Flexible Unequal Error Control Codes with Selectable Error Detection and Correction Levels

Luis-J. Saiz-Adalid, Pedro-J. Gil-Vicente, Juan-Carlos Ruiz-García, Daniel Gil-Tomás, J.-Carlos Baraza, and Joaquín Gracia-Morán

Instituto ITACA – Universitat Politècnica de València
Camino de Vera, s/n, 46022, Valencia, Spain
{ljsaiz,pgil,jcruizg,dgil,jcbaraza,jgracia}@itaca.upv.es

Abstract. Unequal Error Control (UEC) codes provide means for handling errors where the codeword digits may be exposed to different error rates, like in two-dimensional optical storage media, or VLSI circuits affected by intermittent faults or different noise sources. However, existing UEC codes are quite rigid in their definition. They split codewords in only two areas, applying different (but limited) error correction functions in each area. This paper introduces Flexible UEC (FUEC) codes, which can divide codewords into any required number of areas, establishing for each one the adequate error detection and/or correction levels. At design time, an algorithm automates the code generation process. Among all the codes meeting the requirements, different selection criteria can be applied. The code generated is implemented using simple logic operations, allowing fast encoding and decoding. Reported examples show their feasibility and potentials.

Keywords: error detection and correction codes, information redundancy, unequal error control codes.

1 Introduction

Error correction codes (ECCs) are widely used in today's computer systems to provide reliable delivery and storage of digital data over unreliable communication channels and memories. Most ECCs are based on the premise that all bits in a codeword require the same error control level. However, this vision of the problem is not valid for application domains where the bit error rate (BER) does not homogeneously affect to all codeword bits [1]. Far from being marginal, the problem of variable BER (vBER) is getting more and more important. For instance, intermittent faults in VLSI circuits appear repeatedly and non-deterministically in the same place. They only increase the BER of the affected locations, but not the BER of the rest of the bits in the word. Although the specific causes leading to this trend are out the scope of this paper, it is worth noting that, nowadays, 6.2% of errors in memory subsystems [2], and 39% of hardware errors in microprocessors reported to operating systems have an intermittent nature [3]. vBER problem also applies to Volume Holographic Memories (VHM) and other two dimensional optical storage, where the BER of readout data from the edge is much higher than that from the center of the media [4]. This threat must be taken into account as this technology for data-storage hierarchy presents short access time, high aggregate data-transfer rate, and large storage capacity.

The aforesaid examples motivate the increasing interest that asymmetric error control has in current, and will have in future, computer-based systems. Unequal error control (UEC) codes [1] establish different error control levels in diverse codeword areas. In practice, existing UEC codes simply split codewords in two parts. They apply either full error correction (i.e. fixing all the possible errors affecting that word area) [1] or burst error correction (i.e. fixing burst errors of a given length affecting that area) [5] in the strongly controlled area. In contrast, only single error correction (and sometimes double error detection) is applied in the weakly controlled area.

The limitation of using full error correction in the strongly controlled area is the high level of redundancy required. In contrast, burst error correction, although less redundancy demanding, does not cover some classical error patterns, such as double random errors (two errors separated beyond the controlled burst length). On the other hand, single error correction may be insufficient even for a weakly controlled area, as multiple errors are becoming more and more frequent in today's systems [6]. Thus, the main drawback of existing UEC codes is their lack of flexibility, as designers cannot use different error control functions on each area, according to the specifications of each system, application or context of use.

This paper proposes Flexible UEC (FUEC) codes, a new type of UEC codes enhanced for flexibility. They allow to establish any desired number of control areas in a codeword, in relation to the needs, and to deploy the adequate error control strategy in each part. The focus will be placed on how to combine single, multiple and burst error correction and/or detection capabilities, and selectively apply them in each identified codeword area. In case of multiple solutions, different selection criteria can be used.

The rest of this paper is organized as follows. Section 2 introduces the very basic notions related to ECCs and UEC codes, while Section 3 details the methodology to generate FUEC codes. In Section 4, the feasibility and potentials of FUEC codes are discussed. Finally, Section 5 provides some conclusions and ideas for future work.

2 UEC Codes: Background

UEC codes are a type of linear block codes, a kind of ECCs. For a better understanding of their potentials, this section introduces some important notions. Basics will be first applied to ECCs and then extended to UEC codes.

2.1 Basics on Encoding and Decoding

An (n, k) binary ECC encodes a k-bit input word in an n-bit output word [7]. Fig. 1 synthesizes the encoding and decoding processes. The input word $\mathbf{u}=(u_0, u_1, \ldots, u_{k-1})$ is a k-bit vector which represents the original data. The codeword $\mathbf{b}=(b_0, b_1, \ldots, b_{n-1})$ is a vector of n bits, where the code adds the required redundancy. It is transmitted across the channel which 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, $e_i=0$; otherwise, $e_i=1$. In this way, **r** can be interpreted as $\mathbf{r}=\mathbf{b}\oplus\mathbf{e}$.

The parity-check matrix **H** of a linear block code defines the code. For the encoding process, **b** must accomplish the requirement $\mathbf{H}\cdot\mathbf{b}^T=\mathbf{0}$. For syndrome decoding, the syndrome is defined as $\mathbf{s}^T=\mathbf{H}\cdot\mathbf{r}^T$, and it exclusively depends on **e**:

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

There must be a different **s** for each correctable **e**. If **s**=**0**, then **e**=**0**. So, **r** is correct. Otherwise, the syndrome decoding is performed by addressing a lookup table relating each **s** with the decoded error vector $\hat{\mathbf{e}}$. By XORing $\hat{\mathbf{e}}$ from **r**, the decoded codeword $\hat{\mathbf{b}}$ is obtained: $\hat{\mathbf{b}} = \mathbf{r} \oplus \hat{\mathbf{e}}$. If **s** is non-zero, but the lookup table has no entry for that syndrome, the error is detected, but cannot be corrected.

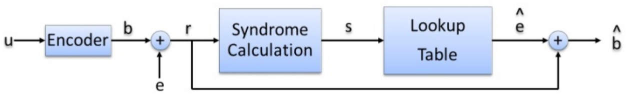

Fig. 1. Encoding, channel crossing and decoding process

Encoding and decoding functions are similar for UEC codes (although **b** is divided in two areas and the *n–k* redundant bits added to **b** are differently located). Such functions and their implementation can be deduced from **H**, as exemplified in Section 3.2. The encoding and decoding circuits can be implemented using XOR trees.

Let us focus now on the various types of errors typically addressed using ECCs and UEC codes. It must be noted that an accurate modeling of such errors is essential to reduce the level of redundancy induced in the resulting codeword, and thus the number of bits required in such a word to limit its length.

2.2 Error Models

As previously stated, the difference between **b** and **r** is induced by an unreliable channel that introduces **e**. This section describes the error models used in this work.

The term **random error** refers to one or more bits in error, distributed randomly in **b**. Random errors can be single (only one bit is affected) or multiple. Single errors are commonly produced by single event effects (SEEs) [8]: single event upsets (SEUs, in random access memories), single event transients (SETs, in combinational logic), etc.

Multiple errors are becoming more frequent as integration scales increase [6][9], although they usually manifest as burst errors, rather than randomly [10]. A **burst error** is a multiple error that spans *l* (named burst length) bits in a word [1], i.e. a group of adjacent bits where, at least, the first and the last bits are in error. The physical causes of a burst error are diverse [9][10]: crosstalk effects induced between neighbor wires in parallel buses, noise affecting several bits in a serial transmission, high energy cosmic ray that hits some neighbor positions in storage elements, etc.

Let us introduce now some notation about error vectors used from now on.

Let $\boldsymbol{E}_*$ be the set of all possible error vectors. Its cardinality is $|\boldsymbol{E}_*| = 2^n$. The elements in this set can be grouped in different subsets. For example, let $\boldsymbol{E}_w$ be the set of error vectors with Hamming weight w (the Hamming weight is the number of 1s in a binary word). For example, if n=3 and w=1, $\boldsymbol{E}_1$ = {(100), (010), (001)}.

Regarding burst errors, let $\boldsymbol{E}_{Bl}$ be the set of burst error vectors, where l is the burst length. For example, if n=6 and l=4, the $\boldsymbol{E}_{B4}$ set is composed by these error vectors:

$\boldsymbol{E}_{B4}$ ={ (111100), (011110), (001111), (101100), (010110), (001011),
(110100), (011010), (001101), (100100), (010010), (001001) }

Let us now consider $\boldsymbol{E}_+$ as the set of all error vectors that determine which errors can be corrected by a given code, including the no-error subset ($\boldsymbol{E}_0$). When using syndrome decoding, $|\boldsymbol{E}_+|$ determines the minimum number of syndromes required to correct the selected errors, i.e. the condition $|\boldsymbol{E}_+| \leq 2^{n-k}$ must be satisfied.

The minimum set of error vectors to be detected (excluding those in $\boldsymbol{E}_+$) is represented by $\boldsymbol{E}_\Delta$. Nevertheless, other error vectors, not included in $\boldsymbol{E}_\Delta$, may be detected. This set is used to indicate the minimum detection requirements of a code.

When related to UEC codes, the previous definitions do not apply to the whole word, but to a subset of bits, defining a *control area*. Superscript numbers are then used to distinguish areas ($\boldsymbol{E}_*^{0..7}$, for example). These error subsets will be useful to define different error control functions in distinct areas of a codeword.

2.3 Unequal Error Control vs. Unequal Error Protection Codes

UEC codes are developed considering that some bits require higher error control than others. Although they share some common features with Unequal Error Protection (UEP) codes, they must not be confused, since their definitions, objectives and applications are quite different.

UEP codes [11], typically used in multimedia and control communications, protect some digits in a codeword against a higher number of errors than others. UEP codes ensure the correct decoding of the strongly protected part, and accept the eventual incorrectly decoding of the weakly protected part, i.e. the integrity of the whole codeword is desired but not required.

Conversely, UEC codes consider distinct error control levels in a codeword, in such a way that a part of the word is more strongly controlled against errors than the rest. UEC codes enable different error control functions to be applied to distinct parts of a codeword, but error protection is applied to the whole word, thus making unacceptable the incorrect decoding of any part of such word.

Thus, from a protection level viewpoint, UEP codes are different from UEC codes. UEC codes decode correctly the word if the error requirements are met; otherwise, the whole word can be corrupted. UEP codes tolerate a wrong decoding if it is "good enough", i.e. errors in the weakly protected bits are acceptable. So, UEP codes have different protection levels, while UEC codes protect all codeword bits equally.

From a control level standpoint, UEP codes usually consider the codeword as a whole, all bits having the same error rate (under a given number of errors, some bits are guaranteed to be correctly decoded, while others could be in error or miscorrected). UEC codes commonly consider multiple areas, and different error control levels are applied to each area (while fault hypothesis are met, errors are covered and the whole word is correctly decoded; otherwise, the integrity of the codeword cannot be guaranteed). From this perspective, UEC codes have various control levels, while UEP codes can only have one.

The aforementioned differences obviously establish different contexts of use for UEP and UEC codes. On one hand, UEP codes are used when some information in part of the word is more important than in other parts (e.g. control information in communication messages or headers in multimedia transmission). On the other hand, UEC codes are appropriate in scenarios where constant bit error rates cannot be assumed (e.g. VHM or VLSI circuits affected by intermittent faults).

As motivated in Section 1, the increasing importance of vBER problem is making UEC codes a more and more interesting solution. Nevertheless, existing UEC codes have some limitations, mainly related to both the limited number of parts and the options for the error control functions applied on each part. This work proposes FUEC codes to solve them.

3 Flexible Unequal Error Control Codes

Existing UEC codes split codewords only in two parts, and they offer limited error control functions. Full error correction in the strongly controlled area requires a high level of redundancy, whereas burst error correction does not cover some classical error patterns. Single error correction may be insufficient even for a weakly controlled area. So, the main drawback of existing UEC codes is their lack of flexibility.

For the sake of understanding, a simple design example will be introduced to provide an overview of FUEC codes, and present their features and potentials.

3.1 Overview

Let us consider a two-dimensional optical storage with vBER [1]. In VHM, for example, the BER of readout data from the edge of the media is much higher than that from the center [4]. Asymmetric error control would reduce the redundancy-coverage ratio. Two control levels are applied in [1]; nevertheless, the BER has not only two levels in VHM actually. In fact, the BER is proportional to the distance from the center of the media [4]. Hence, more than two areas should be considered.

Let us study a simple example. For the sake of simplicity, words are 12-bit long, divided in three 4-bit areas, as presented in Fig. 2(a). This methodology can be applied to a higher number of bits, as it will be shown in Section 4.

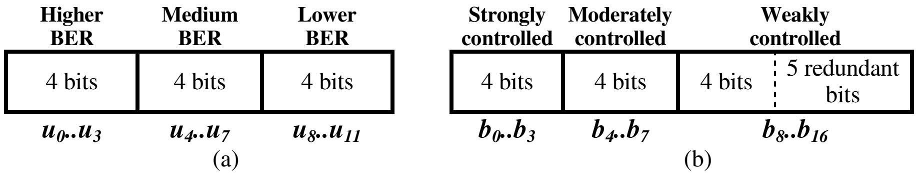

Fig. 2. Layout of the input word (a) and the encoded word (b)

The following fault hypothesis are considered: i) all 1-bit errors and all 2-bit burst errors should be corrected, and all 2-bit errors and 3-bit burst errors should be detected in the strongly controlled area (the one with higher BER); ii) all 1-bit errors and all 2-bit burst errors must be corrected in the moderately controlled area, with no additional detection; iii) 1-bit error correction and 2-bit burst error detection will be implemented in the weakly controlled area (the one with lower BER).

The value of *k* for the code to be designed is 12. The selection of the number of redundant bits (which finally determines the value of *n*) must allow finding an **H** matrix which solves the error control requirements. It is possible to estimate the number of redundant bits depending on these requirements, as explained later. Next, the methodology used to generate FUEC codes is applied to this example.

3.2 Methodology Description

Several parameters must be set to design a code: the data length (*k*), the encoded word length (*n*), the set of error vectors to be corrected $(\boldsymbol{E}_+)$ and detected $(\boldsymbol{E}_\Delta)$. For FUEC codes, other parameters have to be considered to define $\boldsymbol{E}_+$ and $\boldsymbol{E}_\Delta$: the number of control areas of the codeword, the boundaries (that is, the first and last bits) of each area, and the error control level to apply to each area.

With these parameters, it is possible to obtain an *((n–k)×n)* **H** matrix able to correct and detect the selected errors, if it exists. The methodology proposed consists on considering all possible matrices. As stated above, **s** exclusively depends on **e** (see (1)). **H** must satisfy condition (2), as there must be a different syndrome for each correctable error. The condition for additional error detection is (3).

$$\mathbf{H}\cdot\mathbf{e}_i^T \neq \mathbf{H}\cdot\mathbf{e}_j^T;\forall\mathbf{e}_i,\mathbf{e}_j \in \boldsymbol{E}_+ \mid \mathbf{e}_i \neq \mathbf{e}_j \tag{2}$$

$$\mathbf{H}\cdot\mathbf{e}_i^T \neq \mathbf{H}\cdot\mathbf{e}_j^T;\forall\mathbf{e}_i \in \boldsymbol{E}_\Delta,\mathbf{e}_j \in \boldsymbol{E}_+ \tag{3}$$

That is, each detectable error must generate a syndrome which is different to all the syndromes generated by the correctable errors. However, several detectable errors may have the same syndrome.

Searching **H** can be considered a Boolean satisfiability (SAT) problem. Previous proposals to solve this problem [10][12] are focused on specific applications. Our proposal is more general: in three successive steps, our algorithm is able to find any binary linear block code, if it exists, just selecting the set of error vectors to be corrected. So, the first step is to determine $\boldsymbol{E}_+$ and $\boldsymbol{E}_\Delta$. Then, the algorithm tries to find an **H** matrix able to solve conditions (2) and (3). Finally, as several solutions can be found, one of them can be selected using different criteria.

Determining $\boldsymbol{E}_+$ and $\boldsymbol{E}_\Delta$ Error Vector Sets. Taking the example presented in Section 3.1, the set of correctable errors is $\boldsymbol{E}_+ = \boldsymbol{E}_0 \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_{B2}^{0..7}$. It includes the no-error vector $(\boldsymbol{E}_0)$, all single bit errors $(\boldsymbol{E}_1)$ and all 2-bit burst errors in the areas with higher and medium BER $(\boldsymbol{E}_{B2}^{0..7})$. As stated above, the condition $|\boldsymbol{E}_+| \leq 2^{n-k}$ must be satisfied. In this case, $|\boldsymbol{E}_+| = |\boldsymbol{E}_0| + |\boldsymbol{E}_1| + |\boldsymbol{E}_{B2}^{0..7}| = 1 + n + 7 \leq 2^{n-12}$. From this expression, $n \geq 17$, that is, at least 5 redundant bits are required. If n = 17, $|\boldsymbol{E}_{+}| = 25$ and $2^{n-k} = 32$. The layout of the encoded word is shown in Fig. 2(b), and the vectors representing the errors to be corrected are included in Table 1.

Positioning the redundant bits in the weakly controlled area is not a requirement. The methodology allows positioning them in any area, considering that the number of error vectors to be included may increase, and probably the number of redundant bits *(n–k)* too. In fact, the position of redundant bits depends on the design specifications.

Let us define now $\boldsymbol{E}_{\Delta}$. As decided in the requirements of this example, $\boldsymbol{E}_{\Delta} = (\boldsymbol{E}_{2}^{\boldsymbol{0..3}} - \boldsymbol{E}_{B2}^{\boldsymbol{0..3}}) \cup \boldsymbol{E}_{B3}^{\boldsymbol{0..3}} \cup \boldsymbol{E}_{B2}^{\boldsymbol{7..16}}$. The seven (32–25) syndromes not used for correction are employed for the detection of these error vectors, grouped in Table 2.

This is just an example. In the same way, other sets of error vectors can be generated, depending on the requirements of the code to be designed.

Table 1. Vectors representing the errors to be corrected in the considered example

Correctable errors ($\boldsymbol{E}_{+}$)	**Error subset**
(00000000000000000)	No error ($\boldsymbol{E}_0$)
(10000000000000000) (01000000000000000) (00100000000000000) (00010000000000000) (00001000000000000) (00000100000000000) (00000010000000000) (00000001000000000) (00000000100000000) (00000000010000000) (00000000001000000) (00000000000100000) (00000000000010000) (00000000000001000) (00000000000000100) (00000000000000010) (00000000000000001)	Single bit errors ($\boldsymbol{E}_1$)
(11000000000000000) (01100000000000000) (00110000000000000) (00011000000000000) (00001100000000000) (00000110000000000) (00000011000000000)	2-bit burst errors in the areas with higher and medium BER ($\boldsymbol{E}_{B2}^{\boldsymbol{0..7}}$)

Table 2. Vectors representing the errors to be detected in the considered example

Detectable errors ($\boldsymbol{E}_{\Delta}$)	**Error subset**
(10100000000000000) (10010000000000000) (01010000000000000)	2-bit random errors in the area with higher BER, excluding correctable errors ($\boldsymbol{E}_{2}^{\boldsymbol{0..3}} - \boldsymbol{E}_{B2}^{\boldsymbol{0..3}}$)
(1x100000000000000) (01x10000000000000)	3-bit burst errors in the area with higher BER ($\boldsymbol{E}_{B3}^{\boldsymbol{0..3}}$)
(00000001100000000) (00000000110000000) (00000000011000000) (00000000001100000) (00000000000110000) (00000000000011000) (00000000000001100) (00000000000000110) (00000000000000011)	2-bit burst errors in the areas with lower BER ($\boldsymbol{E}_{B2}^{\boldsymbol{7..16}}$)

Computing the Parity-check Matrix (H). After determining $\boldsymbol{E}_{+}$ and $\boldsymbol{E}_{\Delta}$, the algorithm has to find an **H** matrix that satisfies the conditions (2) and (3) with the previously selected set of error vectors. As all the matrices have to be considered, it may require a huge computational effort. A recursive backtracking algorithm has been developed to lighten the process. It checks partial matrices and adds a new column only if the previous matrix satisfies the requirements.

In this way, the algorithm starts with a **partial_H** matrix, with *n–k* rows and only one column. Then, it is checked that this matrix accomplishes the requirements. If it does, new columns are added recursively. Both the initial and the added columns must be non-zero, so there are $2^{n-k} - 1$ combinations for each column. The pseudo code of the partial matrix checker procedure is presented in Fig. 3.

```
Procedure CheckPartialMatrix partial_H $(n-k) \times ncols$      /* $ncols \in [1..n]$ */
    SyndromeSet = { }
    For each error vector e in $E_+$
        partial_e = $(e_1, e_2, ..., e_{ncols})$
        If HammingWeight(e) = HammingWeight(partial_e)
            newSyndrome = CalculateSyndrome(partial_H × Transpose(partial_e))
            If newSyndrome in SyndromeSet then Return       /* Not valid partial matrix */
            Else Add newSyndrome to SyndromeSet
        End if
    End for
    For each error vector e in $E_\Delta$
        partial_e = $(e_1, e_2, ..., e_{ncols})$
        If HammingWeight(e) = HammingWeight(partial_e)
            newSyndrome = CalculateSyndrome(partial_H × Transpose(partial_e))
            If newSyndrome in SyndromeSet then Return       /* Not valid partial matrix */
            Else Do Nothing       /* newSyndrome not stored in this case */
        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: $2^{n-k}-1$ possible values */
            CheckPartialMatrix [partial_H | new_column] $(n-k) \times (ncols+1)$
        End for
    End if
End procedure
```

Fig. 3. Partial matrix checker procedure

The Hamming weight indicates the number of 1s in a vector. It is used to check if the error vectors have all their 1s in the first *ncols* bits, because it is impossible to calculate the syndrome for an error vector with 1s in columns not included in the partial matrix. As new columns are added, more error vectors can be processed.

The first loop generates syndromes for all selected error vectors, and they are added to ***SyndromeSet***. When a new syndrome is generated, it is verified if it has been previously added. In this case, two different error vectors generate the same syndrome, so **partial_H** cannot be part of a valid solution. If the algorithm arrives at the end of the first loop, all selected error vectors have different syndromes.

Then, a second loop calculates syndromes for the detectable errors. Now, it is tested whether these syndromes have been previously added to ***SyndromeSet*** (i.e. the syndrome is associated to a correctable error). In this case, **partial_H** cannot be part of a valid solution. Unlike the first loop, the calculated syndromes are not added to ***SyndromeSet***, as these syndromes are not associated only to one detectable error. If the algorithm arrives at the end of the second loop, all selected correctable errors have non-equal syndromes, and the detectable errors have distinct syndromes from those used for correction.

Now, if **partial_H** has *n* columns, then a solution has been found, and it is added to ***SolutionsSet***. If it has fewer columns, a new column has to be added. A third loop for all possible combinations calls recursively the checker procedure.

Although the condition $|\boldsymbol{E}_+| \leq 2^{n-k}$ was satisfied, it does not guarantee the existence of a code with the selected requirements. If no solution is found, ***SolutionsSet*** is empty, and hence the searched code does not exist. This only can be solved in two ways: increasing the redundancy (i.e. a bigger value of *n*, maintaining *k*), or reducing the error control (that is, decreasing the number of error vectors to be corrected or detected). Once ***SolutionsSet*** is obtained, a solution can be chosen according to different criteria. Let us introduce here some of them:

- First found: this option allows reducing the **H** generation time.
- Smallest Hamming weight of **H**: this solution commonly offers circuits with the lowest number of logic gates in a hardware implementation, for example.
- Smallest Hamming weight of the heaviest row of **H**: the logic depth of each parity or syndrome bit generator circuit usually depends on the Hamming weight of the associated row. The heaviest row determines the speed of the encoder and the decoder circuits.

It is worth noting that other criteria can be applied, depending for example on the technology and requirements of the implementation of encoding and decoding.

Next paragraph presents the code obtained for our case study, applying the smallest Hamming weight of **H** criterion.

Code Implementation. Attending to the error vectors in Table 1 and Table 2, and selecting a solution with smallest Hamming weight in **H**, one possible solution is:

b/r_0	b/r_1	b/r_2	b/r_3	b/r_4	b/r_5	b/r_6	b/r_7	b/r_8	b/r_9	b/r_{10}	b/r_{11}	b/r_{12}	b/r_{13}	b/r_{14}	b/r_{15}	b/r_{16}
u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}	u_{11}	See (4)				
1	0	1	0	0	1	0	0	1	1	1	1	1	0	0	0	0
0	1	0	1	0	0	1	0	1	1	0	0	0	1	0	0	0
0	0	1	0	1	0	1	1	1	0	1	0	0	0	1	0	0
1	0	0	1	1	0	1	0	0	1	0	1	0	0	0	1	0
0	1	0	0	1	1	0	1	0	0	1	1	0	0	0	0	1

As stated in Section 2.1, once calculated **H**, encoding and decoding formulas can be obtained easily from it. In this case, **u** is part of **b**, and the parity bits are located in the columns with only one 1. Each parity bit is calculated by XORing the bits with a 1 in its row. For example, bit b_{13} of the codeword is a parity bit, because the corresponding column in **H** has only one 1, in the second row. Searching the other 1s in the same row, they are in the columns corresponding to u_1, u_3, u_6, u_8 and u_9.

Similarly, **s** is calculated using **r**. Each row generates a syndrome bit by XORing the bits in positions with a 1 in each row. The formulas for the proposed code are:

$$\begin{array}{ll} b_i = u_i, \forall i \in \mathrm{N} : 0 \le i \le 11 & \\ b_{12} = u_0 \oplus u_2 \oplus u_5 \oplus u_8 \oplus u_9 \oplus u_{10} \oplus u_{11} & s_0 = r_0 \oplus r_2 \oplus r_5 \oplus r_8 \oplus r_9 \oplus r_{10} \oplus r_{11} \oplus r_{12} \\ b_{13} = u_1 \oplus u_3 \oplus u_6 \oplus u_8 \oplus u_9 & s_1 = r_1 \oplus r_3 \oplus r_6 \oplus r_8 \oplus r_9 \oplus r_{13} \\ b_{14} = u_2 \oplus u_4 \oplus u_6 \oplus u_7 \oplus u_8 \oplus u_{10} & s_2 = r_2 \oplus r_4 \oplus r_6 \oplus r_7 \oplus r_8 \oplus r_{10} \oplus r_{14} \\ b_{15} = u_0 \oplus u_3 \oplus u_4 \oplus u_6 \oplus u_9 \oplus u_{11} & s_3 = r_0 \oplus r_3 \oplus r_4 \oplus r_6 \oplus r_9 \oplus r_{11} \oplus r_{15} \\ b_{16} = u_1 \oplus u_4 \oplus u_5 \oplus u_7 \oplus u_{10} \oplus u_{11} & s_4 = r_1 \oplus r_4 \oplus r_5 \oplus r_7 \oplus r_{10} \oplus r_{11} \oplus r_{16} \end{array} \tag{4}$$

The syndrome bits determine the next step. If they are all 0, **r** is assumed to be correct. If not, **s** can be associated to a correctable error. In this case, the bit(s) affected are corrected; otherwise, the "error detected" condition is achieved. Table 3 shows the association syndrome-correction/detection for the proposed example.

Table 3. Syndrome lookup table (estimated errors) for the proposed code

$s_4\,s_3\,s_2\,s_1\,s_0$	Error in…	$s_4\,s_3\,s_2\,s_1\,s_0$	Error in…	$s_4\,s_3\,s_2\,s_1\,s_0$	Error in…	$s_4\,s_3\,s_2\,s_1\,s_0$	Error in…
0 0 0 0 0	No error	0 1 0 0 0	bit r_{15}	1 0 0 0 0	bit r_{16}	1 1 0 0 0	*Detection*
0 0 0 0 1	bit r_{12}	0 1 0 0 1	bit r_0	1 0 0 0 1	bit r_5	1 1 0 0 1	bit r_{11}
0 0 0 1 0	bit r_{13}	0 1 0 1 0	bit r_3	1 0 0 1 0	bit r_1	1 1 0 1 0	bits r_6, r_7
0 0 0 1 1	*Detection*	0 1 0 1 1	bit r_9	1 0 0 1 1	*Detection*	1 1 0 1 1	bits r_0, r_1
0 0 1 0 0	bit r_{14}	0 1 1 0 0	*Detection*	1 0 1 0 0	bit r_7	1 1 1 0 0	bit r_4
0 0 1 0 1	bit r_2	0 1 1 0 1	bits r_4, r_5	1 0 1 0 1	bit r_{10}	1 1 1 0 1	*Detection*
0 0 1 1 0	*Detection*	0 1 1 1 0	bit r_6	1 0 1 1 0	bits r_3, r_4	1 1 1 1 0	*Detection*
0 0 1 1 1	bit r_8	0 1 1 1 1	bits r_2, r_3	1 0 1 1 1	bits r_1, r_2	1 1 1 1 1	bits r_5, r_6

These functions are easily implementable. Encoding is as simple as a XOR tree or equivalent circuitry or algorithm per parity bit. **s** is calculated in the same way. Correction can be implemented using a 5-to-32 binary decoder and some logic gates. It is important to note that no correction is performed in case of "error detection".

Experimental Validation. The error coverage provided by this FUEC code has been evaluated by using a software-based implementation of its encoder and decoder circuits. Results show that all errors in $\boldsymbol{E}_+$ are successfully corrected and all those in $\boldsymbol{E}_\Delta$ are correctly detected. A similar experimentation has been carried out for all codes included in this paper.

4 Comparison with Existing Codes and Discussion

FUEC codes have some features which cannot be found in other codes, mainly the ability to deploy the desired error control functions in each part of a codeword. This hinders comparing them with other existing codes.

Let us consider the (72, 64) SEC-F7EC proposed in [1]. This UEC code has $\boldsymbol{E}_+ = \boldsymbol{E}_*^{\boldsymbol{0..6}} \cup \boldsymbol{E}_1^{\boldsymbol{7..71}}$, i.e. it has full error correction in the strongly controlled area and single error correction in the weakly controlled area. Under different fault hypothesis, UEC alternatives are very limited, but FUEC codes allow a great flexibility.

Two FUEC codes proposals are presented here, with the same values of n and k (i.e. the same redundancy). Their encoder and decoder circuits induce similar area and temporal overhead, compared to the (72, 64) SEC-F7EC. The first one has also two areas, with $\boldsymbol{E}_+ = \boldsymbol{E}_0 \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_{B2} \cup \boldsymbol{E}_2^{\boldsymbol{0..6}} \cup \boldsymbol{E}_3^{\boldsymbol{0..6}} \cup \boldsymbol{E}_{B4}^{\boldsymbol{0..6}}$, i.e. it corrects double and triple random errors, as well as 4-bit burst errors, in the strongly controlled area; and 2-bit burst errors in the weakly controlled area. Its parity-check matrix is:

$$\mathbf{H} = \begin{bmatrix}
1100000 & 00010000000100101110110001101010101011001011001000000100100100010 \\
1010000 & 01000000000100001001101100110101110100100001000111010101001001000 \\
0101000 & 00100100000100100011001011010100000000101010011010010001001001010 0 \\
0010100 & 00000001001000001001010101000011001000111100010010101010000100101 \\
0001010 & 00000010010010100100000100010010100110000010010100011001010010110 \\
0000101 & 00100010100001000100010010001000010100000101100010100000101001101 \\
0000010 & 10000000101000010010101010001000010010101000001001010110110101110 \\
0000001 & 00001000000010010000000000100101001001010010101101101101101111011
\end{bmatrix}$$

The second FUEC alternative divides the codeword in three areas (of lengths 7, 20 and 45 bits), and with $\boldsymbol{E}_+ = \boldsymbol{E}_0 \cup \boldsymbol{E}_1 \cup \boldsymbol{E}_2^{\boldsymbol{0..6}} \cup \boldsymbol{E}_3^{\boldsymbol{0..6}} \cup \boldsymbol{E}_{B4}^{\boldsymbol{0..6}} \cup \boldsymbol{E}_{B5}^{\boldsymbol{0..6}} \cup \boldsymbol{E}_{B2}^{\boldsymbol{7..26}} \cup \boldsymbol{E}_{B3}^{\boldsymbol{7..26}}$. The parity-check matrix obtained in this case is:

$$\mathbf{H} = \begin{bmatrix}
1100000\ 01100100101101101100\ 1000000000001101101001000101000100110100 10010 \\
1010000\ 10101100101011010010\ 0100000000010001010100100000100000101000 01000 \\
0101000\ 00100001010100110100\ 0010000001001101011001001001001001000000 1000100 \\
0010100\ 00010101001100100110\ 0001000000001011000111000111000100101100 100000 \\
0001010\ 00001000111001010111\ 0000100001001110000000111100111100000000 11100000 \\
0000101\ 00000010011010001101\ 0000010000000001111111111000000011100000 0011101 \\
0000010\ 10010010010010101001\ 0000001000100000000000000000111111111000 0000000011 \\
0000001\ 01001001001001001100\ 0000000111100000000000000000000000000011 11111111
\end{bmatrix}$$

These are just two examples of how single, multiple and burst error control can be combined in a code using our methodology, and how to divide the codeword in different number of areas, maintaining an accurate level of redundancy. In addition, encoder and decoder circuits are simple, fast and easy to implement from **H**. These are the main features of our proposal.

On the other hand, the computational effort to calculate **H** is a question being improved. The computational time required to completely execute the **H** generation process, as all the possible $((n-k)\times n)$ **H** matrices to check are $(2^{n-k}-1)^n$. In fact, as the backtracking algorithm generates partial matrices and uses branch & bound, a lot of combinations are discarded in early phases. However, a full algorithm execution might be unaffordable, especially with big values of n ($\geq$64). Partial executions and some tricks can be used to reduce the execution time of the algorithm. Anyway, it only affects to the design time, but not to the encoding and decoding time.

5 Conclusions and Future Work

This paper presents Flexible Unequal Error Control (FUEC) codes. The purpose of these codes is to provide enough flexibility to establish any desired number of control areas in a codeword, and to deploy in each one the adequate error control capabilities, combining single, multiple and burst error correction and/or detection.

This challenge is of great interest when the bit error rate (BER) is variable along the same codeword. In addition to VHM, FUEC codes may also result of interest in automotive, aerospace or avionics industries, where different sources of interference, noise or process variations may result in areas with variable BER (vBER). In these applications, intermittent faults in VLSI circuits increase the fault rate in the affected bits. In addition, the occurrence of multiple faults makes necessary to consider diverse error patterns in data storage and transmission.

To show the capabilities of FUEC codes, some examples have been included in the paper. Also, we have shown the flexibility, feasibility and potentials of FUEC codes with regard to existing UEC codes. Moreover, the methodology proposed to generate FUEC codes allows optimizing the silicon area and/or the speed of the encoding and decoding circuits.

An important question related to FUEC codes generation is the computational time required to complete the algorithm. Although partial executions and sub-optimal solutions can be achieved, the performance of current version has to be improved. To cope with this challenge, a parallel version of the algorithm is currently under development. However, the FUEC codes included in the paper have been obtained using the sequential version.

A final important remark relates to the type of (spatial and/or temporal) variability exhibited by the BER in different scenarios. For instance, VHM presents only spatial variability, since the BER depends on the error position in the storage media. On the other hand, intermittent faults show both spatial and temporal BER variability, since errors may vary not only their location but also their frequency or duration. In this latter case FUEC codes are also necessary, but they require adaptive detection and tolerance mechanisms. This is indeed the next big challenge to cope with our research.

Acknowledgement. This work has been partially supported by the Universitat Politècnica de València under the Research Project DesTT (SP20120806), and the Spanish Government under the Research Project ARENES (TIN2012-38308-C02-01).

References

1. Fujiwara, E.: Code Design for Dependable Systems. John Wiley & Sons (2006)
2. Constantinescu, C.: Intermittent faults and effects on reliability of integrated circuits. In: Reliability and Maintainability Symposium (RAMS 2008), pp. 370–374 (January 2008)
3. Nightingale, E.B., Douceur, J.R., Orgovan, V.: Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In: Procs. ECCS (April 2011)
4. Chou, W., Neifeld, M.: Interleaving and error correction in volume holographic memory systems. Applied Optics 37(29), 6951–6968 (1998)
5. Namba, K.: Unequal error control codes with two-level burst and bit error correcting capabilities. Electr. Comm. Japan III: Fund. Electronics Science 87, 21–29 (2004)
6. Gil, P., et al.: Fault Representativeness. Deliverable ETIE2 of Dependability Benchmarking Project. IST-2000-25245 (2002)
7. Neubauer, A., Freudenberger, J., Kühn, V.: Coding Theory: Algorithms, Architectures and Applications. John Wiley & Sons (2007)
8. LaBel, K.A.: Proton single event effects (SEE) guideline, NEPP Program web site (August 2009), `https://nepp.nasa.gov/files/18365/Proton_RHAGuide_NASAAug09.pdf`
9. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE Micro. 23(4), 14–19 (2003)
10. Shamshiri, S., Cheng, K.T.: Error-Locality-Aware Linear Coding to Correct Multi-bit Upsets in SRAMs. In: IEEE International Test Conference, pp. 1–10 (November 2010)
11. Masnick, B., Wolf, J.: On linear unequal error protection codes. IEEE Transactions on Information Theory 13(4), 600–607 (1967)
12. Dutta, A., Touba, N.A.: Reliable network-on-chip using a low cost unequal error protection code. In: Procs. DFT, pp. 3–11 (September 2007), doi:10.1109/DFT.2007.20