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

K.Rama Theertha*, Sri G.Ramesh**

* Department of ECE, G.Pulla Reddy Engineering College
** Department of ECE, G.Pulla Reddy Engineering College

Abstract- Even a small transition delays and little faults create major concern in digital circuits. It Produce greater impact on not only for simple memory but also for most of the memory applications. This paper presents an error-detection method for difference-set cyclic codes with majority logic decoding. Majority logic decodable codes are suitable for memory applications due to their capability to correct a large number of errors. 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. The proposed method detects the occurrences of single error, double error ,triple error in the received code words obtained from the memory system.

Index Terms- majority logic (ML), Low-density parity check (LDPC), Difference-set cyclic codes (DSCCs), Triple modular redundancy (TMR), Error correction codes (ECCs), Soft error rate (SER), ML Detector / Decoder (MLDD).

I.INTRODUCTION

The impact of technology scaling smaller dimensions, higher integration densities, and lower operating volt- ages—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 terres- trial 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] con- sisting 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 consump- tion. 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 multierror cor- rection codes, such as Reed–Solomon (RS) or Bose–Chaud- huri–Hocquenghem (BCH) are not suitable for this task. The reason for this is that they use more sophisticated decoding algo- rithms, 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 prop- agation). 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 sub- group of the low-density parity check (LDPC) codes, which be- longs to the family of the ML decodable codes, has been re- searched 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 teletext 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 -bits, it takes cycles in the decoding process,

posing a big impact on system performance [6].

One way of coping with this problem is to implement par- allel 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 codewords need correction, and therefore the average read memory access is speeded up, at the expense of an increase in hardware cost and power consumption. A similar proposal has been presented in [15] for the case of flash memories..

This paper explores the idea of using the ML decoder cir- cuitry 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.

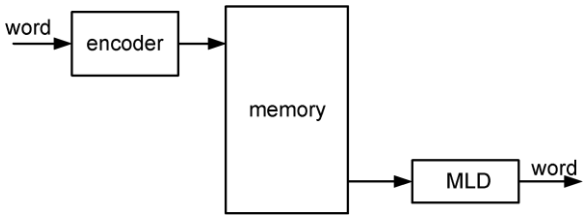


Figure1: Memory system schematic with MLD

II. 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 code- word 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 de- cide 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 Figure1 for the usage of an ML decoder. Initially, the data words are en- coded 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 suf- fered 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.

A. Plain ML Decoder:

As described before, the ML decoder is a simple and pow- erful 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 ma- jority gate; and 4) an XOR for correcting the codeword bit under decoding, as illustrated in Figure 2.

The input signal is initially stored into the cyclic shift reg- ister and shifted through all the taps. The intermediate values in each tap are then used to calculate the results {Bj} of the check sum equations from the XOR matrix. In the Nth cycle, the result has reached the final tap, producing the output signal.

As stated before, 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 into the cyclic shift register, the decoding starts by calculating the parity check equations hardwired in the XOR matrix. The resulting sums are then forwarded to the majority gate for evaluating its correctness. If the number of 1's received in {Bj} 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.

$\{B_j\}$

$\{B_j\}$

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].

N

The previous algorithm needs as many cycles as the number of bits in the input signal, which is also the number of taps, , 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.

N

B. Plain MLD With Syndrome Fault Detector (SFD):

In order to improve the decoder performance, alternative de- signs 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 re- duces the average latency of the decoding process, it also adds complexity to the design (see Figure 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.

III. 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 synthesiz- able into simple hardware;
- modular encoder and decoder blocks that allow an efficient hardware implementation;
- systematic code structure for clean partition of information and code bits in the memory.

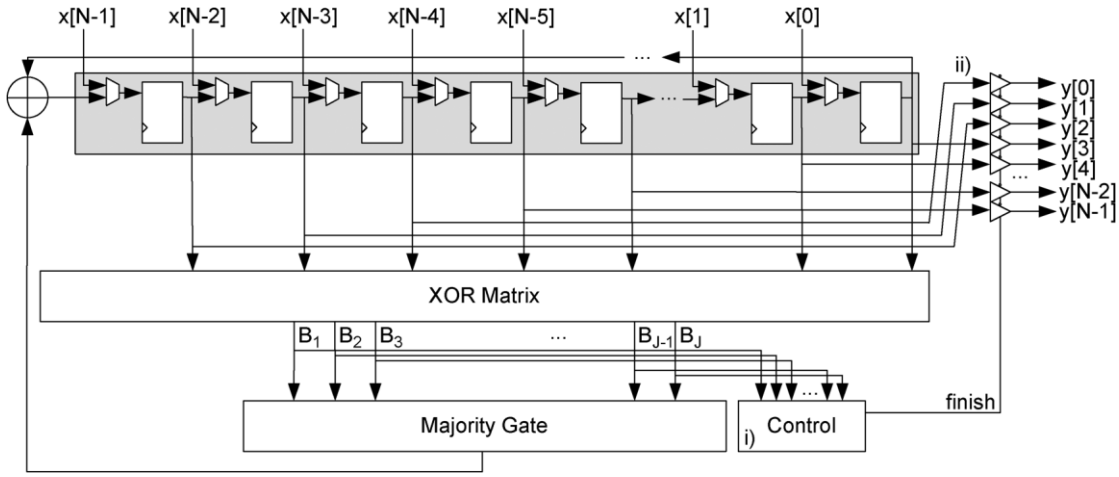


Fig.ure 7. Schematic of the proposed MLDD. i) Control unit. ii) Output tristate buffers

.

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.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

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).

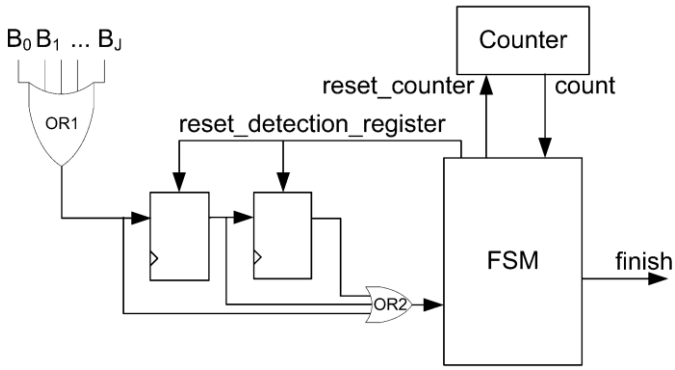
In general, the decoding algorithm is still the same as the one in the plain ML decoder version. The difference is that, instead of decoding all codeword bits by processing the ML decoding during N cycles, the proposed method stops intermediately in the third cycle.

If in the first three cycles the evaluation of the XOR matrix for all {Bj} is "0," the codeword is de- termined to be error-free and forwarded directly to the output. If the {Bj} contain in any of the three cycles at least a "1," the proposed method would continue the whole decoding process in order to eliminate the errors.A detailed schematic of the proposed design is shown.

The figure shows the basic ML decoder with an N-tap shift register, an XOR array to calculate the orthogonal parity check sums and a majority gate for deciding if the current bit under decoding needs to be inverted. Those components are the same as the ones for the plain ML decoder. The additional hardware to perform the error detection is illustrated in Fig. 7 as: i) the control unit which triggers a finish flag when no errors are detected after the third cycle and ii) the output 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 Figure 8.

Figure 8. Schematic of the control unit.



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 itera- tions, the control unit evaluates the {Bj} by combining them with the OR1 function. This value is fed into 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.

IV. RESULTS

The tabular column I shows the number of data bits and parity bits for different values of 'N' according to difference-set codes.

Table I: DSCC LENGTHS

N	data bits	parity bits
73	45	38
273	191	82
1057	813	244

A. Effectiveness:

Table II shows the results for errors with two bit-flips. These results confirm that with only one decoding cycle, the detection method is covering more than 90% of the error patterns for all

. The second cycle increases the percentage of detection and after the third one, 100% of the errors are detected

Table II:
EXHAUSTIVE SEARCH RESULTS FOR TWO BIT-FLIPS

iteration	N=73	N=273	N=1057
1	90.41%	94.51%	99.49%
2	99.20%	99.72%	99.99%
3	100.00%	100.00%	100.00%

Table III:
PERFORMANCE OF THE DIFFERENT MODELS

model	cycles			
	I/O	detecting	no errors	errors
plain MLD	2	N	N+2	N+2
SFD	2	1	3	N+2
MLDD	2	3	5	N+5

Table III shows the performance of three models of MLD in terms of cycles. And it also compares the proposed MLD with previous ones in performance.

Table IV:
SPEED-UP OF THE PROPOSED MLDD FOR ERROR-FREE CODEWORDS

N	plain ML decoding	proposed ML detection	speed-up
73	75	5	15
273	275	5	55
1057	1059	5	211.8

Table IV shows the performance of two models of MLD in terms of cycles. And it also compares the proposed MLD with previous ones in performance.

B.Memory Read Access Delay:

The memory read access delay of the plain MLD is directly dependent on the code size, i.e., a code with length 73 needs 73 cycles, etc. Then, two extra cycles need to be added for I/O. On the other hand, the memory read access delay of the proposed MLDD is only dependent on the word error rate (WER). If there are more errors, then more words need to be fully decoded.

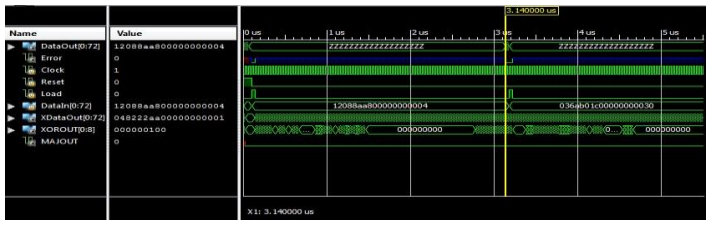
The detection phase of the MLDD is code-length-independent and therefore has the same number of cycles for all N. A summary of the performance of the three different designs.

The “I/O” column represents the number of cycles the design needs to forward the data to the registers and to read from those registers to the output. It has the same value for all the designs. The “detecting” column gives the actual number of cycles the design needs to detect an error in the codeword. In the case that there are no errors in the codeword, the designs would need, in total, the number of cycles given in the “no errors” column (which is the addition of the “I/O” and “detecting” columns). On the other hand, the “errors” column gives the total number of cycles needed by the design to correct the errors in the codeword. The three designs that have been compared are the plain ML decoder (MLD), the ML decoder with a syndrome calculator for error detection (SFD), and the proposed MLDD. As it can be seen, the plain MLD always needs N cycles in all cases. The SFD, however, is able to detect in just one single cycle (plus 2 of I/O) if the codeword is error free and forward it to the output. The performance of the proposed design is closer to that of the SFD rather than to the MLD. It just requires three cycles to detect any error (plus two of I/O).

C.Simulation results:

Figure 1 shows the simulation results of the existing method MLD which requires ‘N’ cycles to decode ‘N’ bits. The figure will be as follows:

Fig.ure 1: Simulation of MLD for 73 bits



Here, figure 2 shows the simulation results of the proposed method MLDD which requires '5' cycles to decode 'N' bits and is independent of code size 'N'.

Fig.ure 11. Simulation of MLDD for 73 bits

D. Area

The previous subsection showed that the performance of the proposed design MLDD is much faster than the plain MLD version, but slightly lower than the design with syndrome calculator (SFD).

As mentioned several times, this is compensated with a clear savings in area. To study this, the three designs have been implemented in VHDL and synthesized, for different values using a TSMC35 library. The obtained results are depicted, in number of equivalent gates.

Table V: Synthesis Results of the three designs for different code lengths

N	MLD	SFD	overhead	MLDD	overhead
21	315	395	25.40%	347	10.16%
73	983	1460	48.52%	1023	4.07%
273	3441	7185	108.81%	3488	1.37%
1057	12935	45148	249.04%	12991	0.43%

The conclusions of Table V on the area results are given as follows:

- The MLD design requires little area compared with the other two designs. However, as seen before, the performance results are not very good.
- The SFD version, which had the best performance, needs more area than the MLD does, ranging from 25.40% to 294.94% depending on N. Notice that the increment of area grows quicker.
- The MLDD version has a very similar performance to SFD, however it requires a much lower area overhead.

These conclusions can be extrapolated to power. The over-head introduced by MLDD is very small, contrary to the SFD.

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 N, 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 has been the mapping of the intermediate delay line values to the output signals. The reason is that this module behaves in two different ways depending if the processed word is erroneous or correct. If it is correct, its output is driven after the third cycle, what means that the word has been shifted three positions in the line register. If it is wrong, the word has to be fully decoded, what implies being shifted positions. So, both scenarios end up with the output values at different positions of the shift register. Then some kind of multiplexing logic would be needed to reorder the bits before mapping them to the output. However, the area of this logic would grow with linearly. In order to avoid this, it has been decided to make three extra shift movements in the case of a wrong word, in order to align its bits with those of a correct word. After this, the output bits are coherent in all situations, not needing multiplexing logic. The penalty for this solution is three extra cycles to decode words with errors, which usually has a negligible impact on performance.

.

V. CONCLUSION

In this paper, a fault-detection mechanism, MLDD, has been presented based on ML decoding using the DSCCs. Exhaustive simulation test results show that the proposed technique is able to detect any pattern of up to five bit-flips in the first three cycles of the decoding process. This improves the performance of the design with respect to the traditional MLD approach.

On the other hand, the MLDD error detector module has been designed in a way that is independent of the code size. This makes its area overhead quite reduced compared with other tra- ditional approaches such as the syndrome calculation (SFD).

In addition, a theoretical proof of the proposed MLDD scheme for the case of double errors has also been presented. The extension of this proof to the case of four errors would confirm the validity of the MLDD approach for a more general case, something that has only been done through simulation in the paper. This is, therefore, an interesting problem for future research. The application of the proposed technique to memo- ries that use scrubbing is also an interesting topic and was in fact the original motivation that led to the MLDD scheme.

VI.REFERENCES

[1] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Trans. Device Mater. Reliabil.*, vol. 5, no. 3, pp. 397–404, Sep. 2005.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semicon- ductor technologies," *IEEE Trans. Device Mater. Reliabil.*, vol. 5, no. 3, pp. 301–316, Sep. 2005.

[3] J. von Neumann, "Probabilistic logics and synthesis of reliable organ- isms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[4] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2004.

[5] I. S. Reed, "A class of multiple-error-correcting codes and the decoding scheme," *IRE Trans. Inf. Theory*, vol. IT-4, pp. 38–49, 1954.

[6] J. L. Massey, *Threshold Decoding*. Cambridge, MA: MIT Press,1963.

[7] S. Ghosh and P. D. Lincoln, " Low-density parity check codes for error correction in nanoscale memory," SRI Comput. Sci. Lab. Tech. Rep. CSL-0703, 2007.

[8] B. Vasic and S. K. Chilappagari, "An information theoretical frame- work for analysis and design of nanoscale fault-tolerant memories based on low-density parity-check codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 54, no. 11, pp. 2438–2446, Nov. 2007.

[9] Y. Kato and T. Morita, "Error correction circuit using difference-set cyclic code," in *Proc. ASP-DAC*, 2003, pp. 585–586.

[10] T. Kuroda, M. Takada, T. Isobe, and O. Yamada, "Transmission scheme of high-capacity FM multiplex broadcasting system," *IEEE Trans. Broadcasting*, vol. 42, no. 3, pp. 245–250, Sep. 1996.

[11] O. Yamada, "Development of an error-correction method for data packet multiplexed with TV signals," *IEEE Trans. Commun.*, vol. COM-35, no. 1, pp. 21–31, Jan. 1987.

[12] P. Ankolekar, S. Rosner, R. Isaac, and J. Bredow, "Multi-bit error cor- rection methods for latency-contrained flash memory systems," *IEEE Trans. Device Mater. Reliabil.*, vol. 10, no. 1, pp. 33–39, Mar. 2010.

[13] E. J. Weldon, Jr., "Difference-set cyclic codes," *Bell Syst. Tech. J.*, vol. 45, pp. 1045–1055, 1966.

[14] C. Tjhai, M. Tomlinson, M. Ambroze, and M. Ahmed, "Cyclotomic idempotent-based binary cyclic codes," *Electron. Lett.*, vol. 41, no. 6, Mar. 2005.

[15] T. Shibuya and K. Sakaniwa, "Construction of cyclic codes suitable for iterative decoding via generating idempotents," *IEICE Trans. Funda- mentals*, vol. E86-A, no. 4, pp. 928–939, 2003.

[16] F. J. MacWilliams, "A table of primitive binary idempotents of odd length ," *IEEE Trans. Inf. Theory*, vol. IT-25, no. 1, pp. 118–123, Jan. 1979.

AUTHORS

K.Rama Theertha received the B.Tech. degree in electronics and communication engineering from Gates Institute of Technology, Gooty, in 2011, and persuing Masters Degree in VLSi & Embedded Systems specialization at G.Pulla Reddy Engineering College.

Sri G.Ramesh received the B.Tech degree in electronics and communication engineering from JNTU, hyderabad and received M.Tech. degree in Embedded Systems specialization from JNTU, hyderabad, respectively.

He has seven years experience in teaching profession. Now, he is working as Assistant Professor in department of electronics and communication engineering at G.Pulla Reddy Engineering College.