SUSTAINABLE FAULT MANAGEMENT AND ERROR CORRECTION FOR NEXT-GENERATION MAIN MEMORIES.

[1]Guguloth Mounika,

M. *Tech, VLSI System Design, Student,*
Department of Electronics and Communication Engineering,

JNTUH University College of Engineering Sulthanpur, Sangareddy, Telangana,INDIA.

Email: gugulothmoinika123@gmail.com

[2]Mr.K. Prabhu,

Assistant Professor(C), Department of Electronics and Communication Engineering,

JNTUH University College of Engineering Sulthanpur, Sangareddy, Telangana,INDIA.

Email: kprabhu2003@gmail.com

ABSTRACT

The authors propose a sustainable and FPGA-implementable approach to error correction in deeply scaled memories, SFaultMap. The novel fault map architecture optimizes operational and embodied energy, achieving lower energy consumption over a five-year lifetime under moderate to high fault rates. SFaultMap+ enhances fault tolerance by shifting focus from error correction overhead to system-level sustainability and hardware efficiency. The modular Verilog design allows real-time fault masking, scalable support for continuation segments, fault threshold tuning, and fast decoding. The architecture has been validated through functional simulation and synthesis using Xilinx Vivado.

***Index Terms:* - SFaultMap, memory fault tolerance, fpga, energy-efficient design, error correction, continuation segment.**

I.INTRODUCTION

The global energy demand has increased significantly due to the growth of modern computing systems, with memory subsystems playing a crucial role in determining the performance, reliability, and energy efficiency of computing infrastructure. Memory technologies like DRAM and emerging alternatives such as PCM, STT-MRAM, and ReRAM are playing a central role in determining the performance, reliability, and energy efficiency of computing infrastructure.

Traditionally, research and optimization efforts have focused on reducing operational energy, but now the focus is shifting to embodied energy, which encompasses the energy used in various processes such as extraction of raw materials, wafer fabrication, photolithography, chemical processing, assembly, testing, packaging, and distribution.

Recent studies indicate that for advanced DRAM modules used in hyperscale data centers, Eembodied can exceed 50% of Etotal, particularly when modules are discarded early due to isolated hardware faults that could be managed with effective error correction mechanisms. This realization has led to a redefinition of memory sustainability: it is no longer sufficient to minimize runtime power alone; true sustainability demands systems that are fault-tolerant, repairable, and long-lived, thus reducing frequent replacement and lowering overall environmental impact.

As memory technologies scale to nanometer regimes, they encounter increased vulnerability to physical faults and manufacturing-induced defects. To address these limitations, several emerging memory technologies (EMTs) have been introduced, offering new trade-offs in density, energy efficiency, and non-volatility. These include Phase-Change Memory (PCM), Spin-Transfer Torque Magnetic RAM (STT-MRAM), and Resistive RAM (ReRAM).
Fault management must evolve to operate at finer granularities, recording defects at the bit or word level using fault maps. This allows higher-level logic to dynamically adapt. Integrating robust Error-Correcting Codes (ECC) and repair-based mechanisms is essential for maintaining high reliability without compromising density or energy efficiency.

Memory fault management is also pivotal in edge devices and IoT systems, where tight energy and area constraints prohibit over-provisioning. Lightweight error detection and graceful degradation, powered by fault-awareness, help ensure continued operation under degraded conditions.

In conclusion, sustainable main memory design must embrace the reality that faults will become more frequent as technology scales. The strategy should be to detect, contain, and adapt to them, prolonging useful life and reducing energy per computation. This calls for cross-layer innovation from circuit design to system software.

II.LITERATURE SURVEY

The relentless pursuit of advancements in computing systems, driven by demands for higher performance and greater data processing capabilities, has inadvertently led to a significant increase in global energy consumption. A considerable and growing portion of this energy is attributed to main memory (DRAM) systems. Early research efforts primarily focused on mitigating "use phase" energy consumption, which refers to the power consumed during a system's active operation. While substantial progress has been made in optimizing operational energy efficiency, more recent studies have highlighted a critical, often overlooked, aspect of the total energy footprint: "embodied energy" [1], [2], [3], [4]. Embodied energy encompasses the energy expended throughout the entire lifecycle of a product, from raw material extraction, manufacturing, and transportation to assembly and eventual disposal. Crucially, for integrated circuits (ICs) and particularly for progressively larger memory systems, embodied energy constitutes a significant and rapidly increasing contribution to the lifetime energy consumption [10]. Life-cycle assessments have identified semiconductor manufacturing as a dominant factor in the environmental impact of modern computing, with the energy consumed during CMOS fabrication steadily rising since the 130nm node. Similarly, while DRAM technology lags slightly, its energy per area is consistently higher than CMOS and has also begun to climb [10]. This growing embodied energy fraction necessitates a shift in design philosophy; truly sustainable system and memory choices must integrate manufacturing impact into energy-related decisions, moving beyond sole optimization of operational energy. Therefore, the development of sustainable reliability solutions that concurrently address both embodied and operational energy is paramount.

Beyond energy considerations, the aggressive scaling of memory technologies, driven by the desire for higher density and lower cost, introduces formidable challenges to memory reliability. In conventional DRAM, as feature sizes shrink, cells become more prone to various fault mechanisms. Wordline crosstalk faults, for instance, occur when "weak" cells in a victim row experience premature charge drainage due to repeated charging of adjacent wordlines without their own cells being refreshed [5]. Similarly, bitline crosstalk manifests as fluctuations in the final read bit values, originating from voltage levels of bitlines connected to the same sense amplifiers, often exacerbated by process variation and minor defects within the memory cell structure [6]. Another significant concern in DRAM is reduced retention time, where data stored in memory cells leaks prematurely, also a result of process variation [11]. This directly conflicts with efforts to reduce refresh power and has led to proposals for selectively expanding refresh intervals in conjunction with additional error correction mechanisms [12].

The landscape of memory technologies is further diversified by the advent of emerging memory technologies (EMTs), designed to overcome the fundamental limitations of DRAM, such as volatility and high power consumption [7]. EMTs like phase-change memory (PCM) and spin-transfer torque magnetic random-access memory (STT-MRAM) are being explored for future main memories due to their non-volatility, potential for increased density, and reduced energy consumption [7]. However, these promising technologies introduce their own distinct reliability challenges. PCM, for example, faces endurance limitations, with write cycles typically restricted to between 108 and 109 cycles, which often result in permanently "stuck-at" faults [7]. Early failures of weak cells due to process variation have also been observed in PCM, underscoring the necessity of robust fault tolerance throughout the lifetime of scaled devices. Similarly, STT-MRAM can be impacted by process variation, leading to individual cell susceptibility to read disturbance and false reads due to poor sensing margins [13]. As the number of faulty cells is projected to increase across both conventional and emerging memory technologies with shrinking technology nodes, scalable and sustainable fault maps that meticulously record individual faulty

cells are becoming increasingly indispensable for maintaining data integrity and system longevity.

In response to these pervasive reliability concerns, various error correction schemes have been developed and widely applied. Error Correction Codes (ECC) are a cornerstone solution for addressing general faults and single event upsets. Single-bit error correction double-bit error detection (SECDED) using Hamming codes, including direct application in DRAM with "chipkill" capabilities, is a common implementation [14]. For low error rates, an additional bank of redundant memory locations, often referred to as row sparing, can be employed, either as a standalone solution or in combination with ECC and other error correction schemes.

More specialized fault tolerance approaches have also emerged. Error Correction Pointers (ECP) represent a significant class of solutions, initially proposed for PCM and subsequently adapted for DRAM [8]. In ECP, each memory row includes pointers to faulty bit locations within that row, along with a bit to store the correct value at the pointed location. Other methods include partition and flip schemes that attempt to encode data to avoid stuck-at faults. To combat wordline crosstalk faults, a probabilistic adjacent row activation method has been proposed, which involves refreshing impacted neighboring rows through a weighted randomization [5]. Complementary to this, counter-based refreshing mechanisms are used to detect "hot rows" that require more frequent refreshing [15]. For bitline crosstalk, the Periodic Flip Encoding (PFE) method combines encoding with knowledge of weak cell locations to bypass problematic patterns [16].

A notable architectural scheme for DRAM correction is ArchShield, which employs a word-level fault map [9]. In ArchShield, each memory row is tagged with two bits to indicate whether it has no faults, a single fault, or multiple faulty words. Rows containing faults are duplicated using row sparing, with multi-fault rows always accessing the redundant copy. Single-fault rows only resort to the redundant copy if an additional fault is detected in conjunction with SECDED ECC. While ArchShield operates effectively for incident fault rates between 10−6 and 10−4, demonstrating space efficiency and minimal performance degradation (typically less than 2%), its overhead increases super-linearly at higher error rates, rendering it non-viable in such scenarios [9].

In contrast to these existing solutions, the presented paper proposes the first centralized fault map with bit-level granularity, termed SFaultMap, explicitly designed for deeply scaled next-generation memories. A key differentiator of SFaultMap is its sensitivity to the projected disproportionate contribution of embodied energy in these highly scaled devices [4], [10]. It achieves this by making strategic, albeit seemingly unconventional, tradeoffs, accepting small concessions in use-phase energy and performance to enable solutions with a minimal area footprint, thereby optimizing for holistic energy consumption. This focus on reducing the physical area occupied by the fault map directly translates to lower embodied energy, positioning SFaultMap as a truly sustainable solution for the increasing fault rates and complex reliability challenges faced by the next-generation of main memories. SFaultMaps are particularly advantageous for next-generation memories due to their ability to provide a very compact, bit-level fault representation. This compact design results in a significantly lower area footprint compared to existing solutions like Error Correcting Pointers (ECP) [8] and ArchShield [9], directly leading to reduced embodied energy, which is a critical concern as technology scales. The document demonstrates that SFaultMap is consistently more energy-efficient than ECP at moderate to high fault rates (e.g., 10−2 and higher), even when ECP would not guarantee fault tolerance. Additionally, within ArchShield's intended fault-incidence range, SFaultMap+ schemes exhibit significantly lower area overheads, making SFaultMap a more scalable and efficient choice for managing increasing fault densities in future memory generations. Even with performance optimizations in SFaultMap+ (e.g., offset segment lookup and a fault-free flag bit), which slightly increase embodied energy, the operational energy advantages often make SFaultMap+ the more sustainable solution within typical usage lifetimes, particularly for memory-dependent workloads. This adaptability and focus on holistic sustainability make SFaultMap a promising solution for the evolving reliability challenges of next-generation main memories.

III. EXISTING SYSTEM

Traditional memory fault tolerance schemes primarily rely on techniques such as Error Correcting Codes (ECC), Error Correction Pointers (ECP), and frameworks like ArchShield. ECC, especially SECDED and Chipkill, is widely adopted in DRAM to handle soft errors; however, its effectiveness significantly declines under multi-bit or burst fault conditions. Additionally, ECC introduces

non-negligible redundancy and offers no awareness of embodied energy.

ECP attempts to correct hard faults at the bit level using location pointers and replacement bits. While effective for isolated faults, its metadata overhead scales poorly with increasing fault rates, making it unsustainable in high-density memory arrays. Similarly, ArchShield employs a fault map at the word level, triggering row duplication and ECC-based fallback recovery. While moderately effective at low to medium error rates, its area and energy costs grow non-linearly under aggressive scaling.

IV PROPOSED SYSTEM

In contrast, the proposed SFaultMap architecture offers a sustainable, fine-grained, and energy-efficient solution. It uses compact {valid, position} encodings to track faults at the bit level and stores this metadata in segments with minimal overhead. Its correction logic is lightweight, using simple XOR operations, and the entire system is implemented in modular Verilog, optimized for FPGA synthesis.

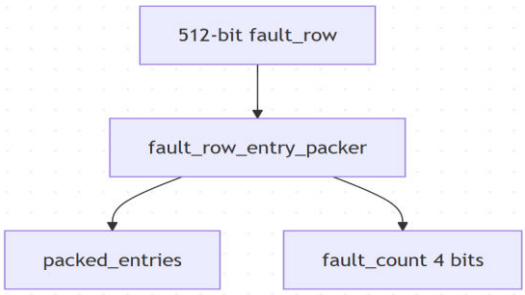
A key feature of SFaultMap is its sustainability-driven design, addressing both operational and embodied energy. It avoids unnecessary hardware redundancy and supports runtime bypass for fault-free rows (SFaultMap+), significantly reducing power consumption. The design is scalable, synthesis-friendly, and adaptable to the increasing fault densities found in future DRAM and emerging non-volatile memories.

V.SYSTEM DESIGN

The SFaultMap system is designed using a clean, pipelined modular architecture where each stage performs a precise role in encoding, storing, and correcting memory faults. The data flow progresses from raw faulty memory row inputs through compact encoding stages, into a memory storage module, and finally into correction logic for masking errors during runtime. Each module is carefully crafted to ensure scalability, low overhead, and synthesis-friendly implementation on modern FPGA or ASIC platforms. This modularity not only simplifies verification and debugging but also allows for selective upgrades or extensions, such as enabling continuation segments for rows with high fault density or integrating advanced fault prediction mechanisms based on historical data patterns.

The system begins with a fault identification and extraction unit that transforms raw row data into a structured fault representation. This representation is further compressed and encapsulated into a segment format that is both storage-efficient and easy to decode. By separating the encoding and correction responsibilities into distinct modules, the SFaultMap system ensures that fault management can be both proactive (during memory testing) and reactive (during runtime accesses). The FSM controller seamlessly orchestrates this entire pipeline, transitioning between states that correspond to fault packing, segment writing, and post-write handling. Moreover, the design maintains compatibility with real-time memory correction workflows by enabling the decoder and correction unit to operate independently of the fault encoder.

What distinguishes this architecture is its support for bit-level granularity, enabling precise fault mapping instead of coarse row- or block-level marking. This fine-grained fault tracking significantly improves memory utilization and longevity, especially in systems where permanent bit-level faults are prevalent due to technology scaling or aging. Furthermore, the correction process is lightweight, involving only an XOR operation with a fault mask, which can be derived from decoded entries this keeps both latency and energy consumption low. Collectively, these features make the SFaultMap methodology a robust and sustainable solution for managing faults in next-generation high-density memory systems, whether used standalone or alongside conventional ECC mechanisms.



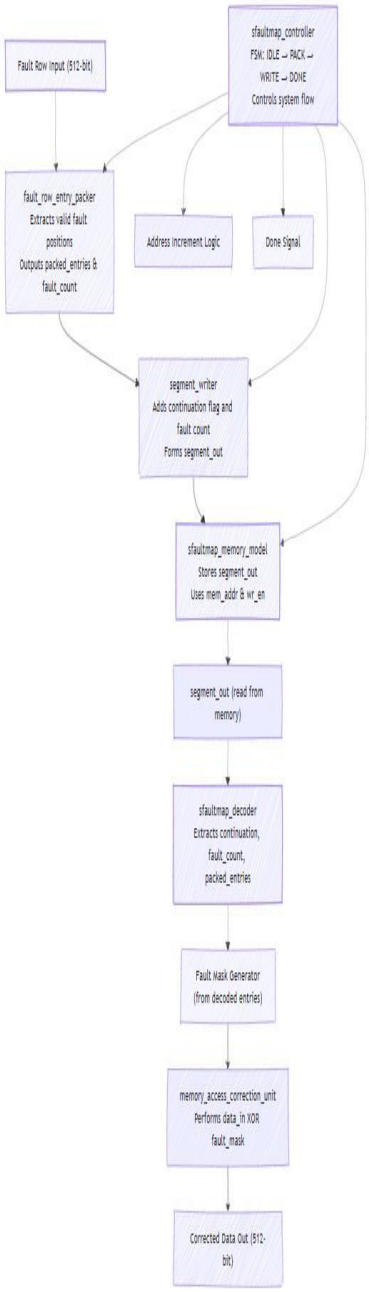


Fig: Block diagram for the implemented design.

Fault_row_entry_packer

The pipeline is fault_row_entry_packer, responsible for scanning a 512-bit memory row and identifying which specific bits are faulty. Each bit in the input vector fault_row represents the health of one memory cell—if the bit is 1, it signifies a fault at that position. The module sequentially checks all 512 bits and records the positions of up to MAX_FAULTS_PER_ROW faults. Each fault position is paired with a 1-bit valid flag, and these {valid, position} entries are then packed into a flat vector. The total size of this packed fault representation depends on the fault pointer width PTR_BITS and the number of allowed faults per row, as captured by the equation:

Packed_Width=MAX_FAULTS_PER_ROW×(PTR_BITS+1)

Fig: Flow of Fault_row+entry_packer

Segment writer

Once the fault entries have been packed, the segment_writer module formats this data into a segment suitable for memory storage. This module accepts the packed fault entries, the 4-bit fault count, and a single-bit continuation flag as inputs. The continuation flag indicates whether this segment continues from a previous fault entry set, which is useful for fault rows with more than the allowed number of faults. Although the continuation logic is unused in the simple implementation, the hardware is prepared for this feature. The writer combines all input fields into a single vector output called segment_out. The total width of this segment is defined by the equation:

Segment_Width=1 (continuation)+4 (fault count)+Packed_Width

Sfaultmap_memory_model

The segment_out vector generated by the writer module is passed to sfaultmap_memory_model, which is a synchronous memory array specifically used to store fault map segments. This module simulates on-chip memory or a register file where each entry corresponds to one memory row's encoded fault data. It supports standard memory access with inputs for write enable (wr_en), data (data_in), and address (addr), and it provides a registered read output (data_out). This module ensures that once fault entries are computed and packed, they can be persistently stored and accessed later for correction during read operations. Addressing is controlled externally, often incremented via an FSM, and allows seamless integration into row-based memory mapping.

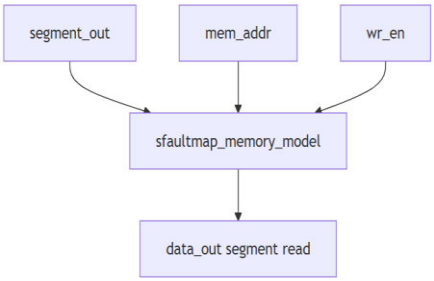


Fig: sfault map memory model implementation

sfaultmap_decoder

Once stored fault data is retrieved from memory, it must be decoded back into actionable correction information. This is the role of the sfaultmap_decoder module. It receives the encoded segment vector and unpacks it into its

three original components: the continuation flag (1 bit), the fault count (4 bits), and the packed entries vector. These fields are extracted using fixed bit slicing. The decoder enables downstream logic, such as correction mask generators, to reconstruct the original set of fault locations. The logical breakdown of its unpacking process is described by the expression:

segment_in={continuation,fault_count,packed_entries}

memory_access_correction_unit

The final step in the pipeline is applying correction logic to incoming memory data. This is performed by the memory_access_correction_unit. It receives raw data (data_in) and a correction mask (fault_mask) as input. The correction mask is generated based on the decoded fault positions and specifies which bits need to be flipped to correct known faults. This module uses a bitwise XOR operation between the raw input and the mask to produce corrected data. The correction logic is simple yet effective and follows the Boolean equation:

data_out=data_in⊕fault_mask

sfaultmap_controller

The sfaultmap_controller is the centralized FSM that manages and synchronizes all other modules. It controls the system flow, beginning from the idle state and advancing through the PACK, WRITE, and DONE stages in response to the start signal. During the PACK state, the controller activates the fault_row_entry_packer to identify faults. It then enables the segment_writer and asserts write enable signals to write the packed segment into the memory model during the WRITE stage. After successful storage, it transitions to DONE, asserting a done flag to signal operation completion. The FSM also manages memory addresses, incrementing them after each write to support sequential row access. Though simplified, the FSM can be extended to support more complex behaviors like multi-segment continuation, dynamic thresholds, or runtime re-correction.

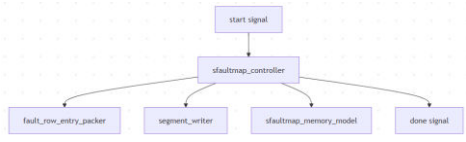


Fig: sequence of actions performed by the controller.

The flow is a controlled pipeline starting from fault detection in a memory row packing fault entries writing the fault segment into memory signaling completion. The sfaultmap_controller is the brain managing all these steps, ensuring correct timing and data flow between modules.

This modular and sequential design enables efficient fault mapping, compact storage, and streamlined control for scalable fault management in next-generation memories.

VI. RESULTS

The implementation of the proposed SFaultMap system was carried out in Verilog-2001, targeting FPGA synthesis using Xilinx Vivado 2018.2. The architecture comprises modular units responsible for memory fault row encoding, segment formatting, fault map storage, and runtime correction. This section presents the **simulation** obtained from validating each submodule, with emphasis on **functional correctness**, **packing logic**, **memory model behavior**, and **controller FSM sequencing**.

Functional Verification

The **SFaultMap controller** was verified to perform all intended operations in a pipelined and deterministic manner. Simulation tests were performed for multiple fault patterns, verifying correctness in encoding, segment generation, and proper memory storage through the following steps:

Fault Detection and Packing:

The input signal fault_row (512 bits wide) was simulated with synthetic fault injections (bit positions set to 1). The fault_row_entry_packer module accurately detected the faulty bits and stored their locations in the compressed form {valid_bit, position} using an internal loop. The output packed_entries and fault_count were checked against

expected values using waveform monitoring and test assertions.

$$\text{packed\_entry}_i=\{\text{valid}_i,\text{position}_i\},0\le i<\text{MAX\_FAULTS\_PER\_ROW}$$

Segment Formation and Metadata Encoding:

The segment_writer module then formats the data into a 1-bit continuation flag, a 4-bit fault count, followed by the packed fault entries. The segment is defined as:

$$\text{segment\_out}=\{\text{continuation},\text{fault\_count}[3:0],\text{packed\_entries}\}$$

For instance, if 3 faults were detected at positions 5, 19, and 88, the segment output correctly showed the packed binary encodings of these indices with valid bits prepended.

Fault Segment Storage to Memory:

The sfaultmap_memory_model successfully stored and retrieved fault segments. Simulations verified that memory write operations occurred only during the WRITE state, controlled by the FSM within sfaultmap_controller. The signal mem_wr_en enabled storage of segment outputs at memory locations indexed by mem_addr. Read-back verification ensured no corruption or overwrites occurred during sequential operations.

Controller FSM Behavior:

The sfaultmap_controller transitions through four states: IDLE → PACK → WRITE → DONE → IDLE. The start signal triggers the flow, and the controller automatically generates the packed data, stores it in memory, and asserts the done signal at completion. Internal counters and segment buffers were cleared at each transition.

FSM Transition Table:

Current State	Condition	Next State
IDLE	start = 1	PACK
PACK	(automatic)	WRITE
WRITE	(automatic)	DONE
DONE	(automatic)	IDLE

Fault Segment Decoding and Runtime Correction:

To validate decoding, the sfaultmap_decoder parsed the previously written segment, extracting the continuation, fault_count, and packed_entries. These were compared against expected results from the encoder to confirm bit-accurate reconstruction.

Lastly, the memory_access_correction_unit correctly performed bitwise XOR correction:

$$\text{data\_out}=\text{data\_in}\oplus\text{fault\_mask}$$

Test inputs demonstrated that corrupted memory values were corrected precisely at the expected fault locations, completing the end-to-end verification of the fault encoding and correction pipeline.

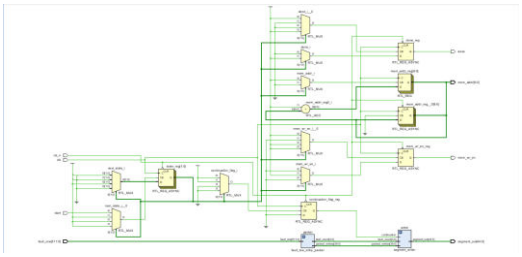


Fig: Implemenetd rtl schematic of the proposed design.

Input data injected:

512'h00000000000000000000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000000
00000000000000000000000000000800,

Input Original data:

512'h00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000012345678

The first input represents a fault_row with only bit position 11 set to 1, indicating a fault at that position:

fault_row=211=0x00000000000000000000000000000000000000000800

The second input is a valid data_in word with the value:

data_in=0x00000000000000000000000000000000000000000000000000000000000000000000000012345678

This test aims to validate the pipeline's ability to encode the fault, store it, decode it, generate a fault mask, and correct the corrupted bit in data_in.

1. Fault Packing Stage (fault_row_entry_packer)

The packer scans the fault_row, identifies the first and only set bit at position 11, and encodes it using the format:

packed_entry={valid,position}={1′b1,9′d11}=10′b1_00001011=10′h10B

Since only one fault is detected:

fault_count=1

The output becomes:

$$\texttt{packed\_entries} = \underbrace{10'h10B}_{\text{Entry 0}}\ \underbrace{10'h000}_{\text{Entry 1}}\ \underbrace{10'h000}_{\text{Entry 2}}\ \underbrace{10'h000}_{\text{Entry 3}}$$

2. Segment Formation Stage (segment_writer)

The writer constructs a memory segment using:

segment_out={continuation,fault_count, packed_entries

Substituting the values:

segment_out={1′b0,4′b0001,10′b100001011,30′b0}

=45′b0_0001_100001011_00000000000000000000000000000000

This becomes:

segment_out=45′h082C000000

This binary-encoded fault information is then written to the memory model.

3. Memory Read & Decode (sfaultmap_decoder)

The stored segment is read and decoded. Bit slicing retrieves:

Continuation flag:

continuation=0

Fault count:

fault_count=4′b0001=1

Packed entries:

packed_entries[9:0]=10′b100001011={1′b1,9′d11

4. Fault Mask Generation

The decoder's output is used to generate a fault mask. Only one fault is active (valid=1), located at bit index 11:

$$\texttt{fault\_mask}[i] = \begin{cases} 1, & \text{if } i = 1 \\ 0, & \text{otherw} \end{cases}$$

Thus:

fault_mask=211=512′h00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000800

5. Correction (memory_access_correction_unit)

The correction logic uses XOR to flip the faulty bit:

data_out=data_in⊕fault_mask

With:

data_in=0x00000000000000000000000000000000000000000000000000000000000000000000000012345678

We flip bit 11:

data_out=0x00000000000000000000000000000000000000000000000000000000000000000000000012345E78

(Notice: 0x12345678 XOR 0x800 = 0x12345E78)

Thus, the corrected value has the fault bit flipped, and this is verified against the expected value to confirm system integrity.

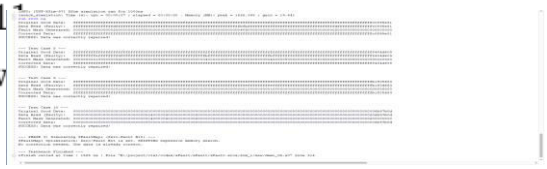


Fig:Design test for the 1-4 fault detection and corrected output.

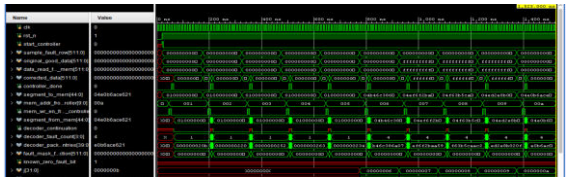


Fig: Fig:Design test for the 1-4 fault detection and corrected output in simulation.

REFERENCES

[1] A. Jones. Y. Chen, W. Collinge, H. Xu, L. Schaefer, A. Landis, and M. Bilec, "Considering fabrication in sustainable computing," ICCAD, 2013.

[2] M. A. Yao et al., "Comparative assessment of life cycle assessment methods used for personal computers.," Env. Sci. & Tech., Vol. 44, No. 19, 2010.

[3] P. Teehan and M. Kandlikar, "Comparing Embodied Greenhouse Gas Emissions of Modern Computing and Electronics Products," Env. Sci. & Tech., Vol. 47, No. 9, 2013.

[4] D. Kline Jr. N. Parshook, X. Ge, E. Brunvand, R. Melhem, P. K. Chrysanthis, and A. K. Jones, "Holistically Evaluating the Environmental Impacts in Modern Computing Systems," IGSC, 2016.

[5] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," ISCA, 2014.

[6] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," DSN, 2016.

[7] C. J. Xue, G. Sun, Y. Zhang, J. J. Yang, Y. Chen, and H. Li, "Emerging non-volatile memories: opportunities and challenges," CODES+ISSS, pp. 325-334, 2011.

[8] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," ISCA, pp. 141-152, 2010.

[9] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," ISCA, 2013.

[10] S. B. Boyd, Life-Cycle Assessment of Semiconductors. Springer, 2012.

[11] C. H. Kim, K. H. Lee, and H. J. Lee, "Retention-aware intelligent DRAM refresh," in ISLPED, 2009.

[12] C.-H. Lin, D.-Y. Shen, Y.-J. Chen, C.-L. Yang, and M. Wang, "SECRET: Selective error correction for refresh energy reduction in DRAMS," ICCD, pp. 67-74, 2012.

[13] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem, "Leveraging ECC to Mitigate Read Disturbance, False Reads and Write Faults in STT-RAM," DSN. pp. 215-226, June 2016.

[14] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory." IBM Microelectronics Division, pp. 1-23, 1997.

[15] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM" IEEE Computer Architecture Letters.

[16] M. Seyedzadeh. D. Kline Jr, R. Melhem, and A. K. Jones, "Mitigating Bitline Crosstalk Noise in DRAM Memories," MEMSYS, 2017.