the additional repair rows can be accessed in fuse box access mode:

```
Core(FuseBoxInterface) {
  FuseBoxInterface {
    write_duration : 5us;
    read_duration : 600ns;
    init_duration : 1us;
    read_word_size : 32;
    programming_method : unbuffered;
    number_of_fuses : 4096;
  }
}
```

To differentiate between data and repair rows, the number of fuses available to store repair information must be specified by the following property during implementation:

```
set_config_value /DftSpecification(top,rtl)/MemoryBisr/Controller/ExternalFuseBoxOptions/number_of_fuses_for_repair
2048
```

With the above settings the eFuse box is enabled for repair through the BISR fuse box access mode.

3.2. eFuse Repair Flow

The eFuse repair flow is illustrated by the flow chart in Figure 2. When an MBIST pattern fails and it is determined that memories on the design are repairable, the autonomous repair process is followed as shown in Figure 2.

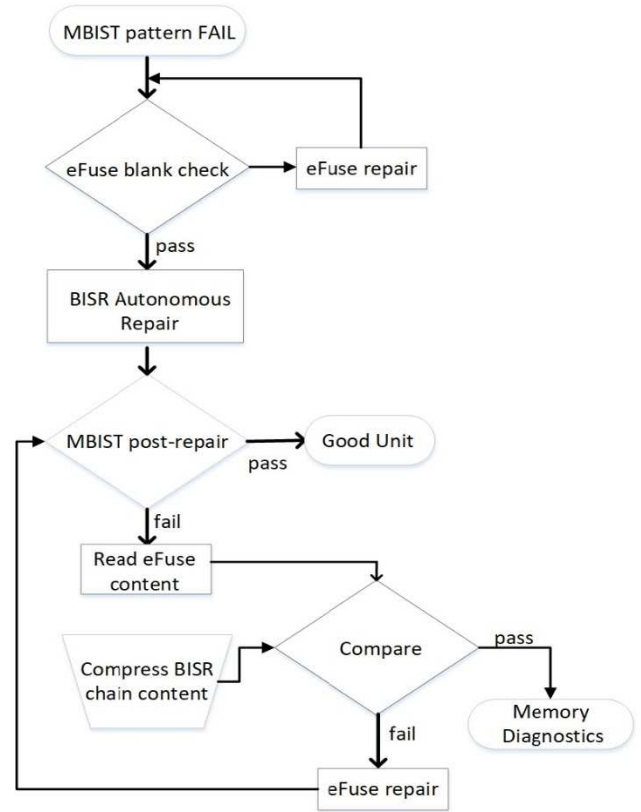


Fig. 2. eFuse Repair Flow

Prior to autonomous repair, it is important to sanity-check the contents of the fuse box. This can be achieved through a simple fuse box read address pattern. An example pattern spec for this is the following:

```
TestStep(FuseBoxRead) {
  MemoryBisr {
    run_mode : fuse_box_access;
    Controller(top_rtl_tessent_mbisr_controller_inst) {
      FuseBoxAccessOptions {
        operation : read;
        read_address(0 : max_repair_address) : 0;
      }
    }
  }
}
```

If the fuse box read returns a non-zero value, it has to be repaired. After the fuse box contents are ensured to be zero, the BISR pattern is executed and the post-repair MBIST pattern is checked. If MBIST is still failing, it is necessary to check if the fuse box contents match with the values that the BISR controller has programmed. For that, the BISR chain contents must be extracted and compressed externally. The BISR chain contents can be scanned out from the chip using the following pattern spec:

```

TestStep (BisrChainAccess) {
MemoryBisr {
run_mode : BisrChainAccess;
Controller (BP0) {
BisrChainAccessOptions{
enable_bira_capture : on;
select_bisr_registers : external;
}
BisrRegisterAccessOptions{
read_value(all_bits) : all_zero ;
}
}
}
}
}

```

The CompressBisrChain utility script provided by the Tessent tool can be used to compress the BISR chain contents using the same compression algorithm applied by the BISR controller. The inputs to the script are a configuration file with the BISR chain settings and a simulation log file. The configuration file can be generated using the “write_memory_repair_dictionary” command in the Tessent flow. The simulation log file must be created based on the chain contents in the following format:

```

Previous Compare: pin tdo = <path to BISR Reg>.ShiftReg[0]
Previous Compare: pin tdo = <path to BISR Reg>.ShiftReg[1]
Previous Compare: pin tdo = <path to BISR Reg>.ShiftReg[6]
Previous Compare: pin tdo = <path to BISR Reg>.ShiftReg[8]

```

The BISR registers and the bit positions holding a value of 1 are captured in the above format. It is required for the script to identify repair values in the BISR chain. The script outputs the pattern spec file to read the fuse box addresses that are programmed by the BISR controller during autonomous mode:

```

TestStep( ProgramFuseBox ) {
MemoryBisr {
run_mode : fuse_box_access;
Controller(top_rtl_tessent_mbisr_controller_inst) {
FuseBoxAccessOptions {
operation : program;
read_address(<addr1>) : 1;
read_address(<addr2>) : 1;
...
}
}
}
}

```

The fuse values at addresses that are not specified must be zero. Based on that, the spec can be modified to generate a pattern that can validate the contents of the fuse box.

3.3. eFuse Repair Strategy

If the validation of the fuse box values against the compressed BISR chain contents fails, a repair strategy must be executed based on the number of spare resources available and the faulty addresses identified. eFuse recovery is done by fuse repair and fuse redundancy. In fuse redundancy, the entire fuse row or column is replaced with the redundant row or column.

The fuse repair scheme must be configured to achieve fuse recovery and fix the failing bits. The repair rows allow for multiple configurations of repairs. The repair scheme that produces highest yield is determined by the foundry based on the faults analyzed from various silicon samples.

When the failures within the fuse array are scattered across multiple rows, using single-bit repairs is preferred. When the failures within the fuse array are within the same row, using row redundancy is preferred. Similarly, when the failures within the fuse array are within the same column, using column redundancy is preferred. In case multiple repair schemes are targeting the same failure bits, only one scheme should take precedence.

After the fuse failure data and their locations are collected, the repair is performed in the order of column, row and then bit. The only exception is when the number of bits failing in the same row is higher than the number of bits failing in the same column. In that case rows are replaced first. The repair loop is iterated until all the failures have been repaired or the resources have been exhausted. If the same bit to repair exists in more than one repair set, the last repair will take effect. For security reasons it is advisable to disable the unused redundant elements that are not utilized after successful fuse repair. Figure 3 below shows the column-first repair strategy which is most recommended repair flow.

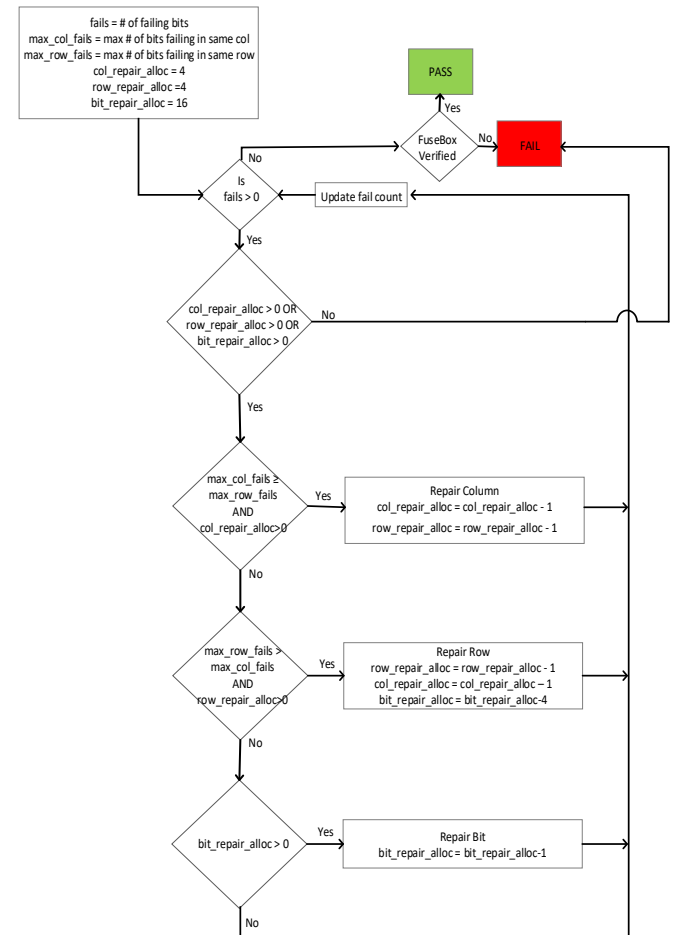


Fig. 3. eFuse Repair Strategy

Once the number of spare elements required to repair the fuse is identified, a repair signature must be programmed into the repair rows to map the spare element to faulty ones. The exact repair signature depends on the repair scheme of the eFuse box and can be obtained from the fuse box datasheet. For every repaired bit there will be a corresponding data bit which also needs to be programmed as it replaces the faulty bit. The following pattern spec can be used to program both the repair signature into repair rows and the valid values into the redundant resources:

```
TestStep( ProgramFuseBox ) {
  MemoryBisr {
    run_mode : fuse_box_access;
    Controller(top_rtl_tessent_mbisr_controller_inst) {
      FuseBoxAccessOptions {
        operation : program;
        write_address : <repair address>;
      }
    }
  }
}
```

4. ECC IMPLEMENTATION OVERVIEW

4.1 BISR Interface

Implementing BISR with Tessent MemoryBIST requires creating a custom interface to the eFuse box. The interface acts as an intermediary between the BISR controller and the eFuse. For that reason, it is referred to as a “BISR bridge”. It interprets the BISR operations and generates the necessary signals that drive the fuse box.

4.2 ECC Module Requirements

The ECC module for the BISR system allows the system to interface seamlessly with the eFuse with expanded functionality. The requirement for such a solution emerged from the following issues with the industry standard BISR bridge Soft IP, which:

- Uses a timing-sensitive serial streaming interface that cannot be stalled.
- Expects to interface directly with an eFuse box.
- Cannot differentiate between data and ECC bits.

This resulted in the requirements for the ECC Module:

- Be functionally transparent to the BISR controller.
- Generate the checksum with minimal input from or modification to the BISR bridge.
- Perform the decoding and correction operations independently from the eFuse bridge.

4.3 ECC Module Features

The system, in exchange for minor additional area and power overhead, confers the following additional features:

- The ECC checksum is calculated and decoded with single-cycle latency, allowing for seamless integration with the BISR controller.
- Automatic single-bit ECC correction is applied within 2 DFT logic clock cycles to the output data bus without any preamble or intervention from the BISR controller.

To enable these features, the ECC module employs 2 separate state machines:

- The ECC encoder initiates on demand and calculates the checksum for each data word in a single cycle. It then hijacks the programming enable signal to conditionally write the checksum into the ECC data columns.
- The ECC decoder compares every read data word against its existing parity checksum and provides single-bit correction where required.

5. ECC MODULE

5.1 System-Level Overview

The ECC module comprises an intermediate logic block that handles encoding, decoding and correction operations with minimal input from the BISR system. In essence, the 2 state machines operate independently with the encoder controlling the programming data bus and the decoder controlling the read data bus. The result is that for functional purposes the BISR bridge “sees” a 64x32 logical eFuse block instead of the actual 64x38 bit implementation and, aside from some programming overhead is unconcerned with the implementation details. This distinction is illustrated below in Figure 4.

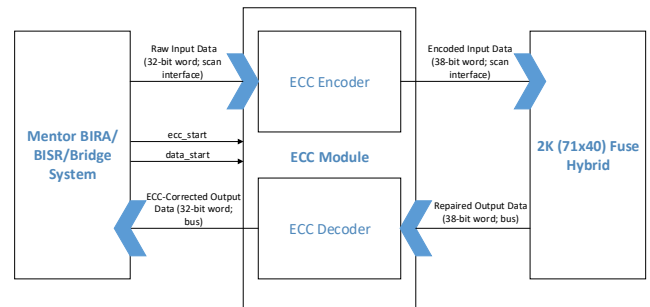


Fig. 4. System-Level Block Diagram

5.2 Additional Usage Requirements

An elaboration on the different usage requirements as compared to those for a normal eFuse box is necessary:

- Following an ordinary power-on and reset sequence the ECC sequence is ready to begin.
- The BISR bridge asserts `ecc_start` and `data_start` signals to coordinate with the ECC module, then performs dummy writes to all the ECC bits which are propagated as necessary.
- The ECC decode and correction sequences execute autonomously. If an error exists and can be corrected the correction is automatically applied to the output data bus.
- If the error was un-correctable, it is guaranteed that BISR system will eventually error out, eliminating the necessity for an external flag and reducing overall system complexity.

6. HAMMING (38,32) CHECKSUM

The employed ECC encoding scheme was chosen as a simple proof-of-concept to aid debugging. The Hamming code originated in the 1950s and is a simple binary linear error-correcting code with a minimum distance of 3. This specific implementation is extended from the general algorithm to cover the 32 data bits in each row.

Due to its limited redundancy, Hamming codes are best used in memory applications where bit errors are rare making it particularly suitable for an eFuse box. Another advantage of the simple algorithm is that it allows for the encoder to be implemented in efficient logic that can correct single-bit errors in the output in just 2 cycles, allowing for seamless operation with the BISR bridge.

The encoder generates a 6-bit checksum for the data in each 32-bit data word, which is stored in the 6 additional MSBs of each eFuse row (bit positions 37:32). It is this checksum along with the data word that will be used by the ECC Decoder to detect any errors and correct any single-bit failures. This scheme is illustrated below in Figure 5.

Bit position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
Encoded data	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25	d26	p32	d27	d28	d29	d30	d31	d32
Fuse data	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25	d26	d27	d28	d29	d30	d31	d32	p1	p2	p4	p8	p16	p32

Fig. 5. ECC Encoding Scheme

7. ECC MODULE OPERATIONS

7.1 High-Level Operation

After power on and reset, the ECC module enters a wait state. Whether the encoder or decoder will be activated next depends on whether the BISR controller is initiating a programming or read operation, which can be determined by comparing the input signals which are passed through the module. The subsequent procedures will be expanded on but proceed as illustrated in Figure 6.

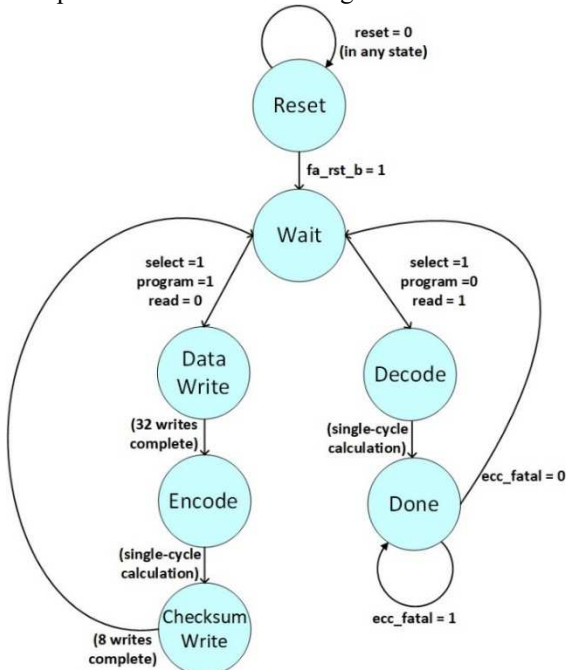


Fig. 6. ECC Module State Diagram

7.2 Write Procedure

The eFuse write operation involves the use of asynchronous pulses to coordinate transactions between the BISR bridge and ECC encoder. The high-level functions for the 5 state sequences of the ECC encoder state machine are as follows:

Reset State (triggered by chip-level reset):

1. Reset all local variables.
2. Go to the Wait State.

Wait State:

1. ECC encoder is stalled until an input is received.
2. eFuse bridge asserts program signal.
3. Go to the Data Write State.

Data Write State:

1. eFuse bridge performs write to data bits.
2. Go to the Encode State.

Encode State:

1. Checksum calculated and available in next cycle.
2. Go to the Checksum Write State.

Checksum Write State:

1. eFuse bridge performs write to all checksum bits.
2. ECC encoder hijacks program signal as required.
3. Go to the Wait State.

7.3 Read Procedure

The eFuse read operation involves a fully autonomous error detection and correction sequence. The high-level functions for the 4 state sequences of the ECC decoder state machine are as follows:

Reset State (triggered by chip-level reset):

1. Reset all local variables.
2. Go to the Wait State.

Wait State:

1. ECC decoder is stalled until an input is received.
2. eFuse bridge asserts read signal.
3. Go to the Decode State.

Decode State:

1. Read the fuse data_out (37:0) bus.
-If no error is detected, pass data to eFuse bridge.
-If an error is detected, and is correctable, invert the offending bit and pass data to eFuse bridge.
-If an error is detected and not correctable, set an ecc_fatal flag 1.
2. Go to the Done State.

Done State:

1. If the ecc_fatal flag has been asserted the decoder will remain locked in this state.
2. Otherwise, go to the Wait State.

8. RESULTS AND BENCHMARKING

This reliability-enhancing approach was benchmarked against a more conventional 2x eFuse bitcell redundancy approach which would typically incur an area overhead of roughly 50% in the eFuse box. This approach required 17% more bitcells (resulting in an area overhead of 9%) and a few thousand conventional logic gates, for a total area overhead of roughly 10%.

While some test time overhead is naturally incurred as a consequence of the presence of 6 additional ECC checksum bits per 32-bit data row, in practice this amounted to roughly 20% versus an unmodified eFuse box in write operations and negligible overhead in read operations or power on and reset. This number, while significant, is at least in part offset by the dramatically reduced area overhead. Moreover, it can be further reduced with more efficient ECC encoding than the proof-of-concept Hamming scheme or by further automating the ECC module's operations.

The failure mode is built on the expectation that a 2-bit error in each row is an acceptable device end-of-life condition. Given the low incidence of these errors in contemporary eFuse implementations under normal operating conditions, the inclusion of ECC functionality can dramatically extend the usable lifespan of an eFuse box.

The design of the ECC module was a one-time effort for a specific technology node but can be easily proliferated to contemporary technologies. Meanwhile, the repair flow is straightforward enough to port to the very latest Tessent MBIST tool flows or other similar solutions.

9. CONCLUSION

eFuses play a crucial role in contemporary memory repair schemes. The growing percentage of memories on chip is resulting in more eFuse boxes in designs. To address the emerging eFuse yield issues redundancy techniques are used that typically provide an additional fuse for every data bit stored. Although the latest eFuse boxes may support built-in ECC and repair features, they are mostly not being utilized due to existing EDA tool limitations.

This paper presents a novel solution to enable ECC and repair features in eFuse macros for reliability-enhanced applications. The ECC module described in the paper can be customized for any ECC algorithm and will seamlessly interface with a minimally modified BISR controller.

The proposed methodology can eliminate the need to wastefully implement fuse boxes with excessive redundancy and thus significantly reduces the required design overhead. With some changes, the approach can be expanded on for diverse applications across the industry.

10. REFERENCES

- [1] Dae-Hyun Kim, Linda S. Milor, "Memory yield and lifetime estimation considering aging errors", 2015 IEEE International Integrated Reliability Workshop.
- [2] V.Sridhar, M.Rajendra Prasad, "Built-in self-repair (BISR) technique widely Used to repair embedded random access memories (RAMs)", International Journal of Computer Science Engineering (IJCSE), Vol. 1 September 2012.
- [3] Harsharaj Ellur, Kalpesh Shah, "A Tag based solution for efficient utilization of efuse for memory repair", INTERNATIONAL TEST CONFERENCE, 2014.
- [4] Norm Robson, John Safran, Chandrasekharan Kothandaraman, Alberto Cestero, Xiang Chen, Raj Rajeevakum, Alan Leslie, Dan Moy, Toshiaki Kirihata, Subramanian Iyer, "Electrically Programmable Fuse (eFUSE): From Memory Redundancy to Autonomic Chips", IEEE Custom Integrated Circuits Conference, 16-19 Sept. 2007.
- [5] Tessent MemoryBIST User's Manual, Software Version 2017.4, December 2017.
- [6] Bejoy Oomann, Maddu Karunaratne, "An Optimal Memory BISR Implementation", JOURNAL OF COMPUTERS, VOL. 8, NO. 9, September 2013.
- [7] Akhil Garg, Prashant Dubey, "Fuse Area Reduction Based on Quantitative Yield Analysis and Effective Chip Cost", 2006 IEEE International SOC Conference.
- [8] Samvel Shoukourian, Valery Vardanian, Yervant Zorian, "SoC Yield Optimization via an Embedded-Memory Test and Repair Infrastructure", IEEE Design and Test of Computers, Volume 21, Issue 3, May-June 2004.
- [9] Shine Chung, Wen-Kuang Fang, "I-fuse: A Disruptive OTP Technology with Excellent Manufacturability", Joint Symposium 2015 – eMDC and ISSM.
- [10] Divya H, Arunkuma P Chavan, Shubha Krishnamurthy, "Design and Verification of ECC Scheme to Optimize Area and Tester Time in OTP ROM Controller", RTEICT – 2019.