Design and Optimization of Adaptable BCH Codecs for NAND Flash Memories

Authors: S. Di Carlo, M. Fabiano, M. Indaco, and P. Prinetto,

Published in the Microprocessors and Microsystems Vol. 37 ,Issues. 4-5, 2013, pp. 407-419.

Design and Optimization of Adaptable BCH Codecs for NAND Flash Memories

S. Di Carlo, M. Fabiano, M. Indaco, and P. Prinetto

Department of Control and Computer Engineering
Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy
E-mail: {stefano.dicarlo, michele.fabiano, marco.indaco, paolo.prinetto}@polito.it

Abstract

NAND flash memories represent a key storage technology for solid-state storage systems. However, they suffer from serious reliability and endurance issues that must be mitigated by the use of proper error correction codes. This paper proposes the design and implementation of an optimized Bose-Chaudhuri-Hocquenghem hardware codec core able to adapt its correction capability in a range of predefined values. Code adaptability makes it possible to efficiently trade-off, in-field reliability and code complexity. This feature is very important considering that the reliability of a NAND flash memory continuously decreases over time, meaning that the required correction capability is not fixed during the life of the device. Experimental results show that the proposed architecture enables to save resources when the device is in the early stages of its lifecycle, while introducing a limited overhead in terms of area.

Key words:

Flash memories, Error correcting codes, memory testing, BCH codes

1. Introduction

NAND flash memories are a widespread technology for the development of compact, low-power, low-cost and high data throughput mass storage systems for consumer/industrial electronics and mission critical applications. Manufacturers are pushing flash technologies into smaller geometries to further reduce the cost per unit of storage. This includes moving from traditional single-level cell (SLC) technologies, able to store a single bit of information, to multi-level cell (MLC) technologies, storing more than one bit per cell.

The strong transistor miniaturization and the adoption of an increasing number of levels per cell introduce serious issues related to yield, reliability, and endurance [**? ? ? ? ?**]. Error correction codes (ECCs) must therefore be systematically applied. ECCs are a cost-efficient technique to detect and correct multiple errors [**?**]. Flash memories support ECCs by providing spare storage cells dedicated to system management and parity bit storage, while demanding the actual implementation to the application designer [**? ?**]. Choosing the correction capability of an ECC is a trade-off between reliability and code complexity. It is therefore a strategic decision in the design of a flash-based storage system. A wrong choice may either overestimate or underestimate the required redundancy, with the risk of missing the target failure rate. In fact, the reliability of a NAND flash memory continuously decreases over time, since program and erase operations are somehow destructive. At the early stage of their life-time, devices have a reduced error-rate compared to intensively used devices [**?**]. Therefore, designing an ECC system whose correction capability can be modified in-field is an attractive solution to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability.

This paper proposes the hardware implementation of an optimized adaptable Bose - Chaudhuri - Hocquenghem (BCH) codec core for NAND flash memories and a related framework for its automatic generation.

Even though there is a considerable literature about efficient BCH encoder/decoder software implementations [? ? ?], modern flash-based memory systems (e.g., Solid State Drives (SSDs)) usually resort to specific high speed hardware IP core [? ?] in order to minimize the memory latency. This is motivated by the fact that contemporary high-density MLC flash memories require a more powerful error correction capability, and, at the same time, they have to meet more demanding requirements in terms of read/write latency.

Given this premise, we will tackle a BCH hardware implementation for encoding and decoding tasks. In particular, the main contribution of the proposed architecture is its adaptability. It enables in-field selection of the desired correction capability, coupled with high optimization that minimizes the required resources. Experimental results compare the proposed architecture with typical BCH codecs proposed in the literature.

The paper is organized as follows: Section **??** shortly introduces basic notions and related works. Sections **??** and **??** present a solution to reduce resources overhead, while Section **??** and **??** overview the proposed adaptable architecture. Section **??** provides experimental results and Section **??** summarizes the main contributions of the work and concludes the paper.

2. Background and related works

Several hard- and soft-decision error correction codes have been proposed in the literature, including Hamming based block codes [? ?], Reed-Solomon codes [?], Bose-Chaudhuri-Hocquenghem (BCH) codes [?], Goppa codes [?], Golay codes [?], etc.

Even though selected classes of codes such as Goppa codes have been demonstrated to provide high correction efficiency [?], when considering the specific application domain of flash memories, the need to trade-off code efficiency, hardware complexity and performances have moved both the scientific and industrial community toward a set of codes that enable very efficient and optimized hardware implementations [? ?].

Old SLC flash designs used very simple Hamming based block codes. Hamming codes are relatively straightforward and simple to implement in both software and hardware, but they offer very limited correction capability [? ?]. As the error rate increased with successive generations of both SLC and MLC NAND flash memories, designers moved to more complex and powerful codes including Reed-Solomon (RS) codes [?] and Bose-Chaudhuri-Hocquenghem (BCH) codes [?]. Both codes are similar and belong to the larger class of cyclic codes which have efficient decoding algorithms due to their strict algebraic architecture, and enable very optimized hardware implementations. RS codes perform correction over multi-bit symbols and are better suited when errors are expected to occur in bursts, while BCH codes perform correction over single-bit symbols and better perform when bit errors are not correlated, or randomly distributed. In fact, several studies have reported that NAND flash memories manifest non-correlated or randomly distributed bit errors over a page [**?**] making BCH codes more suitable for their protection.

An exhaustive analysis of the mathematics governing BCH code is out of the scope of this paper. Only those concepts required to understand the proposed hardware implementation will be shortly discussed. It is worth to mention here that, since several publications proposed very efficient hardware implementations of Galois fields polynomial manipulations, such manipulation will be used in both encoding and decoding operations [**? ? ?**].

Given a finite Galois field $GF(2^m)$ (with $m \geq 3$), a t-error-correcting BCH code, denoted as $BCH\,[n,k,t]$, encodes a k-bit message $b_{k-1}b_{k-2}\ldots b_0$ ($b_i \in GF(2)$) to a n-bit codeword $b_{k-1}b_{k-2}\ldots b_0\,p_{r-1}p_{r-2}\ldots p_0$ ($b_i, p_i \in GF(2)$) by adding r parity bits to the original message. The number r of parity bits required to correct t errors in the n-bit codeword is computed by finding the minimum m that solves the inequality $k + r \leq 2^m - 1$, where $r = m \cdot t$. Whenever $n = k + r < 2^m - 1$, the BCH code is called *shortened* or *polynomial*. In a shortened BCH code the codeword includes less binary symbols than the ones the selected Galois field would allow. The missing information symbols are imagined to be at the beginning of the codeword and are considered to be 0. Let α be a primitive element of $GF(2^m)$ and $\psi_1\,(x)$ a primitive polynomial with α as a root. Starting from $\psi_1\,(x)$ a set of minimal polynomials $\psi_i\,(x)$ having α^i as root can be always constructed [**?**]. For the same $GF(2^m)$, different valid $\psi_1\,(x)$ may exist [**?**]. The generator polynomial $g\,(x)$ of a t-error-correcting BCH code is computed as the Least Common Multiple (LCM) among $2t$ minimal polynomials $\psi_i(x)$ $(1 \leq i \leq 2t)$. Given that $\psi_i(x) = \psi_{2i}(x)$ $(\forall i \in [1,t])$ [**?**], only t minimal polynomials must be considered and $g(x)$ can therefore be computed as:

$$g(x) = LCM\left[\psi_1(x), \psi_3(x) ..., \psi_{2t-1}(x)\right] \tag{1}$$

When working with BCH codes, the message and the codeword can be represented as two polynomials: (1) $b(x)$ of degree $k-1$ and (2) $c(x)$ of degree $n-1$. Given this representation, both the encoding and the decoding process can be defined by algebraic operations among polynomials in $GF(2^m)$. The encoding process can be expressed as:

$$c(x) = m(x) \cdot x^r + Rem\left(m(x) \cdot x^r\right)_{g(x)} \tag{2}$$

where $Rem(m(x) \cdot x^r)_{g(x)}$ denotes the remainder of the division between the message left shifted of r positions and the generator polynomial $g(x)$. This remainder represents the r parity bits to append to the original message.

The BCH decoding process searches for the position of erroneous bits in the codeword. This operation requires three main computational steps: 1) syndrome computation, 2) error locator polynomial computation, and 3) error position computation.

Given the selected correction capability t, the decoding process requires first the computation of $2t$ syndromes of the codeword $c(x)$, each associated with one of the $2t$ minimal polynomials $\psi_i(x)$ generating the code. Syndromes are calculated by first computing the remainders $R_i(x)$ of the division between $c(x)$ and each minimal polynomial $\psi_i(x)$. If all remainders are null, $c(x)$ does not contain any error and the decoding stops. Otherwise, the $2t$ syndromes are computed by evaluating each remainder $R_i(x)$ in α^i: $S_i = R_i(\alpha^i)$. Practically, according to (**??**), given that $\psi_i(x) = \psi_{2i}(x)$, only

t remainders must be computed and evaluated in $2t$ elements of $GF(2^m)$.

The most used algebraic method to compute the coefficients of the error locator polynomial from the syndromes is the Berlekamp-Massey algorithm [?]. Since the complexity of this algorithm grows linearly with the correction capability of the code, it enables efficient hardware implementations. The equations that link syndromes and error locator polynomial can be expressed as:

$$\begin{pmatrix} S_{t+1} \\ S_{t+2} \\ \vdots \\ S_{2t} \end{pmatrix} = \begin{pmatrix} S_1 & S_2 & \dots & S_t \\ S_2 & S_3 & \dots & S_{t+1} \\ \vdots & \vdots & & \vdots \\ S_t & S_{t+1} & \dots & S_{2t-1} \end{pmatrix} \begin{pmatrix} \lambda_t \\ \lambda_{t-2} \\ \vdots \\ \lambda_0 \end{pmatrix} \tag{3}$$

The Berlekamp-Massey algorithm iteratively solves the system of equations defined in (??) using consecutive approximations.

Finally, the Chien Machine searches for the roots of the error locator polynomial $\lambda(x)$ computed by the Berlekamp-Massey algorithm [?]. It basically evaluates the polynomial $\lambda(x)$ in each element α^i of $GF(2^m)$. If α^i satisfies the equation $1 + \lambda_1\alpha^i + \lambda_2\alpha^{2i} + \dots + \lambda_t(\alpha^i)^t = 0$, α^i is a root of the error locator polynomial $\lambda(x)$, and its reciprocal $2^m - 1 - i$ reveals the error position. In practice, this computation is performed exploiting the iterative relation:

$$\lambda\left(\alpha^{j+1}\right) = \lambda_0 + \sum_{k=1}^{t-1}\left[\lambda_k\left(\alpha^j\right)^k\right]\alpha^k \tag{4}$$

Several publications proposed optimized hardware implementations of BCH codecs with fixed correction capability [? ? ? ? ? ?]. However, to the best of our knowledge, only Chen et al. proposed a solution allowing limited adaptation by extending a standard BCH codec implementation [?]. One of the main contributions of Chen et al. is a Programmable Parallel Linear Feedback Shift Register (PPLFSR), whose generic architecture is reported in Fig. ??. It enables to dynamically change the generator polynomial of the LFSR. This is a key feature in the implementation of an adaptable BCH encoder.

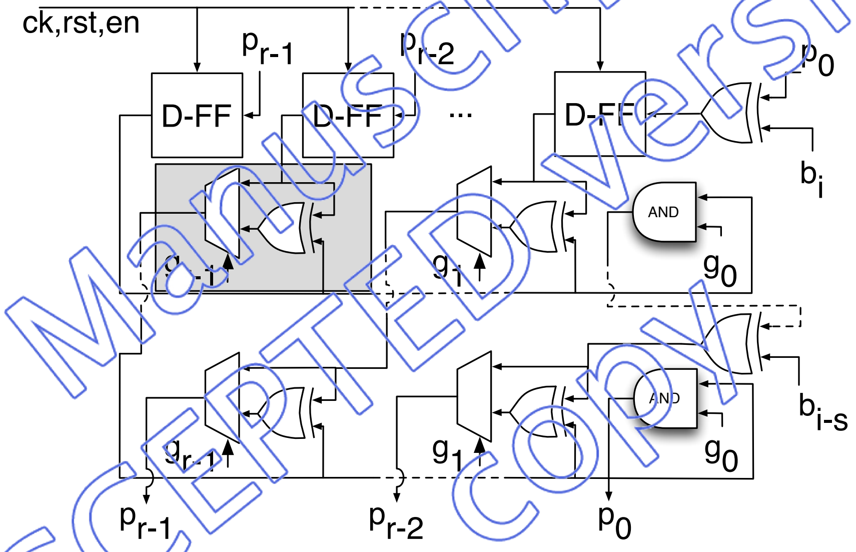


Figure 1: Architecture of a r-bit PPLFSR with s-bit parallelism.

The gray box of Fig. ?? highlights the basic adaptable block of this circuit. It exploits a multiplexer, controlled by one of the coefficients of the desired divisor polynomial, to dynamically insert an XOR gate at the output of one of the related D-type flip-flops composing the register. The s vertical stages of the circuit implement the parallelism of the PPLFSR computing the state at clock cycle $i + s$, based on the state at cycle i. However, this solution has high overhead. In fact such PPLFSR is able to divide by all possible r-bit polynomials, while just well selected divisor polynomials are required.

Although Chen at al. deeply analyze the encoding process and the issues related to the storage of parity bits, the decoding process is scarcely analyzed, without providing details on how adaptability is achieved. Four different correction modes, namely $t = (9, 14, 19, 24)$ are considered in [?] for a BCH code defined on $GF(2^{13})$ with a block size of 512B (every 2KB page of the flash is split in four blocks). The selection of the 4 modes is based on considerations about the number of parity bits to store. However, there is no provision to understand whether additional modes can be easily implemented. As an example, when selecting correction modes in which the size of the codeword is not a multiple of the parallelism of the decoder, alignment problems arise, which are completely neglected in the paper.

3. Optimized Architectures of Programmable Parallel LFSRs

In this section, we will introduce an optimized block to perform an adaptable remainder computation. In fact, one of the most recurring operations in BCH encoding/decoding is the remainder computation between a polynomial representing a message to encode/decode and a generator/minimal polynomial of the code, that depends on the selected correction capability. The PPLFSR of Fig. **??** can perform this operation [?].

A r-bit PPLFSR can potentially divide by any r-bit polynomial by properly controlling its configuration signals $(g_0 \dots g_{r-1})$. However, in BCH encoding/decoding, even considering an adaptable codec, just well selected divisor polynomials are required (e.g., the generators polynomials $g_9(x)$, $g_{14}(x)$, $g_{19}(x)$, $g_{24}(x)$ of the four implemented correction modes of [**?**]). This computational block is therefore highly inefficient. Moreover, the set of divisor polynomials required in a BCH codec usually share common terms among each other. Such terms can be exploited to generate an optimized PPLFSR (OPPLFSR) architecture.

Let us consider, as an example, the design of a r=15-bit programmable LFSR able to divide by two polynomials $p_1(x) = x^{15}+x^{13}+x^{10}+x^5+x^3+x+1$ and $p_2(x) = x^{13}+x^{12}+x^{10}+x^5+x^4+x^3+x^2+x+1$ using a s=8-bit parallelism.

A traditional PPFLSR implementation would require $15 \times 8 = 120$ gray boxes (i.e., 120 XORs-MUXs). According to this implementation, this PP-LFSR could divide by any $2^{15} = 32,768$ possible 15-bit polynomials, even if just 2 polynomials (i.e., the 0.006% of its full potential) are required.

An analysis of the target divisor polynomials can be exploited to optimize the PPLFSR architecture. Table **??** reports the binary representation of the two polynomials.

Looking at Table **??**, three categories of polynomial terms can be identified:

1. Common terms (represented in bold), i.e., terms defined in all considered polynomials (x^{13}, x^{10}, x^5, x^3, x, and 1 in Table **??**). For these terms, an XOR will be always required in the PPLFSR, thus saving the area dedicated to the MUX and the related control logic.
2. Missing terms (represented in underlined italic zeros), i.e., terms not defined in any of the considered polynomials, (x^{14}, x^{11}, x^9, x^8, x^7 and x^6 in Table **??**). For these terms both the XOR and the related MUX

Table 1: An example of the representation of $p_1(x)$ and $p_2(x)$

	x^{15}	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^{9}	x^{8}	x^{7}	x^{6}	x^{5}	x^{4}	x^{3}	x^{2}	x^{1}	1
$p_1(x)$	1	$\underline{\mathit{0}}$	**1**	0	$\underline{\mathit{0}}$	**1**	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	**1**	0	**1**	0	**1**	**1**
$p_2(x)$	0	$\underline{\mathit{0}}$	**1**	1	$\underline{\mathit{0}}$	**1**	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	$\underline{\mathit{0}}$	**1**	1	**1**	1	**1**	**1**

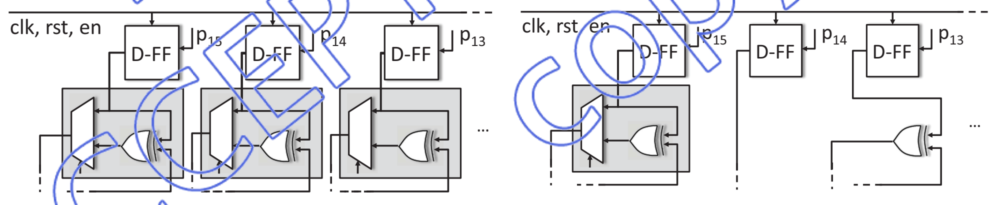
can be avoided.

3. Specific terms, i.e., terms that are specific of a subset of the considered polynomials (x^{15}, x^{12}, x^4, x^2 in Table **??**). These terms are the only ones actually required.

We can therefore implement an optimized programmable LFSR (OPPLFSR) with three main building blocks:

1. each common present term (i.e., columns of all "1" of Table **??**) needs an XOR, only;
2. each common absent term (i.e., columns of all "0" of Table **??**) needs neither XOR nor MUX;
3. each specific term has a gray box, as Fig. **??**;

Fig. **??** shows the resulting design for the portion x^{15}, x^{14} and x^{13}.



(a) PPLFSR (b) OPPLFSR

Figure 2: Example of the resulting PPLFSR (a) and OPPLFSR (b) with 8-bit parallelism for x^{15}, x^{14} and x^{13} of $p_1(x)$ and $p_2(x)$

This optimization also applies on polynomials with very different lengths. As an example, an OPPLFSR with single bit parallelism and able to divide by $p_1(x) = x^{225} + x + 1$ and $p_2(x) = x + 1$, would only require a single adaptable block, compared to the 226 blocks required by a normal PPLFSR.

Furthermore, the advantage of the OPPLFSR increases with the parallelism of the block. In fact, with the same 2 polynomials, a 8-bit OPPLFSR would require 8 adaptable blocks compared to $226 \times 8 = 1,808$ adaptable blocks of a traditional PPLFSR.

For sake of generality, Fig. **??** shows the high-level architecture of a generic OPPLFSR. Such a block is able to divide by a set $p_1(x), ..., p_M(x)$ of polynomials. We denote with q the number of required gray boxes.

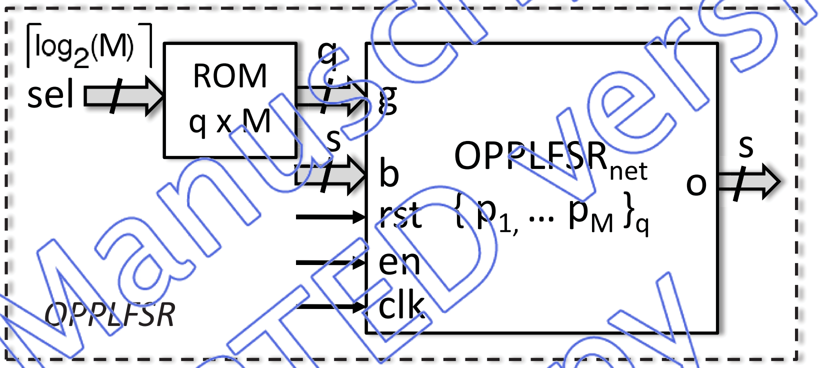


Figure 3: High-level architecture of the OPPLFSR

The OPPLFSR interface includes: a s-bit input port (`b`) used to feed the data, a $\lceil \log_2(\mathrm{M}) \rceil$-bit input port (`sel`) used to select the polynomial of the division, and a s-bit port (`o`) providing the result of the division. Two blocks compose the OPPLFSR: *OPPLFSR$_{net}$* and *ROM*. The OPPLFSR$_{net}$ represents the complete network, partially shown in the example of Fig. **??**. Given the output of the ROM, the q-bit signal `g` controls the MUXs of the q gray boxes (Fig. **??**) according to the selected polynomial. The ROM is optimized accordingly with the design of the OPPLFSR, which leads to a reduced ROM and to a lower area overhead w.r.t. a full PPLFSR.

4. BCH Code Design Optimization

In this section, we address first the issue of choosing the most suitable set of polynomials for an optimized adaptable BCH code. Then, we propose a novel block, shared between the adaptable BCH encoder and the decoder, which reduces the area overhead of the resulting codec core.

4.1. The choice of the set of polynomials

The optimization offered by the OPPLFSR introduced in Section **??**, may become ineffective if not properly exploited. It depends on the number and on the terms of the shared divisor polynomials implemented in the block. As an example, an excessive number of shared polynomials may make it difficult to find common terms, leading to an unwilled increase of the area overhead. Therefore, the choice of the polynomials to share is critical and must be properly tailored to the overall design.

Let us denote by Ω the set of t generators $g_i(x)$ and t minimal polynomials ψ_i which fully characterize an adaptable BCH code (see Section **??**). Since for $GF(2^m)$ several primitive polynomials $\psi_i(x)$ can be used to define the code, several set Ω_i can be constructed. Choosing the most suitable set Ω_i is critical to obtain an effective design of the OPPLFSR. On the one hand, it can be shown that the complexity of Ω_i increases with m [**? ? ?**]. On the other hand, the current trend is to adopt BCH codes with high values of m (e.g., $GF(2^{15})$) because current flash devices features a worse bit error rate [**?**]. Therefore, a simple visual inspection of each set Ω_i is not feasible to find the most suitable set of polynomials. An algorithmic approach is therefore mandatory.

Each set Ω_i can be classified resorting to a *Maximum Correlation Index* (MCI). We define as $MCI\left(p_1, p_2, ..., p_N\right)$ the maximum number of common terms shared by a generic set of polynomials $p_1, p_2, ..., p_N$. As an example, the polynomials of Table **??** have $MCI\left(p_1, p_2\right) = 12$.

In the sequel, we introduce an algorithm to assess each set Ω_i according to its MCI. Given $i = \{1, ..., Y\}$, for each set Ω_i:

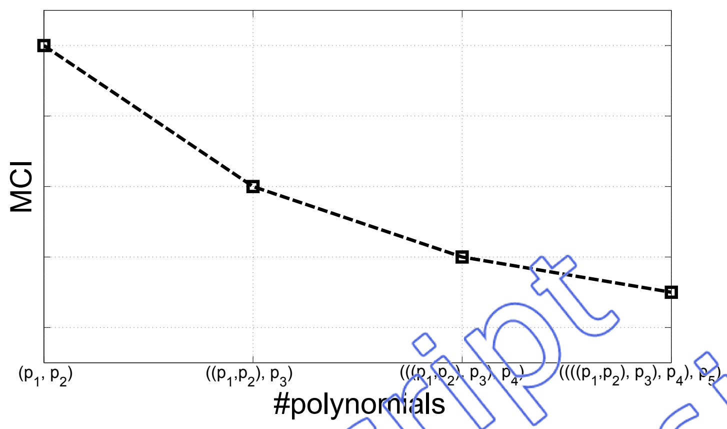
1. consider $\Omega_i = \{p_1, ..., p_N\}$ and $v_0 = p_1$;
2. determine the polynomial p_h such that the partition $S_{i,1} = (v_0, p_h)$ has the maximum $MCI\left(v_0, p_h\right)$, where $h = \{1, ..., N\}$ and $p_h \neq v_0$;
3. determine the polynomial p_k such that the partition $S_{i,1} = \left(\left(v_0, p_h\right), p_k\right)$ has the maximum $MCI\left(v_0, p_h, p_k\right)$, where $k = \{1, ..., N\}$ and $p_k \neq p_h \neq v_0$;
4. repeat step 3 until all polynomials have been considered in the partition $S_{i,1}$;
5. change the starting polynomial to the next one, e.g., $v_0 = p_2$, considering $S_{i,2}$ and repeat steps 2-4;
6. when $v_0 = p_N$, consider the next set Ω_{i+1};

The algorithm ends when all sets Ω_i have been analyzed. For each Ω_i, the output is a set of partitions:

$$S_{i,j} = \{S_{i,1}, S_{i,2}, ..., S_{i,N}\} \tag{5}$$

Fig. **??** graphically shows the MCI of two partitions generated from two different starting points, for an hypothetical set Ω_i.

Fig. **??** shows that MCI always has a decreasing trend with the size of the partition S. This is straightforward since adding a polynomial may only



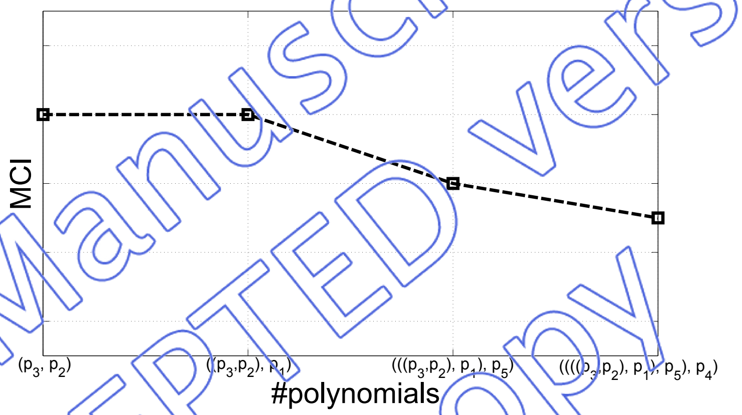


Figure 4: MCI examples of two hypothetical partitions $S_{i,1}$ and $S_{i,2}$

decrease or keep constant the current value of MCI. The curves, reported in **??**, are critical in the choice of the most suitable set of polynomials for an optimized BCH code. For each partition $S_{i,j}$ with $j = \{1...N\}$, we can compute the average MCI (MCI_{avg}) as:

$$MCI_{avg}(S_{i,j}) = \frac{1}{N}\sum_{l=1}^{N-1} MCI_l \tag{6}$$

Eq. **??** applies to each set Ω_i where $i = \{1...Y\}$.

The best partition of the set Ω_i is then computed selecting the one with maximum MCI_{avg}:

$$S_{best_i} = \underset{j}{argmax}\left[MCI_{avg}\left(S_{i,j}\right)\right] \tag{7}$$

Finally, Eq. **??** compares the best partition of each set Ω_i to find the best set of polynomials:

$$S_{bestBCH} = \underset{i}{argmax}\left[S_{best_i}\right] \tag{8}$$

Eq. **??** defines the family of polynomials $S_{bestBCH}$, with the maximum average number of common terms.

Table 2: An example of Ω_i

	x^6	x^5	x^4	x^3	x^2	x^1	1
p_1	1	0	1	0	0	1	0
p_2	1	1	0	1	0	1	1
p_3	1	0	1	1	1	1	1
p_4	0	1	1	0	0	0	1
p_5	1	1	0	1	1	0	1
p_6	0	0	1	0	0	1	1

Let us provide an example to support the understanding of the algorithm. Suppose to consider a single set Ω_i composed of the polynomials of Table **??**. The steps of the algorithm are:

1. Let us start with $v_0 = p_1$

2. We first evaluates $MCI\left(p_1, p_2\right) = 3$, $MCI\left(p_1, p_3\right) = 4$, $MCI\left(p_1, p_4\right) = 3$. Since $MCI\left(p_1, p_3\right) = 4$ is the maximum, the resulting partition is $S_{i,1} = \{p_1, p_3\}$

3. The next step considers $MCI(\left(p_1, p_3\right), p_2) = 3$ and $MCI(\left(p_1, p_3\right), p_4) = 3$. It is straightforward that the choice of either p_2 or p_4 does not affect the final value of the MCI_{avg}.

Given Ω_i with starting point p_1, it can be shown that the final partition is $S_{i,1} = \{\left(\left(p_1, p_3\right), p_4\right), p_2\}$ with a $MCI_{avg} = {}^{(4+3+3)}/_4 = 2.5$ from Eq. **??**.

The complete algorithm iterates this computation for all possible starting points. Fig. **??** graphically shows the output of the MCI associated with each partition $S_{i,j}$ calculated for the following starting point $j = \{1, 2, 3, 4\}$.

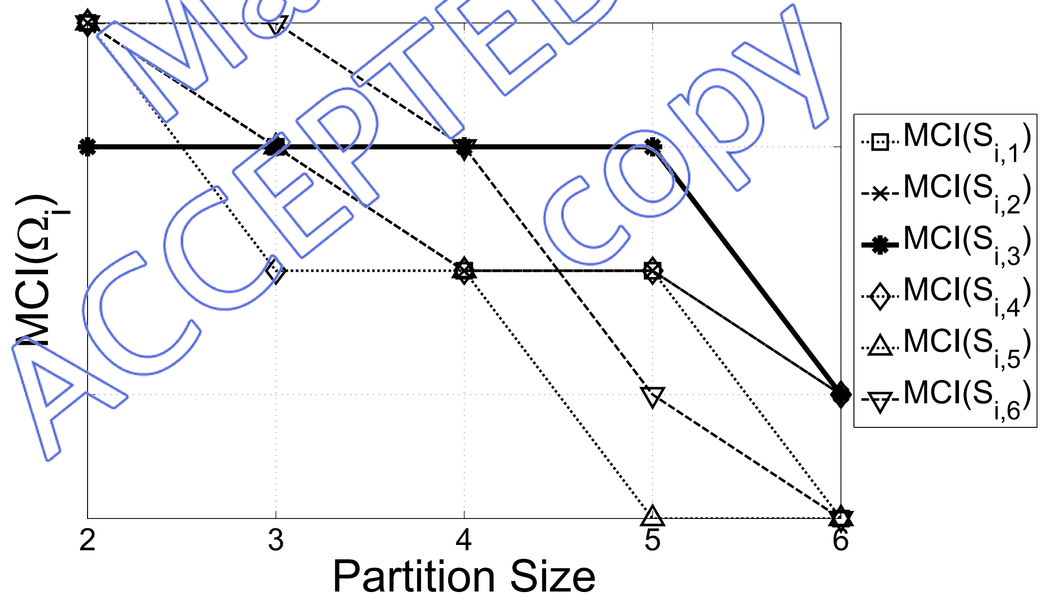


Figure 5: The MCI Trend of Table **??**

According to Eq. **??**, $S_{i,2}$ (the bold line) is the S_{best_i} of the example of

Table **??**, with a $MCI_{avg}\left(S_{i,j}\right) = 4$.

4.2. Shared Optimized Programmable Parallel LFSRs

Let us assume to design an adaptable BCH code with correction capability from 1 up to t_M. Such a code needs to compute remainders of the division of:

- the message $m\left(x\right)$ by (potentially) all generator polynomials from g_1 up to g_{t_M}, for the encoding (**??**);
- the codeword $c\left(x\right)$ by (potentially) all minimal polynomials from $\psi_1\left(x\right)$ up to $\psi_{2t_M-1}\left(x\right)$, to compute the set of syndromes required during the decoding phase.

In a traditional implementation, these computations are performed by two separate set of LFSRs. In this paper, we propose to devise a shared set of LFSRs able to: (i) perform all these computations, and (ii) reduce the overall cost in terms of resources overhead. Therefore, we can adopt the same shared set of LFSRs both in the encoding and decoding processes. This is possible since in a flash memory these operations are, in general, not required at the same time.

The OPPLFSR, introduced in Section **??**, is the main building block of the set of shared LFSRs. Therefore, we will refer hereafter to such set of LFSRs as shared OPPLFSR (shOPPLFSR). Fig. **??** shows the high-level architecture of the shOPPLFSR. Its interface includes: a s-bit input port (`IN`) used to input the data to be divided, a $\lceil \log_2\left(\mathrm{N}\right)\rceil$-bit input port (`en`) used to enable each OPPLFSR, an input port (`sel`) used to select the proper polynomial by which each OPPLFSR has to divide, and a N × s-bit port (p) providing the result of the division.

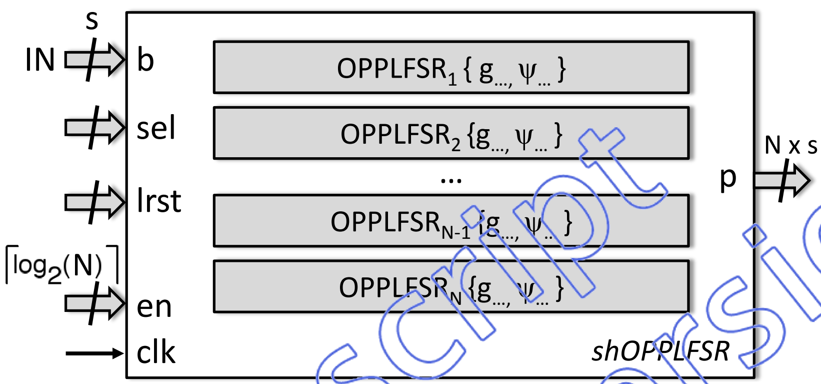


Figure 6: The shOPPLFSR architecture is composed by multiple OPPLFSRs

Given N OPPLFSRs and a maximum correction capability t_M, each OPPLFSR$_i$ performs the division by a set of generator polynomials $g(x)$ and minimal polynomials $\psi(x)$. Such shOPPLFSR can be seen as an optimized programmable LFSR able to:

- divide by all generator polynomials from $g_1(x)$ to $g_{t_M}(x)$;
- divide by specific subsets of minimal polynomials from Eq. **??**, as well.

An improper choice of the shared polynomials $g(x)$ and $\psi(x)$ can dramatically reduce the performance of the overall BCH codec. Also the partitioning strategy adopted is critical to maximize the optimization in terms of area, minimizing the impact on the latency of encoding/decoding operations.

The algorithm presented in Section **??** provides a valuable support for the exploration of this huge design space. In fact, the proposed method can be exploited to properly partition polynomials into the different OPPFLSRs of

Fig. **??**, in order to maximize the optimization of the resulting shOPPFLSR. Such optimization should not be obtained following blindly the outcomes of the algorithm, but always tailoring them to the specific design. Regarding this topic, Section **??** provides more details about our experimental setup and the related experimental results.

5. Adaptable BCH Encoder

In this section, we propose an adaptable BCH encoder which exploits the shOPPLFSR of Section **??**. According to the BCH theory, the shOPPLFSR of Fig. **??** is a very efficient circuit to perform the computation expressed in Eq. **??**. However, in the encoding phase, the message $m(x)$ must be multiplied by x^r before calculating the reminder of the division by $g(x)$ (see Eq. **??**). This can be obtained without significant modifications of the architecture of shOPPFLSR. It is enough to input the bits of the message directly in the most significant bit of the LFSR, instead than starting from least significant bit. Fig. **??** shows the high-level architecture of the adaptable encoder.

The encoder's interface includes: a s-bit input port (`IN`) used to input the k-bit message to encode starting from the most significant bits, a $\lceil \log_2 (t_M) \rceil$-bit input port (`t`) selecting the requested correction capability in a range between 1 and t_M, a **`start`** input signal used to start the encoding process and a s-bit output port (`OUT`) providing the r parity bits. Three blocks compose the encoder: a *shOPPLFSR*, a *flush logic* and a *controller*.

The shOPPLFSR performs the actual parity bits computation. According to the BCH theory, adaptation is achieved by supporting the computation

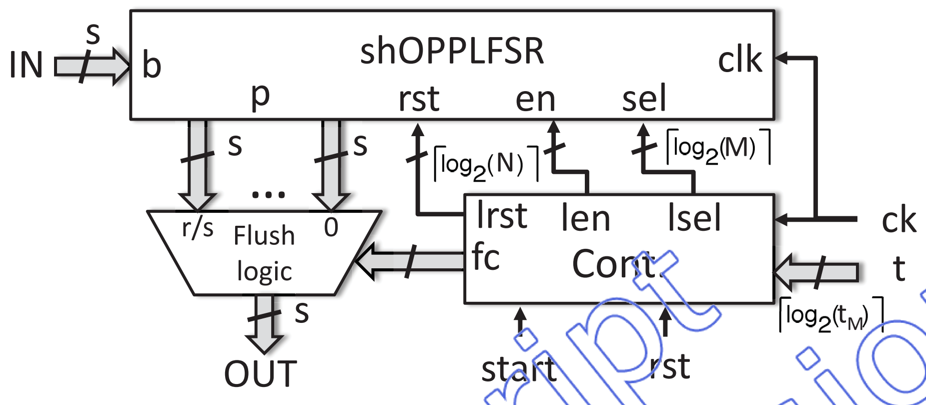


Figure 7: High-level architecture of the adaptable encoder highlighting the three main building blocks and their main connections.

of remainders with t_M generator polynomials, one for each value t may assume. The controller achieves this task in two steps: (i) enabling the proper OPPLFSR through the `len` signal, and (ii) selecting the proper polynomial through the `lsel` signal, according to the desired correction capability `t`. Then, it manages the overall encoding process based on two internal parameters: 1) the number of s-bit words composing the message (fixed at design time) and 2) the number of produced s-bit parity words, that depends on the selected correction capability. The flush logic splits the r parity bits into s-bit words, providing them in output, one per clock cycle.

To further optimize the encoding and the decoding process, since in a flash memory these operations are not required at the same time, the encoder's shOPPLFSR can be merged with the shOPPLFSRs that will be employed in the syndrome computation (see Section ??), thus allowing additional area saving.

6. Adaptable BCH Decoder

Fig. **??** presents the high-level architecture of the proposed adaptable decoder. The decoder's interface includes: a $s-$bit input port (`IN`) used to input the $n-$bit codeword to decode (starting from the most significant bits), a $\lceil \log_2 (t_M) \rceil -$bit input port (`t`) to select the desired correction capability, a `start` input signal to start the decoding and a set of output ports providing information about detected errors. In particular:

- `deterr` is a $\lceil \log_2 (t_M) \rceil -$bit port providing the number of errors that have been detected in a codeword. In case of decoding failure it is set to 0;
- `erradd` and `errmask` provide information about the detected error positions. Assuming the codeword split into $h-$bit words, `erradd` is used as a word address in the codeword and `errmask` is a $h-$bit mask whose asserted bits indicate detected erroneous bits in the addressed word. The parallelism h of the error mask depends on the parallelism of the Chien machine, as explained later in this section;
- `vmask` is asserted whenever a valid error mask is available at the output of the decoder;
- `fail` is asserted whenever an error occurred during the decoding process (e.g., the number of errors is greater than the selected correction capability);
- `end` is asserted when the decoding process is completed.

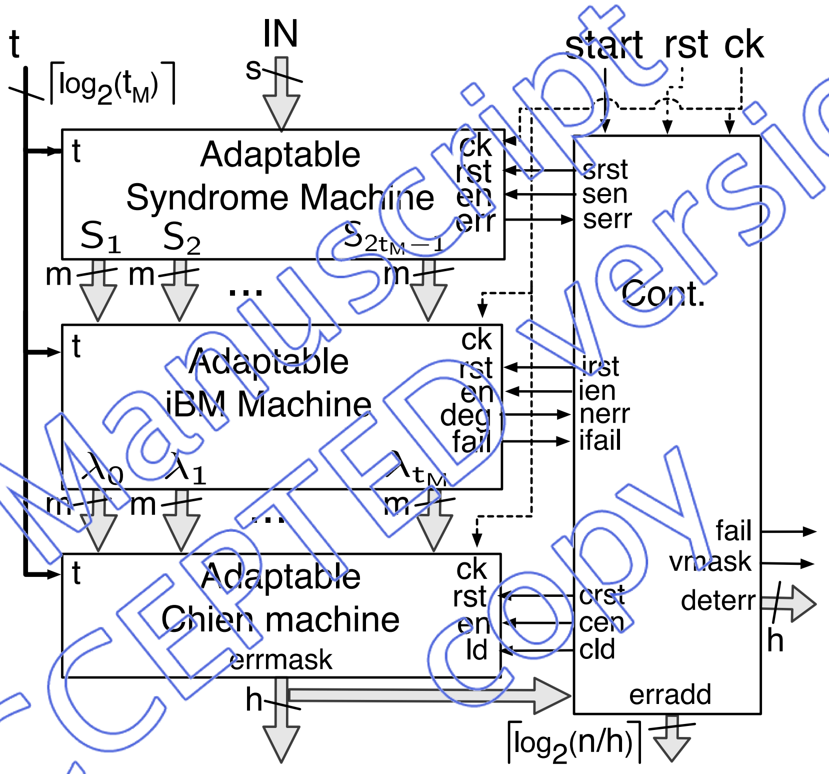


Figure 8: High-level architecture of the adaptable decoder, highlighting the four main building blocks: the adaptable syndrome machine, the adaptable iBM machine, the adaptable Chien machine, and the controller in charge of managing the overall decoding process

The full decoder therefore includes four main blocks: (1) the *Adaptable Syndrome Machine*, computing the syndromes of the codeword, (2) the*Adaptable inversion-less Berlekamp Massey (iBM) Machine*, that elaborates the syndromes to produce the error locator polynomial, (3) the *Adaptable Chien Search Machine* in charge of searching for the error positions, and (4) the *Controller* coordinating the overall decoding process.

6.1. Adaptable Syndrome Machine

Fig. **??** shows the high-level architecture of the proposed adaptable syndrome machine with correction capability $1 \leqslant t \leqslant t_M$.

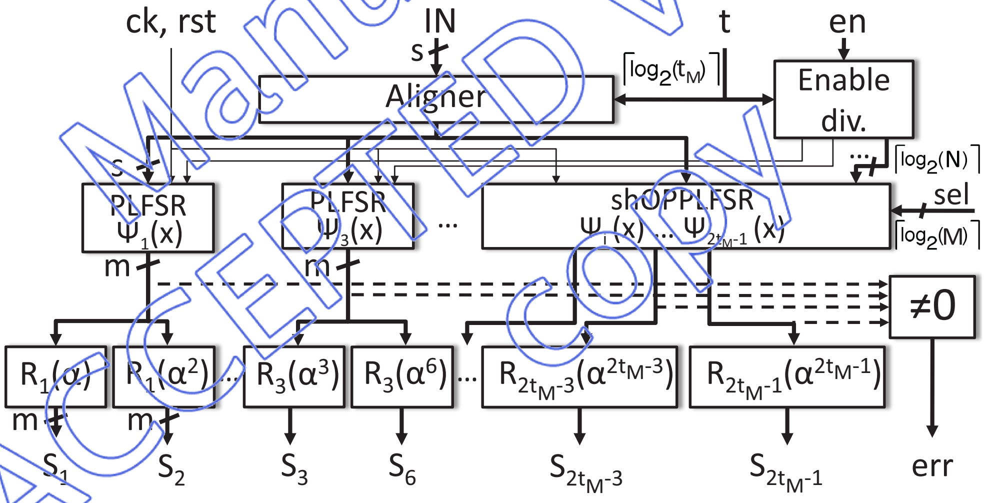

ck, rst
IN
s
t
en
$\lceil \log_2(t_M) \rceil$
Aligner
Enable div.
$\lceil \log_2(N) \rceil$
s
PLFSR $\Psi_1(x)$
PLFSR $\Psi_3(x)$
shOPPLFSR $\Psi_i(x) \dots \Psi_{2t_M-1}(x)$
sel
$\lceil \log_2(M) \rceil$
m
≠0
$R_1(\alpha)$
$R_1(\alpha^2)$
$R_3(\alpha^3)$
$R_3(\alpha^6)$
$R_{2t_M-3}(\alpha^{2t_M-3})$
$R_{2t_M-1}(\alpha^{2t_M-1})$
S_1
S_2
S_3
S_6
S_{2t_M-3}
S_{2t_M-1}
err

Figure 9: Architecture of the adaptable Syndrome Machine

According to Section **??**, remainders can be calculated by a set of Parallel LFSRs (PLFSRs) whose architecture is similar to the one of the PPLFSR of Fig. **??**, with the only difference that the characteristic polynomial is fixed (XOR gates are inserted only where needed, without multiplexers).

Each PLFSR computes the remainder of the division of the codeword by a different minimal polynomial $\psi_i(x)$. Given two correction capabilities t_1 and t_2 with $t_1 < t_2 \leq t_M$, the set of $2t_1$ minimal polynomials generating the code for t_1 is a subset of those generating the code for t_2. To obtain adaptability of the correction capability in a range between 1 and t_M, the syndrome machine can therefore be designed to compute the maximum number t_M of remainders required to obtain $2t_M$ syndromes. Based on the selected correction capability t, only the first t PLFSRs out of the t_M available in the circuit are actually enabled through the *Enable div.* network of Fig. **??**.

A full parallel syndrome calculator, including t_M PLFSRs, requires a considerable amount of resources that are underutilized in the early stages of the flash lifetime when reduced correction capability is required. To optimize the adaptable syndrome machine and to trade-off between complexity and performance, we exploit the shOPPLFSR introduced in Section **??**. The architecture proposed in Fig. **??** includes two sets of LFSRs for remainder computation: (i) conventional PLFSRs, and (ii) shOPPLFSR. Conventional PLFSRs are exploited for parallel fast computation of low order syndromes required when the requested correction capability is below a given threshold. shOPPLFSR is designed to divide for selected groups of minimal polynomials not covered by the fixed PPLFSRs. It represents a shared resource utilized when the requested correction capability increases. It enables area reduction at the cost of a certain time overhead. The architectural design, chosen for the fixed PLFSRs and the OPPLFSR, enables to trade-off hardware complexity and decoding time, as it will be discussed in Section **??**.

It is worth to mention here that the parallel architecture of the PLFSR,

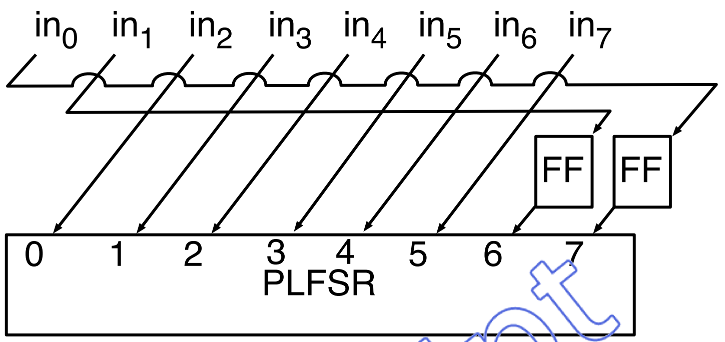


Figure 10: Example of the schema of a byte aligner for $t = 2$ and $s = 8$

coupled with the adaptability of the code, introduces a set of additional word alignment problems that must be addressed to correctly adapt the syndrome calculation to different values of t. The syndrome machine receives the codeword in words of s bits, starting from the most significant word. When the number of parity bits does not allow to align the codeword to the parallelism s, the unused bits of the last word are filled with 0. To correctly compute each syndrome, the parity bit r_0 of the codeword must enter the least significant bit of each LFSR. The aligner block of Fig. **??** assures this condition by properly right-shifting the codeword while it is input into the syndrome machine. Let us consider the following example: k = 2KB, $m = 15$, $t = 2$, $s = 8$ and therefore $r = m \cdot t = 30$. Since 30 is not multiple of $s = 8$, the codeword is filled with two zeros and $\mathtt{p}_0$ is saved in position 2 of the last byte of the codeword ($\mathtt{m}_{2047}\,\mathtt{m}_{2046}...\mathtt{m}_1\,\mathtt{m}_0\,\mathtt{p}_{29}\,\mathtt{p}_{28}...\mathtt{p}_1\,\mathtt{p}_0\,0\,0$). In this case the PLFSRs require a 2-bit alignment, implemented by the network of Fig. **??**. It simply delays the last 2 input bits resorting to two flip-flops, whose initial state has to be zero, and properly rotates the remaining input bits. Changing the correction capability of the decoder changes the number of parity bits of the codeword, and therefore the required alignment. Given the parallelism s of the decoder, a maximum of s alignments must be provided and implemented in the *Aligner* block of Fig. **??**.

With the proper alignment, the PLFSRs can perform the correct division and the evaluators can provide the required syndromes. The evaluators are simple combinational networks involving XOR operations, according to the Galois Fields theory (the reader may refer to [?] for specific implementation details).

6.2. Adaptable Berlekamp Massey Machine

In our adaptable codec we implemented the inversion-less Berlekamp-Massey (iBM) algorithm proposed in [?] which is able to compute the error locator polynomial $\lambda(x)$ in t iterations.

The main steps of the computation are reported in Alg. **??**. At iteration i (rows 2 to 12), the algorithm finds an error locator polynomial $\lambda(x)$ whose coefficients solve the first i equations of (**??**) (row 4). It then tests if the same polynomial solves also $i+1$ equations (row 5). If not, it computes a discrepancy term δ so that $\lambda(x)+\delta$ solves the first $i+1$ equations (row 9). This iterative process is repeated until all equations are solved. If, at the end of the iterations, the computed polynomial has a degree lower than t, it correctly represents the error locator polynomial and its degree represents the number of detected errors; otherwise, the code is unable to correct the given codeword.

The architecture of the iBM machine is intrinsically adaptive as long as one guarantees that the internal buffers and the hardware structures are sized to deal with the worst case design (i.e., $t = t_M$). The coefficients of $\lambda\,(x)$ are

Algorithm 1 Inversion-less Berlekamp-Massey alg.

```
1: $\lambda(x) = 1$, $k(x) = 1$, $\delta = 1$
2: for i = 0 to t − 1 do
3:     $d = \sum_{j=1}^{t} (\lambda_j \cdot S_{2i-j})$
4:     $\lambda(x) = \delta\lambda(x) + d \cdot x \cdot k(x)$
5:     if $d = 0$ OR $Deg(\lambda(x)) > i$ then
6:         $k(x) = x^2 \cdot k(x)$
7:     else
8:         $k(x) = x \cdot k(x)$
9:         $\delta = d$
10:    end if
11:    i=i+1
12: end for
13: if $Deg(\lambda(x)) < t$ then
14:    output $\lambda(x)$, $Deg(\lambda(x))$
15: else
16:    output FAILURE
17: end if
```

$m-$bit registers whose number depends on the correction capability. In the worst case, up to t_M coefficients must be stored for each polynomial.

The adaptable iBM machine therefore includes two $m-$bit register files with t_M registers to store these coefficients. Whenever the requested correction capability is lower than t_M some of the registers will remain unused. The number of multiplications performed during the computations also depends on t. Row 3 requires t multiplications, while row 4 requires t multiplications to compute $\delta\lambda_i(x)$ and t multiplications to compute $d \cdot x \cdot k(x)$.

We implemented a serial iBM Machine including 3 multipliers for GF(2^m) to perform multiplications of rows 3 and 4. It can perform each iteration of the iBM algorithm in $2t$ clock cycles (t cycles for row 3 and t cycles for row 4) achieving a time complexity of $2t^2$ clock cycles. This implementation is a good compromise between performance and hardware complexity. An input t dynamically sets the number of iterations of the algorithm, thus implementing the adaptation.

6.3. Adaptable Chien Machine

The overall architecture of the proposed adaptable Chien Machine is shown in the Fig. **??**. The machine first loads into t_M m-bit registers the coefficients from λ_1 to λ_{t_M} of the error locator polynomial $\lambda(x)$ computed by the iBM machine ($\texttt{ld} = \texttt{0}$). The actual search is then started ($\texttt{ld} = \texttt{1}$). At each clock cycle, the block performs h parallel evaluations of $\lambda(x)$ in GF(2^m) and outputs a h-bit word, denoted as `errmask`. Each bit of `errmask` corresponds to one of the h candidate error locations that have been evaluated. Asserted bits denote detected errors. This mask can then be XORed (outside the Chien Machine) with the related bits of the codeword in order to correct the detected erroneous bits.

The architecture of Fig. **??** provides an adaptable Chien machine with lower area consumption than other designs [**?**], having, at the same time, a marginal impact on performance. Four interesting features contribute to such optimization: (i) constant multipliers substructure sharing, (ii) adaptability to the correction capability, (iii) improved fast skipping to reduce the decoding time, and (iv) reduced full GF multipliers area. In the sequel, we briefly address each feature.

The first feature is represented by the optimized GF Constant Multipliers (optGFCM) networks of Fig. **??**. The h parallel evaluations are based on

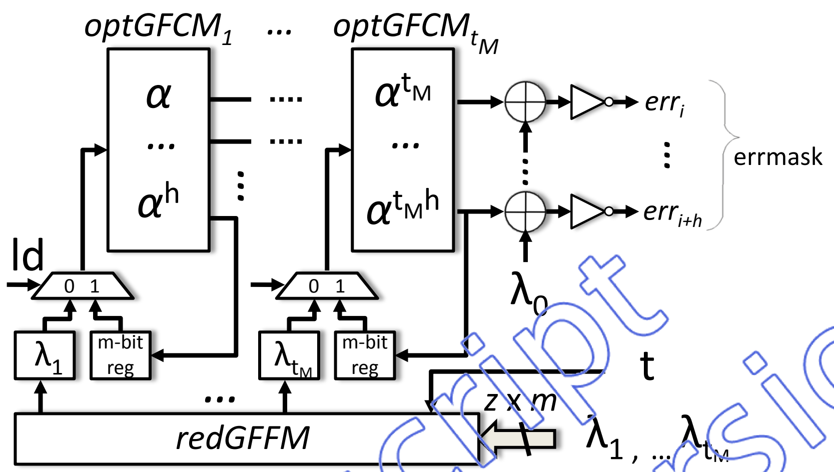


Figure 11: Architecture of the proposed parallel adaptable Chien Machine with parallelism equal to h

equation (??). In the worst case ($t = t_M$), the parallel evaluation of equation (??) requires a matrix of $t_M \times h$ constant Galois multipliers. They multiply the content of the t_M registers by $\alpha, \alpha^2, ..., \alpha^{t_M}$, respectively. However, we can note that each column of constant GF multipliers shares the same multiplicand. Therefore, we can iteratively group their best-matching combinations [?] into the t_M optGFCM networks of Fig. ??. Such optGFCMs provide up to 60% reduction of the hardware complexity of the machine with no impact on performance.

The second feature is the adaptability of the Chien machine. The rows of the matrix define the parallelism of the block (i.e., the number of evaluations per clock cycles), while the columns define the maximum correction capability of the block. Whenever the selected correction capability t is lower than t_M, the coefficients of the error locator polynomial of degree greater than t are equal to zero and do not contribute to equation (??), thus allowing us to adapt the computation to the different correction capabilities.

The third feature stems from a simple observation. Depending on the selected correction capability t, not all the elements of GF(2^m) represent realistic error locations. In fact, considering a codeword composed of k bits of the original message and $r = m \cdot t$ parity bits, only $k + m \cdot t$ out of 2^m elements of the Galois field represent realistic error locations. Given that an error location L is the inverse of the related GF element ($L = 2^m - 1 - i$), the elements of GF(2^m) in which the error locator polynomial must be evaluated are in the following range:

$$\left[\underbrace{\alpha^{2^m-1}}_{\text{error location L=0}}, \underbrace{\alpha^{2^m-k-m\cdot t}}_{\text{error location } L=k+m\cdot t-1}\right] \tag{9}$$

All elements between α^0 and $\alpha^{2^{m-k-m\cdot t}}$ can be skipped to reduce the computation time. Differently from fixed correction capability fast skipping Chien machines this interval is not constant here but depends on the selected t. The architecture of Fig. **??** implements an adaptable fast skipping by initializing the internal registers to the coefficients of the error corrector polynomial multiplied by a proper value $\beta_{ini}^t = \alpha^{2^m-k-m\cdot t-1}$. For each value of t, t_M m−bit constant values corresponding to β_{ini}^t, $(\beta_{ini}^t)^2$, ..., $(\beta_{ini}^t)^{t_M}$ must be stored in an internal ROM (not shown in Fig. **??**) and multiplied by the coefficients λ_i using a full GF multiplier.

This is connected with the last feature, the reduced GF Full Multipliers (redGFFM) network of Fig. **??**. Each full GF multiplier has a high cost in terms of area. Since they are used only during initialization of the Chien, the redGFFM adopts only $z \leqslant t_M$ full GF multipliers. It also includes a (λ) input port to input z coefficients, per clock cycles, of the error locator polynomial. This network enables to reduce area consumption, at a reasonable cost in terms of latency.

For the sake of brevity, a detailed description of the controller required to fully coordinate the decoder's modules interaction is out of the scope of this paper.

7. Experimental Results

This section provides experimental data from the implementation of the adaptable BCH codec proposed on a selected case study.

7.1. Automatic generation framework

To cope with the complexity of a manual design of these blocks, a semi-automatic generation tool named ADAGE (ADaptive ECC Automatic GEnerator) [?] able to generate a fully synthesizable adaptable BCH codec core following the proposed architecture has been designed and exploited in this experimentation extending a preliminary framework previously introduced in [?]. The overall architecture of the framework is in Fig. **??**.

The code analyzer block represents the first computational step required to select the desired code correction capability based on the Bit Error Rate (BER) of a page of the selected flash [?]. The BER is the fraction of erroneous bits of the flash. It is the key factor used to select the correction capability. Two values of BER must be considered. The former is the raw bit error rate (RBER), i.e., the BER before applying the error correction. It is technology/environment dependent and increases with the aging of the page [? ?]. The latter is the uncorrectable bit error rate (UBER), i.e.,

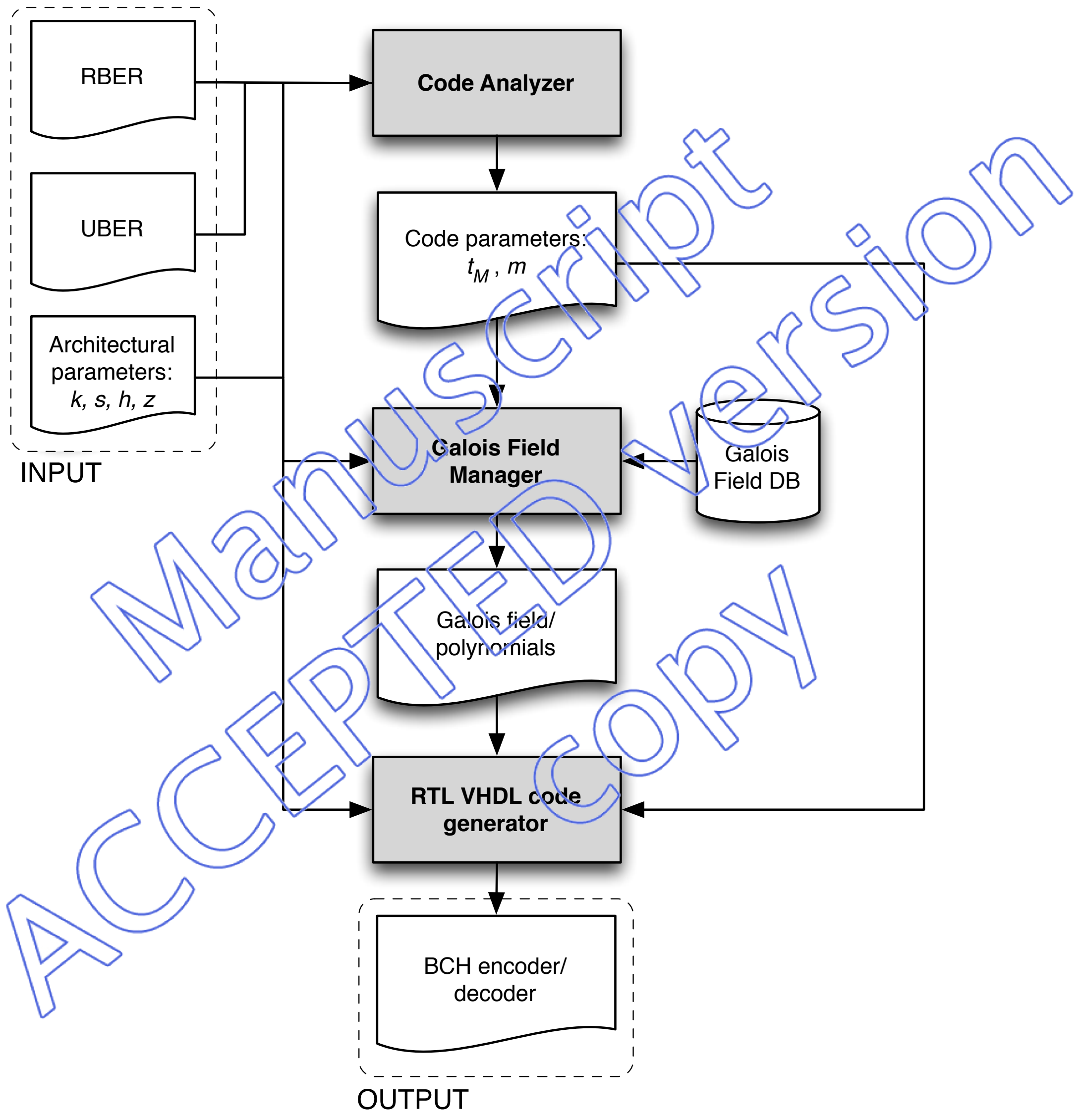


Figure 12: BCH codec automatic generation framework.

the BER after the application of the ECC, which is application dependent. It is computed as the probability of having more than t errors in the codeword (calculated as a binomial distribution of randomly occurred bit errors) divided by the length of the codeword [?]:

$$UBER = \frac{P(E > t)}{n} = \frac{1}{n} \sum_{i=t+1}^{n} \binom{n}{i} \cdot RBER^{i} \cdot (1 - RBER)^{n-i} \qquad (10)$$

Given the RBER of the flash and the target UBER, Eq. ?? can be exploited to compute the maximum required correction capability of the code and consequently the value of m that defines the target GF. Given these two parameters, the Galois Field manager exploits an internal polynomials database to generate the set of minimal polynomials and the related generator polynomials for the selected code.

Finally, the RTL VHDL code generator combines these parameters and generates a RTL description of the BCH encoder and decoder implementing the architecture illustrated in this paper.

The whole framework combines Matlab software modules with custom C programs. The full framework code is available for download at `http://www.testgroup.polito.it` in the Tools section of the website.

7.2. Experimental setup

Experiments have been performed, using as a case study a 2-bit per cell MLC NAND Flash Memory featuring a 45nm manufacturing process designed for low-power applications, with page size of 2KB plus 64B of spare cells. The memory has an 8-bit I/O interface. Considering the design of the BCH code, the current trend is to enlarge the block size k over which

ECC operations are performed. In fact, longer blocks better handle higher concentrations of errors, providing more protection while using fewer parity bits [?]. For this reason, we adopted a block size $k = 2$KB, equal to the page size of the selected memory.

Experiments performed on the flash provided that, in a range between 10 and 100,000 program/erase (P/E) cycles on a page, the estimated RBER changes in a range $[9 \times 10^{-6} \div 3.5 \times 10^{-4}]$ [?]. With a target UBER of 10^{-13}, which is typical for commercial applications [? ?], according to equation (**??**) we need to design a codec with correction capability in the range $t_{min} = 5$ up to $t_M = 24$. Since $k = 2^{14}$ and $t_M = 24$, from the expression $k + m \cdot t_M \leq 2^m - 1$ we deduce $m = 15$, thus obtaining a maximum of $r = m \cdot t_M \simeq 45$B of parity information. Given the 8-bit I/O interface of the memory, both the encoder and the decoder have been designed with an input parallelism of $s = 8$ bits. The values of h and z of the Chien Machine are a trade-off between the complexity of the decoder and the decoding time. Given the I/O parallelism of the flash and the area optimizations of Fig. **??**, we opted for a Chien machine with parallelism $h = 8$ and $z = 1$ full GF multipliers.

In this experimentation we analyzed the three architectures summarized in Table **??**.

Arch. 1 is classic BCH architecture with fixed correction capability of 24 errors per page. It represents the reference to compare our adaptable architectures.

Arch. 2 is an adaptable architecture with $t_{min} = 5 < t \leq 24$ using a traditional PPLFSR for the encoder and 24 PLFSRs for the syndrome calculation. It is worth mentioning here that, differently from what reported in the previous sections, the minimum required correction capability of the codec is higher than 1. This allows us to save space in the encoder PPLFSR since less polynomials must be stored, and in the Chien Machine's ROM since less β_{ini} terms must be stored.

Arch. 3 is an optimized version of Arch. 2 exploiting the use of a shOP-PLFSR shared between the encoder and the decoder, to trade-off design complexity and decoding time. In order to optimize the use of the shOP-PLFSR, we exploited the algorithm proposed in Section ??. Given our adaptable BCH code, a set of ad-hoc Matlab simulation scripts implement this preliminary analysis of 1,800[1] set Ω_i of polynomials. Each set Ω_i contains $t_M - t_{min} - 1 = 20$ generator polynomials required in the encoder and $t_M = 24$ minimal polynomials required in the decoder. This analysis aimed at finding the most suitable set of shared generator and minimal polynomials to trade-off between decoder's area and latency. A reasonable trade-off has been found using a shOPPLFSR composed of `N` $= 5$ OPPLFSRs, each of which dividing by the following set of polynomials: $\{g_5, \psi_{29}, \psi_{39}\}$, $\{g_6, \psi_{31}, \psi_{41}\}$, $\{g_7, \psi_{33}, \psi_{43}\}$, $\{g_8, \psi_{35}, \psi_{45}\}$, and $\{g_9, ..., g_{24}, \psi_{37}, \psi_{47}\}$. The reader may refer to the appendix of this paper for the full list of employed polynomials. All other structures remain almost unchanged. The comparison between Arch.1 and Arch. 2 enables to highlight the benefits of using an adaptable codec, while the comparison between Arch. 2 and Arch. 3 shows the advantages of adding optimized shared blocks.

[1]our BCH code has 1,800 primitive polynomials $\psi_1(x)$

Table 3: Characteristics of the analyzed architectures

	Adaptable	**OPPLFSRs**	**Chien Machine**
Arch. 1	No	-	$h = 8,\ t = 24$
Arch. 2	Yes	-	$h = 8,\ t \in [5, 24]$
Arch. 3	Yes	5	$h = 8,\ t \in [5, 24]$

7.3. Performance evaluations

Table **??** summarizes the main implementation details of the three selected architectures in terms of required parity bits and worst case encoding/decoding latency, expressed in terms of clock cycles.

Let us start with the evaluation of the amount of redundancy introduced by the two architectures. Arch. 1, which has a fixed correction capability of 24 errors per page, requires to store $m \cdot t_M = 24 \cdot 15 = 360$ parity bits (about 45B) for each 2KB page of the flash. This accounts for about 70% of the full spare area available for each page. Since the spare area cannot be fully reserved for storing ECC information (high-level functions, such as file system management and wear-leveling need to save considerable amount of information in this area), this percentage represents a considerable overhead for the selected device. Based on the results of Table **??**, Fig. **??** shows how, for the adaptable codecs of both Arch. 2 and Arch. 3, the percentage of spare area dedicated for storing parity bits changes with the selected correction capability. The total occupation ranges in this case from 15% to 70% of the total spare area. This mitigates the overhead for storing parity bits whenever the error rate enables to select low correction capabilities (e.g., for devices in

Table 4: Worst case Parity Bits and Encoding/Decoding Latency. sh_{poly} denotes the maximum number of minimal polynomials shared in the shOPPLFSR of the syndrome machine

	Correction Capability	Parity Bits	Encoding latency (#Clk cycles)	Decoding latency (#Clk cycles)		
				Syndrome	iBM	Chien
		$m \cdot t$	$\frac{k}{s}$	$sh_{poly} \cdot \frac{k+mt}{s}$	$2t^2$	$\frac{h}{z} + \frac{k+mt}{h}$
Arch. 1	$t = 24$	360	$2,048$	$2,093$	1152	$2,093$
Arch. 2	$t = \{5, 6, ..., 24\}$	$15 \cdot t$	$2,048$	$\frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$	$2t^2$	$\frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$
Arch. 3	$t = \{5, 6, ..., 24\}$	$15 \cdot t$	$2,048$	$\frac{2 \times (2{,}048 \cdot 8 + 15 \cdot t)}{8}$	$2t^2$	$8 + \frac{2{,}048 \cdot 8 + 15 \cdot t}{8}$

the early stages of their life). 670

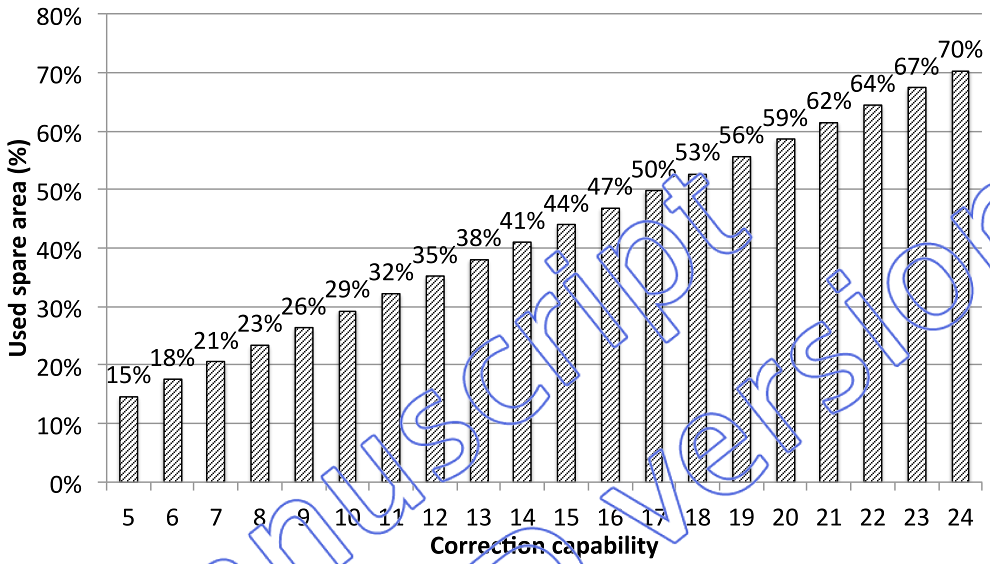


Figure 13: Percentage of spare area dedicated to parity bits while changing the correction capability of the adaptable codes of Arch. 2 and Arch. 3

For all implementations, the encoding latency depends on the size of the incoming message and is therefore constant regardless the adaptability of the encoder (see Table ??). The decoding latency is instead influenced by the correction capability, as reported in Table ??. Fig. ?? compares the decoding latency of the three architectures for each considered correction capability. Results are provided in number of clock cycles. It is worth mentioning here that timing estimations of Table ?? and Fig. ?? depict the worst-case scenario in which the Chien Machine must search all possible positions prior to find the detected number of errors. Fig. ?? highlights that, for the lowest correction capability, both Arch. 2 and Arch. 3 enable 22% of decoding time reduction when compared to the fixed decoding time of Arch. 1. The decod-

ing time increases with the correction capability. For Arch. 2, it reaches the 682
same level of the fixed architecture when the correction capability reaches 683
$t = 24$. Arch. 3 deviates from this behavior for $t \geqslant 20$. This penalty is intro- 684
duced by the use of the shOPPLFSR in the Syndrome Machine. In this case, 685
the codec includes 5 blocks to perform remainder computation with 10 min- 686
imal polynomials $\{\psi_{29}, \psi_{39}, \psi_{31}, \psi_{41}, \psi_{33}, \psi_{43}, \psi_{35}, \psi_{45}, \psi_{37}, \psi_{47}\}$. This implies 687
doubling the syndrome computation time every time the required correction 688
capability reaches a level in which all these polynomials must be used. Nev- 689
ertheless, we will show that this reduced performance is counterbalanced by 690
a reduced area overhead. 691

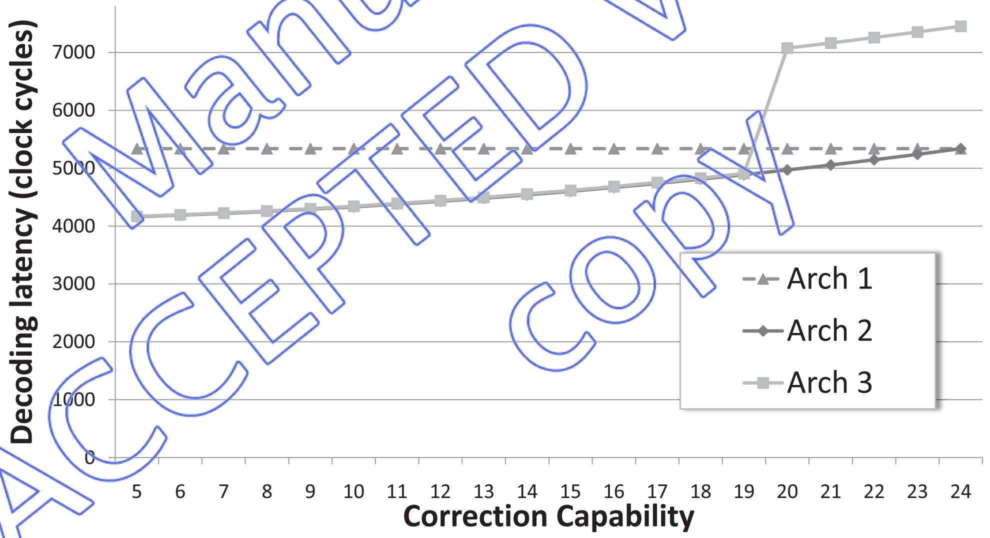


Figure 14: Worst case decoding latency for the three architectures considered.

7.4. Synthesis Results 692

Synopsys Design Vision and a CORE 45nm technology cell library have 693
been exploited to synthesize the designs. Table **??** shows the results of the 694

synthesis of the three architectures. The hardware structures required to obtain the adaptability of the code introduce a certain area overhead. Considering Arch. 2, the area of the encoder increases since 19 generator polynomials must be stored in its ROM, while the area of the decoder increases due both to the aligners in the syndrome machine and to the ROM in the Chien machine to adapt the fast skipping process. Nevertheless, the introduced overhead is about 14% which is still acceptable. Considering Arch. 3, the introduced overhead is halved w.r.t. Arch. 2. The area of the encoder is almost comparable with Arch. 2. However, it now includes the shOPPLFSR and a smaller ROMs which contribute, with the LFSR sharing, at decreasing the area of the decoder. For both architectures we obtained a maximum clock frequency of 100MHz, which confirms that the adaptability does not impact the maximum speed of the circuit. This area result is interesting if compared with an estimation of the area for the adaptable architecture proposed in [**?**]. [**?**] designed a codec working on blocks of data of 512B, smaller than the 2KB used in this paper. Given the same maximum correction capability ($t_M = 24$), [**?**] uses a code defined on $GF(2^{13})$ instead of the code defined on $GF(2^{15})$ used in this paper. However, even if the code is simpler and the number of correction modes is smaller (only 4 correction modes), the area of the codec accounts about 158.9K equivalent gates[2], which is higher than the 111.4K and the 105.2K equivalent gates of the Arch. 2 and Arch. 3 proposed.

Fig. **??** compares the decoder's dynamic power dissipation of the three architectures computed using Synopsys PrimeTime. As for the decoding

[2]Equivalent gates for state-of-the-art architectures have been estimated from the information provided in the papers

Table 5: Synthesis Results

	Comp.	Max Clock	Equiv. Gates	Over-head
Arch. 1	Encoder	100 MHz	33.3 K	
	Decoder	100 MHz	64.1 K	
	Overall	100 MHz	97.4 K	(ref.)
Arch. 2	Encoder	100 MHz	40.8 K	
	Decoder	100 MHz	70.6 K	
	Overall	100 MHz	111.4 K	14%
Arch. 3	Encoder	100 MHz	39.2 K	
	Decoder	100 MHz	66.0 K	
	Overall	100 MHz	105.2 K	7%

latency the analysis has been performed for a worst-case simulation in which t errors are injected at the end of the codeword so that the Chien Machine must search all possible positions prior to detect all errors. Considering Arch. 2, results show that the introduction of the adaptability enables up to 15% of dynamic power saving when the lowest correction capability can be selected. This is due to the fact that the portions of the circuits not required for low correction capabilities are disabled. The introduction of the optimizations proposed in Arch. 3 has no significant impact on the dynamic power that remains almost equal to the one of Arch. 2.

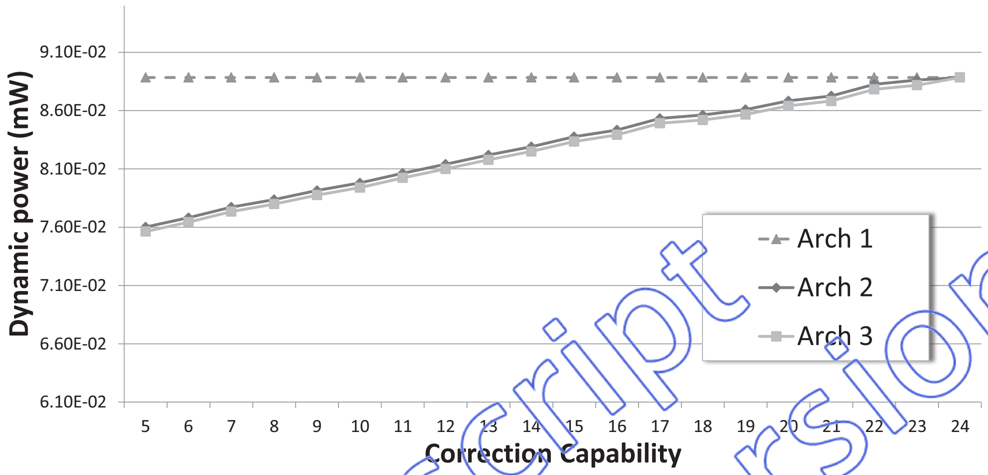


Figure 15: Worst case dynamic power consumption of the three decoders for the three considered architectures. Power is expressed in mW.

8. Conclusions

This paper proposed a BCH codec architectures and its related automatic generation framework which enables its code correction capability to be selected in a predefined range of values. Designing an ECC system whose correction capability can be modified in-field has the potentiality to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability.

Experimental results on a selected NAND flash memory architecture proved that the proposed solution reduces spare area usage, decoding time, and power dissipation whenever small correction capability can be selected.

Table 6: Minimal polynomials expressed with the corresponding hexadecimal string of coefficients

ψ_1	0x F465	ψ_{17}	0x B13D	ψ_{33}	0x 8011
ψ_3	0x C209	ψ_{19}	0x B305	ψ_{35}	0x BA2B
ψ_5	0x B3B7	ψ_{21}	0x A495	ψ_{37}	0x D95F
ψ_7	0x E6EB	ψ_{23}	0x 88C7	ψ_{39}	0x BFF5
ψ_9	0x E647	ψ_{25}	0x C357	ψ_{41}	0x BA87
ψ_{11}	0x D4E5	ψ_{27}	0x B2C1	ψ_{43}	0x 9BEB
ψ_{13}	0x 8371	ψ_{29}	0x 97DD	ψ_{45}	0x 93CB
ψ_{15}	0x EDD9	ψ_{31}	0x FA49	ψ_{47}	0x F385

Table 7: Generator polynomial expressed with the corresponding hexadecimal string of coefficients

g_5	0x 0163C68D766635253
g_6	0x 018FBE36E3B716D8BCE32
g_7	0x 01E573FBB06E46A828C1C770C
g_8	0x 01F28E94D9B550543AC42286CF418
g_9	0x 01D6634FC565E6012E441926C07B8D59
g_{10}	0x 018B24C1E935C04DC6BC73E0BDB98405C4EA
g_{11}	0x 01E8B4BA11F717E75A1F5E0EC4FBCD65DA8FFF24
g_{12}	0x 018FB50FA2969CDC5EAFA1C24BD9E5AA92A2227EC668
g_{13}	0x 012E919C715C15310DA7103C0AB656C7FE33061319763lD
g_{14}	0x 01E59154D4757E35CBDCE8247F4686EACC2C96C8209D848BDCE
g_{15}	0x 01E12C4539A437988318B8B0A756426E93CD5001031DCB5DC430A0C
g_{16}	0x 01BE62D0F7C4D16FCCDD3CE20D7998280B591702D452F3541A51DA955D8
g_{17}	0x 019755B57BEBA0DD4C284FE4B4F4549C194CA6E75E542322123EAB270447821712
g_{18}	0x 016240D5F338473A9653892D4DDC334AEF9FE78E9B835C10D1C9106B14AA4AB4BD5CD4
g_{19}	0x 01B54AFF801C5EBB55EA214ADCCB051347A164182682642642 99431B25E5B7CE34F402D938
g_{20}	0x 01CA788668B1303E48C4A41BE62900685C4A42DB04E267A642AC82884176194501F076D19CF53
g_{21}	0x 015E830624B4D708788177787CA2DC6C89F7558E799E84DD1027034F4DEC7476ADA565B11240FB4EE
g_{22}	0x 01D6ECB0041A40258ADA46542DB3657CFA042227D7CAADD770809ACC680C2886C0EACDC8D81D34565F7FC
g_{23}	0x 0102924C5CEA2B43968EFF54D1E0FAB54DEBFDC54428EDAE6FE2EE724B79CBC072C19CEB766864091E5551A38
g_{24}	0x 0141AE126215097403F13F41BE936020FAA0D6D486AD40BE0BED62DC87C4D8CF945A4D2A804411217E82829127AD