electronics

MDPI

Article

Reducing the Overhead of BCH Codes: New Double Error Correction Codes

Luis-J. Saiz-Adalid *, Joaquín Gracia-Morán, Daniel Gil-Tomás, J.-Carlos Baraza-Calvo and Pedro-J. Gil-Vicente

Institute of Information and Communication Technologies (ITACA), Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain; jgracia@itaca.upv.es (J.G.-M.); dgil@itaca.upv.es (D.G.-T.); jcbaraza@itaca.upv.es (J.-C.B.-C.); pgil@itaca.upv.es (P.-J.G.-V.)

* Correspondence: ljsaiz@itaca.upv.es

Received: 15 October 2020; Accepted: 7 November 2020; Published: 11 November 2020

Abstract: The Bose-Chaudhuri-Hocquenghem (BCH) codes are a well-known class of powerful error correction cyclic codes. BCH codes can correct multiple errors with minimal redundancy. Primitive BCH codes only exist for some word lengths, which do not frequently match those employed in digital systems. This paper focuses on double error correction (DEC) codes for word lengths that are in powers of two (8, 16, 32, and 64), which are commonly used in memories. We also focus on hardware implementations of the encoder and decoder circuits for very fast operations. This work proposes new low redundancy and reduced overhead (LRRO) DEC codes, with the same redundancy as the equivalent BCH DEC codes, but whose encoder, and decoder circuits present a lower overhead (in terms of propagation delay, silicon area usage and power consumption). We used a methodology to search parity check matrices, based on error patterns, in order to design the new codes. We implemented and synthesized them, and compared their results with those obtained for the BCH codes. Our implementation of the decoder circuits achieved reductions between 2.8% and 8.7% in the propagation delay, between 1.3% and 3.0% in the silicon area, and between 15.7% and 26.9% in the power consumption. Therefore, we propose LRRO codes as an alternative for protecting information against multiple errors.

Keywords: reliability; fault tolerance; error control codes; double error correction; BCH codes

1. Introduction

Error control codes (ECCs) are frequently employed for fault tolerance in dependable systems. Coding theory has been studied for over half a century, and it is still going stronger than ever [1,2], because of its extensive application in computing, data storage, communications, etc.

The Bose-Chaudhuri-Hocquenghem (BCH) codes are one of the best-known ECCs. They were discovered in 1959 by Hocquenghem [3] and, independently, in 1960 by Bose and Ray-Chaudhuri [4]. Since then, BCH codes have been extensively employed in several applications, namely: flash memories in solid-state drives (SSDs) [5], optical storage like compact disks (CDs) or digital versatile disks (DVDs) [6], ethernet [7], video codecs [8], digital video broadcasting (DVB) [9], and satellite communications [10]. Their algebraic features make BCH codes very useful in a wide range of situations, and their error coverage is achieved with minimum redundancy.

Nevertheless, when applied to computers, these codes have two main weaknesses. First, primitive BCH codes (those generated according to their strict definition) only exist for a limited number of word lengths [11]. This problem can be easily solved by shortening a longer code. However, although they maintain their error coverage, shortened BCH codes may lose other properties, such as cyclicity or minimal redundancy, as stated later.

The second problem with these codes is the latency of the decoding process. Although their algebraic structure is useful in simplifying the encoding and decoding procedures, most of the decoding algorithms have a sequential structure that requires several clock cycles to complete the decoding [11]. Commonly, this is not a problem in software implementation or even in hardware when the speed requirements are not very high. However, the usefulness of BCH codes is limited when very fast encoding and decoding operations are required.

The information stored in key elements of a computer system, such as registers and memories, may be perturbed by different physical mechanisms [12–14]. As technology increases in integration scale, single error correction (SEC) or single error correction-double error detection (SEC-DED) codes may not be enough for present and future computers. Multiple error correction codes, like BCH, become an interesting alternative, but designers have to deal with the two problems described previously. First, the lengths of data words are commonly in a power of two (8, 16, 32, etc.), and they do not match to the block sizes of primitive BCH codes. Second, their "slow" decoding may reduce the system performance.

This paper focuses on binary codes for double error correction (DEC), applied to common data word lengths in computers (i.e., 8, 16, 32, and 64). We propose new low redundancy and reduced overhead (LRRO) DEC codes, which maintain the same error coverage and redundancy as the equivalent BCH codes, but reduce the overhead introduced by the encoder and decoder circuits in terms of the propagation delay, silicon area, and power consumption. Different encoder and decoder circuits have been implemented and synthesized in order to validate the error coverage and to measure and compare those parameters. The results validate the improvements achieved by our proposal, and confirm LRRO codes as an interesting alternative to BCH codes in high-speed applications, like random access memories (RAM) and processor registers. These codes are especially well suited for RAM memories, where low redundancy, high-speed operations, and low power consumption are mandatory.

This paper is organized as follows. Section 2 introduces basic concepts about coding theory and BCH codes. The proposed LRRO codes are described in Section 3. Section 4 includes the evaluation of the proposal and a comparison with the BCH codes. Finally, Section 5 presents some conclusions and ideas for future work.

2. Coding Theory and BCH Codes

2.1. Basics on Error Control Coding

An (n, k) binary linear block ECC encodes a k-bit input word in an n-bit output word [15]. The input word, $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$, is a k-bit vector that represents the original data. The code word, $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$, is an n-bit vector, where the $(n - k)$ added bits are called the parity, code, or redundant bits. $\mathbf{b}$ is transmitted through an unreliable channel that delivers the received word, $\mathbf{r} = (r_0, r_1, \ldots, r_{n-1})$. The error vector, $\mathbf{e} = (e_0, e_1, \ldots, e_{n-1})$, models the error induced by the channel. If no error has occurred in the i-th bit, then $e_i = 0$; otherwise, $e_i = 1$. Therefore, $\mathbf{r}$ can be interpreted as $\mathbf{r} = \mathbf{b} \oplus \mathbf{e}$. Figure 1 synthesizes this encoding, channel crossing, and syndrome decoding processes.

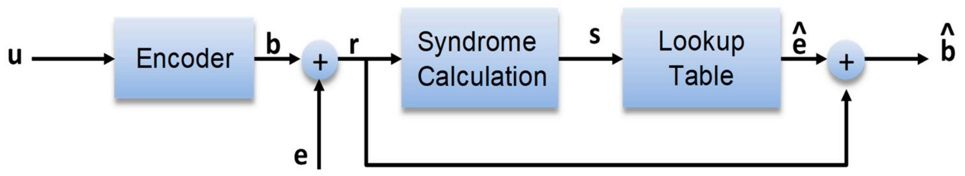


Figure 1. Encoding, channel crossing, and syndrome decoding processes.

The generator matrix, $\mathbf{G}_{k \times n}$, of a linear code (together with its related parity check matrix, $\mathbf{H}_{(n-k) \times n}$) defines the code [1]. For the encoding process, $\mathbf{b} = \mathbf{u} \cdot \mathbf{G}$. The encoded word, $\mathbf{b}$, meets the requirement

$\mathbf{H}\cdot\mathbf{b}^{\mathrm{T}} = \mathbf{0}$, which means that it is a correct code word. For syndrome decoding, syndrome is defined as $\mathbf{s}^{\mathrm{T}} = \mathbf{H}\cdot\mathbf{r}^{\mathrm{T}}$, and it exclusively depends on **e**, as follows:

$$\mathbf{s}^{\mathrm{T}} = \mathbf{H}\cdot\mathbf{r}^{\mathrm{T}} = \mathbf{H}\cdot(\mathbf{b} \oplus \mathbf{e})^{\mathrm{T}} = \mathbf{H}\cdot\mathbf{b}^{\mathrm{T}} \oplus \mathbf{H}\cdot\mathbf{e}^{\mathrm{T}} = \mathbf{H}\cdot\mathbf{e}^{\mathrm{T}}. \tag{1}$$

There must be a different syndrome, **s**, for each correctable error vector, **e**. Syndrome decoding is done through a lookup table that relates each syndrome to the decoded error vector, $\hat{\mathbf{e}}$. If $\mathbf{s} = \mathbf{0}$, we can assume that $\hat{\mathbf{e}} = \mathbf{0}$, and hence **r** is correct. Otherwise, an error has occurred. The decoded code word, $\hat{\mathbf{b}}$, is calculated as $\hat{\mathbf{b}} = \mathbf{r} \oplus \hat{\mathbf{e}}$. From $\hat{\mathbf{b}}$, it is easy to obtain $\hat{\mathbf{u}}$ by discarding the parity bits. If the fault hypothesis used to design the ECC is consistent with the behavior of the channel, $\hat{\mathbf{u}}$ and **u** must be equal with a very high probability.

For a binary word, the term Hamming weight, w, denotes the number of ones in that word. As explained later, the Hamming weights of the rows and columns of the parity check matrix determine the complexity of a code.

The Hamming distance between two binary words is the number of bits in which they differ. The minimum Hamming distance of a code (d_{min}) is the minimum of the distances between all pairs of valid code words. This parameter determines the error coverage of a code.

Code shortening is a code construction where, starting from a longer linear block code, the number of data bits is reduced, while maintaining the same number of parity bits [15]. In this way, we can construct a code for shorter data words, maintaining the same Hamming distance, and, therefore, the same error coverage.

As stated above, the encoding process in linear block codes can be easily implemented. However, the decoding process (mainly the lookup table implementation) may be difficult. Cyclic codes form an important subclass of linear block codes [11]. Encoding and syndrome computation can be implemented easily using linear feedback shift register circuits operated sequentially. Because of their algebraic structure, there are different practical methods for decoding.

Cyclic codes are linear block codes with the additional property that, for each code word, all cyclically shifted words are also valid code words.

Polynomial representation [11] is frequently employed to work with these codes. For example, the input word, **u**, can be represented as a $(k - 1)$ or lower degree polynomial with the variable X, where the power of X is used to locate the bit u_i in the following word:

$$\mathbf{u}(X) = u_{k-1} X^{k-1} + u_{k-2} X^{k-2} + \ldots + u_1 X + u_0 \tag{2}$$

Cyclic codes can be represented using a generator polynomial, $\mathbf{g}(X)$. The data words, $\mathbf{b}(X)$, are computed as $\mathbf{b}(X) = \mathbf{u}(X)\cdot\mathbf{g}(X)$. A parity check polynomial can be obtained from the generator polynomial. In fact, $\mathbf{h}(X)\cdot\mathbf{g}(X) = X^n + 1$ [11]. The generator and parity check matrices can be obtained from these polynomials.

BCH codes are one of the best-known cyclic codes. Reed–Solomon codes are the most important subclass of non-binary BCH codes [11]. However, we will focus on binary BCH codes from now on.

2.2. Binary BCH Codes

For any positive integers where $m \geq 3$ and $t < 2^{m-1}$, there exists a binary BCH code with code word length, $n = 2^m - 1$; number of parity bits, $(n - k) \leq mt$; and minimum distance $d_{min} \geq 2t + 1$. Therefore, this code can correct t errors [11].

For a t-error correction BCH code, its generator polynomial is given by $\mathbf{g}(X) = \mathbf{m}_1(X) \times \mathbf{m}_3(X) \times \ldots \times \mathbf{m}_{2t-1}(X)$, where $\mathbf{m}_i(X)$ is the minimal polynomial of α^i, $(i = 1, 3, \ldots, 2t–1)$ and α is an element of GF(2^m) of order n. GF refers to Galois fields (or finite fields). If $\mathbf{m}_1(X)$ is a primitive polynomial of degree m over GF (2), the code is called a primitive binary BCH code. Then, α is a primitive element and its order is $n = 2^m - 1$. The minimal polynomials $\mathbf{m}_i(X)$ and/or the generator polynomials for binary primitive BCH codes can be found in the literature [11]. More information about algebra for coding theory can be found, for example, in the literature [1,11,15].

There are different algorithms to calculate the error-location polynomial (the key step of the decoding): Berlekamp–Massey (mainly used to implement software decoders), Euclidean (mostly employed in hardware implementations), etc. [11,16]. These methods are sequential algorithms with iterative steps. Therefore, they require several clock cycles to perform the decoding process.

A combinational decoder (sometimes referred to as a parallel decoder) is proposed in the literature [17]. Because of its importance in this work, it is briefly explained in Section 2.4. It is first necessary to understand how to obtain the generator and parity check matrices from the equivalent polynomials.

2.3. Matrices and Polynomials

As an example, let us consider the binary BCH code with $m = 3$ and $t = 1$. It is a (7, 4) BCH SEC code. It is equivalent to the cyclic Hamming code. Its generator polynomial is $\mathbf{g}(X) = X^3 + X + 1$ [11]. As stated in Section 2.1, $\mathbf{h}(X) = (X^7 + 1)/\mathbf{g}(X) = X^4 + X^2 + X + 1$.

Combinational encoders and decoders may require the equivalent generator and parity check matrices. The binary representation of the generator polynomial is 1011. Arranging and shifting this word in the matrix, we obtain the following:

$$\mathbf{G} = \begin{bmatrix} \mathbf{1011}000 \\ 0\mathbf{1011}00 \\ 00\mathbf{1011}0 \\ 000\mathbf{1011} \end{bmatrix}. \quad (3)$$

As it can be observed, it is not in a systematic form. Obtaining the remainder by dividing X^{n-1}, X^{n-2}, …, X^{n-k} by the generator polynomial, we get the matrix in a systematic form [16]:

$$\mathbf{G_s} = \begin{bmatrix} \mathbf{1000}101 \\ \mathbf{0100}111 \\ \mathbf{0010}110 \\ \mathbf{0001}011 \end{bmatrix}. \quad (4)$$

The same procedure can be followed to obtain the parity check matrix:

$$\mathbf{H} = \begin{bmatrix} \mathbf{10111}00 \\ 0\mathbf{10111}0 \\ 00\mathbf{10111} \end{bmatrix} \quad \rightarrow \quad \mathbf{H_s} = \begin{bmatrix} \mathbf{100}1011 \\ \mathbf{010}1110 \\ \mathbf{001}0111 \end{bmatrix}. \quad (5)$$

This method is applied to BCH DEC codes in the literature [17], as presented in the following.

2.4. Combinational Circuits for BCH Codes

The work presented in the literature [17] proposes combinational encoder and decoder circuits for BCH DEC codes for SRAM protection. As stated before, BCH DEC codes have not found favorable application in SRAMs because of the non-alignment of their block sizes to typical memory word lengths, and particularly because of the large multi-cycle latency of traditional iterative decoding algorithms. The authors propose a solution based on the generator and parity check matrices. Shortened codes can be easily obtained from the matrices of the primitive codes. The encoder circuit can be simply designed from the generator matrix. Syndrome computation (the first step of the decoding process) is generated using the parity check matrix.

The key element in this proposal is the error pattern decoder, equivalent to the lookup table shown in Figure 1. Combinational logic is employed to map the syndromes for correctable error patterns. This mapping is precomputed by multiplying all correctable error patterns with the parity check matrix, **H**.

Once the estimated error vector is determined, an erroneous bit is corrected by complementing it; hence, the error corrector circuit (final step of the decoding process) is simply a stack of XOR gates.

The authors of this work implemented BCH DEC codes for 16, 32, and 64 bits, and synthesized them using a 90-nm standard cell library. They reported, for example, decoding latencies ranging from 1.4 ns (for 16 bits) to 2.2 ns (for 64 bits).

3. Low Redundancy and Reduced Overhead (LRRO) Double Error Correction Codes

The Hamming weight for a row of a parity check matrix determines the number of terms to be XORed in order to calculate the parity and syndrome bits. Therefore, in hardware implementation, the heaviest row defines the logic depth of the XOR tree, which implements the encoder circuit and the syndrome computation in the decoder circuit. It influences the delay introduced by the ECC.

In the same way, the Hamming weight of the whole parity check matrix determines the number of logic gates required in the encoder circuit and the syndrome computation in the decoder circuit. This number has an important influence on both the silicon area occupied and the power consumed by those circuits.

Finally, the complexity of the ECC is mainly determined by its error coverage, and affects all overheads (delay, silicon area, and power consumption). Nevertheless, as the error coverage is determined by the design, nothing can be done about this parameter.

Therefore, to reduce the overhead introduced by the encoder and decoder circuits, we must focus on the Hamming weights of the heaviest row and the whole matrix. Frequently, ECC designers can achieve these objectives by increasing the number of parity bits. Depending on the application, it could be an interesting alternative (e.g., [18]). However, this is not a good idea when protecting memories, as all redundant bits must be stored per individual word in the whole memory, leading to a much higher silicon area being occupied and more power being consumed by the memory circuitry.

Is it possible to find ECCs with the same redundancy as BCH codes, but reducing the overhead introduced? For thid, we employed a searching methodology based on the errors to be corrected and/or detected [19]. Although this methodology was initially employed to design flexible unequal error control (FUEC) codes, it has been successfully used to design different families of codes (e.g., [18,20,21]). First, let us briefly describe this methodology. Later, we describe the procedure needed in order to obtain matrices for the existing BCH DEC codes. Next, we present the new LRRO DEC codes. Finally, some important considerations have been included at the end of this section.

3.1. Searching Parity Check Matrices

Searching a parity check matrix that achieves the required error coverage may end up being very complex. We used a methodology based on searching matrices that can correct and/or detect a given set of error vectors. This methodology was first presented in the literature [19] to design flexible unequal error control (FUEC) codes. Although a detailed explanation of the methodology is out of the scope of this paper, it is briefly summarized in Figure 2, and is described in the following.

After determining the values of n and k for the code to be designed, the error patterns to be corrected must be selected. These error patterns can be represented using error vectors, according to the definition given in Section 2.1. Then, the parity check matrix, **H**, that satisfies (6) is searched, where $\boldsymbol{E}_+$ represents the set of error vectors to be corrected.

$$\mathbf{H}\cdot\mathbf{e}_i^T \neq \mathbf{H}\cdot\mathbf{e}_j^T; \forall \mathbf{e}_i, \mathbf{e}_j \in \boldsymbol{E}_+ \mid \mathbf{e}_i \neq \mathbf{e}_j, \tag{6}$$

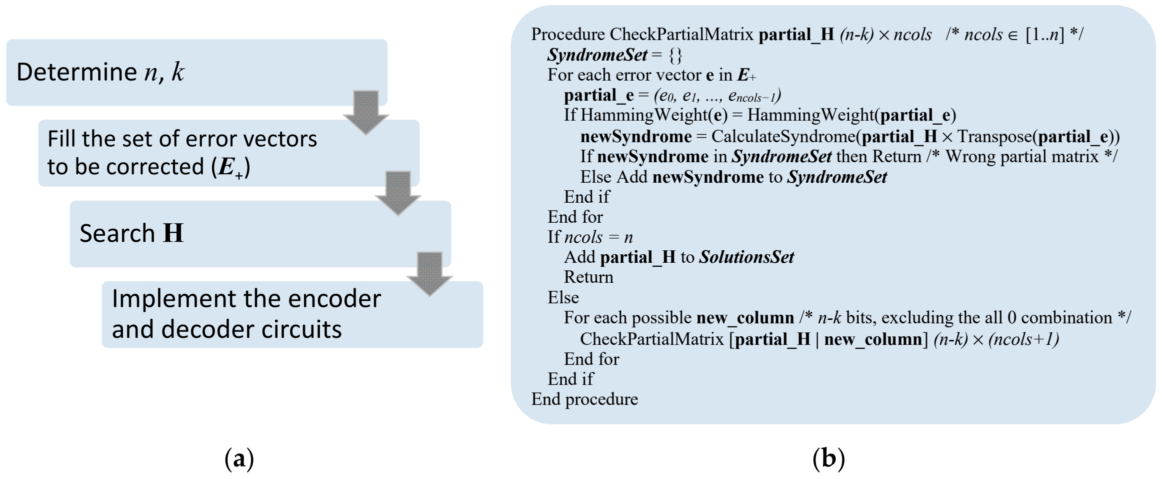
that is, each correctable error must have a different syndrome. In this case, $\boldsymbol{E}_+$ must include the vectors for the single and double errors. Single errors are represented with vectors (...1...), and double errors with vectors (...1...1...), where the dots represent zero or more zeroes.

To find the matrix, the recursive backtracking algorithm shown in Figure 2b is used. It checks partial matrices and adds a new column only if the previous matrix satisfies the requirements. The added columns must be non-zero, so there are $2^{n-k} - 1$ combinations for each column. In order to find matrices with a low Hamming weight, columns with a lower number of ones are checked first.

Once the **H** matrix is selected, it is easy to determine the logic equations in order to calculate each parity and syndrome bit, as well as the syndrome lookup table. They are required for encoder and decoder implementation.

In addition, we can improve the matrix generation so as to reduce the number of 1 s in those rows with a higher number of 1 s of the parity check matrix, as well as reducing the total number of 1 s in the whole matrix. As said before, these reductions will lead to faster, smaller, and less power-consuming circuits.

A detailed explanation of this algorithm, as well as a code design example, can be found in the literature [19].



Determine n, k
Fill the set of error vectors to be corrected (E_+)
Search **H**
Implement the encoder and decoder circuits
(**a**)
(**b**)

```
Procedure CheckPartialMatrix partial_H $(n-k) \times ncols$   /* $ncols \in [1..n]$ */
   SyndromeSet = {}
   For each error vector e in $E_+$
      partial_e = $(e_0, e_1, \ldots, e_{ncols-1})$
      If HammingWeight(e) = HammingWeight(partial_e)
         newSyndrome = CalculateSyndrome(partial_H × Transpose(partial_e))
         If newSyndrome in SyndromeSet then Return /* Wrong partial matrix */
         Else Add newSyndrome to SyndromeSet
      End if
   End for
   If $ncols = n$
      Add partial_H to SolutionsSet
      Return
   Else
      For each possible new_column /* $n-k$ bits, excluding the all 0 combination */
         CheckPartialMatrix [partial_H | new_column] $(n-k) \times (ncols+1)$
      End for
   End if
End procedure
```

Figure 2. Searching parity check matrices methodology based on the set of error patterns to be corrected: (**a**) flow chart and (**b**) recursive backtracking algorithm (extracted from [19]).

3.2. BCH DEC Codes for 8, 16, 32, and 64 Bits

As stated above, in most cases, the block sizes of the primitive BCH codes do not align with the sizes employed in memories. Hence, if we want to use BCH DEC codes with data lengths of 8, 16, 32, or 64 bits, they must be shortened from primitive BCH DEC codes of longer data lengths. As described in Section 2.1, code shortening allows for the design of codes with any block size using longer block size codes. The primitive BCH DEC codes required for our purpose are (31, 21), (63, 51), and (127, 113). From the (31, 21) code, we can obtain (18, 8) and (26, 16) BCH DEC codes. From the (63, 51) code, a (44, 32) code can be designed. The (127, 113) code allows for the construction of a (78, 64) BCH DEC code.

As an example, let us consider the primitive (31, 21) BCH DEC code. Its generator polynomial, obtained by multiplying two primitive polynomials [11], is as follows:

$$\mathbf{g}(X) = (X^5 + X^2 + 1)\cdot(X^5 + X^4 + X^3 + X^2 + 1) = (X^{10} + X^9 + X^8 + X^6 + X^5 + X^3 + 1). \tag{7}$$

Following the steps described in Section 2.3 to get the matrices in a systematic form, and shortening the code as described in the literature [17], the parity check matrix for the (26, 16) BCH DEC code can be obtained as follows:

$$\mathbf{H}_{\mathbf{BCH\ 26,16}} = \begin{bmatrix} 10000000001101010111100100 \\ 01000000000110101011110010 \\ 00100000000011010101111001 \\ 00010000001100111101011000 \\ 00001000000110011110101100 \\ 00000100001110011000110010 \\ 00000010001010011011111101 \\ 00000001000101001101111110 \\ 00000000101111110001011011 \\ 00000000011010101111001001 \end{bmatrix}. \tag{8}$$

In the same way, the matrices for all of the aforementioned codes have been obtained. These matrices and their resulting encoding and decoding circuits will be compared with our proposal in Section 4.

3.3. LRRO DEC Codes for 8, 16, 32, and 64 Bits

In this paper, we propose new low redundancy and reduced overhead (LRRO) codes for double error correction, equivalent (in terms of redundancy and error coverage) to the BCH DEC codes described in Section 3.2. That is, as primitive BCH codes offer the minimum possible redundancy, the new codes have been designed to maintain the same redundancy and coverage as the BCH DEC codes mentioned previously, but attempt to reduce the overhead induced by such codes. In Section 3.4, some comments about the redundancy of non-primitive BCH codes, as well as a new code, have been included.

Using the methodology described in Section 3.1, and taking the values $n = 18$ and $k = 8$, we have found an (18, 8) LRRO DEC code. This is its parity check matrix:

$$\mathbf{H}_{\mathbf{LRRO}\ 18,8} = \begin{bmatrix} 100000000011100000 \\ 010000000011010000 \\ 001000000010100000 \\ 000100000010011100 \\ 000010000001101010 \\ 000001000001010110 \\ 000000100000111001 \\ 000000010000000101 \\ 000000001000001011 \\ 000000000100000111 \end{bmatrix}. \tag{9}$$

In the same way, varying the values of n and k, matrices for the new (26, 16), (44, 32), and (78, 64) LRRO DEC codes have been found. They are shown in (10), (11), and (12), respectively:

$$\mathbf{H}_{\mathbf{LRRO}\ 26,16} = \begin{bmatrix} 10000000001110000110010100 \\ 01000000001101100000111000 \\ 00100000001010110001100001 \\ 00010000001001011010001100 \\ 00001000000110101011001000 \\ 00000100000101010101000011 \\ 00000010000011001100100110 \\ 00000001000000111100010001 \\ 00000000100000000011110011 \\ 00000000010000000000001111 \end{bmatrix}, \tag{10}$$

$$\mathbf{H}_{\mathbf{LRRO}\ 44,32} = \begin{bmatrix} 10000000000011100001100101000110101000000010 \\ 01000000000011011000001110100000100010000101 \\ 00100000000010101100011000001000011000000011 \\ 00010000000010010110100011001101000101000000 \\ 00001000000001101010110010001000000100011100 \\ 00000100000001010101010000100100011100101000 \\ 00000010000000110011001001010010000011100001 \\ 00000001000000001111000100010001000010010011 \\ 00000000100000000000111100110011000001001011 \\ 00000000010000000000000011110000110000101111 \\ 00000000001000000000000000001111110000010101 \\ 00000000000100000000000000000000001111111110 \end{bmatrix}, \tag{11}$$

$$\mathbf{H}_{\mathbf{LRRO}\ 78,64} = \begin{bmatrix} 100000000000001110000110010100011010100000000100011000000000000110111000011010 \\ 010000000000001101100000111010000010001000100001011001010001001000100100100100 \\ 001000000000001010110001100000100001100000100000000010010100011000011100111001 \\ 000100000000001001011010001100110100010100001001000110001000000001100001110000 \\ 000010000000000110101011001000100000010001010110010001100000010000000100111010 \\ 000001000000000101010101000010010001110010111000000000000000100010100101000111 \\ 000000100000000011001100100101001000001110010000001000000001001101010001010011 \\ 000000010000000000111100010001000100001001000100000100100000100101101010101101 \\ 000000001000000000000011110011001100000100001100100011000000000000011010001111 \\ 000000000100000000000000001111000011000010000010100000100011100010010111110011 \\ 000000000010000000000000000000111111000001000001100000001101011110000010101111 \\ 000000000001000000000000000000000000111111000000010000011011110000101010011000 \\ 000000000000100000000000000000000000000000111111110000000110110001011001000111 \\ 000000000000010000000000000000000000000000000000001111111110001111000111000100 \end{bmatrix}. \tag{12}$$

It must be considered that these are just some examples of word sizes frequently employed in digital systems. Codes for different sizes (e.g., 128) can also be found using this methodology.

From the parity check matrices, it is easy to obtain the logic equations for the encoder circuits and the syndrome computation in the decoder circuits. An example can be found in the literature [19]. Later, it is necessary to relate each syndrome with the corresponding error vector, implementing a lookup table for each code. In fact, these are the truth tables of a group of logic functions where the inputs are the syndrome bits, and the outputs are the bits of the estimated error vectors.

Each logic function is a sum of minterms representing the syndromes that affect each bit. A good explanation of how to generate the lookup table and the logic equations for the bits of the estimated error vector can be found in the literature [18].

3.4. Redundancy and Non-Primitive BCH Codes

As stated before, although shortened (or non-primitive) BCH codes maintain the error coverage of the primitive codes, they may lose other properties. For example, do they maintain the minimum redundancy property for a given data block size? The answer is no for a general case, although it depends on the number of columns shortened. Using our methodology, we did not find LRRO DEC codes with a lower redundancy than the equivalent BCH DEC codes for 16, 32, and 64 bits. This does not mean that such codes do not exist, as a complete search is unfeasible nowadays; but if they exist, they are not easy to find. Using our methodology, if a code exists, it is frequently found after a short searching time.

However, when the (31, 21) BCH DEC code is shortened to get the (18, 8) code, 13 columns are discarded—more than a half of the data columns. In this case, we found a (16, 8) LRRO DEC code, whose parity check matrix is as follows:

$$\mathbf{H}_{\mathbf{LRRO\ 16,8}} = \begin{bmatrix} 1000000011100001 \\ 0100000011011000 \\ 0010000010101100 \\ 0001000010010110 \\ 0000100001101010 \\ 0000010001010101 \\ 0000001000110011 \\ 0000000100001111 \end{bmatrix}. \tag{13}$$

This code has a better redundancy than the equivalent BCH code. This is important when protecting memories, as the redundant bits are included in all of the words in the whole memory.

In any case, this code is not included in the comparisons performed in the next section. A deeper study of codes with lower redundancy than the equivalent BCH codes, as well as the implementation and synthesis of the code shown in (13), are out of the scope of this paper, and will form part of a future work.

4. Evaluation and Comparison

4.1. Theoretical Study

The real values for the overhead induced by the encoder and decoder circuits of an ECC depend on each implementation: the technology, the synthesis process, etc. From a practical point of view, a logic synthesis is required to obtain these values. However, it is possible to do a more generic, theoretical study. As stated above, the Hamming weight of the heaviest row and the Hamming weight of the whole parity check matrix are important parameters that influence the overhead.

Figure 3 shows the Hamming weights of the heaviest row (i.e., the maximum number of 1 s in a row) of all matrices for the codes considered. As can be observed, all LRRO codes reduce this number with respect to the equivalent BCH codes, and the difference increases as *k* increases.

Figure 4 shows the Hamming weights of the whole matrix (i.e., the total number of 1 s of each parity check matrix) for the codes considered. Again, all LRRO codes reduce this number with respect to the equivalent BCH codes, and the difference increases with *k*.

The results shown here indicate that the proposed LRRO codes improve, in all cases, the attributes of their parity check matrices, when compared with the equivalent BCH codes, maintaining the same redundancy and error coverage. From a theoretical point of view, these improvements should result in encoder and decoder circuits with a lower overhead. The question now is to confirm these good results when the circuits are implemented. This analysis is performed in the following.

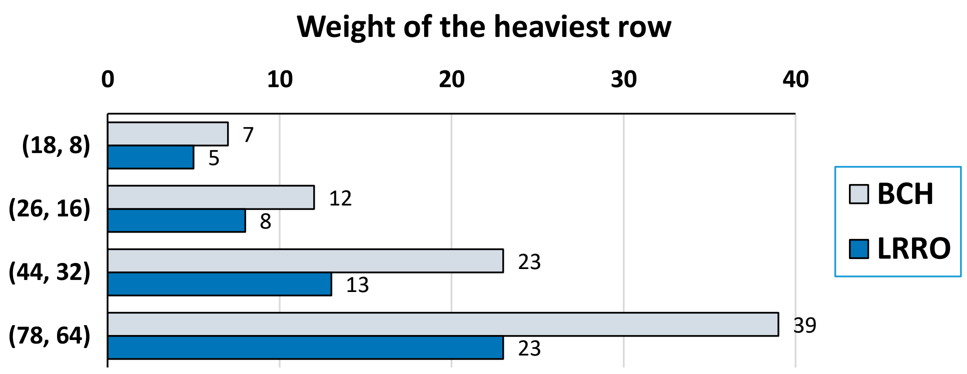


Figure 3. Hamming weights of the heaviest row.

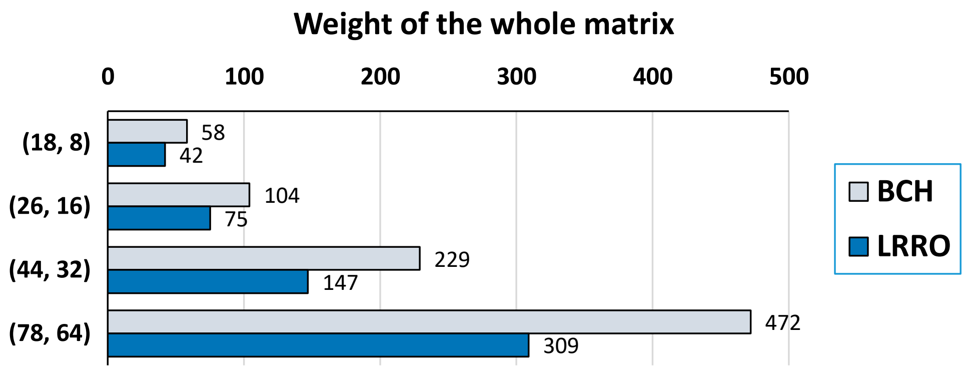


Figure 4. Hamming weights of the whole matrix.

4.2. Implementation and Logic Synthesis

All LRRO DEC codes presented in this paper, as well as the considered BCH DEC codes, have been implemented and simulated to check if they achieve the expected error coverage. We implemented a simulation-based error injection tool [21] with which we could analyze the error coverage of the different ECCs. This tool can inject diverse error models. Particularly, single and multiple random errors were injected. These errors emulate bit-flips in memory cells.

The basic scheme of our error injector is shown in Figure 5. This tool can verify whether the injected error provokes an incorrect or a correct decoding. To do this, the simulation-based error injector tool compares the input and output words.

As expected, all of the studied codes corrected 100% of single and double errors.

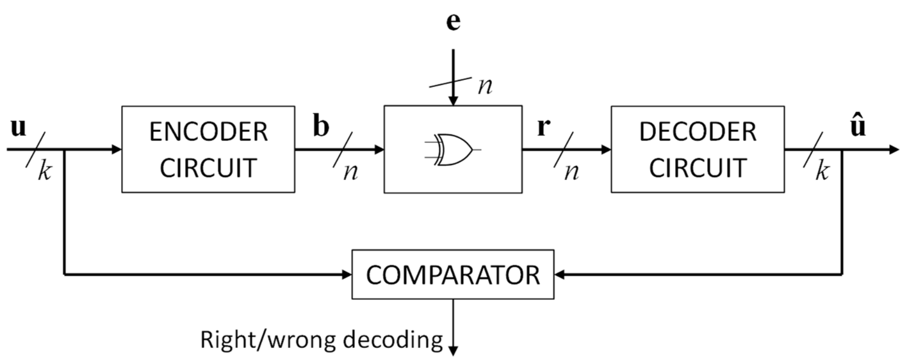


Figure 5. Block diagram of the error injector [21].

On the other hand, the encoder and decoder circuits for all ECCs were implemented in Very High speed integrated circuit Hardware Description Language (VHDL). For example, Figure 6 shows an excerpt of the VHDL implementation of the (26, 16) LRRO DEC code. Then, using CADENCE software [22], we carried out a logic synthesis for 45-nm technology using the NanGate FreePDK45 Open Cell Library [23,24]. Standard cells were based on Scalable Complementary Metal Oxide Semiconductor (SCMOS) design rules. The power voltage and temperature conditions were 1.1 V and 25 °C, respectively.

```
entity Encoder is
      port(u : in  std_logic_vector(0 to 15); -- Original data
           b : out std_logic_vector(0 to 25)  -- Encoded word
        );
end;
architecture behavioral of Encoder is
begin
b(0)  <= u(0) XOR u(1) XOR u(2) XOR u(7) XOR u(8) XOR u(11) XOR u(13);
...         -- Parity bits (according to H)
b(9)  <= u(12) XOR u(13) XOR u(14) XOR u(15);
b(10) <= u(0);
...         -- Data bits
b(25) <= u(15);
end;
```

(**a**)

```
entity Decoder is
      port(r : in  std_logic_vector(0 to 25); -- Received word
           u : out std_logic_vector(0 to 15)  -- Estimated original data
        );
end;
architecture behavioral of Decoder is
  signal e : std_logic_vector(10 to 25); -- Estimated error vector
  signal s : std_logic_vector(0 to 9);   -- Syndrome
begin
s(0) <= r(0) XOR r(10) XOR r(11) XOR r(12) XOR r(17) XOR r(18) XOR r(21) XOR r(23);
...         -- Syndrome bits (according to H)
s(9) ...

e(25) <= (
  NOT s(9) AND NOT s(8) AND s(7) AND s(6) AND NOT s(5) AND NOT s(4) AND NOT s(3) AND s(2) AND NOT s(1) AND NOT s(0)
    ) OR (
  NOT s(9) AND s(8) AND s(7) AND NOT s(6) AND s(5) AND NOT s(4) AND NOT s(3) AND s(2) AND NOT s(1) AND NOT s(0)
    ) OR ...  -- All syndromes affecting bit 25
...         -- Estimated error bits
e(10) ...

u(0) <= r(10) XOR e(10);
...
u(15) <= r(25) XOR e(25);
end;
```

(**b**)

Figure 6. Excerpt of the VHDL implementation for the (26, 16) low redundancy and reduced overhead (LRRO) double error correction (DEC) code: (**a**) encoder and (**b**) decoder.

Although less generic than the theoretical study presented previously, logic synthesis allows for obtaining more realistic information about the overhead induced by different ECCs for a given technology. The values considered for comparison are the propagation delay of the circuits, the silicon area occupied, and the power consumption.

Figure 7 shows the delay induced by the encoder circuits. As can be observed, the proposed LRRO codes greatly reduce this delay for all block sizes.

Figure 8 depicts the propagation delay of the decoder circuits. In this case, the reduction achieved by the LRRO codes is less important than for the encoders. Anyway, all of the LRRO decoders reduce the delay with respect to the equivalent BCH circuits, for all block sizes.

When protecting memories using an ECC, encoding and decoding delays may influence the clock cycle and the working frequency of the processor. Therefore, the reduction achieved by LRRO codes makes them a better option than BCH codes.

The silicon areas occupied by the encoder and decoder circuits (in Figures 9 and 10, respectively) show the same trends observed with the propagation delay. All LRRO circuits use less area than the equivalent BCH circuits, and the reduction is proportionally more important in the encoder circuits. Because of the high integration density of the nanometric manufacturing processes, the silicon area is not a problem nowadays, in most cases. However, LRRO codes are superior to the equivalent BCH codes in this comparison.

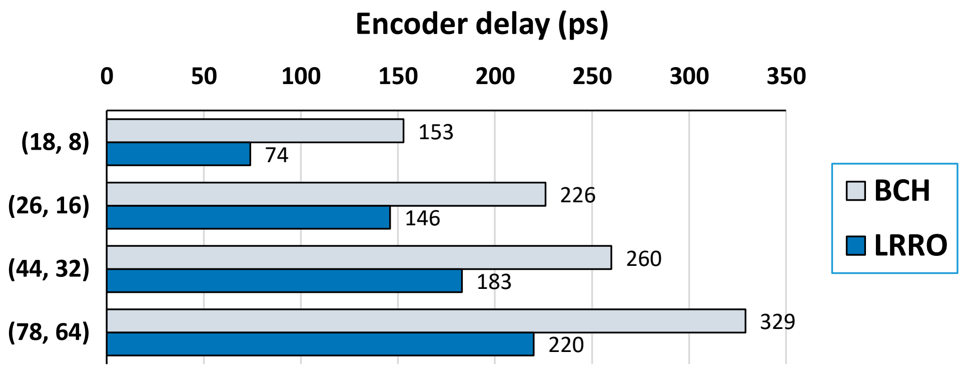


Figure 7. Propagation delay of the encoder circuits (in ps).

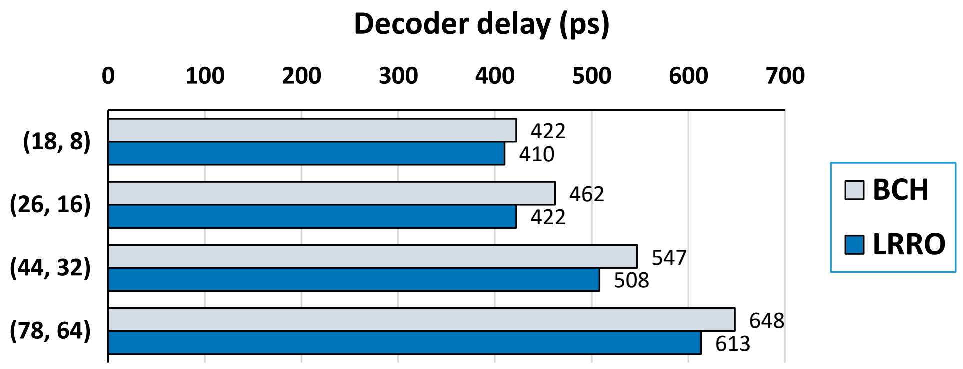


Figure 8. Propagation delay of the decoder circuits (in ps).

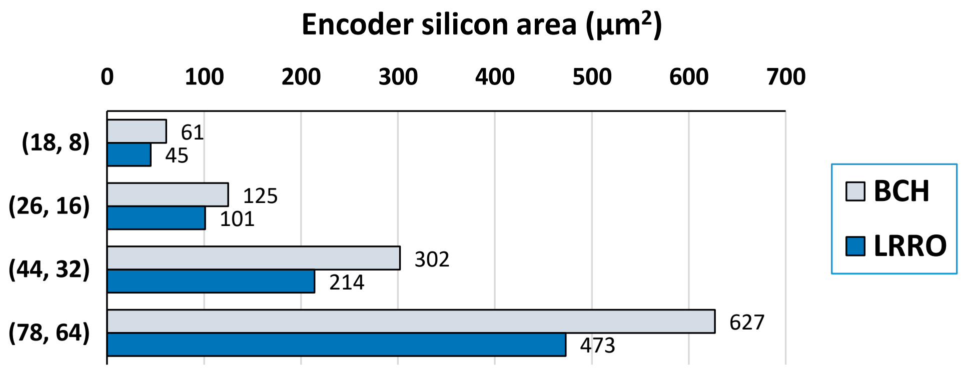


Figure 9. Silicon area occupied by the encoder circuits (in µm^2).

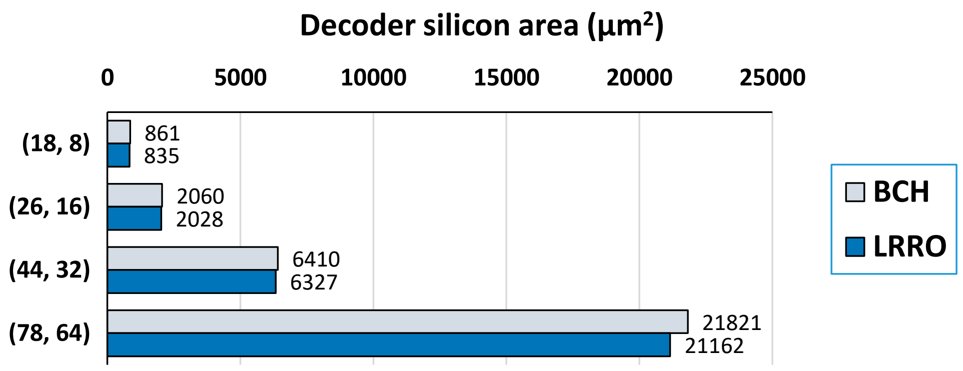


Figure 10. Silicon area occupied by the decoder circuits (in µm^2).

Finally, the power consumed by the encoder circuits is shown in Figure 11, while Figure 12 presents the power consumed by the decoder circuits. Again, the same trends can be observed. All LRRO circuits consume less power than the equivalent BCH circuits. The reduction is proportionally higher in the encoder circuits, but conversely to previous comparisons, the reductions observed in the decoder circuits are also high. The static power, more related to the silicon area, represents a small fraction of the total power consumed. In this case, the dynamic power consumption has been greatly reduced, leading the total power consumed to the results shown.

Nowadays, a low power consumption is a must in most digital systems. Therefore, the reductions achieved by the LRRO codes make them a clearly better option compared with BCH codes.

To sum up, the low redundancy and reduced overhead DEC codes presented in this paper are a better option than the equivalent BCH DEC codes, as they maintain the same redundancy and error coverage, while inducing a reduced overhead. Comparing the codes for the 8-, 16-, 32-, and 64-bit data lengths against BCH, LRRO circuits achieve the following:

- Lower delay: at least 30% reduction in the encoder, up to 8.7% in the decoder circuits.
- Less silicon areas: reductions up to 29% in the encoder and 3.0% in the decoder circuits.
- Lower power consumption: about 40% in the encoder and up to 26.9% in the decoder.

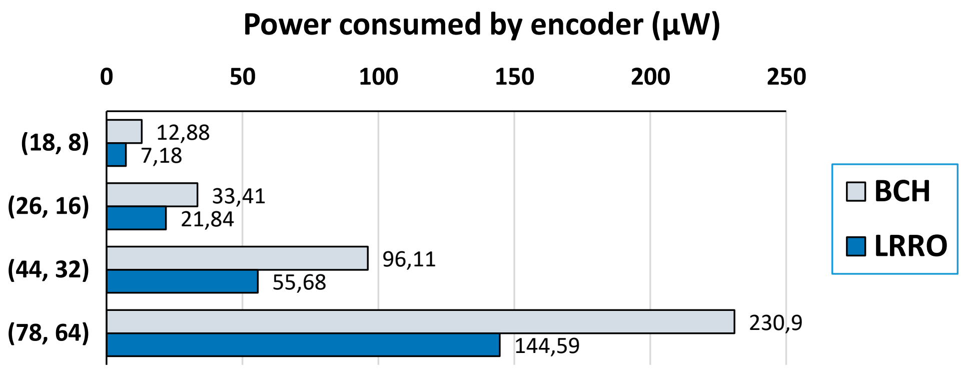


Figure 11. Power consumed by the encoder circuits (in μW).

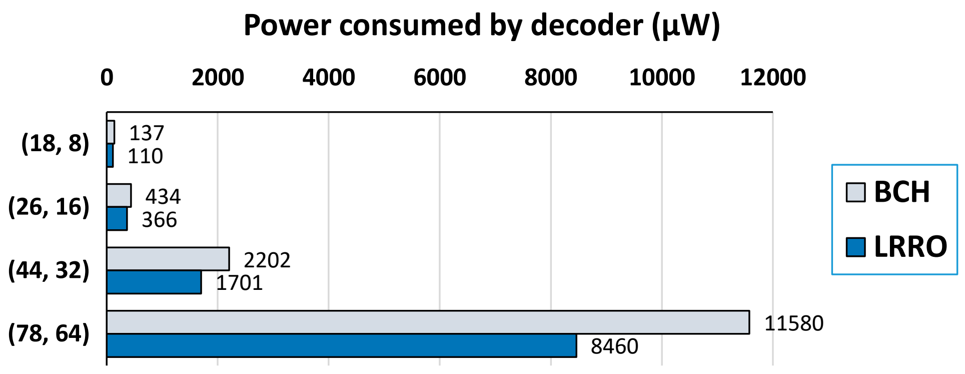


Figure 12. Power consumed by the decoder circuits (in μW).

5. Conclusions

The Bose–Chaudhuri–Hocquenghem (BCH) codes are one of the best-known error control codes. They are extensively employed in several fields, but they have not found favorable application in memories. The non-alignment of their block sizes to typical memory word lengths and the large multi-cycle latency of traditional iterative decoding algorithms are the main reasons for this.

This paper presents low redundancy and reduced overhead (LRRO) double error correction (DEC) codes as an alternative to the BCH DEC codes for protecting registers and memories, i.e., for hardware implementations and block sizes that are in powers of 2 (8, 16, 32, 64, etc.). The objective is to maintain the redundancy and the error coverage of BCH DEC codes, while reducing the overhead induced by the encoder and decoder circuits.

We have demonstrated that all overhead comparisons are favorable to LRRO DEC codes. The theoretical study shows that all LRRO codes have a lower Hamming weight in the heaviest row of their parity check matrices, as well as a lower Hamming weight for the whole matrix, compared with the equivalent BCH DEC codes. Moreover, the logic synthesis illustrates that for all word lengths, the LRRO DEC circuits have a lower propagation delay, employ less silicon areas, and consume less power than the corresponding BCH DEC codes.

Therefore, we propose the LRRO DEC codes as an interesting alternative to BCH DEC codes for protecting registers and memories against single and double errors.

Finding ECCs with a lower redundancy, thereby maintaining affordable overheads, is part of our ongoing and future work.

Author Contributions: conceptualization, L.-J.S.-A. and P.-J.G.-V.; methodology, L.-J.S.-A. and J.G.-M.; software, L.-J.S.-A. and J.G.-M.; validation, L.-J.S.-A., J.G.-M., D.G.-T., and J.-C.B.-C.; formal analysis, L.-J.S.-A., D.G.-T., and P.-J.G.-V.; investigation, L.-J.S.-A. and J.G.-M.; resources, J.-C.B.-C. and P.-J.G.-V.; data curation, L.-J.S.-A. and J.G.-M.; writing—original draft preparation, L.-J.S.-A.; writing—review and editing, J.G.-M., D.G.-T., J.-C.B.-C., and P.-J.G.-V.; visualization, L.-J.S.-A.; supervision, P.-J.G.-V.; project administration, J.G.-M. and P.-J.G.-V.; funding acquisition, J.G.-M. and P.-J.G.-V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the Spanish Government, project TIN2016-81075-R, by Primeros Proyectos de Investigación (PAID-06-18), Vicerrectorado de Investigación, Innovación y Transferencia de la Universitat Politècnica de València (UPV), project 20190032, and by the Institute of Information and Communication Technologies (ITACA).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Fujiwara, E. *Code Design for Dependable Systems*; John Wiley & Sons: Chichester, UK, 2006; doi:10.1002/0471792748.
2. Zhang, X. *VLSI Architectures for Modern Error-Correcting Codes*; CRC Press: Boca Raton, FL, USA, 2016; doi:10.1201/b18673.
3. Hocquenghem, A. Codes correcteurs d'erreurs. *Chiffres* **1959**, *2*, 147–156.
4. Bose, R.C.; Ray-Chaudhuri, D.K. On a Class of Error Correcting Binary Group Codes. *Inf. Control* **1960**, *3*, 68–79, doi:10.1016/S0019-9958(60)90287-4.
5. Chen, P.; Zhang, C.; Jiang, H.; Wang, Z.; Yue, S. High performance low complexity BCH error correction circuit for SSD controllers. In Proceedings of the IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC), Singapore, 1–4 June 2015; pp. 217–220, doi:10.1109/EDSSC.2015.7285089.
6. Nguyen, J.P. Applications of Reed Solomon Codes on Optical Media Storage. Master's Thesis, San Diego State University, San Diego, CA, USA, October 2011.
7. IEEE 802.3-2018 - IEEE Standard for Ethernet. Available online: https://standards.ieee.org/standard/802_3-2018.html (accessed on 14 October 2020).
8. H.263: Video Coding for Low Bit Rate Communication. Available online: https://www.itu.int/rec/T-REC-H.263/en (accessed on 14 October 2020).
9. Vangelista, L.; Benvenuto, N.; Tomasin, S.; Nokes, C.; Stott, J.; Filippi, A.; Vlot, M.; Mignone, V.; Morello, A. Key technologies for next-generation terrestrial digital television standard DVB-T2. *IEEE Commun. Mag.* 2009, *47*, 146–153, doi:10.1109/MCOM.2009.5273822.
10. McEliece, R.J.; Swanson, L. Reed–Solomon codes and the exploration of the solar system. In *Reed–Solomon Codes and Their Applications*; Wicker, S.B., Bhargava, V.K., Eds.; IEEE Press: Piscataway, NJ, USA, 1994; pp. 25–40.
11. Lin, S.; Costello, D.J. *Error Control Coding*, 2nd ed.; Pearson Prentice-Hall: Upper Saddle River, NJ, USA, 2004.

12. 2013 ITRS—International Technology Roadmap for Semiconductors. Available online: http://www.itrs2.net/2013-itrs.html (accessed on 14 October 2020).
13. Ibe, E.; Taniguchi, H.; Yahagi, Y.; Shimbo, K.; Toba, T. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Trans. Electron Devices* 2010, *57*, 1527–1538, doi:10.1109/TED.2010.2047907.
14. Gil-Tomás, D.; Gracia-Morán, J.; Baraza-Calvo, J.C.; Saiz-Adalid, L.J.; Gil-Vicente, P.J. Studying the effects of intermittent faults on a microcontroller. *Microelectron. Reliab.* **2012**, *52*, 2837–2846, doi:10.1016/j.microrel.2012.06.004.
15. Neubauer, A.; Freudenberger, J.; Kühn, V. *Coding Theory: Algorithms, Architectures and Applications*; John Wiley & Sons: Chichester, UK, 2007; doi:10.1002/9780470519837.
16. Morelos-Zaragoza, R.H. *The Art of Error Correcting Coding*, 2nd ed.; John Wiley & Sons: Chichester, UK, 2006; doi:10.1002/0470035706.
17. Naseer, R.; Draper, J. DEC ECC design to improve memory reliability in sub-100nm technologies. In Proceedings of the IEEE International Conference on Electronics, Circuits and Systems, St. Julien's, Malta, 31 August–3 September 2008; pp. 586–589, doi:10.1109/ICECS.2008.4674921.
18. Saiz-Adalid, L.J.; Gracia-Morán, J.; Gil-Tomás, D.; Baraza-Calvo, J.C.; Gil-Vicente, P.J. Ultrafast Codes for Multiple Adjacent Error Correction and Double Error Detection. *IEEE Access* **2019**, *7*, 151131–151143, doi:10.1109/ACCESS.2019.2947315.
19. Saiz-Adalid, L.J.; Gil-Vicente, P.J.; Ruiz, J.C.; Gil-Tomás, D.; Baraza, J.C.; Gracia-Morán, J. Flexible unequal error control codes with selectable error detection and correction levels. In *Lecture Notes in Computer Science, vol 8153, Proceedings of the 2013 Computer Safety, Reliability, and Security Conference (SAFECOMP 2013), Toulouse, France, 24–27 September 2013*; Bitsch, F., Guiochet, J., Kaâniche, M. Eds.; Springer: Berlin, Germany, 2013; pp. 178–189, doi:10.1007/978-3-642-40793-2_17.
20. Saiz-Adalid, L.J.; Reviriego, P.; Gil, P.; Pontarelli, S.; Maestro, J.A. MCU Tolerance in SRAMs Through Low-Redundancy Triple Adjacent Error Correction. *IEEE Trans. VLSI Syst.* **2015**, *23*, 2332–2336, doi:10.1109/TVLSI.2014.2357476.
21. Gracia-Moran, J.; Saiz-Adalid, L.J.; Gil-Tomás, D.; Gil-Vicente, P.J. Improving Error Correction Codes for Multiple Cell Upsets in Space Applications. *IEEE Trans. VLSI Syst.* **2018**, *26*, 2132–2142, doi:10.1109/TVLSI.2018.2837220.
22. Cadence: Computational Software for Intelligent System Design. Available online: https://www.cadence.com (accessed on 14 October 2020).
23. Stine, J.E.; Castellanos, I.; Wood, M.; Henson, J.; Love, F.; Davis, W.R.; Franzon, P.D.; Bucher, M.; Basavarajaiah, S.; Oh, J.; et al. FreePDK: An open-source variation-aware design kit. In Proceedings of the IEEE International Conference on Microelectronic Systems Education (MSE), San Diego, CA, USA, 3–4 June 2007; pp. 173–174, doi:10.1109/MSE.2007.44.
24. NanGate FreePDK45 Open Cell Library. Available online: http://www.nangate.com/?page_id=2325 (accessed on 28 May 2018).