International Journal of Scientific & Engineering Research, Volume 4, Issue 5, May-2013
ISSN 2229-5518

Performance Improved Architecture of Majority Logic Decoder with Difference-Set decoder for High data rate Applications

C.Rajeswari Sengunthar Engineering College, chandranrajes@gmail.com

Abstract

To prevent soft errors from causing data corruption, memories are commonly protected with Error Correction Codes (ECCs). To minimize the impact of the ECC on memory complexity simple codes are commonly used. For example, Single Error Correction (SEC) codes, like Hamming codes are widely used. Power consumption can be reduced by first checking if the word has errors and then perform the rest of the decoding only when there is errors .Nowadays, single event upsets (SEUs) altering digital circuits are becoming a bigger concern for memory applications. 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. 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 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. 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 initiative of using the ML decoder circuitry as a fault detector so that read operations are accelerated with almost no supplementary hardware cost. The results show that the properties of DSCC-LDPC enable efficient fault detection.

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 . 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 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.

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. The usual multierror 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 .

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. 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 .

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 that checks if the codeword contains an error and then triggers the correction mechanism accordingly. In this case, only the faulty code word 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 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.

International Journal of Scientific & Engineering Research, Volume 4, Issue 5, May-2013
ISSN 2229-5518

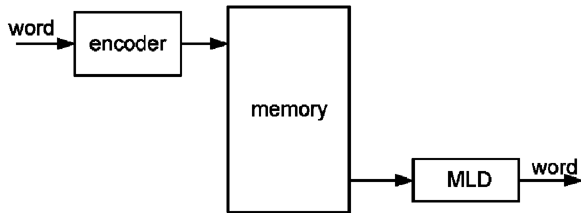


Fig. 1. Memory system schematic with MLD.

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 for the Reed–Muller codes. Then, it was extended and generalized for all types of systematic linear block codes that can be totally orthogonal zed 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. 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 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 x is initially stored into the cyclic shift register and shifted through all the taps. The intermediate values in each tap are then used to calculate the results $\{B_J\}$ of the check sum equations from the XOR matrix. In the Nth cycle, the result has reached the final tap, producing the output signal y

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 $\{B_J\}$ are then forwarded to the majority gate for evaluating its correctness. If the number of 1's received in $\{B_J\}$ 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. 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, 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.

B. 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 codeword are decoded. Since most of the code words 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). 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 partition of information and code bits in the memory.

An important thing about the DSCC is that its systematical

IJSER © 2013
http://www.ijser.org

distribution allows the ML decoder to perform error detection in a simple way, using parity check sum. 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 with length $N = 73$ 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 bit- flips can be directly detected, producing a "1" in the corresponding . The problem is in those cases with an even numbers 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 cannot 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 N 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.
- 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 N cycles, the proposed method stops intermediately in the third cycle, as illustrated in Fig. 6.

If in the firt three cycles of the decoding process, the evaluation of the XOR matrix for all $\{B_J\}$ is "0," the codeword is determined to be error-free and forwarded directly to the output. If the $\{B_J\}$ 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 in Fig. 7. The figure shows the basic ML decoder with an -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 shown in Fig. 2. 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 y.

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 $\{B_J\}$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.

The schematic for this memory system (see Fig. 9) is very similar to the one in Fig. 1, adding the control logic in the MLDD module.

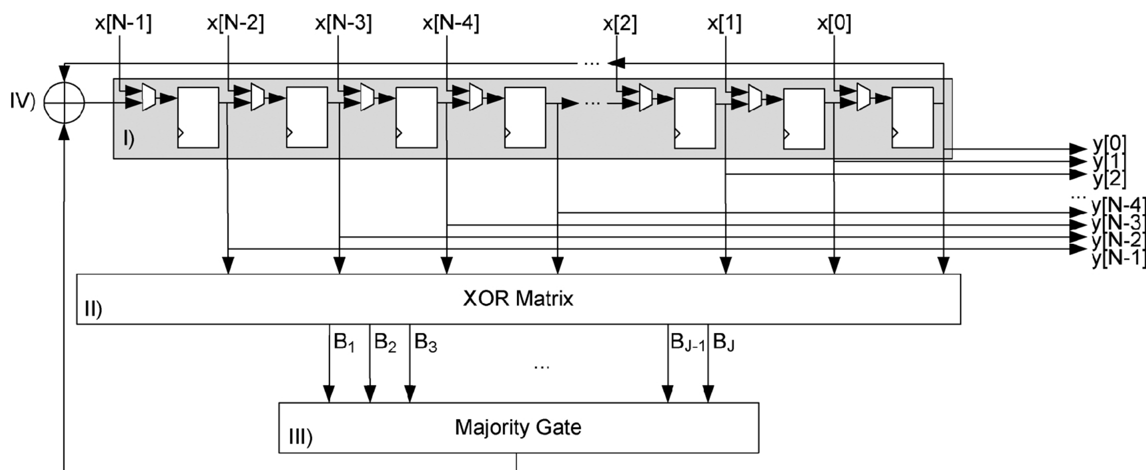


Fig. 2. Schematic of an ML decoder. I) cyclic shift register. II) XOR matrix. III) Majority gate. IV) XOR for correction.

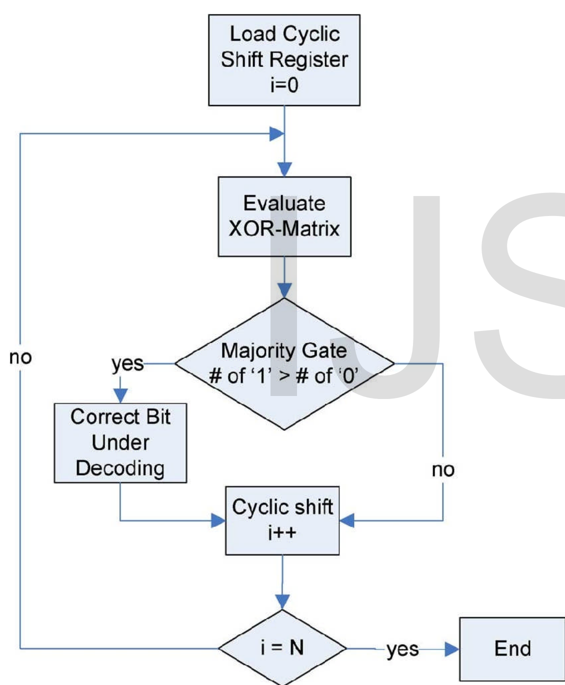


Fig. 3. Flowchart of the ML algorithm.

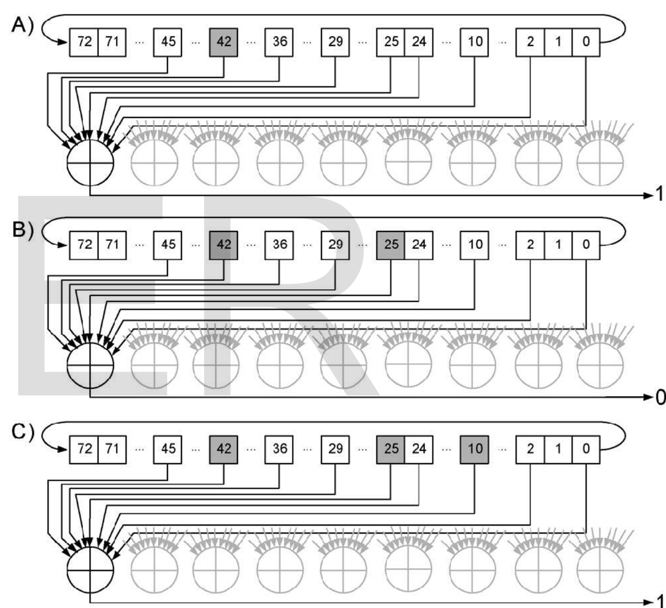


Fig. 5. Single check equation of a β 3 ‡3 ML decoder. (a) One bit-flip. (b) Two bit-flips. (c) Three bit-flips.

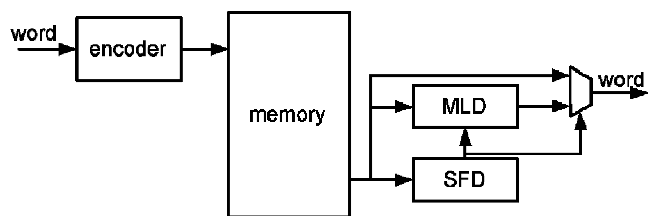


Fig. 4. Memory system schematic of an ML decoder with SFD.

IJSER

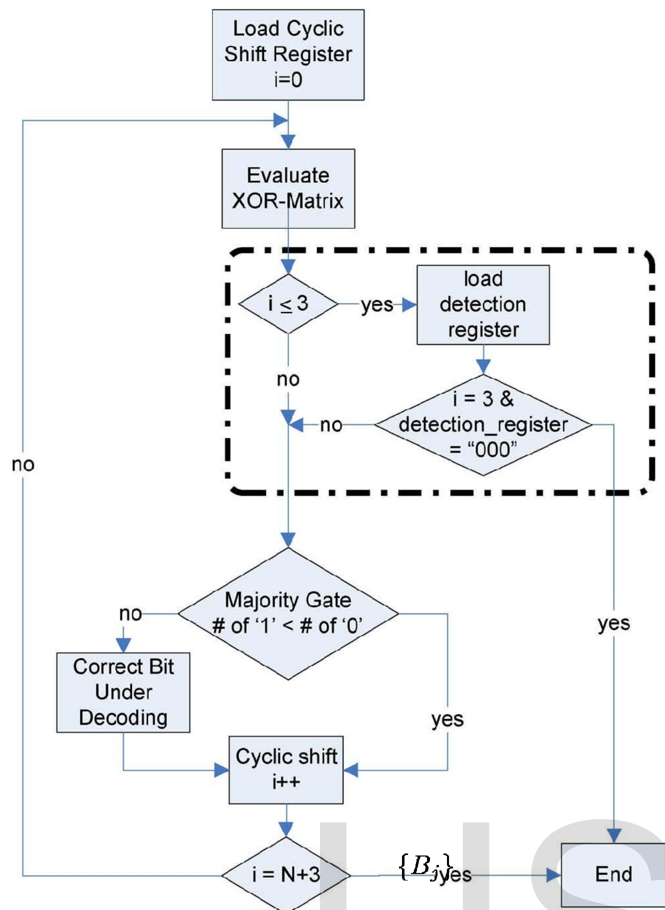


Fig. 6. Flow diagram of the MLDD algorithm.

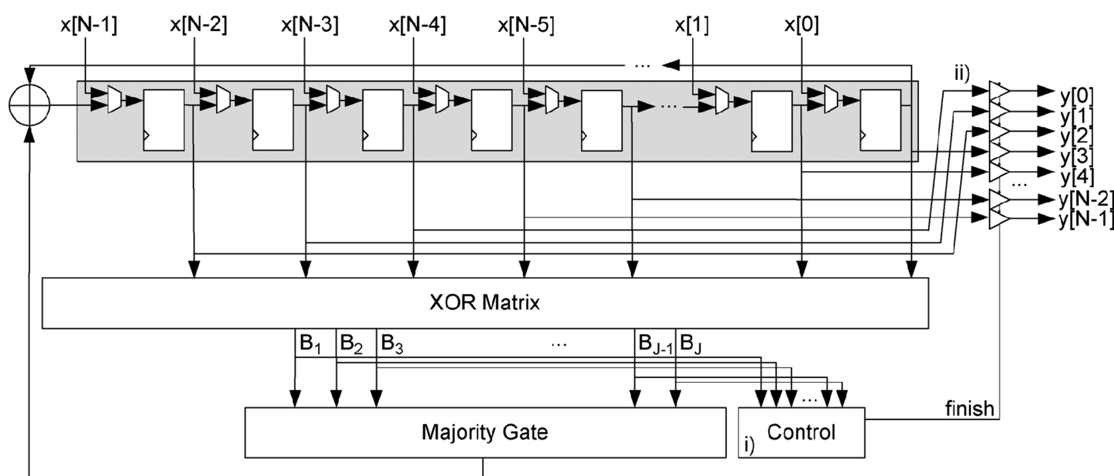


Fig. 7. Schematic of the proposed MLDD. i) Control unit. ii) Output tristate buffers.

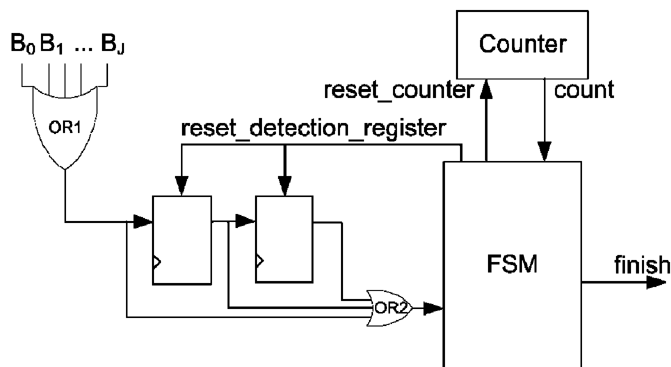


Fig. 8. Schematic of the control unit.

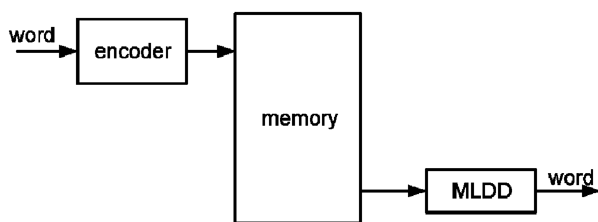


Fig. 9. Memory system schematic of an MLDD.

4. SIMULATION RESULTS AND DISCUSSION

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

A. *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 N, the number of data and parity bits are stated.

Given a size N, 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 is given by

$$\binom{N}{M} = \frac{N!}{M!(N-M)!} \quad (1)$$

where M is the number of bit-flips.

The number of combinations can be seen in Table II for different values of N with double and quadruple errors. As expected, increasing the code length implies an exponential growth of the number of combinations, and therefore, of the computational time.

All combinations in Table II have been simulated, and the results can be seen in Tables III and IV.

TABLE I
DSCC LENGTHS

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

TABLE II
NUMBER OF COMBINATIONS FOR TWO AND FOUR BIT-FLIPS WITH DIFFERENT CODE LENGTHS

M	N	# of combinations
2	73	2,628
	273	37,128
	1057	559,153
4	73	1,088,430
	273	226,387,980
	1057	51,911,764,520

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

TABLE III
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 IV
EXHAUSTIVE SEARCH RESULTS FOR FOUR BIT-FLIPS

iteration	N=73	N=273	N=1057*
1	97.35%	99.12%	99.98%
2	99.92%	99.99%	99.99%
3	100.00%	100.00%	100.00%

*given values are extrapolations

The results for the case of four bit-flips are documented in Table IV.

The percentage of errors that can be detected with just one iteration has increased respect to the results presented in Table III (two bit-flips). This increase is quite large and can be explained by the higher amount of bit-flips which are not participating in the same check sum equation. The second cycle of the MLDD is already providing a percentage of detection very close to 100%. Again, with the third cycle the MLDD is capable of finding any error pattern. The case of $N=1057$ is marked with an asterisk because the results are extrapolations of the first one billion combinations. No exhaustive results have been calculated in this case due to the huge amount of combinations needed.

Up to this point, experiments have been oriented to prove that all situations with four bit-flips or less can be detected. Since having five bit-flips is not a problem (because an odd number of errors are always correctable), the issue would be with six or

more bit-flips. The next experiments have been conducted to explore this scenario.

From the presented codes, the $N = 73$ code is capable of correcting up to four errors. So, studying what happens with a higher number of errors is not very important. For *N=273* and 1057, the codes can correct up to 8 and 16 errors respectively. Therefore, additional injection tests have been carried out with one billion random bit-flip patterns for the case of six bit-flips and eight bit-flips.

The results show that for N = 273 code there are 28 cases six-bit errors and one case of eight-bit errors that were not detected. For N = 1057 all cases were detected. In the light of these results, the simulations for N=1057 were extended to 5 billion cases in an attempt to find errors that were not detected. The results show no undetected errors. This means that for er-rors affecting more than five bits, there is a very small probability of the error not being detected in the first three iterations. However, the values are so small that they would be acceptable in many applications. Anyway, it is important to notice that, in most real scenarios, erroneous words in a memory will usually suffer a limited number of bit-flips, and cases with M>5 are not frequent.

As a complement to the presented experiments, a theoretical proof for the case of double errors is presented in the final Appendix.

TABLE V
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

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 is given in Table V.

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 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). This result has the same order of magnitude as the SFD, since both are independent of , *N* and, therefore, it would be feasible for very large size codes. However, the advantage of our technique with respect to SFD is, as it will be explained later, a more reduced area and power requirement.

.

In the case that an error is detected, all of the techniques need to launch the whole decoding process. For MLD and SFD, this represents N+2 cycles (decoding cycles plus two I/O cycles). For MLDD, the situation is the same, but, instead of N+2 cycles, three extra cycles are needed (for a total of N+5). These three extra cycles have been added to the process in order to simplify the multiplexing logic of the design (see Section IV-C for more details). It represents a negligible impact on performance, but it provides significant savings in area.

In TableVI ,a comparison of the MLD and MLDD techniques is provided for several values of *N* . Although this is only a best-case scenario, because it is assumed that all words come without errors, it gives the idea of how much speed-up can be obtained in an ideal situation.

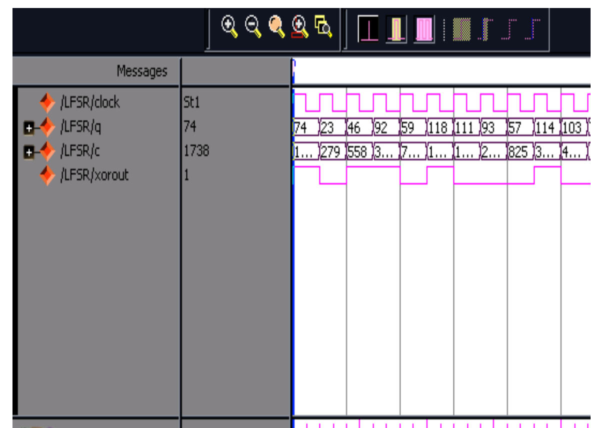
TABLE VI
SPEED-UP OF THE PROPOSED MLDD FOR ERROR FREE CODE WORDS

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

TABLE VII
SYNTHESIS RESULTS OF THE 3 DESIGNS FOR DIFFERENT CODE LENGTH

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%

SIMULATION RESULT



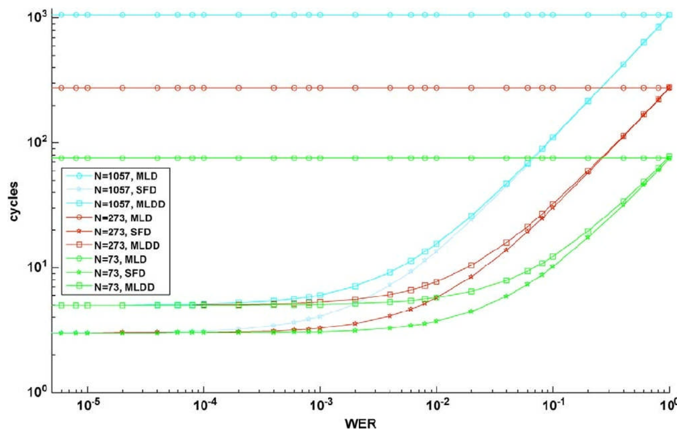


Fig. 10. Performance of the different code lengths versus the WER.

In a real situation, a fraction of the words would have bit-flips. This fraction is represented by the WER. Since MLDD needs five cycles to handle correct words N+5 and for erroneous words, the average performance would be

$$\text{MLDD_performance} = (1-\text{WER})\cdot 5+\text{WER}\cdot(\text{N}+5). \quad (2)$$

Using this expression, the performance of the three techniques has been studied for different values of the WER. These results can be seen in Fig. 10 for different values of N.

The first comment on these results is that the MLD technique has the worst performance, whose value is independent of the WER (i.e., it needs the same number of cycles to handle correct and erroneous data).

Another comment is that the SFD version has the best performance, as expected. But our proposed technique (MLDD) is very similar in this aspect, since both values are very close. This small performance difference is compensated for with the area savings that MLDD provides. This difference is even smaller for large values of N and WER.

C. 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 of N, using a TSMC35 library. The obtained results are depicted, in number of equivalent gates, in Table VII.

Apart from the designs used in the previous experiments ($N = 73, 273,$ And1057), the case of $N = 21$ has also been Synthesized in order to provide more area information.

The conclusions 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 than N does.
- The MLDD version has a very similar performance to SFD, however it requires a much lower area overhead, ranging from 10.16% to 0.43%.

These conclusions can be extrapolated to power. The overhead introduced by MLDD is very small, contrary to the SFD case.

An important final comment is that the area overhead of the MLDD actually *decreases* with N 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 N. 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 N 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 N 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 N N 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.

That is why the MLDD technique needs N+5 cycles to detect and correct an erroneous word, instead of N+2 cycles (see Table V).

IJSER

5. 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 traditional 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.

Appendix

The basis of the presented paper is the hypothesis that all er-rors affecting up to five bits in a block protected by DSCC can be detected in just three decoding cycles. This hypothesis, which has been used to propose a more efficient design of MLD de-coding, may seem somewhat abrupt in the way it has been in-troduced in the paper. Initially, the authors' goal was to propose a quick way to perform scrubbing in protected data blocks. The idea was to handle a reduced number of parity check equations in each scrubbing cycle, tolerating a certain number of errors which would be corrected in the next cycle. Unexpectedly, the results offered the property that, in only three decoding cycles, all errors affecting up to five bits in the block were detected, which motivated the idea of this paper.

Although the mentioned property has been verified in Section IV through an experimental procedure, a mathematical demonstration for the case of double errors will be offered at the end of this appendix. Before this, and in order to understand the demonstration, some basic information about the DSCC codes will be provided.

A. DSCCs

DSCCs are one-step ML decodable codes with high error-cor-rection capability and are linear cyclic block codes

1) Perfect Difference Set: DSCCs work on the difference-set concept for which a brief description follows. Given a set P and a difference of the elements D, we have

$$P = \{l_0, l_1, \cdots, q\} \, (0 \leq l_1 < l_2 \cdots < l_q \leq (q+1)) \quad (3)$$
$$D = \{l_i - l_j : i \neq j\}. \quad (4)$$

The perfect difference set P must satisfy the three following conditions.

1) All positive differences in D are distinct.
2) All negative differences in D are distinct.
3) If $l_i - l_j$ is a negative difference in D, then $q(q+1) + 1 + (l_i - l_j)$ is not equal to any positive difference in D.

2) DSCC Construction: For a binary code, the perfect difference-set is constructed using the relationship

$$q = 2^s : s \in N. \quad (5)$$

Using the set elements as powers in the terms of the polynomial $z(X)$

$$z(X) = 1 + X^{l_1} + X^{l_2} + \cdots + X^{l_q} \quad (6)$$

and the syndrome polynomial $h(X)$ for the difference-set, the cyclic code is given by the greatest common divisor of $z(X)$ And $X^N + 1$

$$h(X) = GCD\{z(X), X^N - 1\}$$
$$= 1 + h_1 X + h_2 X^2 + \cdots + h_{k-1} X^{k-1} + X^k. \quad (7)$$

Finally, the DSCC code is generated by

$$g(X) = \frac{X^N - 1}{h(X)}$$
$$= 1 + g_1 X + g_2 X^2 + \cdots + X^{N-k}. \quad (8)$$

3) DSCC Parameters: Besides from the definitions and equations previously explained, the following parameters completely define the DSCC codes:

- Code length: $N = 2^{2s} + 2^s + 1$.
- Message bits: $k = 2^{2s} + 2^s - 3^s$.
- Parity-check bits: $(N - k) = 3^s + 1$.
- Minimum distance: $d = 2^s + 2$.

As $\{0 \leq l_1 \leq l_2 \leq \ldots \leq l_q \leq q(q+1)\}$ is a perfect difference-set, not two polynomials $w_i(X)$ and $w_j(X)$, given by (9) (see [6]), can have any common term except X^{n-1}, for $i \neq j$:

$$w_i(X) = X^{l_i - l_{i-1} - 1} + X^{l_i - l_{i-2} - 1} + \cdots + X^{l_i - l_1 - 1} + X^{l_i - 1}$$
$$+ X^{N-1-l_{2^s}+l_i} + X^{N-1-l_{2^s-1}+l_i} + \cdots + X^{N-1}. \quad (9)$$

Thus, $w_0(X), w_1(X), \ldots, w_2^s(X)$ form a set of $J = 2^s + 1$ polynomials orthogonal on the bit at position X^{N-1}. This implies that there will be $J = 2^s + 1$ parity check-sums able to correct up to $t_{ML} = 2^{(s-1)}$ errors.

B. Detection Method Theoretical Proof for Double Errors

In a DSCC-LDPC code, the check equations used in majority logic decoding are cyclically shifted versions of a vector wi(X). The shift distances are also specified by the perfect difference set of the code.

Lemma 1: Given condition 1 and 2 in A.1, the DSCC-LDPC codes are such that there are no three consecutive values in the difference set.

Proof: Let us suppose that there are three consecutive differences in the set $l_1, l_2 = l_1 + 1, l_3 = l_2 + 1$. Then, the differences $l_2 - l_1$ And $l_3 - l_2$ would be equal to one and therefore the condition 1 or 2 in Section A1 would not be true.

Lemma 2: Given vector $w_i(X)$ in (9), if a pair of 1's in it is separated by a distance k then there cannot be any more pairs of 1's in the same vector at a distance k or $N - k$

Proof: From (9), the 1's in $w_i(X)$ occur at positions of the form $l_i - l_j - 1$ or $N - 1 - l_{j'} + l_i$. Therefore, distances between pairs of 1's in $w_i(X)$ belong to the difference set D, which are unique per condition 3 in Section A1.

Theorem: Given a block protected with DSCC codes and affected by two bit-flips, these can be detected in only three decoding cycles.

Proof: When decoding a block with two errors at a distance d, two situations may occur, given here.

1) If both errors occur in different equations, they are detected in the first iteration.
2) If both errors occur in the same equation that corresponds to difference l_1, then a necessary condition in the second iteration for the error not to be detected is that there is a difference $l_2 = l_1 + 1$ in the perfect difference set, as bits are shifted by one. Otherwise, there would be an equation with two 1's at a distance d corresponding to a difference $l' \neq l_1 + 1$ and another corresponding to the position of the bits with error. This would violate Lemma 2. The same applies to the third iteration, but this time, with Lemma 1, such a difference does not exist, and, therefore, the error will be detected.

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 semiconductor 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] M. A. Bajura *et al.*, "Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 935–945, Aug. 2007.

[5] R. Naseer and J. Draper, "DEC ECC design to improve memory reliability in sub-100 nm technologies," in *Proc. IEEE ICECS*, 2008, pp. 586–589.

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

[7] 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.

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

[9] 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.

[10] B. Vasic and S. K. Chilappagari, "An information theoretical framework 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.

[11] H. Naeimi and A. DeHon, "Fault secure encoder and decoder for NanoMemory applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 473–486, Apr. 2009.

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] F. J. MacWilliams, "A table of primitive binary idempotents of odd length ▴, ↨ 3 . 3 3," *IEEE Trans. Inf. Theory*, vol. IT-25, no. 1, pp. 118–123, Jan. 1979.

IJSER