

Efficient Fault Detection Majority Logic Correction with in Memory with Difference-Set Codes

N. V. P. Naidu Babu¹, P. M. Francis², B. Prasad Kumar³

¹M.Tech, GITAS College, Bibbili, A. P, India

²Department of ECE, Head of Department, Assistant Professor, GITAS, College, Bobbili, A. P, India

³Department of ECE, Assistant Professor, GITAS, College, Bobbili, A. P, India

Abstract: Now a-days on soc applications the major problem is with the on chip memory to be faster and we require without any error correction and disrupt the altering digital circuit are becoming the major concern for memory application. This paper presents an error-detection method for difference-set cyclic codes with majority logic decoding. To correct a large no of correction Majority logic decodable codes are suitable for memory applications. However, they require a large decoding time that impacts memory performance. The proposed fault-detection method significantly reduces memory access time when there is no error in the data read. The technique uses the majority logic decoder itself to detect failures, which makes the area overhead minimal and keeps the extra power consumption low.

Keywords: Block codes, difference-set, error correction codes (ECCs), low-density parity check (LDPC), majority logic, memory.

1. Introduction

The impact of technology scaling smaller dimensions, higher integration densities, and lower operating voltages has come to a level that reliability of memories is put into jeopardy, not only in extreme radiation environments like spacecraft and avionics electronics, but also at normal terrestrial environments [1], [2]. Especially, SRAM memory failure rates are increasing significantly, therefore posing a major reliability concern for many applications. Some commonly used mitigation techniques are:

- Triple Modular Redundancy (TMR);
- Error Correction Codes (ECCs).

TMR is a special case of the von Neumann method [3] consisting of three versions of the design in parallel, with a majority voter selecting the correct output. As the method suggests, the complexity overhead would be three times plus the complexity of the majority voter and thus increasing the power consumption. For memories, it turned out that ECC codes are the best way to mitigate memory soft errors [2]. For terrestrial radiation environments where there is a low soft error rate (SER), codes like single error correction and double error detection (SEC-DED), are a good solution, due to their low encoding and decoding complexity. However, as a consequence of augmenting integration densities, there is an increase in the number of soft errors, which produces the need for higher error correction capabilities [4], [5]. The usual multi error correction codes, such as Reed-Solomon (RS) or Bose-Chaudhuri-Hocquenghem (BCH) are not suitable for this task. The reason for this is that they use more sophisticated decoding algorithms, like complex algebraic (e.g., floating point operations or logarithms) decoders that can decode in fixed time, and simple graph decoders, that use iterative algorithms (e.g., belief propagation). Both are very complex and increase

computational costs [6]. Among the ECC codes that meet the requirements of higher error correction capability and low decoding complexity, cyclic block codes have been identified as good candidates, due to their property of being majority logic (ML) decodable [7], [8]. A subgroup of the low-density parity check (LDPC) codes, which belongs to the family of the ML decodable codes, has been researched in [9]–[11]. In this paper, we will focus on one specific type of LDPC codes, namely the difference-set cyclic codes (DSCCs), which is widely used in the Japanese tele-text system or FM multiplex broadcasting systems [12]–[14]. The main reason for using ML decoding is that it is very simple to implement and thus it is very practical and has low complexity. The drawback of ML decoding is that, for a coded word of n -bits, it takes n cycles in the decoding process, posing a big impact on system performance [6]. One way of coping with this problem is to implement parallel encoders and decoders. This solution would enormously increase the complexity and, therefore, the power consumption. As most of the memory reading accesses will have no errors, the decoder is most of the time working for no reason. This has motivated the use of a fault detector module [11] that checks if the codeword contains an error and then triggers the correction mechanism accordingly. In this case, only the faulty code words need correction, and therefore the average read memory access is speeded up, at the expense of a n increase in hardware cost and power consumption. A similar proposal has been presented in [15] for the case of flash memories. The simplest way to implement a fault detector for an ECC is by calculating the syndrome, but this generally implies adding another very complex functional unit. This paper explores the idea of using the ML decoder circuitry as a fault detector so that read operations are accelerated with almost no additional hardware cost. The results show that the properties of DSCC-LDPC enable efficient fault detection. The remainder of this paper is organized as follows. Section II gives an overview of

existing ML decoding solutions; Section III presents the novel ML detector/decoder (MLDD) using difference-set cyclic codes; Section IV discusses the results obtained for the different versions in respect to effectiveness, performance, and area and power consumption. Finally, Section V discusses conclusions and gives an outlook onto future work.

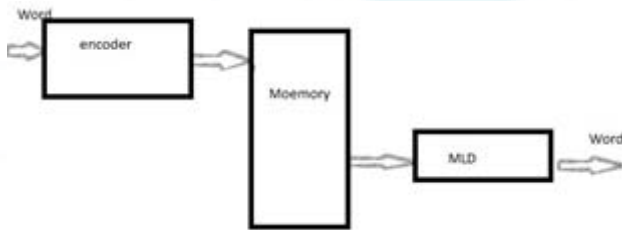


Figure 1: Memory system schematic with MLD

2. Existent Majority Logic Decoding (MLD) Solutions

MLD is based on a number of parity check equations which are orthogonal to each other, so that, at each iteration, each codeword bit only participates in one parity check equation, except the very first bit which contributes to all equations. For this reason, the majority result of these parity check equations decide the correctness of the current bit under decoding. MLD was first mentioned in [7] for the Reed-Müller codes. Then, it was extended and generalized in [8] for all types of systematic linear block codes that can be totally orthogonalized on each codeword bit.

A generic schematic of a memory system is depicted in Fig. 1 for the usage of an ML decoder. Initially, the data words are encoded and then stored in the memory. When the memory is read, the codeword is then fed through the ML decoder before sent to the output for further processing. In this decoding process, the data word is corrected from all bit-flips that it might have suffered while being stored in the memory. There are two ways for implementing this type of decoder. The first one is called the Type-I ML decoder, which determines, upon XOR combinations of the syndrome, which bits need to be corrected [6]. The second one is the Type-II ML decoder that calculates directly out of the codeword bits the information of correctness of the current bit under decoding [6]. Both are quite similar but when it comes to implementation, the Type-II uses less area, as it does not calculate the syndrome as an intermediate step. Therefore, this paper focuses only on this one.

2.1 Plain ML Decoder

As described before, the ML decoder is a simple and powerful decoder, capable of correcting multiple random bit-flips depending on the number of parity check equations. It consists of four parts: 1) a cyclic shift register; 2) an XOR matrix; 3) a majority gate; and 4) an XOR for correcting the codeword bit under decoding, as illustrated in Fig. 2. The input signal is initially stored in the cyclic shift register and shifted through all the taps. The intermediate values in each tap are then used $\{B_j\}$ to calculate the results of the

check sum equations from the XOR matrix. In the N cycle, the result has reached the final tap, producing the output signal Y (which is the decoded version of input x). As stated before, x input might correspond to wrong data corrupted by a soft error. To handle this situation, the decoder would behave as follows. After the initial step, in which the codeword is loaded in to the cyclic shift register, the decoding starts by calculating the parity check equations hardwired in the XOR matrix. The resulting sums $\{B_j\}$ are then forwarded to the majority gate for evaluating its correctness. If the number of 1's received is greater than the number of 0's that would mean that the current bit under decoding is wrong and a signal to correct it would be triggered. Otherwise, the bit under decoding would be correct and no extra operations would be needed on it. In the next step, the content of the registers are rotated and the above procedure is repeated until all N codeword bits have been processed. Finally, the parity check sums should be zero if the codeword has been correctly decoded. Further details on how this algorithm works can be found in [6]. The whole algorithm is depicted in Fig. 3. The previous algorithm needs as many cycles as the number of bits in the input signal, which is also the number of taps of, N , in the decoder. This is a big impact on the performance of the system, depending on the size of the code. For example, for a codeword of 73 bits, the decoding would take 73 cycles, which would be excessive for most applications.

2.2 Plain MLD with Syndrome Fault Detector (SFD)

In order to improve the decoder performance, alternative designs may be used. One possibility is to add a fault detector by calculating the syndrome, so that only faulty codewords are decoded [11]. Since most of the codewords will be error-free, no further correction will be needed, and therefore performance will not be affected. Although the implementation of an SFD reduces the average latency of the decoding process, it also adds complexity to the design (see Fig. 4). The SFD is an XOR matrix that calculates the syndrome based on the parity check matrix. Each parity bit results in a syndrome equation. Therefore, the complexity of the syndrome calculator increases with the size of the code. A faulty codeword is detected when at least one of the syndrome bits is "1." This triggers the MLD to start the decoding, as explained before. On the other hand, if the codeword is error-free, it is forwarded directly to the output, thus saving the correction cycles. In this way, the performance is improved in exchange of an additional module in the memory system: a matrix of XOR gates to resolve the parity check matrix, where each check bit results into a syndrome equation. This finally results in a quite complex module, with a large amount of additional hardware and power consumption in the system.

3. Proposed ML Detector/Decoder

This section presents a modified version of the ML decoder that improves the designs presented before. Starting from the original design of the ML decoder introduced in [8], the proposed ML detector/decoder (MLDD) has been implemented using the difference-set cyclic codes (DSCCs)

[16]–[19]. This code is part of the LDPC codes, and, based on their attributes, they have the following properties:

- Ability to correct large number of errors;
- Sparse encoding, decoding and checking circuits synthesizable into simple hardware;
- Modular encoder and decoder blocks that allow an efficient hardware implementation;
- Systematic code structure for clean partitioning of information and code bits in the memory

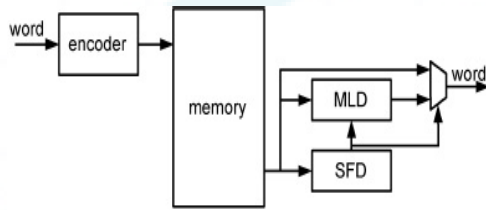


Figure 2: Memory system schematics for ML decoder with SFD

An important thing about the DSCC is that its systematic distribution allows the ML decoder to perform error detection in a simple way, using parity check sums (see [6] for more details). However, when multiple errors accumulate in a single word, this mechanism may misbehave, as explained in the following. In the simplest error situation, when there is a bit-flip in a codeword, the corresponding parity check sum will be “1,” as shown in Fig. 5(a). This figure shows a bit-flip affecting bit 42 of a codeword $N=73$ with length and the related check sum that produces a “1.” However, in the case of Fig. 5(b), the codeword is affected by two bit-flips in bit 42 and bit 25, which participate in the same parity check equation. So, the check sum is zero as the parity does not change. Finally, in Fig. 5(c), there are three bit-flips which again are detected by the check sum (with a “1”). As a conclusion of these examples, any number of odd bitflips can be directly detected, producing a “1” in the corresponding B_j . The problem is in those cases with

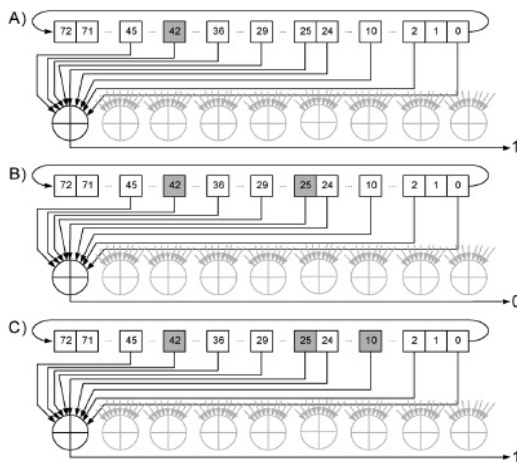


Figure 3: single check equation of a $N = ML73$ decoder a) one bit flip b) two bit flip c) three bit flip

An even number of bit-flips, where the parity check equation would not detect the error. In this situation, the use

of a simple error detector based on parity check sums does not seem feasible, since it can not handle “false negatives” (wrong data that is not detected). However, the alternative would be to derive all data to the decoding process (i.e., to decode every single word that is read in order to check its correctness), as explained in previous sections, with a large performance overhead. Since performance is important for most applications, we have chosen an intermediate solution, which provides a good reliability with a small delay penalty for scenarios where up to five bit-flips may be expected (the impact of situations with more than five bit-flips will be analyzed in Section IV-A). This proposal is one of the main contributions of this paper, and it is based on the following hypothesis: Given a word read from a memory protected with DSCC codes, and affected by up to five bit-flips, all errors can be detected in only three decoding cycles. This is a huge improvement over the simpler case, where decoding cycles are needed to guarantee that errors are detected. The proof of this hypothesis is very complex from the mathematical point of view. Therefore, two alternatives have been used in order to prove it, which are given here.

- Through simulation, in which exhaustive experiments have been conducted, to effectively verify that the hypothesis applies (see Section IV).

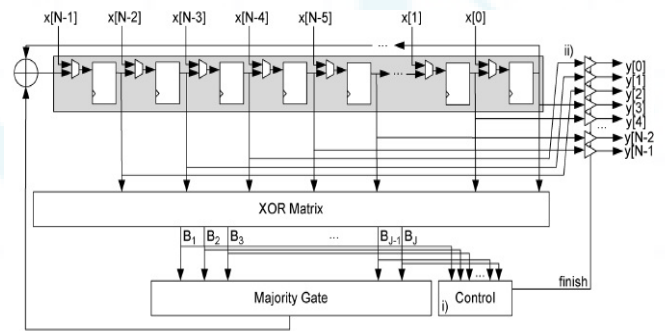


Figure 4: Proposed MLDD schematic 1) control unit 2) tri-state buffer

- Through a simplified mathematical proof for the particular case of two bit-flips affecting a single word (see Appendix). For simplicity, and since it is convenient to first describe the chosen design, let us assume that the hypothesis is true and that only three cycles are needed to detect all errors affecting up to five bits (this will be confirmed in Section IV). In general, the decoding algorithm is still the same as the one in the plain ML decoder version. The difference is that, instead of decoding all codeword bits by processing the ML decoding during cycles, the proposed method stops intermediately in the third cycle, as illustrated in Fig. 6.

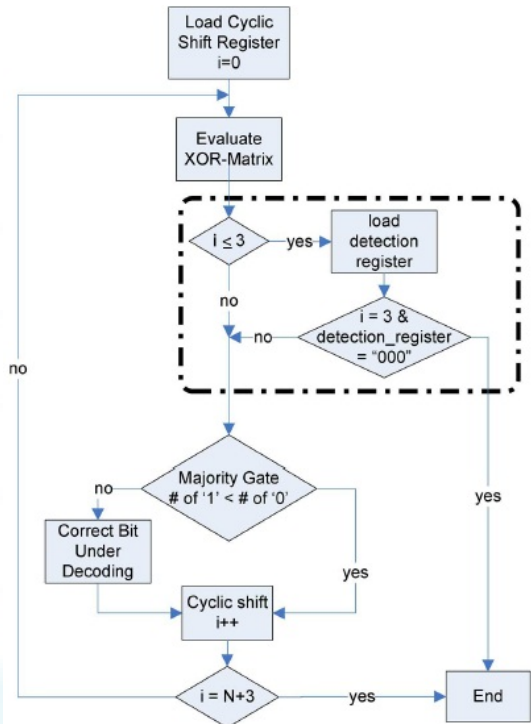


Figure 5: flowchart of MLDD algorithm

tristate buffers. The output tristate buffers are always in high impedance unless the control unit sends the finish signal so that the current values of the shift register are forwarded to the output. The control schematic is illustrated in Fig. 8. The control unit manages the detection process. It uses a counter that counts up to three, which distinguishes the first three iterations of the ML decoding. In these first three iterations, the control unit evaluates the by combining them with the OR1 function. This value is fed in to a three-stage shift register, which holds the results of the last three cycles. In the third cycle, the OR2 gate evaluates the content of the detection register. When the result is “0,” the FSM sends out the finish signal indicating that the processed word is error-free. In the other case, if the result is “1,” the ML decoding process runs until the end. This clearly provides a performance improvement respect to the traditional method. Most of the words would only take three cycles (five, if we consider the other two for input/output) and only those with errors (which should be a minority) would need to perform the whole decoding process. More information about performance details will be provided in the next sections. The one in Figure 1, adding the control logic in the MLDD module.

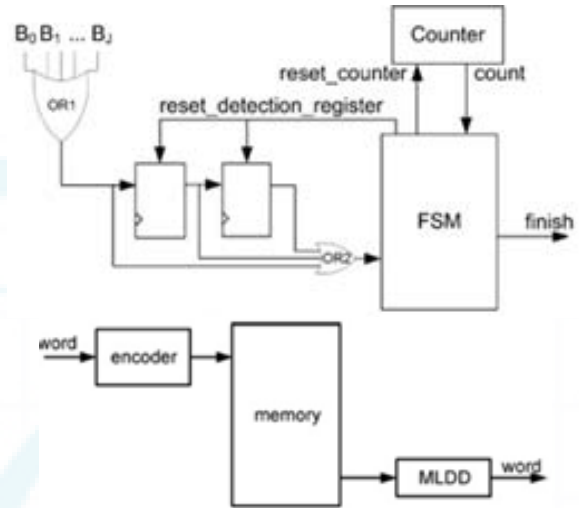


Figure 6: the proposed MLDD memory design

4. Results

The experimental results to measure the effectiveness, performance and area of the proposed technique will be presented.

4.1 Effectiveness

Here, the hypothesis that any error pattern affecting up to five bits in a word can be detected in just three cycles of the decoding process will be verified. Additionally, the detection of errors affecting a larger number of bits is also briefly discussed. As stated in previous sections, an odd number of errors will not pose any problem to a traditional parity check detector, but an even number will. Therefore, this is the scenario that has been studied. Several word widths have been considered in order to perform the experiments. The details are shown in Table I, where, for each size, the number of data and parity bits are stated. Given a size, all combinations of two and four bit-flips on a word have been calculated, in order to study all of the possible cases. The number of combinations can be seen in Table I for different values of width double and quadruple errors.

Table 1: Data word length

N	Data bits	Parity bits
73 48		35
273 191		82
1057 833		244

As expected, increasing the code length implies an exponential growth of the number of combinations, and therefore, of the computational time. An important final comment is that the area overhead of the MLDD actually decreases with respect to the plain MLD version. For large values of, both areas are practically the same. The reason for this is that the error detector in the MLDD has been designed to be independent of the size code. The opposite situation occurs, with the SFD technique, which uses syndrome calculation to perform error detection: its complexity grows quickly when the code size increases. One of the problems to make the MLDD module independent of

