Microelectronics Reliability 128 (2022) 114444

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

New decoding techniques for modified product code used in critical applications

David C.C. Freitas [a,*], César Marcon [b], Jarbas A.N. Silveira [a], Lirida A.B. Naviner [c], João C. M. Mota [a]

[a] Federal University of Ceará, Fortaleza, CE, Brazil
[b] Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, RS, Brazil
[c] LTCI, Télécom Paris, Institut Polytechnique de Paris, 91128 Palaiseau, France

ARTICLE INFO

Keywords:
Error Correction Code (ECC)
Fault tolerance
Radiation effect
ECC algorithms

ABSTRACT

The shrinking of memory devices increased the probability of system failures due to the higher sensitivity to electromagnetic radiation. Critical memory systems employ fault-tolerant techniques like Error Correction Code (ECC) to mitigate these failures. This work explores error correction techniques and algorithms employing the Line Product Code (LPC), a product-like ECC. We propose to decode LPC codewords using a single error correction algorithm (AlgSE) followed by a double error correction algorithm (AlgDE). Both algorithms explore the LPC characteristics to attain greater decoding efficiency. AlgSE is implemented with an iterative technique associated with a correction heuristic, while AlgDE is an innovative proposal that allows increasing correction effectiveness through the inference of errors. AlgDE allows increasing the efficiency of the LPC decoder significantly when used together with AlgSE. It corrects 100% of the cases up to three bitflips as well as 98% and 92%, respectively, for four and five upsets in exhaustive tests. Besides, we present tradeoffs concerning the error correction potential versus the costs of implementing the correction algorithms.

1. Introduction

The continued miniaturization of electronic components allows adding many more features within the same Integrated Circuit (IC), providing higher performance and energy efficiency. As a result, the functional and non-functional requirements of several applications were implemented entirely within the same IC, known as System-on-Chip (SoC) [1]. These applications include critical safety systems, where a failure can lead to significant economic losses or damage to people or the environment [2]. The scaling down of the electronic components implies physical changes that make them more susceptible to failures due to electromagnetic radiation [3]. These failures studied almost 60 years ago [4–6] occur when high-energy alpha particles or neutrons collide with a device, changing the content of one or more cells permanently or transiently [7–10].

This work focuses on random transient errors, usually caused by Single Event Effects (SEEs), such as (i) Single Event Transient (SET), which changes the current or voltage of a combinational node; (ii) Single Event Upset (SEU), which changes the value of a memory cell; (iii) Multiple Cell Upsets (MCU), which changes the values of more than one memory cell. Multiple Bit Upsets (MBU) is an MCU, in which the affected memory cells belong to the same logical word; and (iv) SEFI (Single Event Function Interrupt) that produces an error in a unit affecting other components; e.g., an error that occurs in a memory control unit and affects several words in memory or even the entire system [11,12].

Several techniques mitigate failures in space applications, such as Hardened Memory Cell, Triple Modular Redundancy (TMR), and Error Correction Code (ECC). In hardened memory cells, parts of the original circuit are replaced by their hardened versions that are less susceptible to failure [7]. TMR uses three copies of the same information and uses a voter to decide the correct result [13,14]. In turn, ECCs use encoding and decoding algorithms to restore the correct data value, protecting data from faults occurring in memory cells or transmission channels [7]. Nowadays, one-dimensional ECCs fail to achieve the effectiveness needed to address the increasing number of bitflips caused by a single radiation event [15–18]. Consequently, n-dimensional ECCs have been

* Corresponding author.
E-mail addresses: davidciarlinifreitas@gmail.com (D.C.C. Freitas), cesar.marcon@pucrs.br (C. Marcon), lirida.naviner@telecom-paris.fr (L.A.B. Naviner), mota@gtel.ufc.br (J.C.M. Mota).

https://doi.org/10.1016/j.microrel.2021.114444
Received 5 January 2021; Received in revised form 25 October 2021; Accepted 1 December 2021
Available online 13 December 2021

proposed to provide higher error detection and correction power [16]. Gracia-Morán et al. [18] explain that these complex ECCs, build for use in critical applications, increase error correction and detection capacity but implying higher redundancy, area usage, energy consumption, and critical path delay [16]. This work focuses on Two-Dimensional (2D) ECCs, also called product codes, designed to protect memories used in critical applications like the space ones [18–24].

Section 2 presents a brief history of product codes, the state-of-the-art, and the product ECCs focusing on fault-tolerant memories used on space applications. Section 3 describes the capabilities of the Line Product Code (LPC), exploring its decoding algorithms and error correction techniques in detail. Section 4 shows the methodology adopted in this work to examine the potential of the proposed decoding algorithms and techniques. Sections 5 to 7 describe, respectively, the algorithm employed for Single Error (SE) correction, the correction technique with row-column correlation, and the Double Error (DE) correction method. Finally, Section 8 describes and discusses the experimental results used to validate the proposed algorithms and techniques.

2. Related work

Elias [25] introduced in 1954 the pioneering work in the product code area, proposing a simple and efficient way to build long codes based on smaller ones. Later, several authors proposed improving the product code performance, changing the encoder, decoder, or code structure. The algorithms presented in this work employ the iterative decoding of product codes, originally proposed in [25]. Alexey, Victor, and Eygene [26] report that the iterative decoder through rows and columns is the most used algorithm for product codes, as it has low complexity and can correct various error patterns. The iterative algorithm decodes all columns in the matrix correcting errors with the ECC placed in each column; next, the algorithm makes the same on the matrix row. The algorithm is called iterative because it repeats these correction steps several times. Changuel, Bidan, and Pyndiah [27] describe the standard iterative decoder, which repeats the number of iterations in rows and columns until it cannot fix any more errors. The authors claim that this iterative decoder is effective, correcting error patterns with more bits than half the Hamming distance.

Li, Miao, and Wang [28] proposed an encoding scheme using Turbo Product Code (TPC) for reducing transmission latency and improving the bit error rate. Zhou and Li [29] proposed a parallel TPC decoder based on GPU that simultaneously decodes all rows and columns of the two-dimensional matrix of the product code. The experimental results show that the decoding latency is significantly reduced compared to the TPC decoder based on a conventional CPU. Lopacinski et al. [30] proposed a new concept that uses vertical and horizontal Bose-Chaudhuri-Hocquenghem (BCH) codewords to increase the TPC decoder effectiveness. Swaminathan, Madhukumar, and Guohua [31] presented new algorithms for estimating two-dimensional parameters of the product codes BCH and Reed-Solomon on noisy channel conditions. The experimental results showed that the proposed code is more effective than BCH with an area overhead. Senger [32] described an iterative decoding algorithm for product codes based on Reed-Solomon, which performs better than the classic method without adding significant complexity. Guo et al. [33] proposed the Decimal Matrix Code (DMC) that detects and corrects errors by adding and subtracting decimal integers. The DMC matrix structure is divided into eight four-bit symbols checked throughout vertical parity bits and horizontal check bits, containing the decimal result.

Khittiwitchayakul, Phakphisut, and Supnithi [34] introduced two Weighted Bit-Flipping (WBF) decoding algorithms for the Low-Density Parity-Check (LDPC) product code; the authors use the Page-WBF and Row-Column-WBF algorithms to evaluate tradeoffs in error correction capacity and implementation complexity. Jeong and Lee [35] [36] proposed a product code using LDPC to improve the performance of a magnetic storage system, reaching effectiveness for random and burst error scenarios. Erozan and Çavuş [15] proposed 2D ECC based on Euclidian Geometry-LDPC and single parity check bits to reach high error correction levels. Arslan et al. [37] introduced a product code for transferring data from specific storage media. ECCs are placed in the headers of these transfers, allowing errors to be detected or corrected.

Sheikh, Amat, and Liva [38] proposed the iterative Bounded Distance Decoding with scalable reliability (iBDD-SR) algorithm for product codes to improve performance with low complexity increase; experiments using LDPC and BCH show the efficacy of the proposed algorithm for high-transfer rate applications. Li, Lin, and Wang [39] proposed the Soft-Assisted-iBDD to improve the iBDD algorithm; they presented a voting strategy to assess whether a codeword should be corrected. This proposal enhances decoding performance without significantly increasing complexity and requiring extra memory. Coskun, Jerkovits, and Liva [40] investigated the Successive Cancellation List decoding of product codes with single parity-check and extended Hamming for product codes used on wireless communication systems.

Sheikh et al. [41] proposed to improve the iBDD algorithm with the iterative Generalized Minimum Distance (iGMD) with scaled reliability decoding algorithm, which is more effective than its predecessor with a small complexity increase. To increase the decoding efficiency, Sheikh, Amat, and Liva [42] propose Binary Message Passing-GMD Decoding (BMP-GMDD), a new algorithm for decoding product codes based on iGMD that uses the Hamming distance metric in the last stage.

Liu et al. [43] introduced a technique to improve the reliability of product code encoders and decoders that detects errors based on the parity forecast. Li et al. [44] described a method for reducing the number of parity bits in a product code. The method also reduces the MCU memory probability due to fewer redundancy cells. Tawfeek, Mahran, and Abdel-Hamid [45] proposed a new ending criterion in the iterative decoding of product codes based on the reliability limit; the purpose of this criterion is to prevent the iterative nature of decoding from increasing computational latency and complexity. Condo et al. [46] proposed a decoder for polar code in a matrix format that uses two steps to reduce the decoding latency. Yang et al. [47] used a product code in NAND Flash to avoid retention errors and program interference errors. The proposal uses a linear block code for parity check on rows and columns.

Some works on product codes focus on correcting errors in space application memories [19–24]. In 2007, Argyrides et al. [48] published the Matrix code, a 2D ECC with 16 data bits and 16 redundancy bits for correcting bitflips in space applications with Hamming check bits in rows and parity bits in columns. Four years later, the same authors [49] improved Matrix to a 32-bit code, obtaining about 100 citations in both works and making Matrix ECC a reference in this field. Several other authors based on [48,49] develop advances in the area, as Freitas et al. [19], that introduced the Product Code for Space Applications (PCoSA), which applies extended Hamming in rows and columns. PCoSA corrects any three bitflips and achieves high correction rates for up to five upsets. Silva et al. [20] proposed the Matrix Region Section Code (MRSC) for detecting and correcting MBUs in volatile memories with a low implementation cost. These same authors [21] extended the basic MRSC to a 32-bit data format, employing a technique to reduce the number of redundancy bits with high reliability and low implementation cost. Castro et al. [22] introduced the Column-line-Code (CLC), a product code with a high MCU-correction rate, considering the costs of area, energy, and delays. Silva et al. [23] extended the CLC to a format supporting an additional correction operation called the CLC-extended, reaching high correction rates with more complex MCU patterns. Gracia-Morán et al. [18] discussed the tradeoffs between the encoding/decoding costs and correction capacity, displaying low redundancy ECCs, capable of correcting MCUs with reduced delay and area and power consumptions. Finally, Magalhães, Alcântara, and Silveira [24] introduced the Parity Hamming Interleaved Correction Code (PHICC), an ECC product that achieves a fair correction rate with a minimal

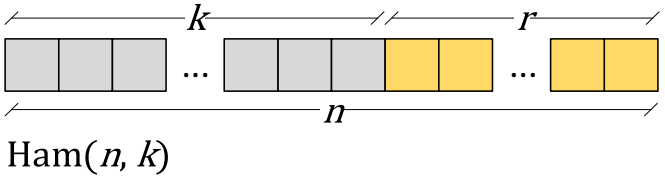

Fig. 1. Representation of a generic Hamming code Ham(n, k); k and r are the numbers of data and redundancy bits, respectively.

$$n = r + k \tag{1}$$

$$r = log_2(n+1) \tag{2}$$

$$k = 2^r - r - 1 \tag{3}$$

additional implementation cost.

As the miniaturization of electronic devices is a determining factor for increasing memory bitflips during space applications, this work proposes to increase the error correction efficacy of a 16-bit data ECC using the same structure and base codes already known. We introduce LPC, a modified product code with a mixed decoding approach, composed of an iterative simple error correction algorithm (AlgSE), followed by a DE correction algorithm (AlgDE). The main originality of AlgSE is to carry out a preliminary analysis of the number of SEs in the rows and columns to apply correction heuristics. AlgDE is an original proposal for increasing the error correction efficacy through error inference.

3. Theoretical foundations of Hamming code and line product code (LPC)

Fig. 1 generalizes the linear code structure proposed by R. Hamming [50], which is referenced by Ham(n, k). Eqs. (1), (2), and (3) describe the relations among n, r, and k, the numbers of bits of the codeword, data, and redundancy, respectively.

The Minimum Hamming Distance (*MD*) is the smallest number of bit changes required to pass from any codeword to any other codeword. *MD* is a metric used to measure the code capacity in correcting and detecting errors. Eqs. (4) and (5) calculate the maximum number of errors *in any codeword position* that a Hamming-based code can correct *EC or* detect *EC* [51], respectively.

$$EC = \lfloor \frac{MD-1}{2} \rfloor \tag{4}$$

$$ED = MD - 1 \tag{5}$$

Eqs. (4) and (5) are exclusives, i.e., *EC or ED*, but not *EC and ED* simultaneously. The simultaneity relationship among *EC*, *ED*, and *MD* is given by Eq. (6) (further details in [52]).

$$ED = MD - EC - 1 \tag{6}$$

The basic Hamming code has *MD*=3, so it can correct a SE (i.e., *EC*=1) or detect an error caused by a double bitflip without knowing whether this error has a SE or DE source. The extended Hamming adds a parity bit into the basic Hamming code, increasing *MD* to 4. Eqs. (4) and (6) show that extended Hamming can correct a SE and detect a DE simultaneously, i.e., an SEC-DED code [53].

A product code is the combination of two linear codes $C_1(n_1, k_1)$ and $C_2(n_2, k_2)$, which is denoted by C_1C_2 [53]. A product code applies two sets of check bits on the same data field; additionally, it employs check on check bits, increasing the correction and detection capacity [53]. However, this large code also implies a significant increase in the Redundancy Rate (RR), i.e., the relation between the number of redundancy bits and the total number of codeword bits.

LPC is a *modified product code* whose *MD* depends on the composition of the *MDs* of each linear code C_1 and C_2, as described by Eq. (7). LPC was designed to enforce fault-tolerance on data bits only, as check and parity can be recalculated from the data bits; thus, this code does not implement checks on check bits, reaching a high correction rate while reducing the RR.

$$MD_{C_1C_2} = MD_{C_1} + MD_{C_2} - 1 \tag{7}$$

LPC has the same linearity as the product code; thus, the LPC encoding can respect the format C_1, followed by C_2, or vice versa [51,54]. Fig. 2 illustrates the LPC generic structure composed by two symmetric Ham(8, 4) codes; i.e., C_1=C_2, and n_1=n_2=8, k_1=k_2=4 and r_1=r_2=4. Additionally, LPC adopts even parity so that the total of 1 s in the codeword is even, including the additional parity bit [55].

Eq. (7) shows that the LPC has $MD = 7$ since it implements Ham(8, 4) as C_1 and C_2 that have $MD = 4$. The logical reason for *MD*=7 lies in the LPC organization implying that rows and columns always cross in one, and only one, data bit; a bitflip in one data bit implies varying three other bits in the column, and three other bits in the row since rows and columns use Ham(8, 4), which has $MD = 4$. Additionally, Eqs. (4) and (6) show that LPC can correct 3 errors ($EC = 3$) and detect 3 errors ($ED = 3$). Besides, LPC can correct even more errors, depending on their location.

Fig. 3 shows LPC in a (48, 16) format - 48 bits encode 16 data bits (*D*), 12 row-check bits (*Cr*), 4 row-parity bits (*Pr*) 12 column-check bits (*Cc*), and 4 column-parity bits (*Pc*).

Let q be a bit position index and $\oplus$ be an XOR operation, then Eqs. (8) to (10) and (11) to (13) compute the recalculated check bits of rows ($\widehat{Cr}$) and columns ($\widehat{Cc}$), respectively. Additionally, Eqs. (14) and (15) compute the recalculated parity bits of rows ($\widehat{Pr}$) and columns ($\widehat{Pc}$), respectively.

$$\widehat{C}r_{q,0} = D_{q,1} \oplus D_{q,2} \oplus D_{q,3} \quad \forall 0 \le q \le 3 \tag{8}$$

$$\widehat{C}r_{q,1} = D_{q,0} \oplus D_{q,2} \oplus D_{q,3} \quad \forall 0 \le q \le 3 \tag{9}$$

$$\widehat{C}r_{q,2} = D_{q,0} \oplus D_{q,1} \oplus D_{q,3} \quad \forall 0 \le q \le 3 \tag{10}$$

$$\widehat{C}c_{0,q} = D_{1,q} \oplus D_{2,q} \oplus D_{3,q} \quad \forall 0 \le q \le 3 \tag{11}$$

$$\widehat{C}c_{1,q} = D_{0,q} \oplus D_{2,q} \oplus D_{3,q} \quad \forall 0 \le q \le 3 \tag{12}$$

$$\widehat{C}c_{2,q} = D_{0,q} \oplus D_{1,q} \oplus D_{3,q} \quad \forall 0 \le q \le 3 \tag{13}$$

$$\widehat{P}r_q = D_{q,0} \oplus D_{q,1} \oplus D_{q,2} \oplus D_{q,3} \oplus Cr_{q,0} \oplus Cr_{q,1} \oplus Cr_{q,2} \quad \forall 0 \le q \le 3 \tag{14}$$

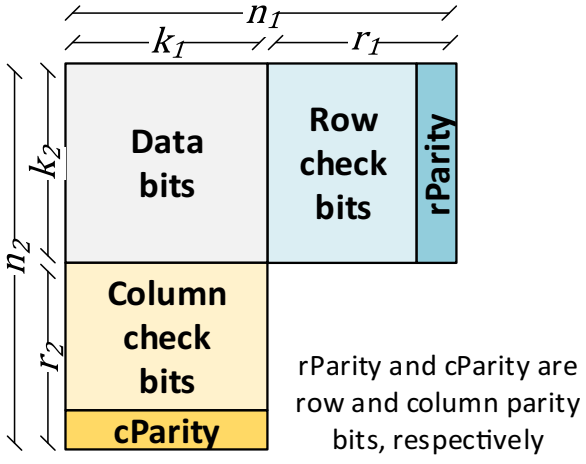

Fig. 2. Generic representation of the LPC structure.

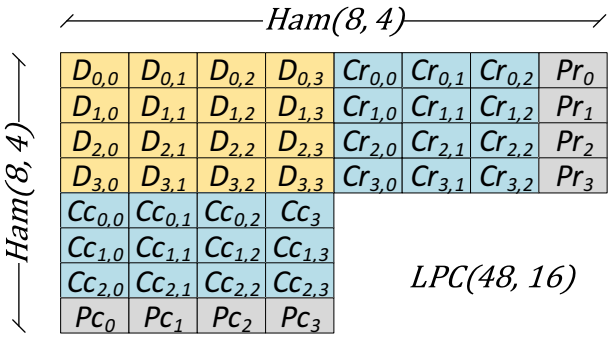

Fig. 3. LPC(48, 16) structure encompassing five regions: data (*D*), row-check (*Cr*), column-check (*Cc*), row-parity (*Pr*), and column-parity (*Pc*).

$$\widehat{P}c_q = D_{0,q} \oplus D_{1,q} \oplus D_{2,q} \oplus D_{3,q} \oplus Cc_{0,q} \oplus Cc_{1,q} \oplus Cc_{2,q} \quad \forall 0 \le q \le 3 \quad (15)$$

Eqs. (16) to (19) compute the syndromes of each row and column, which are used to verify if the check bits calculated during the LPC encoding are equal to the ones recalculated by the LPC decoding; thus, enabling us to detect bitflips.

$$sCr_{q,k} = Cr_{q,k} \oplus \widehat{C}r_{q,k} \quad \begin{matrix}\forall 0 \le q \le 3 \\ \forall 0 \le k \le 2\end{matrix} \quad (16)$$

$$sPr_q = Pr_q \oplus \widehat{P}r_q \quad \forall 0 \le q \le 3 \quad (17)$$

$$sCc_{k,q} = Cc_{k,q} \oplus \widehat{C}c_{k,q} \quad \begin{matrix}\forall 0 \le q \le 3 \\ \forall 0 \le k \le 2\end{matrix} \quad (18)$$

$$sPc_q = Pc_q \oplus \widehat{P}c_q \quad \forall 0 \le q \le 3 \quad (19)$$

Eqs. (20) and (21) describe the computations of sCR_q and sCC_q, which are achieved by applying a logical OR in the check-bit syndromes of the rows and columns, respectively.

$$sCr_q = sCr_{q,0} \text{ OR } sCr_{q,1} \text{ OR } sCr_{q,2} \quad \forall 0 \le q \le 3 \quad (20)$$

$$sCc_q = sCc_{0,q} \text{ OR } sCc_{1,q} \text{ OR } sCc_{2,q} \quad \forall 0 \le q \le 3 \quad (21)$$

Table 1 shows that sCr_q and sCc_q, together with sPr_q and sPc_q, are used to analyze whether the decoded data contains errors and the type of error that has been detected. For each row and column q, a SE is represented by SEr_q and SEc_q, respectively; similarly, a DE is denoted by DEr_q and DEc_q.

Once an SE is detected, the row and column error positions are obtained by combining the weights of the check-bit syndromes. Eqs. (22) and (23) describe the error addresses in the row and column q, respectively.

$$EAr_q = 4 \times sCr_{q,0} + 2 \times sCr_{q,1} + sCr_{q,2} \quad \forall 0 \le q \le 3 \quad (22)$$

$$EAc_q = 4 \times sCc_{0,q} + 2 \times sCc_{1,q} + sCc_{2,q} \quad \forall 0 \le q \le 3 \quad (23)$$

4. Applied methodology

Fig. 4 illustrates the main aspects of the work methodology to explore the error correction effectiveness achieved by LPC decoding

Table 1
Meaning of the combinations of the syndrome bits.

sC	*sP*	Error detection
0	**0**	None – or a possible quadruple error
0	**1**	Parity bit – or a possible triple error
1	**0**	Even error – a possible Double Error (DE)
1	**1**	Odd error – a possible Single Error (SE)

(sC, sP) are the tuples (sCr_q, sPr_q) or (sCc_q, sPc_q) $\forall 0 \le q \le 3$.

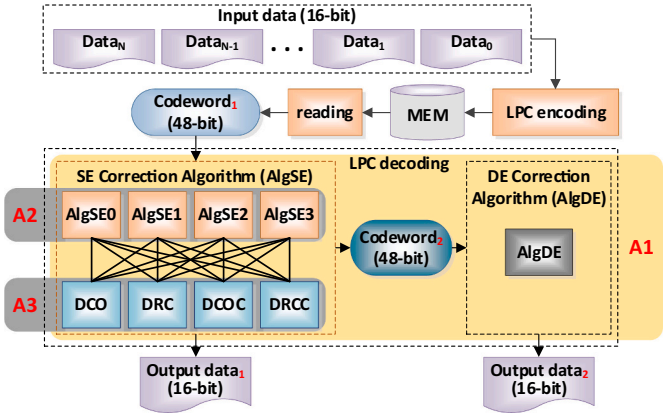

Fig. 4. Methodology applied to LPC decoding.

algorithms. The research axis *A1* proposes to perform the LPC decoding with two successive algorithms, an algorithm that corrects SEs (AlgSE) and another one that corrects DEs (AlgDE); the structure of the modified product code results in lines and columns crossing, which allows using DE detection information to infer errors occurring in these crossings, increasing the correction power.

This work employs 16-bit data encoded with LPC and stored in a memory in codewords of 48-bit format. Each data reading implies using an LPC decoding process over $codeword_1$. Initially, AlgSE decodes $codeword_1$, which was extracted from a memory containing LPC-encoded data. AlgSE corrects the SEs of $codeword_1$, generating the $output\ data_1$ and the $codeword_2$, which is an input to AlgDE that uses the error address provided by Hamming coding to infer DE combinations. Finally, AlgDE produces the $output\ data_2$, which contains fewer or equal errors than the $output\ data_1$.

Correcting an error placed on a row/column a of the data area changes the coding of the column/row b that crosses with the row/column a. This interdependence allows reducing the number of errors in the corresponding column/row b, enabling to compute a new Hamming coding for exploring novel error corrections. This logical reasoning, illustrated in axis *A2*, led to the exploration of an AlgSE that performs consecutive interlaced row/column loops. The goal is to identify the maximum number of loops after which an AlgSE can no longer correct SEs or the number of errors corrected is insignificant compared to the total number of errors.

The *A3* research axis explores the algorithm efficacy when the error correction procedure is performed only to the data bits and applied to all codeword bits (i.e., data, check, and parity). Besides, the modified product code allows us to verify the rows/columns that cross the errors occurring in the data area and decide whether the correction should be made from this analysis.

All experiments have a codeword as input produced by the LPC coding algorithm. This codeword is changed according to synthetic error patterns, enabling the analysis of the correctness of the proposed techniques. The data corrected by the LPC decoder is obtained from the AlgSE and AlgDE outputs. Additionally, we implemented versions of AlgSE and AlgDE to analyze tradeoffs about the ability to correct versus the implementation cost.

5. Correction technique correlating row-column

While Ham(8, 4) allows fixing the data and check bits with the same efficacy, the way it is used in the LPC affects the error correction rate. The LPC matrix format enables to cross row and column data and choose fixing errors based on this information. Besides, as the LPC decoder focuses on the data correction, it is possible to decide whether to correct errors in the redundancy bits; this decision increases the data correction ability in a subsequent step of the decoder.

This work implemented and analyzed four error correction techniques applied to AlgSE, which use Ham(8, 4) and rows and columns correlation: Data Correction Only (DCO), Data and Redundancy bits Correction (DRC), Data Correction Only with Crosscheck (DCOC) and Data and Redundancy bits Correction with Crosscheck (DRCC).

The decoder fixes only the data bits pointed out by the check-bit syndromes (i.e., Eqs. (22) and (23)) when applying the DCO technique; thus, instead of correcting the check and parity bits, they are recalculated from the data.

The DRC technique enables the decoder to correct data, check, or parity bits depending on the error address pointed out by the check and parity syndromes.

The decoder corrects data, check, or parity bits pointed by the check and parity syndrome bits when using the DRCC technique; however, the data bits are only corrected if there is a corresponding row/column informing the existence of some error. To correct an error in a row/column, the decoder checks through the SE and DE control variables if there is any bit with an error within the corresponding column/line, "independent" of this error position.

The decoder implementing the DCOC technique corrects only the data bits and only when the corresponding row/column indicates an error. Table 2 summarizes the correction region and whether there is crosscheck verification of all AlgSE correction techniques.

6. Single error correction algorithm - AlgSE

AlgSE applies SE corrections of type Ham(8, 4) for rows and columns; this correction starts recalculating the check and parity bits ($\widehat{Cr}$, $\widehat{Cc}$, $\widehat{Pr}$, $\widehat{Pc}$) using Eqs. (8) to (15), and with these values and the input codeword, Eqs. (16) to (19) calculate the syndromes. The *Control bits calculation* box of Fig. 5 implements the set of calculus described above.

SEs can be corrected iteratively by applying Hamming first on rows and then on columns, or vice-versa. We assessed several alternatives combining the number of successive errors on column or row corrections. Our experiments demonstrated that AlgSE is more effective when starting the error correction for the set (i.e., rows or columns) with the highest number of SEs followed by one correction with the other set (i.e., rows followed by column or vice-versa). Therefore, AlgSE performs a heuristic that decides the correction order using the *SEr* and *SEc* variables, calculated by Eqs. (24) and (25), respectively.

$$SEr = \sum_{q=0}^{3} SEr_q \tag{24}$$

$$SEc = \sum_{q=0}^{3} SEc_q \tag{25}$$

If $SEc = SEr = 0$, the algorithm considers that no SE was detected and ends the codeword decoding. Otherwise, AlgSE executes a loop sequence controlled by the *cont* counter. Each algorithm loop corrects first columns and then rows if $SEc \geq SEr$, or the opposite when $SEc < SEr$. Corrections to columns and rows occur in the *Apply Hamming on columns/rows* boxes.

The experiments used in this work explore up to 4 iterations through a sequence of column/row error corrections; because the LPC organization containing 4 rows and 4 columns does not allow a SE pattern needing more than 4 correction passages.

Table 2
Crosscheck and correction regions of the AlgSE correction techniques.

Technique	Correction		Crosscheck verification
	Data bits	Data + check bits	
DCO	X		
DRC		X	
DCOC	X		X
DRCC		X	X

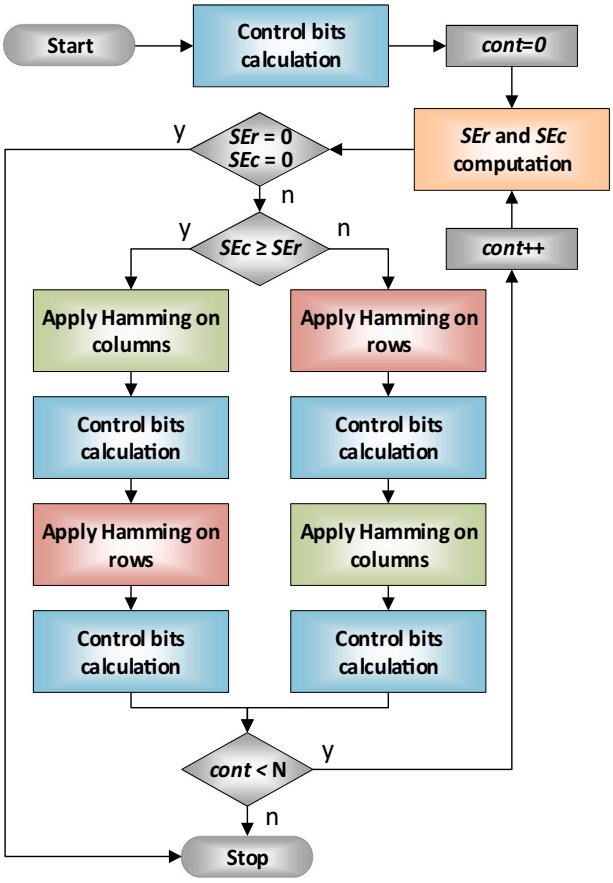

Fig. 5. High-level description of AlgSE.

Fig. 6(a) shows a pattern with 7 SEs in the data area. Fig. 6(b) shows these same errors with the corresponding control variables indicating whether the algorithm detects an SE or DE in each row and column. The double arrows show whether the correction refers to the row or column, and the arrow number describes the sequence of correction steps.

Fig. 6(b) shows that the execution of the method *Apply Hamming on columns* box on steps 1 allows correcting $D_{0,\ 2}$. Next, Fig. 6(c) displays that the execution of the *Control bits calculation* box produces two SEs ($D_{0,\ 0}$ and $D_{1,\ 1}$) on rows, which are corrected executing *Apply Hamming on rows* box. Subsequently, AlgSE starts a new loop recomputing *SEr* and *SEc* variables. Once again, $SEc \geq SEr$, thus, Fig. 6(d) illustrates that AlgSE applies *Hamming on columns* to fix bits $D_{3,\ 0}$ and $D_{2,\ 1}$ in step 3. Finally, Fig. 6(d) shows that the control bits are recalculated, resulting in two SEs in rows ($D_{2,\ 3}$ and $D_{3,\ 3}$) that are corrected executing *Apply Hamming on rows*. AlgSE starts the last loop recomputing *SEr* and *SEc* variables. At this moment, Fig. 6(f) shows that this error pattern was entirely correct; thus, $SEr = SEc = 0$ forcing to stop the AlgSE execution.

The $cont < N$ test defines the number of loops to be executed by the algorithm. The name of the algorithm illustrated in Fig. 4 is associated with *N*; i.e., AlgSE0, …, AlgSE3 correspond to $N = 0, \ldots, 3$, respectively. Any AlgSE greater or equal than AlgSE1 could correct the error pattern scenario illustrated in Fig. 6(a).

The AlgSE loop technique can be done automatically if the algorithm evaluates the number of loops needed to correct each pattern dynamically. This evaluation can be done by checking if at least one error correction was performed after each loop, and the algorithm ends if there are no more error corrections. However, this type of exploration is not the focus of this work.

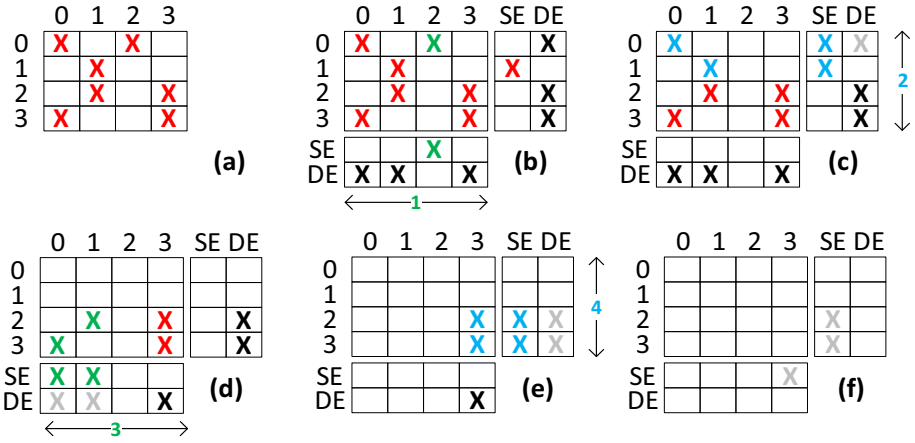

Fig. 6. Example of SE correction in the data area obtained through consecutive AlgSE loops. The check and parity bits did not contain errors and were omitted to avoid overloading the figure.

7. Double error correction strategy

The DE correction makes a cross-analysis of DEs detected in rows and columns, increasing the ability to correct data in a product code. This technique is based on the LPC matrix format, which checks all data bits using Ham(8, 4) arranged in columns and rows. While Ham(8, 4) is a SECDED code, the rows and columns crossing allows inferring DEs by the majority analysis of the error events.

A. Technique description

The correction technique analyzes all DE combinations with the corresponding error values obtained by the syndrome computation. Each DE combination produces a one-bit codeword address ***that the syndromes would point to as the cause of a SE***. Although this address does not point to the real source of the error, it is used to associate groups of DEs.

The extended Hamming code does not employ the parity bit in the error-bit address calculation; thus, although Ham(8, 4) consists of 8 bits, only 7 bits are used, resulting in 21 combinations of DEs $\binom{7}{2}$. The syndromes produce 7 error addresses computed by Eqs. (22) and (23), with DEs being homogeneously distributed in 21 combinations, so each address corresponds to 3 DEs. For example, address 1 is produced whenever there are DEs described in the following three tuples: (D2,D3), (D0,C1) and (D1,C0). Fig. 7 partially shows the 21 DE combinations,

Comb.	Codeword							Recomp.			Syndromes				Double error	
	D0	D1	D2	D3	C0	C1	C2	C0	C1	C2	sC0	sC1	sC2	EA		
1			E	E						E	0	0	1	1	D2	D3
2	E					E			E	E	0	0	1	1	D0	C1
3		E			E			E		E	0	0	1	1	D1	C0
4		E		E					E		0	1	0	2	D1	D3
5	E						E		E	E	0	1	0	2	D0	C2
6			E		E			E	E		0	1	0	2	D2	C0
...																
19	E				E				E	E	1	1	1	7	D0	C0
20		E				E		E		E	1	1	1	7	D1	C1
21			E				E	E	E		1	1	1	7	D2	C2
	3	5	6	7	4	2	1									

Error address (**EA**)

Legend: symbol 'E' represents an error in a bit within the codeword.

Fig. 7. DE combinations for Ham(8, 4). The parity bit is not represented, as it is not used to calculate the reference address, which is limited to 21 combinations $\binom{7}{2}$.

with the corresponding syndromes and the reference address for each combination.

Fig. 7 does not show the parity bits of the recalculated codeword and the syndrome fields to emphasize only the combinatorial analysis. However, it is essential to note that no error tuple changes the recalculated parity bits, as they are only DEs; thus, the parity syndromes are zeroed. For each codeword, the OR-logic of the verification syndrome bits results in one (Eqs. (20) and (21)), and the parity syndrome is zero, indicating the DE presence (refer to Table 1).

Table 3 displays the 21 combinations, shown in Fig. 7, grouped in sets of three tuples. Table 3 also shows the address used to reference the bitflip in the case of SE correction.

B. Description of the DE correction algorithm - AlgDE

Fig. 8 illustrates that AlgDE encompasses two steps. The first step contains two nested loops (between lines 5 and 37) used to fill the data matrix, which points out the bits of the data area where the DEs may have occurred. The second step contains two other nested loops (between lines 38 and 43) that invert the data bits identified as DE in the data matrix.

AlgDE has, as inputs, the EAr and EAc integer vectors (Eqs. (22) and (23)), the Boolean vectors DEr and DEc, and the tab matrix, which is constructed from the logic described in Table 3, as illustrated in Fig. 9. Additionally, the algorithm can read/write from/in the decWord matrix that contains the data in the LPC format.

AlgDE starts by zeroing the data matrix, which has the same size as the LPC data area; the redundancy bits are not part of this matrix, as only the data region is double-checked.

The outermost loop of the first step controls the correction of the DEs detected in the rows (rc = 0) and, later, in the columns (rc = 1), using the variable rc in all decisions corresponding to the row or column. The innermost loop runs through the four rows or four columns of the data matrix. The variable vDE informs that rows or columns are analyzed

Table 3
The 21 combinations of DEs grouped according to the address produced by the check-bit syndromes.

Address	Single error	Double errors		
1	C2	D2, D3	D0, C1	D1, C0
2	C1	D1, D3	D0, C2	D2, C0
3	D0	D1, D2	D3, C0	C1, C2
4	C0	D0, D3	D1, C2	D2, C1
5	D1	D0, D2	D3, C1	C0, C2
6	D2	D0, D1	D3, C2	C0, C1
7	D3	D0, C0	D1, C1	D2, C2

AlgDE

```
1   IN NATURAL tab[7][3][2], EAr[4], EAc[4]
2   IN BOOLEAN DEr[4], DEc[4]
3   INOUT BOOLEAN decWord[4][4]
4   NATURAL data[4][4] ← 0s
5   for rc ← 0 to 1 {
6      for k ← 0 to 3 {
7         BOOLEAN vDE ← rc = 0 ? DEr[k] : DEc[k]
8         if vDE = 1 {
9            NATURAL add ← rc = 0 ? EAr[k] : EAc[k]
10           BOOLEAN expt ← 0
11           for j ← 0 to 2 {
12              NATURAL b1 ← tab[add-1][j][0]
13              NATURAL b2 ← tab[add-1][j][1]
14              BOOLEAN e1 ← 0, e2 ← 0
15              if b1 ≤ 3
16                 e1 ← rc = 0 ? DEc[b1] : DEr[b1]
17              if b2 ≤ 3
18                 e2 ← rc = 0 ? DEc[b2] : DEr[b2]
19              if (b1≥4 OR e1=1) AND (b2≥4 OR e2=1) {
20                 if b1 ≤ 3 {
21                    rc=0 ? data[k][b1]++ : data[b1][k]++
22                    expt ← 1
23                 }
24                 if b2 ≤ 3 {
25                    rc=0 ? data[k][b2]++ : data[b2][k]++
26                    expt ← 1
27                 }
28              }
29           }
30           if expt = 0 {
31              NATURAL b ← add = 3 ? 0 : add - 4
32              if b ≥ 0 AND b ≤ 3
33                 rc=0 ? data[k][b]++ : data[b][k]++
34           }
35        }
36     }
37  }
38  for j ← 0 to 3 {
39     for k ← 0 to 3 {
40        if data[j][k] = 2
41           decWord[j][k] ← !decWord[j][k]
42     }
43  }
```

Fig. 8. Pseudo-code of the DE correction algorithm - AlgDE.

Double errors			Address
D2, D3	D0, C1	D1, C0	1
D1, D3	D0, C2	D2, C0	2
D1, D2	D3, C0	C1, C2	3
D0, D3	D1, C2	D2, C1	4
D0, D2	D3, C1	C0, C2	5
D0, D1	D3, C2	C0, C1	6
D0, C0	D1, C1	D2, C2	7

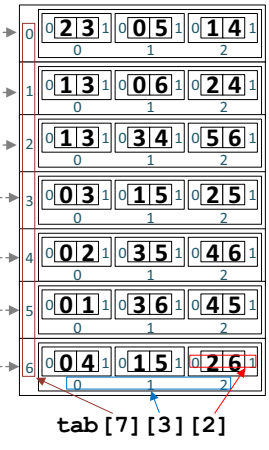

Fig. 9. Composition of the tab matrix from Table 3.

only if there is a DE. In case of an error, the add variable receives the address that identifies the DE in the row or column. The add variable is decremented from 1 to point to the first index of the tab matrix (see the relationship of the Address column to the tab indexes in Fig. 9).

The variables b1 and b2 receive the first and second elements of the error tuples associated with the second index of the tab matrix; this second level has dimension 3, implying that the loop between lines 11 and 29 is executed three times. The DE pointed in a row is checked with the corresponding columns that must also point to a DE; if so, e1 and/or e2 receive 1. However, this check is not performed when b1 or b2 points to a redundancy bit (i.e., b1 ≥ 4 or b2 ≥ 4) since LPC does not double-check redundancies.

Lines 21 and 25 increment the positions equivalent to each tuple element in the data matrix for any valid error tuple. The goal is to get the data matrix to indicate the correct DE combination by crossing rows and columns. If there is a DE in the data, the data matrix will have at least one bit incremented twice due to the passage through the rows and columns, resulting in a cell with value 2. The combinations (C1,C2), (C0, C2), and (C0,C1), described in lines 3, 5, and 6 of Table 3, respectively, have only check bits. Thus, these combinations do not change the data matrix; they are placed in the tab matrix only for the logical completeness of the matrix.

AlgDE uses the variable expt to catch an exception when the variable vDE informs that there is a DE in a row/column, but there is no indication of this DE in the corresponding column/row. This exception happens when DE is a combination of the parity bit and a data or check bit. AlgDE considers only the parity and data bits combinations since the code correction is done only in the data area. If the loop execution between lines 11 and 29 does not produce at least one DE entry, lines 22 and 26 will not be executed, keeping expt at 0; thus, the algorithm deduces that the DE was related to the parity, causing the data associated with the parity bit to be increased in the data matrix (line 33). Line 31 associates the address of each data with the address obtained in the EAr or EAc vector. In this case, the address of the DE is the same as the address of an SE since the parity bit does not change EAr or EAc; i.e., add equal 3, 5, 6, and 7 means data bits D0, D1, D2 and D3, respectively.

Instead of using an exception mechanism, the tab matrix could have dimensions [7,4,2], making each address have four possible DE combinations instead of just 3. However, the analysis performed within the AlgDE execution showed that the number of errors fixed was higher when using an exception mechanism. By increasing the number of valid combinations, the number of false DEs also increased, reducing the efficacy of the algorithm.

AlgDE finishes by performing the second step, which makes a nested double loop changing all the positions of the codeword that had a double increment in the data matrix.

C. AlgDE application examples

Fig. 10 to Fig. 12 exemplify three scenarios containing several types

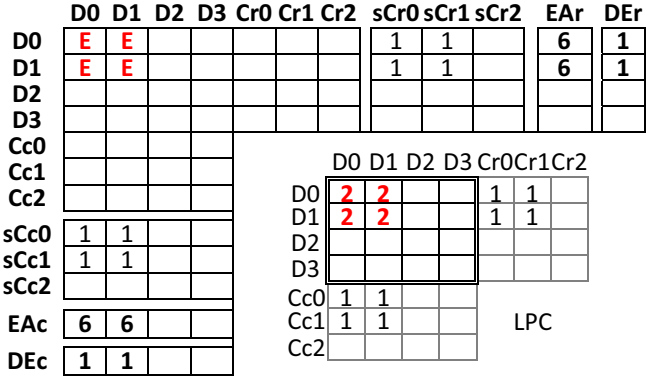

Fig. 10. Scenario containing four bitflips that generate four DEs annotated in the row control variables (DEr) and columns (DEc). A rectangle with double edges represents the data matrix.

of DEs with the related syndromes (sC0, sC1, sC2), DE control signals (DEr and DEc), as well as addresses regarding the computed error (EAr and EAc). The lower right rectangle of each figure depicts the structure of the LPC code with the data matrix computation; additionally, the areas referring to the check bits are presented, although they are not present within AlgDE, to show the computed error tuples. For easy viewing, cells with a value of 0 are shown without content.

Fig. 10 contains only DEs with address 6, indicating that the error tuples are (D0,D1), (D3,C2), and (C0,C1). The execution of AlgDE verifies that the pair (D3,C2) cannot be a source of DE since the column and the row associated with D3 have vDE = 0. Therefore, only the tuples (D0,D1) and (C0,C1) could be the source of DE. Since the redundancy bits are not double-checked, only the pair (D0,D1) is incremented in the data matrix. Consequently, at the end of the first step, the data matrix will have only the bits that had an error noted with the value 2, allowing the second step of AlgDE to invert these four bits, correcting the entire data area.

Fig. 11 illustrates three columns and three rows with DEs; i.e., only the row associated with D3 and the column associated with D1 are correct. These lines are used to reduce the possibility of valid error tuples. Thus, address 4 for the first line indicates that only the tuples (D0,D3) and (D2,C1) are valid, while the tuple (D1,C2) is not valid. A similar situation occurs with address one on the line, which invalidates the tuple (D1,C0), and with addresses 5, 3, and 6 in the columns, where the tuples (D3,C1), (D3,C0), and (D3,C2), respectively, are not valid. After only increasing the cells referring to the valid tuples, the data matrix has a set of cells with a value of 2, another with a value of 1, and the rest with a value of 0 (cells without content). Again, the final step of AlgDE can correctly invert all the DEs contained in the data, reaching 100% efficiency in correction and errors.

The error scenario of Fig. 12 contains DEs in data and redundancy bits, as well as SEs in some redundancy bits. SEs are not reported to AlgDE, so the algorithm assumes that rows or columns have no error in these cases. This situation causes EAr = 5 to refer only to the tuples (D3, C1) and (C0,C2), and EAr = 2 to refer only to the tuple (D0,C2). Similarly, EAc = 6 points only to the tuples (D3,C2) and (C0,C1), and EAc = 7 points only to the tuple (D0,C0). When executing AlgDE, the data matrix will contain only the DEs occurring in the data bits annotated with the value 2. The final step of AlgDE inverts the two corresponding bits in the decWord matrix, making all the data have the correct values; i.e., the algorithm achieves 100% effectiveness in correcting errors.

D. LPC miscorrection discussion

This section discusses some miscorrection cases of LPC regarding AlgSE and AlgDE algorithms.

Ham(8, 4) presents anomalies with three or more error patterns. An anomaly example happens when having errors in the first three data bits

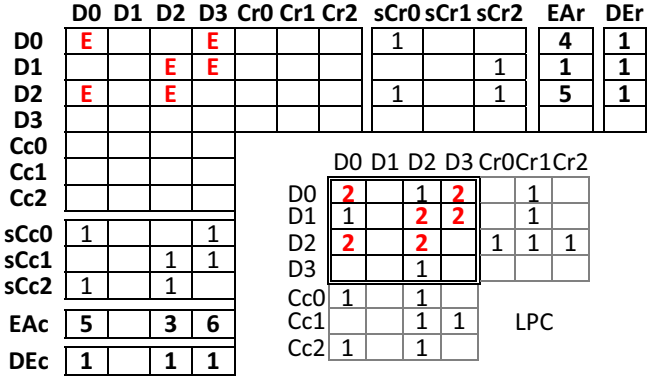

Fig. 11. Scenario containing six bitflips that generate six DEs noted in the row (DEr) and column (DEc) control variables.

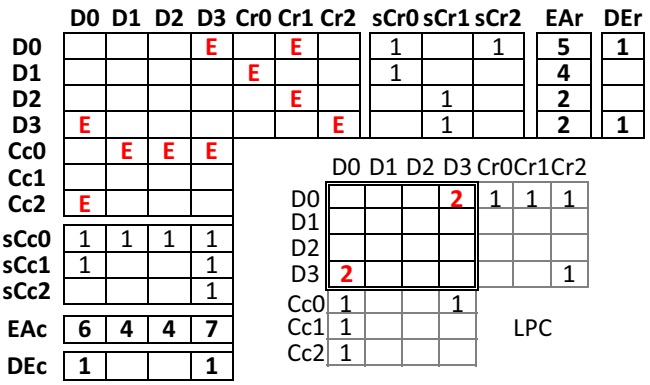

Fig. 12. Scenario containing ten bitflips that generate two DEs annotated in the row control variables (DEr) and columns (DEc), in addition to 4 unique errors not reported to AlgDE.

(D0,D1,D2) since the check bits remain the same, preventing error detection. The parity bit included in the extended Hamming code enables to infer this anomaly but not detect the error position; thus, reducing the efficacy of 1D ECCs based on Hamming. Another anomaly example is the occurrence of errors in the three check bits; in this case, the code points to an error in D3, and the parity would wrongly confirm the error and, consequently, 1D ECCs based on Hamming perform a miscorrection. Fortunately, 2D ECCs like LPC can crosscheck rows and columns to mitigate this anomaly and other complex error patterns, only failing when the number of errors is very high. Fig. 13 displays two specific error patterns cases of miscorrection.

On the one hand, Fig. 13(a) shows that the first three bits of the first row are changed and the check-bit syndromes of this row cannot identify the errors. However, the syndromes of the first three columns identify the errors in the D0 bit, enabling to carry out the correction through the AlgSE algorithm (see Fig. 5). On the other hand, Fig. 13(b) shows a more aggressive error pattern that leads the AlgSE algorithm to miss correct the D3 bit of the last data row. This miscorrection happens because crosschecking rows and columns point to the same bit error.

AlgDE does not have SE information as input; thus, it must be used in conjunction with AlgSE, and the experimental results show that the AlgDE efficacy is superior when performed after AlgSE. The DEs correction technique does not correct redundancy bits, as they can be recalculated from the data. Consequently, whenever the technique can correct 100% of the data, it will achieve 100% effectiveness.

Like AlgSE, AlgDE does not guarantee the correction of all DEs for any scenario. Additionally, this technique is subject to anomalous situations; e.g., DEs generated in the check bits can be wrongly calculated as

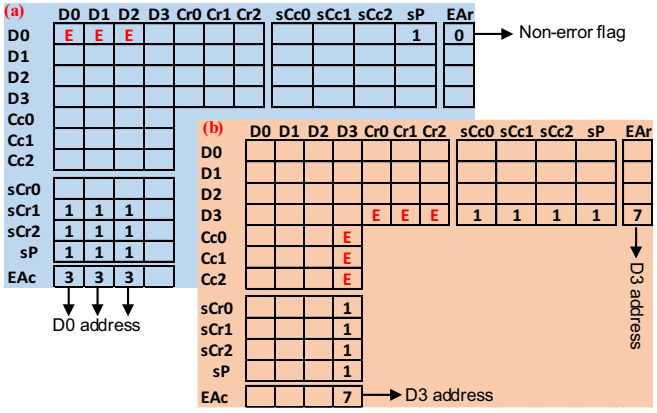

Fig. 13. Two error scenarios whose AlgSE (a) can correct errors and (b) miss correct errors. Each bitflip is represented by E.

DEs in the data bits. However, these anomalies only exist when the number of errors increases a lot, more precisely with eight or more errors. Fig. 14 exemplifies an anomaly case; two error scenarios generate the same syndrome values, preventing AlgDE from inferring the right error positions; thus, the AlgDE execution corrects the data area, independent of the error scenario.

8. Experimental results

This section reports the experimental setup and results regarding the decoding techniques and algorithms described in Sections 4 to 7.

A. Control-Data Flow of the Experiments

The central aspect addressed in this work is the LPC correction capacity concerning different techniques and algorithmic approaches. Therefore, Fig. 15 displays the control-data flow followed by all experiments, except for the implementation costs evaluation. The control flow, which contains a double nested loop, is described by double arrows, while dashed arrows describe the data flow.

The control-data flow starts by assigning 1 to the number of errors $\#e$ that each scenario will have, then executing the outermost loop controlled by the $\#e \leq totErr$ comparison. For each outermost loop execution, the flow performs all combinations of scenarios with $\#e$ errors, and $totErr$ is an arbitrary number; in experiments that evaluate errors in the data and redundancy areas exhaustively, $totErr$ is equal to 16 and 32, respectively; however, for scenarios with errors placed throughout the codeword, only 10 or 11 errors were explored due to the high number of combinations.

At each entry in the outermost loop, the *errSE* and *errDE* variables that control the number of errors found in the AlgSE and AlgDE executions for a given $\#e$ are reset. The innermost loop starts with the *Error pattern Generator*, which represents the procedure that receives $codeword_0$, containing a set of data encoded in LPC, and $\#e$, producing $codeword_1$, which is equivalent to $codeword_0$ with the injection of $\#e$ errors.

$codeword_1$ is input from AlgSE, which applies SE correction rules. If AlgSE corrects all errors, it receives a new pattern with the same number of $\#e$ errors. This procedure is done exhaustively in the entire evaluated area (containing $\#a$ bits). Depending on the experiment, $\#a$ can be 16, 32, or 48, corresponding to the data area, redundancy, or the entire codeword, respectively. This exhaustive procedure makes the number of combinations evaluated for each $\#e$ equal to $\binom{\#a}{\#e}$. For example, in an experiment that evaluates all possible scenarios of 8 errors in the data area, the number of combinations is equal to $\binom{16}{8} = 12,870$.

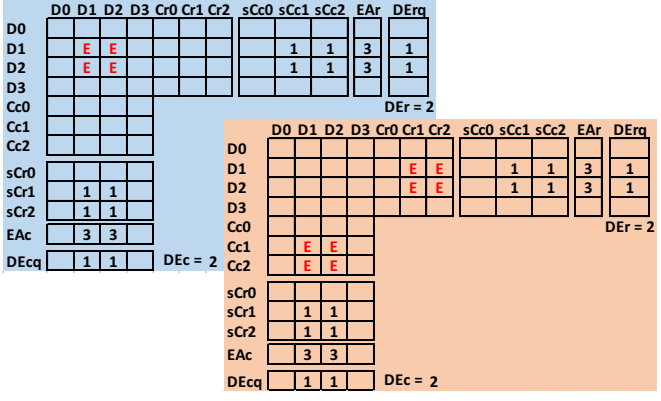

Fig. 14. Two error scenarios that generate the same syndrome values. Consequently, AlgDE produces a miscorrection if the errors are in the check bits since the four internal data bits will be flipped.

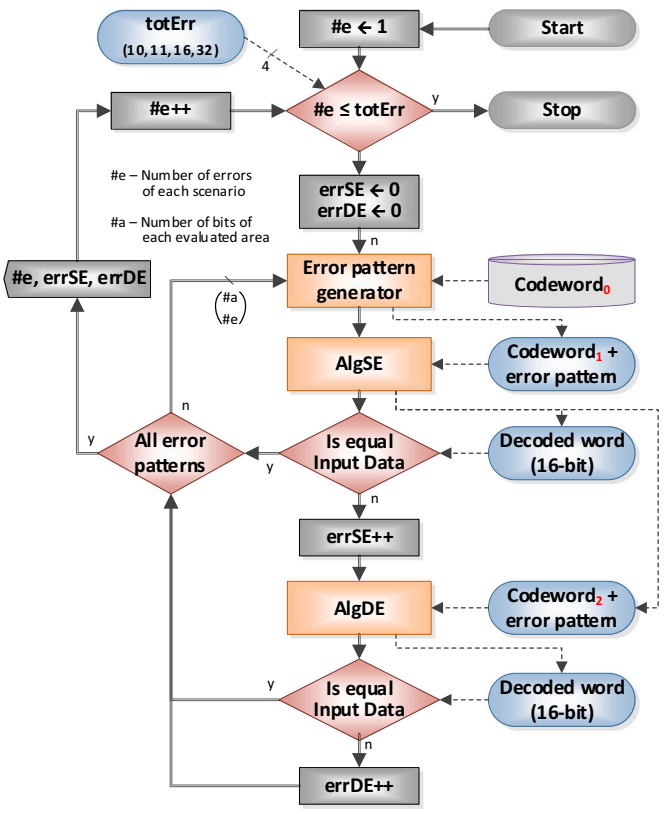

Fig. 15. Basic control-data flow applied in the experiments. The control flow is represented by double lines, while dashed lines represent the data flow.

If AlgSE does not correct all the errors injected into the $codeword_1$, then the variable *errSE* is incremented, and AlgDE receives the $codeword_2$, which is the $codeword_1$ containing the corrections made by AlgSE, as input. Similarly, AlgDE applies the DE correction rules, producing a new data area that is next compared with the $codeword_0$ data area. If the data is the same, then AlgDE was able to correct an error scenario that had not been entirely corrected by AlgSE. Otherwise, the variable *errDE* is incremented.

The innermost loop continues executing until all $\#e$ error scenarios are generated. When the innermost loop ends, *errSE* and *errDE* will contain the total number of errors not corrected in the AlgSE and AlgDE stages, respectively, for all combinations of $\#e$ errors. Subsequently, the flow re-executes the procedures of the outermost loop, which increments $\#e$, while $\#e \leq totErr$. Eq. (26) computes the total number of combinations ($\#totComb$) that the decoding flow performs; e.g., to evaluate error scenarios across the codeword (i.e., $\#a = 48$) in the range $\#e = [1, 10]$, the decoding algorithm is executed 8,682,997,470.

$$\#totComb = \sum_{\#e=1}^{totErr} \binom{\#a}{\#e} \tag{26}$$

B. Evaluation of the error correction technique with row-column crosschecking

Section 5 presents the DCO, DCOC, DRC, and DRCC techniques that allow correcting only data or data and redundancy, inferring or not errors by crosschecking rows and columns. Fig. 16 shows the correction capacity of each technique for scenarios from 1 to 10 errors, taking as a reference the correction values obtained with DRCC, which resulted in fewer error corrections for all scenarios.

All techniques achieved a 100% error correction rate in scenarios with 1 or 2 bitflips; however, with 3 or more bitflips, the techniques

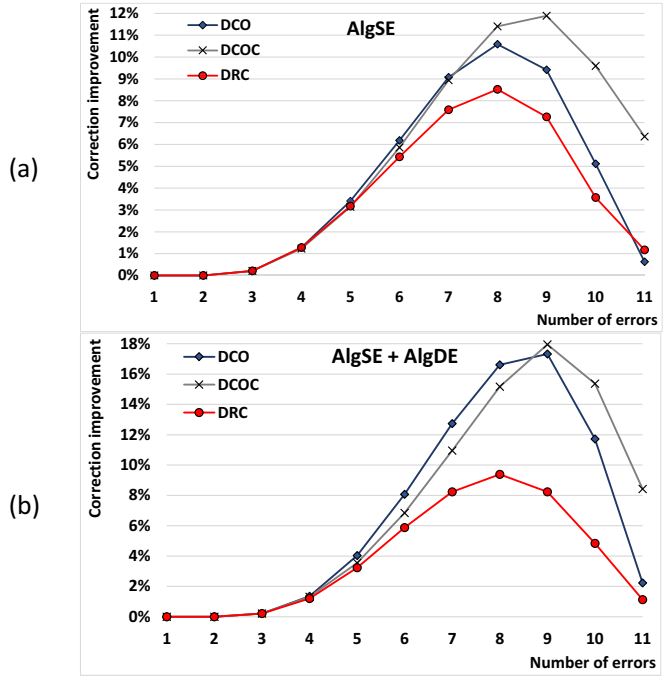

Fig. 16. Variation of the error correction capacity using the DCO, DCOC, DRC, and DRCC techniques for the LPC decoder using AlgSE and AlgDE. The values show the percentage difference for the worst case (DRCC).

exhibit different correction capabilities. Fig. 16(a) shows that DCO, DRC, and DCOC increase the error correction capacity for scenarios of up to 8 errors when compared to the DRCC reference. With 9 error scenarios, only DCOC still shows a growth trend, but the relative error correction capacity tends to reduce from this point on.

DCO and DCOC provide to AlgSE almost the same efficacy up to 7 error scenarios, with a small advantage for the DCO technique. However, Fig. 16(b) shows that when associating AlgDE with the decoder, DCO obtains even more significant results, which are only overcome from scenarios with 9 errors or more. DCO results in greater correction efficacy for AlgSE with up to 7 error scenarios, and this efficacy is enhanced with the application of AlgDE. However, aggregating AlgDE in the decoder produces an anomalous correction behavior. From 8 error scenarios, although DCOC exhibits better results than DCO for AlgSE, the same does not happen in the decoding result when AlgDE is added, suggesting that some corrections made during the AlgSE execution prevented some corrections performed by AlgDE.

The experimental results suggest that LPC is more effective when correcting only data errors, without considering the redundancy errors, due to the matrix format allowing the data verification to occur both by columns and by rows. Additionally, the iterative method with the crosschecking technique, which corrects errors only when the equivalent row/column also points out an error, is effective only when the number of errors grows a lot, as in this case, the technique minimizes the possibility of wrong corrections.

C. Evaluation of the AlgSE iterative approach

This section presents the results of AlgSE error correction capacity in terms of the correction heuristics and iterative degree. All the experiments performed in this subsection use AlgSE with the DCO technique. As described in Section 6, this work explored several heuristics to determine the most efficacy error correction procedure. In this experiment, we describe the results of four heuristics: BasicLoop - each iteration first corrects rows then corrects columns (the procedure, starting with columns and then rows, produces the same results due to the symmetry of the code and the exhaustive evaluation); InvertLoop - in one iteration corrects rows after columns, in the subsequent iteration inverts the order, and so on; PriorityLoop - each iteration corrects only rows or columns, privileging those with the highest number of SEs. This method makes twice as many iterations so that the number of row/column corrections is the same for all methods; FairPriorityLoop - each iteration corrects rows and then corrects columns or vice versa; the number of SEs defines the row/column or column/row order.

Fig. 17(a) and (b) show the relative error correction capacity of each heuristic, considering only AlgSE and the joint effect of applying AlgDE, respectively. The experiment shows that the FairPriorityLoop heuristic is the most effective for all evaluated error scenarios. The BasicLoop heuristic, adopted by most of the works presented in Section 2, is less efficacious than FairPriorityLoop, but much higher than the other heuristics. We adopted the FairPriorityLoop heuristic to execute all other experiments due to the results obtained in this experiment.

Next, we evaluated the iterative effectiveness of error correction for the LPC. Table 4 presents the correction percentage for scenarios from 1 to 10 errors, with AlgSE performing from one to four loops; the experiment uses the DCO technique with the FairPriorityLoop heuristic.

The results display that AlgSE obtains the same correction efficacy for up to three bitflips regardless of the number of loops. For scenarios with four and five bitflips, the second iteration level increases the correction efficacy significantly. With six and seven bitflips, a third iterative level increases the number of corrected errors, and a fourth iterative level is needed from scenarios with eight bitflips. However, the gains obtained become smaller in percentage as the number of errors in the scenarios grows.

Additionally, as discussed in Section 6, all the iterative levels higher than four achieve the same error correction efficacy. This experiment concludes that AlgSE1 is almost as good as AlgSE3, with much less execution time.

D. Evaluation of the AlgDE error correction efficacy

Table 5 displays pairs of lines containing error correction values; the

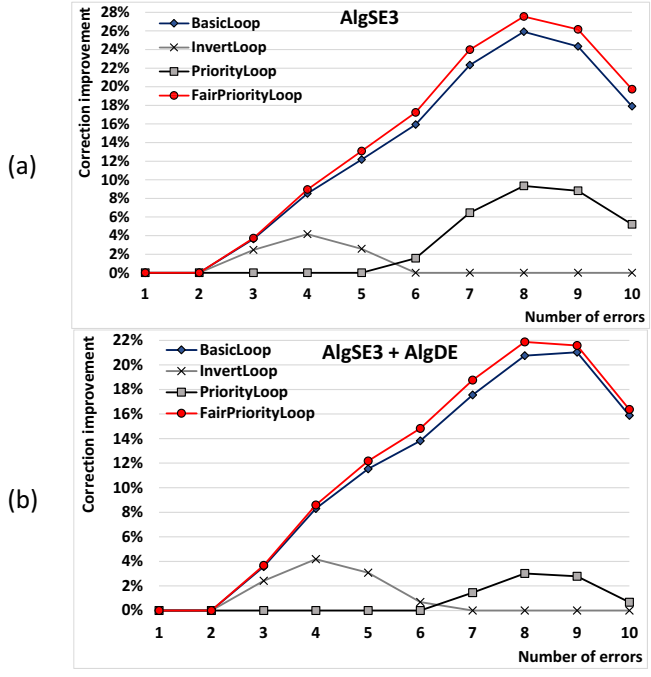

Fig. 17. Comparison of four heuristics used to control AlgSE iterations. The values presented are normalized according to the least efficacy heuristic for each error scenario.

Table 4
Error correction percentage considering the AlgSE iterative procedure and scenarios from 1 to 10 errors.

Errors	1	2	3	4	5	6	7	8	9	10
AlgSE0	100	100	98.52	92.31	79.94	62.46	43.07	25.97	13.6816	6.344343
AlgSE1	100	100	98.52	93.83	84.15	68.81	49.69	30.73	16.0830	7.150572
AlgSE2	100	100	98.52	93.83	84.15	68.91	49.93	30.97	16.1790	7.166544
AlgSE3	100	100	98.52	93.83	84.15	68.91	49.93	30.98	16.1793	7.166537

Table 5
Error correction efficacy of AlgSE alone and AlgSE together with AlgDE, considering scenarios from 1 to 11 errors.

Errors	1	2	3	4	5	6	7	8	9	10	11
AlgSE0	100	100	98.52	92.31	79.94	62.46	43.07	25.97	13.68	6.34	2.62
AlgSE0 + AlgDE	100	100	100	97.80	92.01	81.55	65.31	44.97	25.21	11.19	4.13
AlgSE1	100	100	98.52	93.83	84.15	68.81	49.69	30.73	16.08	7.15	2.78
AlgSE1 + AlgDE	100	100	100	99.30	96.22	88.02	72.61	51.07	28.71	12.36	4.29
AlgSE2	100	100	98.52	93.83	84.15	68.91	49.93	30.97	16.18	7.17	2.78
AlgSE2 + AlgDE	100	100	100	99.30	96.22	88.12	72.85	51.32	28.83	12.38	4.29
AlgSE3	100	100	98.52	93.83	84.15	68.91	49.93	30.98	16.18	7.17	2.78
AlgSE3 + AlgDE	100	100	100	99.30	96.22	88.12	72.85	51.32	28.83	12.38	4.29

first line describes the error correction capacity obtained with AlgSE and the second one depicts the increase of this capacity when inserting AlgDE. This experiment uses AlgSE with 1 to 4 iterations, DCO technique, and FairPriorityLoop heuristic. AlgDE improves the correction capacity for all error scenarios, allowing LPC to achieve 100% correction for 3 error scenarios; besides, for 4 error scenarios, LPC decoding reaches an efficacy between 97.8% and 99.3%.

Fig. 18, obtained from Table 5, highlights the ability to correct errors with and without AlgDE. Fig. 18(a) shows, in values relative to AlgSE, that the maximum gain of AlgDE occurs with scenarios of 7 errors; e.g., AlgSE3 manages to achieve only 49.93% error correction, and when inserting AlgDE, error correction reaches 72.85%, i.e., a 22.92% increase. Fig. 18(b) reveals that when inserting AlgDE, the percentage of absolute gain in error correction is increased up to scenarios with 9 errors, and then gradually, the gain is reduced. Due to the high computational cost, we did not extend this assessment to scenarios with more errors; however, it is possible to note that for scenarios with 11 errors, the LPC encoding results in values lower than 5%, not justifying explorations of more aggressive error scenarios.

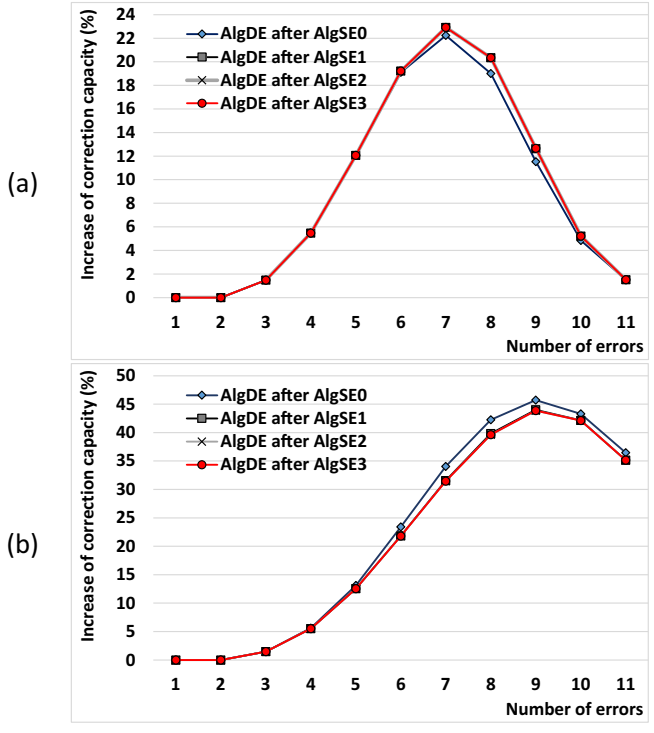

Fig. 18. The increase in error correction capacity when inserting AlgDE - (a) shows the relative difference between AlgSE and AlgDE; (b) shows the difference in absolute value.

The results of this experiment emphasize that the effectiveness in correcting errors provided by AlgDE is not negligible. Additionally, it allows us to verify that the gains for AlgSE0, which has the least complexity, are higher than for the more complex algorithms (i.e., AlgSE1, AlgSE2, and AlgSE3), where there is practically no relative difference.

E. Data and redundancy implementations in memory

The previous subsection focuses on correcting the data and the parity regions together. This subsection assesses the effect of errors occurring in the data and redundancy regions separately; consequently, it is necessary to emphasize that the error correction rates presented in this subsection differ from those shown in the previous subsection. The importance of this analysis lies in the possibility of choosing Commercial Off The Shelf (COTS) or RADiation HARDening (RAD-HARD) memories in critical applications. On the one hand, COTS components in space applications provide state-of-the-art memory technologies that reduce the design and implementation costs compared to RAD-HARD memory costs. On the other hand, although a RAD-HARD memory is not entirely insensitive to radiation, it is much more reliable than a COTS memory [56–59].

We propose implementing LPC in a heterogeneous memory system - a 16-bit memory containing data and a 32-bit memory covering the redundancy bits. The data writing and reading are carried out simultaneously in these memories by an encoder/decoder circuit responsible for synchronizing the information. Therefore, we explored the efficiency of memory technology in data and redundancy regions.

The experiments presented in this subsection use AlgSE with one iteration, DCO technique, and FairPriorityLoop heuristic. Fig. 19 displays the correction capacity for all combinations with up to 16 bitflips in the data area only, considering both the potential of only applying AlgSE and the joint use of AlgSE with AlgDE. The experiment shows that for scenarios of up to 3 errors, the correction capacity is 100%. This capacity remains above 90% with 4 and 5 errors; however, the correction efficacy declines dramatically - the error correction capacity is null from 9 errors on.

Fig. 20 illustrates the correction capacity for all combinations with up to 32 bitflips, regarding only AlgSE since AlgDE is not applicable in the redundancy area. Although the errors are in the redundancy region,

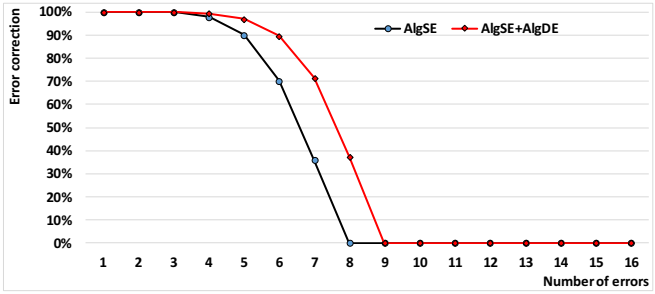

Fig. 19. Error correction capacity of AlgSE alone and combined with AlgDE, for errors affecting only the data region.

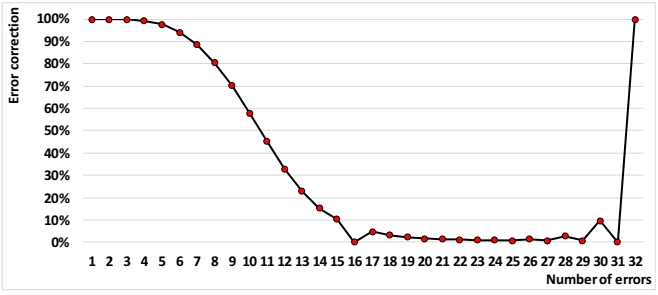

Fig. 20. Error correction capacity of AlgSE with all combinations of errors in the redundancy area.

the corrections are applied to the data. Thus, we are evaluating errors in the redundancy area that generate ***false errors*** in the data area, implying wrong corrections that modify the data.

The correction capacity is maintained above 90% up to 6 bitflips, decreasing smoothly until zero with 16 bitflips. From 16 to 31 bitflips, the correction capacity is less than 10%, with less than 3%, on average. Finally, when the entire redundancy area is in error (i.e., 32 upsets), AlgSE reaches 100% correction because when inverting all redundancy bits, there is no correction made in the data area.

These experiments show that the data region degrades more quickly than the redundancy region. Thus, we propose that the physical implementation of the data area be made with a less-sensitive radiation memory, such as a RAD-HARD, and the redundancy area be implemented with a COTS memory.

F. LPC encoder and decoder syntheses

Table 6 presents the synthesis results of the LPC encoder and AlgSE0, AlgSE1, and AlgDE algorithms used in LPC decoding. We implemented all AlgSEs with DRC and FairPriorityLoop. The results include area consumption, power dissipation, and delay, achieved with the Cadence RTL Compiler software for CMOS technology with the 65 nm CORE65GPSVT standard cell library under normal operating conditions. The entire hardware implementation was performed with combinational circuits described in Verilog. We only show a subset of the encoding and decoding algorithms, aiming to elucidate the order of magnitude of the synthesis costs.

Table 6
Analysis of area consumption, power dissipation, and delay for the LPC encoder and decoder circuits.

LPC circuit		Area (μm^2)	Power (uW)	Delay (ns)
Encoder		340	17	0.14
	AlgSE0	2724	260	1.94
Decoder	AlgSE1	5890	730	2.00
	AlgDE	2218	290	1.64

The LPC encoder has a very low implementation complexity that reflects values of magnitude lower than those obtained in decoding. The comparison between AlgSE0 and AlgSE1 shows that the iterative degree more than doubles the area and power values. AlgSE0 and AlgSE1 have critical paths of near delay; however, AlgSE1 requires one more clock cycle for the second algorithm iteration.

Additionally, AlgDE has an implementation cost close to AlgSE0. Concerning area consumption and power dissipation, the implementation of AlgSE0 associated with AlgDE results in a lower cost than the implementation of AlgSE1 alone. Finally, the combined evaluation of the synthesis information together with the error correction values (Table 6) shows that AlgSE0 + AlgDE is more effective and efficient than AlgSE1.

G. LPC compared to other Space Application ECCs

This last section correlates the LPC correction algorithms with other ECC correction methods designed for space applications. Except for BCH, we used the error correction and synthesis costs provided by the related works; as illustrated by Fig. 21, this consideration limited the error correction rates for scenarios with 1 to 5 errors.

We highlight that only LPC and PCoSA [19] were evaluated by exhaustive methods of inserting faults, taking all possible error scenarios into account. The other ECCs [15,21,23,24,60] considered specific error patterns (e.g., scenarios with only adjacent or burst errors), which drastically reduces the number of scenarios to be assessed. This fact privileges designing high-effective algorithms with low implementation costs, not allowing a fair comparison among ECCs. We chose the AlgSE0 + AlgDE combination to represent the LPC decoding algorithms, a highly effective combination that does not penalize the synthesis cost excessively. Fig. 21 reveals that although the algorithms used in PCoSA and LPC focus on correcting any error pattern, they reach the highest correction rates. Also, the union of AlgSE with AlgDE makes LPC have a correction capacity superior to PCoSA. On the one hand, BCH can correct all 3-error patterns (in burst format), whereas there are 3-error patterns where EG-LDPC cannot correct all bitflips. On the other hand, the correction capacity of EG-LDPC degrades very slowly with the increase in the number of errors, but the BCH quickly reduces the correction capacity from three errors.

Table 7 presents the synthesis costs of the six ECCs for the same 65 nm CMOS technology. Additionally, the last column of Table 7 shows the redundancy rate (RR) for each code, given the number of

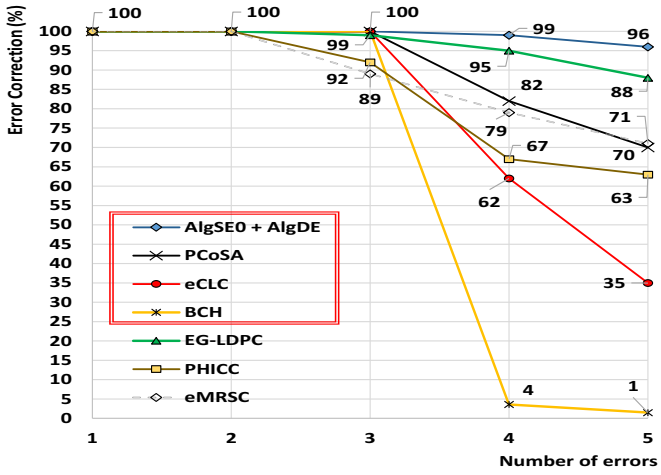

Fig. 21. Analysis of the error correction capacity of seven 2D ECCs; the red double-bordered rectangle surrounds ECCs that achieve 100% correction with up to three errors. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 7
Decoder implementation costs in hardware and redundancy rate of six 2D ECCs.

ECC	Area consumption		Power dissipation		Delay		RR (%)
	(μm²)	(%)	(mW)	(%)	(ns)	(%)	
AlgSE0+AlgDE	4,942	49.2	0.550	64.9	3.58	187.4	67
BCH	3,302	32.9	NA	NA	1.07	56.0	38.5
eCLC	3,360	33.4	0.331	39.0	2.50	130.9	60
eMRSC	1,709	17.0	0.374	44.1	1.54	80.6	50
PCoSA	10,051	100.0	0.848	100.0	1.91	100	75
PHICC	1,761	17.5	0.344	40.6	0.96	50.3	60

NA – not available data; Cells marked in green and red have the best and worst results, respectively.

redundancy bits compared to the total number of codeword bits. We use the work of Ma, Cui, and Lee [60] to calculate the BCH data, scaling the 90 nm to 65 nm CMOS technology with the estimate provided by DeepScaleTool [61].

Table 7 shows that LPC surpasses PCoSA in almost all the items evaluated in this work, except for the delay caused by executing two algorithms in series. However, the low complexity of the other ECC algorithms implies gains in all synthesis aspects compared to AlgSE0 + AlgDE.

9. Conclusions

This work shows that LPC allows exploring techniques in several axes, attaining high error correction efficacy with low synthesis costs. We proposed implementing the LPC decoder with two consecutive algorithms, which correct single errors (AlgSE) and double errors (AlgDE).

We explored three research axes with AlgSE: (i) the number of iterations of the algorithm, (ii) the use of a heuristic to choose the row/column or column/row correction order, and (iii) the decision of the region and the way to correct errors. The results showed a slight gain from the second iteration, with no improvement observed with more than four iterations. The FairPriorityLoop heuristic, which always corrects the row/column pair at each iteration, prioritizes the one with more SEs, showed greater effectiveness without additional implementation cost. The DCOC technique that checks for errors and corrects only the data has achieved the highest efficacy, in general, standing behind DCO, which does not check for errors, only when the number of errors is in the range 4 to 8.

AlgDE is an innovative algorithm based on the inference of errors by crossing rows and columns; it allows increasing the efficiency of the LPC decoder significantly when used in conjunction with AlgSE. Finally, the comparative results show that when implemented with AlgSE0 + AlgDE, the LPC decoder is much more effective than the other ECCs evaluated here, although the synthesis costs penalize it.

CRediT authorship contribution statement

All persons who meet authorship criteria are listed as authors, and all authors certify that they have participated sufficiently in the work to take public responsibility for the content, including participation in the concept, design, analysis, writing, or revision of the manuscript. Furthermore, each author certifies that this material or similar material has not been and will not be submitted to or published in any other publication.

Declaration of competing interest

The authors whose names are listed immediately below certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

References

[1] O. Brand, Microsensor integration into systems-on-chip, Proc. IEEE 94 (6) (Jul. 2006) 1160–1176.
[2] J. Knight, Safety critical systems: challenges and directions, in: Proceedings of the International Conference on Software Engineering (ICSE), 2002, pp. 547–550.
[3] S. Dhia, M. Ramdani, E. Sicard, Electromagnetic Compatibility of Integrated Circuits, Springer, 2006, 473p.
[4] G. Kinoshita, C. Kleiner, E. Johnson, Radiation induced regeneration through the P-N junction isolation in monolithic I/C's, IEEE Trans. Nucl. Sci. 12 (5) (Oct. 1965) 83–90.
[5] C. Kleiner, G. Kinoshita, E. Johnson, Simulation and verification of transient nuclear radiation effects on semiconductor electronics, IEEE Trans. Nucl. Sci. 11 (5) (Nov. 1964) 82–104.
[6] C. Rosenberg, D. Gage, R. Caldwell, G. Hanson, Charge-control equivalent circuit for predicting transient radiation effects in transistors, IEEE Trans. Nucl. Sci. 10 (5) (Nov. 1963) 149–158.
[7] T. Li, H. Liu, H. Yang, Design and characterization of SEU hardened circuits for SRAM-based FPGA, IEEE Trans. Very Large Scale Integr. Syst. 27 (6) (Jun. 2019) 1276–1283.
[8] W. Wei, K. Namba, Y. Kim, F. Lombardi, A novel scheme for tolerating single Event/Multiple bit upsets (SEU/MBU) in non-volatile memories, IEEE Trans. Comput. 65 (3) (Mar. 2016) 781–790.
[9] I. Villalta, U. Bidarte, J. Cornejo, J. Lázaro, A. Astarloa, Estimating the SEU failure rate of designs implemented in FPGAs in presence of MCUs, Microelectron. Reliab. 78 (Nov. 2017) 85–92.
[10] A. Neale, M. Jonkman, M. Sachdev, Adjacent-MBU-tolerant SEC-DED-TAEC-yAED codes for embedded SRAMs, IEEE Trans. Circuits Syst. Express Briefs 62 (4) (Apr. 2015) 387–391.
[11] S. Chandrashekhar, H. Puchner, J. Mitani, S. Shinozaki, M. Sardi, D. Hoffman, Radiation induced soft errors in 16 nm floating gate SLC NAND flash memory, Microelectron. Reliab. 108 (May 2020) 1–8.
[12] A. Cóbreces, A. Regadio, J. Tabero, P. Reviriego, A. Macian, J. Maestro, SEU and SEFI error detection and correction on a ddr3 memory system, Microelectron. Reliab. 91 (1) (Dec. 2018) 23–30.
[13] M. Cannon, A. Keller, H. Rowberry, C. Thurlow, A. Celis, M. Wirthlin, Strategies for removing common mode failures from TMR designs deployed on SRAM FPGAs, IEEE Trans. Nucl. Sci. 66 (1) (Jan. 2019) 207–215.
[14] X. She, N. Li, Reducing critical configuration bits via partial TMR for SEU mitigation in FPGAs, IEEE Trans. Nucl. Sci. 64 (10) (Oct. 2017) 2626–2632.
[15] A. Erozan, E. Çavuş, An EG-LDPC based 2-dimensional error correcting code for mitigating MBUs of SRAM memories, in: Proceedings of the FPGAworld Conference (FPGAworld), 2015, pp. 21–26.
[16] H. Farbeh, F. Mozafari, M. Zabihi, S. Miremadi, RAW-tag: replicating in altered cache ways for correcting multiple-bit errors in tag array, IEEE Trans. Dependable Secure Comput. 16 (4) (July-Aug. 2019) 651–664, https://doi.org/10.1109/TDSC.2017.2706263.
[17] A. Olazábal, J. Guerra, Multiple cell upsets inside aircrafts. New fault-tolerant architecture, IEEE Trans. Aerosp. Electron. Syst. 55 (1) (Feb. 2019) 332–342, https://doi.org/10.1109/TAES.2018.2852198.
[18] J. Gracia-Morán, L. Saiz-Adalid, D. Gil-Tomás, P. Gil-Vicente, Improving error correction codes for multiple-cell upsets in space applications, IEEE Trans. Very Large Scale Integr. VLSI Syst. 26 (10) (Oct. 2018) 2132–2142.
[19] D. Freitas, D. Mota, R. Goerl, C. Marcon, F. Vargas, J. Silveira, J. Mota, PCoSA: a product error correction code for use in memory devices targeting space applications, Integr. VLSI J. (May 2020) 1–10, on-line version.
[20] F. Silva, W. Freitas, J. Silveira, O. Lima, F. Vargas, C. Marcon, An efficient, low-cost ECC approach for critical-application memories, in: Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI), 2017, pp. 198–203.
[21] F. Silva, W. Freitas, J. Silveira, C. Marcon, F. Vargas, Extended matrix region selection code: an ECC for adjacent multiple cell upset in memory arrays, Microelectron. Reliab. 106 (Mar. 2020) 1–9.
[22] H. Castro, J. da Silveira, A. Coelho, F. Silva, P. Magalhães, O. Lima Jr., A correction code for multiple cells upsets in memory devices for space applications, in: Proceedings of the IEEE International New Circuits and Systems Conference (NEWCAS), 2016, pp. 1–4.
[23] F. Silva, J. Silveira, J. Silveira, C. Marcon, F. Vargas, O. Lima Jr., An extensible code for correcting multiple cell upset in memory arrays, J. Electron. Test. 34 (Jul. 2018) 417–433.
[24] P. Magalhães, O. Alcântara, J. Silveira, PHICC: an error correction code for memory devices, in: Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI), 2019, p. 1.
[25] P. Elias, Error-free coding, in: Transactions of the IRE Professional Group on Information Theory vol. 4, Sep. 1954, pp. 29–37.
[26] K. Alexey, Z. Victor, R. Eygene, A new iterative decoder for product codes, in: Proceedings of the International Workshop on Algebraic and Combinational Coding Theory (ACCT), 2014, pp. 211–214.

[27] S. Changuel, R. Bidan, R. Pyndiah, Iterative decoding of block turbo codes over the binary erasure channel, in: Proceedings of the IEEE International Conference on Signal Processing and Communications (ICSPC), 2008, pp. 1539–1542.
[28] Z. Li, M. Miao, Z. Wang, Parallel coding scheme with turbo product code for Mobile multimedia transmission in MIMO-FBMC system, IEEE Access 8 (Dec. 2019) 3772–3780.
[29] X. Zhou, R. Li, A parallel turbo product codes decoder based on graphics processing units, in: Proceedings of the International Conference on High Performance Computing and Communications (HPCC/SmartCity/DSS), 2019, pp. 337–344.
[30] L. Lopacinski, J. Nolte, S. Buechner, M. Brzozowski, R. Kraemer, Improved turbo product coding dedicated for 100 Gbps wireless terahertz communication, in: Proceedings of the IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016, pp. 1–6.
[31] R. Swaminathan, A. Madhukumar, W. Guohua, Blind estimation of code parameters for product codes over Noisy Channel conditions, IEEE Trans. Aerosp. Electron. Syst. 56 (2) (Aug. 2019) 1460–1473.
[32] C. Senger, Improved iterative decoding of product codes based on trusted symbols, in: Proceedings of the IEEE International Symposium on Information Theory (ISIT), 2019, pp. 1342–1346.
[33] J. Guo, L. Xiao, Z. Mao, Q. Zhao, Enhanced memory reliability against multiple cell upsets using decimal matrix code, IEEE Trans. Very Large Scale Integr. VLSI Syst. 22 (1) (Jan. 2014) 127–135.
[34] S. Khittiwitchayakul, W. Phakphisut, P. Supnithi, Weighted bit-flipping decoding for product LDPC codes, in: Proceedings of the International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2016, pp. 132–134.
[35] S. Jeong, J. Lee, Iterative LDPC–LDPC product code for bit patterned media, IEEE Trans. Magn. 53 (3) (Mar. 2017) 1–4.
[36] S. Jeong, J. Lee, Iterative Channel detection with LDPC product code for bit-patterned media recording, IEEE Trans. Magn. 53 (11) (Apr. 2017) 1–4.
[37] S. Arslan, J. Lee, J. Hodges, J. Peng, H. Le, T. Goker, MDS product code performance estimations under header CRC check failures and missing syncs, IEEE Trans. Device Mater. Reliab. 14 (3) (Sep. 2014).
[38] A. Sheikh, A. Amat, G. Liva, Binary message passing decoding of product-like codes, IEEE Trans. Commun. 67 (12) (Dec. 2019) 8167–8178.
[39] W. Li, J. Lin, Z. Wang, Improved soft-assisted iterative bounded distance decoding for product codes, in: Proceedings of the IEEE International Conference on Computer and Communication (ICCC), 2020, pp. 710–714.
[40] M. Coskun, T. Jerkovits, G. Liva, Successive cancellation list decoding of product codes with reed-muller component codes, IEEE Commun. Lett. 23 (11) (Aug. 2019) 1972–1976.
[41] A. Sheikh, A. Amat, G. Liva, C. Hager, H. Pfister, On low-complexity decoding of product codes for high-throughput fiber-optic systems, in: Proceedings of the IEEE International Symposium on Turbo Codes & Iterative Information Processing (ISTC), 2019, pp. 1–5.
[42] A. Sheikh, A. Amat, G. Liva, Binary message passing decoding of product codes based on generalized minimum distance decoding, in: Proceedings of the Annual Conference on Information Sciences and Systems (CISS), 2019, pp. 1–5.
[43] S. Liu, L. Xiao, J. Guo, Z. Mao, Fault secure encoder and decoder designs for matrix codes, in: Proceedings of the International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics), 2016, pp. 181–185.
[44] S. Li, J. Li, Y. Zhou, Z. Mao, Low redundancy matrix-based codes for adjacent error correction with parity sharing, in: Proceedings of the International Symposium on Quality Electronic Design (ISQED), 2017, pp. 76–80.
[45] H. Tawfeek, A. Mahran, G. Abdel-Hamid, A reliability-based stopping criterion for turbo product codes, in: Proceedings of the International Computer Engineering Conference (ICENCO), 2019, pp. 141–145.
[46] C. Condo, V. Bioglio, H. Hafermann, I. Land, Practical product code construction of polar codes, IEEE Trans. Signal Process. 68 (Mar. 2020) 2004–2014.
[47] C. Yang, D. Muckatira, A. Kulkarni, C. Chakrabarti, Data storage time sensitive ECC schemes for MLC NAND flash memories, in: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 2513–2517.
[48] C. Argyrides, H. Zarandi, D. Pradhan, Matrix codes: multiple bit upsets tolerant method for SRAM memories, in: Proceedings of the IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT), 2007, pp. 340–348, https://doi.org/10.1109/DFT.2007.29.
[49] C. Argyrides, D. Pradhan, T. Kocak, Matrix codes for reliable and cost-efficient memory chips, IEEE Trans. Very Large Scale Integr. VLSI Syst. 19 (3) (Mar. 2011) 420–428, https://doi.org/10.1109/TVLSI.2009.2036362.
[50] R. Hamming, Error detecting and error correcting codes, Bell Syst. Tech. J. 29 (2) (Apr. 1950) 147–160.
[51] F. Macwilliams, N. Sloane, in: The Theory of Error-Correcting Codes, 3rd ed. vol. 16, North-Holland, 1977, pp. 568–570.
[52] S. Lin, D.J. Costello, in: Error Control Coding: Fundamentals and Applications, 1st ed. vol. 1, Prentice-Hall, 1983, pp. 67–68.
[53] T. Moon, in: Error Correcting Code – Mathematical Methods, Algorithms, 1st ed. vol. 1, Wiley, 2005, pp. 430–432.
[54] R. Zaragoza, in: The Art of Error Correcting Coding, 2nd ed., Wiley, 2006, pp. 170–201.
[55] R. Tocci, N. Widmer, G. Moss, in: Digital Systems – Principles, Applications, 10th ed., Pearson, 2007, pp. 41–46.
[56] S. Esposito, C. Albanese, M. Alderighi, F. Casini, L. Giganti, M. Esposti, C. Monteleone, M. Violante, COTS-based high-performance computing for space applications, IEEE Trans. Nucl. Sci. 62 (6) (Dec. 2015) 2687–2694.
[57] A. Agnesina, A. Sidana, J. Yamaguchi, C. Krutzik, J. Carson, J. Scharlotta, S. Lim, A novel 3D DRAM memory cube architecture for space applications, in: Proceedings of the ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1–6, on-line version.
[58] D. Shim, A. Sidana, J. Yamaguchi, C. Krutzik, D. Nakamura, S. Lim, FLASHRAD: a reliable 3D rad hard flash memory cube utilizing COTS for space, in: Proceedings of the IEEE Aerospace Conference, 2019, pp. 1–8.
[59] A. Agnesia, J. Yamaguchi, C. Krutzik, J. Carson, J. Scharlotta, S. Lim, Bringing 3D COTS DRAM memory cubes to space, in: Proceedings of the IEEE Aerospace Conference, 2019, pp. 1–11.
[60] W. Ma, X. Cui, C.-L. Lee, Enhanced error correction against multiple-bit-upset based on BCH code for SRAM, in: Proceedings of the IEEE International Conference on ASIC, 2013, pp. 1–4.
[61] S. Sarangi, B. Baas, DeepScaleTool: a tool for the accurate estimation of technology scaling in the deep-submicron era, in: Proceedings of the IEEE International Symposium on Circuits and Systems, 2021, pp. 1–5.