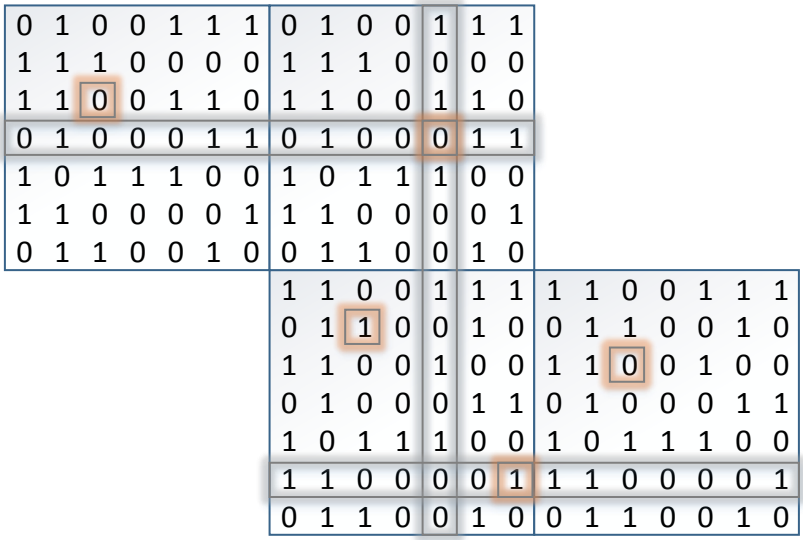

Hard-decision Staircase Decoder in 28-nm Fully-Depleted Silicon-on-Insulator

Master of Science Thesis in Embedded Electronic System Design

Elma Hurtic
Henri Lillmaa

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, June 2016

Hard-decision Staircase Decoder in 28-nm Fully-Depleted Silicon-on-Insulator
Elma Hurtic
Henri Lillmaa

Supervisor: Per Larsson-Edefors, Department of Computer Science and Engineering
Examiner: Sven Knutsson, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31 772 1000

Cover: Code-blocks arranged in staircase decoder, where errors are highlighted in red

Department of Computer Science and Engineering
Gothenburg, Sweden, June 2016

Abstract

The continuous advancements in optical communication channels have propelled the development of new error-correcting codes, e.g., staircase codes, which belong to a class of hard-decision algebraic codes. The staircase code is a new in-line error-correcting code that promises near-capacity performance. In this Master's Thesis BCH component codes and staircase codes are analysed in MATLAB. A bit-parallel BCH component code decoder is described in VHDL and synthesised in a 28-nm fully-depleted silicon-on-insulator (FD-SOI) library. Based on synthesis and simulation, the area and power consumption of staircase decoders for 100 Gbps throughput are estimated.

Keywords: staircase-decoder, 28nm-technology, error-correcting codes, forward error correction, BCH-decoder, MATLAB, VHDL.

Acknowledgements

Sincere gratitude to our supervisor Per Larsson-Edefors, and to our co-supervisors Kevin Cushon and Christoffer Fougstedt. First off, thanks to Per for providing the opportunity and encouraging us to take on the challenging project. The comprehensive support, guidance and encouragement throughout the project will be always remembered and much appreciated. The meetings were always motivational and optimistic towards the project progress.

Thanks to our families, close-ones and friends for support and for putting up with the long lasting anti-social behaviour along with witnessing engineering/academical research struggles.

Elma Hurtic, Henri Lillmaa; Gothenburg, June 2016

Contents

Acronyms

ARQ	Automatic Repeat Request
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BCH	Bose-Hocquenghem-Chaudhuri
BER	Bit-Error Rate
BM	Berlekamp-Massey
BMA	Berlekamp-Massey Algorithm
EEP	Error Evaluator Polynomial
ELP	Error-Locator Polynomial
FEC	Forward Error Correction
FIFO	First-In First-Out
FFM	Finite-Field Mulitplier
FPGA	Field-Programmable Gate Array
GF	Galois Field
LDPC	Low-Density Parity-Check
OTN	Optical Transport Network
PB	Polynomial Basis
RS	Reed-Solomon
riBM	Reformulated Inverse-free BM
RTL	Register Transfer Level
SiBM	Simplified Inverse-free BM
SNR	Signal-to-Noise Ratio
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very-large-scale integration

1
Introduction

In the digital world of today the importance of reliable data transmission has never been greater, received data is expected to be an exact copy of the transmitted data. Unfortunately this is not always the case since the received data is often to a lesser or greater extent erroneous. The cause of the errors is often the channel noise [1].

One way of ensuring that correct data is received is to resend the same data if the previous transmit resulted in an erroneous receive, this method is called *automatic repeat request (ARQ)*. However, the effectiveness of this method (when it comes to reliable data transmission) is at the expense of decreased channel throughput and increased transmission time [2]. Another commonly used method is *forward error-correction (FEC)*, that in addition to detecting the errors also tries to correct them. This is done by utilisation of *error-correcting codes* [1]. The basic idea behind forward error-correction is the addition of redundant symbols to the input data so that the errors, which are a product of a noisy channel, can be corrected at the receiving end. To achieve this FEC will employ an encoder and a decoder. The encoder will take the source data and add redundant data to it before sending it through the channel. At the receiving end the decoder will make use of the redundancy to correct the errors induced by the channel [3].

In a 1948 paper, *A Mathematical Theory of Communication*, American mathematician C. E. Shannon, demonstrated that it is possible to send data through a channel with a small probability of error if the data rate is smaller than or equal to the channel capacity [4]. In brief this means that in theory, data rate needs not to be sacrificed for the sake of the reliable data transmission. Shortly thereafter in 1950 Richard Hamming, also an American mathematician, introduced the Hamming code, the first error-correcting code, which has revolutionised the field of error-correcting codes in data communications [5, 3].

The two aforementioned historical events have prompted the search for the perfect error-correcting code deemed possible by Shannon's theorem. Since then many error-correcting codes have been developed, two of the most popular being the *Bose--Hocquenghem-Chaudhuri (BCH)* codes and *Reed-Solomon* codes [6]. However, the development of new and more advanced communication mediums like optical fibre has also increased the performance demands on the error-correcting codes, and meeting those continuously increasing demands in terms of low bit error rate and high throughput is becoming more and more challenging [7, 8].

One of the recent additions amongst error-correcting codes is the *low-density parity-check (LDPC)* code [9] that facilitates transmissions up to 10 Gbps. Fibre-optic communication systems offer transmission rates in the 100 Gbps range and require bit-error rates below 10^{-15}, 3 dB shy of the Shannon limit at the required

bit rate RS-LDPC (LDPC implemented with Reed-Solomon codes) is simply not good enough [8]. It was soon found that the performance of the LDPC code could be greatly improved if it was combined with algebraic component codes, which led to a birth of a new class of error-correcting codes, *the staircase codes* [8], which outperformed all other error-correcting codes recommended by the ITU-T G.975.1 (Telecommunication Standardisation Sector) [10]. Staircase codes are therefore relevant for further study and possible implementation in hardware in order to evaluate the power consumption and performance of the design. The only (known to us) hardware implementation (on an FPGA) of staircase codes resulted in a performance within 0.56 dB of the Shannon limit at 10^{-15} [8].

To the best of our knowledge research for staircase code ASIC implementation has not been published, nor have any numbers on power consumption been presented in the scientific literature or publications. The aim for this thesis work is therefore to develop and implement a hard-decision staircase decoder using a 28-nm process technology, for power and performance analysis. In the process of evaluating the synthesised decoder's power and performance characteristics, the coding performance and practically relevant code parameters will be evaluated in the MATLAB simulation environment beforehand. Due to limited time of a master thesis project, the place-and-route and manufacturing of the chip is out of the scope for this thesis work.

1.1 Methodology

The staircase decoder is expected to be used in high-speed fibre-optic communication systems. It must therefore satisfy the performance requirements stated by the Optical Transport Network (OTN) standard (ITU-T G.709), from which it can be deduced that a successfully implemented decoder must have a bit error rate (BER) below 10^{-15} at a bit rate above 10 Gbps [8, 11].

To get a better understanding of the staircase decoder as well as to ensure that it meets the throughput and coding gain performance requirements of the OTN standard [8, 11], the thesis work starts with an exploratory phase. In this phase, see Chapter 2, we study theory and coding concepts, e.g., linear and cyclic codes and decoders, in relevant literature, mainly textbooks and articles.

In Section 3.1, we carry out modelling and simulation of very basic error-correcting codes in MATLAB, which helps us obtain a general understanding of how different error-correcting codes perform. This is done in parallel with the literature studies.

In Section 3.3 the next step, which is to build (in MATLAB) the staircase code itself and compare the performance of the staircase code to that of the product code, is described. For the product code, the coding is performed both column- and row-wise. Looking at the overall performance of the staircase code we then can verify that staircase codes perform better compared to product codes. In the end-phase of the staircase code implementation and study in MATLAB, we start manual BCH block code implementation and study, where the aim is to verify some of the recently published algorithms. The detailed MATLAB implementation in Section 3.2 also helps to guide the concurrent VHDL design in Section 4.1.1.

We prove the concept of the staircase codes in MATLAB, and start with the

hardware implementation phase in Chapter 4. The hardware implementation phase is started with the implementation of the BCH decoder, after which we implement the staircase architecture. The successfully implemented staircase decoder is then synthesised and analysed in a 28-nm fully-depleted silicon-on-insulator (FD-SOI) process technology.

2

Theory

This chapter covers the theory and terminology of coding theory and concepts. We start off by introducing the error-control principle and *forward error-correction* (FEC) technique, after which some preliminary theory on finite fields is provided.

The theory behind BCH block code is briefly described, along with the components upon which the BCH block code is built on.

2.1 Introduction to Error-Correction Codes

The digital data bandwidth used in communication or audio-video systems is rapidly increasing [12]. The advantage of digital signals over analog signals is that they are more reliable in a noisy environment, since for each symbol the detector in a digital system only needs to detect either a '1' or a '0' [2].

If the data are carried over a noisy environment and the strength of the noise is enough to change value of the original data symbols, then the detector might make an erroneous decision. However, if the data is *coded*, by adding appropriate check (redundant) symbols to the data symbols, the receiver (including detector and decoder) can detect, and possibly correct certain errors, making the signal reception more reliable. Adding redundant check symbols to data symbols is called *error-control coding* [2]. It is often referred to as a FEC technique, which is widely used in digital and real-time communication systems. FEC implies that the error-correcting code is applied to the data prior to transmission, and bit errors occurring during transmission are corrected at the receiving end of the system. One of the advantages of FEC, when compared to other error-control techniques using retransmission (e.g., *automatic repeat request* – ARQ), is that the channel-utilisation efficiency for the FEC is constant and depends only on the *code-rate* of the FEC code that is used. One of the disadvantages however is that if the number of bit errors is higher than the error-correcting capability of the FEC decoder, then the uncorrected bit errors will remain in the decoded data [2].

Based on the error-correcting capability of the code, the FEC code will correct a specified number of bit errors caused by the transmission channel by utilising the redundant bits added to the data by the encoder. By applying FEC before transmitting the data over a noisy channel, the bit-error probability of the transmitted data is reduced, thus increasing channel capacity and reducing both cost and required transmission power. For example, due to gained power and cost savings FECs are widely used in satellite-communication systems [2].

One important characteristic of the FEC code, which expresses the transmission

power advantage of FEC, is the *coding gain*. Coding gain is defined as reduction of the ratio of *energy per information bit* (Eb) to *noise power density* (N_0)— Eb/N_0 compared to the uncoded system. The coding gain at a specific *bit-error rate* (BER) is calculated as the difference of required Eb/N_0 between the uncoded and coded systems, where BER is the ratio of occurred bit errors over transmitted bits during the studied period of time [2].

The type of errors that occur in the system (e.g., digital communication or recording system) is one of the characteristics that needs to be considered when choosing an appropriate error-control or FEC code. Upon choosing an appropriate FEC code, the occurrence of different *types* of errors should be considered, i.e., whether they are random, burst or byte errors. Random errors take place independently for each information symbol. Burst errors occur over a contiguous sequence of information symbols. Byte errors occur in number of bits confined to one byte. In some scenarios a combination of two of the three types may occur, where byte error can be considered as the more restricted case of a burst error. For each error type there exist codes for effectively detecting and/or correcting that particular type of error, however a code which is designed to detect and/or correct one type of error may not be effective for other types [2].

Other characteristics for consideration when choosing a FEC code are the *error-correction capability*, the *number of redundant symbols*, *code-rate* and the *size*, *complexity* and *speed of the decoder*. When implementing a decoder, the three last properties are part of the design trade-offs that have to be made. An example of a trade-off is choosing between two codes with the same error-correction capability, where one (e.g., smaller) code has lower code-rate while another (e.g., larger) code has higher code-rate. This trade-off in the example of two arbitrary code classes is illustrated in Fig. 2.1. For example, compared to a class A code, class B code has higher code-rate, yielding higher efficiency. For the codes with the same error-correction capability, the higher the code-rate, the higher the efficiency, as there is less space for redundancy (symbols) needed. However, codes that are efficient in terms of code-rate and coding gain typically need a complicated decoder, which operates at a relatively low speed. Therefore, when deciding if a simple or a fast decoder is required, code-rate and coding gain must be traded off to a certain extent. The problem of selecting a code can be solved by finding an acceptable trade-off between requirements for error-correction capability, code-rate, hardware implementation size, and speed of the decoder [2].

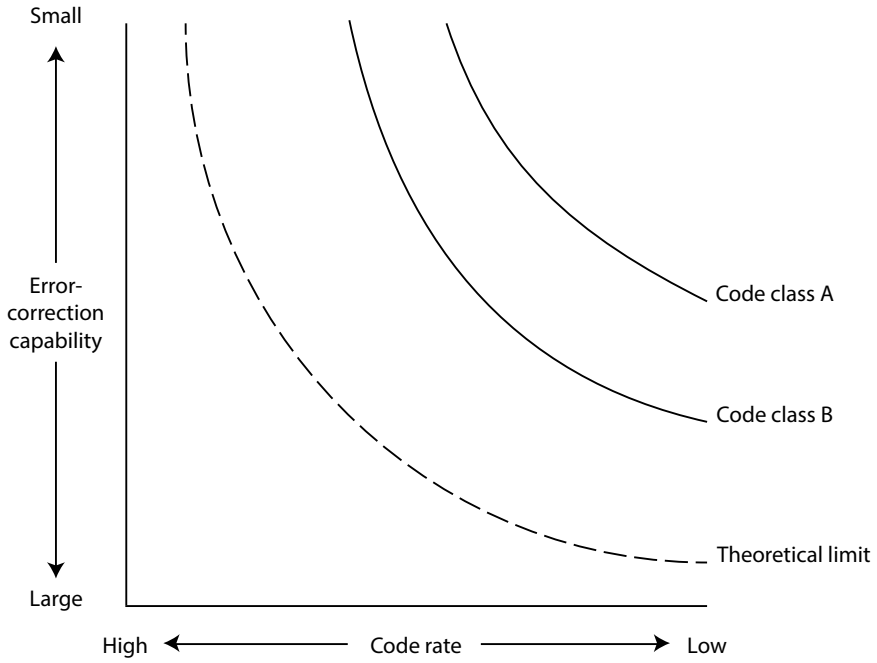

Figure 2.1: Error-correction capability and code-rate trade-off

2.1.1 Model of a Digital Communications System

Error-control coding techniques are applied on digital communication channels for detecting and/or correcting occurred errors. In Fig. 2.2, the transmitted message is a sequence of binary symbols of length k and denoted by $u = (u_1, u_2, ..., u_k)$, where for any element i the values in vector u are u_i ='0' or '1'.

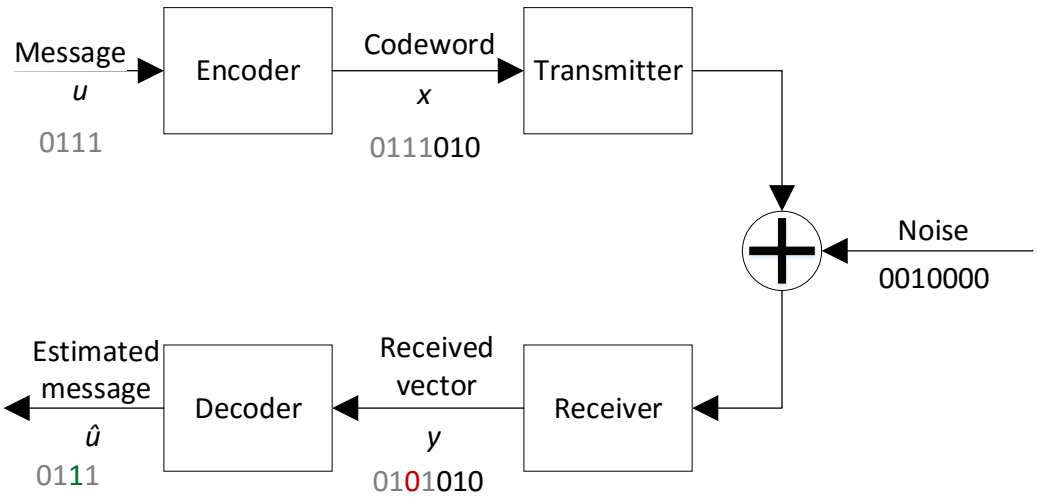

Figure 2.2: Model of digital communication system

The useful information *message* is input to an *encoder*, where the codeword is generated. The *codeword* is a binary sequence of length n, which is denoted by

$x = (x_1, x_2, ..., x_n)$, where $x_i = 0$ or 1 and $n > k$. The mapping of the encoder enables detection and/or correction of errors at the receiving-end. As a result the number of overall symbols is increased from k to n, where the ratio of k/n is called *code-rate* and denoted by R.

There are two types of encoding methods: the first one, where each codeword is generated from one block of k message symbols, is called block encoding. The block encoder does not hold any information about a message block after an encoding operation has been completed. The second type of encoding, where each codeword is generated from several consecutive message blocks, is called convolutional encoding [2].

Assuming block encoding is applied, let the set of all possible codewords generated by the encoder be $C = x_1, x_2, ..., x_M$, where $M = 2^k$ is the number of possible messages. The set C is called a *code* [2].

The codeword from the encoder is transmitted on the channel and the receiving-end of the channel obtains the binary message sequence of length $n, y = (y_1, y_2, ..., y_n)$, called the *received vector*. From the received vector y, the *decoder* estimates the transmitted codeword x and delivers the estimated codeword $\hat{x}$, or equivalently the estimated message $\hat{u}$ to the destination. In some cases, the decoder may only evaluate whether or not an error occurred on the channel (e.g., cyclic redundancy check - CRC) [2]. Decoding methods can be divided into two classes: *hard-decision* (HD) and *soft-decision decoding* (SD). Hard-decision decoding is where the receiver decides whether each transmitted symbol was a '0' or a '1' in the received binary sequence vector y. Soft-decision decoding on the other hand generates some multi-valued information on each received symbol and the decoder recovers the message from the multi-valued or (more than 2 level) quantised received vector y. The SD decoding is not only using binary decision of value being '1' or '0', but also the reliability of that decision. SD decoding method usually provides a better coding gain than the HD method, but it is not easy to implement in the optical communication technology due to high complexity of the decoding algorithm and high computational expense of obtaining the reliability information about the decision [13].

In the noiseless channel, the received vector y is equal to the transmitted codeword x. In the noisy channel however, y may differ from x, so the received vector can be expressed as

$$y = x + e, \tag{2.1}$$

where $e = (e_1, e_2, ..., e_n)$ represents the effect of the channel noise. Vector e is called the *error vector*. Similarly for the received vector y, in the case of soft-decision decoding, e is an analog or multiple-valued vector and in the case of hard-decision decoding, e is a binary vector, where each bit in vector e represents whether or not an error occurred on the transmitted bit. In hard-decision decoding, the addition in Eq.2.1 is performed by modulo 2 addition or effectively with an exclusive OR gate [2]. Although, SD decoding can offer higher coding gain, the more high-speed operation oriented and less computations demanding [14] HD decoding method will be used in this thesis project and considered to be used from now on.

2.2 BCH Codes

Bose-Hocquenghem-Chaudhuri (BCH) codes were developed twice, first by Hocquenghem in 1959 and then by Bose and Chaudhuri in 1960, and have since become the most influential amongst the error-correcting codes [6, 2]. The efficiency of the binary BCH codes, especially in correcting random errors, makes them suitable in the implementation of staircase codes [15].

BCH codes are cyclic codes (see Def. 2.2.1), that can be both binary and non-binary [3]. Binary BCH codes, which due to reduced hardware complexity are of interest to us, are codes with elements in GF(2^m) (see Sec. 2.2.1), where m is a positive integer [6, 16].

Definition 2.2.1. *If a codeword is an element of C, then every cyclic shift of the codeword will produce another codeword that is also an element of C [6].*

BCH codes as well as other error-correcting codes are characterised by their code parameters. The parameters n and k signify the length of the message and the codeword respectively, the (n,k)-values are often prefixed in the code name, e.g., (n,k) BCH code. d is the distance between two codewords, i.e, the number of positions where two codewords belonging to the same code differ. The parameter t defines the error-correcting capability of the code [17]. For the BCH codes following is true [6]:

$$n = 2^m - 1$$

$$n - k \leq mt$$

$$d_{min} \geq 2t + 1$$

Code parameters can give us a first glance at the efficiency of the error-correcting code. For example assume a (15,7) BCH code, from the n,k values we can conclude $(15 - 7 \leq 4t)$ that $t = 2$, i.e., the (15,7) code can correct up to two erroneous bits.

To ease the understanding of the mathematical concepts used in the BCH codes the following section gives a quick introduction to algebra.

2.2.1 Galois Field (GF)

Use of algebra is quite common in the field of error-correcting codes, reason for that is the usefulness of the algebraic alphabet when it comes to doing the basic mathematical operations like addition, subtraction, multiplication and division on error-correcting codes [16]. Because of this an introduction is given on basic algebraic concepts like fields, Galois field in particular.

If elements of a set can be added, subtracted, multiplied and divided with each other and if the resulting elements of these operations can be found in the same set, all of these elements are then part of a *field*, assuming that both addition and multiplication have commutative, associative, and distributive properties.

A Galois field, GF(q), is like any other algebraic field with the exception that q must be a prime number or a power of prime number [6]. The previously mentioned binary Galois field, GF(2^m), consists of binary elements of size m.

The elements of the Galois field, during the decoding, will be operated on using binary addition and multiplication. Addition and multiplication in binary Galois field are modulo-2 operations, where modulo-2 addition corresponds to XOR-operation and modulo-2 multiplication to an AND-operation [1]. Examples of multiplication and addition are brought in Table 2.1.

Table 2.1: Examples of addition and multiplication tables

$\oplus$	**0**	**1**
0	0	1
1	1	0

$\otimes$	**0**	**1**
0	0	0
1	0	1

From now on we are going to use the greek letter α (*alpha*) to denote the elements of the Galois field. Elements of the Galois field are all powers of α (i.e., $\alpha^0, \alpha^1, \alpha^2, ...\alpha^{q-2}$) and are constructed using the primitive polynomial p(x), which is the first irreducible polynomial for Galois field of order q that has a root in α [6].

GF table contains all elements of the GF(2^m). The first element of the GF table is $\alpha^0(1)$ and is represented by binary '1'. Bit '1' is then shifted one bit to the right (assuming that the leftmost bit is the LSB) until MSB is '1'. The $(m+1)_{th}$ element of the GF table (or α^m) is deduced from the primitive polynomial by utilising the statement that the primitive polynomial has a root in α, (i.e., $p(\alpha) = 0$). The element after, which is α^{m+1} is naturally deduced by multiplying previous element with α, i.e., $\alpha^{m+1} = \alpha(\alpha^m)$, and so forth.

To get a better understanding of the construction of GF tables, let us do just that and construct a GF(2^4) table as seen in Table 2.2. The primitive polynomial for $m = 4$ is $p(\alpha) = 1 + \alpha + \alpha^4$. The first element of the table is naturally "1000", shifting one bit to the right will give us the second element which is "0100", the third element is "0010", and so forth until we reach the 5^{th} element which is α^4. To deduce α^4 a primitive polynomial is used:

$$p(\alpha) = 1 + \alpha + \alpha^4 = 0 \Rightarrow \alpha^4 = 1 + \alpha$$

By multiplying α^4 by α we will get α^5:

$$\alpha^5 = (\alpha)(\alpha^4) = (\alpha)(1 + \alpha) = \alpha + \alpha^2$$

Table 2.2: Example of constructing a Galois field – GF(2^4) table

	Vector	Polynomial
	1000	$\alpha^0 = 1$
XOR	0100	$\alpha^1 = \alpha$
	1100	$\alpha^4 = 1 + \alpha$

	Vector	Polynomial
	0100	$\alpha^1 = \alpha$
XOR	0010	α^2
	0110	$\alpha^5 = \alpha + \alpha^2$

Multiplication by α is continued until we reach the last element of the GF table:

$$\alpha^6 = (\alpha)(\alpha^5) = (\alpha)(\alpha + \alpha^2) = \alpha^2 + \alpha^3$$

$$\alpha^7 = (\alpha)(\alpha^6) = (\alpha)(\alpha^2 + \alpha^3) = \alpha^3 + \alpha^4 = 1 + \alpha + \alpha^3$$

$$\vdots$$

A Galois field table for GF(2^4) is shown in Table 2.3.

Table 2.3: Galois field table for GF(2^4) constructed with $p(x) = 1 + x + x^4$ (1100)

Polynomial representation	Vector representation (LSB → MSB)
$\alpha^0 = 1$	1000
$\alpha^1 = \alpha$	0100
α^2	0010
α^3	0001
$\alpha^4 = 1 + \alpha$	1100
$\alpha^5 = \alpha + \alpha^2$	0110
$\alpha^6 = \alpha^2 + \alpha^3$	0011
$\alpha^7 = 1 + \alpha + \alpha^3$	1101
$\alpha^8 = 1 + \alpha^2$	1010
$\alpha^9 = \alpha + \alpha^3$	0101
$\alpha^{10} = 1 + \alpha + \alpha^2$	1110
$\alpha^{11} = \alpha + \alpha^2 + \alpha^3$	0111
$\alpha^{12} = 1 + \alpha + \alpha^2 + \alpha^3$	1111
$\alpha^{13} = 1 + \alpha^2 + \alpha^3$	1011
$\alpha^{14} = 1 + \alpha^3$	1001

2.2.2 Finite-Field Multiplication in Polynomial Basis

Finite-field multiplication is of great relevance in this project, we will therefore begin this section with a short elaboration on why this is the case.

Due to the multiplication between two primitive elements $\alpha^i \cdot \alpha^j = \alpha i + j$, where modulo $2^m - 1$ is performed on the resulting power, the extra overhead from conversion of representation (e.g. lookup tables) would increase the overall system complexity. Using polynomial basis (PB) multiplication enables more coherent representation of finite field elements through out the design.

First let us define a monic irreducible polynomial $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$ of degree m, where $p_i \in \mathrm{GF}(2^m)$ for $i = 0, 1, \ldots, m-1$. Also let $\alpha \in \mathrm{GF}(2^m)$ be root of $P(x)$, where for example $P(\alpha) = 0$. The acquired set $1, \alpha, \alpha^2, \ldots, \alpha^{m-1}$ is referred to as the polynomial or standard basis, where each element of GF(2^m) can be expressed with respect to the polynomial basis (PB). Let there be an element A in GF(2^m) for which the PB representation is $A = \sum_{i=0}^{m-1} a_i \alpha^i$, where $a_i \in 0, 1$ and $a_i s$ are the coordinates. The α coordinates are denoted in vector notation (in small, bold font) as $\mathsf{a} = [a_0, a_1, a_2, \ldots, a_{m-1}]^T$, where T denotes transposition. This GF(2^m) element A in vector notation can be written as $A = \alpha^T \mathbf{a}$, where $\alpha = [1, \alpha, \alpha^2, \ldots, \alpha^{m-1}]^T$. Let a binary polynomial S of degree $\leq 2m - 2$ that is the result of direct multiplication of the PB representation of two GF(2^m) elements A and B, as shown in Eqs. 2.2 and 2.3 [18, 19].

$$S = (\sum_{i=0}^{m-1} a_i \alpha^i)(\sum_{j=0}^{m-1} b_j \alpha^j) = \sum_{k=0}^{2m-2} s_k \alpha^k, \tag{2.2}$$

$$s_k = (\sum_{i+j=k} a_i b_j), 0 \leq i, j \leq m-1, 0 \leq k \leq 2m-2 \tag{2.3}$$

The resulting product $C = A \cdot B$ can be expressed as a modulo reduction:

$$C \triangleq \sum_{i=0}^{m-1} c_i \alpha^i \equiv S \mod P(\alpha) \equiv AB \mod P \tag{2.4}$$

In a previous study by Mastrovito [19], it is shown that using Eqs. 2.2 and 2.4 the product coordinates $c_i s$, which are calculated in terms of $a_i s, b_i s$, and the irreducible polynomial $P(x)$; these coordinates can be calculated using the following matrix equation:

$$\mathsf{c} = \mathsf{F}\mathsf{b} \tag{2.5}$$

In Eq. 2.5 vectors $\mathbf{b} = [b_0, b_1, b_2, \ldots, b_{m-1}]^T$ and $\mathbf{c} = [c_0, c_1, c_2, \ldots, b_{m-1}]^T$ are associated with B and C respectively [18]. In Eq. 2.5 vectors $\mathbf{b} = [b_0, b_1, b_2, \ldots, b_{m-1}]^T$ and $\mathbf{c} = [c_0, c_1, c_2, \ldots, b_{m-1}]^T$ are associated with B and C respectively. The full definition of the *product matrix* $\mathbf{F} = [f_{i,j}]_{i,j=0}^{m-1}$ can be found in [20].

Previously the three coefficient vectors $\mathsf{a}, \mathsf{b}, \mathsf{c}$, of the degree m elements in GF(2^m) were defined. Next, let us define three polynomials as shown in Eq. 2.6 and in Eq 2.7 coefficient vectors to polynomials are shown.

$$\begin{aligned} d(x) = a(x)b(x) = d_0 + d_1 x + ... + d_{2m-2} x^{2m-2}, \\ d(L) = d_0 + d_1 x + ... + d_{m-1} x^{m-1}, \\ d(H) = d_m + d_{m+1} x + ... + d_{2m-2} x^{2m-2} \end{aligned} \tag{2.6}$$

$$\begin{aligned} \mathbf{d} = [d_0 + d_1 + ... + d_{2m-2}]^T, \\ \mathbf{d^{(L)}} = [d_0 + d_1 + ... + d_{m-1}]^T, \\ \mathbf{d^{(H)}} = [d_m + d_{m+1} + ... + d_{2m-2}]^T \end{aligned} \tag{2.7}$$

The previous work carried out by Barreto et al. in [21] shows the reduction of the polynomial multiplication *d(x)* using an $(m \times m-1)$ *reduction matrix* Q and how to obtain the field product *c(x)*, also shown in Eq. 2.8 [22, 18, 20].

$$\mathsf{c} = \mathsf{d}^{(L)} + \mathsf{Q} \cdot \mathsf{d}^{(H)} \tag{2.8}$$

2.2.3 BCH Decoder

To decode BCH codes a BCH decoder is needed, see Fig 2.3. Input to the BCH decoder is the message received from the channel, *r(x)*.

First step in the decoding process is the computation of the syndromes. Syndromes serve only as error indicators, i.e., they will notify us of error occurrence but not of position of the error(s).

To be able to correct the erroneous bits, we need to know the position of those. By utilising the computed syndrome, an error-location polynomial, *λ(x)*, can be calculated using the Berlekamp Massey algorithm (BMA) [3].

Error positions are roots of the error-location polynomial and are found with Chien search, see Sec. 2.2.3.3. Output of the Chien search is an error position vector, *e(x)*, that indicates position(s) of erroneous bit(s) with a one [3, 1].

Figure 2.3: Overview of the decoding steps of a BCH decoder

2.2.3.1 Syndrome Calculation

As previously mentioned, syndromes are not only used as error indicators but also as input data to the BMA. Syndromes are calculated according to Eq. 2.9, where r is the received message and H^T is the transposed parity check matrix [15].

$$S = rH^T \tag{2.9}$$

The parity check matrix for the multiple-error-correcting BCH decoder, with the error-correcting capability of t, is defined as [2]:

$$H = \begin{bmatrix} \alpha^{n-1} & \alpha^{n-2} & \dots & \alpha^2 & \alpha & 1 \\ \alpha^{2(n-1)} & \alpha^{2(n-2)} & \dots & \alpha^4 & \alpha^2 & 1 \\ \alpha^{3(n-1)} & \alpha^{3(n-2)} & \dots & \alpha^6 & \alpha^3 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \alpha^{2t(n-1)} & \alpha^{2t(n-2)} & \dots & \alpha^{4t} & \alpha^{2t} & 1 \end{bmatrix}$$

In total *2t* number of syndrome elements are calculated and sent to the BMA. Following is the calculation for the i_{th} syndrome element [15]:

$$S_i = r(\alpha^i) = \sum_{j=0}^{n-1} r_j \alpha^{ji}$$

$$S_i = \begin{bmatrix} r_0 r_1 r_2 \dots r_{n-1} \end{bmatrix} \begin{bmatrix} \alpha^{0i} \\ \alpha^{1i} \\ \alpha^{2i} \\ \vdots \\ \alpha^{(n-1)i} \end{bmatrix}$$

It should be noted that the elements of syndrome $S = (S_1, S_2, \dots, S_{2t})$, are all elements of Galois field.

2.2.3.2 Berlekamp Massey Algorithm, BMA

The BMA [23, 24] is used for solving Toeplitz systems of linear equations of form, e.g., $Ax = b$, where A is diagonal-constant matrix/Toeplitz matrix. BMA is used for solving *error-locator polynomial (ELP)* $\Lambda(x)$ and *error evaluator polynomial (EEP)* $\Omega(x)$, which are also referred to *key equation* components. For example in the case of the Reed-Solomon (RS) codes, knowing that there is an error in a m-bit codeword, is not sufficient, while the location and the size of the error(s) are unknown. In the case of BCH codes, the size of the error is always known and of size binary 1 and therefore only the error-location is needed for error-correction.

Error-locator polynomial $\Lambda(x)$ of degree dependant on number of errors e and error-locator polynomial $\Omega(x)$ of degree at most $e-1$ can be defined as [25]

$$\Lambda(x) = \prod_{j=1}^{e}(1 - X_j x) = 1 + \lambda_1 x + \lambda_2 x^2 + ... + \lambda_e x^e$$

and

$$\Omega(x) = \sum_{i=1}^{e} Y_i X_i^{m_0} \prod_{j=1, j\neq i}^{e} (1 - X_j x).$$

By convention [25], the *error values* $Y_1, Y_2, ..., Y_e$ being either '1' or '0', have corresponding *error-locations* of $X_1 = a^{i_1}, X_2 = a^{i_2}, ..., X_e = a^{i_e}$, where the error have occurred. Both $\Lambda(x)$ and $\Omega(x)$ are related to syndrome polynomial $S(x)$ through the key equation

$$\Lambda(x)S(x) \equiv \Omega(x) mod\ x^{2t}.$$

When solving the key equation both Euclidean and BM algorithms can be used. If the number of errors is less than the error-correction capability of the code ($e \leq t$), then aforementioned two algorithms find $\Lambda(x)$ and $\Omega(x)$. However, if $e > t$ then the algorithms almost always fail to find the $\Lambda(x)$, $\Omega(x)$ polynomials.

For this thesis, BMA was considered, due to possible hardware efficient implementations [26, 27]. BMA iteratively solves key equation

$$\Lambda(r, x)S(x) \equiv \Omega(r, x) mod\ x^r,$$

where $r = 1, 2, ..., 2t$. It can be seen that for t error-correction capability, the code has *2t* syndromes and it takes *2t* clock cycles to find both the ELP and the EEP [25].

BMA actually solves a more general problem, of finding the least common multiple of a given sequence. For example, given a sequence $E - E_0, E_1, E_2, ..., E_{N-1}$ of length N, the BMA finds the recursion

$$E_i = -\sum_{j=1}^{L} \Lambda_j E_{i-j}, \text{where } i = L, ..., N-1,$$

for the smallest value of L. The r-th iteration gives the shortest recursion of $(\Lambda^r(x), L_r)$ for producing the first r terms of the E sequence, where $L_r = deg(\Lambda^r(X))$. The produced shortest recursion can be written as

$$E_i = -\sum_{j=1}^{L_r} \Lambda_j^r E_{i-j}, \text{where } i = L_r, ..., r-1$$

having $\Lambda_0^r = 0$ [28].

2.2.3.3 Chien Search

By locating the roots of the Λ polynomial that we get from BMA, we can find position(s) of the erroneous bits. This is done by employment of the Chien search algorithm, which is an iterative process in which x in $\Lambda(x)$ is iteratively replaced by elements of the finite field, $1, \alpha, \alpha^2, ..., \alpha^{n-1}$. α^i $(0 \leq i \leq n-1)$ is a root of the Λ polynomial if $\Lambda(\alpha^i) = 0$ [6, 1].

For example, let us assume that $\Lambda(x) = 1 + \alpha^{13}x + x^2$ is the received error-location polynomial for $(15, 7)$ BCH code. Chien search algorithm would start off by substituting x with the first element of the Galois field, i.e., $\alpha^0 = 1$.

$$\Lambda(1) = 1 + \alpha^{13}1 + 1^2 = 1 + \alpha^{13} + 1 = \alpha^{13}$$

Since $\Lambda(1) \neq 0$, we can conclude that 1 is not a root of the given error-location polynomial. It should be noted that two instances of the same element, in this case 1, will cancel each other out. This is because addition in Galois field, as mentioned in Section 2.2.1, is an XOR function. Next element to substitute x is $\alpha^1 = \alpha$, but since we know that roots of this particular error-location polynomial are at α^4 and α^{11}, let us speed up the search process and prove that substituting x with either α^4 or α^{11} will result in $\Lambda = 0$.

$$\begin{aligned}
\Lambda(\alpha^4) &= 1 + \alpha^{13}\alpha^4 + (\alpha^4)^2 \\
&= 1 + \alpha^{13+4} + \alpha^{4\times 2} \\
&= 1 + \alpha^{17} + \alpha^8 \\
&= 1 + \alpha^2 + 1 + \alpha^2 \\
&= 0
\end{aligned}$$

Chien search is a simple but effective algorithm, however it is quite considerable in size when implemented in hardware [1, 15], more about that in Chapter 4.

2.3 Product Code

In this section one of the techniques upon which the staircase code concept is based on – the combination of several existing codes – is discussed. The product code is a long code built from short component codes. Compared to the long length block codes, product code is a better performing code due to the cross parity check [6] and low circuitry overhead, since for the similar coding gain the component codewords for product code can be of lower error-correction capability [27].

Assume that both C_1 and C_2 are (n_1, k_1, d_1)-, (n_2, k_2, d_2)-systematic codes. To construct a product code C from C_1 and C_2 (i.e., $C = C_1 \times C_2$) with parameters $(n_1 n_2, k_1 k_2, d_1 d_2)$, the following steps need to be taken:

1. Information symbols (messages) – $k_1 k_2$ need to be arranged in a $(k_1 \times k_2)$ matrix A.
2. A $(n_1 \times n_2)$ matrix B needs to be constructed, so that the upper-left corner of matrix B is matrix A, see Fig. 2.4.

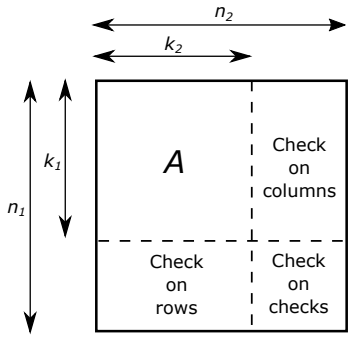

Figure 2.4: Two-dimensional $[n_1 n_2, k_1 k_2]$ product code

From Fig. 2.4 it can be concluded that the first row of A is message k_2 of C_2, while the first column of A is message k_1 of C_1. Resulting minimum distance d of product code C satisfies $d = d_1 d_2$[3].

2.4 Binary Convolutional Codes

If block coding is a memory-less operation, meaning that codewords are independent from each other, then convolutional codes have memory in a sense that not only do the convolutional codes depend on the current input information, but also on previous inputs or outputs. The binary convolutional code processes binary sequences or information serially — in bit-by-bit or continuously — block-by-block manner. Due to the information dependency between consecutive bits or blocks, the convolutional encoder/decoder is a sequential circuit or a finite-state machine, where the state of the encoder/decoder is defined as the contents of the memory. In finite-state-machine, the decoding can be described in a transition table, indicating the relation between the input and the current output, based on the previous and current state [3].

2.5 Staircase Codes

The high-rate binary staircase code is as mentioned earlier a new class of FEC which combines ideas from convolution and block coding [8]. Due to limited amount of computations that can be spent on the error-correction decoding, the hard bit flipping (hard-decision) decoding brings advantages (e.g., reduced optimisation complexity, efficient hardware implementation) in high-speed operation of optical transceivers [14, 7]. As a form of spatially coupled codes on graphs, the staircase codes, designed for optical communication, were recently also proposed to the optical communication standard ITU G.975.1 [14].

Staircase code is a new class of FEC codes, that can be considered as a hybrid of both convolutional and block codes. From the block codes, the idea of combining existing codes in a product-like manner is merged with the idea of block-to-block dependency from the convolutional codes that one block depends on the decoding of another block. In other words, the staircase code obtains the product-like structure from the block codes and the block-to-block dependency — the memory from the convolutional codes. Staircase codes take advantage of the streaming nature of the communication over optical transport networks (OTNs), where data is input to the decoder at a constant rate, which allows the overall code to have an indeterminate block length [7].

3

Analysis of Staircase Codes

Before the actual hardware implementation of the staircase decoder, we have to make sure that the decoder meets the specified requirements (mentioned in previous chapter). Section 3.4 describes the implementation of the staircase codes in MATLAB and reasons behind the parameter selection, along with performance evaluation of the staircase decoder. The purpose of the MATLAB implementation was not only to get the performance numbers, but also to further improve the understanding on how to implement the staircase decoder in VHDL.

Chapter 4 on the other hand describes the hardware implementation phase (in a 28-nm FD-SOI technology), from writing and implementing the functional blocks in VHDL to synthesis of the VHDL code. Here, the power and performance data gained from the synthesis is accounted for. This phase was started with the implementation of a simpler staircase design (e.g., constant parameters) after which further improvements were implemented (e.g., optimisation and parameterisation of the code).

3.1 BCH Code Parameters

Since one of the main components of the staircase codes is the BCH block code decoder, we studied the behaviour of the hard-decision BCH binary cyclic codes when changing the decoder parameters, like different codeword sizes and error-correction capabilities (n-, t- values respectively). For BCH code parameter and coding gain study, MATLAB built-in functions were used.

An *additive white Gaussian noise* (AWGN) channel was simplified for the use case of hard-decision decoding, where an error either occurred or not. The AWGN was simulated by using a *Binary Symmetric Channel* with the crossover probability, calculated according to Eq. 3.1, where Eb/N_0 is energy bit to noise spectral density ratio, R is code-rate and Q is the tail probability of the standard normal distribution. Noise was added to the data a using binary symmetric channel with crossover probability Pb.

$$Pb = Q\sqrt{\frac{2Eb}{N_0 \cdot R}} \tag{3.1}$$

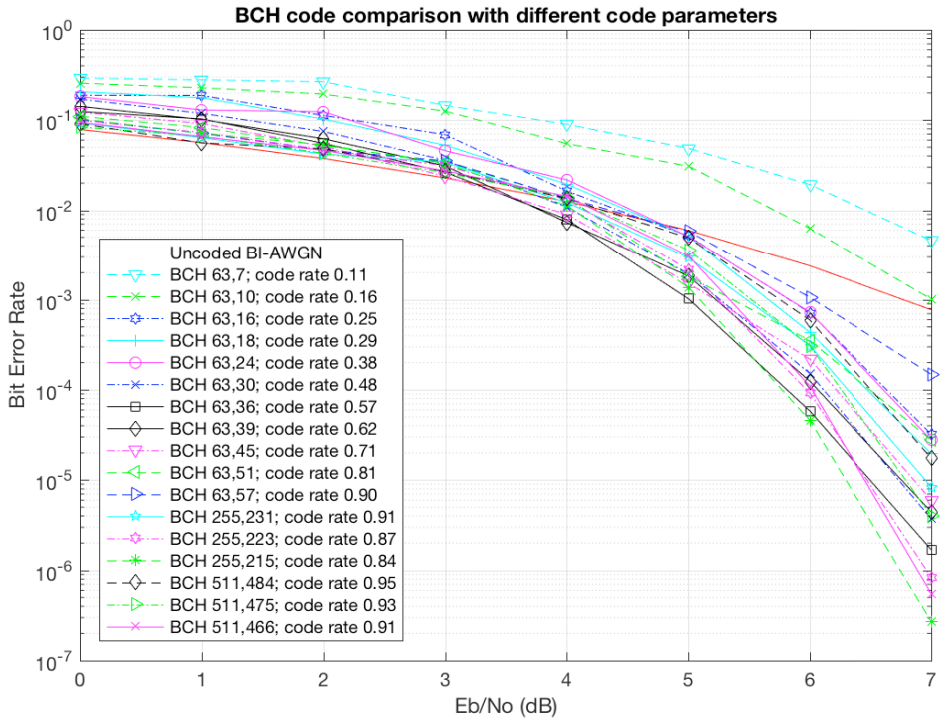

Figure 3.1: Comparison of BCH codes with different parameters

BCH code simulations with different parameter values are shown in Fig. 3.1, from which it can be concluded that the coding gain is better for larger codes with similar or constant code rates, which could be attributed to larger t values. Also, for constant error-correction capability, t, the coding gain was better for larger codes.

From these observations we estimated which BCH block parameters that were best suited for the staircase code. However, observations from the MATLAB simulations together with findings from a previous study by Zhang and Kschischang [29], resulted in a consensus of choosing codes with block length of $n = 255, 511$ with t-values of $t = 3, 4, 5$.

3.2 BCH Code Components

To speed-up MATLAB simulations, built-in encoder and decoder MATLAB-functions were used during the performance testing of the hard-decision staircase decoder. Additionally a BCH decoder utilising non-built-in decoding functions was used to verify the syndrome calculation algorithm, error-locator polynomial generation algorithm and Chien search algorithm.

A self-written MATLAB code for syndrome generation was used to generate syndromes of random input data, which was then used as input stimuli for the syndrome component test bench. The generated syndromes were also used for comparison to the outputed syndromes of the VHDL code. In the case of data mismatch the test bench would issue a report of data mismatch and stop the simulation.

The MATLAB simulations were more extensively used for the BMA block design and verification. Due to optimisations for the binary BCH codes and resulting smaller logic size, simplified inverse-free Berlekamp-Massey (SiBM) algorithm was used for generating error-locator polynomial, which it does in t cycles [26, 30]. Two main optimisations of reconfigured systolic architecture (RiBM) [25, 30, 31] are utilised in the SiBM algorithm. The first optimisation is that due to discrepancy in odd iterations being always 0, the odd iterations of the Berlekamp-Massey algorithm can be skipped for BCH codes [32]. The second optimisation is due to fact that the error-evaluator polynomial $\Omega(x)$ is not needed, since for binary BCH codes the error value is always 1. The pseudo-code describing the SiBM algorithm is shown in Algorithm 1 [26, 30].

Algorithm 1 Simplified inverse-free Berlekamp-Massey

Input: $S_j(j = 0, 1, 2, ..., 2t - 2)$
Initialisation: $\delta_{2t}(0) = 1, \delta_{2t-1}(0) = 0, \theta_{2t}(0) = 1, \theta_{2t-1}(0) = 0, \delta_i(0) = \theta_i(0) = S_j(j = 0, 1, 2, ..., 2t - 2), k(0) = 0, \gamma(0) = 1$
for $i = 0$ step 1 **to** $t - 1$ **do**
  Step 1:
  $\delta_j(i + 1) = \gamma(r)\delta_{j+2}(i) + \delta_0(i)\theta_{j+1}(i)(j = 0, 1, 2, ..., 2t)$
  Step 2:
  if $\delta_0(i) \neq 0$ **and** $k(i) \geq 0$ **then**
    $\theta_j(i + 1) = \delta_{j+1}(i)(j \neq 2t - 2 - 2i, 2t - 3 - 2i)$
    $\gamma(i + 1) = \delta_0(i)$
    $k(i + 1) = -k(i)$
  else
    $\theta_j(i + 1) = \theta_j(i)(j \neq 2t - 2 - 2i, 2t - 3 - 2i)$
    $\gamma(i + 1) = \gamma(i)$
    $k(i + 1) = k(i) + 1$
  end if
  $\theta_j(i + 1) = 0(j = 2t - 2 - 2i, 2t - 3 - 2i)$
end for
Output:
$\lambda_0 = \delta_0(t), \lambda_1 = \delta_1(t), ..., \lambda_t = \delta_t(t)$

Due to syndrome S_{2t-1} taking effect at the $(2t - 1)$-th iteration, according to the algorithm, the last syndrome can be excluded. During the i-th iteration, both the discrepancy polynomial updates in Step 1 – coefficients of the j-th; and Step 2 – coefficients of the *(j+1)*-th are computed simultaneously. After t iterations, the first $t + 1$ discrepancy coefficients – $\delta_0, ..., \delta_t$ are answer to $t + 1$ error-locator coefficients $\lambda_0, ..., \lambda_t$ – error-locator polynomial.

The pseudo-code [26, 30] in Algorithm 1 was used as reference for implementing the SiBM function in MATLAB, which was simulated and compared to built-in decoding function to verify the functionality, see Fig. 3.2.

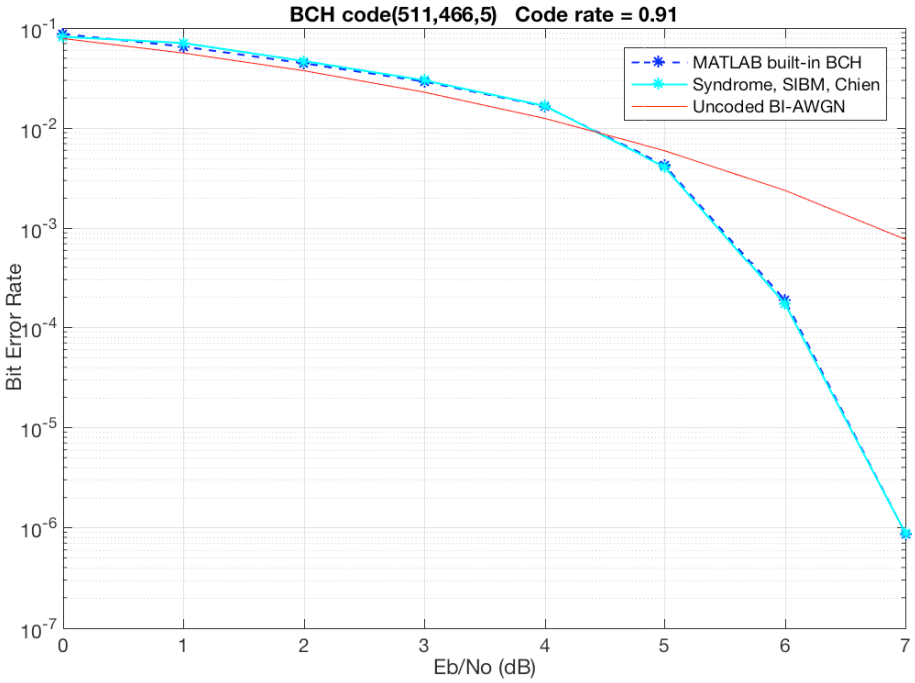

Figure 3.2: Comparison of coding gain for different BCH implementations in MATLAB, where the red line is BER of an uncoded data sent through the channel

From Fig. 3.2 it can be concluded that the both functions give very similar results, which reassures that the SiBM block was correctly implemented in MATLAB.

3.3 BCH Product Code

From the articles it can be deduced that the correctly implemented staircase codes should perform better than the product code [8, 29]. Therefore as a reference point for evaluating staircase code performance, a BCH product code was written and simulated in MATLAB. Simple row-by-row and column-by-column encoding and decoding algorithms were used. Due to the product code algorithm being suboptimal, simulations for evaluating coding gain and coding performance were necessary.

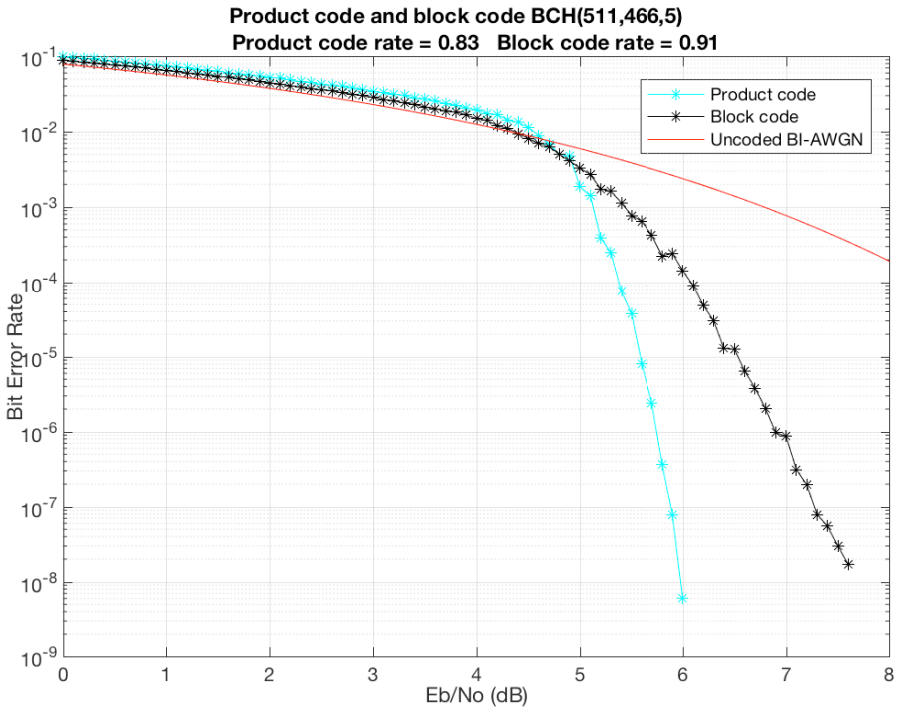

Figure 3.3: Comparison block code and product code

An example of a $n = 511$, $t = 5$ code is shown in Fig. 3.3. When simulating the product code, no iterative decoding was used, due to diminishing returns, the inner code (e.g., C_1) cannot fix the outer code's (e.g C_2) parity bits. Fig. 3.3 shows that the coding performance of the product code is better than the compared block code, using the same codeword parameters, which is to be expected, since the redundancy/parity bit overhead is increased in the case of the product codes. From the figure, the coding gain at bit-error rate of $2 \cdot 10^{-4}$ for the block code is evaluated to be about 2.1 dB and coding gain for the product code is about 2.7 dB. The obtained results give good basis for evaluating the staircase code.

3.4 Hard-decision BCH Staircase Code

The staircase code consists of blocks B and BT (where BT is the transposed data block) connected together in a staircase fashion seen in Fig. 3.4. All blocks are of the same size, i.e., $m \times m$, where $m = (n - 1)/2$.

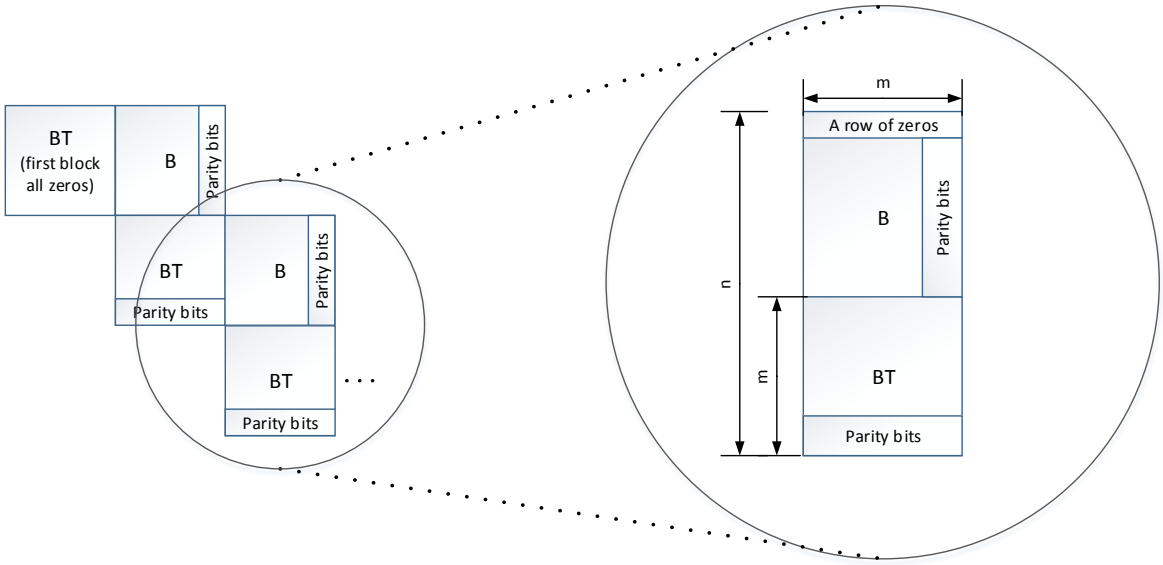

Figure 3.4: Illustrative model of the staircase codes

During the encoding, two blocks and (depending on the constellation of the blocks) a row (B+BT) or a column (BT+B) of zeros (needed to reach the length of n) are concatenated and then encoded, see Fig. 3.4. The result of this concatenation is of size $m \times n$, where m is the number of codewords of length n. The flow diagram for the staircase encoder is shown in Fig. 3.5.

Once all of the blocks are encoded, they are sent through the channel (described in Section 3.1). Last encoded block is the first block that is decoded, i.e., if the encoding is done down the staircase, the decoding should be done up the staircase and vice versa.

Staircase codes use a sliding window approach, which offers more accurate decoding through utilisation of the data of all of the blocks in the window combined with iterative decoding. Once a block is decoded correctly or the iteration bound is reached the sliding window will slide one step down the staircase, shown in Fig. 3.6, and the process is repeated.

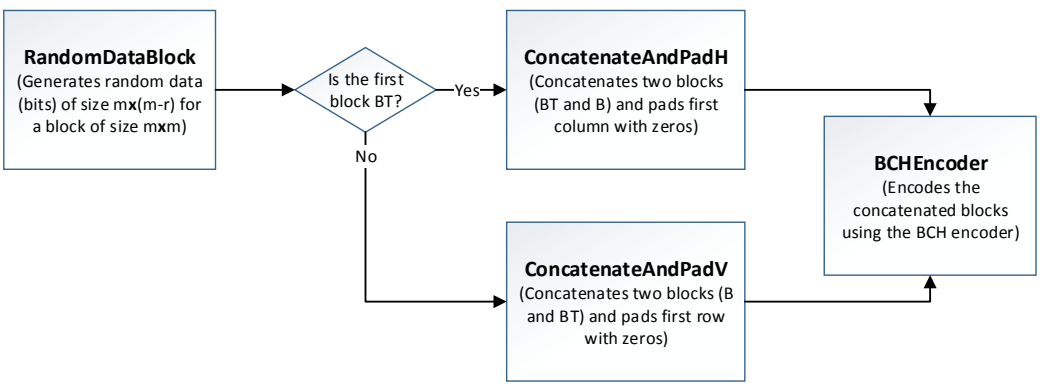

Figure 3.5: MATLAB model of the staircase encoder

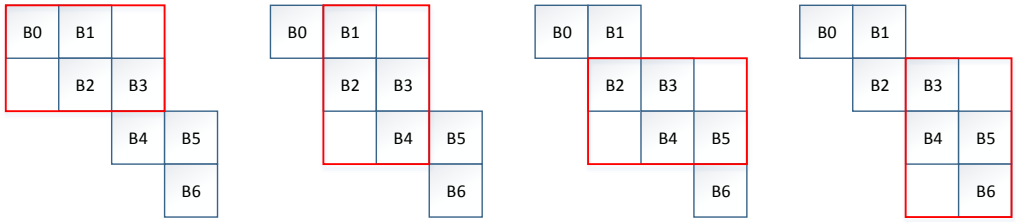

Figure 3.6: Example of the Staircase sliding window

After the channel, the decoding process is started by decoding the blocks placed in the sliding window of the staircase code. Once inside the window the syndrome for every block is checked. A MATLAB model of the syndrome check is shown in Fig. 3.7. The *GeneratorMatrix* function generates a generator matrix using the (n,k) values as input. The built-in MATLAB function *gen2par* generates a parity-check matrix using the generator matrix as input. Each parity-check vector is ANDed with the received message $r(x)$, if the number of ones in the resulting vector is 0 or an even number (i.e., if the result of the bitwise XORing of the vector is 0) the syndrome for that parity-check vector is equal to 0, otherwise 1. Syndrome checks for all of the parity-check vectors must be zero for the received message to be identified as error free.

Looking at the location of the parity bits in each block shown in Fig. 3.4 and the sliding window in Fig. 3.6 we can conclude that each block except for the first (all zero block) and the last, which are both discarded in the end, are decoded a minimum of two times.

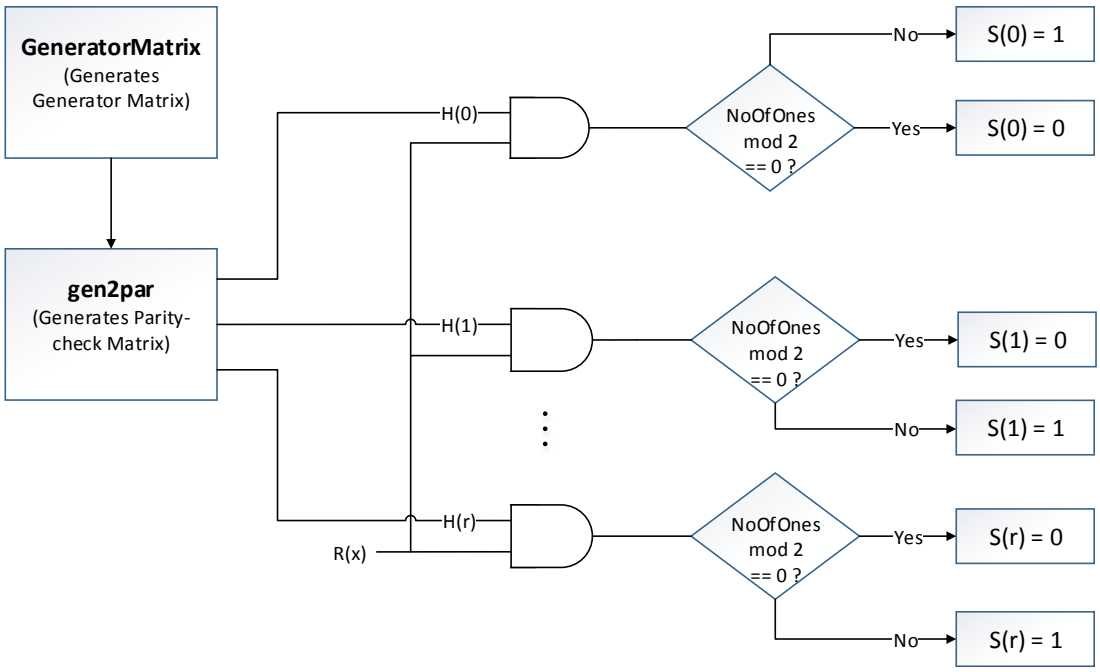

Figure 3.7: MATLAB model of the syndrome check

3.5 Performance of the Staircase Code

In the following section the three different versions of staircase code, written in MATLAB, are described and the results shown. The three versions of staircase code are:

1st version utilises the window size that changed dynamically from 2 to 4 blocks

2nd version uses fixed window length of 4

3rd version uses fixed window length of 6

The algorithm for the 1st version of the staircase code can be described as follows. When decoding is started, the decoder takes in one block B_1 and appends an all zero block B_0 to it. The resulting two blocks can then be decoded, where the decoding is performed until the iteration bound or the syndrome for the (first) two blocks is zero, meaning the decoding was successful (given that occurred errors $e \leq t$). If the decoding was successful, the data in the first block B_0 could be sent out and a new block B_2 is then appended to B_1, but due to first block from the initialisation being all-zero block, it is disregarded. However, if the decoding for the first two blocks was unsuccessful, meaning that the syndrome for the two first blocks at iteration bound was not zero ($S_1 \neq 0$), a new block B_3 is appended to the two blocks. Those three blocks are then decoded in similar manner, where during one decoding iteration, first B_3, B_2 are decoded, and then B_2, B_1 are decoded. Again the decoding is flagged being successful if the syndrome in the first block B_0 is zero ($S_1 = 0$), if not, B_4 is appended to the first three blocks.

When window size of 4 is achieved and the decoding fails ($S_1 \neq 0$), the first block is written out as is and a new block B_5 is taken in. In the initial case, the B_0 would still be disregarded, due to not containing any transmitted data. Although, upon decoding failure for the first block containing useful data, a flag can be appended to that written out block, notifying the system that the data is corrupt.

The idea of the 1st version of the staircase code was that, depending on the syndromes of consecutive first blocks being equal to zero ($S_1 = 0; S_1 = S_2 = 0; S_1 = S_2 = S_3 = 0$), more than one block can be written out, decreasing the effective window length, while at the same time after each decoding iteration, only one new block at a time is appended to the existing blocks.

In the hardware implementation, it is more beneficial to keep a static staircase window size, as it simplifies the control algorithm and overall hardware complexity. Also, if the window size is kept constant and if more than one consecutive syndrome after the decoding iteration are equal to zero (e.g., $S_1 = S_2 = 0; S_1 = S_2 = S_3 = 0$), then more than one new data block can be added into the window. This also allows for the speed up of both MATLAB simulations as well as hardware implementation (though a larger input buffer is needed). Therefore a simplified version of hard-decision BCH staircase decoder was implemented in MATLAB.

The 2nd version of the staircase decoder used a fixed window length of 4 and the 3rd version used a fixed window of 6. Both the fixed window and the dynamic window size codes performed similarly regarding the coding performance. Although due to fixed window simulations being faster and having more relevance to hardware implementation, the fixed window implementation is considered from here on as the primary algorithm to be used. The comparison between the block, product and

staircase codes is seen in Fig. 3.8.

As expected, the staircase code takes full advantage of the iterative decoding, as every new block can improve the decoding of the previous blocks in the staircase window. The staircase code with a 6-block window performed better than the 4-block window, which is to be expected. In the case of the 6-block window, by having more blocks in the window, the likelihood of successful decoding for the last block(s) by the additional new blocks in the window is improved. The observed results on the staircase window length are also in accordance with previous study by Smith et al. [8].

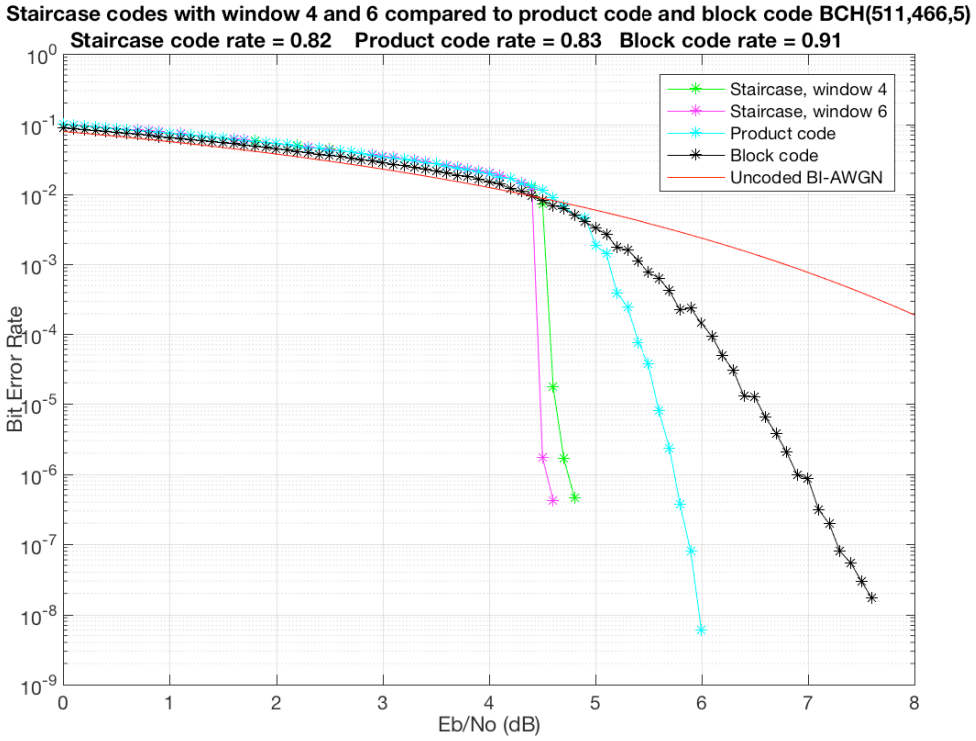

Figure 3.8: Comparison block code, product code and staircase codes with fixed window size of 4 and 6 blocks

3.6 Error Floor

In the case of iteratively decoded codes, an error floor is the occurrence probability of a particular error pattern that limits the maximum achievable BER to a certain level. For iteratively decoded product-like codes, those certain error patterns causing the decoder to fail are often called *stall patterns* [8]. Stall pattern refers to a state in which no (valid error-correcting) updates can be made by the decoder and thereby effectively stalling error-correcting capability of the decoder. A stall pattern can be defined as [8]: a set s of codeword positions, for which every row and column involving positions in set s has minimum $t+1$ positions in the same set s. Although stall patterns can be corrected, e.g., incorrect decoding that by accident causes one

or more bits to be corrected; it is assumed here, in order to evaluate a possible error floor, by the worst-case criteria that all the stall-patterns are uncorrectable. The staircase code has continuous, successive nature, which requires (possibly multiple) consecutive blocks to be accounted for stalling patterns [8].

A minimal stall pattern can be declared as exactly $t+1$ rows and columns with positions in set s. More about the proof of the number of minimal error pattern sets that can be assigned to a block can be read from [8]. For the staircase code, the number of minimal stall pattern sets that can be assigned to a block is defined as $\binom{a}{t+1}(\binom{2a}{t+1} - \binom{a}{t+1})$. The size of the minimal stall pattern set is defined as $S_{min} = (t+1)^2$, where $a = \frac{n_c}{2}$[8, 33].

The probability of positions in minimal stall pattern s being erroneous and caused by transmission errors is $p^{(t+1)^2}$. In some cases the errors in minimal stall pattern s could besides transmission errors also be caused by incorrect decoding(s), leading at some point in time during decoding to simultaneously erroneous positions in s. For fixed sets of s and l, where $1 \leq l \leq (t+1)^2$, there are $\binom{(t+1)^2}{l}$ ways in which $(t+1)^2 - l$ positions in set s can be received as being erroneous [8].

The error floor for generalised product codes can be approximated by

$$BER_{floor} = \frac{s_{min} M p^{s_m in}}{B}$$

where

$$B = a^2 = \frac{n_c^2}{4}$$

is defined as the number of bits in one of the staircase blocks [33]. Based on the previous studies in [8, 33], it is approximated that the error floor for the BCH component codes used in this thesis should stay way below the $\approx 10^{-15}$.

4
Implementation of Staircase Codes

In this chapter the hardware implementation and ASIC synthesis is covered. The staircase code synthesis using a a 28-nm fully-depleted silicon-on-insulator (FD-SOI) process technology library allows us to extract the timing, area and power consumption of the design. Based on the timing, also the throughput of the staircase decoder can be estimated.

4.1 Staircase Components

4.1.1 BCH Decoder

Input to the BCH decoder is the received message, *Rx*, and the output is the decoded message, *RxDecoded*. To avoid delays, the BCH decoder will execute all three of the BCH decoder computations (Syndrome, BMA and Chien search) but also the final XOR function simultaneously, see Fig. 4.1. In addition to that Syndrome and Chien search are implemented concurrently, which is evident looking at the block diagram shown in Fig. 2.3, since neither syndrome- nor Chien search component have a clock input.

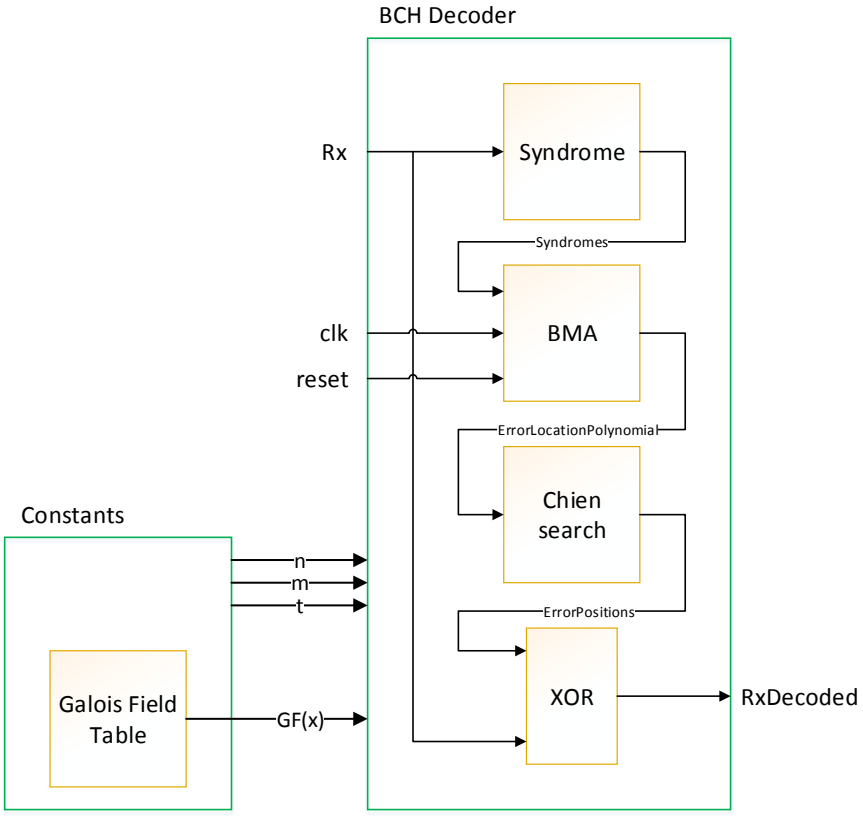

Figure 4.1: Block diagram of the BCH decoder

The final step of the BCH decoder is the XORing of the *ErrorPosition* vector (from the Chien search) with the received message. The *ErrorPosition* vector is of the same length as the received message with binary '1' on the error position(s), and binary '0' on all other positions.

For example, assume that an all zero codeword of length 15 is sent through the channel, and $R(x) = x^3+x^5+x^{12}$ or in binary $R = 000101000000100$ is received. We can see that three bits (on positions 4, 6, and 13) have been flipped, which means that the error position vector will have binary ones in those positions, i.e., *ErrorPosition* = 000101000000100. XORing the received message with the *ErrorPosition* vector will result in the original codeword.

Table 4.1: XORing $R(x) = x^3 + x^5 + x^{12}$ with *ErrorPosition*$= x^3 + x^5 + x^{12}$ will result in all zero vector, which is the original message

Rx	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
ErrorPosition	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
RxDecoded	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

It should be noted that an assumption is made that this is a (15,5) BCH-code with error-correcting capability of three bits (t=3), which means that all three errors are detected and corrected by the BCH decoder.

4.1.1.1 Syndrome Calculation

Fig. 4.2 shows the block diagram of the syndrome component. The method used to generate syndromes is explained in Section 2.2.3.1. Input to the syndrome component is *Rx*, which is a vector of size n, and the output is *Syndromes*, which is a matrix of size $2t \times m$. Elements of the Galois field are fetched from the Galois field table as shown in Fig. 4.2.

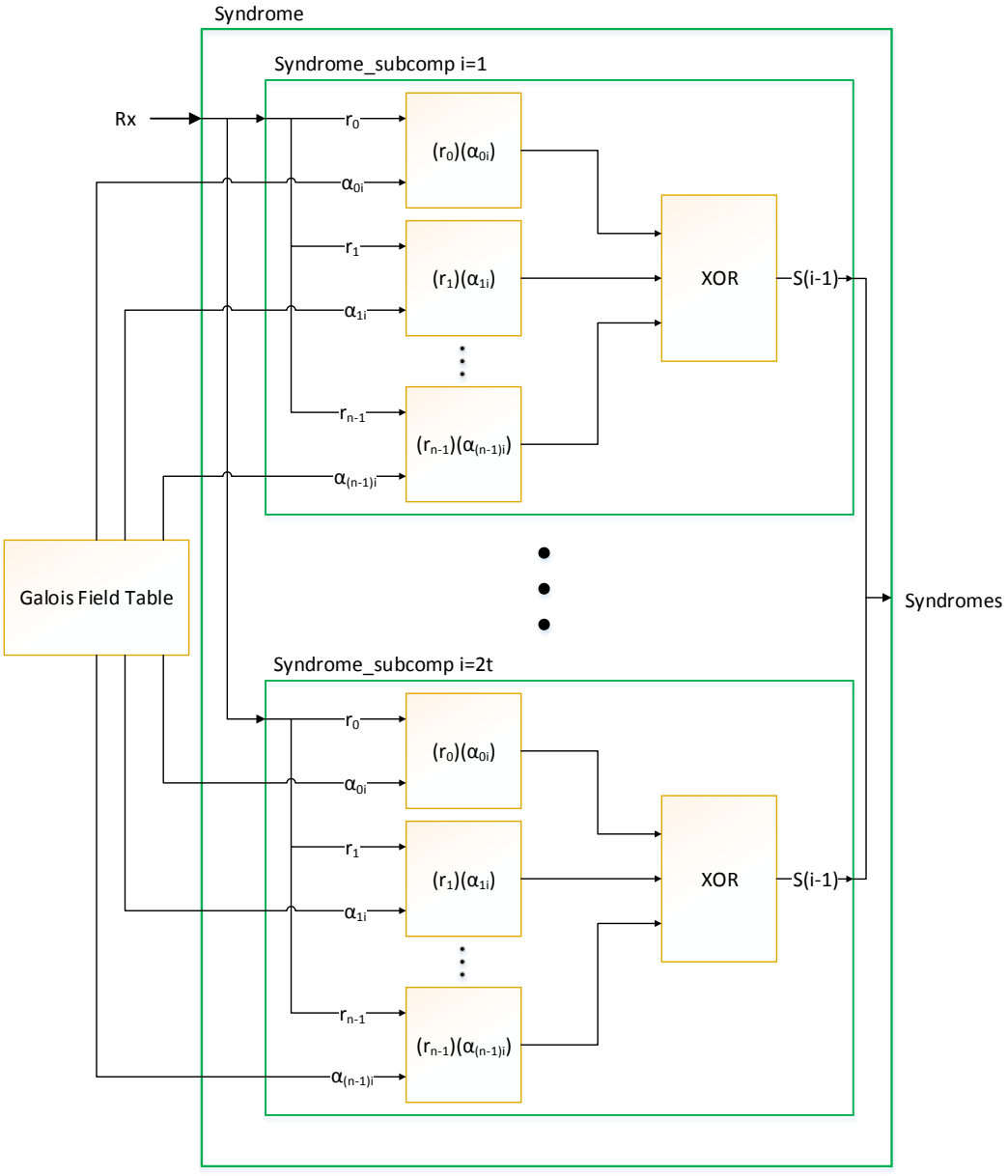

Figure 4.2: Block diagram of the Syndrome calculation

4.1.1.2 BMA

The SiBM algorithm, as discussed in Section 3.2, was chosen to be used in the BMA block of the BCH decoder. The hardware design was based on the block diagram in Fig. 4.3 showing the architecture [26, 30]. The $2t$ registers δ and θ registers are initialised according to Algorithm 1, where indexes i and $i+1$ denote current and operation cycle respectively.

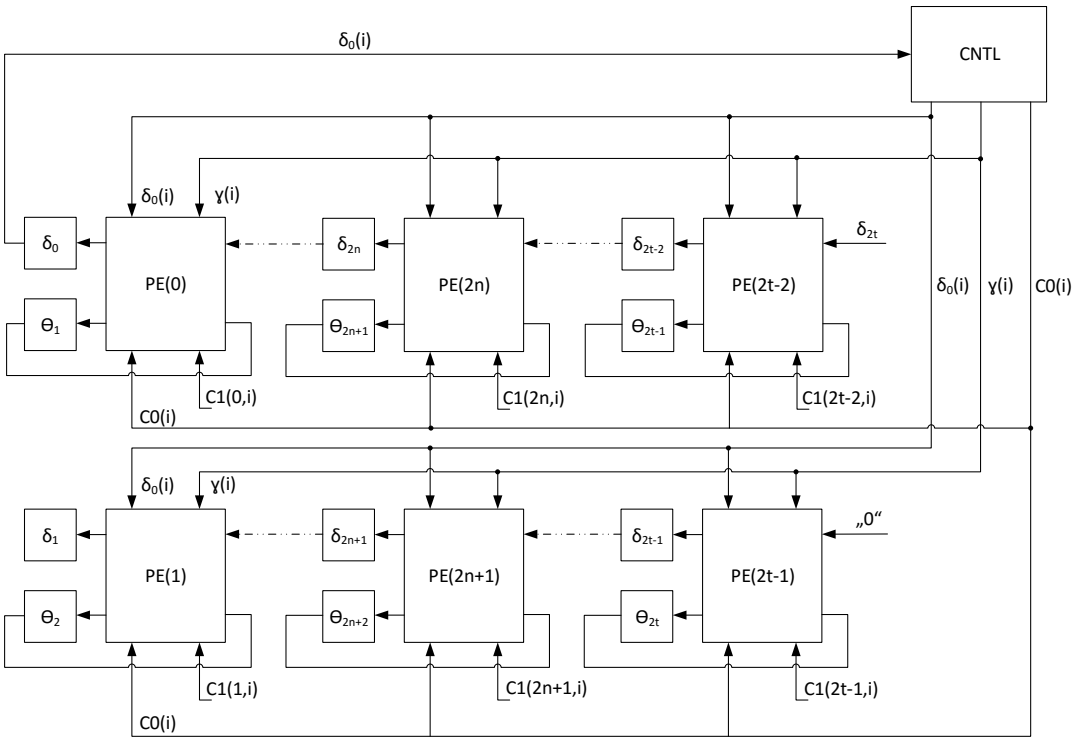

Figure 4.3: Block diagram of SIBM architecture

The original (no folding applied) SiBM architecture, without counting the control logic (CNTL), needs $2t$ finite field adders (FFA), $4t$ finite field multipliers (FFM), $2t+1$ registers and 2t multiplexers for hardware. The block diagram for the processing element (PE) in SiBM is shown in Fig. 4.4.

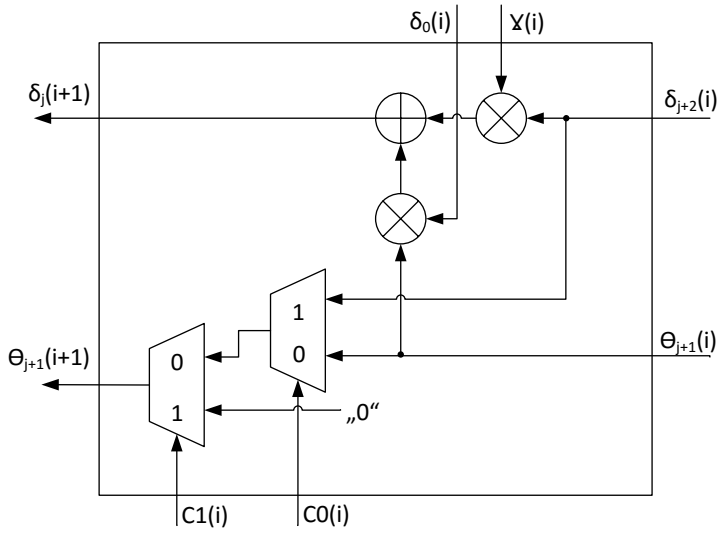

Figure 4.4: Block diagram of processing element architecture

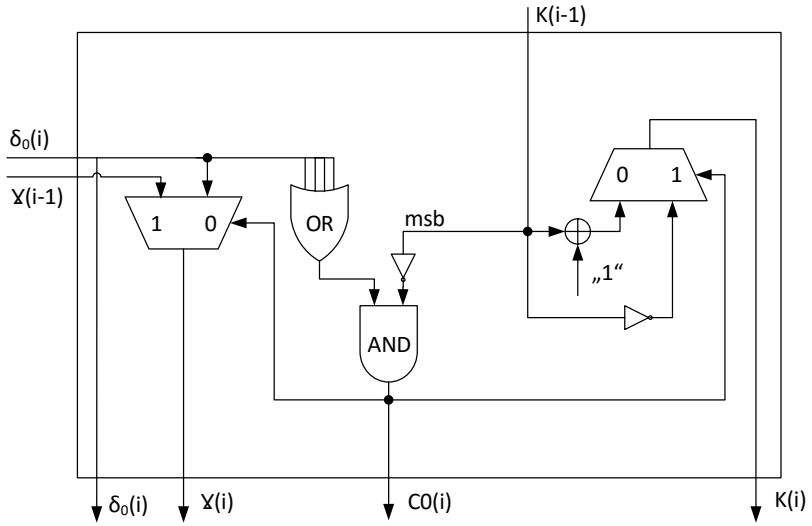

Figure 4.5: Block diagram of control (CNTL) unit architecture

Block diagram of the control logic (CNTL) is shown in Fig. 4.5 [26, 25, 30]. The control unit computes bitwise OR of the m-bit $\delta(i)$ vector in order to determine whether the discrepancy $\delta(i)$ is a nonzero value. If discrepancy is nonzero, then no change is necessary to the current state of the polynomial, otherwise the polynomial will be altered so that the discrepancy would become zero. The bitwise OR can be solved using $m-1$ two-input OR gates arranged in a binary tree of depth $\lceil \log_2 m \rceil$. The counter *k(i)* is implemented in twos-complement representation and therefore results in $k(i) \leq 0$ if and only if the MSB is '0'. The twos-complement arithmetic

addition in CNTL block is $k(i+1) = k(i) + 1$, whereas negation complements all bits in k and then adds one, i.e., $k(i+1) = -k(i) + 1$ [25].

The additional control signal C_1 is implemented as a shifter with $2t$ bits and where the contents shift 2 bits towards LSB at every iteration. The C_1 register is initialised as $(2t-2)$ and $(2t-3)$ -th bits $'1'$, others $'0'$. [26].

As previous studies have shown, the SiBM algorithm when compared to both reformulated inverse-free BM (riBM) and reconfigured systolic architecture (RiBM), gives about 1/3 reduction in number of processing elements and also halves the needed iteration clock cycles.

One of the more complicated operations carried out in the SiBM is the *finite-field multiplication* between two GF(2^m) polynomials. For the concise implementation of the SIBM algorithm, it was decided to keep coherent PB representation throughout the SiBM block. Keeping the GF(2^m) in PB representation yielded in relatively easy addition operation of logical XOR or modulo 2 addition. On the other hand multiplication in the PB representation requires more complicated logic for bit-parallel operation.

For flexibility of the decoder, the *multiplier* needed to be easily implementable for any field and also due to possible large degrees of m, the multiplier needed to be area efficient. The bit-parallel FFM algorithm by Reyhani-Masoleh and Hasan was chosen [18], as it was shown to need fewer routed signals than Mastrovito algorithm [19] and thus making it useful for VLSI design. Given multiplier is using optimisation and reformulation of the PB multiplication from Mastrovito's multiplier, achieving multiplier algorithm in terms of the *reduction matrix* Q that can be applied to any field (GF(2^m)) defining irreducible polynomial [18].

The gate-level schematic of a bit-parallel multiplier for minimal polynomial of $m=4$ is shown in Fig. 4.6. As can be seen, a multiplier requires m^2 AND and m^2-1 XOR gates, which meets the gate count expectations for trinomial in. Although it must be mentioned that the XOR gate count can vary depending of the used irreducible polynomial class, e.g., trinomials, all-one polynomials (AOPs) and equally spaced polynomials (ESPs) [18].

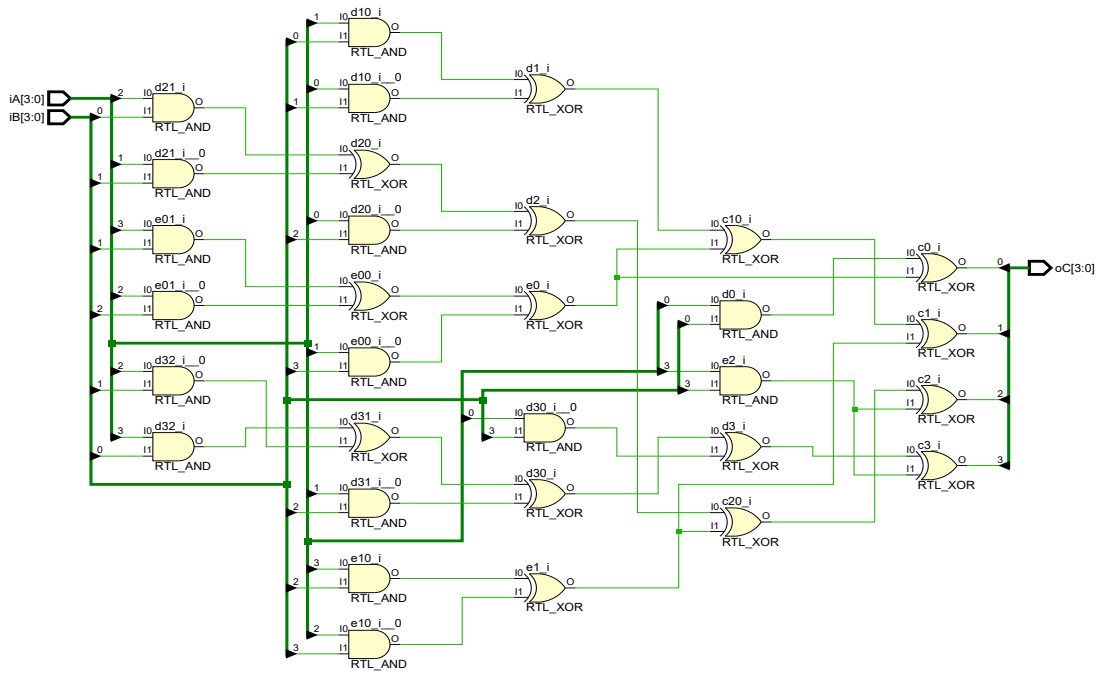

Figure 4.6: Gate-level schematic of bit-parallel polynomial basis (PB) multiplier for trinomial with $P(x) = x^4 + x + 1$

4.1.1.3 Chien Search

Fig. 4.7 shows the block diagram of the Chien search. The method used for generation of the error-location polynomial is explained in Section 2.2.3.3. Input to the component is the *ErrorLocPol* which is a matrix of size $(t + 1) \times m$.

To showcase the matrix representation of the error-location polynomial in the implementation let us use $\Lambda(x) = \alpha^4 + x^2 + \alpha^8 x^3$, as an example. Since this error-location polynomial is taken from an (15,5) BCH-code with an error-correcting capability of three bits (t=3), the size of the matrix is 4×4.

Table 4.2: Matrix representation of $\Lambda(x) = \alpha^4 + x^2 + \alpha^8 x^3$

0	0	1	1
0	0	0	0
0	0	0	1
0	1	0	1

Error-location polynomial received from the BMA has the following form:

$$\Lambda(x) = \alpha_{c1} + \alpha_{c2}x + \alpha_{c3}x^2 + \cdots + \alpha_{t+1}x^t$$

and each row represents a α_c-value starting from α_{c1}.

From Fig. 4.7 it can be concluded that the finite field multiplicators (FFM) are used for multiplications in GF, and that the elements of the GF are fetched from prestored GF-table.

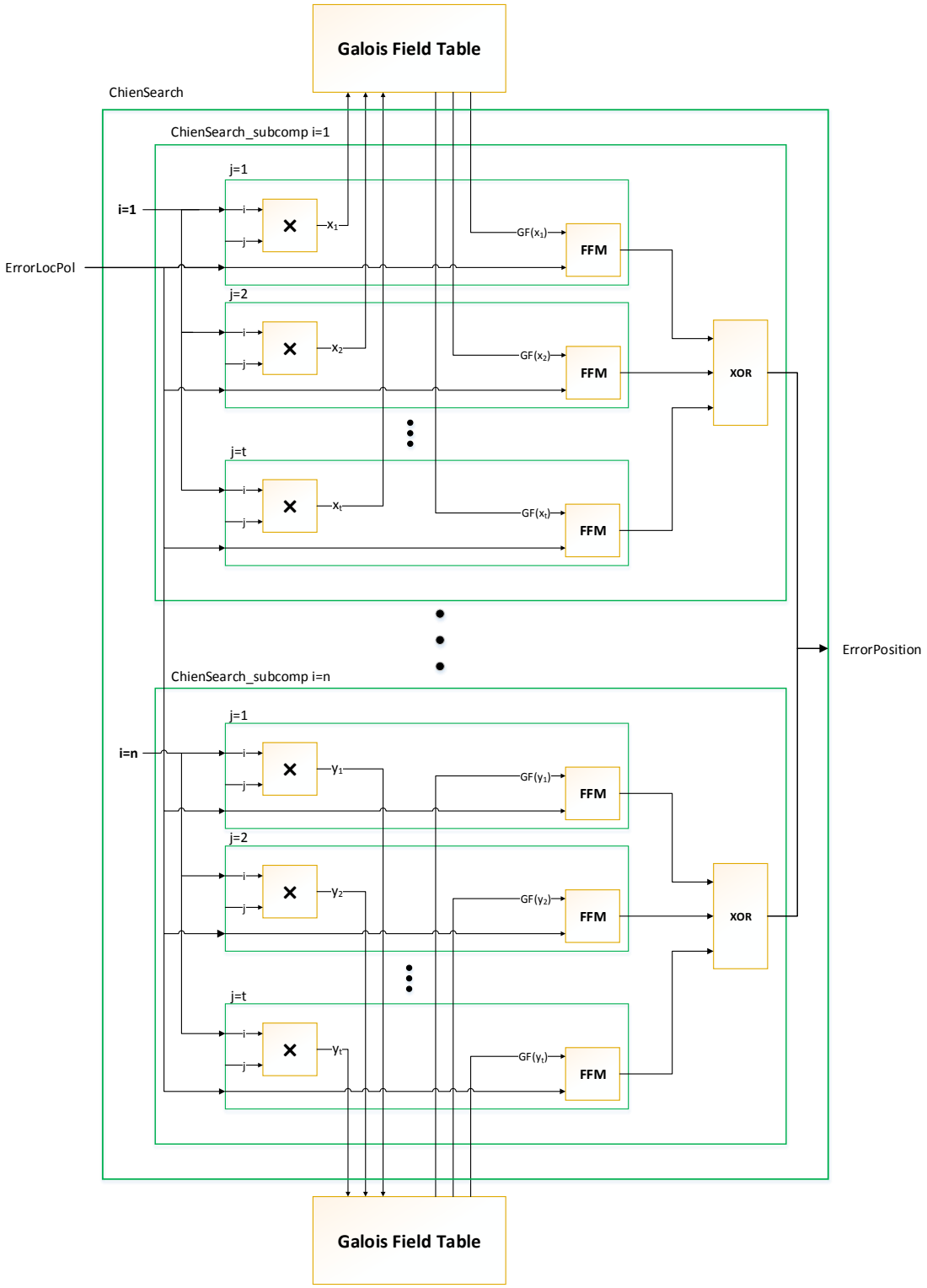

Figure 4.7: Block diagram of the Chien search

As was previously mentioned the output of the Chien search component is an ErrorPosition vector of size n with binary '1' on the error position(s) and binary '0' on all other positions.

4.1.2 Circular Buffer

The encoded blocks of data are sent through the channel and with each rising edge of the clock one block, starting with the block that is encoded last, will be sent to the circular buffer until the buffer is full, shown in Fig. 4.8. The circular buffer has a storing capacity of six blocks, and is a representation of the sliding window of Section 3.4.

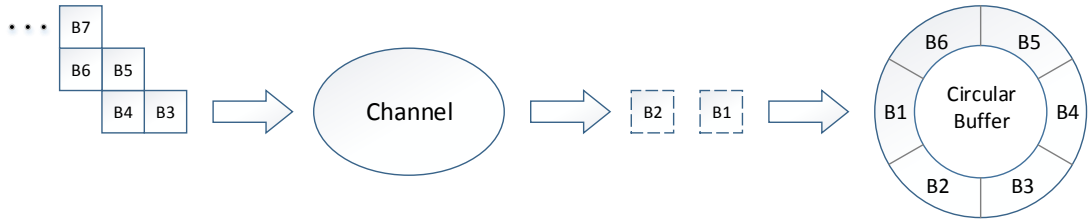

Figure 4.8: Circular buffer

The circular buffer is a first-in first-out (FIFO) dual-port memory. To keep track of the memory slots two address counters are used (hence the attribute dual-port), where one counter will keep track of the read data, while the other of the written data [34]. A circular buffer is used to avoid shifting the data since the last encoded block is the first one saved and eventually decoded. Shifting of data is associated with increasing power consumption, and is therefore to be avoided if possible.

The number of successfully decoded blocks is also the number of available slots in the buffer, since the successfully decoded blocks are sent through and the now empty memory slots are filled with new blocks arriving from the channel.

4.1.3 Staircase Control Logic

Control logic in the staircase decoder controls the decoder module and the data-flow from and back to the circular buffer. Control logic chooses which of the two data blocks from the circular buffer that is being transposed; how the blocks and zero-padding are concatenated; and read into the decoder module. From the decoder module, the control logic monitors which of the codeword syndromes that are zeros and thus further decoding can be skipped. Also an important task for the control logic is to have a row and column based mapping of the zero-syndrome codewords as illustrated in Fig. 4.9.

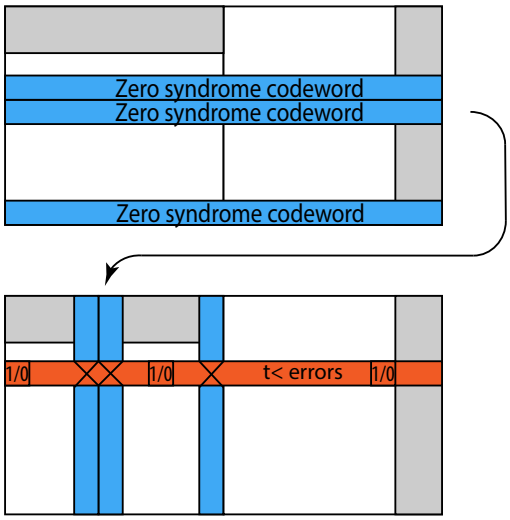

Figure 4.9: Mapping of row and column-wise zero syndromes for two adjacent decoding operations

In Fig. 4.9, it is shown how the zero-syndrome rows are mapped and kept in memory and once a new decoding operation is started, bit-flips for previously detected zero-syndrome columns will be disregarded, as marked with X-s. This granular control over zero-syndrome rows and columns complicates the design for the control logic, while on the other hand it improves the decoding performance, as next block in line cannot introduce errors to already fixed rows/columns.

5

Implementation Results

In the following chapter the hardware power consumption, timing and area of different staircase decoders are discussed.

The hardware synthesis was carried out using Cadence Encounter RTL Compiler (RC) environment and the slow-slow (SS), 0.9V, 125°C, 28-nm fully depleted silicon on insulator (FD-SOI) process technology library. The library uses nominal supply voltage and high operating temperature, which should give realistic power consumption of the circuit, along with SS corner, which gives pessimistic delay performance and thus the overall synthesis results are for relatively robust design.

5.1 BCH Decoder

When analysing the BCH decoder synthesis reports, the area in Table 5.2 was taken as the total area, where cell and net area are combined. When estimating the power consumption of the design, the circuit netlist was utilised and a simulation was run using a testbench and $10,000$ codewords as test vectors, where Eb/N_0 was chosen so that no uncorrectable errors occurred, i.e., occurred errors $\leq t$. The erroneous bit-flip probability for the channel, when generating test-vectors for the simulation, was chosen so that the decoder would operate at the Eb/N_0 region right of the code Eb/N_0 at BER of 10^{-15}. The operating region was estimated from the code graphs produced in MATLAB.

When comparing how the scaling of the decoder code size impacts the area, a difference was noted between the *SiBM* block and the *Chien search (CS)*, which can be explained by the parameter the component scales with. Namely, the *SiBM* scales with the parameters t and m, whereas the *CS* block scales with the parameters n and t.

The first synthesis analysis was carried out for the same clock delay constraint of 900 ps. This clock delay constraint was chosen as a baseline due to largest considered $n = 511, t = 5$ code size decoder requirement for achieving 100 Gbps data throughput.

Since the *syndrome* and *Chien search* blocks are parallel, while *SiBM* block is sequential, registers at the input and output were needed in order to correctly evaluate the timing throughout the BCH decoder design. However, those added registers at the input and output account for extra power consumption (see Table 5.1).

Table 5.1: BCH component code decoder synthesis results for the same timing requirement of 900 ps, where design hierarchy ungrouping is disabled

Parameter	n=15, t=3	n=255, t=5	n=511, t=5
SC area	142 μm^2	6654 μm^2	15362 μm^2
BM area	1529 μm^2	7880 μm^2	9153 μm^2
CS area	704 μm^2	67591 μm^2	166926 μm^2
Total area	2688 μm^2	86611 μm^2	200596 μm^2
Critical path	SiBM	SiBM and CS	SiBM and CS
Clock delay	900 ps	900 ps	900 ps
Timing slack	+ 249 ps	0 ps	0 ps

In Table 5.1, the hardware component grouping is disabled, so that no hardware is reused or shared between different parts of the design. This is useful when comparing the area scaling of different code size decoder sub-blocks (e.g., *syndrome, SiBM, Chien search*).

Additionally, the Cadence Encounter software can optimise the design by unwrapping/ungrouping the original design hierarchy, so that the circuit area can be reduced as shown in Table 5.2. For example, the re-usage of software modules is the reason why syndrome block is not showing up on the synthesis reports, as the syndrome block can be constructed by reusing the hardware blocks from *Chien search* and *SiBM* blocks. Hardware design ungrouping will from now on be considered as a default setting. This is also a reason why the synthesis reports provided here will only cover the total decoder area and power results. Due to the hardware ungrouping, there is no distinct and intact blocks remaining (e.g., *Syndrome, SiBM, Chien search*), since some of the hardware is shared in the design. Comparing two decoders with the same parameters, an 29% - 50% area decrease can be observed when ungrouping in Cadence Encounter is enabled. The aforementioned design optimisation through hardware sharing points towards a highly regular design (as seen previously in Section 4.1.1), which the tool's heuristics and optimisation algorithms are able to leverage. Although, it was noted that for the same decoder size of $n = 255, t = 5$ and clock delays of 440 ps and 450 ps, the area reduction was greater (50%) compared to the same decoder with looser timings of 900 ps (29%).

Table 5.2: BCH component code decoder synthesis results for the same timing requirement of 900 ps, with design ungrouping enabled

Parameter	n=15, t=3	n=127, t=5	n=255, t=5	n=511, t=5
Total area	1736 μm^2	20665 μm^2	61975 μm^2	$---$ μm^2
Total power	549 μW	3.25 mW	6.98 mW	— mW
Clock delay	900 ps	900 ps	900 ps	900 ps
Timing slack	+192 ps	0 ps	0 ps	0 ps

For one of the synthesis runs, for the $n = 255$ and $t = 5$ code, the clock delay constraint was set to 440 ps, which would result in a 97.7 Gbps throughput. From

Table 5.4 it can be seen that for achieving high throughput with smaller code size, there is a trade off in terms of power and area.

Although using clock delay constraint of 440 ps leaves the throughput just short of 100 Gbps, the results are still valid for feasibility analysis for usage as a component code in the staircase codes. The synthesis for clock delay constraint lower than 440 ps resulted in a negative slack and wasn't therefore successful.

Table 5.3: $n = 255$, $t = 5$ BCH component code decoder synthesis results for the timing requirements of 450 and 440 ps, with design hierarchy ungrouping disabled

Parameter	n=255, t=5	n=255, t=5
SC area	11294 μm^2	12313 μm^2
BM area	8974 μm^2	9561 μm^2
CS area	90684 μm^2	103366 μm^2
Total area	114962 μm^2	129238 μm^2
Clock delay	450 ps	440 ps

Table 5.4: $n = 255$, $t = 5$ BCH component code decoder synthesis results for the timing requirements of 450 and 440 ps, where design hierarchy ungrouping enabled

Parameter	n=255, t=4	n=255, t=5	n=255, t=5
Total area	44451 μm^2	60390 μm^2	63044 μm^2
Total power	9.81 mW	13.61 mW	14.25 mW
Clock delay	550 ps	450 ps	440 ps
Throughput	101.4 Gbps	95.5 Gbps	97.7 Gbps

From the synthesis results of the BCH decoder it can be seen that for code of $n = 255$ and $t = 5$, the 100 Gbps is unfeasible, as it falls short of 100 Gbps and would result in fairly high power density for the circuit. The SiBM would need fewer cycles for computing the ELP if code with low error-correcting capability was used, thus increasing the BCH decoder throughput. It is estimated that a clock delay of 557ps is required to achieve 100 Gbps with $n = 255$, $t = 4$ code. From the attained results we know that this is achievable.

For the upcoming synthesis, the timing requirement for larger codes of code size $n = 511$ and $n = 255$ will be adjusted accordingly to fulfill the 100 Gbps throughput requirement. For smaller codes, e.g., code $n = 15$, $t = 3$, the clock delay of the circuit should be around 16 ps (62.5 GHz) and for $n = 127$, $t = 5$ – 254 ps (3.94 GHz) in order to achieve 100 Gbps throughput, which makes achieving high throughput with small codes unfeasible.

5.2 Staircase Decoder Analysis

The hardware synthesis reports for BCH decoder are shown in the Tables 5.2 and 5.4. Based on the synthesis results, it is possible to estimate the staircase decoder size

and power consumption. As there are two staircase decoder architecture corner cases in terms of possible degree of parallelism, we also provide two different estimation schemes in Tables 5.5 and 5.6 respectively. Based on the estimates for the whole staircase, the final degree of concurrency can be decided on, e.g., component code decoder for every other, every fifth or every tenth codeword. The estimation formulas for the whole staircase decoder architecture are provided for both 4 and 6 blocks in the staircase, as well as for $\frac{n}{2}$ and one component decoder as parallel and serial block decoder respectively.

Table 5.5: Staircase decoder complexity estimation for higher level of block decoder parallelism

Circular buffer	$(6\ or\ 4) \times n \times n$
Decoder	$\frac{n}{2} \times BCH\_decoder(n)$
Staircase CNTL logic	$n \times SiBM\_CNTL(m)$
In total	$(6\ or\ 4) \times \frac{n^4}{2} \times BCH\_decoder(n) \times SiBM\_CNTL(m)$

Table 5.6: Staircase decoder complexity estimation for lower level of block decoder parallelism

Circular buffer	$(6\ or\ 4) \times n \times n$
Decoder	$BCH\_decoder(n)$
Staircase CNTL logic	$SiBM\_CNTL(m)$
In total	$(6\ or\ 4) \times n^3 \times BCH\_decoder(n) \times SiBM\_CNTL(m)$

By prioritising the requirement for the throughput analysis, staircase decoder size and power consumption will first be carried out on the fully parallel architecture, where there is a decoder for every codeword in the block. As shown in Table 5.7, the staircase decoder throughput can be calculated as

$$\frac{BCH\ throughput \times \frac{n-1}{2}}{(window\ size - 1) \times iteration\ bound}\ ,$$

where the iteration bound is equal or larger than size of the staircase window. In the following it is assumed that the iteration bound is equal to a staircase window size, as it allows the possible error correction to propagate through the decoding window once.

Table 5.7: Staircase decoder size estimation, when there is a component decoder for every codeword in decoded block

Parameter	n=255, t=4	n=255, t=5	n=255, t=5
Total area	5.65 mm^2	7.67 mm^2	8.01 mm^2
Total power	1.25 W	1.73 W	1.81 W
Clock delay	550 ps	450 ps	440 ps
Throughput for window length 4	1073.1 Gbps	1010.7 Gbps	1034.0 Gbps
Throughput for window length 6	429.3 Gbps	404.3 Gbps	413.6 Gbps

As shown in the table, the high throughput of 100 Gbps for the staircase decoder is achievable, furthermore, it is possible to reduce the design area by trading some of the excess throughput. In Table 5.8 the staircase decoder size is estimated for a semi-parallel design, where there is a BCH decoder for every fourth code row.

Table 5.8: Staircase decoder size estimation 6 block window size and semi-parallel design, when there is a component decoder for every fourth codeword in decoded block

Parameter	n=255, t=4	n=255, t=5	n=255, t=5
Total area	1.4 mm^2	1.9 mm^2	2.0 mm^2
Total power	314 mW	436 mW	456 mW
Throughput for window length 6	107.3 Gbps	101.1 Gbps	103.6 Gbps

In Table 5.9 the staircase decoder size is estimated for the 4 block window and decoder is for every tenth codeword in the block being decoded The following semi-parallel architecture would give closest throughput to fulfilling the 100 Gbps requirement along with area and power consumption reduction.

Table 5.9: Staircase decoder size estimation 4 block window size and semi-parallel design, when there is a component decoder for every tenth codeword in decoded block

Parameter	n=255, t=4	n=255, t=5	n=255, t=5
Total area	0.6 mm^2	0.8 mm^2	0.8 mm^2
Total power	128 mW	177 mW	185 mW
Throughput for window length 4	107.3 Gbps	101.1 Gbps	103.4 Gbps

6
Discussion

In the following sections an overview of observations made throughout the project are provided, along with limitations and future work regarding the project.

6.1 Observations

From the MATLAB analysis we can conclude that the staircase code provides good results in comparison with product code, which is also in line with articles on staircase codes [29, 8, 8]. Additionally it was observed from the MATLAB analysis that the inclusion of the single row/column-based syndrome check and effective handling of the incorrect error-corrections is necessary for reaching the expected low coding gain at low BER of 10^{-15}. Since MATLAB simulations for the staircase code are computation heavy, emulation on an FPGA is required in order to reach the low BER of 10^{-15}.

From the hardware implementation and synthesis, it can be concluded that not only does the parallel BCH decoder implementation facilitate the expected throughput, it also exceeds it. The component code level parallelism allows for fine tuning of the staircase decoder design in terms of area, power consumption and coding performance, thus giving wide degree of design choices.

6.2 Limitations

Fully parallel implementation requires a highly optimised design, a significant design time, and quite large area, which could be problematic for emulating long staircase codes on an FPGA.

6.3 Future Work

As the project ended up involving a bit-parallel BCH decoder implementation in hardware, the given time for Master thesis project fell short of achieving all the expected goals, e.g., staircase decoder hardware implementation.

Regarding the simulation and analysis tools it is proposed that the MATLAB staircase decoder model is improved to leverage parallel code execution and therefore reduce run times. Additionally the row and column wise syndrome mapping would be of benefit in the implemented staircase decoder model in MATLAB. The

aforementioned two proposals would make it feasible to evaluate the coding performance at low BER before FPGA emulation.

Future work proposed regarding the hardware would be to continue the implementation of the staircase decoder, mainly the control logic, also to emulate the hardware implementation on an FPGA in order to evaluate the achieved coding performance at a low BER of 10^{-15}. The hardware implementation of the BCH decoder could be even more improved. For example the BMA block in BCH decoder could be further optimised by using *Further-optmised inverse-free BM* (FiBM) algorithm, opposed to SiBM, which would decrease the needed logic by about 25% [26].

7
Conclusion

In this Master's Thesis we have implemented and analysed both the BCH component codes and the staircase codes in MATLAB. We have also implemented the BCH decoder in hardware, and synthesized it in 28-nm fully-depleted silicon-on-insulator (FD-SOI) library.

Even though the staircase decoder has not been implemented, power and performance numbers of the decoder have been presented. The presented numbers are an estimation made possible by successful implementation, synthesis and analysis of the BCH decoder. Of course the estimation is still just that and validity of the numbers needs to proved by implementation and then synthesis of the staircase decoder. The validation of the estimated performance numbers would be made possible through emulation of the implemented staircase decoder on an FPGA.

By analysing the estimated performance numbers we can for now conclude that the performance requirements stated by the OTN are surpassed by the staircase decoder, possibly making the staircase codes *de facto* error-correcting code for high-speed fibre-optic communication systems.

Bibliography

[1] Y. Jiang, A practical guide to error-control coding using MATLAB, 1st Edition, Artech House, Boston, 2010.

[2] H. Imai, Essentials of error-control coding techniques, Academic Press, San Diego, Calif, 1990;2014;.

[3] R. H. Morelos-Zaragoza, The Art of error correcting coding, 2nd Edition, Wiley, Chichester, 2006.

[4] W. W. Peterson, W. E. J., Error-correcting codes, 2nd Edition, The MIT press, 1972.

[5] Wikipedia, Hamming Code — Wikipedia, the free encyclopedia, [Online; accessed 09-February-2016] (2016).
URL `https://en.wikipedia.org/wiki/Hamming_code`

[6] S. Lin, D. J. Costello, Error control coding: fundamentals and applications, 2nd Edition, Pearson Prentice Hall, Upper Saddle River, N.J, 2004.

[7] B. P. Smith, Error-correcting codes for fibre-optic communication systems, Ph.D. thesis, Department of Electrical & Computer Engineering, University of Toronto (2011).

[8] B. P. Smith, A. Farhood, A. Hunt, F. R. Kschischang, J. Lodge, Staircase Codes: FEC for 100 Gb/s OTN, Journal of Lightwave Technology 30 (1) (2012) 110–117.

[9] Wikipedia, Low-density parity-check code — Wikipedia, the free encyclopedia, [Online; accessed 09-February-2016] (2016).
URL `https://en.wikipedia.org/wiki/Low-density_parity-check_code`

[10] C. Häger, A. Amat, H. D. Pfister, A. Alvarado, F. Brännström, E. Agrell, On Parameter Optimization For Staircase codes, Optical Fiber Communication Conference and Exposition.

[11] G. Tzimpragos, C. Kachris, I. B. Djordjevic, M. Cvijetic, D. Soudris, I. Tomkos, A survey on FEC codes for 100 G and beyond optical networks, IEEE Communications Surveys & Tutorials 18 (1) (2014) 209–221.

[12] G. Bell, T. Hey, A. Szalay, Beyond the data deluge, Science 323 (5919) (2009) 1297–1298.

[13] G. Hu, J. Sha, Tail-biting code: A modification to staircase code for high-speed networking, in: Signal Processing Systems (SiPS), 2015 IEEE Workshop on, IEEE, 2015, pp. 1–5.

[14] D. Truhachev, L. M. Zhang, F. Kschischang, Information transfer bounds on iterative thresholds of staircase codes, 2015 Information Theory and Applications Workshop.

[15] H. Kristian, H. Wahyono, K. Rizki, T. Adiono, Ultra-fast-scalable BCH decoder with efficient-Extended Fast Chien Search, in: Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, Vol. 4, IEEE, 2010, pp. 338–343.
[16] R. Hill, A first course in coding theory, Oxford University Press, 1986.
[17] S. S. Adams, Introduction to algebraic coding theory, Franklin W. Olin College: NSF CCLI, 2008.
[18] A. Reyhani-Masoleh, M. A. Hasan, Low complexity bit parallel architectures for polynomial basis multiplication over GF(2m), IEEE Transactions on Computers 53 (8) (2004) 945–959.
[19] E. D. Mastrovito, VLSI designs for multiplication over finite fields GF (2m), in: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Springer, 1988, pp. 297–309.
[20] E. D. Mastrovito, VLSI architectures for computations in Galois fields, Linköping univ. Dep. of electrical engineering, 1991.
[21] P. Barreto, V. Rijmen, et al., The Whirlpool hashing function, in: First open NESSIE Workshop, Leuven, Belgium, Vol. 13, 2000, p. 14.
[22] F. Rodríguez-Henríquez, N. A. Saqib, A. D. Perez, C. K. Koc, Cryptographic algorithms on reconfigurable hardware, Springer Science & Business Media, 2007.
[23] E. R. Berlekamp, Algebraic coding theory, McGraw-Hill, New York, 1968.
[24] J. Massey, Shift-register synthesis and BCH decoding, IEEE Transactions on Information Theory 15 (1) (1969) 122–127.
[25] D. V. Sarwate, N. R. Shanbhag, High-speed architectures for Reed-Solomon decoders, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9 (5) (2001) 641–655.
[26] M. Yin, M. Xie, B. Yi, Optimized algorithms for binary BCH codes, in: 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), IEEE, 2013, pp. 1552–1555.
[27] C. Yang, Y. Emre, C. Chakrabarti, Product Code Schemes for Error Correction in MLC NAND Flash Memories, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20 (12) (2012) 2302–2314.
[28] R. Pellikaan, X.-W. Wu, S. Bulygin, R. Jurrius, Error-correcting codes, 2015.
[29] L. M. Zhang, F. R. Kschischang, Staircase Codes With 6% to 33% Overhead, Journal of Lightwave Technology 32 (10) (2014) 1999–2002.
[30] W. Liu, J. Rho, W. Sung, Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories, in: 2006 IEEE Workshop on Signal Processing Systems Design and Implementation, IEEE, 2006, pp. 303–308.
[31] H. Burton, Inversionless decoding of binary BCH codes, IEEE Transactions on Information Theory 17 (4) (1971) 464–466.
[32] L. L. Joiner, J. J. Komo, Decoding binary BCH codes, in: Southeastcon'95. Visualize the Future., Proceedings., IEEE, IEEE, 1995, pp. 67–73.
[33] C. Häger, H. D. Pfister, A. G. i Amat, F. Brännström, Density Evolution and Error Floor Analysis for Staircase and Braided Codes, in: Optical Fiber Communication Conference, Optical Society of America, 2016, pp. Th2A–42.

[34] P. J. Ashenden, Digital Design An Embedded Systems Approach Using VHDL, Morgan Kaufmann, 2008.