Thesis/Project No.: CSER-22-17.

TOLERATING SOFT ERRORS IN TERNARY CONTENT ADDRESSABLE MEMORY

By

Abdullah-Al-Zobayer

Roll: 1607008

&

Samrat Alam

Roll: 1607022

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

March 2022

TOLERATING SOFT ERRORS IN TERNARY CONTENT ADDRESSABLE MEMORY

By

Abdullah-Al-Zobayer

Roll: 1607008

&

Samrat Alam

Roll: 1607022

A thesis submitted in partial fulfillment of the requirements for the degree of

“Bachelor of Science in Computer Science and Engineering”

Supervisor:

Dr. Kazi Md. Rokibul Alam

Professor
Department of Computer Science and Engineering
Khulna University of Engineering & Technology ……………………..
Khulna, Bangladesh Signature

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

March 2022

Acknowledgment

With the blessings and limitless mercy of the Almighty, we are able to do this. We express our heartiest gratitude to Almighty Allah. Then we express our indebtedness to our honorable supervisor Prof. Dr. Kazi Md. Rokibul Alam for his helpful contribution, necessary guidance, suggestions, and encouragement to us. He inspired us to delve into the research works of several researchers related to the topic to have an idea of the efforts and works that were done. We also thank our honorable sir Prof. Dr. Muhammad Sheikh Sadi for his valuable contribution and assistance in deciding on the thesis idea.

We would also like to thank all the teachers and staff of the CSE department of KUET who helped us by providing guidelines to perform the work.

Abstract

Ternary Content Addressable Memory (TCAM) is a specialized network device that is used as a memory element and searches its entire elements throughout one clock cycle. As TCAM is used to implement Software Defined Networks (SDNs), it plays a vital role in highly reliable network applications. However, TCAM is becoming vulnerable due to soft error. These soft errors corrupt the stored bits in the memory that may cause a complete system failure. This research work proposes an efficient approach to tolerate soft errors in TCAM by the efficient application of the rules that are generated from the input data word. The major advantage of this approach is that it can correct 100% of errors with relatively low redundant bits. The proposed method outperforms one of the existing dominant approaches.

Contents

List of Figures

Figure No. **Description** **Page**

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

In-memory systems, soft errors are becoming a matter of concern [1]. Due to the collision of radiation particles with semiconductors, charged particles are generated [2]. These charged particles create data distortion contained in the memory cells and generate soft errors. When there requires high reliability in the system, soft errors are becoming a matter of great concern for computer systems [3], [4], [5]. A single bit of error can cause the whole system failure. As TCAM is used to design SDNs, it plays a vital role in highly reliable network applications such as a router. They are used for different important functions such as looking up the entire router, packet forwarding, and packet classification [6], [7] and for this reason, TCAM is one of the most significant elements of networking devices [8]. The prevention of soft errors in TCAM is a challenging task since it needs extra care to design routers or other network elements for tolerating soft errors [9].

The amount of network applications is increasing rapidly and these network applications are very sensitive to soft errors. So, the demand for safety against soft errors is also increasing [10]. To protect memory from these soft errors, Error Correction Codes (ECC) are normally used [11]. Bose-Chaudhuri-Hocquenghem (BCH), SEC–Double-adjacent Error Detection (SEC–DED), Matrix Codes, Hamming code etc. are some popular error recovery ECC methods. But in these ECC methods, extra bits per data word are needed. Moreover, ECC cannot check all words in parallel in TCAM easily. For the ECC, using a decoder per word is costly and leads to a huge memory and high-power consumption.

The proposed method shows how the erroneous bits in TCAM can be efficiently detected and corrected. The major contributions of this research work are: it can correct a hundred percent of soft errors for the given 24-bit data words (for all possible combinations), it requires relatively lesser redundant bits in comparison to existing dominant approaches, and it can detect and correct soft errors in both sequential and random data words in TCAM.

1.2 Objective of the Thesis

In this thesis, an efficient coding method has been proposed by which multi-bit errors can be detected and corrected easily. The main objectives of this thesis are as follows:

- To propose an efficient soft error detection and correction method for TCAM.
- To minimize information redundancy and other overheads without sacrificing error detection ability.
- To make an efficient technique that has a higher accuracy rate than other techniques for TCAM.

1.3 Scope of the Thesis

As it was mentioned earlier, the primary objective of this thesis is to develop an efficient technique to detect & correct multiple-bit soft errors in TCAM. The proposed method has higher error detection and correction rate. In fact, the proposed method increases the detection and correction rate. The vision was to research and develop systems that can detect and correct multi-bit errors of any pattern in a large data block.

1.4 Contribution of the Thesis

In computing systems, the error detection and correction approach is much needed. The traditional approach towards error detection and correction has some limitation that needs to be addressed. To overcome these problems, we proposed a new method. The contribution of this proposed method is as follows:

- Detect and correct multiple bit errors with 100% accuracy.
- Reducing overhead.

1.5 Thesis Organization

Chapter 2 introduces the formal structures and terminologies used in this thesis. Also discusses some methods of error detection and correction related work that have been done in this field by other researchers. A brief discussion about their limitations is also given in this chapter.

Chapter 3 describes our study and implementation of the proposed method for improving the error detection & correction coding approach.

Chapter 4 represents the experimental analysis of our proposed method. There is also an elaborate discussion about the experimental result in this chapter.

Chapter 5 draws the conclusion of our proposed method. It also states some future works that can be done for improving the system.

Chapter 2

Review of Soft Errors and Literature

2.1 Introduction

In this chapter, some formal statements and terminologies related to this thesis will be discussed. Some specifications will also be discussed which will be used to describe the proposed method. These statements and terminologies will be elaborated on using some examples and pictorial representation.

Error detection and correction schemes will also be discussed. Some problems with the existing method will also be explained. Many of the existing methods have some problems. Some methods are used efficiently and some are not. In this chapter, some existing methods are described.

2.2 Soft Errors

An error occurred in a computer’s memory system that changes an instruction in a program or a data value. Soft errors typically can be remedied by cold booting the computer. A soft error will not damage a system’s hardware; the only damage is to the data that is being processed. In electronics and computing, a soft error is a type of error where a signal or datum is wrong. Errors may be caused by a defect, usually understood either to be a mistake in design or construction or a broken component. A soft error is also a signal or datum which is wrong but is not assumed to imply such a mistake or breakage. After observing a soft error, there is no implication that the system is any less reliable than before. In the spacecraft industry, this kind of error is called a single-event upset. A soft error is any measurable or observable change in the state or performance of a microelectronic device, component, subsystem, or system (digital or analog) resulting from a single energetic particle strike. The particle includes but is not limited to alpha particles, neutrons, and cosmic rays.

There are two types of soft errors:

1. Chip-level Soft Error.
2. System-level Soft Error.

2.2.1 Chip-level Soft Error

These errors occur when the radioactive atoms in the chip's material decay and release alpha particles into the chip. Because an alpha particle contains a positive charge and kinetic energy, the particle can hit a memory cell and cause the cell to change its state to a different value. The atomic reaction is so tiny that it does not damage the actual structure of the chip. Chip-level errors are rare because modern memory is so stable that it would take a typical computer with a large memory capacity at least 10 years before the radioactive elements of the chip's materials begin to decay.

2.2.2 System-level Soft Error

These errors occur when the data being processed is hit with a noise phenomenon, typically when the data is on a data bus. The computer tries to interpret the noise as a data bit, which can cause errors in addressing or processing program code. The bad data bit can even be saved in memory and cause problems at a later time.

2.3 Soft Error: A Matter of Concern

Embedded systems face particular challenges from soft errors. An embedded system involves software and hardware that is designed to achieve a specific solution. The

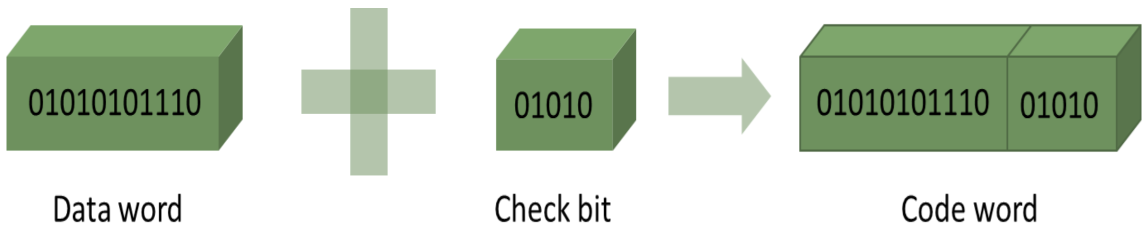


Figure 2.1: Coding Process

the system is expected to offer high reliability and meet real-time criteria. The very high level of complexity and the fact that the software and hardware are so intricately linked means that the system may be very sensitive to soft errors. Specifically, reliability is a matter of great concern when designing high availability systems or systems used in electronic hostile environments.

2.4 Coding Theory and Code Efficiency

Coding is an established area of research and practice, especially in the communication field. When coding, a d-bit data word is encoded into a c-bit codeword, which consists of a larger number of bits than the original data word, i.e. c > d. this coding introduces information redundancy. A consequence of this information redundancy is that not all 2c binary combinations of the c-bits are valid codeword. As a result, when attempting to decode the c-bit word to extract the original d data bits, we may encounter an invalid codeword and this will indicate that an error has occurred.

Coding theory is the study of the properties of codes and their fitness for a specific application. Coding theory is used for data compression, cryptography, error-correction and more recently also for network coding.

This typically involves the removal of redundancy and the correction (or detection) of errors in the transmitted data. There are essentially two aspects to coding theory:

1. Data compression (or, source coding)
2. Error correction (or channel coding)

Source encoding attempts to compress the data from a source in order to transmit it more efficiently. The second, channel encoding, adds extra data bits to make the transmission of data more robust to disturbances present on the transmission channel. The ordinary user may not be aware of many applications using channel coding. A typical music CD uses the Reed-Solomon code to correct for scratches and dust. In this application, the transmission channel is the CD itself. Cell phones also use coding techniques to correct for the fading and noise of high-frequency radio transmission. Data modems, telephone transmissions, and NASA all employ channel coding techniques to get the bits through, for example, the turbo code and LDPC codes.

We evaluate the efficiency of a code using the following three criteria [23]:

1. Number bit errors a code can detect/correct, reflecting the fault-tolerant capabilities of the code.
2. Code rate, reflecting the amount of information redundancy added.
3. The complexity of encoding and decoding schemes reflects the amount of hardware, software and time redundancy added.

$$\text{Bit Overhead} = \frac{\text{number of parity bits}}{\text{number of data bits}}$$

$$\text{Code Rate} = \frac{\text{number of data bits}}{\text{number of codeword}}$$

2.5 Error Detection

Error detection is the ability to detect the presence of errors caused by noise or other impairments during transmission from the transmitter to the receiver.

Regardless of the design of the transmission system, there will be errors, resulting in the change of one or more bits in a transmitted frame. When a codeword is transmitted one or more numbers of transmitted will be reversed due to transmission impairments. Thus error will be introduced. It is possible to detect these errors if the received codeword is not one of the valid codewords.

The concept of including extra information in the transmission of error detection is a good one. But instead of repeating the entire data stream, a shorter group of bits may be appended to the end of each unit. This technique is called redundancy because the extra bits are redundant to the information, they are discarded as soon as the accuracy of the transmission has been determined.

2.6 Error Detection Scheme

In telecommunication, a redundancy check is extra data added to a message for the purposes of error detection. Several schemes exist to achieve error detection and are generally quite simple. All error detection codes transmit more bits than were in the original data. Most codes are systematic; the transmitter sends a fixed number of original data bits, followed by a fixed number of check bits (usually referred to as redundancy in the literature) which are derived from the data bits by some deterministic algorithm. The receiver applies the same algorithm to the received data bits and compares its output to the received check bits; if the values do not match, an error has occurred at some point during the transmission.

2.7 Error Correction

Error correction is the detection of errors and reconstruction of the original, error-free data. So that even if some of the original data is corrupted during transmission, the receiver can still recover the original message intact.

Various method for error correction is as follows

- Parity
- Hamming
- BCH code
- Golay code
- DMC code
- MC codes

2.7.1 Parity Schemes

The small of the transistors or capacitors, combined with cosmic ray effects, cause occasional errors in stored information in large, dense RAM chips, particularly those that are dynamic. These errors can be detected and corrected by employing error-detecting and error-correcting codes in RAM. One of the most common error detection & correction code is the parity scheme.

The most common application of parity is error detection in memories of computer systems [20]. A diagram of a memory protected by a parity code is shown in Figure 2.2.

Before being written into a memory, the data is encoded by computing its parity. The generation of parity bits is done by a parity generator (PG). When data is written into memory, parity bits are written along with the corresponding bytes of data.

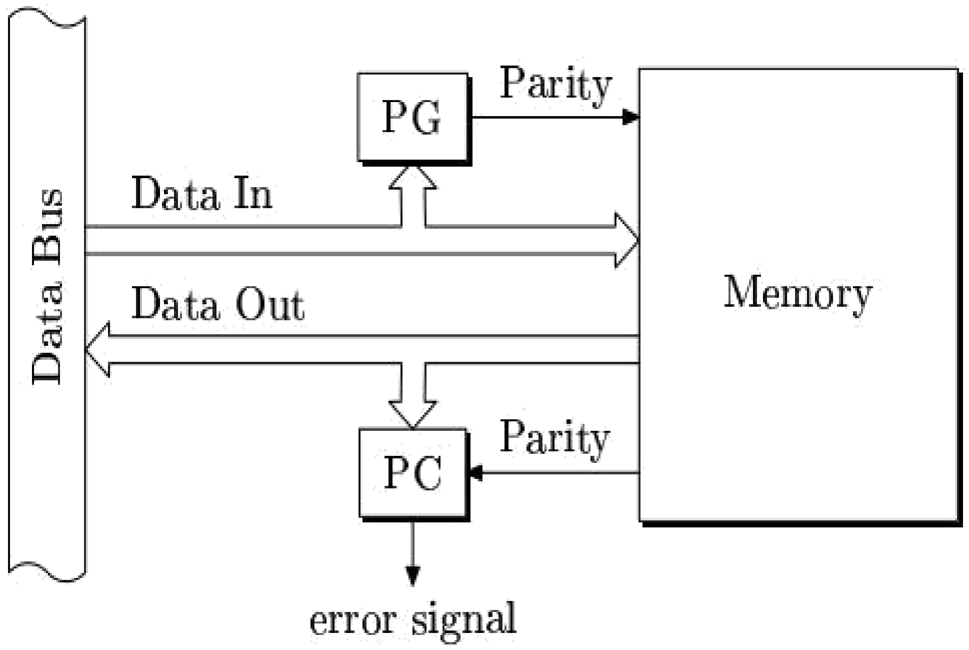


Figure 2.2: A Memory Protected by a Parity Code

When the data is read back from the memory, parity bits are re-computed and the result is compared to the previously stored parity bits. Re-computation of parity is done by a parity checker (PC).

2.7.2 Golay Code

A binary Golay code [21] is a type of error-correcting code used in digital communications. The binary Golay code, along with the ternary Golay code, has a particularly deep and interesting connection to the theory of finite sporadic groups in mathematics. These codes are named in honor of Marcel J. E. Golay.

There are two closely related binary Golay codes. They are as follows:

- Extended binary Golay code (23,12,7)
- Perfect binary Golay code (23,12,7)

The extended binary Golay code encodes 12 bits of data in a 24-bit word in such a way that any 3-bit errors can be corrected or any 7-bit errors can be detected.

The other, the perfect binary Golay code, has codewords of length 23 and is obtained from the extended binary Golay code by deleting one coordinate position. Conversely, the extended binary Golay code is obtained from the perfect binary Golay code by adding a parity bit.

In standard code notation the codes have parameters [24, 12, 8] and [23, 12, 7], corresponding to the length of the codewords, the dimension of the code, and the minimum Hamming distance between two codewords, respectively.

2.7.3 Bose Chaudhuri Hocquenghem Codes

In coding theory, the BCH [24] codes form a class of cyclic error-correcting codes that are constructed using finite fields. One of the key features of BCH codes is that during code design, there is precise control over the number of symbol errors correctable by the code.

Given a prime power q and positive integers m and d with d q^m 1, a primitive narrow-sense BCH code over the finite field GF(q) with code length n = q^m 1 and distance at least d is constructed by the following method.

Let be a primitive element of GF(q^m). For any positive integer i, let $m_i(x)$ be the minimal polynomial of $_i$. The generator polynomial of the BCH code is defined as the least common multiple g(x) = lcm($m_1(x)$; $m_{d\ 1}(x)$). It can be seen that g(x) is a polynomial with coefficients in GF(q) and divides x^n 1. Therefore, the polynomial code defined by g(x) is a cyclic code.

Let q=2 and m=4 (therefore n=15). We will consider different values of d. There is a primitive root in GF (16) satisfying:

$$\alpha4+\alpha+1=0$$

it’s minimal polynomial over GF (2) is:

$$m_1(x) = x^4 + x + 1$$

The minimal polynomials of the rest seven powers of are:

$$m_1(x) = m_2(x) = m_4(x) = x^4 + x + 1;$$

$$m_3(x) = m_6(x) = x^4 + x^3 + x^2 + x + 1;$$

$$m_5(x) = x^2 + x + 1;$$

$$m_7(x) = x^4 + x^3 + 1:$$

The BCH code with d = 2,3 has generator polynomial

$$g(x) = m_1(x) = x^4 + x + 1:$$

It has minimal Hamming distance at least 3 and corrects up to one error. Since the generator polynomial is of degree 4, this code has 11 data bits and 4 checksum bits. The BCH code with d = 4,5 has generator polynomial

$$g(x) = \mathrm{lcm}(m_1(x); m_3(x) = (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) = x^8 + x^7 + x^6 + x^4 + 1).$$

It has a minimal Hamming distance of at least 5 and corrects up to two errors. Since the generator polynomial is of degree 8, this code has 7 data bits and 8 checksum bits. The BCH code with d=8 and higher has a generator polynomial:

$$g(x) = \mathrm{lcm}(m_1(x); m_3(x); m_5(x); m_7(x)$$

$$(x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)(x^2 + x + 1)(x^4 + x^3 + 1)$$

$$x^{14} + x^{13} + x^{12} + ::: + x^2 + x + 1$$

This code has a minimal Hamming distance of 15 and corrects 7 errors. It has 1 data bit and 14 checksum bits. In fact, this code has only two codewords: 000000000000000 and 111111111111111.

BCH (31,16,7) implies that there are 15 data bits, 16 check bits in the codeword and hamming distance of 7. BCH code can correct up to the 3-bit error.

2.7.4 Matrix Code

In MC Coding technique [25], the n-bit codeword is divided into k_1 sub words of width k_2 (i.e., $n = k_1 \times k_2$). A $(k_1; k_2)$ matrix is formed where k_1 and k_2 represent the numbers of rows and columns, respectively. For each of the k_1 rows, the check bits are added for single error correction/double error detection. Another k_2 bits are added as vertical parity bits.

2.7.5 Decimal Matrix Code

The Decimal Matrix code [26] technique assures reliability in the presence of MCUs with reduced performance overheads. First, during the encoding (write) process, in-formation bits D are fed to the DMC encoder, and then the horizontal redundant bits H and vertical redundant bits V are obtained from the DMC encoder. When the encoding process is completed, the obtained DMC codeword is stored in the memory. If MCUs occur in the memory, these errors can be corrected in the decoding (read) process.

Besides, there are many methods to correct errors during data transmission. In this chapter, we only discussed about parity. How parity helps to detect and correct errors in the data bits.

2.7.6 Hamming Code

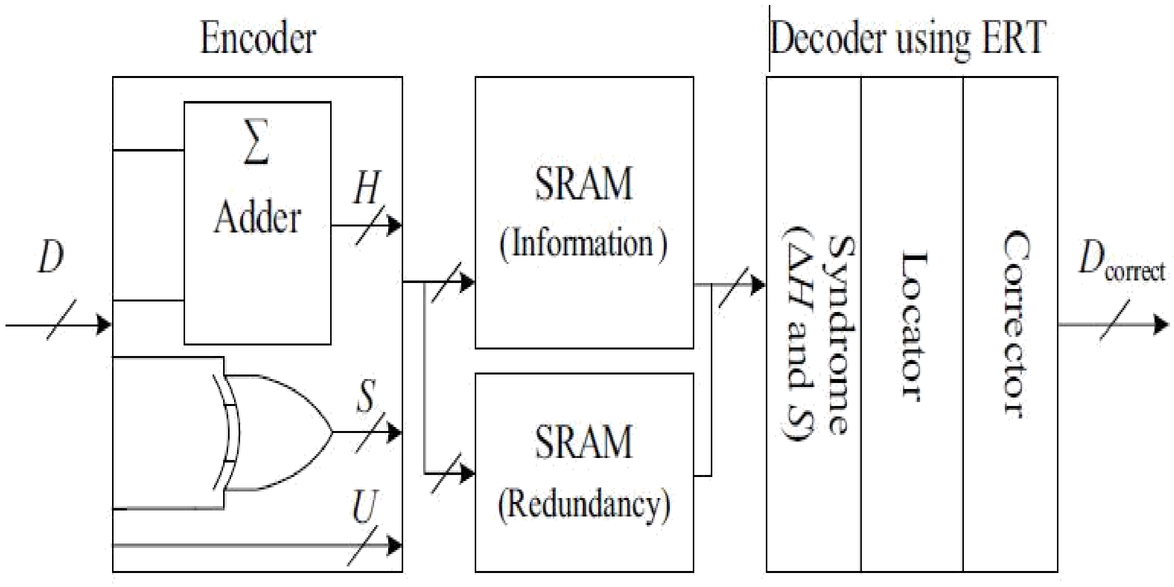


Figure 2.3: Decimal Matrix Code

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver. It is a technique developed by R.W. Hamming for error correction [19].

General Algorithm of Hamming code:

1. The Hamming Code is simply the use of extra parity bits to allow the identification of an error.
2. Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).
3. All the bit positions that are a power of 2 are marked as parity bits (1, 2, 4, 8, etc).
4. All the other bit positions are marked as data bits.
5. Each data bit is included in a unique set of parity bits, as determined by its bit position in binary form.
    a. Parity bit 1 covers all the bits positions whose binary representation includes a 1 in the least significant
    b. Position (1, 3, 5, 7, 9, 11, etc).
    c. Parity bit 2 covers all the bits positions whose binary representation includes a 1 in the second position from
    d. The least significant bit (2, 3, 6, 7, 10, 11, etc).
    e. Parity bit 4 covers all the bits positions whose binary representation includes a 1 in the third position from
    f. The least significant bit (4–7, 12–15, 20–23, etc).
    g. d. Parity bit 8 covers all the bits positions whose binary representation includes a 1 in the fourth position from
    h. The least significant bit bits (8–15, 24–31, 40–47, etc).
    i. e. In general, each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.
6. Since we check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd.
7. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

If a byte of data to be encoded is 10011010, then the data word (using _ to represent the parity bits) would be __1_001_1010, and the codeword is 011100101010.

Table 2.1: Hamming codes

Parity bits	**Total bits**	**Data bits**	**Name**
2	3	1	Hamming(3,1)
3	7	4	Hamming(7,4)
4	15	11	Hamming(15,11)
5	31	26	Hamming(31,26)
6	63	57	Hamming(63,57)
7	127	120	Hamming(127,120)
8	255	247	Hamming(255,247)
m	n = 2m-2	K = 2m-m-1	Hamming(2m-1,2m-m-1)

2.7.7 Single-Error Correction Codes

A parity check matrix consists of r rows and n columns (n = k + r), where k is the number of data bits and r is the number of check bits. Each column in the parity check matrix corresponds to either a data bit or a code bit.

C1	C2	D1	C3	D2	D3	D4
0	0	0	1	1	1	1
0	1	1	0	0	1	1
1	0	1	0	1	0	1

Figure 2.4: Parity check matrices for the codeword: *D1D2D3D4* with corresponding code bits *C1C2C3* where the code bit columns are in the power of two positions.

The positions of the 1s in a row in the parity check matrix indicate the bit positions that are involved in the parity check equation for that row. For example, the parity check matrix in Figure 5.6a defines the following parity check equations:

$$C3 = D2 \oplus D3 \oplus D4$$

$$C2 = D1 \oplus D3 \oplus D4$$

$$C1 = D1 \oplus D2 \oplus D4$$

where $\oplus$ denotes the bit-wise XOR operation. If *D1D2D3D4* = 1010, then C3=1, C2 =0, C1=1. The corresponding codeword *C1C2C3D1D2D3D4* is 1011010. If E represents the codeword vector, then in matrix notation, E = [1011010].

The parity check matrix is created carefully to allow the generation of the syndrome, which identifies the bit position in error in a given codeword. If P is the parity check matrix and in E is the codeword vector, then the syndrome is expressed as

$$S = P \bullet ET$$

where E^T is the transpose of E. For example, if P is the parity check matrix in Figure 5.6a and E = [1011010], then one can derive S as:

$$S = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If S is expressed as [S3 S2 S1], then S3 = (0 AND 1) $\oplus$ (0 AND 0) $\oplus$ (1 AND 1) $\oplus$ (0 AND 1) $\oplus$ (1 AND 0) $\oplus$ (1 AND 1) $\oplus$ (1 AND 0) = 0, where the dot represents the bit-wise AND operation and $\oplus$ represents the bit-wise XOR operation. S_2 and S_1 can be computed in a similar fashion. The syndrome in this example is zero, indicating that there is no single-bit error in the codeword E.

2.7.8 Single-Error Correct Double-Error Detect Code

An SEC code can detect and correct single-bit errors but cannot detect double-bit errors. As suggested by Figure 5.2, a SECDED code's minimum Hamming distance must be 4, whereas this SEC code's minimum Hamming distance is 3 and hence it cannot simultaneously correct single-bit errors and detect double bit errors.

An SEC code's parity check matrix can be extended easily to create an SEC-DED code that can both correct single-bit errors and detect double-bit errors. Today, such SEC-DED codes are widely used in computing systems to protect memory cells, such as microprocessor caches, and even to register files in certain architectures. To create an SEC-DED code from a SEC code, one can add an extra bit that represents the parity over the seven bits of the SEC codeword. The syndrome becomes a 4-bit entity, instead of a 3-bit entity, for SEC codes. Then, one can distinguish between the following cases:

1. If the syndrome is zero, then there is no error.
2. If the syndrome is non-zero and the extra parity bit is incorrect, then there is a single-bit correctable error.
3. If the syndrome is nonzero, but the extra parity bit is correct, then there is an uncorrectable double-bit error.

The parity check matrix of this SECDED code can be represented as:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The additional first row now represents the extra parity bit. The additional last column places the extra parity bit in the power-of-two position to ensure it is a Hamming code. Hence, the codeword with the double-bit error would look like [1 0 0 1 0 1 1 0], where the last 0 represents the extra parity check bit. Then, $H \times E^T$ gives us:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Here the extra bit (last bit in codeword) is correct, but the syndrome 0100 is non zero. Therefore, this SECDED code can correctly identify a double-bit error. However, if the codeword is [1 0 0 1 0 1 0 0], then the extra bit is zero and incorrect, which suggests a single-bit error. The resulting syndrome is 1011, the lower three bits of which indicate the bit position in error.

2.8 Ternary Content Addressable Memory (TCAM)

Ternary content-addressable memory (TCAM) is a form of content-addressable memory (CAM) that adds flexibility to the search by allowing a third state of "don't care" or "X" in one or more of the bits of stored data. The term "ternary" refers to the number of inputs the memory may store and query: 0, 1, and X. This TCAM is a type of high-speed memory that searches the complete contents of the memory in a single clock cycle. It can store and query data using three different inputs: 0, 1 and X.

2.8.1 Working Procedure of TCAM

TCAM can store and query data using three different inputs: 0, 1 and X [17]. For example, TCAM may have “10XX0” as one of its stored words. The “don’t care” state allows the TCAM to flexibly match any one of four search words: "10000," "10010," "10100," or "10110." A priority encoder in the TCAM ensures that only the first matching entry is output. The flexible coding with X reduces the number of stored entries, thus further improving the efficiency of the search. It is most useful for building tables for searching for longest matches such as IP routing tables organized by IP prefixes. It can search all contents in parallel, which makes it much faster [18]. In Figure 2.5, the parallel search procedure in TCAM is shown.

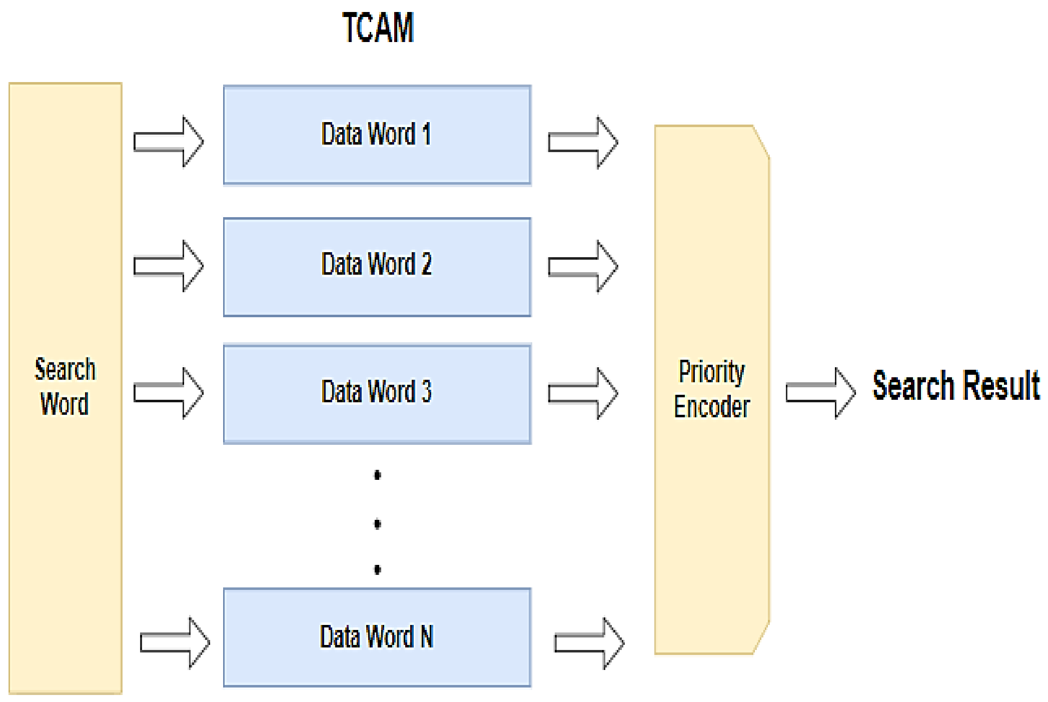


Figure 2.5: Parallel Search in TCAM

2.8.2 Example of Address Lookup with TCAM

Here is an example of address lookup with TCAM. First, search data is given which is the input of the TCAM table. It matches the pattern with the table's corresponding value. For the X value, it can match either 0 or 1. So, it mainly matches all other positions of the search data. From Figure 2.6, we see that 01110 is the search data and "0111" data matches the 2nd Row of the TCAM table. The last position of the search data is 0 and in the 2nd row of the table is X. That is mean that any value of the search data can match with this table's X value. So, the search data is matched with the table's 2nd row. After matching the data with the table's row, it searches the address of the corresponding row in the memory. And it gives the corresponding result which is in the memory. We get the search data in the 2nd row and the corresponding port 2 is stored in the memory. And this port 2 is the output of the search data.

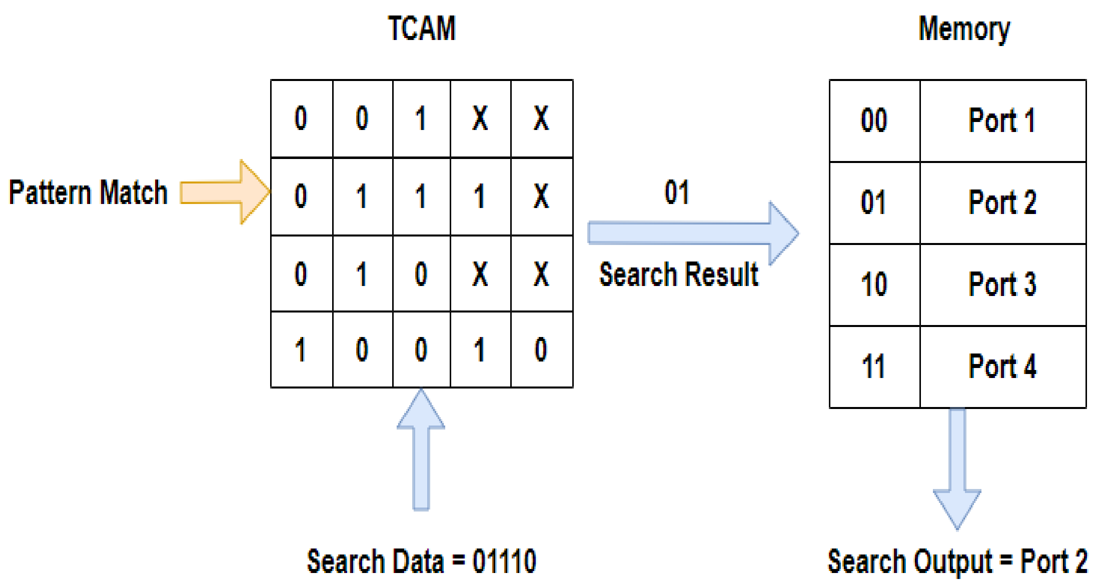


Figure 2.6: Address lookup with TCAM.

2.9 Related Work

There are few existing approaches that are proposed to tolerate soft error. These are outlined shortly as follows.

Reviriego et al. [1] proposed a methodology where a technique was developed for correcting single-bit errors in the data words. In this method, the total number of '1' in every column must be equal to 2^n, where n is an integer number. Without this sequence, the weight is defined as an illegal weight. There must be only one column with weights of 0 and 2 and may have many columns with weights 1 in a block. However, the method could not correct more than a one-bit error in a data word. In TCAM, there are some search keys and keys are divided into small blocks of b bits. Even for a single bit error, if the number of ‘b’ is not large, it cannot correct 100% errors. For example, when the number of ‘b’ is equal to 9, it can correct nearly 99.6% of single-bit errors.

Syafalni et al. [12] proposed a soft error tolerant TCAM methodology where they used some selected search keys to detect soft errors. However, their approach is able to detect merely single-bit errors, and this method is relatively costly because it used a backpack SRAM and an extra TCAM.

Onawa et al. [13] proposed a hardware-based methodology where the authors used electronic transistors and magnetic-tunnel-junction (MTJ). In this methodology, a total of twenty transistors and four MTJ devices were used to find a single bit error. This methodology is very expensive than most other alternative approaches.

Pontarelli et al. [14] proposed a filter-based technique. This method used a bloom filter for detecting and correcting single-bit soft errors. However, there is a false positive problem. If there is no information in the Bloom Filter but detects the existence of data, then, this is termed a false positive.

Inayat et al. [15] proposed a soft-error-resilient method, in which, the ER-TCAM stores a parity bit for each SRAM word to detect single-bit soft errors. This method used extra SRAMs for fast searching. When an error is detected, the ER-TCAM operates on the redundant information from the binary-coded TCAM table for correction. Though this method searched fast and has a high correction rate, it is relatively costly because of using Extra SRAM, redundant information and parity bits.

2.10 Summary

In this chapter, we have discussed the existing method. There are many methods existing to detect and correct errors. Here in this chapter some of them are mentioned and discussed. The parity coding scheme has been discussed with proper examples and the limitations of this method are finally mentioned.

Chapter 3

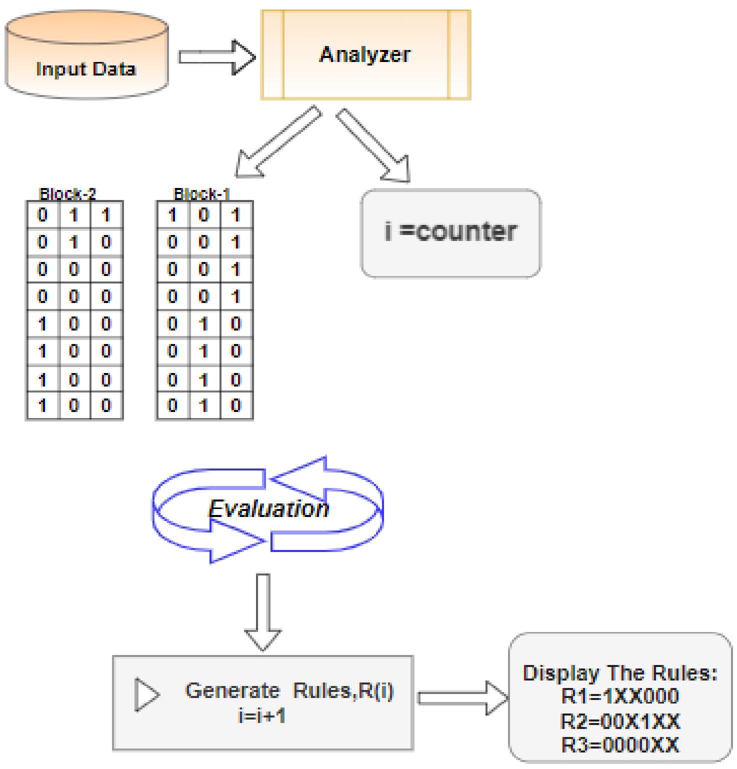
Proposed Error Detection and Correction Methodology

3.1 Introduction

In this chapter, some formal statements and terminologies related to this thesis will be discussed. Some specifications will also be discussed which will be used to describe the proposed method. These statements and terminologies will be elaborated on using some examples and pictorial representation. In this chapter error detection and correction scheme will also be discussed. This section is divided into 3 sub-sections: Generating Rules from Input Data, Fault Injection, and Error Detection and Correction. These are discussed as follows.

3.2 Generating Rules From Input Data

First, input data is taken from the sender. The sender passes the total data word size ‘n’, number of blocks ‘k’, number of data words ‘m’, and the corresponding Data.



Block-2

0	1	1
0	1	0
0	0	0
0	0	0
1	0	0
1	0	0
1	0	0
1	0	0

Block-1

1	0	1
0	0	1
0	0	1
0	0	1
0	1	0
0	1	0
0	1	0
0	1	0

Display The Rules:
R1=1XX000
R2=00X1XX
R3=0000XX

Figure 3.1: The flow diagram of generating rules from the input data.

The m*n bit message is divided into k blocks (here, m*n must be divisible by k). Then, each block has m rows and p columns. Now, p rules (r^1, r^2, . . . , r^p) can be generated because the number of columns in every block is p. Now, each of these rules has k*ceil($\log_2$(m)) bits (here, ceil($\log_2$(m)) bits are required for addressing each bit position of m rows in every column). In each rule, 1st ceil(log2(m)) bits are required for 1st block, 2nd ceil($\log_2$(m)) bits are required for 2nd block, . . ., and similarly k-th ceil($\log_2$(m)) bits are required for k-th block. In this way, p rules can be generated. For example, from the positions of 1's in the 1st column of 1st block, r1 rule's first part (in here, two blocks will form two parts of a rule and each part will contain ceil($\log_2$(m)) bits) can be generated. In Figure 3.1, the process of generating rules is illustrated using a flow diagram.

3.3 Fault Injection

To verify our methodology, all combinations of faults are generated for a given data word at first. Then, these faults are injected into that data word to evaluate the performance of our method. The fault injection procedure is detailed in Algorithm 1.

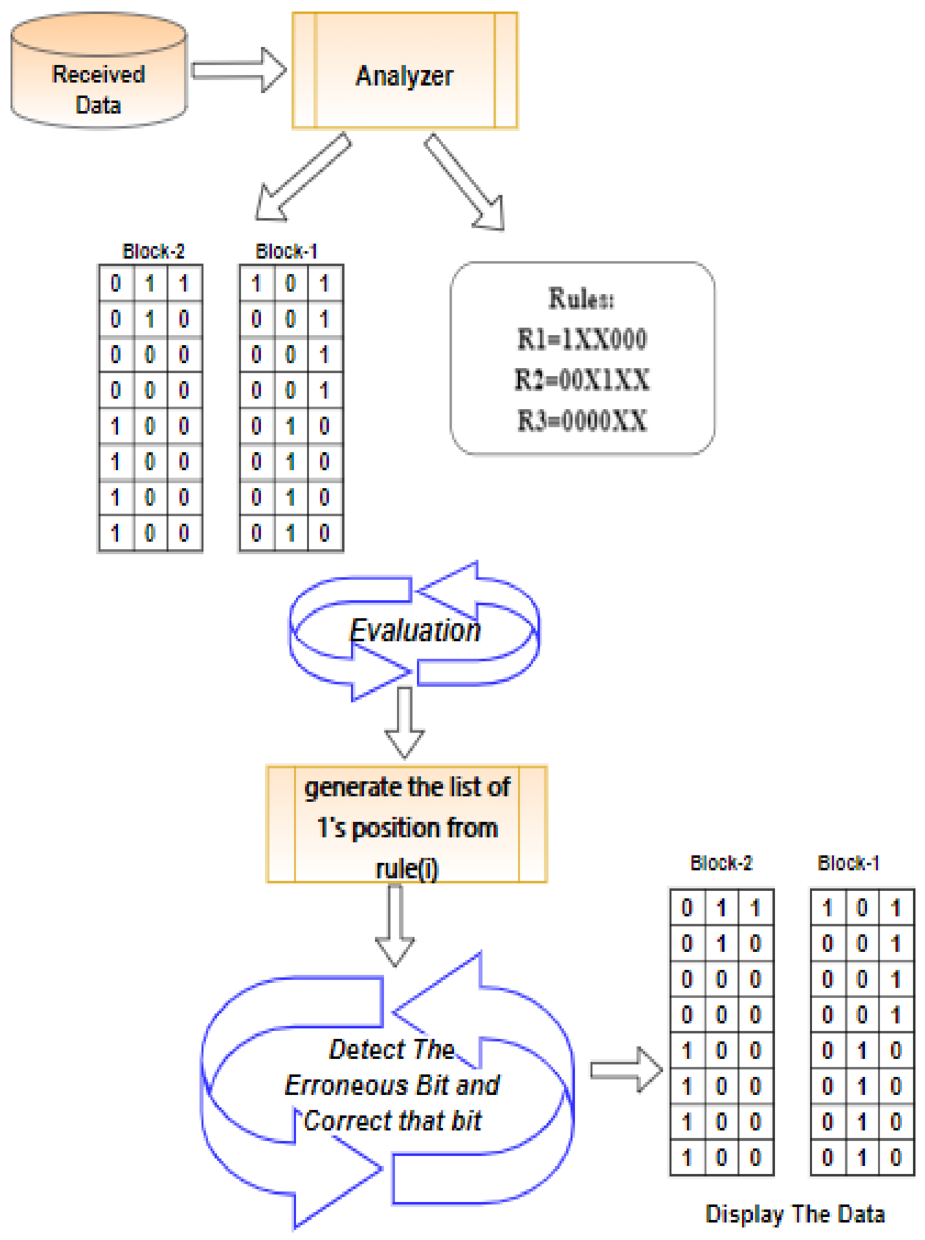
Algorithm 1: Error injection in TCAM

Here, m = number of data words, n = word size and x = number of bit positions where faults are injected.

```
Step 1: Declare b, r and c
Step 2: Input x
Step 3: Initialize b to 0
Step 4: Repeat step 3 until b <= x
Step 5: Input r and c
Step 6: If MSG[r][c] is equal to 0
Step 7:     Flip MSG[r][c] to 1
Step 8: Else
Step 9:     Flip MSG[r][c]  to 0
Step 10:      End If
Step 11:      Return
```

3.4 Error Detection and Correction

In the receiver's part, the receiver decodes p rules and generates the list of 1's positions for each of the p columns in each of the k blocks. Then, the receiver receives the message of size m*n size where m is the number of rows and n is the number of columns and divides the message into k blocks. Then, the receiver checks every bit position in each block, whether it contains 1 or 0. When checking, if the current bit position is not in the list of 1's positions for that column and block, but it contains value 1 then the receiver's program detects it as an error and corrects it by flipping the bit. On the other hand, if the current bit position is in the list of 1's positions for that column and block, but the bit position contains value 0 then the receiver's program detects it as an error and flips the bit to correct it. In Figure 3.2, the process of error detection and correction is illustrated using a flow diagram.



Received Data
Analyzer
Rules:
R1=1XX000
R2=00X1XX
R3=0000XX
Evaluation
generate the list of 1's position from rule(i)
Detect The Erroneous Bit and Correct that bit
Display The Data

Block-2

0	1	1
0	1	0
0	0	0
0	0	0
1	0	0
1	0	0
1	0	0
1	0	0

Block-1

1	0	1
0	0	1
0	0	1
0	0	1
0	1	0
0	1	0
0	1	0
0	1	0

Block-2

0	1	1
0	1	0
0	0	0
0	0	0
1	0	0
1	0	0
1	0	0
1	0	0

Block-1

1	0	1
0	0	1
0	0	1
0	0	1
0	1	0
0	1	0
0	1	0
0	1	0

Figure 3.2: The flow diagram of error detection and correction

The algorithm of this methodology is shown below in Algorithm 2.

Algorithm 2: Proposed Algorithm for Error Detection and Correction

```
Step 1: Input Data (sender)
Step 2: Divide the data into block(s)
Step 3: Generate Rules from the blocks of the data
Step 4: Receive Data (receiver)
Step 5: Generate the list of 1's positions for each rule
Step 6: Check and compare every bit position with the lists generated in step 5
Step 7: If there is an error in bit(s)
Step 8:     Detect the erroneous bit and correct it
Step 9: End If
Step 10:    Show the data
```

3.5 An Example

In this section, the detailed procedure of detecting and correcting errors using the proposed method is illustrated with an example and is shown as follows.

Let us consider,

n (data word size) = 6

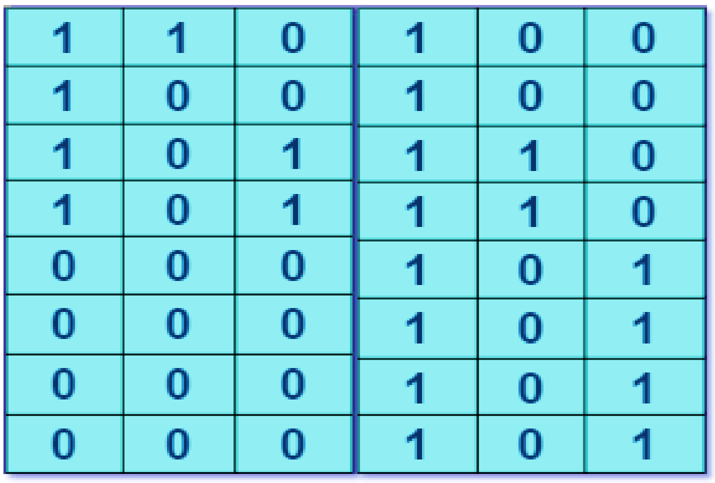
m (number of data words) = 8

k (number of blocks) =2

p (number of rules) = 3

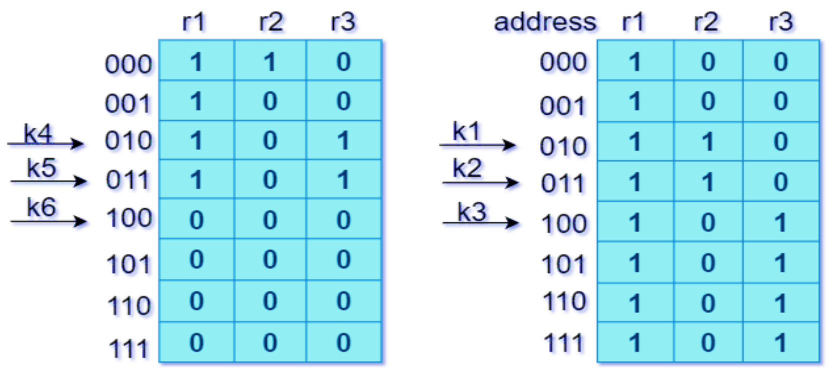
3.5.1 Generating Rules Using the Given Example

In Figure 3.3, a 6-bit message of size 8*6 is shown, and it is divided into 2 blocks in Figure 3.4.



1	1	0	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	0	1	1	1	0
0	0	0	1	0	1
0	0	0	1	0	1
0	0	0	1	0	1
0	0	0	1	0	1

Figure 3.3. Example of an (8X6) input message.



		r1	r2	r3
	000	1	1	0
	001	1	0	0
k4 →	010	1	0	1
k5 →	011	1	0	1
k6 →	100	0	0	0
	101	0	0	0
	110	0	0	0
	111	0	0	0

	address	r1	r2	r3
	000	1	0	0
	001	1	0	0
k1 →	010	1	1	0
k2 →	011	1	1	0
k3 →	100	1	0	1
	101	1	0	1
	110	1	0	1
	111	1	0	1

Block-2 Block-1

Figure 3.4: Generating rules from (8X6) input message.

Here, in the 1st column of the 1st block, 1's positions can be listed as 000, 001, 010, 011, 100, 101, 110, and 111. From the list of 1's positions, the first 3 bits (XXX) of rule r1 are generated. In the 1st column of the 2nd block, 1's positions are listed as 000, 001, 010, and 011. From these positions, the second 3 bits (0XX) of rule r1 are generated. The rule r1 is 0XXXXX. In the 2nd column of the 1st block, 1's positions are 010 and 011. From these positions, the first 3 bits (01X) of rule r2 are generated. In the 2nd column of the 2nd block, 1 is found only at position 000. Using this list, the second 3 bits (000) of rule r2 is generated. The rule r2 is 00001X. From this procedure, r3 can also be generated from Figure 3.4. Here, the first 3 bits of rule r3 are 1XX and the second 3 bits are 01X. The rule r3 is 01X1XX. Hence, all the rules can be generated from each block of the message.

Table 3.1: Generating rules from (8X6) input message.

Rules	Bit-1's position in 2nd block	Bit-1's position in 1st block	Rules for 2nd block	Rules for 1st block
r1	000,001,010,011	000,001,010,011, 100,100,110,111	0XX	XXX
r2	000	010,011	000	01X
r3	010,011	100,101,110,111	01X	1XX

3.5.2 Fault Injection in the Given Example

In Figure 3.5, let us assume that the contents of the bit positions 001 in the 1st column and 110 in the 2nd column of the 1st block are flipped to 0 and 1 respectively by the error injection program automatically.

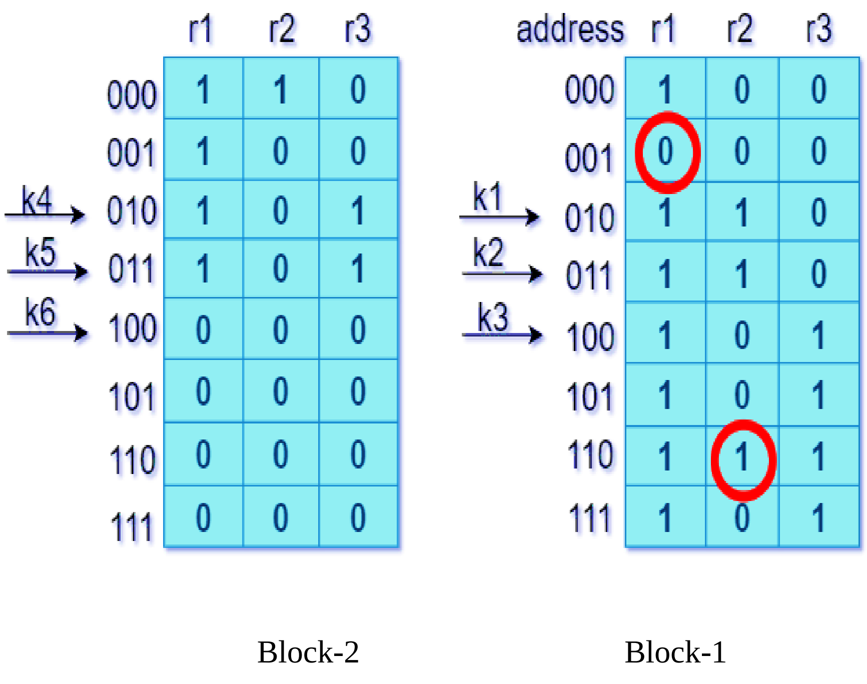
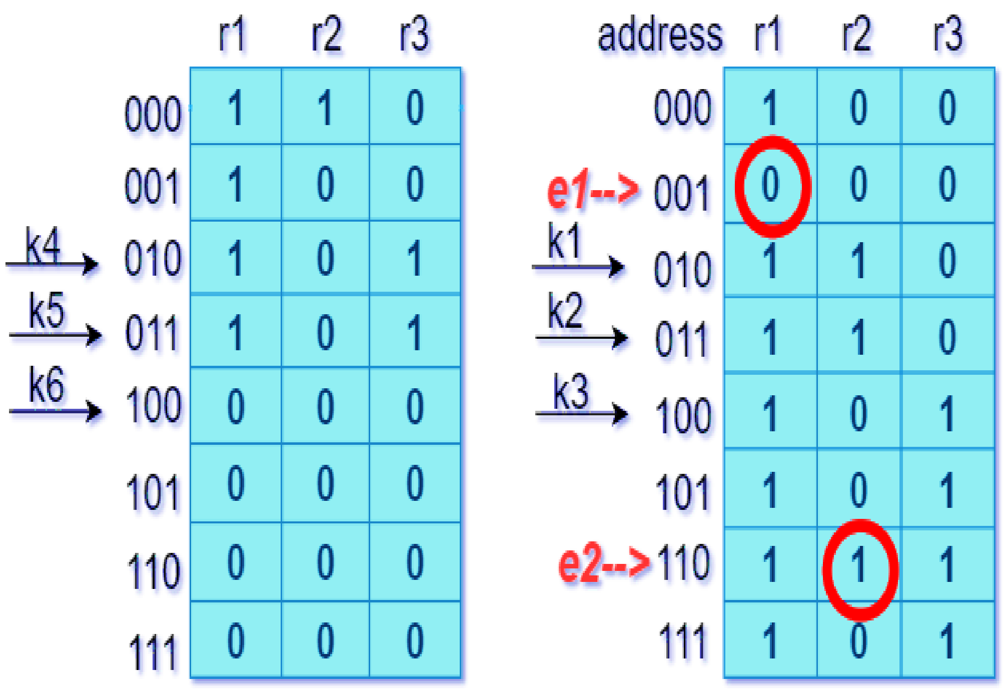


Figure 3.5: Injecting errors in bit position 001 in the 1st column and 110 in the 2nd column in the first block of the message.

3.5.3 Error Detection and Correction for the Given Example

In Figure 3.5, the bit at position 001 in the 1st column of the 1st block was flipped to 0 (as shown in Error Injection Part). From rule r1, 1st 3 positions (XXX) are for the 1st column of the 1st block. Now the list of the 1's positions for 1st column of 1st block (000, 001, . . ., 111) is generated. When the receiver reads the bit's position 001 and found that this position does not contain the value 1 then the proposed method detects it as an error which is shown in Figure 3.6 (highlighted as e1). Then, it will check whether the bit of the position 001 is 0 or 1. If it is 0 then the proposed method corrects this erroneous bit by flipping it.

Another example: if the bit at position 110 of the 2nd column in the 1st block is flipped from 0 to 1 then an error occurs. The list of 1's position of rule r2 for the 2nd column of the 1st block (010 and 011) is generated. When the receiver reads this bit position 110 and derives that this bit cannot be 1 because position 110 is not in the list of 1's positions (which was generated from the rule r2) then the receiver detects it as an error which is shown in Figure 3.6 (highlighted as e2). In the same way, the receiver corrects all errors in any position of all blocks.



	r1	r2	r3
000	1	1	0
001	1	0	0
010	1	0	1
011	1	0	1
100	0	0	0
101	0	0	0
110	0	0	0
111	0	0	0

address	r1	r2	r3
000	1	0	0
001	0	0	0
010	1	1	0
011	1	1	0
100	1	0	1
101	1	0	1
110	1	1	1
111	1	0	1

Block-2 Block-1

Figure 3.6: Detecting and correcting errors using rules.

Table 3.2: Detecting and correcting errors by generating 1's positions from the rules

Columns	Bit-1's position for 1st block	Detecting and correcting bit position
1	000,001,010,011,100,101, 110	001
2	010,011	110
3	100,101,110,111	No error!!!

3.6 Implementation of the Proposed Method with PSpice

3.6.1 Explanation of the Proposed Method

In this section, the implementation of the proposed method is shown below using an example. Let us consider,

n (data word size) = 4

m (number of data words) = 4

k (number of blocks) =2

p (number of rules) = 2

Generated two rules are R1 = XX00 and R2 = 1X01. Here, for R1, the first two bits (00) and last two bits(XX) from the LSB are for the 1st column of the 1st block and the 1st column of the 2nd block of the message respectively. Similarly, for R2, the first two bits (01) and last two bits(1X) from the LSB are for the 2nd column of the 1st block and 2nd column of the 2nd block of the message respectively.

In this example, Here, error detection and correction method detects these erroneous bit positions and corrects these bit flipped errors. Here, in block 2, the bit in position 10 in column 1 is flipped from 1 to 0. Bits in position 01 in column 2 are also flipped from 1 to 0. This error message is shown in Figure 3.7.

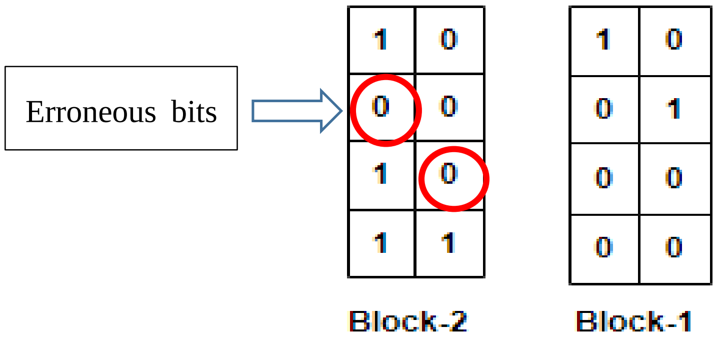


Figure 3.7. Erroneous received message

Now on the receiver side, the error detection and correction mechanism detect and corrects errors by using the generated rules. The bit position for bit 1s and 0s are calculated from the rules. Here, bit positions for 1's for column 1 and column 2 are 10, 11 and 00, 01, 10, 11 for block 2 by using R2. Now, the error detection mechanism detects an error in positions 10 of column 1 and 01 of column 2 in block 2. Then it checks the bits in position 10 of column 1 and 01 of column 2 whether these are 0 or 1. It then checks from 1's positions from R2. From R2, the bit in position 10 in column 1 in block 2 must be 1. It then corrects it from 0 to 1. Similarly, the bit in position 01 in column 2 in block 2 is corrected from 0 to 1. This process is shown in Figure 3.8.

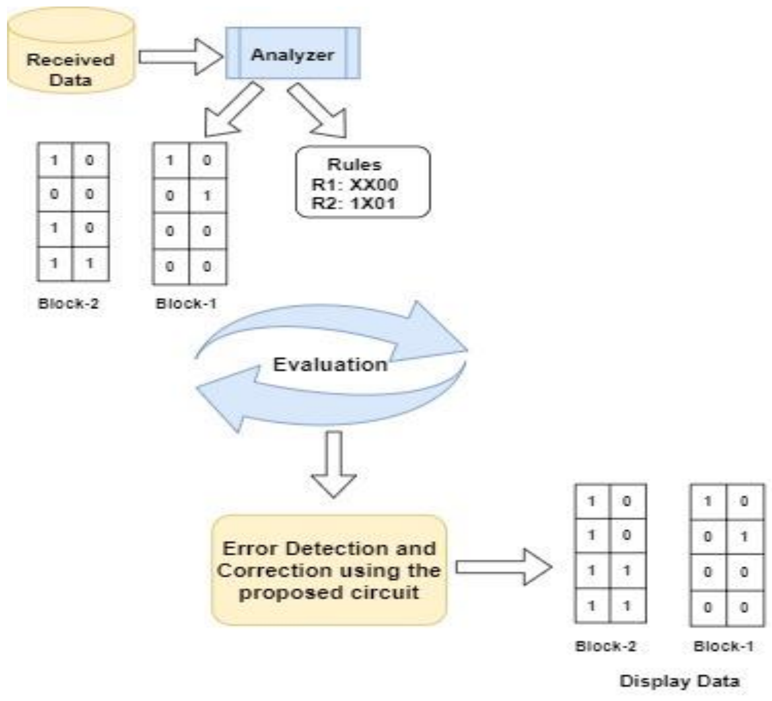


Figure 3.8. Flow diagram of Error Detection and Correction.

Here, a 4*4 memory is used to illustrate the detection and correction method. It detects and corrects errors column-wise. For example, when it selects the 1st row of the 4*4 memory cell, the selected address for every element in the first row is 00. Similarly, when it selects the 2nd row, the selected address for every element in the 2nd row is 01. The selected address for every element is then checked with specified parts of specified rules.

When the counter selects the 1st row of the 4*4 memory, the selected address is checked with TCAM cell and true/zero circuits. The selected address is 00. The selected address is then matched with the rules for the 1st column of the 1st block. Here, the selected address (00) is checked with the 1st part of the 1st rule (00). It is matched. So the 1st element of the 1st block in that row is set to 1. Else, it is set to 0. Similarly, when the selected address is (00), it is checked with the 1st part of the 2nd rule (01). It is not matched. So, the 2nd element of the 1st block in that row is set to 1. The circuit diagram is displayed in figure 3.9.

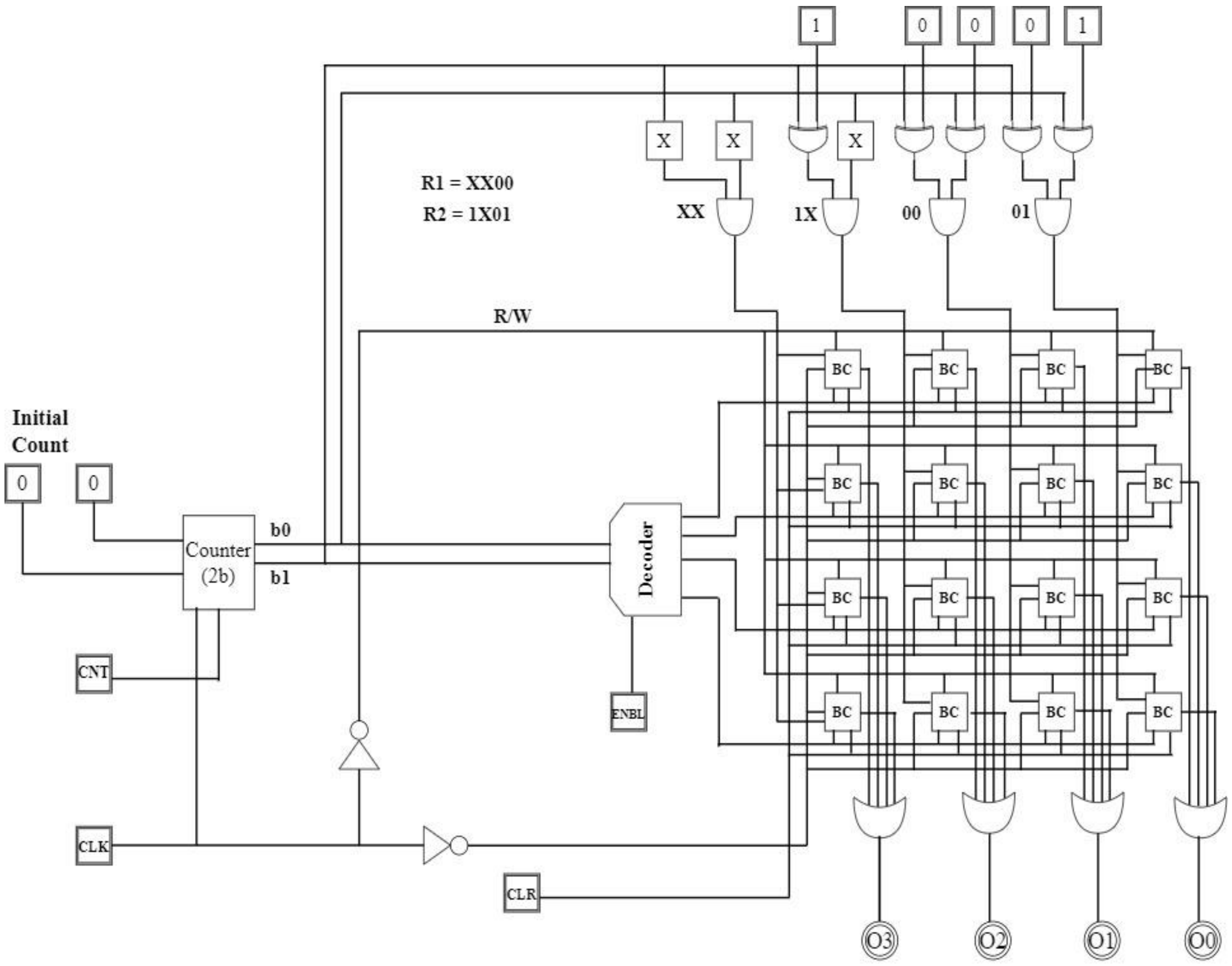


Figure 3.9. Circuit diagram of the proposed method for 16 bits message.

The pseudocode for the proposed method is given below.

Algorithm 1: Proposed Algorithm for Error Detection and Correction

```
Step 1: Receive Data (receiver)
Step 2: Receive rules
Step 3: Check and compare every position with the corresponding portion of the rules
Step 4: If there is an error in bit(s)
Step 5:     Detect the erroneous bit and correct it
Step 6: End If
Step 7: Pass the output through the memristor circuit
Step 8: Show the data
```

3.7 Summary

In this chapter, the proposed method has been presented and a method has been developed. Since it uses the same method of SEC-DED and XOR it can be easily implemented. The proposed method detects and corrects the soft error by using the checks bits that are smeared on the successive coding scheme of a data block in memory cells.

Chapter 4

Experimental Analysis

4.1 Introduction

The experimental analysis of this research work is divided into three sections: Experimental Setup, Result Analysis and Proposed Circuit Analysis. These are outlined shortly as follows.

4.2 Experimental Setup

To implement the proposed methodology, we used a system with the following configuration.

- Intel(R) Core (TM) i7-7700HQ CPU @ 2.80GHz
- 16 GB RAM
- Operating system Windows 10 (64 bit)

We have used the following software platforms.

- CodeBlocks IDE (v16.03)
- Pspice Cadence SPB OrCAD for circuit simulation

4.3 Result Analysis

Here, total data bits (6x4=24) are chosen for evaluating the effectiveness of the proposed method. Then, the error correction rates of the proposed technique and Reviriego et al. [1] are measured.

When the message is sent, we need to send the check bits to detect errors and correct them in the receiver's data. However, the number of check bits affects the performance of the error correction methods and hence, a metric 'Bit Overhead' is used to measure this performance. Bit overhead is derived using (1).

$$\text{Bit overhead } = \frac{No\ of\ Check\ Bits}{No\ of\ Data\ Bits} \times 100 \qquad (1)$$

It is better to have a lower bit overhead because it consumes lower memory space. On the other hand, it consumes a higher memory space when there is a high rate of bit overhead. So, we should minimize the bit overhead to minimize the space requirement of the error correction methodology.

In Figure 4.1, the bit overheads between Reviriego et al. [1] and the proposed method are presented for 16x8, 16x16, 16x32 and 16x64 data words. From Figure 4.1, we can observe that if the data word size increases, the percentage of bit overhead of Reviriego et al. [1] decreases, but for all data words, the percentage of bit overhead of the proposed method is a constant and it is always lower than Reviriego et al. [1].

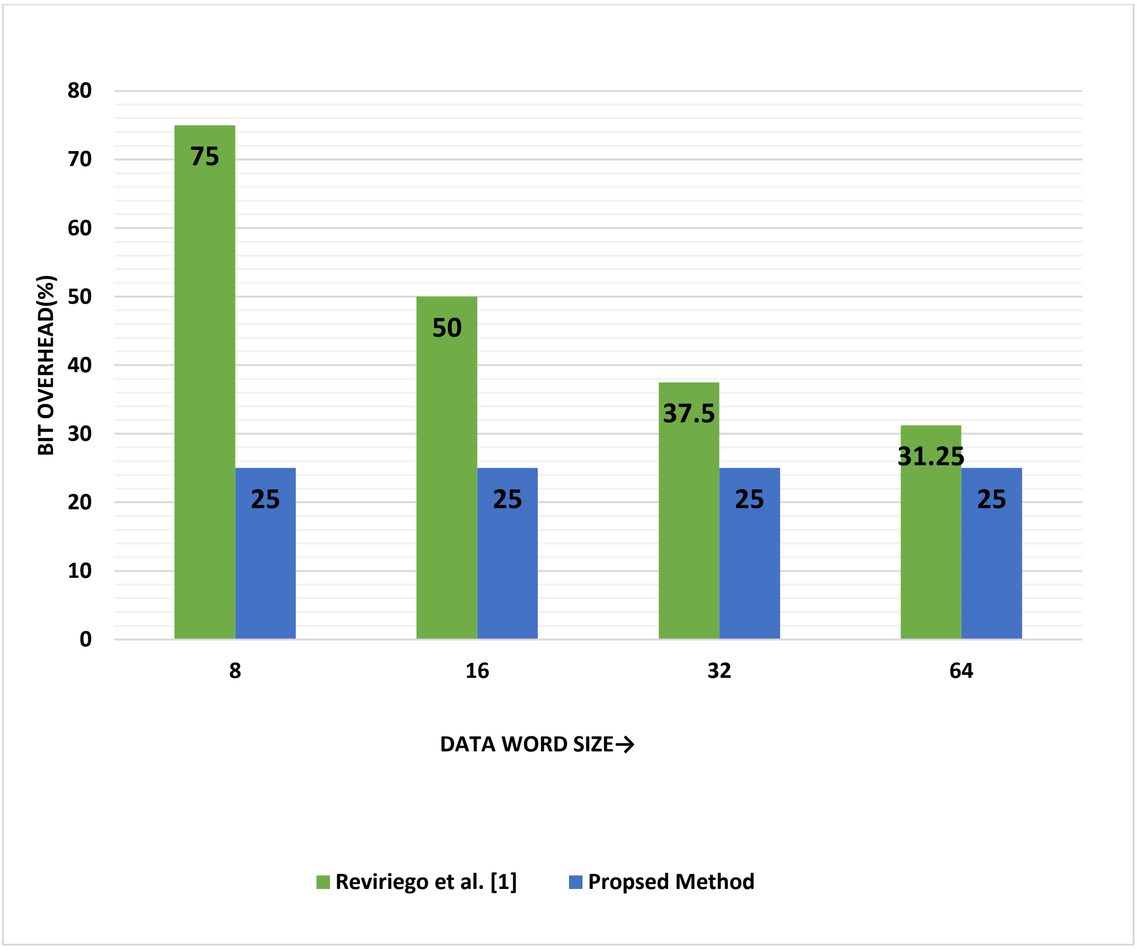


Figure 4.1: Comparison of bit overhead between reviriego et al. [1] and the proposed method

In Figure 4.2, the comparison of execution time between Reviriego et al.[1] and the proposed method is presented for data word sizes of 8, 16, 32, and 64. From this figure, we can observe that the execution times for the proposed method are a bit higher than that of Reviriego et al.[1].

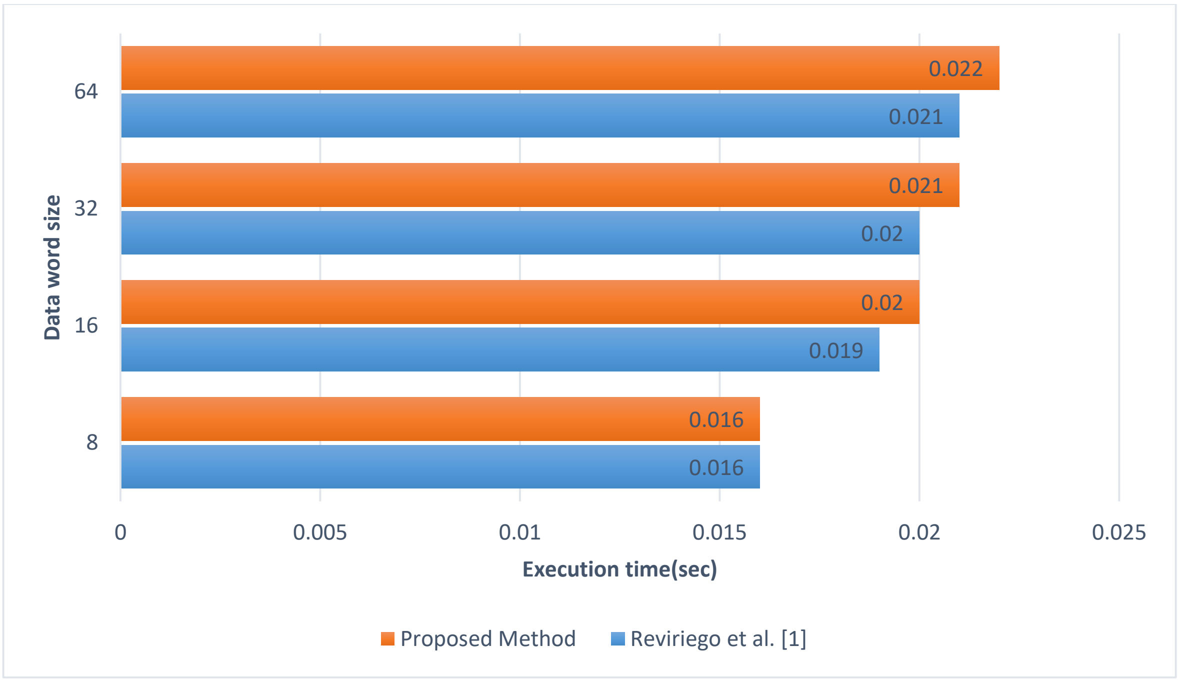


Figure 4.2: Comparison of execution time between reviriego et al. [1] and the proposed method.

The comparison of correctable single bit-error patterns between the proposed method and Reviriego et al. [1] is presented in TABLE I. In this table, b is the number of bits in the divided blocks of each rule and weight is the total number of '1' in every column which must be equal to 2^n, where n is an integer number. Otherwise, the weight is defined as an illegal weight. Reviriego et al. [1] proposed two general formulas to derive single bit-error correction methodology for weights 1 and 2 and these are shown in (2) and (3) respectively.

$$100 \times (1 - {}^{b}/_{2^b}) \qquad (2)$$

$$100 \times (1 - {}^{2}/_{2^b}) \qquad (3)$$

Reviriego et al. [1] can correct all single-bit error patterns for any weight except weights 1 and 2. However, the proposed method has overcome this limitation.

As shown in TABLE I, when the values of b (b is the number of bits in smaller blocks of each rule) are chosen as 2, 3, 4, and 5. Reviriego et al. [1] can correct all single-bit errors for any weight (total number of 1's per column) except weight 1 and 2 but the proposed method can correct all single-bit errors for any weight. For weights 1 and 2, if b increases, the single bit-error correction rate for Reviriego et al. [1] also increases and can correct nearly a hundred percent of single bit-error. On the other hand, the proposed method does not depend on the values of b and corrects all errors.

Table 4.1: Percentage of correctable single bit-error patterns for block memories

b →	**2**		**3**		**4**		**5**	
Weight→	***Reviriego et al. [1]***	***Proposed Method***	***Reviriego et al. [1]***	***Proposed Method***	***Reviriego et al. [1]***	***Proposed Method***	***Reviriego et al. [1]***	***Proposed Method***
0	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
1	**50**	**100**	**62.5**	**100**	**75**	**100**	**84.4**	**100**
2	**50**	**100**	**75**	**100**	**87.5**	**100**	**93.8**	**100**
≥4	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**

4.4 Circuit Analysis

In this section, we discuss the circuit. The erroneous message is shown in Figure 3.6. There are two rules R1= XX00 and R2 = 1X01 from Figure 3.7 for error detection and correction. Here, XX from R1 and 1X from R2 for block 1. Similarly, 00 from R1 and 01 from R2 for block 2. The 1st row of the erroneous message is 1010. Here, the first two bits and last two bits from LSB are for block 1 and block 2 respectively. Here, the common address for selecting the 1st row is 00. When the counter sends this address, it also sends this to the error detection and correction circuit. When the error detection and correction method gets the address, it matches the address with XX, 1X, 00 and 01 for binary cell 3, binary cell 2, binary cell 1 and binary cell 0 respectively. If the address 00 is matched any of these portions of the rules R1 and R2, the corresponding cell is set to 1. Else, it is set to 0. Here, the 1st row of the message is 1010. Now, the error detection and correction method does the rest of the work. By using the rule portions (XX, 1X, 00, 01) for corresponding outputs, the address 00 is matched only with XX and 00. Here, binary cell 3 and binary cell 1 are set to 1 and the rest are set to 0. Now the result for the first row is 1010. There is no error in the 1st row of the message. But for the 2nd row of the message is 0001. The common address for the 2nd row is 01. Now, this common address is matched only with (XX, 01) and binary cell 3 and binary cell 0 is set to 1. The rest are set to 0 in the 2nd row of the message. The result is for the 2nd row is 1001. The bit in binary cell 3 is corrected. Similarly, the 3rd row (1000) is selected. Then it is corrected to 1100. Here, the bit in binary cell 2 is corrected.

In Figure 4.3, we see the output of each row after error correction. The symbols □, ◊, ∇ and Δ are for binary cell 0, binary cell 1, binary cell 2 and binary cell 3. In the waveform in Figure 4.3, the 1st pulse is for 1st row. 2nd pulse is for 2nd row and so on. In the 1st pulse, we see that only binary cell 3 (Δ) and binary cell 1 (◊) are set to 1. Similarly, in the 2nd pulse, binary cell 3 and binary cell 0 are set to 1. For the 3rd and 4th pulse, the corresponding row is corrected. The waveform is shown in Figure 4.3.

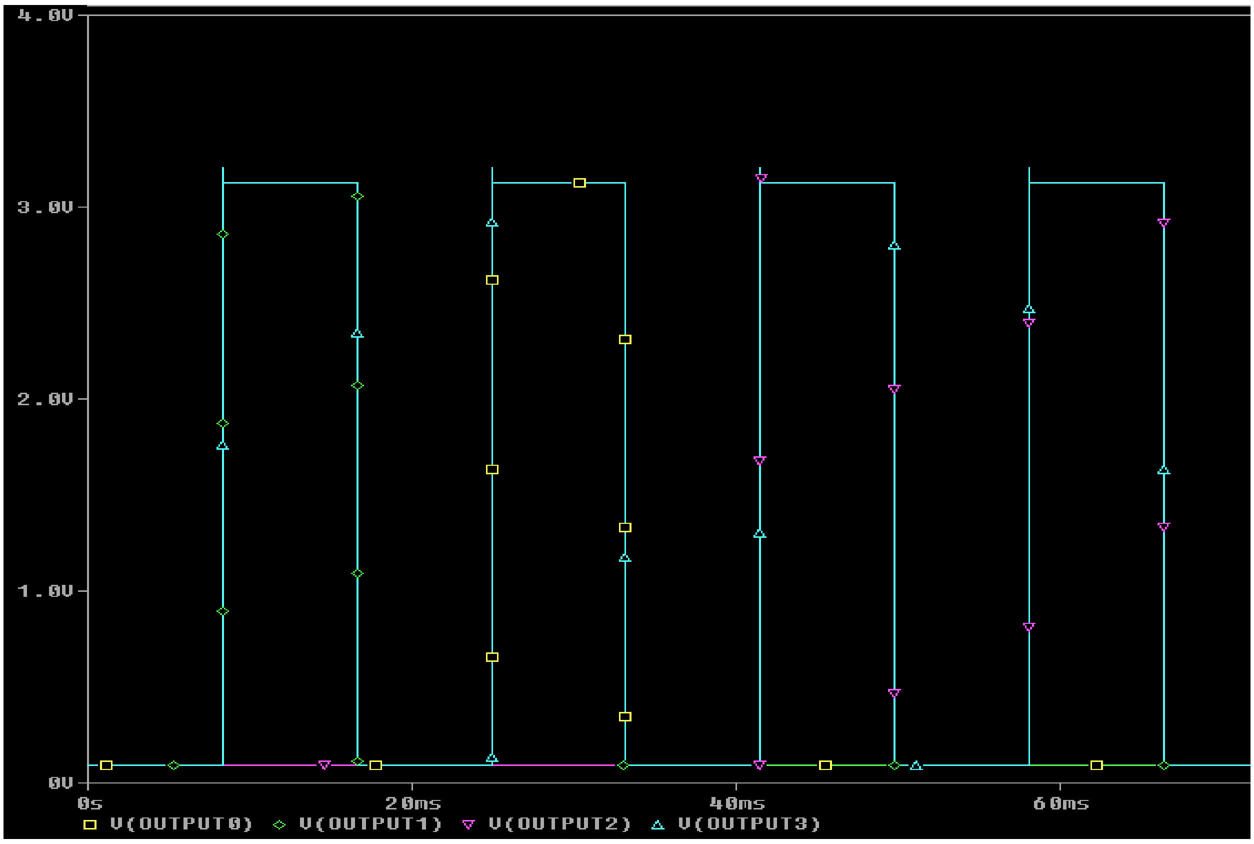


Figure 4.3. Outputs of the proposed method without memristor for 16 bits data.

Chapter 5

Conclusions

5.1 Concluding Remarks

This research work has shown a better tolerance against soft errors. It has proposed a simple methodology that can easily detect and correct the erroneous bits with relatively lesser redundant bits. It can correct soft errors at any position and protect the contents of the TCAM. Though this research work requires comparatively more time for error correction because of correcting all possible errors, it can be used as an effective error-tolerant method for those structures where reliability is a matter of concern. There is an open scope to enhance this research which is to lower the time requirement needed for error correction.

5.2 Future Work

In our thesis, our proposed method needs a little bit more execution time than other methods because of the detection and correction of random errors at any position. In our future work, we will improve our method which will require less time for execution.

List of Publication

Our thesis work was published in the 2nd International Conference on Robotics, Electrical and Signal Processing Techniques 2021 (ICREST'21). The source is given below.

S. Alam, A. A. Zobayer and M. S. Sadi, "Towards Tolerating Soft Errors in TCAM," *2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, 2021, pp. 722-726, doi: 10.1109/ICREST51555.2021.9331148.

Bibliography

[1] P. Reviriego, S. Pontarelli and A. Ullah, "Error Detection and Correction in SRAM Emulated TCAMs," in *IEEE Transactions on Very Large Scale Integration (VLSI) System*s, vol. 27, no. 2, pp. 486-490, Feb. 2019.

[2] S. Tambatkar, S. N. Menon, V. Sudarshan, M. Vinodhini and N. S. Murty, "Error detection and correction in semiconductor memories using 3D parity check code with hamming code," *2017 International Conference on Communication and Signal Processing (ICCSP),* Chennai, 2017, pp. 0974-0978.

[3] N. Sunderajan, "Quantifying Soft Error Rate and Demonstrating Systematic Capability of Programmable Electronic System," *2020 Annual Reliability and Maintainability Symposium (RAMS)*, Palm Springs, CA, USA, 2020, pp. 1-6.

[4] W. Toghuj, "Mitigation of Soft Errors in Implantable Medical Devices," *2020 7th International Conference on Electrical and Electronics Engineering (ICEEE),* Antalya, Turkey, 2020, pp. 95-100.

[5] T. Li, Q. Yu, H. Wan and S. Li, "Application specified soft-error failure rate analysis using sequential equivalence checking techniques," in *Tsinghua Science and Technology*, vol. 25, no. 1, pp. 103-116.

[6] R. Avazeh and N. Yazdani, "A New TCAM Architecture for IP Routing With Update Complexity Equal to O(1)," in *Canadian Journal of Electrical and Computer Engineering*, vol. 43, no. 4, pp. 207-217, Fall 2020.

[7] R. Afrin, M. S. Sadi and J. Jürjens, "An Efficient Soft Error Tolerant Approach to Enhance Reliability of TCAM," *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT), Dhaka, Bangladesh, 2019*, pp. 1-6.

[8] Sriram C. Krishnan, Rina Panigrahy, and Sunil Parthasarathy, "Error-Correcting Codes for Ternary Content Addressable Memories", *IEEE Transactions on Computer*, vol. 58, no. 2, pp. 275 – 279, Feb. 2009.

[9] I. Ullah, J. Yang and J. Chung, "ER-TCAM: A Soft-Error-Resilient SRAM-Based Ternary Content-Addressable Memory for FPGAs," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 28, no. 4, pp. 1084-1088, April 2020.

[10] S. A. Shahriyar, M. A. Golap and M. S. Sadi, "An Efficient Error Correction Approach by using Successive Parity Generation,*" 2018 4th International Conference on Electrical*

Engineering and Information & Communication Technology (ICEEiCT), Dhaka, Bangladesh, 2018, pp. 302-307.

[11] C. L. Chen and M. Y. Hsiao, “Error-correcting codes for semiconductor memory applications: A state-of-the-art review*,” IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, Mar. 1984.

[12] I. Syafalni, T. Sasao, X. Wen, S. Holst and K. Miyase, "Soft-error tolerant TCAMs for high-reliability packet classifications," *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Ishigaki, 2014*, pp. 471-474.

[13] N. Onizawa, S. Matsunaga and T. Hanyu, "A Compact Soft-Error Tolerant Asynchronous TCAM Based on a Transistor/Magnetic-Tunnel-Junction Hybrid Dual-Rail Word Structure," *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems, Potsdam, 2014*, pp. 1-8.

[14] S. Pontarelli, M. Ottavi, A. Evans and S. Wen, "Error detection in Ternary CAMs using Bloom Filters," *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2013,* pp. 1474-1479.

[15] I. Ullah, J. Yang and J. Chung, "ER-TCAM: A Soft-Error-Resilient SRAM-Based Ternary Content-Addressable Memory for FPGAs," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 1084-1088, April 2020.

[16] R. Arakawa, N. Onizawa, J. -P. Diguet and T. Hanyu, "Multi-Context TCAM-Based Selective Computing: Design Space Exploration for a Low-Power NN," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 1, pp. 67-76, Jan. 2021, doi: 10.1109/TCSI.2020.3030104.

[17] Y. -J. Chang, K. -L. Tsai and Y. -C. Cheng, "Data Retention-Based Low Leakage Power TCAM for Network Packet Routing," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 2, pp. 757-761, Feb. 2021, doi: 10.1109/TCSII.2020.3014154.

[18] R. W. Hamming, "Error detecting and error correcting codes," in *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147-160, April 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[19] S. Muppalla and K. R. Vaddempudi, "A novel VHDL implementation of UART with single error correction and double error detection capability," *2015 International Conference on Signal Processing and Communication Engineering Systems*, 2015, pp. 152-156, doi: 10.1109/SPACES.2015.7058236.

[20] S. Vafi, "Cyclic Low Density Parity Check Codes With the Optimum Burst Error Correcting Capability," in *IEEE Access*, vol. 8, pp. 192065-192072, 2020, doi: 10.1109/ACCESS.2020.3032837.

[21] X. . -H. Peng and P. G. Farrell, "On Construction of the$(24, 12, 8)$Golay Codes," in *IEEE Transactions on Information Theory*, vol. 52, no. 8, pp. 3669-3675, Aug. 2006, doi: 10.1109/TIT.2006.876247.

[22] Costas Argyrides, Dhiraj K. Pradhan, and Taskin Kocak, "Matrix Codes for Reliable and Cost Efficient Memory Chips," *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, vol. 19, NO. 3, Mar. 2011.

[23] Dubrova, E. 2012. "Fault Tolerant Design: An Introduction," 1st ed. *Springer New York Heidelberg Dordrecht London*. p. 87-131.

[24] Muhammad Imran, Zaid Al-Ars, Georgi N. Gaydadjiev "Improving Soft Error Correction Capability of 4-D Parity Codes," *14th IEEE European Test Symposium*, May. 2009.

[25] Costas Argyrides, Dhiraj K. Pradhan, and Taskin Kocak, "Matrix Codes for Reliable and Cost Efficient Memory Chips," *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, vol. 19, NO. 3, Mar. 2011.

[26] Jing Guo, Liyi Xiao, Zhigang Mao and Qiang Zhao, "Enhanced Memory Reliability Against Multiple Cell Upsets Using Decimal Matrix Code," *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, vol: 22, Issue: 1, pp. 1-9, Jan. 2014.