Design and Implementation of Reed-Solomon Decoder using Decomposed Inversion less Berlekamp-Massey Algorithm

by

Hazem Abd Elall Ahmed Elsaid

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
May 2010

Acknowledgment

I would like to thank my supervisors Prof. Dr. Amin Nassar, and Dr. Hossam Aly Hassan Fahmy who devoted a lot of their precious time to this work. Also I would like to thank Dr. Tallal El-shabrawy, owner of the credit in this work. Nothing could be achieved without their help. They helped me not only to complete this work but also to plan for my academic life.

I am actually indebted to the faculty members in Cairo University and in GUC Prof. Dr. Serag El din Habib, Prof. Dr. Yasser Hegazy, Prof. Dr.Ahmed El mahdy, and Dr Amr Tallat, for their sincere advices are very fruitful.

I would like also to thank my study partner Hamed Salah. Also I would like to thank Walid Galal and Soukry Ibrahim for their valuable advices. I can not explain my gratitude to my father, and mother. They all helped me to achieve progress through my life. And my wife who always prays for me and encourage me to do my best, and my Sweet baby Salma.

Abstract

Reed-Solomon (RS) codes have a widespread use to provide error protection especially for burst errors. This feature has been an important factor in adopting RS codes in many practical applications such as wireless communication system, cable modem, computer memory. The most popular RS decoder architecture , can be summarized into four steps : 1) calculating the syndromes from the received codeword; 2) computing the error locator polynomial and the error evaluator polynomial; 3) finding the error locations; and 4) computing error values. This thesis proposes an area efficient, low energy, high speed architecture for a Reed-Solomon RS(255,239) decoder based on Decomposed Inversionless Berlekamp-Massey Algorithm, where the error locator and evaluator polynomial can be computed serially. In the proposed architecture, a new scheduling of t finite field multipliers is used to calculate the error locator and evaluator polynomials to achieve a good balance between area, latency, and throughput. This architecture is tested in two different decoders. The first one is a two parallel decoder, as two parallel syndrome and two parallel Chien search are used. The second one is a serial decoder, as serial syndrome and Chien search are used. In our architectures we have investigated hardware area, throughput, and energy per symbol and we did a good optimization between the latency, throughput, and energy per symbol while maintaining a small area.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Information and Coding Theory

Information theory analyzes the communication between a transmitter and a receiver through an unreliable channel. We can divide the information theory into two main parts, the first part is used to analysis sources information, especially the information produced by a given source, and, the second part states the conditions for performing reliable transmission through an unreliable channel. One of the most used techniques in information theory is called "Coding", which is used to optimize transmission and to make efficient use of the capacity of a given channel.

In general, coding is a technique used to transmit a set of messages in a set of binary bits called codewords. This called "Source coding", see an example in Table 1.1, in this table we have four symbol messages and we coded each symbol by two bits and this is called source coding. We notice that the optimum number of bits to represent the messages is two bits and if we increased these bits, the effective rate will be decreased.

If these symbols are transmitted through a noisy channel, an error may occur and the symbol may be decoded as another symbol leading to an undetectable

Table 1.1: Codeword after only source coding

Messages	Codewords
S_1	00
S_2	01
S_3	10
S_3	11

Table 1.2: Codeword after source coding and channel coding

Messages	Binary messages	Codewords
S_1	00	0000
S_2	01	0101
S_3	10	1010
S_3	11	1111

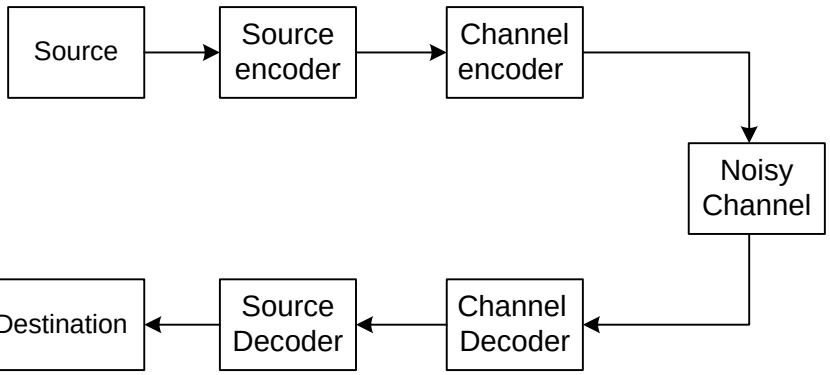

Figure 1.1: A communication system: source and channel coding

error. So another coding block is needed called channel coding which adds parity check bits to each message to make a distance between valid codewords as shown in Table 1.2. When we increase the parity check length the distance between each two codewords is increased and the probability of error is decreased but the effective rate is decreased, so it is a trade off between the rate and the probability of error.

A block diagram of a communication system as related to information theory is shown in Figure 1.1.

The block diagram seen in Figure 1.1 shows two types of encoders/decoders.

- Source encoder/decoder.
- Channel encoder/decoder.

1.2 Error control Coding

Last years there were an increasing interest in the reliability of data transmission and storage mediums, as if a single error happened all the system may be damaged due to an unacceptable corruption for the data, e.g. in a bank account [1]. The simplest way of detecting a single error is a parity check sum [2]. But

in some applications this method is not sufficient and different methods must be implemented.

If the transmission system transmits data in both directions, an error control strategy may be determined by detecting an error and then, if an error is occurred, retransmitting the corrupted data. These systems are called Automatic Repeat Request (ARQ) [3]. If transmission transfers data in only one direction, e.g. information recorded on a compact disk, the only way to control the error is with Forward Error Correction (FEC) [3 - 5]. In FEC systems some redundant data is concatenated with the information data in order to allow for the detection and correction of the corrupted data without having to retransmit it.

1.3 Linear Block Codes

Error control coding mechanism is done in two inverse operations. The first one is a mechanism of adding redundancy bits to the message and form a codeword, this operation called (encoding operation), the second operation is excluding the redundancy bits from the codeword to achieve the message and this operation called (decoding operation).

These types of codes are called block codes and are denoted by $C(n, k)$. The rate of the code, $R = k/n$, where k represents the message bits and n represents the coded bits. Since the 2^k messages are converted into codewords of n bits. This encoding procedure can be understood a conversion from message vector of k bits located in space of size 2^k to a coded vector of size n bits in a space of size, and 2^k only selected to be valid codewords.

Linear block codes [2,6] are considered the most common codes used in channel coding techniques. In this technique, message words are arranged as blocks of k bits, constituting a set of 2^k possible messages. The encoder takes each block of k bits, and converts it into a longer block of $n > k$ bits, called the coded bits or the bits of the codeword. In this procedure there are $(n-k)$ bits that the encoder adds to the message word, which are usually called redundant bits or parity check bits. As explained in the previous section. The codewords generated from the encoder is linearly combined as the summation of any two codeword is an existing codeword so it is called Linear Block Codes.

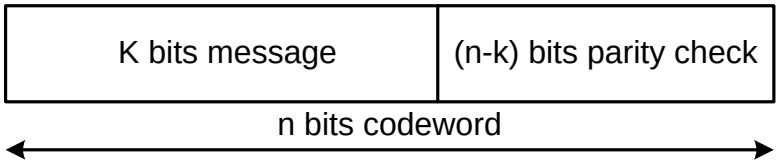

Figure 1.2: Systematic form of a codeword of a linear block code

1.4 Linear Block Codes in Systematic Form

In this case, the first k bits are the message bits as it is and the remaining $(n-k)$ bits are called parity check or redundancy bits. The structure of a codeword in systematic form is shown in Figure 1.2

In this thesis, the message bits is converted into symbols "non-binary code" called Reed Solomon (RS) code [7] which is a special type of cyclic code which will be explained briefly in chapter 2. The message symbols are placed at the beginning of the codeword, while the redundancy symbols are placed at the end of the codeword.

1.5 Hardware solution

Last years implementation of Reed Solomon decoders were using Digital Signal Processors (DSP) micro controllers, which are based on microprocessors but specialized in signal processing. Recently, Field Programmable Gate Arrays (FPGAs) [8] is used for these applications, as they provide similar performance with more customized design.

FPGAs are customizable logic devices, as they give fast solution for specific problems. FPGAs is considered a good step towards the ASIC design which is the most optimum way in area, power consumption, and price.

1.6 Overview of thesis

The organization of this thesis is as follows. In Chapter 2, a good introduction to Reed Solomon (RS) codes and their properties and applications and Galois Fields will be discussed, then RS encoder is presented. In Chapter 3, the general architecture of RS decoder is discussed, then the decoding algorithm of Berlekamp Massey and Euclidean Algorithms are discussed. In Chapter 4, The decomposed inversionless Berlekamp Massey algorithm with a new architecture called the modified

evaluator decomposed inversionless Berlekamp Massey architecture is discussed with two proposals, serial proposal and two parallel proposal, finally the simulation results and synthesis reports and comparison between these proposals and the other architectures are discussed. In Chapter 5, the conclusions and the future work are made.

Chapter 2

REED-SOLOMON CODE

Reed Solomon code is considered non-binary code based on Finite field called Galois Fields (GF). So first of all it is necessary to clarify the area of Galois Fields.

2.1 Galois Field (GF)

2.1.1 Properties of Galois Field

The main properties of a Galois field [5] are:

- All elements of GF are defined on two operations, called addition and multiplication.
- The result of adding or multiplying two elements from the Galois field must be an element in the Galois field.
- Identity of addition "zero" must be exist, such that $a + 0 = a$ for any element a in the field.
- Identity of multiplication "one" must be exist, such that $a * 1 = a$ for any element a in the field.
- For every element a in the Galois field, there is an inverse of addition element b such that $a + b = 0$. This allows the operation of subtraction to be defined as addition of the inverse.

- For every non-zero element b in the Galois field, there is an inverse of multiplication element b^{-1} such that $bb^{-1} = 1$. this allows the operation of division to be defined as multiplication by the inverse.

- Both addition and multiplication operations should satisfy the commutative, associative, and distributive laws.

2.1.2 Galois field $GF(2)$ "Binary Field"

The simplest Galois field is $GF(2)$. Its elements are the set $\{0, 1\}$ under modulo-2 algebra. The addition and multiplication tables of $GF(2)$ are shown in Tables 2.1 and 2.2.

+	0	1
0	0	1
1	1	0

Table 2.1: Modulo-2 addition (XOR operation).

×	0	1
0	0	0
1	0	1

Table 2.2: Modulo-2 Multiplication (AND operation).

There is a one-to-one correspondence between any binary number and a polynomial with binary coefficients as every binary number can be presented as a polynomial over $GF(2)$. A polynomial of degree K over $GF(2)$ has the following general form:

$$f(x) = f_0 + f_1 X + f_2 X^2 + \cdots + f_K X^K \tag{2.1}$$

where the coefficient $f_0, \ldots, f_K$ are the elements of $GF(2)$ i.e it can take only values 0 or 1. A binary number of $(K+1)$ bits can be represented as a polynomial of degree K by taking the coefficients equal to the bits and the exponents of X equal to bit locations. In the polynomial representation, a multiplication by X represents a shift to the right.

For example, the binary number 10011 is equivalent to the following polynomial:

$$10011 \leftrightarrow 1 + 0X + 0X^2 + X^3 + X^4$$

The first bit ("position zero" the coefficient of X^0) is equal to 1, the second bit ("position one" the coefficient of X) is equal to 0,the third bit ("position two"the coefficient of X^2) is equal to 0, and so on.

2.1.3 Galois Field $GF(2^m)$

Galois field is considered extension to Binary Field. Let us suppose that we wish to create a finite field $GF(q\ )$ where q must be a prime number [4,5].

For Example: assume we have $GF(8)$ with the elements of $\{0, 1, 2, 3, 4, 5, 6, 7\}$. this can not be considered Galois Field as for the reasons:

- There is no multiplicative inverse for all elements in the field (e.g., 6 has no inverse).
- The identity element under multiplication is not unique for some elements (e.g., $4 * 1 = 4 * 3 = 4$).

So we can construct the $GF(2^m)$, where (m is an integer) by taking a primitive element of the field and assign the symbol α "alpha" to it. The powers of $\alpha$are from α^0 to α^{2^m-2}, $(2^m - 1)$ terms and the last term is zero.

The element α^{2^m-1} will be equal to α^0 [4], and higher powers of α will repeat the lower powers found in the finite field. The best way to understand how to add the powers of alpha is to examine the case:

$$\alpha^{2^m-1} = \alpha^0 = 1$$

Since in $GF(2^m)$ algebra, plus (+) and minus (-) are the same [4], the last one can be represented as follows:

$$\alpha^{2^m-1} + 1 = 0$$

Construction of Galois field $GF(2^m)$ elements [4] is based on primitive polynomial called $p(X)$ with degree m, this polynomial must be factor of $X^n + 1$, and should be not only irreducible but also primitive to guarantee unique elements representation.

For example: In $GF(2^3)$ the factors of (X^7+1) are:

$$X^7+1=(X+1)(X^3+X+1)(X^3+X^2+1)$$

Both the polynomials of degree 3 are primitive and can be chosen, so let's choose the polynomial shown in equation 2.2

$$p(X)=X^3+X+1 \tag{2.2}$$

This polynomial has no solution in binary field. The primitive element α is the solution for the primitive polynomial [4], so equation 2.2 is converted to equation

$$p(\alpha)=\alpha^3+\alpha+1=0 \tag{2.3}$$

Since in $GF(2^m)$, $+1=-1$, α^3 can be represented as follows:

$$\alpha^3=1+\alpha$$

So the other non-zero elements of the field are now found to be

$$\begin{aligned}
\alpha^4 &= \alpha.\alpha^3=\alpha.(1+\alpha)=\alpha+\alpha^2\\
\alpha^5 &= \alpha.\alpha^4=\alpha.(\alpha+\alpha^2)=\alpha^2+\alpha^3=\alpha^2+(1+\alpha)=1+\alpha+\alpha^2\\
\alpha^6 &= \alpha.\alpha^5=\alpha.(1+\alpha+\alpha^2)=\alpha+\alpha^2+\alpha^3=1+\alpha^2\\
\alpha^7 &= \alpha.\alpha^6=\alpha(1+\alpha^2)=\alpha+\alpha^3=1=\alpha^0
\end{aligned}$$

Note that $\alpha^7=\alpha^0$, and therefor the eight finite field elements ($2^m=2^3=8$) of $GF(2^3)$, generated by equation (2.2), are $\{0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\}$, and all elements starting from α^4 to α^6 are presented function of α^0,α, and α^2 which are called the basis of the Galois field, and this is will be discussed in details in the following sections.

In general, extended Galois field $GF(2^m)$ includes 2^m elements, where m is the symbol size (in bits). For example, in ADSL systems, the Galois field is always $GF(2^8)=GF\ (256)$, where $m=8$ (is fixed number). It is generated by the following primitive polynomial:

$$P(X)=1+X^2+X^3+X^4+X^8 \tag{2.4}$$

As we said in the previous section that there were an one-to-one mapping between polynomials over $GF(2)$ and binary numbers, now here in $GF(2^m)$ there is one-to-one mapping between polynomials over $GF(2^m)$ and symbols of length m.

each symbol can be presented in binary form of length m.

2.1.4 Representation of Galois Field Elements

Let α be a primitive element of $GF(2^3)$ such that the primitive polynomial is given by:

$$p(\alpha) = \alpha^3 + \alpha + 1 = 0 \tag{2.5}$$

The following tables shows three different ways to represent elements in $GF(2^3)$:

Power Form	Polynomial Form	Binary Form $\alpha^0, \alpha^1, \alpha^2$
–	0	000
0	1	100
1	α	010
2	α^2	001
3	$1 + \alpha$	110
4	$\alpha + \alpha^2$	011
5	$1 + \alpha + \alpha^2$	111
6	$1 + \alpha^2$	101

Table 2.3: Different representations of $GF(2^3)$ elements.

The first column of Table 2.3 represents the powers of α. The second column shows the polynomial representation of the field elements. This polynomial representation is obtained from equation 2.5. And the last column of Table 2.3 is the binary representation of the field elements, where the coefficient of α^2, α^1 and α^0, taken from the second column, are represented as binary numbers and present the basis of the field and it will be discussed in details in the next section.

2.1.5 Basis of Galois Field $GF(2^m)$

Basis in $GF(2^m)$ is a set of m linearly independent symbols which can represent other symbols with a unique form with a linear combination of these basis based on the primitive polynomial [9].

Assume that we have $\beta = \beta_0, \beta_1, ..., \beta_{m-1}$ are the basis of $GF(2^m)$. Let a is any general element in the field and denoted by $(a_0, a_1, ..., a_{m-1})$, a can be represented as shown in equation 2.6.

$$a = a_o\beta_0 + a_1\beta_1 + ... + a_{m-1}\beta_{m-1} \qquad a_i \in GF(2). \tag{2.6}$$

There are a large number of possible basis for any $GF(2^m)$.

Basis can be classified into three types:

- Polynomial (Standard) Basis.
- Normal Basis.
- Dual Basis.

This classification based on the optimization for the hardware and the need of the applications.

- Polynomial Basis

In this type of basis we choose the first m symbols excluding zero [9], i.e. we choose $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$.

For example consider $GF(2^3)$ with $p(x) = x^3 + x + 1$. Take α as a root of $p(x)$ then the polynomial basis of this field will be $\{1, \alpha, \alpha^2\}$ and all 8 elements can be represented as:

$$a = a_0 + a_1\alpha + a_2\alpha^2 \tag{2.7}$$

where the $a_i \in GF(2)$. These basis coefficients can be stored in a basis table of the kind shown in Appendix B.

- Dual Basis

Dual basis is one of the most important types of basis which used to gain an efficient hardware for RS encoders and decoders [10], as it is used in Galois field multipliers. The difference between polynomial basis and dual basis is only reordering to the symbols.

For example: Let's deal with $GF(2^4)$ with primitive polynomial $p(X) = X^4 + X + 1$

- The standard (polynomial) basis for this field is $\{\alpha^0, \alpha^1, \alpha^2, \alpha^3\}$.

- The dual basis for this field is $\{\alpha^0, \alpha^3, \alpha^2, \alpha^1\}$.

For more details in conversion from polynomial basis to dual basis or from dual basis to polynomial basis refer to [11].

Appendix C includes the tables of conversions from polynomial basis to dual and vice versa.

- Normal Basis

Normal basis is a basis which is useful when we need squaring in our calculations [12].

since if $(a_0, a_1, ..., a_{m-1})$ are the normal basis representation of $a \in GF(2^m)$ then $(a_{m-1}, a_0, a_1, ..., a_{m-2})$ is the normal basis representation of a^2. This property make the hardware is more efficient [12].

In this thesis we used only polynomial basis as in circuit implementation we needed only polynomial basis multipliers.

2.1.6 Implementation of $GF(2^m)$ Arithmetic

Implementation of Galois field arithmetic is totally different to the implementation of infinite field arithmetic, for example, multiplication in Galois field, if we need to multiply $2 * 4$ it may give 3 or 5 or any other value based on the field implementation, but in the infinite field arithmetic $2 * 4$ must equal to 8. So implementation of Galois field arithmetic circuits is different.

The most common implementation in Galois field arithmetic are addition and multiplication.

- Addition operation may be considered as an XOR operation. Implementation of Galois field adders regardless the basis representation of field elements can be formulated as shown in equation 2.8. Hence a $GF(2^m)$ adder circuit can be implemented with 1 or m XOR gates depending on whether the basis coefficients are represented in series or parallel. This is an important feature of $GF(2^m)$ and make the implementation of addition is easy operation and need limited hardware.

$$\begin{aligned} a &= b + c \\ &= (b_0 + b_1\alpha + ... + b_{m-1}\alpha^{m-1}) + (c_0 + c_1\alpha + ... + c_{m-1}\alpha^{m-1}) \\ &= (b_0 + c_0) + (b_1 + c_1)\alpha + ... + (b_{m-1} + c_{m-1})\alpha^{m-1} \end{aligned} \tag{2.8}$$

- Multiplication is more complex than addition as the addition is only XOR operation, but multiplication is different and depends on the Galois field structure and its primitive polynomial and the type of basis coefficients. Finite field multipliers can be classified into three types:

1. Multipliers by constant.
2. Serial multipliers.
3. Parallel multipliers.

1- Multipliers by constant

The General Galois field multipliers are with two variable inputs, but sometimes in RS encoders and decoders we need special type of Galois field multipliers with only one variable input this type of multipliers called multiplier with constant [9].

Let $a = a_0 + a_1\alpha + \ldots + a_{m-1}\alpha^{m-1}$ be an element in $GF(2^m)$ where α is a root of the primitive polynomial $p(x) = x^m + \sum_{j=0}^{m-1} p_j.x^j$ Thus

$$a * \alpha = a_0\alpha + a_1\alpha^2 + \ldots + a_{m-1}\alpha^m \tag{2.9}$$

but since $p(\alpha) = 0$

$$a * \alpha = a_0\alpha + a_1\alpha^2 + \ldots + a_{m-1}(p_0 + p_1\alpha + p_2\alpha^2 + \ldots + p_{m-1}\alpha^{m-1}) \tag{2.10}$$

For example consider, multiplication by α in $GF(2^3)$, where $p(x) = x^3 + x + 1$. Then

$$\begin{aligned} a &= a_0 + a_1\alpha + a_2\alpha^2 \\ a * \alpha &= a_0\alpha + a_1\alpha^2 + a_2\alpha^3 \\ &= a_3 + (a_3 + a_0)\alpha + a_1\alpha^2 + a_2\alpha^3 \end{aligned} \tag{2.11}$$

and this multiplication can be implemented in two ways

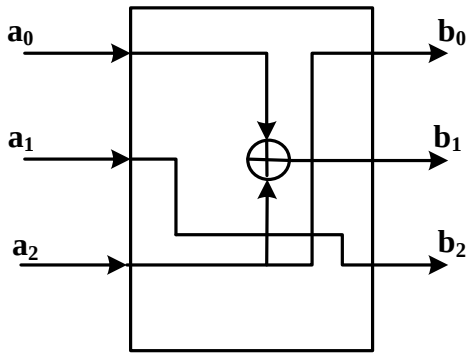

Figure 2.1: Circuit for computing $a \leftarrow a * \alpha$in $GF(2^3)$combinational

- Combinational as shown in figure 2.1 as the value of $a * \alpha$ is generated instantaneous .

- Sequential as shown in figure 2.2 here the value of $a * \alpha$ is generated after only one clock cycle .

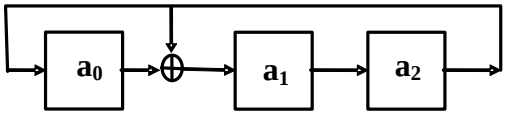

Figure 2.2: Circuit for computing $a \leftarrow a * \alpha$ in $GF(2^3)$ sequential

These multipliers can be implemented with general multipliers (two variable multipliers) but these multipliers is optimized in area, delay, and power consumption. In this thesis combinational multipliers is used.

2- Serial Multiplier

In This type of multipliers we have two variable inputs, and either these inputs enter the circuit serially and generate the output parallel or enter the circuit in parallel and generate the output serially.

we have a lot of methods to perform the serial multipliers, but the most common serial multipliers are Berlekamp multiplier [13], and Massey-Omura Multiplier [12], and Polynomial basis multipliers.

the most common multipliers in these multipliers are polynomial basis multipliers, as they do not need basis conversion circuit, so this multiplier can be summarized as follow:

Polynomial basis multipliers

Polynomial basis multipliers operate over polynomial basis i,e both the two input variables and the output are with the same type of basis (polynomial basis), so we are not in need of basis conversion circuit.

There are two different types of polynomial basis multipliers:

- least significant bit (LSB) first.
- most significant bit (MSB) first.

a- Option L - LSB first

In Option L - LSB first multiplier [14] one of the two inputs enter serially from the least significant bit to the most significant bit, and the other is entered in parallel, after m clock cycles the output will be valid.

Let a, b, c $\in GF(2^m)$ and

$$\begin{aligned} a &= a_0 + a_1\alpha + ... + a_{m-1}\alpha^{m-1} \\ b &= b_0 + b_1\alpha + ... + b_{m-1}\alpha^{m-1} \\ c &= c_0 + c_1\alpha + ... + c_{m-1}\alpha^{m-1} \end{aligned} \tag{2.12}$$

$$c = a * b \tag{2.13}$$

$$c = (...(((a_0 b) + a_1 b\alpha) + a_2 b\alpha^2) + ...) + a_{m-1} b\alpha^{m-1} \tag{2.14}$$

It is clear from equation (2.14) that b is multiplied by a_0 then we enter the second coefficient a_1 and during this b is multiplied by α using LFSR, then multiply $a_1$by $b\alpha$ and generate $a_1 b\alpha$. In general at step i we generate $a_i b\alpha^i$ where $(i = 0, 1, ..., m-1)$, which are accumulated in the registers.

Figure 2.3 clarify the circuit implementation of Option L - LSB first under $GF(2^3)$, and primitive polynomial $P(x) = x^3 + x + 1$.

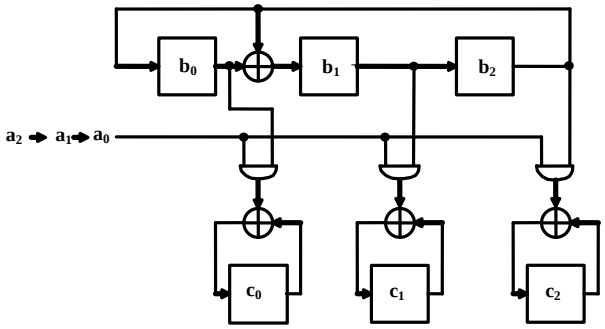

Figure 2.3: Circuit for Option L - LSB first multiplier in $GF(2^3)$.

b- Option M - MSB first

In Option M - MSB first multiplier [15] one of the two inputs enter serially from the most significant bit to the least significant bit, and the other is entered in parallel, after m clock cycles the output will be valid.

Let a, b, c $\in GF(2^m)$ and

$$\begin{aligned} a &= a_0 + a_1\alpha + ... + a_{m-1}\alpha^{m-1} \\ b &= b_0 + b_1\alpha + ... + b_{m-1}\alpha^{m-1} \\ c &= c_0 + c_1\alpha + ... + c_{m-1}\alpha^{m-1} \end{aligned} \tag{2.15}$$

$$\begin{aligned} c &= a * b \\ &= (a_0 + a_1\alpha + ... + a_{m-1}\alpha^{m-1}) * b \end{aligned} \tag{2.16}$$

$$c = (...(((a_{m-1}b)\alpha + a_{m-2}b)\alpha + a_{m-3}b)\alpha + ...)\alpha + a_0 b \tag{2.17}$$

It is clear from equation (2.17) that b is multiplied by a_{m-1} then the result is multiplied by α using LFSR, then the second coefficient a_{m-2} is entered and multiplied by b and the result is accumulated in the registers and multiplied by α using LFSR, after m clock cycle the output is generated.

Figure 2.4 clarify the circuit implementation of Option L - LSB first under $GF(2^3)$, and primitive polynomial $P(x) = x^3 + x + 1$.

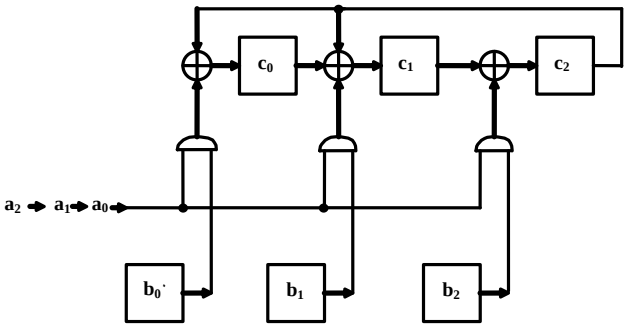

Figure 2.4: Circuit for Option M - MSB first multiplier in $GF(2^3)$.

3-Parallel multiplier

This type of multipliers take two variable inputs, these inputs enter to the circuit in parallel to generate the output in parallel instantaneous. This feature is required in some applications, like RS codes, the serial multipliers is very slow to be adopted to its speed, so we used in RS codes parallel multipliers.

we have a lot of method to perform the parallel multipliers, but the most common parallel multipliers are Dual Basis Multipliers [16], Normal basis multipliers [15], and Polynomial basis multipliers [12].

the most common multipliers in these multipliers are polynomial basis multipliers, as they do not need basis conversion circuit, so this multiplier can be summarized as follow:

Polynomial basis multipliers

Polynomial basis multipliers operate over polynomial basis i.e, both the two input variables and the output are with the same type of basis (polynomial basis), so we are not in need of basis conversion circuit.

Let a, b, c $\in GF(2^m)$ and

$$\begin{aligned} a &= a_0 + a_1\alpha + ... + a_{m-1}\alpha^{m-1} \\ b &= b_0 + b_1\alpha + ... + b_{m-1}\alpha^{m-1} \\ c &= c_0 + c_1\alpha + ... + c_{m-1}\alpha^{m-1} \end{aligned} \tag{2.18}$$

$$c = a * b \tag{2.19}$$

$$c = (...(((a_0 b) + a_1 b\,\alpha) + a_2 b\alpha^2) + ...) + a_{m-1} b\alpha^{m-1} \tag{2.20}$$

From equation 2.20 we should implement $m-1$ multipliers by constant (α) to get $b * \alpha^j$, where ($i = 0, 1, \ldots, m-1$).

we can present $b * \alpha^j$ as:

$$b * \alpha^j = b_{j,0} + b_{j,1}\alpha + b_{j,2}\alpha^2 + ... + b_{j,m-1}\alpha^{m-1}. \tag{2.21}$$

Therefore using 2.20 and 2.21

$$c_j = a_0 b_{0,j} + a_1 b_{1,j} + a_2 b_{2,j} + ... + a_{m-1} b_{m-1,j} \tag{2.22}$$

Figure 2.5 present the PPBML for $GF(2^3)$. B-Module is shown in Figure 2.6. multiplication by α is shown in Figure 2.1.

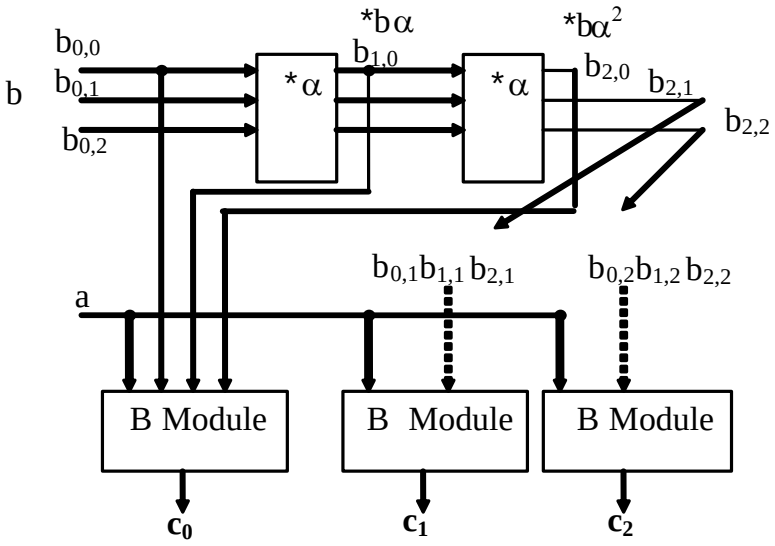

Figure 2.5: Architecture of PPBML for $GF(2^3)$.

.

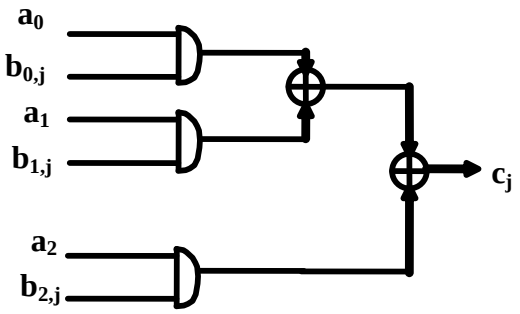

Figure 2.6: Module B of the PPBML

2.2 Cyclic Codes

Reed Solomon code is considered non-binary class of cyclic code, so first we will give a brief introduction to cyclic code.

2.2.1 Description

Cyclic codes are considered class of linear block codes, with the advantage of being easily implemented using sequential logic or shift registers [1].

Let C be a codeword where $C = (c_0, c_1, ..., c_{n-1})$. The i^{th} shifted version of this codeword is:

$$C^{(i)} = (c_{n-i}, c_{n-i+1}, ..., c_{n-1}, c_0, c_1, ..., c_{n-i-1}) \tag{2.23}$$

A linear block code is said to be cyclic code if two properties are exist [1]:

- properties of linear block codes is valid.
- The i^{th}cyclic rotation for any valid codeword is also valid codeword.

2.2.2 Codewords in Polynomial Forms

A codeword can be represented in a polynomial form $C(X)$ function of X, with coefficients c_i are defined under $GF(2^m)$ where i is an integer number corresponds to the position of this coefficient

Let $C = (c_0, c_1, ..., c_{n-1})$ be a code vector, the polynomial representation for this vector is $C(X)$ in the form of:

$$C\left(X\right) = c_0 + c_1 X + \cdots + c_{n-1} X^{n-1} \tag{2.24}$$

2.2.3 Generator Polynomial of a Cyclic Code

The i-position right-shift rotation of a code vector C has the following polynomial expression:

$$C^{(i)}(X) = c_{n-i} + c_{n-i+1} X + \cdots + c_{n-i-1} X^{n-1} \tag{2.25}$$

The i-position right-shift rotated polynomial is denoted as $C^{(i)}(X)$ and the original code polynomial $C\left(X\right)$, with relation shown in equations 2.26 and 2.27 :

$$X^i C\left(X\right) \;\; = \;\; q(X)(X^{n+1}) + C^{(i)}(X) \tag{2.26}$$

$$C^{(i)}(X) = X^i C\left(X\right) mod\left(X^{n+1}\right) \tag{2.27}$$

There is a certain polynomial with a minimum degree r [6], where $r = n - k$ among all polynomials which generate cyclic code $C_{cyc}(n, k)$, this polynomial is called generator polynomial with a valid coefficient g_r and the coefficient $g_0 = 1$, so we can form this polynomial as:

$$g(X) = 1 + g_1 X + \cdots g_{r-1} X^{r-1} + X^r \tag{2.28}$$

This polynomial is used in the encoding procedure for a linear cyclic code, as $C_{cyc}(n, k)$ can be introduced as a multiplication between the message polynomial $m(X)$ and the generator polynomial $g(X)$ as shown in equation 2.29, and this operation is sufficient to generate any code polynomial of the code [6].

$$C(X) = m(X) * g(X) \tag{2.29}$$

2.2.4 Generation of Cyclic Codes in Systematic Form

As we expressed in the previous chapter the linear block code in systematic form, we can express the cyclic code in systematic form based on certain steps:

- The message polynomial is of the form:

$$m(X) = m_0 + m_1X + \cdots + m_{k-1}X^{k-1} \tag{2.30}$$

- Multiply X^{n-k} by $m(X)$ to give the following polynomial:

$$X^{n-k}m(X) = m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-1}X^{n-1} \tag{2.31}$$

- Divided the previous expression by the generator polynomial $g(X)$:

$$X^{n-k}m(X) = q(X)g(X) + p(X) \tag{2.32}$$

- Here $p(X)$ is the remainder polynomial of the division of equation 2.32, which has degree $n-k-1$ or less, since the degree of $g(X)$ is $r = n-k$. By reordering equation 2.32, we obtain can get equation 2.33 as we discussed in the previous section that there is no difference between $(+)$ and $(-)$.

$$X^{n-k}m(X) + p(X) = q(X)g(X) \tag{2.33}$$

Where it is seen that the polynomial $X^{n-k}m(X) + p(X)$ is a code polynomial because it is a factor of $g(X)$. In this polynomial, the term $X^{n-k}m(X)$ represents the message polynomial right shifted $n-k$ positions, where $p(X)$ is the remainder polynomial of this division and acts as the redundancy polynomial. This procedure allows the code polynomial to be in systematic form:

$$\begin{aligned} C(X) &= X^{n-k}m(X) + p(X) \\ &= p_0 + p_1X + \cdots + p_{n-k-1}X^{n-k-1} \\ &+ m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-1}X^{n-1} \end{aligned} \tag{2.34}$$

which can be expressed in the code vector form of:

$$C = (p_0, p_1, ..., p_{n-k-1}, m_0, m_1, ..., m_{k-1}) \tag{2.35}$$

2.3 Properties of Reed-Solomon Codes

This section summarizes the introduction to Reed Solomon (RS) codes and their properties.

RS codes are non-binary cyclic codes. A RS code is specified as RS(n, k) with m-bit symbols. RS(n, k) codes on m-bit symbols exist for all n and k for which

$$0 < k < n \leq 2^m - 1 \tag{2.36}$$

where k is the number of data symbols to be encoded, and n is the total number of code symbols after encoding, called codeword. This means that the RS encoder takes k data symbols and adds parity symbols (redundancy) of $(n - k)$ symbols to make an n symbol codeword in systematic form as discussed in the previous section.

For the most conventional RS(n, k) code,

$$(n,\, k) = (2^m - 1, (2^m - 1) - 2t\,) \tag{2.37}$$

where t is the number of symbols that can be corrected with this code, where t can be expressed as

$$t = \lfloor (n - k)/2 \rfloor \tag{2.38}$$

Equation 2.38 clarifies that for the case of RS codes, we need not more than $2t$ parity symbols to correct t symbol errors. For each error, one redundant symbol is used to find the location of the error in the codeword, and another redundant symbol is used to find the value of the error.

Let the number of errors with an unknown location is n_{errors} and the number of errors with known locations (erasures) as $n_{erasures}$, the RS algorithm guarantees [5] to correct a codeword, provided that the following is true

$$2n_{errors} + n_{erasures} \leq 2t \tag{2.39}$$

Expression 2.39 is called simultaneous error-correction and erasure-correction capability. Erasure information can often be supplied by the demodulator in digital communication system. In this thesis we do not deal with erasures, we only consider the error correction.

Keeping the same symbol size m, RS codes may be shortened by making a number of data symbols zero at the encoder, not transmitting them,and then re-inserting them at decoder. For example, the RS(255, 239) code with ($m = 8$) can be shortened to RS$(200, 184)$ with the same $m = 8$. The encoder takes a block of 184 data bytes, then adds 55 zero bytes, creates a RS$(255, 239)$ codeword and transmits only the 184 data bytes and 16 parity bytes.

The main advantage of RS code is that it performs well against burst noise.

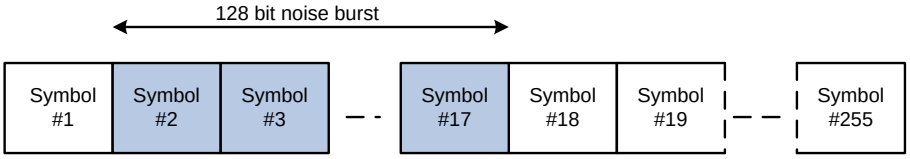

Figure 2.7: A codeword of 255 bytes disturbed by 128-bit noise burst

Consider a popular Reed-Solomon code RS$(255, 223)$, where each symbol is made up of $m = 8$ bits . Since $(n - k) = 32$, Equation 2.38 indicates that this code can correct any 16 symbol errors in a codeword of 255 bytes. Now assume that we have burst error in a 128 bits durations and affected one codeword during transmission, as shown in Figure 2.7

In this example, a burst of noise that lasts for a duration of 128 contiguous bits corrupted exactly 16 symbols. The RS decoder for the (255,223) code will correct any 16 symbol errors regardless the type of damage suffered by the symbol. When the decoder corrects a byte, it replace the incorrect byte by the correct one, whether the error was caused by one bit being corrupted or all eight bits being corrupted. Thus if a symbol is wrong, it might as well be wrong in all its bit positions. That is why RS codes are extremely used. because of their capacity to correct burst errors.

2.4 Applications of Reed-Solomon Codes

Due to the feature of burst error, RS codes provide powerful correction and high rates with high channel efficiency, and thus have a wide range of applications in digital communications and storage,e.g.:

- Storage devices: Compact Disk (CD),DVD,etc;
- Wireless or mobile communications:cellular phones,microwave links,etc;
- Satellite communications;
- Digital television /DVB;
- High-speed modems: ADSL,VDSL,etc.

2.5 Reed-Solomon Encoding

The main idea of RS encoding [2] is to convert the message to codeword by adding parity symbols to the message, where the message length is k symbols, and the codeword length is $n = (2^m - 1)$ symbols, where m is the symbol length.

Each message block is equivalent to a message polynomial of degree $k - 1$, denoted as:

$$m(X) = m_0 + m_1 X + m_2 X^2 + \cdots + m_{k-1} X^{k-1} \tag{2.40}$$

where the coefficients $m_0, m_1, \ldots, m_{k-1}$ of the polynomial $m(X)$ are the symbols of message block. These coefficients are elements of $GF(2^m)$. So the information sequence is mapped into a polynomial by setting the coefficients equal to the symbol values.

For example consider the Galois field $GF(2^8)$, so the information sequence is divided into symbols of eight consecutive bits as shown in Figure(2.8). The first symbol in the sequence is 10000000. In the power representation, 10000000 becomes $\alpha^0 \epsilon GF(2^8)$. Thus, α^0 becomes the coefficient of X^0. The second symbol is 00100000, so the coefficient of X^1 is α^2. The third symbol is 10111111, so the coefficient of X^2 is α^{80} and so on.

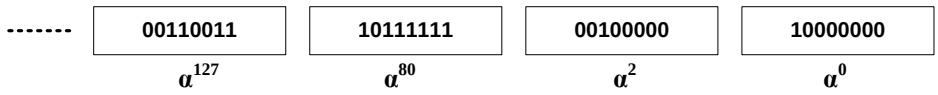

Figure 2.8: The information bit sequence divided into symbols.

The corresponding message polynomial is $m(X) = \alpha^0 + \alpha^2 X + \alpha^{80} X^2 + \cdots$

RS codes are considered class of cyclic codes so the conventional encoding process is done as mentioned before for the cyclic codes by multiplying the message polynomial $m(X)$ by the generator polynomial $g(X)$ to get the codeword polynomial $C(X)$.

2.5.1 Systematic Encoding

The encoding of RS codes can be performed in systematic form. In systematic encoding, the encoded block (codeword) is formed by simply adding parity or

redundant symbols to the end of the k-symbols message block, as shown in Figure 2.9. So codewords are consist of k-symbols message block, and $2t$ parity symbols, where t is the number of error correction capability and $2t = n - k$.

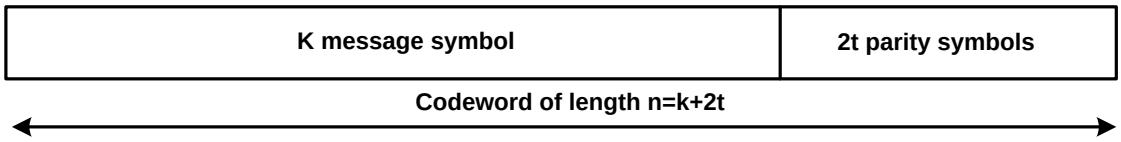

Figure 2.9: A codeword is formed from message and parity symbols.

Applying the polynomial notation, we can shift the information into the left most bits by multiplying by X^{2t}, leaving a codeword of the form

$$C(X) = X^{2t}m(X) + p(X) \tag{2.41}$$

where $C(x)$ is the codeword polynomial, m(X) is message polynomial and $p(x)$ is the redundant polynomial.
The parity symbols are obtained from the redundant polynomial $p(X)$,which is the remainder obtained by dividing $X^{2t}m(X)$ by the generator polynomial, which is expressed as

$$p(X) = (X^{2t}m(X)) mod\, g(x) \tag{2.42}$$

So, RS codeword is generated using generator polynomial, which has such property that all valid codewords are exactly divisible by the generator polynomial. The general form of the generator polynomial is:

$$\begin{aligned} g(X) &= (X+\alpha)(X+\alpha^2)(X+\alpha^3)\cdots(X+\alpha^{2t}) \\ &= g_0 + g_1X + g_2X^2 + \cdots + g_{2t-1}X^{2t-1} + X^{2t} \end{aligned} \tag{2.43}$$

where α is a primitive element in $GF(2^m)$, and g_0, g_1, g_2, $\cdots$, g_{2t-1} are the coefficients from $GF(2^m)$. The degree of the generator polynomial is equal to number of parity symbols $(n-k)$. Since the generator polynomial is of degree $2t$, there must be precisely $2t$ consecutive powers of α that are roots of this polynomial. We designate the root of $g(X)$ as $\alpha, \alpha^2, \ldots, \alpha^{2t}$. It is not necessary to start with the root α, because starting with any power of α is possible. The root of a generator polynomial, $g(X)$, must also be roots of the codeword generated by

$g(X)$, because a valid codeword is of the following form:

$$c(X) = q(X)\, g(X) \tag{2.44}$$

where $q(X)$ is a message-dependent polynomial. Therefore, an arbitrary codeword, when evaluated at any root of $g(X)$, must yield zero, or in other words

$$g(\alpha^i) = c_{valid}(\alpha^i) = 0, \text{ where } i = 1, 2, \ldots, 2t \tag{2.45}$$

2.5.2 Implementation of Encoding

A general circuit for parity calculation in encoder for RS code is shown in Figure2.10

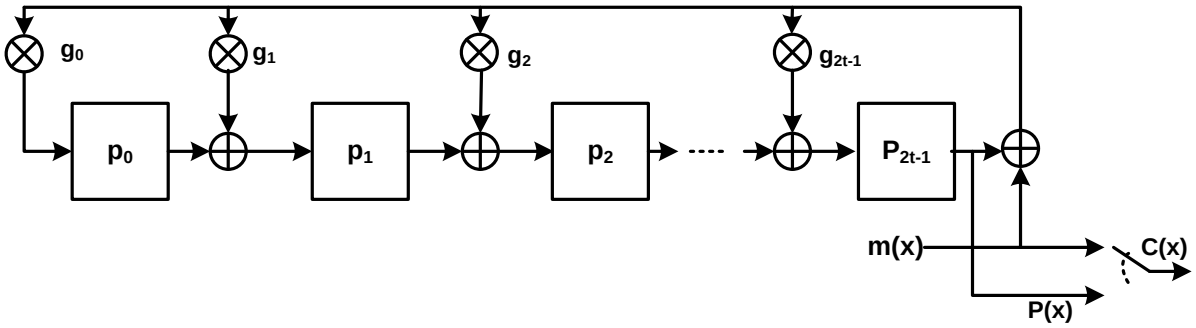

Figure 2.10: LFSR encoder for a RS code

It is a Linear Feedback Shift Register (LFSR) circuit [6], or sometimes called, division circuit.

After the information is completely sifted into the LSFR input,the contents of the register form the parity symbols.

Chapter 3

REED SOLOMON DECODER RS(255,239)

The received codeword is entered to RS decoder to be decoded, the decoder first tries to check if this codeword is a valid codeword or not. If it dose not, errors occurred during transmission. This part of the decoder processing is called error detection. If errors are detected, the decoder try to correct this error using error correction part.

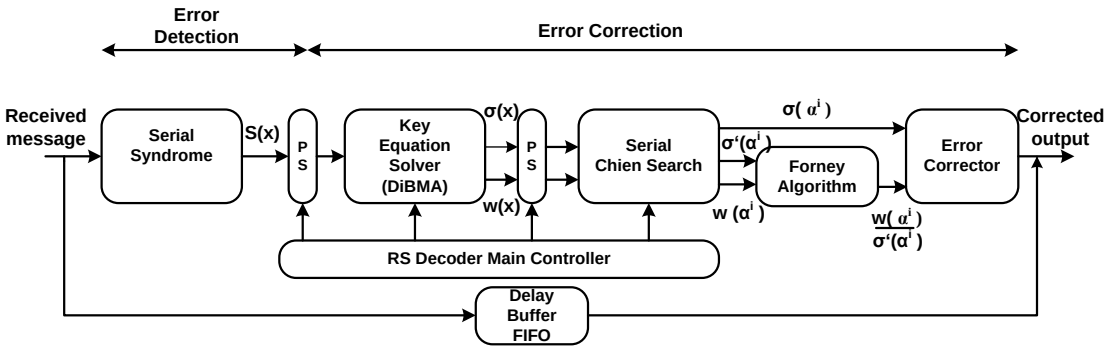

Figure 3.1: Block Diagram of RS Decoder

Figure 3.1 shows the main block diagram of Reed Solomon decoder which consists of two main parts:

1. Error detection part, in this part we use "Syndrome computation" block.

2. Error correction part, this part consists of three blocks:

- Decoding algorithm which used to find the coefficients of error-location polynomial $\sigma(x)$ and error-evaluator polynomial $W(x)$ it sometimes called "Key equation solver".
- Chien search block which used to find the roots of $\sigma(x)$ which present the inverse of the error locations.
- Forney algorithm block which used to find the values of the errors.

After getting the values and locations of the error, we can correct the received codeword by xoring the received vector with the error vector.

3.1 Error Detection "Syndrome Calculation"

The first step in RS decoder is to check if there is any error in the received codeword or not. This done using Syndrome computation block.

- Let the transmitted codeword polynomial $c(X)$ formed as follow:

$$c(x) = c_0 + c_1X + \cdots + c_{n-1}X^{n-1} \ \text{, where}\, c_i \in GF(2^m) \tag{3.1}$$

- Let the received codeword polynomial $r(X)$ formed as follow:

$$r(X) = r_0 + r_1X + \cdots + r_{n-1}X^{n-1} \ \text{, where}\, r_i \in GF(2^m) \tag{3.2}$$

- Let the error polynomial $e(X)$ which added by the channel formed as:

$$e(X) = e_0 + e_1X + \ldots + e_{n-1}X^{n-1} \ \text{, where}\, e_i \in GF(2^m) \tag{3.3}$$

which is related to the received polynomial $r(X)$ and the transmitted polynomial $c(X)$ as follows:

$$r(X) = c(X) + e(X) \tag{3.4}$$

From equations 2.44, and 3.4, the transmitted polynomial $c(x)$ must be multiple of the generator polynomial $g(X)$, and the received polynomial $r(X)$ is evaluated form the addition between $c(X)$ and $e(X)$. So the roots of $g(X)$ should give zero in the received polynomial if the error polynomial is zero. i.e, no errors occurred.

- Let the syndrome polynomial $S(x)$ formed as:

$$S(x) = \sum_{i=1}^{2t} S_i x^{i-1} \tag{3.5}$$

where $i = 1, 2, \ldots, 2t$.

Each coefficients can be described as follows:

$$S_i = r(\alpha^i),\ i = 1, 2, \ldots, 2t \tag{3.6}$$

From equation 3.6. If there is no errors, all syndrome coefficients must give zero. if there is any non-zero coefficient, it means that there is an occurrence for error.

3.2 The Decoding Algorithm

After calculation of the syndrome coefficients we can detect if there exist errors in the received codeword or not by checking these values, if all these coefficients are zeros there will be no errors if not there will be error in an unknown location in the codeword with an unknown value.

The main function of the decoding algorithm is to get the error location polynomial $\sigma(x)$, and the error evaluator polynomial $W(x)$, which represent the locations and the values of the errors respectively.

The first error correction procedure for Reed Solomon codes was found by Gornstien and Zierler [17], and improved by Chien [18] and Forney [19]. This procduer is known as the key equqtion solver, as it will be discussed later.

Decoding algorithms can be categorized into two types:

- Serial algorithms in which the error locator polynomial $\sigma(x)$ is calculated first then we substituted in the key equation to calculate the error evaluator polynomial $W(x)$, e.g (Berlekamp–Massey algorithm [3]).

- Parallel algorithms in which the error locator polynomial $\sigma(x)$ and the error evaluator polynomial $W(x)$ are calculated are in parallel, e.g. (Euclidean algorithm [4]).

3.2.1 Decoding of RS Codes Using Berlekamp-Massey Algorithm

The Berlekamp–Massey (B-M) algorithm [1, 3] is a method used as decoding algorithm used for RS and BCH codes. This method used in RS codes to calculate the coefficients of the error locator polynomial for the error locations, and the coefficients of the error evaluator polynomial for the error values. In BCH the values of the errors are binary so we only calculate the coefficients of the error locator polynomial.

In this section Berlekamp-Massey algorithm will be described for RS codes without any proof and for more details refer to [3].

Let the error polynomial $e(X)$ contains τ errors placed at positions X^{j_1}, X^{j_2}, ... , X^{j_τ} with error values e_{j_1}, e_{j_2}, ..., e_{j_τ} then:

$$e(X) = e_{j_1}X^{j_1} + e_{j_2}X^{j_2} + \cdots + e_{j_\tau}X^{j_\tau} \tag{3.7}$$

Now our target is to calculate the values of e_{ji} and the powers of X^{ji}.

From equation 3.6 we have $2t$ syndrome coefficients. Each syndrome coefficient S_i can be expressed as:

$$s_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i) \tag{3.8}$$

From Equations [3.8 and 3.7] we can obtain set of equations that relate the error locations and values to the syndrome coefficients in the form of:

$$\begin{aligned}
s_1 &= r(\alpha) = e(\alpha) = e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + \cdots + e_{j_\tau}\alpha^{j_\tau} \\
s_2 &= r(\alpha^2) = e(\alpha^2) = e_{j_1}\alpha^{2j_1} + e_{j_2}\alpha^{2j_2} + \cdots + e_{j_\tau}\alpha^{2j_\tau} \\
&\vdots \\
s_{2t} &= r(\alpha^{2t}) = e(\alpha^{2t}) = e_{j_1}\alpha^{2tj_1} + e_{j_2}\alpha^{2tj_2} + \cdots + e_{j_\tau}\alpha^{2tj_\tau}
\end{aligned} \tag{3.9}$$

This set of equations can be simplified in the form:

$$\begin{aligned} s_1 &= r(\alpha) = e(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \cdots + e_{j_\tau}\beta_\tau \\ s_2 &= r(\alpha^2) = e(\alpha^2) = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \cdots + e_{j_\tau}\beta_\tau^2 \\ \vdots &\ \vdots\ \vdots \\ s_{2t} &= r(\alpha^{2t}) = e(\alpha^{2t}) = e_{j_1}\beta_1^{2t} + e_{j_2}\beta_1^{2t} + \cdots + e_{j_\tau}\beta_\tau^{2t} \end{aligned} \tag{3.10}$$

where $\beta_i = \alpha^{j_i}$ and $i = 1, 2, 3, \ldots, \tau$

From equation 3.10 we have $2t$ equations in $2t$ unknowns as worst case, but these equation is not linear equations so we define the two polynomials:

- The error locator polynomial $\sigma(x)$ which present the locations of the error.
- The error evaluator polynomial $W(x)$ which present the values of the errors.

As mentioned before that Berlekamp-Massey algorithm is a serial algorithm so the error location polynomial $\sigma(x)$ is calculated first then the error evaluator polynomial $W(x)$.

a- B–M Iterative Algorithm for Finding the Error-Location Polynomial

Let's assume that we have binary errors, as the values of the errors will not affect the location of the errors, so for B-M algorithm, the error location polynomial can be defined as:

$$\begin{aligned} \sigma_{BM}(X) &= (1 - \beta_1 X)(1 - \beta_2 X)\cdots(1 - \beta_\tau X) \\ &= \sigma_0 + \sigma_1 X + \cdots + \sigma_\tau x^\tau \end{aligned} \tag{3.11}$$

- The roots of this polynomial are $\beta_1^{-1}, \beta_2^{-1}, \ldots, \beta_\tau^{-1}$, the inverse of the error location numbers.
- Coefficients of this polynomial can be expressed as:

$$\begin{aligned}
\sigma_0 &= 1 \\
\sigma_1 &= \beta_1 + \beta_2 + \cdots + \beta_\tau \\
\sigma_2 &= \beta_1\beta_2 + \beta_2\beta_3 + \cdots + \beta_\tau\beta_{\tau-1} \\
&\vdots \\
\sigma_\tau &= \beta_1\beta_2 \ldots \beta_\tau
\end{aligned} \tag{3.12}$$

It is possible to get a relation between the coefficients of $\sigma(X)$ and the syndrome coefficients S_i's :

$$\begin{aligned}
s_1 &+ \sigma_1 = 0 \\
s_2 &+ \sigma_1 s_1 = 0 \\
s_3 &+ \sigma_1 s_2 + \sigma_2 s_1 + \sigma_3 = 0 \\
&\vdots \\
s_\tau &+ \sigma_1 s_{\tau-1} + \cdots + \sigma_{\tau-1} s_2 + \sigma_\tau s_1 = 0
\end{aligned} \tag{3.13}$$

These equations are called Newton identities [6], and we can verify them as follow:

$$\begin{aligned}
s_1 + \sigma_1 &= (\beta_1 + \beta_2 + \cdots + \beta_\tau) + (\beta_1 + \beta_2 + \cdots + \beta_\tau) = 0 \\
s_2 + \sigma_1 s_1 &= (\beta_1)^2 + (\beta_2)^2 + \cdots + (\beta_\tau)^2 + \\
&\quad (\beta_1 + \beta^2 + \cdots + \beta_\tau)(\beta_1 + \beta_2 + \cdots + \beta_\tau) = 0 \\
\vdots \quad &\vdots \quad \vdots
\end{aligned}$$

The remaining Newton identities can be derived in the same way.

The objective from the algorithm is to find the minimum degree polynomial $\sigma(X)$ whose coefficients satisfy these newton identities.

The algorithm proceeds as follows [6]:

1. The first step is to determine a minimum-degree polynomial $\sigma^{(1)}_{BM}(X)$ that satisfies the first Newton identity described in 3.13.

2. The second Newton identity is tested. If the polynomial $\sigma^{(1)}_{BM}(X)$ satisfies the second Newton identity in 3.13, then $\sigma^{(2)}_{BM}(X) = \sigma^{(1)}_{BM}(X)$. Otherwise

the decoding procedure adds a correction term to $\sigma_{BM}^{(1)}(X)$ in order to make the polynomial $\sigma_{BM}^{(2)}(X)$, able to satisfy the first two Newton identities.

3. At the k^{th} step, the polynomial of minimum degree will be:

$$\sigma_{BM}^{(k)}(X) = 1 + \sigma_1^{(k)} X + \sigma_2^{(k)} X^2 + \cdots + \sigma_{l_k}^{(k)} X^{l_k} \tag{3.14}$$

where l_k presents the order of the polynomial, and $1 \leq l_k \leq k$,
which coefficients satisfy the following l_k identities:

$$\begin{aligned} s_1 &+ \sigma_1^{(k)} = 0 \\ s_2 &+ \sigma_1^{(k)} s_1 = 0 \\ s_3 &+ \sigma_1^{(k)} s_2 + \sigma_2^{(k)} s_1 + \sigma_3^{(k)} = 0 \\ &\vdots \\ s_{l_k} &+ \sigma_1^{(k)} s_{l_{k-1}} + \cdots + \sigma_{l_k-1}^{(k)} s_2 + \sigma_{l_k}^{(k)} s_1 = 0 \end{aligned} \tag{3.15}$$

4. In the next step the new polynomial with minimum degree will be:

$$\sigma_{BM}^{(k+1)}(X) = 1 + \sigma_1^{(k+1)} X + \sigma_2^{(k+1)} X^2 + \cdots + \sigma_{l_{k+1}}^{(k+1)} X^{l_{k+1}} \tag{3.16}$$

with coefficients that satisfy the following l_{k+1} identities:

$$\begin{aligned} s_1 &+ \sigma_1^{(k+1)} = 0 \\ s_2 &+ \sigma_1^{(k+1)} s_1 = 0 \\ s_3 &+ \sigma_1^{(k+1)} s_2 + \sigma_2^{(k+1)} s_1 + \sigma_3^{(k+1)} = 0 \\ &\vdots \\ s_{l_{k+1}} &+ \sigma_1^{(k+1)} s_{l_k} + \cdots + \sigma_{l_{k+1}-1}^{(k+1)} s_2 + \sigma_{l_{k+1}}^{(k+1)} s_1 = 0 \end{aligned} \tag{3.17}$$

5. Once the algorithm reaches step $2t$, the polynomial $\sigma_{BM}^{(2t)}(X)$ is called as the error-location polynomial $\sigma_{BM}(X)$, i.e $\sigma_{BM}(X) = \sigma_{BM}^{(2t)}(X)$.

To formalize these steps in a closed iteration form:

Assume that we just completed the k^{th} iteration and got $\sigma^{(k)}(X)$. To find $\sigma^{(k+1)}(X)$, we check whether $\sigma^{(k)}(X)$ satisfy the following Newton identity

$$s_{l_{k+1}} + \sigma_1^{(k)} s_{l_k} + \cdots + \sigma_{l_k-1}^{(k)} s_2 + \sigma_{l_k}^{(k)} s_1 = 0 \tag{3.18}$$

If yes, therefore $\sigma^{(k+1)}(X) = \sigma^{(k)}(X)$ and there will not be any change in the polynomial. If no, we add correction d_μ, called the k^{th} discrepancy. This term can be obtained by using the following expression:

$$d_k = s_{k+1} + \sigma_1^{(k)} s_k + \sigma_2^{(k)} s_{k-1} + \cdots + \sigma_{l_k}^{(k)} s_{k+1-l_k} \tag{3.19}$$

- If $d_k = 0$, then the minimum-degree polynomial $\sigma_{BM}^{(k)}(X)$ satisfies $(k+1)^{th}$ Newton identity, and it becomes $\sigma_{BM}^{(k+1)}(X)$:

$$\sigma_{BM}^{(k+1)}(X) = \sigma_{BM}^{(k)}(X) \tag{3.20}$$

- If $d_\mu = 0$, then the minimum-degree polynomial $\sigma_{BM}^{(k)}(X)$ will not satisfy the $(\mu+1)^{th}$ Newton identity, and a correction term is calculated to be added to $\sigma_{BM}^{(k)}(X)$, in order to form $\sigma_{BM}^{(k+1)}(X)$ as shown in the following equation:

$$\sigma_{BM}^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d_\mu d_\rho^{-1} X^{(\mu-\rho)} \sigma^{(\rho)}(X) \tag{3.21}$$

where $\sigma_{BM}^{(\rho)}(X)$ is a previous polynomial such that the discrepancy $d_\rho = 0$ and $\rho - l$ is a maximum, and the number l_ρ is the degree of the polynomial $\sigma_{BM}^{(\rho)}(X)$.

So the closed form of the algorithm will be:

$$\text{If } d_k = 0 \text{ then } \sigma_{BM}^{(k+1)}(X) = \sigma_{BM}^{(k)}(X),\ l_{k+1} = l_k.$$

If $d_\mu \neq 0$, the algorithm take the previous row ρ, such that $d_\rho \neq 0$ and $\rho - l_\rho$ is maximum.

Then

$$\begin{aligned} \sigma_{BM}^{(k+1)}(X) &= \sigma_{BM}^{(k)}(X) + d_k d_\rho^{-1} X^{(k-\rho)} \sigma^{(\rho)}(X), \\ l_{k+1} &= \max(l, l_\rho + k - \rho), \\ d_{k+1} &= s_{k+2} + \sigma_1^{(k+1)} s_{k+1} + \sigma_2^{(k+1)} s_k + \cdots + \sigma_{l_{k+1}}^{(k+1)} s_{k+2-l_k} \end{aligned} \tag{3.22}$$

The B–M algorithm can be implemented in the form of a table with $2t$ rows to give the final value of the minimum degree error locator polynomial $\sigma_{BM}^{(2t)}(X)$, as given in Table 3.1.

Table 3.1: B–M algorithm table for determining the error-location polynomial

k	$\sigma^{(k)}(X)$	d_k	l_k	$k - l_k$
-1	1	1	0	-1
0	1	S_1	0	0
1	$1 + S_1X$	$\vdots$	$\vdots$	$\vdots$
2	$\vdots$			
$\vdots$				
$2t$				

Note that, if the degree of $\sigma_{BM}^{(2t)}(X)$ is larger than t, it means that its roots do not correspond to a real error-location numbers, it means also that the number of errors are more than t errors, which is more than the error-correction capability of the code.

For example: consider $(15, 9)$ RS code under $GF(2^4)$ with the following syndrome coefficients:

$$\begin{aligned} S_1 &= \alpha^{12} \\ S_2 &= 1 \\ S_3 &= \alpha^{14} \\ S_4 &= \alpha^{10} \\ S_5 &= 0 \\ S_6 &= \alpha^{12} \end{aligned}$$

By applying Berlekamp-Massey algorithm on it to calculate the minimum degree error locator polynomial $\sigma(X)$. The following table 3.2, clarify the steps of the algorithm:

Table 3.2: B–M algorithm table for determining the error-location polynomial for $(15, 9)$ RS Code.

k	$\sigma^{(k)}(X)$	d_k	l_k	$k - l_k$
-1	1	1	0	-1
0	1	α^{12}	0	0
1	$1 + \alpha^{12}X$	α^{7}	1	0
2	$1 + \alpha^{3}X$	1	1	1
3	$1 + \alpha^{3}X + \alpha^{3}X^2$	α^{7}	2	1
4	$1 + \alpha^{4}X + \alpha^{12}X^2$	α^{10}	2	2
5	$1 + \alpha^{7}X + \alpha^{4}X^2 + \alpha^{6}X^3$	0	3	2
6	$1 + \alpha^{7}X + \alpha^{4}X^2 + \alpha^{6}X^3$	$-$	$-$	$-$

From Table 3.2, the minimum degree error locator polynomial $\sigma(X)$ using Berlekamp-Massey algorithm is : $\sigma(X) = 1 + \alpha^{7}X + \alpha^{4}X^2 + \alpha^{6}X^3$.

After the determination of the error-location polynomial, the roots of this polynomial are calculated by applying the Chien search, which will be explained in the following sections, by replacing the variable X with all the elements of the Galois field $GF(2^m)$, 1, α, α^2, ... , α^{2^m-2}, in the expression of the obtained error-location polynomial, looking for the condition $\sigma_{BM}(\alpha^i) = 0$, which present the inverse of the error locations.

b- B–M Algorithm for Finding the Error-Evaluation Polynomial

RS codes are non-binary codes, this means that for a given error location we have error value. This value is under $GF(2^m)$, which add to the algorithm another step to get the error evaluator polynomial $W(X)$.

As mentioned before that B-M algorithm is a serial algorithm so, once the B–M algorithm determines the error-location polynomial $\sigma(X)$, it substitute in the following equation:

$$\sigma(X)S(X) = W(X) + \mu(X)X^{2t} \tag{3.23}$$

This equation is called the Key equation [6], where $\mu(X)$ is a polynomial such that the polynomials $\sigma(X)$, $S(X)$ and $W(X)$ fit the key equation.

Equation 3.23 can be proofed as follow [6]:

- Let syndrome polynomial $S(X)$ defined as:

$$S(X) = \sum_{j=0}^{2t-1} s_{j+1} X^j = \sum_{j=0}^{2t-1} (\sum_{i=1}^{\tau} e_{ji} \alpha^{ji(j+1)}) X^j = \sum_{i=1}^{\tau} e_{ji} \alpha^{ji} \sum_{j=0}^{2t-1} (\alpha^{ji} X)^j$$

$$S(X) = \sum_{i=1}^{\tau} e_{ji} \alpha^{ji} \frac{(\alpha^{ji} X)^{2t} - 1}{(\alpha^{ji} X) - 1} = \sum_{i=1}^{\tau} e_{ji} \frac{(\alpha^{ji} X)^{2t} - 1}{X - \alpha^{-ji}} \tag{3.24}$$

- then from equation 3.24, the result of $\sigma(X)S(X)$ can be shown as:

$$\begin{aligned}
\sigma(X)S(X) &= \sum_{i=1}^{\tau} e_{ji} \sigma_{BM}^{(2t)}(X) \frac{(\alpha^{ji} X)^{2t} - 1}{X - \alpha^{-ji}} \prod_{l=1}^{\tau} (X - \alpha^{-jl}) \\
&= \sum_{i=1}^{\tau} e_{ji} \left[(\alpha^{ji} X)^{2t} - 1 \right] \prod_{i=1, i \neq l}^{\tau} (X - \alpha^{-jl}) \\
&= \sum_{i=1}^{\tau} e_{ji} \prod_{i=1, i \neq l}^{\tau} (X - \alpha^{-jl}) \\
&+ \left[\sum_{i=1}^{\tau} e_{ji} \alpha^{ji(2t)} \prod_{i=1, i \neq l}^{\tau} (X - \alpha^{-jl}) \right] \\
&= W(X) + \mu(X) X^{2t}
\end{aligned}$$

- Also the key equation can be written like that

$$S(X)\sigma(X) = W(X) \bmod X^{2t} \tag{3.25}$$

By substituting in the Key equation shown in equation 3.25, we can get the coefficients of the error evaluator polynomial $W(X)$, so we B-M algorithm called serial architecture as the error locator polynomial is calculated first the the error evaluator polynomial.

After above explanation, it is clear from equation 3.21, the evaluation of $\sigma_{BM}^{(\mu+1)}$needs the inverse of $d_\rho(d_\rho^{-1})$ at each iteration which needs GF inverter. There is two methods to implement the GF inverter. One of them by designing actual GF inverter to get the inverse of GF elements. The other method by using inverse ROM to calculate the inverse of each element. But using the GF inverter at each iteration will consume extra delay in the calculation of equation3.21 and also extra hardware which increases the complexity of the decoder either we used first or second method. So to overcome this drawback of B-M algorithm we will use Decomposed inversion-less Berlekamp-Massey (DiB-M) algorithm [20] which will be discussed in the next chapter .

3.2.2 Decoding of RS Codes Using Euclidean Algorithm

We have seen that the Berlekamp-Massey algorithm is a serial algorithm as it can get the error locator polynomial first then substitute in the key equation to get the error evaluator polynomial. In this section, we show that the Euclidean algorithm can be used to construct error-location polynomial and error evaluation polynomial simultaneously, so we called it by parallel algorithm.

The Euclidean algorithm [21] is a recursive technique to find the Greatest Common Divisor (GCD) between two polynomials, this section will give a brief explanation for this algorithm, and for more details see [6], and [21].

Let $a(X)$ and $b(X)$ are two polynomials over $GF(2^m)$, and assume that

$$\deg a(X) \geq \deg b(X)$$

To get the greatest common divisor (GCD) between the two polynomials $a(X)$ and $b(X)$ can be calculated in an iterative division as follows:

$$\begin{aligned}
a(X) &= q_1(X)b(X) + r_1(X) \\
b(X) &= q_2(X)r_1(X) + r_2(X) \\
r_1(X) &= q_3(X)r_2(X) + r_3(X) \\
&\vdots \\
r_{i-2}(X) &= q_i(X)r_{i-1}(X) + r_i(X) \\
&\vdots \\
r_{n-2}(X) &= q_n(X)r_{n-1}(X) + r_n(X) \\
r_{n-1}(X) &= q_{n+1}(X)r_n(X)
\end{aligned} \tag{3.26}$$

where $q_i(X)$ is the quotient polynomial for the i th division, and $r_i(X)$ is the remainder polynomial for the i th division. The iteration stops when the remainder is zero, then the last nonzero remainder $r_n(X)$ is the GCD of $a(X)$ and $b(X)$.

From equation 3.26, it is possible to say that

$$\text{GCD}\,[a(X),\, b(X)] = f(X)a(X) + g(X)b(X) \tag{3.27}$$

where $f(X)$,and $g(X)$ are polynomials over $GF(2^m)$, so we get a different remainder at each division from the n iterations as follows:

$$\begin{aligned} r_1(X) &= f_1(X)a(X) + g_1(X)b(X) \\ r_2(X) &= f_2(X)a(X) + g_2(X)b(X) \\ &\vdots \\ r_i(X) &= f_i(X)a(X) + g_i(X)b(X) \\ &\vdots \\ r_n(X) &= f_n(X)a(X) + g_n(X)b(X) \end{aligned} \tag{3.28}$$

From equations 3.26, and3.28 we can get the recursive equations for $r_i(X)$, $f_i(X)$, and $g_i(X)$ as follows:

$$\begin{aligned} r_i(X) &= r_{i-2}(X) - q_i(X)r_{i-1}(X) \\ f_i(X) &= f_{i-2}(X) - q_i(X)f_{i-1}(X) \\ g_i(X) &= g_{i-2}(X) - q_i(X)g_{i-1}(X) \end{aligned} \tag{3.29}$$

where $1 \leq i \leq n$.

The initial condition for the recursion equations in equation 3.29 are:

$$\begin{aligned} r_{-1} &= a(X) \\ r_0 &= b(X) \\ f_{-1}(X) &= g_0(X) = 1 \\ f_0(X) &= g_{-1}(X) = 0 \end{aligned} \tag{3.30}$$

To clarify these equations, let's assume that we have two polynomials $a(X) = X^3 + 1$, and $b(X) = X^2 + 1$ over $GF(2)$ and we want to get the GCD$[a(X), b(X)]$ using Euclidean's algorithm.

Table 3.3 clarify the steps of calculation for the GCD between $a(X)$, and $b(x)$. as shown from Table 3.3 the last nonzero remainder is:

$$r_1(X) = X + 1$$

which present the GCD between $X^3 + 1$ and $X^2 + 1$.

Now from the key equation in equation 3.23, this equation can be rearranged

Table 3.3: Calculations of GCD for X^3+1 and X^2+1.

i	$r_i(X)$	$q_i(X)$	$f_i(X)$	$g_i(X)$
-1	X^3+1	$-$	1	0
0	X^2+1	$-$	0	1
1	$X+1$	X	1	X
2	0	$X+1$	$X+1$	X^2+X+1

to be in the following formula:

$$W(X) = Q(X)X^{2t} + \sigma(X)S(X) \tag{3.31}$$

Equation 3.31 can be mapped to equation 3.27, if we set $a(X) = X^{2t}$ and $b(X) = S(X)$.

- Let

$$\begin{aligned} W^{(i)}(X) &= r_i(X) \\ \sigma^{(i)}(X) &= g_i(x) \\ Q^{(i)}(X) &= f_i(X) \end{aligned} \tag{3.32}$$

By applying the Euclidean algorithm get the GCD for X^{2t} and $S(X)$ will give us $W(X)$ which present the error evaluator polynomial and $\sigma(X)$ which present the error locator polynomial. The recursion equations will be as follows:

$$\begin{aligned} W^{(i)} &= W^{(i-2)}(X) - q_i(X)W^{(i-1)}(X) \\ \sigma^{(i)}(X) &= \sigma^{(i-2)}(X) - q_i(X)\sigma^{(i-1)}(X) \\ Q^{(i)}(X) &= Q^{(i-2)}(X) - q_i(X)Q^{(i-1)}(X) \end{aligned} \tag{3.33}$$

with initial conditions:

$$\begin{aligned} W^{(-1)}(X) &= X^{2t} \\ W^{(0)}(X) &= S(X) \\ Q^{(-1)}(X) &= \sigma^{(0)}(X) = 1 \\ Q^{(0)}(X) &= \sigma^{(-1)}(X) = 0 \end{aligned}$$

The iteration stops when the following condition is satisfied:

$$\deg W(X) < \deg \sigma(X) \leq t \tag{3.34}$$

Example: for RS(7, 3) if we received the following syndrome coefficients:

$$\begin{aligned} S_1 &= 0 \\ S_2 &= \alpha^6 \\ S_3 &= \alpha^4 \\ S_4 &= \alpha^4 \end{aligned}$$

we can apply the iteration method of Euclidean algorithm to get $\sigma(x)$ and $W(x)$ in parallel

i	$W^{(i)}(X)$	$Q^{(i)}(X)$	$\sigma(X)$
-1	X^4	1	0
0	$\alpha^4X^3 + \alpha^4X^2 + \alpha^6X$	0	1
1	$\alpha^3X^2 + \alpha^2X$	$\alpha^3X + \alpha^3$	$\alpha^3X + \alpha^3$
2	α^4X	$\alpha X + \alpha^5$	$\alpha^4X^2 + \alpha^3X + \alpha^3X + \alpha^5$

Table 3.4: Euclidean algorithm table for RS(7,3) Example

The Euclidean algorithm is applied using Table 3.4. From the table we can get the Error locator and evaluator polynomials

$$\begin{aligned} \sigma(X) &= \alpha^4X^2 + \alpha^3X + \alpha^5 \\ W(X) &= \alpha^4X \end{aligned}$$

3.2.3 Relationship Between the Error-Location Polynomials of the Euclidean and B–M Algorithms

Error-location polynomials defined in both of these algorithms are practically the same. As an example, for the case of RS codes able to correct error patterns of size $t = 2$ or less, and for the Euclidean algorithm, the error-location polynomial is equal to

$$\begin{aligned}
\sigma(X) &= (X-\alpha^{-j1})(X-\alpha^{-j2}) \\
&= (X+\alpha^{-j1})(X+\alpha^{-j2}) \\
&= (X+\beta_1^{-1})(X+\beta_2^{-1}) \\
&= (1+\beta_1 X)(1+\beta_2 X)/(\beta_1\beta_2) \\
&= \sigma_{BM}(X)/(\beta 1 \beta 2)
\end{aligned}$$

So, for the same error event, both the Euclidean and the B–M error-location polynomials have the same roots, since they differ only by a constant factor. In general,

$$\sigma(X) = \frac{\sigma_{BM}(X)}{\prod_{i=1}^{\tau} \beta i}$$

3.3 Chien Search Calculation

After getting the error locator and evaluator polynomials from the decoding algorithm, we need to find the roots of the error location polynomial $\sigma(X)$, which present the inverse of the error locations. There is no closed form solution for solving the roots of $\sigma(X)$. Since the root has to be one of the elements of the field $GF(2^m)$, so we search for the roots by substituting each of the finite field elements in the error location polynomial $\sigma(X)$ and checking for the following condition:

$$\sigma(\alpha^i) = 0 \tag{3.35}$$

- If this condition is satisfied, an error occurred in the inverse position of i, i.e. in position $(n - i)$.
- If this condition is not satisfied, there is no error.

For example, assume that the error location polynomial is

$$\sigma(X) = 1 + \sigma_1 X + \sigma_2 X^2 + \sigma_3 X^3$$

We evaluate $\sigma(X)$ at each non-zero element in $GF(2^m)$ in sequence:

$$X = \alpha, \quad X = \alpha^2, \quad X = \alpha^3, \ldots, X = \alpha^{2^m - 1}$$

This gives us the following:

$$\begin{aligned} \sigma(\alpha) &= 1 + \sigma_1(\alpha) + \sigma_2(\alpha)^2 + \sigma_3(\alpha)^3 \\ \sigma(\alpha^2) &= 1 + \sigma_1(\alpha^2) + \sigma_2(\alpha^2)^2 + \sigma_3(\alpha^2)^3 \\ &\vdots \quad \vdots \\ \sigma(\alpha^{2^m-1}) &= 1 + \sigma_1(\alpha^{2^m-1}) + \sigma_2(\alpha^{2^m-1})^2 + \sigma_3(\alpha^{2^m-1})^3 \end{aligned} \tag{3.36}$$

After substitution we can evaluate the condition in equation 3.35.

The Chien's search block gets also the value of $W(x)$ at the field elements, i.e, $W(\alpha)$, $W(\alpha^2)$, $W(\alpha^3), \ldots, W(\alpha^{255})$. The only difference is the loaded coefficients, they are $w_0 \sim w_7$ instead of $\sigma_0 \sim \sigma_8$, which is used in calculating the error values.

3.4 Forney Algorithm

The final stage in decoding algorithm is to calculate the value of the errors. To calculate the error value, there are two popular methods, the first one is "transform decoding process" [22] in the frequency domain and the second one is "Forney algorithm" [19] in the time domain. Although the transform decoding process does not need neither FFI nor Chien search, but it requires t variable FFMs and N constant FFMs which are very large area. the Forney algorithm is preferred because of its lower circuit complexity

where the error value at location $\beta_l = \alpha^{j_l} (1 \leqslant l \leqslant k)$ for a RS code is computed by the following formula:

$$e_{j_l} = \frac{W(\beta_l^{-1})}{\sigma'(\beta_l^{-1})} \tag{3.37}$$

Where $W(X)$ is the error evaluation polynomial, k is the number of errors, and

$\sigma'(X)$ is the first derivative of the error locator polynomial $\sigma(x)$ with respect to X.

Finally, after getting the error locations and error values, we finally can form the error polynomial $e(X)$ and correct the received polynomial $r(X)$ just by adding (with XOR operation) these two polynomials together, as shown in Figure 3.1.

Chapter 4

ARCHITECTURE OF DIBM DECODER

4.1 Syndrome Computation Architectures

The syndrome computation block calculates all the syndromes $S_i (1 \leq i \leq 16)$ by putting the roots of generator polynomial $g(x)$ into the received codeword polynomial $R(x)$.

4.1.1 Serial Architecture

The serial syndrome computation block [22] is implemented by following equation.4.1.

$$\begin{aligned} S_i &= R(\alpha^i) = r_{254}(\alpha^i)^{254} + r_{253}(\alpha^i)^{253} + \cdots \\ &+ r_1(\alpha^i) + r_0 \end{aligned} \tag{4.1}$$

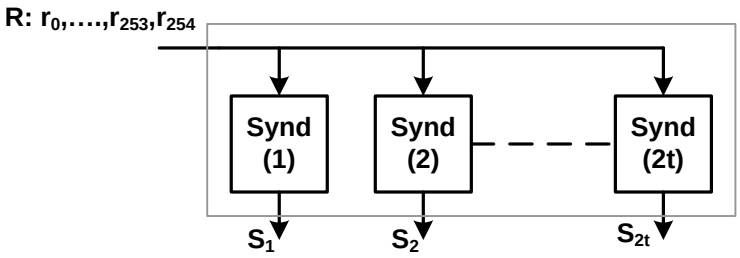

Figure 4.1: Serial Syndrome Block Diagram

The received vector enters serially to the circuit to compute the syndrome coefficients in parallel in n or (255) clock cycles, as shown in Figure 4.1, each cell is shown in Figure 4.2.

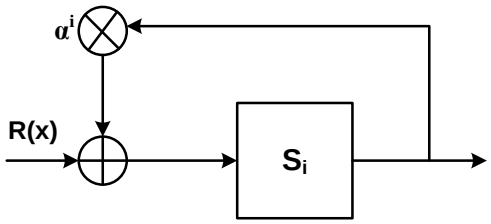

Figure 4.2: Serial Syndrome cell

4.1.2 Two Parallel Architecture

The two-parallel syndrome computation block [23] can be expressed by the equation presented in equation. 4.2 which equivalent to equation.4.1 but in another form.

$$\begin{aligned} S_i &= R(\alpha^i) = (...(((r_{254}(\alpha^i)^2 + r_{253}(\alpha^i) + r_{252})(\alpha^i)^2 \\ &+ (r_{251}(\alpha^i) + r_{250}))(\alpha^i)^2 + (r_{249}(\alpha^i) + r_{248}))(\alpha^i)^2 \\ &\cdots + r_1\alpha^i + r_0 \end{aligned} \tag{4.2}$$

The input patterns of the two-parallel syndrome computation cell is shown in Figure 4.4 . as shown in Figure 4.4, The input B is delayed one clock cycle relative to input A to compute $r_{254}(\alpha^i)^2 + r_{253}\alpha^i + r_{252}$ at the same clock cycle, at the first clock, r_{254} is stored in FF and then $r_{254}(\alpha^i)^2 + r_{253}\alpha^i + r_{252}$ is computed at the next clock cycle.

The input r_{254} is the first input symbol of the new received codeword at the 128^{th}clock, after the $n/2$ or 128 clock syndromes S_i is totally computed. As shown in Figure 4.3. Each cell is shown in Figure 4.4.

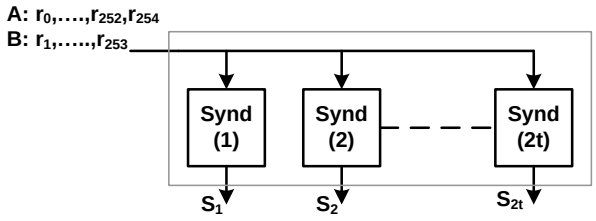

Figure 4.3: Syndrome Block Diagram

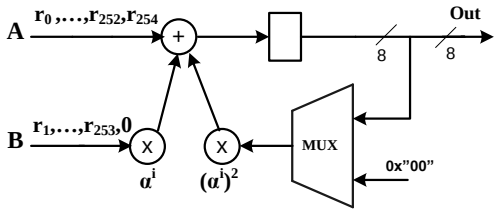

Figure 4.4: Two Parallel Syndrome Cell

4.2 Decoding of RS Codes Using DiBM Algorithm

A decomposed inversion-less B–M algorithm [20] is generally used to compute the error locator polynomial $\sigma(X)$ and error evaluator polynomial $W(X)$.

4.2.1 Computation of Error Locator Polynomial $\sigma(X)$

The Error Locator Polynomial $\sigma(x)$ is computed in a $2t$ - step iterative algorithm, as shown in the following:

Initial condition:

$$\begin{aligned} D^{(-1)} &= 0 \\ \delta &= 1 \\ \sigma^{(-1)}(X) &= T^{(-1)}(X) = 1 \\ \Delta^{(0)} &= S_1 \end{aligned}$$

for $(i = 0 \text{ to } 2t-1)$

$$\begin{cases} \sigma^{(i)}(X) & = \delta.\sigma^{(i-1)}(X) + \Delta^{(i)} X T^{(i-1)}(X), \\ \Delta^{(i+1)} & = S_{i+2}\sigma_0^{(i)} + S_{i+1}\sigma^{(i)} + ... + S_{i-t+2}\sigma_t^{(i)} \end{cases} \tag{4.3}$$

if $(\Delta^{(i)} = 0 \text{ or } 2D^{(i-1)} \geq i+1)$ then

$$D^{(i)} = D^{(i-1)},\ T^{(i)}(X) = XT^{(i-1)}(X)$$

else

$$D^{(i)} = i + 1 - D^{(i-1)},\ \delta = \Delta^{(i)},\ T^{(i)}(X) = \sigma^{(i-1)}(X)$$

where $\sigma^{(i)}(X)$ is the ith step error locator polynomial and $\sigma_j^{(i)}$'s are the coefficients of $\sigma^{(i)}(X)$; $\Delta^{(i)}$ is the ith step discrepancy and δ is a previous nonzero discrepancy; $T^{(i)}(X)$ is an auxiliary polynomial and $D^{(i)}$ is an auxiliary degree variable in i^{th} step.

It is clear that the idea of DiB-M is similar to the idea of B-M algorithm but the computation of $\sigma(X)$ does not need any inversion. So using DiB-M overcomes the drawback of B-M algorithm.

4.2.2 Computation of Error Evaluator Polynomial $W(X)$

If $\sigma(x)$ is first obtained, from the key equation and the Newton's identity we could derive $W(x)$ as follows:

$$\begin{aligned} W(x) &= S(x)\sigma(x) \bmod x^{2t} \\ &= (S_1 + S_2 x + ... + S_{2t}x^{2t-1}) \\ &\quad .(\sigma_0 + \sigma_1 x + ... + \sigma_t x^t) \bmod x^{2t} \\ &\equiv W^{(0)} + W^{(1)}x + ... + W^{(t-1)}x^{t-1} \end{aligned} \tag{4.4}$$

$$\begin{aligned} W^{(i)} &= S_{i+1}\sigma_0 + S_i\sigma_1 + ... + S_1\sigma_i, \\ i &= 0,\ 1, \ldots ,\ t-1. \end{aligned} \tag{4.5}$$

Furthermore, it can be seen that the computation of $W^{(i)}$ is very similar to

Table 4.1: Data Dependency Table

Cycle	$\Delta^{(i+1)}$			$\sigma^i(x)$
$i=0$	$\Delta^{(i)}$	$=$	$\Delta_t^{(i)} + S_{i-t+1}\sigma_t^{(i-1)}$	$\sigma_0^{(i)} = \delta\sigma_0^{(i-1)}$
	$\Delta_0^{(i+1)}$	$=$	0	
$i=1$	$\Delta_1^{(i+1)}$	$=$	$S_{i+2}\sigma_i^{(i)}$	$\sigma_1^{(i)} = \delta\sigma_1^{(i-1)} + \Delta^{(i)}T_0^{(i-1)}$
$i=2$	$\Delta 2^{(i+1)}$	$=$	$\Delta_1^{(i+1)} + S_{i+1}\sigma_1^{(i)}$	$\sigma_2^{(i)} = \delta\sigma_2^{(i-1)} + \Delta^{(i)}T_1^{(i-1)}$
$\vdots$		$\vdots$		$\vdots$
$i=t$	$\Delta t^{(i+1)}$	$=$	$\Delta_{t-1}^{(i+1)} + S_{i-t+3}\sigma_{t-1}^{(i)}$	$\sigma_t^{(i)} = \delta\sigma_t^{(i-1)} + \Delta^{(i)}T_{t-1}^{(i-1)}$

$\Delta^{(i)}$ with some minor differences. Therefore, the same hardware used to compute $\sigma(X)$ can be reconfigured to compute $W(X)$ after $\sigma(X)$ is computed. Also DiB-M algorithm gives a chance to reduce the hardware complexity significantly as only 3 finite field multipliers (FFM) can be used to compute $\sigma(X)$ and $W(X)$ [20] and this will be seen in the following section.

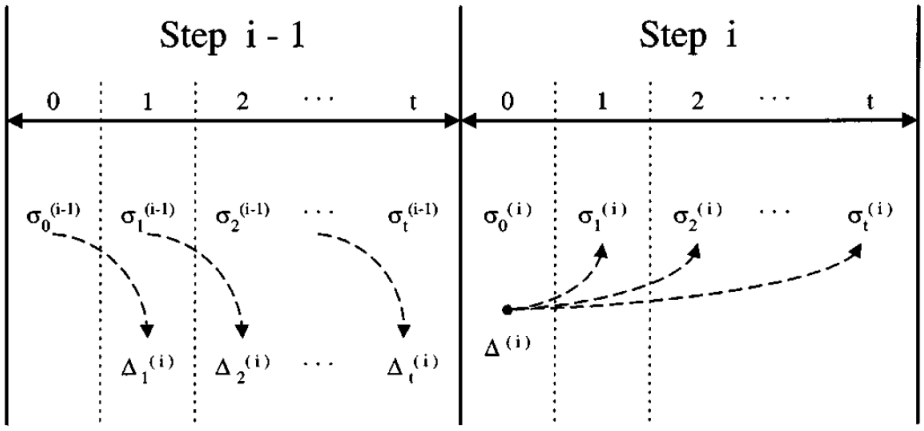

Figure 4.5: Scheduling and data dependency of the decomposed inversionless Berlekamp–Massey algorithm.

4.3 Three-FFM architecture for implementing the DiB-M algorithm

In the RS decoder, an inversionless Berlekamp–Massey algorithm is adopted not only to eliminate the finite-field inverter (FFI) but also introduces additional parallelism. A clever scheduling of three finite-field multipliers [20] was discover to implement the algorithm. To explain the architecture define

$$\sigma_j^{(i)} = \begin{cases} \delta.\sigma_0^{(i-1)}, & \text{for } j = 0 \\ \delta.\sigma_{j-1}^{(i-1)} \quad +\Delta^{(i)} T_{j-1}^{(i-1)}, & \text{for } 1 \le j \le t \end{cases} \tag{4.6}$$

$$\Delta_j^{(i+1)} = \begin{cases} 0, & \text{for} \quad j = 0 \\ \Delta_{j-1}^{(i+1)} + S_{i-j+3}.\sigma_{j-1}^{(i)}, & \text{for} \quad 1 \le j \le t \end{cases} \tag{4.7}$$

where $\sigma^{(i)}(x) = \sigma_0^{(i)} + \sigma_1^{(i)}x + ... + \sigma_t^{(i)}x^t$, $T_j^{(i)}$'s are the coefficients of $T^{(i)}(x)$, and $\Delta_j^{(i+1)}$'s are the partial results in computing $\Delta^{(i+1)}$.

At the first cycle of $(i+1)$th step, we get

$$\begin{aligned} \Delta^{(i+1)} &= \Delta_t^{(i+1)} + s_{i-t+2}\sigma_t^{(i)} \\ &= ... \\ &= S_{i+2}\sigma_0^{(i)} + S_{i+1}\sigma_1^{(i)} + ... + S_{i-t+2}\sigma_t^{(i)} \end{aligned} \tag{4.8}$$

In other words, the ith iteration can be decomposed into $t+2$ cycles. In each cycle $\sigma_j^{(i)}$ requires at most two FFMs and $\Delta_j^{(i+1)}$requires only one FFM. The data dependency of the decomposed algorithm can be seen in Table 4.1. It is evident from Table 4.1 that, at cycle j, the computation of $\Delta_j^{(i+1)}$requires $\sigma_{j-1}^{(i)}$and Δ_{j-1}^{i+1}, which have been computed at cycle $(j-1)$. Similarly, at cycle j, the computation of $\sigma_j^{(i)}$ requires $\Delta^{(i)}$ and $\sigma_j^{(i-1)}$, which have been computed at cycle 0 and the $(i-1)$th step, respectively. Note that the original Berlekamp–Massey algorithm [13] can not be scheduled as efficiently because the computation of equation 3.21 requires two sequential multiplications and one inversion. The inversionless Berlekamp–Massey algorithm [20] provides the necessary parallelism to allow efficient scheduling. The scheduling and data dependency of the decomposed algorithm are further illustrated in Figure 4.5

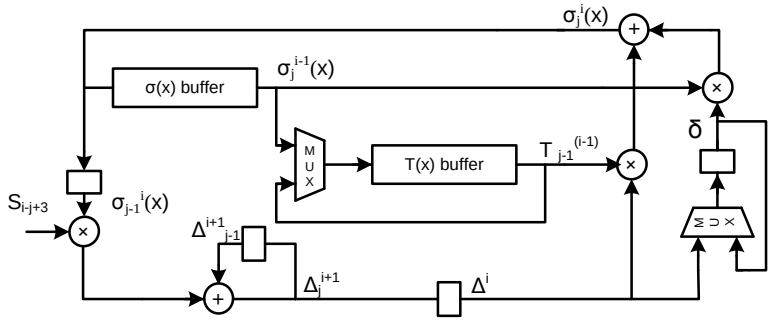

Figure 4.6: Three-FFM architecture to compute $\sigma(X)$ for the decomposed inversionless Berlekamp–Massey algorithm.

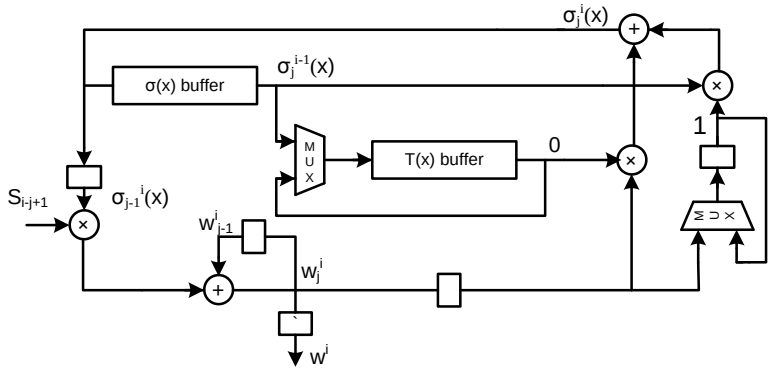

Figure 4.7: Three-FFM architecture reconfigured to compute $W(X)$ for the decomposed inversionless Berlekamp–Massey algorithm.

The decomposed algorithm shown above suggests a three-FFM implementation of the inversionless Berlekamp–Massey algorithm, which is shown in Figure 4.6.

The computation of $W(X)$ can be performed directly after $\sigma(X)$ is computed [20]. Note that the direct computation requires fewer multiplications than the iterative algorithm which computes many unnecessary intermediate results. The penalty of this efficient computation is the additional latency because $\sigma(X)$ and $W(X)$ are computed in sequence.

Furthermore, it can be seen that the computation of W^i is very similar to that of Δ^i except for some minor differences. Therefore, the same hardware used to

compute $\sigma(X)$ can be reconfigured to compute $W(X)$ after $\sigma(X)$ is computed. Like Δ_j^i, W_j^i are the partial results in computing W^i and we could derive it as follows:

$$W_j^i = \begin{cases} s_{i+1}\sigma_0, & for \quad j = 0 \\ W_{j-1}^i + s_{i-j+1}\sigma_j, & for \quad 1 \leqslant j \leqslant i \end{cases} \tag{4.9}$$

At the last cycle of the iteration in equation 4.9 $W_j^i = W_{j-1}^i + s_1\sigma_i = \cdots = W^i$, In Figure 4.7, we show how the same three-FFM architecture can be reconfigured to compute $W(X)$.

4.4 Modified Evaluator DiBM Architecture

4.4.1 Error Locator Polynomial $\sigma(X)$

It will be the same as in the regular DiBM architecture as the three-FFM architecture discussed in the previous section is used for implementing the error locator polynomial $\sigma(X)$.

4.4.2 Error Evaluator Polynomial $W(X)$

The conventional way to compute the error evaluator polynomial $W(x)$ is to do it after the computation of $\sigma(x)$. Using the Berlekamp-Massey algorithm, this involves an iterative algorithm to compute $W^{(i)}(x), i = 0, \ldots, t-1$. However, if $\sigma(x)$ is first obtained, from the key equation and the Newton's identity we could derive $W(x)$ as follows:

$$\begin{aligned} W(x) &= S(x)\sigma(x) \, mod \, x^{2t} \\ &= (S_1 + S_2x + ... + S_{2t}x^{2t-1}) \\ &\quad .(\sigma_0 + \sigma_1x + ... + \sigma_tx^t) \, mod \, x^{2t} \\ &= W^{(0)} + W^{(1)}x + ... + W^{(t-1)}x^{t-1} \qquad (4.10) \\ W^{(i)} &= S_{i+1}\sigma_0 + S_i\sigma_1 + ... + S_1\sigma_i, \qquad (4.11) \\ i &= 0, 1, ..., t-1. \end{aligned}$$

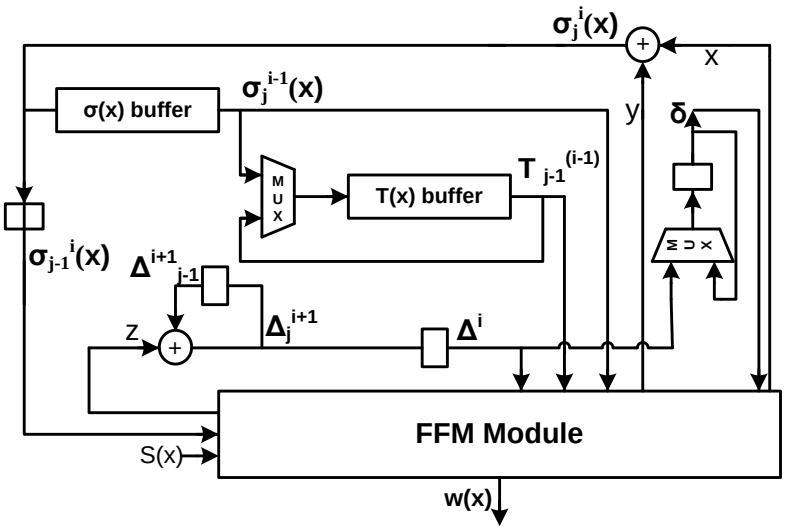

Figure 4.8: Implementation of the serial decomposed inversionless Berlekamp–Massey algorithm

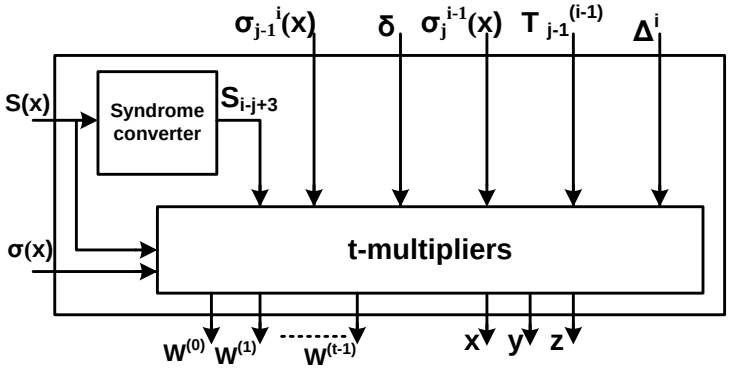

Figure 4.9: Finite Field Multiplier Module Block

That is, the computation of $W(x)$ can be performed directly after $\sigma(x)$ is computed. Note that the direct computation requires fewer multiplications than the iterative algorithm which computes many unnecessary intermediate results, but it needs a lot of FFMs.

The proposed modified evaluator DiBM architecture suggests a t FFM implementation to evaluate $\sigma(x)$ and $W(x)$.

The error evaluator polynomial is evaluated by using 3 FFMs only like in serial architecture [20]. However, the error evaluator polynomial $W(x)$ will need all t

FFMs, as $W^{(t-1)}$ is formed from t terms at worst case.

We reuse these multipliers for each $W^{(i)}$, proposed architecture Compared to the previously parallel architectures [24] our architecture reduces the hardware complexity significantly. Compared to a previously serial architecture [20], our architecture reduces the latency significantly because of the reduction of number of clock cycles. Therefore, our proposed architecture achieves an optimization in the area-delay product.

We can summarize the modified evaluator architecture of MEDiBM architecture in the flow chart shown in Figures 4.10 and 4.11.

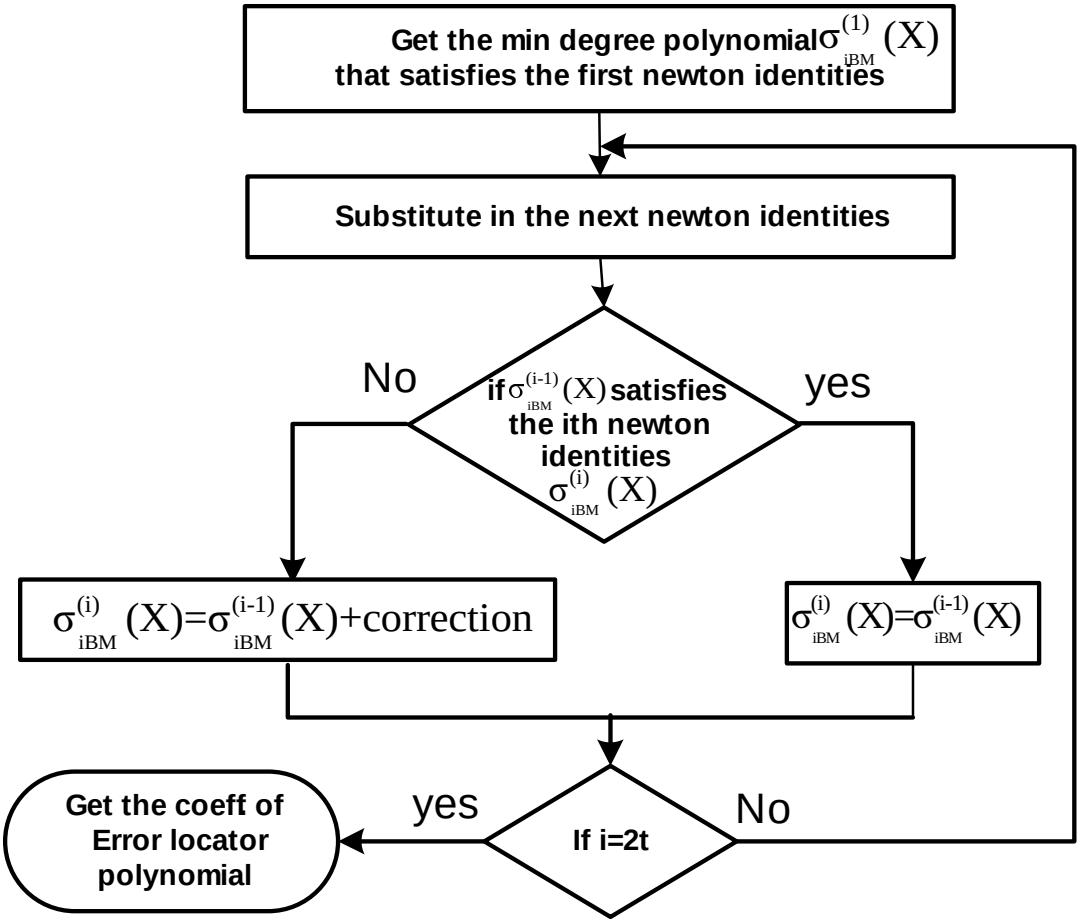

Figure 4.10: Flow chart of MEDiBM architecture to calculate $\sigma(X)$

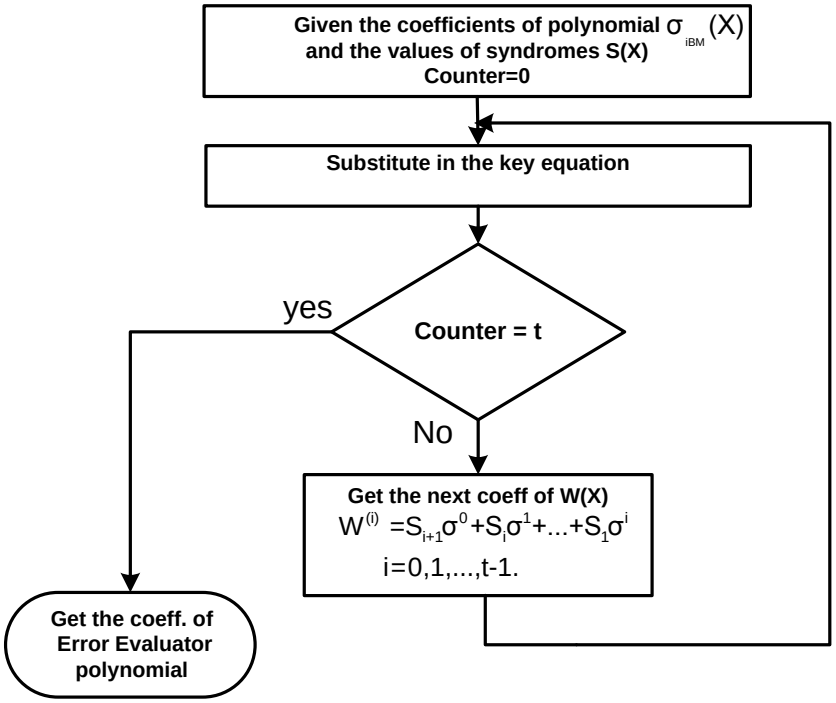

Figure 4.11: Flow chart of MEDiBM architecture to calculate $W(X)$

4.5 Chien Search Architectures

4.5.1 Serial Architecture

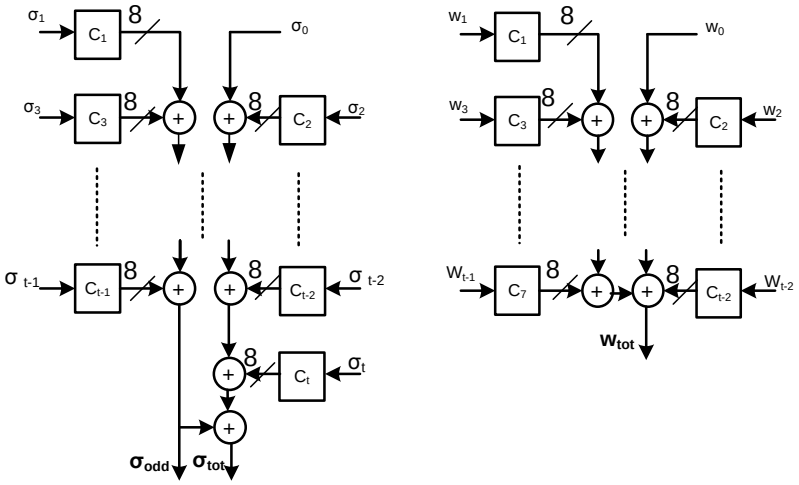

Figure 4.12: Serial Chien Search Block

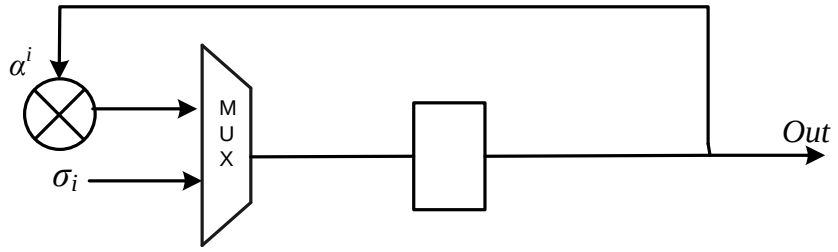

Figure 4.13: Serial Chien Search Cell

The error locator polynomial $\sigma(x)$ and error value polynomial $W(x)$ are obtained by KES block (MEDiBM).

The Chien's algorithm calculates the location of the erroneous symbols in each codeword as discussed in the previous chapter. Figure 4.12 shows the implementation of the Chien search block, each cell in it is clarified in Figure 4.13.

The serial Chien search block computes from $\sigma(\alpha^0)$ to $\sigma(\alpha^{245})$ each one in one clock cycle , after 255 clock cycle the total codeword symbols will be substituted, as shown in Figure 4.12

Note that cells $C_1 \sim C_8$ in Figure 4.12 are all the same as the Chien search cells in Figure 4.15 . The only difference is the loaded coefficients are $w_0 \sim w_7$ instead of $\sigma_0 \sim \sigma_8$.

4.5.2 Two Parallel Architecture

The two-parallel Chien search block computes $\sigma(\alpha^1)$ and $\sigma(\alpha^2)$ simultaneous by the first clock cycle , and then $\sigma(\alpha^{253})$ and $\sigma(\alpha^{254})$ are calculated at 127^{th} clock cycle. At the next clock cycle, $\sigma(\alpha^{255})$ is obtained as shown in Figure 4.15.

Note that cells $C_1 \sim C_8$ in Figure 4.14 are all the same as the Chien search cells in Figure 4.15 . The only difference is the loaded coefficients are $w_0 \sim w_7$ instead of $\sigma_0 \sim \sigma_8$.

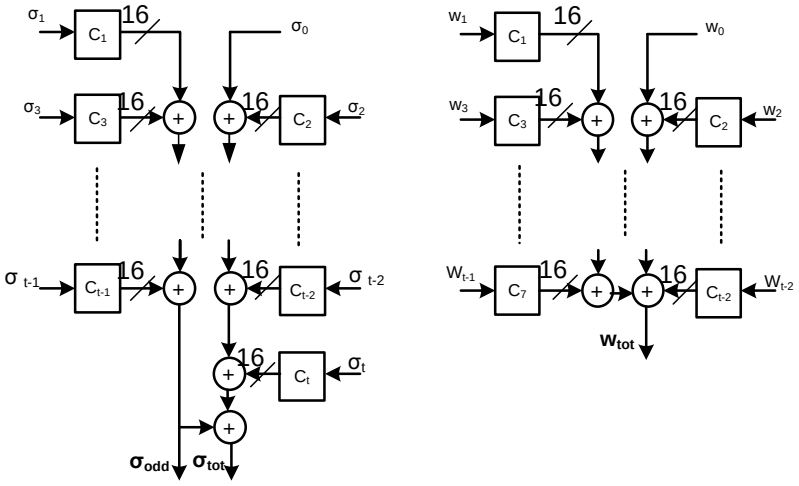

Figure 4.14: Two Parallel Chien Search Block

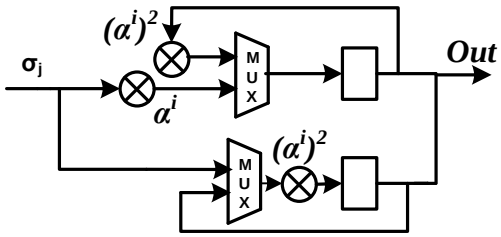

Figure 4.15: Two Parallel Chien Search Cell

4.6 Forney Architectures

Error Value Evaluator used for calculating the error value, there are two popular methods, namely the transform decoding process in the frequency domain and the Forney algorithm in the time domain. Although the transform decoding process does not need any FFI and Chien search, it requires t variable–variable FFMs and N constant–variable FFMs. While N and t are large, the Forney algorithm is preferred because of its lower circuit complexity. An "INVERSE ROM" is implemented as a look-up table to store the inverse field of the Galois field elements since current state of art's reconfigurable devices have resources for look-up tables.

In equation(4.17), dividing operation is implemented by $256 * 8$ ROM in which the inverse of field elements are stored.

$$S_i = r(\alpha^i) = e(\alpha^i) = \sum_{l=1}^{n} e_{ml}\alpha^{ml.i} \tag{4.12}$$

$$S(x) = \sum_{i=0}^{15} S_i X^i = \sum \sum Y_l X_l^i x^i \tag{4.13}$$

$$\sigma(x) = (x - X_1^{-1})(x - X_2^{-1})...(x - X_n^{-1}) \tag{4.14}$$

$$W(x) = S(x).\sigma(x)\, mod\, x^{2t} \quad (t = 8) \tag{4.15}$$

$$= \sum_{l=1}^{n} Y_l \,.\, \prod_{i=1, i\neq l}^{n} (x - X_i^{-1}) \tag{4.16}$$

$$Y_l = W(X_l^{-1})/.\sigma'(X_l^{-1}) \tag{4.17}$$

So having found the error locations and error values, we finally can form the error polynomial $e(X)$ and correct the received polynomial $r(X)$ just by adding (with XOR operation) these two polynomials together, as shown in Figure 3.1.

4.6.1 Serial Architecture

Figure 4.16 shows the serial architecture of the Forney error evaluator which calculate symbol in each clock cycle.

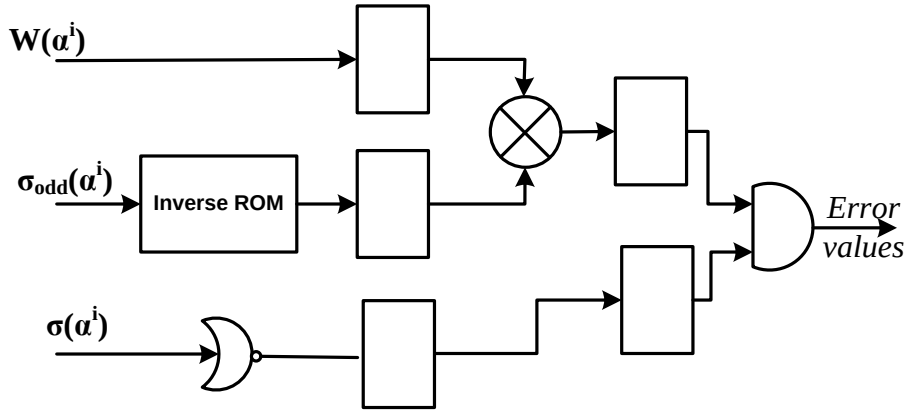

Figure 4.16: Forney Error Evaluator Architecture

4.6.2 Two Parallel Architecture

Figure 4.17 shows the two parallel architecture. which calculate two values of error symbols in each clock cycle.

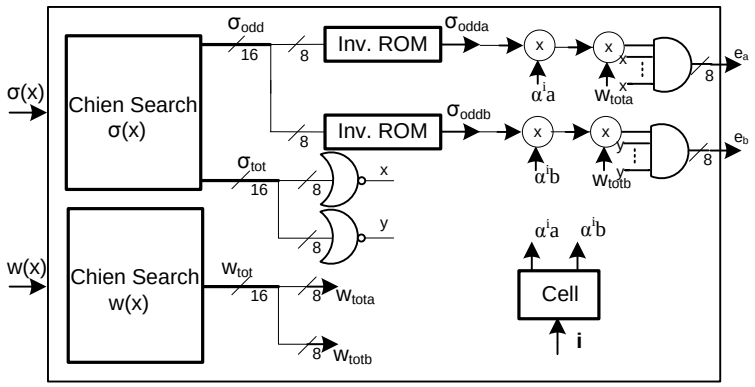

Figure 4.17: Two Parallel Error Corrector Block

4.7 Two Parallel Architecture (Proposal A)

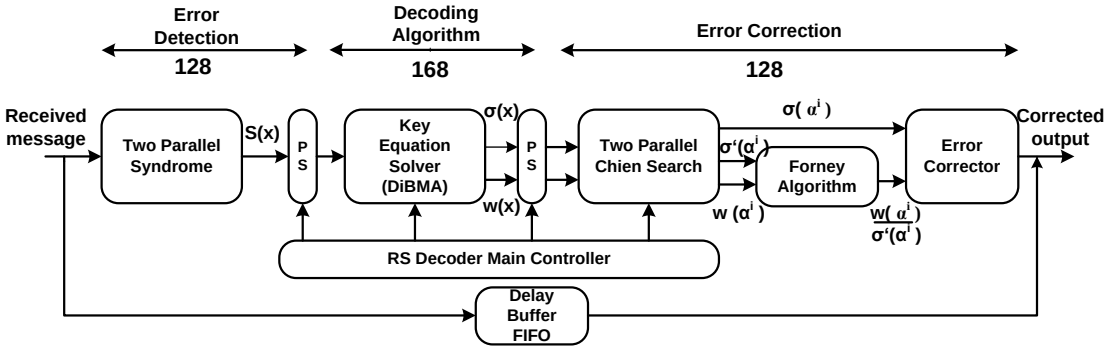

Figure 4.18: Two parallel Block Diagram

In this proposal we present a pipelined two parallel decoder. The decoding process is divided into four steps as shown in Figure 4.18. The syndrome calculator calculates a set of syndromes from the received codewords. The two parallel syndrome circuit [23] is used in this proposal and Figure 4.3 presents the two parallel syndrome circuit. From the syndromes, the key equation solver (KES) produces the error locator polynomial and the error evaluator polynomial as discussed in the previous section, then a Chien search algorithm is used to produce the error locations as shown in Figure 4.14. The two parallel Chien search circuit is used and Figure 4.15 presents the two parallel Chien search cell, then Forney algorithm is used to calculate the error values, Figure 4.17 present the circuit diagram of the complete error corrector.

The bottleneck for this architecture is in the KES as the two parallel syndrome circuit has a latency of 128 clock cycles, and the KES latency is 168 clock cycles which make this proposal gain high throughput and low latency and reasonable area. Figure 4.19 shows the pipelining diagram of this proposal.

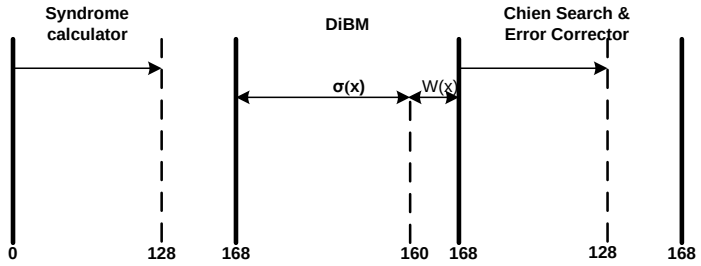

Figure 4.19: The pipelining diagram of (255, 239)RS Two parallel Architecture

4.8 Serial Architecture (Proposal B)

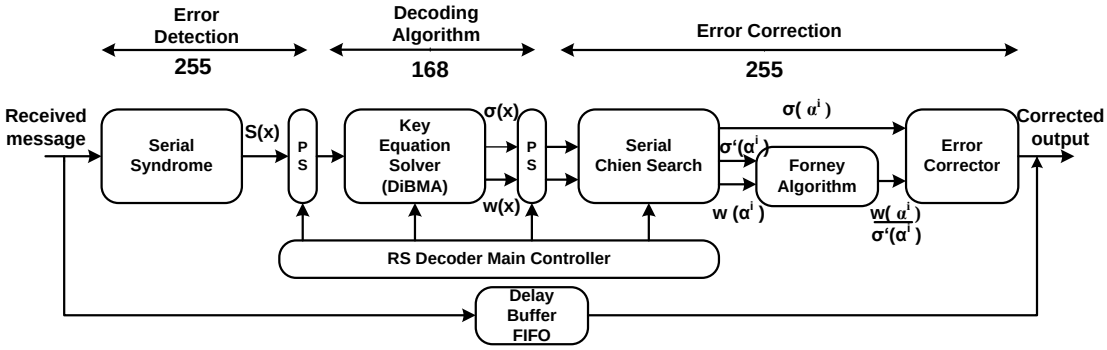

Figure 4.20: Serial Block Diagram

In this proposal we present a pipelined serial decoder. We used the same KES block which is used in proposal A but we changed the syndrome computation block and error corrector block from two parallel blocks to conventional blocks [3]. Figure 4.2 presents the serial syndrome cell. Figure 4.13 presents the serial Chien search cell, then Forney algorithm is used to calculate the error values like the previous proposal.

The bottleneck for this architecture is in the syndrome circuit as it has 255 clock cycles latency which makes the latency of this proposal higher than that of proposal A but it has lower area and reasonable throughput. Figure 4.21 shows the pipelining diagram of this proposal.

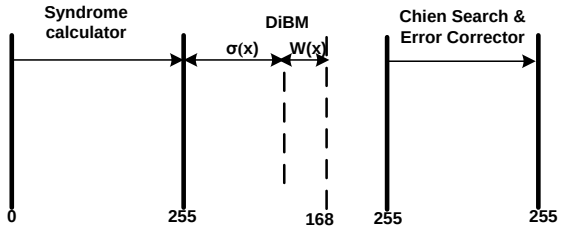

Figure 4.21: The pipelining diagram of (255, 239)RS serial Architecture

Table 4.2: Implementation Results of RS(255,239) Decoders

Architecture	Technology (μm)	No. of Gates	max Freq. F_{max}	Latency (clocks)	Latency $n\,sec$	Throughput Mbps
Proposed A	0.13	37,600	606	298	491.7	7,357
Proposed B	0.13	30,700	606	425	701.3	4,850
ME[24]	0.13	115,500	770	355	461	6,160
pDCME[25]	0.13	53,200	660	355	537.9	5,300
EA[26]	0.13	44,700	300	287	956.7	2,400
ME[27]	0.18	20,614	400	512	1280	3,200
DCME[28]	0.25	42,213	200	288	1440	1,600
EA[29]	0.25	55,240	75	321	4280	600
BM[30]	0.25	32,900	84	192	2285.7	2,500

4.9 Hardware Synthesis

The two architectures were modeled in VHDL and simulated to verify their functionality. After complete verification of the design functionality, it was then synthesized using appropriate time and area constraints. Both simulation and synthesis steps were carried out on $0.13\mu m$ CMOS technology and optimized for a $1.35V$ supply voltage, we used this technology to make our comparison fair with the previously published architectures. The total number of gates for proposal A and B are $37,600$ and $30,700$ respectively from the synthesized results excluding the FIFO memory, and the clock frequency is $606MHz$ for both designs. The total power dissipation $50mW$ for proposal A and $29.28mW$ for proposal B. The latency of proposals A and B are $491.7ns$ and $701.3ns$ respectively which leads to an energy per symbol of $24.585nJ$ for proposal A and $20.534nJ$ for proposal B .

Table 4.2 shows a comparison between different architectures of RS(255, 239) decoders. The Modified Euclidean (ME) algorithm is used in [24], and [27]. The results show that [24] has much larger area than our two proposals as it uses par-

allel architecture in KES. So its latency is lower than proposals A and B and its throughput is in between them. But [27] made a modification to the algorithm to have an area efficient architecture so its area is lower than proposals A and B but with lower rate. Its latency is larger than both proposals. The Degree Computation less Modified Euclidean (DCME) architecture is used in [25] and [28] to reduce the gate count in comparison to ME [24]. However, both [25]and [28] have a larger area compared to our two proposals. The latency and throughput of [30] are in between proposals A and B while the latency of [28] is much larger our two proposals and its throughput is lower than ours. The Euclidean Algorithm is used in [26] and [29]. Both designs use the same architecture but they are implemented on different technologies and they have larger area and latency and lower throughput than the two proposals. The Modified Berlekamp-Massey (MBM) algorithm is used in [30]. This algorithm is used to reduce the latency in clock cycles and also to reduce the power consumption ($68.5mW$ at $84MHz$). But this architecture has higher latency than our two proposals in ns and has more power consumption.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis presents two architectures for a low energy high-speed pipelined serial RS(255, 239) decoder.

In proposal A two parallel syndrome and Chien search circuits are used, where the syndrome and Chien search circuits finish in 128 clock cycles,The KES block is the Modified Evaluator Decomposed inversionless Berlekamp Massey which includes t FFMs with 168 clock cycles latency. This proposal make our design optimized between the serial architecture which uses 3 FFM and latency 216 clock cycles and the parallel architectures which uses multiples of t FFM.

In proposal B a conventional syndrome and Chien search circuits with latency of 255 clock cycles are used and the same KES block of proposal A is used. We have investigated hardware gate count, throughput, and energy per symbol for RS decoders. It is clear that proposal A requires more gate count, but the throughput is higher than proposal B by 34%. But proposal B has energy per symbol lower than proposal A by 16%. Compared to previous architectures, our two proposals optimize the latency, throughput, and energy per symbol while maintaining a small area.

5.2 Future Work

The system is big enough to have a lot of work in the future. Mainly. In fact, we can think about converting the VHDL code to Application Specific Integrated Circuit (ASIC) environment and get the layout and get the post layout simulation.

We can also reduces the latency of the decoding algorithm by increasing the number of FFMs which include the bottleneck of the design to increase the throughput and reduce the latency. we can replace the architecture of Berlekamp Massey algorithm with another architecture for Euclidean algorithm and calculate the area , delay, and power, then compare between the two results.

We can also try to choose another finite field multiplier (the main element in our design) which will increase the maximum frequency and increase the throughput and decrease the latency and the area for the design.

We can integrate the proposed decoder with convolutional decoder to design a concatenated decoder, and we can use the results of the convolutional decoder in the calculations of the Reed Solomon decoder. If we are sure from the convolutional decoder that we have certain number of symbols with error free, we can skip them in the Reed Solomon decoder.

We can convert the proposed decoder from RS$(255, 239)$ to multimode decoder, and this will depend on the type of the channel and the application, as we can control the values of n, and k to increase the rate if we have less noise in the channel or decrease the rate, if we have more noise in the channel

Bibliography

[1] S. Lin, D. J. Castello, Error control coding, Fundamentals and applications, Prentice-Hall, 1983.

[2] W.W. Peterson, E.J. Weldon, Error correcting codes, MIT Press, 1972.

[3] E. R. Berlekamp, Algebraic coding theory, McGraw-Hill, 1968.

[4] I.S. Reed, G. Solomon, "Polynomial codes over certain finite fields", SI AM J. on Applied Mathematics, Vol. 8, June 1960.

[5] R.E. Blahut, "Theory and practice of error-control codes", Addison-Wesley, 1983.

[6] J. C. Moriera, P. G. Frell, J. Wiley "Essentials of Error-Control Coding", 2006.

[7] S. Wicker and M. Bhargava, " Reed-Solomon Codes and Their Applications. IEEE Press", 1994.

[8] B. Schoner, Villasenor, J., Molloy, S., Jain, R., "Techniques for FPGA Implementation of Video Compression Systems", 3rd International ACM symposium on FPGA, 1995.

[9] E. Jamro "The Design of a VHDL based synthesis tool for BCH codecs", master of philosophy, september 1997.

[10] S. T. J. Fenn, M. Benaissa, D. Taylor, "$GF(2^m)$ Multiplication and division over the dual field", IEEE Trans. on computers, Vol. 45, No. 3, March 1996.

[11] S. Choomchuay, "On the Implementation of Finite Field Basis Conversions", Ladkrabang Engineering Journal, Vol. 11, No. 1, June 1994.

[12] J. L. Massey, J.K. Omura, "Computational method and apparatus for finite field arithmetic", U.S. Patent Aplication, No. 4587627, May 1981.

[13] E.R. Berlekamp, "Bit-serial Reed-Solomon encoders", IEEE Trans. Information Theory, Vol. 28, No. 6, November 1982.

[14] T. Beth, D. Gollmann, "Algorithm engineering for public key algorithms", IEEE J. on Selected Areas in Communications. Vol. 7, No. 4, May 1989.

[15] B. A. Laws FR., C. K. Rushforth, "A cellular-array multiplier for GF(2m)", IEEE Trans. Computers, Vol. C-20, No. 12, December 1971.

[16] S. T. J. Fenn, M. Benaissa, D. Taylor, "$GF(2^m)$ Multiplication and division over the dual field", IEEE Trans. on computers, Vol. 45, No. 3, March 1996.

[17] D. Georenstien and N. Zieler, "A Class of Cyclic Linear Error-Correcting codes in Symbols", SI AM J. on Applied Mathematics, June 1961.

[18] R. T. Chien, "Cyclic Decoding Procedure for the Bose-Chaudhuri- Hocquenghem Codes", IEEE Trans. Information Theory, Vol. 10, No. 4, October 1964.

[19] G. D. Forney, "On Decoding BCH Codes", IEEE Trans. Information Theory, Vol. 11, No. 4, October 1965.

[20] H. Chia Chang and C. Shung, "New Serial Architecture for the Berlekamp–Massey Algorithm", IEEE Trans. on communications, Vol. 47, No. 4, April 1999.

[21] Y.Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A method for solving key equation for decoding Goppa codes", Information Control, Vol. 27, No.1, January 1975.

[22] J. L. Massey, "Step-by-step decoding of the Bose–Chaudhuri–Hocq uenghem codes", IEEE Trans. Inf. Theory, Vol. 11, No. 4, October 1965.

[23] S. Lee, C. Choi, and H. Lee, "Two-parallel Reed-Solomon based FEC architecture for optical communications", IEICE Electronics Express, Vol. 5, No. 10, May. 2008.

[24] H. Lee, "High-Speed VLSI Architecture for Parallel Reed-Solomon Decoder," IEEE Trans. on VLSI Systems, Vol. 11, No. 2, April. 2003.

[25] S. Lee, H. Lee, J. Shin and J. Ko, "A High-Speed Pipelined Degree- Computationless Modified Euclidean Algorithm Architecture for Reed- Solomon Decoders", IEEE International symposium on Circuits and System, ISCAS,2007.

[26] H. Lee, "An Area-Efficient Euclidean Algorithm Block for Reed-Solomon Decoder", IEEE Annual Symposium on VLSI, 2003.

[27] H. Yi Hsu, A. Yeu (Andy) Wu, and J. Yeo, "Area-Efficient VLSI Design of Reed–Solomon Decoder for 10GBase-LX4 Optical Communication Systems",IEEE Trans. on Circuits and Systems-II: express briefs, Vol. 53, No. 11, November 2006.

[28] J. H. Baek and M. H. Sunwoo, "New Degree Computationless Modified Euclidean Algorithm and Architecture for Reed-Solomon Decoder", IEEE Trans. on VLSI Systems, Vol. 14, No. 8, Augast, 2006.

[29] H. Lee, M. Yu, and L. Song, "VLSI design of Reed–Solomon decoder architectures", IEEE International Symposium Circuits and System, Vol. 5, 2000.

[30] H. Chang, C. Ching Lin and C. Yi Lee, "A low-power Reed-Solomon decoder for stm-16 optical communications", IEEE Asian-Pasific Conference on ASIC 2002.

Appendix A

The list of optimal irreducible polynomial m $\leq$ 10

$$
\begin{aligned}
m = 3 \qquad & p(x) = \quad x^3 + x + 1 \\
m = 4 \qquad & p(x) = \quad x^4 + x + 1 \\
m = 5 \qquad & p(x) = \quad x^5 + x^2 + 1 \\
m = 6 \qquad & p(x) = \quad x^6 + x + 1 \\
m = 7 \qquad & p(x) = \quad x^7 + x + 1 \\
m = 8 \qquad & p(x) = \quad x^8 + x^4 + x^3 + x^2 + 1 \\
m = 9 \qquad & p(x) = \quad x^9 + x^4 + 1 \\
m = 10 \qquad & p(x) = \quad x^{10} + x^3 + 1
\end{aligned}
$$

Appendix B

Polynomial and dual basis of $GF(2^3)$.

power of α	Standard basis $1, \alpha, \alpha^2$	Dual basis $1, \alpha^2, \alpha$
-	000	000
0	100	100
1	010	001
2	001	010
3	110	101
4	011	011
5	111	111
6	101	110

Appendix C

Basis Conversions

C.1 Dual basis to polynomial basis conversions

Irreducible polynomials $p(x)$ for $GF(2^m)$ are given in Appendix A

-To convert dual basis to polynomial basis:

- for an irreducible trinomial "primitive polynomial with three terms only" of the form: $x^m + x^p + 1$
  $a_0, a_1, a_2, ..., a_{m-1} \leftarrow b_{p-1}, b_{p-2}, ..., b_0, b_{m-1}, b_{m-2}, ..., b_p$

- for an irreducible pentanomial "primitive polynomial with five terms only" of the form: $x^m + x^{p+2} + x^{p+1} + x^p + 1$
  $a_0, a_1, a_2, ..., a_{m-1} \leftarrow b_p, b_{p-1}, ..., b_1, b_0 + b_p, b_{p+1} + b_{m-1}, b_{m-2}, ..., b_{p+1}$
  where a_i present the basis of the polynomial basis symbols and b_i presents the dual basis symbols.

Table C.1: Dual basis coefficients → Polynomial basis coefficients

m	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
3	b_0	b_2	b_1							
4	b_0	b_3	b_2	b_1						
5	b_1	b_0	b_4	b_3	b_2					
6	b_0	b_5	b_4	b_3	b_2	b_1				
7	b_0	b_6	b_5	b_4	b_3	b_2	b_1			
8	b_2	b_1	$b_0 + b_2$	$b_3 + b_7$	b_6	b_5	b_4	b_3		
9	b_3	b_2	b_1	b_0	b_8	b_7	b_6	b_5	b_4	
10	b_2	b_1	b_0	b_9	b_8	b_7	b_6	b_5	b_4	b_3

C.2 Polynomial basis to dual basis conversions

-To convert polynomial basis to dual basis:

- for an irreducible trinomial of the form: $x^m + x^p + 1$
  $b_0, b_1, b_2, ..., b_{m-1} \leftarrow a_{p-1}, a_{p-2}, ..., a_0, a_{m-1}, a_{m-2}, ..., a_p$
- for an irreducible pentanomial of the form: $x^m + x^{p+2} + x^{p+1} + x^p + 1$
  $b_0, b_1, b_2, ..., b_{m-1} \leftarrow a_p, a_{p-1}, ..., a_1, a_0 + a_p, a_{p+1} + a_{m-1}, a_{m-2}, ..., a_{p+1}$
  where a_i present the basis of the polynomial basis symbols and b_i presents the dual basis symbols.

Table C.2: Polynomial basis coefficients $\rightarrow$ Dual basis coefficients

m	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9
3	a_0	a_2	a_1							
4	a_0	a_3	a_2	a_1						
5	a_1	a_0	a_4	a_3	a_2					
6	a_0	a_5	a_4	a_3	a_2	a_1				
7	a_0	a_6	a_5	a_4	a_3	a_2	a_1			
8	$a_0 + a_2$	a_1	a_0	a_7	a_6	a_5	a_4	$a_3 + a_7$		
9	a_3	a_2	a_1	a_0	a_8	a_7	a_6	a_5	a_4	
10	a_2	a_1	a_0	a_9	a_8	a_7	a_6	a_5	a_4	a_3