# ASIC IMPLEMENTATION OF HIGH-THROUGHPUT REED-SOLOMON PRODUCT CODES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL AND ELECTRONICS ENGINEERING

By

Evren Göksu Sezer

July 2021

ASIC Implementation of High-Throughput Reed-Solomon Product Codes
By Evren Göksu Sezer
July 2021

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Erdal Arıkan (Advisor)

Tolga Mete Duman

Ferruh Özbudak

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

# ABSTRACT

## ASIC IMPLEMENTATION OF HIGH-THROUGHPUT REED-SOLOMON PRODUCT CODES

Evren Göksu Sezer
M.S. in ELECTRICAL AND ELECTRONICS ENGINEERING
Advisor: Erdal Arıkan
July 2021

A detailed ASIC implementation study of a decoder architecture for the product of two Reed-Solomon (RS) codes is presented. The implementation aims to achieve high throughput (more than 1 Tb/s) under low power and area consumption constraints while having more than 9 dB coding gain compared to uncoded transmission when concatenated with an inner polar code. The scope of work includes a comprehensive design space exploration for very high rate RS codes. Novel algorithms and architectures are introduced to achieve the design goals. High-throughput is achieved through a combination of pipelining and unrolling methods, while a fully-automated register balancing technique is used to minimize the implementation complexity. The implementation has been carried out using the 28nm TSMC library.

*Keywords:* Reed Solomon Codes, Fiber Optics, High-Throughput, ASIC, FEC.

# ÖZET

# YÜKSEK VERİ HIZLI REED-SOLOMON ÇARPIM KODLARIN ASIC ÜZERİNDE GERÇEKLENMESİ

Evren Göksu Sezer
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Erdal Arıkan
Temmuz 2021

Çarpım Reed-Solomon kod çözücü ve kod çözücünün mimarisinin ASIC üzerinde gerçeklemesine dair detaylı bir çalışma sunulmuştur. Gerçekleme düşük güç harcayan, az alan kaplayan fakat yüksek veri hızına sahip bir kod çözücü tasarlamaktır. Bu çalışma yüksek kodlama oranlı Reed-Solomon kodların detaylı bir incelemesini de kapsamaktadır. Belirlenen hedeflere ulaşmak amacıyla bazı yeni algoritmalar ve mimariler kullanılmıştır. Gerçekleme 28nm TSMC kütüphanesi kullanılarak yapılmıştır.

*Anahtar sözcükler*: Reed-Solomon kodlar, FEC, ASIC, yüksek veri hızı.

# Acknowledgement

I would like to thank Prof. Erdal Arıkan for his invaluable guidance and unending patience. I would like to thank jury members of my thesis defense, Prof. Tolga Mete Duman and Prof. Ferruh Özbudak, for their feedbacks that improved my thesis.

I appreciate the help and support of my colleagues from Polaran; Altuğ Süral, Ayça Osunluk, Yiğit Ertuğrul and in particular, Ertuğrul Kolağasıoğlu.

I would like to thank my family, who helped me anyway they can throughout my education. Also, I thank my in-laws for all their good wishes and mental support during my thesis process.

Lastly, I wholeheartedly thank my wife; who always supports me in all my endeavors and is a helping hand that I can rely on whenever I need her.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Objectives of the Thesis

The goal of this thesis is to investigate the possibility of finding a Reed-Solomon (RS) based product decoding scheme in order to satisfy current demands of fiber optical communication. If necessary, new decoder algorithm and architecture will be developed. The developed RS decoder will be coded using Very High-Speed Integrated Circuit Hardware Description Language (VHDL) and its implementation will be carried out using the Taiwan Semiconductor Manufacturing Company (TSMC) 28nm library synthesized by Genus tool of Cadence. Our performance criteria is as follows:

- Throughput, higher than 1 Tb/s
- Coding gain, higher than 9dB at $10^{-15}$ bit error rate (BER) compared to uncoded transmission
- Power consumption, lower than 20 Watts
- Area consumption, lower than 40 $mm^2$

## 1.2 Work Done

First, we have developed RS(208,204) decoder with the purpose of using it as a component code for the product architecture. It is a rather unconventional design for Reed-Solomon decoders as it has an error correction capability of only two symbols. Such decoder would not be used as a stand-alone decoder due to its low error correction capability. On the other hand, it has a very low complexity. By taking advantage of the low complexity of the RS(208,204), we have developed product RS(208,204) (PRS(208,204)) by using RS(208,204) as building blocks.

The main output of this thesis is PRS(208,204), iteratively working RS(208,204) in product structure. Thanks to the very low implementation complexity of RS(208,204), it is possible to use multiple RS(208,204) decoders in parallel, in series or in this case both. Moreover, this product RS code is designed with the purpose of concatenating it with another forward error correction (FEC) code: polar code. Therefore, given that an efficient concatenation scheme is chosen, it could provide very high coding gain as well as very high throughput. In this thesis, a polar decoder is chosen as an inner decoder and a Reed-Solomon based decoder with product architecture is chosen as an outer decoder. Transmission and reception chain is shown in Figure 1.1. This concatenated decoder puts out 1.040 Tb/s net throughput while achieving 11.5 dB coding gain at $10^{-15}$ BER compared to uncoded transmission [1]. This thesis focuses on the implementation of the outer decoder, namely, the product Reed-Solomon (PRS) decoder.

Implementation of the PRS(208,204) is carried out on ASIC, using the 28 nm technology of TSMC. The design of PRS(208,204) is carried out using the bottom to top methodology. First, multiplier circuits using the Karatsuba Algorithm is designed and implemented. A special methodology is developed to reduce the complexity of the multiplier unit when one of the multipliers is constant. Folded structure is used for syndrome calculation and inversionless Berlekamp-Massey algorithm is used for the calculation of error locator polynomial. In order to solve the error locator polynomial, novel approach is developed where we solve the equation by taking advantage of its low degree. Error locations are calculated

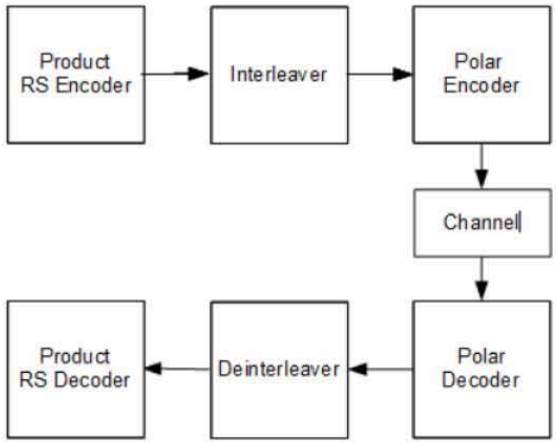


Figure 1.1: Transmit and receiver chain

using the Forney algorithm. Several correction methods are developed to check correctness of decoding process of reached codeword, which is different from the regular RS decoders. Using RS(208,204) as a building block, PRS(208,204) is implemented. PRS(208,204) can provide more than 1 Tb/s throughput while consuming around 6.24 W power and fitting into 13.34 $mm^2$ area.

## 1.3 Organization of the Thesis

This thesis is organized as follows. Chapter 2 presents a review of Reed Solomon codes and explains the mathematics behind it. Then shows the place of Reed-Solomon codes in the standards and literature. Chapter 3 explains the design process of RS(208,204), presents the algorithms, architecture and implementation result of RS(208,204). Chapter 4, shows the design of PRS(208,204) and presents the implementation results. Chapter 5 concludes the thesis with a brief explanation of the results and our comments on them.

# Chapter 2

# Review of Reed-Solomon Codes

Reed-Solomon (RS) codes are class of forward error correcting (FEC) codes that are developed by Irving Reed and Gus Solomon in 1960 [2]. They are subset of the Bose-Chaudhuri-Hocquenghem (BCH) [3] codes. RS codes have various application areas such as storage devices, satellite communications, digital television broadcast, wireless communications, QR codes etc. In Section 2.1, the properties and behavior of RS codes are presented. In Section 2.2, Galois Fields and their properties are explained. In Section 2.3, the encoder of RS codes and in Section 2.4 RS decoder and its historical development are presented. In Section 2.5, some of the standards that are using RS codes and in Section 2.6, literature survey of Reed-Solomon codes with high throughput are presented.

## 2.1 What are Reed-Solomon Codes?

RS codes are linear non-binary cyclic block codes that are mapped on m dimensional vector space. A RS code can be specified as RS(n,k). 'k' shows the number of m-bit data symbols which are to be encoded. These symbols represent the coefficients of a (k-1)th order polynomial in Galois Field ($2^m$) (GF($2^m$)). In order to encode 'k' symbols, they are multiplied with the generator polynomial. 'n'

shows the number of m-bit symbols in an encoded block. Encoded symbols also represent the coefficients of a (n-1)th order polynomial in GF($2^m$).
Given a symbol size m, symbol wise block length 'n' is bounded by the inequality:

$$n \leq 2^m - 1 \tag{2.1}$$

When $n - k$ is even, number of parity symbols are equal to:

$$n - k = 2t \tag{2.2}$$

where t is the symbol wise error correction capability of RS(n,k) code.

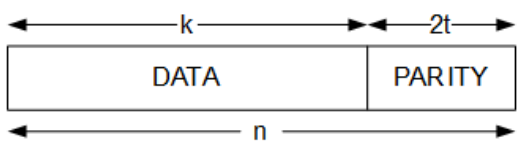


Figure 2.1: Structure of RS code

RS codes are maximum distance separable (MDS) codes and they achieve Singleton bound [4]. RS(n,k) has a minimum distance of $d_{min} = n - k + 1$.

## 2.2 Galois Field

Galois Field (GF) is a field with finite elements whose multiplicative group is cyclic. GF($2^m$) is constructed on a prime polynomial ($p(x)$) of order $m$. Primitive element ($\alpha$) is a value such that each non-zero element of the field can be expressed as the power of $\alpha$. [4] The sequence of the elements in the GF and their binary representations are calculated using the prime polynomials and primitive element. Non-zero elements of GF($2^3$) constructed with the prime polynomial $p(x) = x^3 + x + 1$ are shown in Table 2.1 as a small example. As it can be seen from the Table 2.1, $\alpha^7$ is equal to $\alpha^0$ hence the cyclic group.

Table 2.1: Different representations of non-zero elements of GF($2^m$), constructed with $p(x) = x^3 + x + 1$

| Primitive Element Representation | Polynomial Representation | Binary Representation |
|---|---|---|
| $\alpha^0$ | 1 | 001 |
| $\alpha^1$ | $x$ | 010 |
| $\alpha^2$ | $x^2$ | 100 |
| $\alpha^3$ | $x+1$ | 011 |
| $\alpha^4$ | $x^2+x$ | 110 |
| $\alpha^5$ | $x^2+x+1$ | 111 |
| $\alpha^6$ | $x^2+1$ | 101 |
| $\alpha^7$ | 1 | 001 |

Addition operation can be easily performed in polynomial or binary representation, it is a bit-wise XOR operation. For example:

$$\begin{aligned} \alpha^4 + \alpha^5 &= (x^2 + x) \oplus (x^2 + x + 1) \\ &= 1 \\ &= \alpha^0 \end{aligned} \tag{2.3}$$

Multiplication operation can be easily performed in primitive element representation, summing the powers under the modulo $2^m - 1$ is enough. However, multiplication in polynomial representation can be calculated as polynomial multiplication modulo prime polynomial. For example:

$$\begin{aligned} \alpha^4 \times \alpha^5 &= (x^2 + x) \times (x^2 + x + 1)\ (mod(x^3 + x + 1)) \\ &= x^4 + x^3 + x^2 + x^3 + x^2 + x\ (mod(x^3 + x + 1)) \\ &= x^4 + x\ (mod(x^3 + x + 1)) \\ &= x^2 \\ &= \alpha^2 \end{aligned} \tag{2.4}$$

## 2.3 Encoding

Encoding of RS codes are done by multiplying the data polynomial with the generator polynomial ($g(x)$). The calculation of $g(x)$ is carried out as follows [4] :

$$g(x) = (x - \alpha^{j_0}) \times (x - \alpha^{j_0+1}) \times (x - \alpha^{j_0+2}) \times ... \times (x - \alpha^{j_0+2t-1}) \tag{2.5}$$

Any integer value can be chosen for $j_0$. In order to simplify the circuitry, $j_0$ is usually chosen as '1', which simplifies $g(x)$ to:

$$g(x) = (x - \alpha^1) \times (x - \alpha^2) \times (x - \alpha^3) \times ... \times (x - \alpha^{2t}) \tag{2.6}$$

Using GF($2^3$) with the prime polynomial $p(x) = x^3 + x + 1$, $g(x)$ for RS(7,5) can be calculated as follows:

$$\begin{aligned} g(x) &= (x - \alpha^1) \times (x - \alpha^2) \\ &= x^2 - \alpha^2 x - \alpha x + \alpha^3 \\ &= x^2 + (\alpha^2 + \alpha)x + \alpha + 1 \\ &= x^2 + \alpha^4 x + \alpha^3 \end{aligned} \tag{2.7}$$

Calculated $g(x)$ can be represented in binary as '001 110 011'. Assume input signal, $i(x)$, for RS(7,5) is $i(x) = x^4 + \alpha^2 x^3 + \alpha^6 x^2 + x + \alpha^5$ . Multiplication of the input signal, $i(x)$, and generator polynomial, $g(x)$, results in non-systematic encoding. Non-systematic encoding example is shown in Equation 2.8.

$$
\begin{aligned}
g(x) \times i(x) &= (x^2 + \alpha^4 x + \alpha^3) \times (x^4 + \alpha^2 x^3 + \alpha^6 x^2 + x + \alpha^5) \\
g(x) \times i(x) &= x^6 + \alpha^2 x^5 + \alpha^6 x^4 + x^3 + \alpha^5 x^2 + \alpha^4 x^5 + \alpha^6 x^4 \\
&\quad + \alpha^{10} x^3 + \alpha^4 x^2 + \alpha^9 x + \alpha^3 x^4 + \alpha^5 x^3 \\
&\quad + \alpha^9 x^2 + \alpha^3 x + \alpha^8 \\
g(x) \times i(x) &= x^6 + (\alpha^2 + \alpha^4) x^5 + (\alpha^6 + \alpha^6 + \alpha^3) x^4 \\
&\quad + (1 + \alpha^3 + \alpha^5) x^3 + (\alpha^5 + \alpha^4 + \alpha^2) x^2 \\
&\quad + (\alpha^2 + \alpha^3) x + \alpha \\
g(x) \times i(x) &= x^6 + \alpha x^5 + \alpha^3 x^4 + \alpha^6 x^3 + \alpha^6 x^2 + x + \alpha
\end{aligned}
\tag{2.8}
$$

In binary representation, the resulting non-systematic codeword is represented in seven symbols as '001 010 011 101 101 001 010'. This encoding procedure is not systematic because the input signal $i(x)$ does not appear in the codeword. Systematic encoding is somewhat more complex. Systematic codeword is calculated with Equation 2.9 where $p(x)$ is the polynomial that carries parity symbols [4]. A way of calculating $p(x)$ is given in Equation 2.10. Using the value of the modulo $g(x)$ as parity symbols ensures that, encoded signal is polynomial multiplicative of $g(x)$.

$$
c(x) = x^{n-k} i(x) + p(x) \tag{2.9}
$$

$$
p(x) = x^{n-k} i(x) \; (mod(g(x)) \tag{2.10}
$$

Calculation of $p(x)$ for our example is as follows:

$$
\begin{aligned}
p(x) &= (x^2) \times (x^4 + \alpha^2 x^3 + \alpha^6 x^2 + x + \alpha^5) \; (mod(x^2 + \alpha^4 x + \alpha^3) \\
&= x^6 + \alpha^2 x^5 + \alpha^6 x^4 + x^3 + \alpha^5 x^2 \; (mod(x^2 + \alpha^4 x + \alpha^3) \\
&= \alpha^2 x + \alpha^4
\end{aligned}
\tag{2.11}
$$

Systematic codeword $c(x) = x^6 + \alpha^2 x^5 + \alpha^6 x^4 + x^3 + \alpha^5 x^2 + \alpha^2 x + \alpha^4$ contains the sequence from $i(x)$. The codeword can be represented in binary form as '001 100 101 001 111 100 110'.

Encoders are implemented using Linear Feedback Shift Registers (LFSR) [5]. The design of the LFSR for the previously calculated example $g(x)$ is given in Figure 2.2. $g_0$ represents the coefficient of the 0th order term of $g(x)$ and $g_1$ represents the coefficient of the first order term. If $g(x)$ has the $r$th order, $r-1$ number of Galois multipliers and adders are needed to implement the circuitry. From 0th coefficient to $(r-1)$th coefficient should be inputs of the multipliers in the increasing order from left to right. Input symbols in descending order should be fed into the circuitry one symbol at a clock period. After $k$th clock period, values registered in the registers are the parity symbols. Since our example is RS(7,5), encoding takes 5 clock cycles for the example.

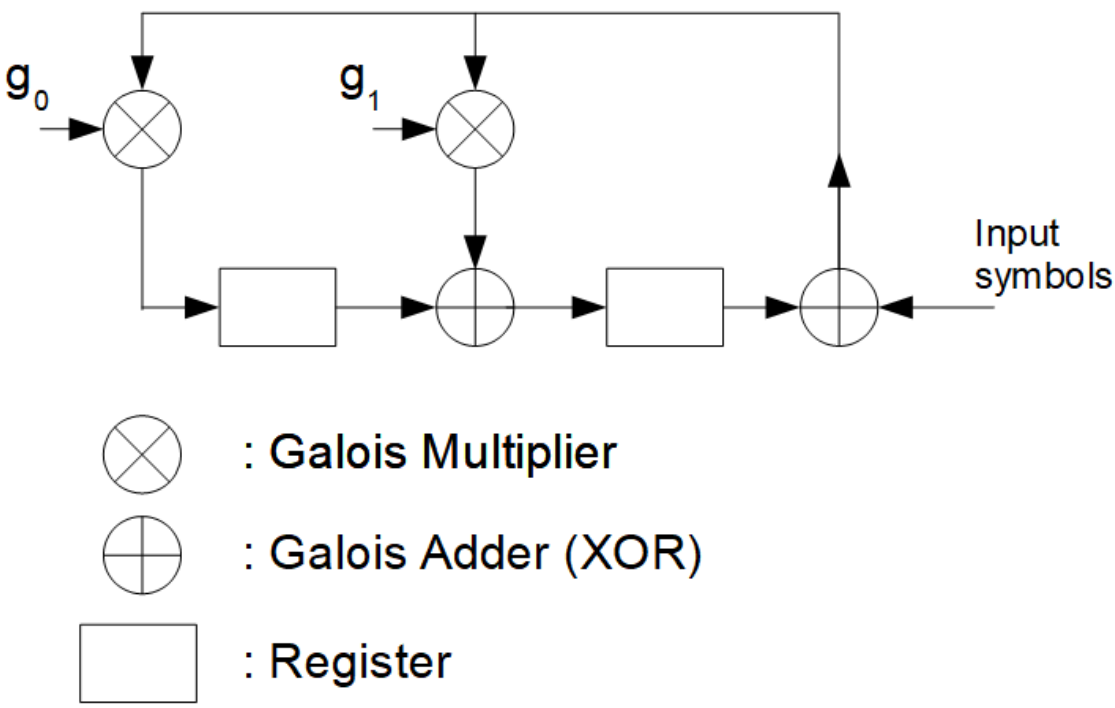


Figure 2.2: LFSR encoder example

The state of the registers and signals are given in Table 2.2. The state of the registers after 5th clock cycle matches the parity symbols calculated using

Equation 2.10.

Table 2.2: The state of the registers and signal of LFSR example

| **Input** | **MS Register** | **Feedback** | **Registers** | | **Clock Cycle** |
|---|---|---|---|---|---|
| 1 | 0 | 1 | $\alpha^3$ | $\alpha^4$ | 1 |
| $\alpha^2$ | $\alpha^4$ | $\alpha$ | $\alpha^4$ | $\alpha^2$ | 2 |
| $\alpha^6$ | $\alpha^2$ | 1 | $\alpha^3$ | 0 | 3 |
| 1 | 0 | 1 | $\alpha^3$ | $\alpha^6$ | 4 |
| $\alpha^5$ | $\alpha^6$ | $\alpha$ | $\alpha^4$ | $\alpha^2$ | 5 |

## 2.4 Decoding

RS decoder decodes the received signal, $r(x)$, in five steps, shown in Figure 2.3. These steps are syndrome calculation, finding error polynomial, finding the location of the errors, finding the magnitude of the errors and correcting the errors to reach a valid codeword, $c(x)$. The following sections explain the purpose of each step, mathematical calculations, algorithms and architectures for their implementations.

### 2.4.1 Syndrome Calculator

The main purpose of the syndrome calculator is to check if the received signal, $r(x)$, is a valid codeword, in another words, whether it has errors or not. If it is not a valid codeword and some transmission errors are present in $r(x)$, the syndrome calculator catches that there is at least one erroneous symbol which needs to be corrected. If the syndrome calculator shows that there is no error, this indicates that $r(x)$ is a valid codeword and output of the decoder.

Whether a signal encoded with non-systematic or systematic encoder, the encoded signal yields zero at the roots of the generator polynomial, $g(x)$. If $r(x)$ is a valid member of the codeword set, syndrome values yield zero. If all the syndrome values are not equal to zero, it shows the presence of one or more errors.

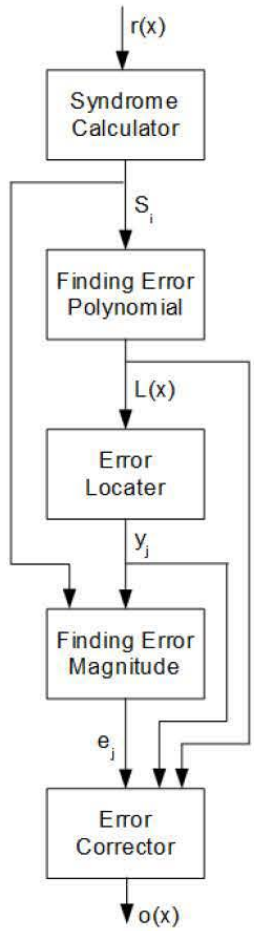


Figure 2.3: Steps of Reed-Solomon decoder

For simplicity of the implementation, $g(x)$ is usually calculated using Equation 2.6. There are $2t$ roots of $g(x)$; thus, $2t$ syndromes should be calculated. Equation 2.12 shows the syndromes.

$$\begin{aligned} S_1 &= r(\alpha) = r_0 + r_1\alpha + r_2\alpha^2 + ... + r_{n-1}\alpha^{n-1} \\ S_2 &= r(\alpha^2) = r_0 + r_1\alpha^2 + r_2(\alpha^2)^2 + ... + r_{n-1}(\alpha^2)^{n-1} \\ &\vdots \\ S_{2t} &= r(\alpha^{2t}) = r_0 + r_1\alpha^{2t} + r_2(\alpha^{2t})^2 + ... + r_{n-1}(\alpha^{2t})^{n-1} \end{aligned} \tag{2.12}$$

Implementing the calculation of the various powers of $\alpha$ is quite costly. Therefore, the implementation is performed using Equation 2.13. This architecture uses only one multiplier and one adder per a syndrome calculation, total of $2t$ multipliers and $2t$ adders. Recursive calculation of one multiplication followed by one addition operation is performed $n$ times. If one multiplication and one addition operation can be performed in one clock cycle, the latency of the syndrome calculator is $n$ clocks.

$$\begin{aligned}
S_1 &= ((\alpha r_{n-1} + r_{n-2})\alpha + r_{n-3})\alpha + ... + r_1)\alpha + r_0 \\
S_2 &= ((\alpha^2 r_{n-1} + r_{n-2})\alpha^2 + r_{n-3})\alpha^2 + ... + r_1)\alpha^2 + r_0 \\
&\vdots \\
S_{2t} &= ((\alpha^{2t} r_{n-1} + r_{n-2})\alpha^{2t} + r_{n-3})\alpha^{2t} + ... + r_1)\alpha^{2t} + r_0
\end{aligned} \tag{2.13}$$

Latency of the decoder is also an important parameter. Depending on the design or the use case the decoder is going to be used, a designer might wish to use an architecture that has lower latency than the architecture shown in Equation 2.13. Thankfully, architecture is a foldable architecture, which means there is a trade-off between latency and implementation complexity. It is possible to reduce the latency of the syndrome calculator by constant $c$ times by increasing the complexity $c$ times. Equation 2.14 shows how $S_1$ can be calculated in $n/2$ clock cycles. The same approach can be used for other syndromes or for higher $c$ values.

$$\begin{aligned}
S_{1_{even}} &= ((\alpha^2 r_{n-2} + r_{n-4})\alpha^2 + r_{n-6})\alpha^2 + ... + r_2)\alpha^2 + r_0 \\
S_{1_{odd}} &= (((\alpha^2 r_{n-1} + r_{n-3})\alpha^2 + r_{n-5})\alpha^2 + ... + r_3)\alpha^2 + r_1)\alpha \\
S_1 &= S_{1_{even}} + S_{1_{odd}}
\end{aligned} \tag{2.14}$$

After the calculation of the syndromes are done; if all of the syndromes are equal to zero, $r(x)$ is the output of the decoder; however, if any of the syndromes are non-zero, the decoder continues with the calculation of the error polynomial.

### 2.4.2 Finding Error Locator Polynomial

The step called 'Finding Error Locator Polynomial' takes the syndrome values as input and calculates a polynomial whose roots are equal to the inverse of the location of the erroneous symbols. The mathematical relation between the error locator polynomial, $\Lambda(x)$, and syndromes is given in Equation 2.15 where $\Lambda_j$ represents the coefficients of $\Lambda(x)$ and $\Lambda_0$ is equal to 1. The degree of $\Lambda(x)$ is equal to the number of erroneous symbols. If the number of errors is less than $t$, the coefficients of higher orders will be calculated as 0.

$$\begin{bmatrix} S_1 & S_2 & ... & S_{t-1} & S_t \\ S_2 & S_3 & ... & S_t & S_{t+1} \\ & & \vdots & & \\ S_t & S_{t+1} & ... & S_{2t-2} & S_{2t-1} \end{bmatrix} \begin{bmatrix} \Lambda_t \\ \Lambda_{t-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} S_{t+1} \\ S_{t+2} \\ \vdots \\ S_{2t-1} \\ S_{2t} \end{bmatrix} \tag{2.15}$$

Calculation of the error locator polynomial is the core component of RS decoder. Therefore, throughout the years, it has been the main focus of studies in order to improve the decoder. There are three main algorithms for calculating $\Lambda(x)$: Berlekamp-Massey (BM) algorithm [6], extended Euclidian algorithm (EEA) [7] and Welch-Berlekamp (WB) algorithm [8]. WB algorithm has not been used much due to its irregularity and high implementation complexity. The first VLSI implementation of RS decoder is implemented using EEA [9]. During the early days of VLSI implementation of RS decoders, EEA was more commonly used compared to BM algorithm due to its high regularity. However, in 1991 Reed et al. discovered inversionless Berlekamp-Massey algorithm (IBMA) which is highly regular and easy to implement. After the discovery of IBMA, BM based algorithms dominated the scene such that nowadays the step 'Finding Error Locator Polynomial' is called as 'Berlekamp-Massey step'. Some improvements to reduce the complexity or shorten the critical path etc. of IBMA have been discovered; however, the main structure has stayed the same. Reformulated Inversionless Berlekamp-Massey Algorithm (RiBMA) [10], enhanced IBMA (eIBMA) [11], enhanced parallel IBMA (ePIBMA)[11] can be named as examples

of improved algorithms.

After $\Lambda(x)$ is calculated, the roots of the polynomial are calculated to find the locations of the errors.

### 2.4.3 Error Locator

When the order of $\Lambda(x)$ is high, it is very costly to solve the equation mathematically to find its roots. Such an approach would require enormous amount of circuit components and power usage. Therefore, instead of solving the equation, brute force extensive search operation is performed. This search is called Chien search [12], named after inventor of the method. For this method; first, we calculate

$$\Lambda(\alpha^j) \;\; for \;\; j \in \{1, 2, ....., n-1, n\} \tag{2.16}$$

If $\Lambda(\alpha^j) = 0$, $(\alpha^j)$ is one of the roots of the polynomial, $r_i$ $(i \in \{1, 2, ..., t\})$, inverse of the roots:

$$(\alpha^j)^{-1} = \alpha^{-j} = \alpha^{2^m-1-j} \tag{2.17}$$

$\alpha^{2^m-1-j} = \ell_i$ shows the location of an erroneous symbol. $\ell_i$ $(i \in \{1, 2, ..., t\})$ values point to the error locations.

### 2.4.4 Finding Error Magnitude

The magnitude of the errors at each location is calculated by using the error evaluator polynomial $\Omega(x)$ defined in Equation 2.18.

$$\Omega(x) = S(x)\Lambda(x) \;\; (mod\, x^{2t}) \tag{2.18}$$

The Forney Algorithm, developed by Forney in 1965 [13], utilizes both the error locator polynomial ($\Lambda(x)$) and error evaluator polynomial ($\Omega(x)$) to calculate the error magnitude $e_i$ associated with each location $\ell_i$. Equation 2.19 shows the mathematical formula to find the error magnitudes.

$$e_i = \frac{\Omega({\ell_i}^{-1})}{\Lambda'({\ell_i}^{-1})} \tag{2.19}$$

### 2.4.5 Error Corrector

Computationally, correcting the errors is rather trivial after error locations and magnitudes are calculated. Error polynomial is written in 2.20.

$$e(x) = \sum_{i=1}^{t} e_i \ell_i \tag{2.20}$$

Simple addition of $e(x)$ to the received signal, $r(x)$, is enough to get the desired output signal: the codeword, $c(x)$.

$$\begin{aligned} r(x) &= c(x) + e(x) \\ r(x) + e(x) &= c(x) + e(x) + e(x) \\ r(x) + e(x) &= c(x) \\ o(x) &= c(x) \end{aligned} \tag{2.21}$$

If r(x) has less than or equal $t$ symbol errors, RS decoder catches all the errors and corrects them with the explained process. If $r(x)$ has more than $t$ error, error locator polynomial fails to find all the error locations; therefore, decoder cannot output a valid codeword.

## 2.5 Reed-Solomon Codes in Standards

For our point of interest, there are various standards for data transmission and FEC. RS codes are selected error correcting codes for some of these FEC standards. The following sections present the RS codes that appear in standards and their technical properties and requirements.

### 2.5.1 RS(255,239)

RS(255,239) is one of the most popular RS codes. RS(255,239) has been the choice for many standards such as IEEE 802.16 (Worldwide Interoperability for Microwave Access, commonly known as WiMAX), ITU-T G.975 [14] (Optical Fiber Submarine Cable Systems), ITU-T G.709 (Digital terminal equipments) and ITU-T Y.1331 (Internet Protocol Aspects  Transport). Due to its common appearance in standards, RS(255,239) decoders have been studied and researched quite extensively. It has a 8-bit symbol length and an error correction capability of up to 8 symbols. It has a code rate of 0.937 and at $10^{-15}$ BER 6.2 dB net coding gain compared to the uncoded transmission. Because of the higher coding gain requirements of the Optical Fiber Submarine Cable Systems, ITU-T G.975 is no longer actively used. It has been replaced by ITU-T G.975.1 [15]. However, in other mentioned standards, RS(255,239) is still actively in use.

### 2.5.2 RS(255,223)

RS(255,223) has a 8-bit symbol length and an error correction capability of up to 16 symbols with a code rate of 0.875 and; thus, it has better communication performance compared to RS(255,239). However, it is more complex due to the increased computations resulting from the increased error correction capability. RS(225,223) appears in the use-cases where better performance is required and complexity does not pose a huge problem. The Consultative Community for Space Data Systems (CCSDS) recommends using RS(255,239) in their standard document "Recommendation for Space Data System Standards - Synchronization and Channel Coding (CCSDS 131.0-B-3)."

### 2.5.3 RS(255,191)

RS(255,191) has a 8-bit symbol length and an error correction capability of up to 32 symbols. It is very complex and rarely used. However, it is the chosen standard error correcting code for DVBH (Digital Video Broadcasting Handheld). High reliability is required for this standard and RS(255,191) is able to provide it with its high error correction capability. RS(255,191) has a code rate of 0.749.

### 2.5.4 RS(204,188)

RS(204,188) has a 8-bit symbol length and an error correction capability of up to 8 symbols. It is the chosen standard error correcting code for DVBT (Digital Video Broadcasting Terrestrial). RS(204,188) has a code rate of 0.922.

### 2.5.5 RS(2720,2550)

RS(2720,2550) has a 12-bit symbol length and an error correction capability of 85 symbols with a code rate of 0.937. It is significantly more complex than RS codes that have lower symbol lengths or block lengths. Nonetheless, communication performance is considerably better; at $10^{-15}$ BER it has a coding gain of 8 dB compared to uncoded transmission. Due to its high coding gain, RS(2720,2550) is one of the recommended error correcting codes in ITU-T G.975.1 [15]. Although there have not been any newer standards to replace this code; nowadays, other FEC codes are used for fiber optical communication due to demand for increased coding gain and higher throughput.

### 2.5.6 Summary of the Standards

RS codes are the chosen error correcting codes for some of the most critical standards. However, they have lost some of their value due to increasing demand for higher throughputs and better communication performance. Their implementation complexity increases rapidly if these demands are met. Therefore, as it can be seen from Table 2.3, RS codes that appear in the standards have high code rates and relatively short block lengths and the maximum throughput requirements are not up to par with current high throughput demands. The highest throughput of these standards is 40 Gb/s while throughput requirements, especially for fiber optical communications, could reach up to 1 Tb/s. The capability of RS codes whether they can satisfy higher throughput demands needs to be studied.

## 2.6 Literature Survey

In this chapter, the state of the art of the Reed-Solomon codes and other error correcting codes developed for fiber optical communications are presented. The

Table 2.3: RS codes in standards, their properties and requirements

| Code | RS(255,239) | | | | RS(255,223) | RS(255,191) | RS(204,188) | RS(2720,2550) |
|---|---|---|---|---|---|---|---|---|
| Standard | IEEE 802.16 | ITU G.975 | ITU G.709 | ITU Y.1331 | CCSDS 131.0-B-3 | DVB-H | DVB-T | ITU G.975.1 |
| Symbol Length | 8 | | | | 8 | 8 | 8 | 12 |
| Code Rate | 0.937 | | | | 0.875 | 0.749 | 0.922 | 0.937 |
| Error Corr. Capability | 8 | | | | 16 | 32 | 8 | 85 |
| Maximum TP | 1 Gb/s | 40 Gb/s | 40 Gb/s | 40 Gb/s | NS | 10 Mb/s | 32 Mb/s | 40 Gb/s |
| Status of the Standard | Active | Inactive | Active | Active | Active | Active | Active | Inactive |

criteria for the decoders are high throughput, coding gain and complexity of the implementation with an emphasis on the high throughput. Section 2.6.1 presents the state of the art of RS codes while Section 2.6.2 presents the other error correcting codes with high throughputs. We will present our conclusions of the chapter in Section 2.6.3.

### 2.6.1 Survey on Reed-Solomon Decoders

Reed-Solomon decoders with at least 100 Gb/s throughput are presented in this section. Due to its lower implementation complexity, RS(255,239) is more suitable for higher throughputs compared to other RS codes with more error correction capability; thus, all of the presented decoders are of RS(255,239). RS(255,239) is also the most used RS code as explained in Section 2.5; therefore, it is widely researched in literature, which is another reason why all of the presented decoders are different implementations of RS(255,239). Unfortunately, none of the papers mentions the power consumption of the decoder and only one of them mentions the area consumption. Their results are presented using the criteria "number of gates", which will be used in our comparison.

Decoder presented in [16] uses two parallel channels in order to re-use some of the components and match the latency of the blocks. For solving the key equation, Euclidian based algorithm, pipelined degree-computationless modified Euclidean (pDCME) [17], is used. For calculating the roots, Chien Search [12] is used as

for all the other RS algorithms mentioned in this section. When implemented on 180nm CMOS technology, the net throughput of the decoder is 96 Gb/s.

Lee developed a decoder [18] which uses the same algorithms as [16]; however, on three channels instead of two, thus it can reach a higher throughput. Throughput of the decoder is 108 Gb/s when implemented on 130nm CMOS.

Decoder developed by Park [19] uses three parallel channels, as well. As a key equation solver algorithm, it deploys BM based pipelined truncated Inversionless Berlekamp Massey algorithm (pTiBM). When implemented on 90nm CMOS, the decoder can reach 225 Gb/s net throughput.

16 parallel channels are used by [20]. For solving the key equation BM based compensated simplified reformulated inversionless Berlekamp-Massey (CS-RiBM) algorithm is used. The decoder is implemented using 90nm CMOS technology and can reach to net throughput of 146 Gb/s.

Decoder developed by Perrone [21] is a single channel RS decoder published in 2018. Therefore, it employs state-of-the-art RS algorithms, namely enhanced Parallel Inversionless Berlekamp-Massey algorithm (ePIBMA) [11] for key equation solver and Chien Search for calculating the roots from key equation. When implemented on 90nm CMOS technology, its net throughput is 132 Gb/s with only 113,442 gates.

The summary of the mentioned decoders and their parameters are shown in Table 2.4. [19] reaches the highest throughput among the decoders; however, it falls behind [21] on throughput/complexity ratio.

### 2.6.2 Survey on FEC Decoders for Optical Communications

FEC decoders with high throughput and high coding gain, which are developed for optical communications, are presented in this section. Achieving both high

throughput and high coding gain is a formidable task and decoders that can achieve both usually have very high implementation complexity. Due to their high implementation complexity, most of the decoders in this category leave the research at algorithm level and do not actually implement it on FPGA or ASIC. Our main priority is comparing the complexity of the decoders for optical communications. Therefore, decoders without implementation results are not presented. Survey of such decoders can be read from [1].

BCH codes in staircase architecture running iteratively are examined in [22]. Several different inner BCH codes are implemented. Decoder with the highest net throughput achieves 1 Tb/s with 1.87 W power dissipation when implemented on 28nm CMOS technology. Communication performance of the decoders are measured analytically and with extrapolation. All of the configurations can achieve more than 9 dB coding gain compared to uncoded transmission.

BCH codes implemented in product architecture running iteratively are presented in [23]. Decoder has the code block length of 65025 bits. The inner code is BCH(255,231,3). Different configurations with different number of iterations are implemented. When the number of iterations is 5, decoder achieves 1 Tb/s throughput with 0.633 W power dissipation while having 10.3 dB estimated coding gain.

Table 2.4: State of the art of high throughput RS codes

| Reference | Code | Freq (MHz) | Latency (CC) | Latency (ns) | Net TP (Gb/s) | No. of Gates | Technology (nm) | Voltage (V.) |
|---|---|---|---|---|---|---|---|---|
| [16] | $(255, 239)$ | 400 | 260 | 650 | 96 | 434800 | 180 | 1.8 |
| [18] | $(255, 239)$ | 300 | 242 | 800 | 108 | 378000 | 130 | 1.2 |
| [19] | $(255, 239)$ | 640 | 161 | 260 | 225 | 417600 | 90 | 1.2 |
| [20] | $(255, 239)$ | 625 | 260 | 416 | 146 | 269000 | 90 | 1.2 |
| [21] | $(255, 239)$ | 555 | 31 | 56 | 132 | 113442 | 90 | 1.2 |

### 2.6.3 Conclusion of the Literature Survey

Surveys on RS decoders and other FEC decoders developed for optical communications clearly show that RS decoders are falling behind on both high throughput and high coding gain demands. They lose their places in the standards where such are demanded. Reaching 1 Tb/s throughput while having more than 10 dB coding gain is distant target for regular RS decoders.

Section 2.6.2 shows that it is very hard to implement high throughput and good performance decoder because they are very complex. It is not a coincidence that two decoders that can be implemented use iterative architecture and relatively simple inner codes. Iterative architectures are highly regular and with simple inner codes, they are much simpler compared to other highly complex decoders developed for fiber optics. This has led us to consider using simple RS codes with low error correction capability in product architecture.

The idea of using PRS codes concatenated with polar codes is introduced in [1] and satisfactory results are presented. This novel approach is promising enough to be contender for the decoder to satisfy requirements of fiber optical communications. In this thesis, we have focused on the implementation of the PRS decoder and carried on the synthesis of PRS(208,204) using the Genus tool with the 28nm TSMC library. The results show that PRS(208,204) can be implemented with an energy efficiency of 6 pJ/bit.

## 2.7 Summary of Review of Reed-Solomon Codes

The essence of the Reed-Solomon codes is linear algebra in Galois Fields. In order to calculate $2t$ unknowns ($t$ error locations and $t$ error values at those corresponding locations), $2t$ syndrome values are calculated to generate $2t$ equations. Decoding process of RS codes is very structured. Strict mathematical model of RS codes does not allow flexible structure; therefore, there has not been a major algorithmic change in decoding of RS codes at least for a decade. Thus, studies of RS codes have been slowed down and RS codes have started to lose their places in some of the standards such as fiber optics. In fiber optics area, the state of RS decoders compared to other decoders is shown in Section 2.6. We have tried to tackle the problem of RS codes falling behind and in Chapters 3 and 4, our proposed solution, a polar decoder concatenated with PRS, and its implementation are explained.

# Chapter 3

# Implementation of RS(208,204)

In this chapter, the implementation of RS(208,204) is addressed. We discovered some new methods and some new architectures for existing methods during the implementation studies of RS(208,204). Implementation parameters, architectures and algorithms are explained and results such as power consumption, area usage, communication performance etc. are presented in their respective sections. This section presents the design of the RS(208,204) decoder. Although implementation is carried with RS(208,204) many of the developed methodology and algorithms can be used for RS decoders with symbol length ($m$) equal to 8, and error correction capability ($t$) equal to 2. Prime polynomial used for the design is $p(x) = x^8 + x^4 + x^3 + x^2 + 1$ .

Chronological order of the algorithms that are deployed during RS(208,204) is shown in Figure 3.1. As it can be seen from Figure 3.1, the latency of the decoder is 19 clock cycles (CCs). After the syndrome calculation is performed, new input signal is fed to the decoder; thus, making the pipeline level of the decoder equal to 2. Every 10 CCs, a new output is delivered.

As it is shown in Section 2.4, addition and multiplication in GF($2^8$) are frequently used operations during the decoding process of RS codes. Addition in GF($2^8$) is a very simple operation, bit-wise XOR. However, multiplication is

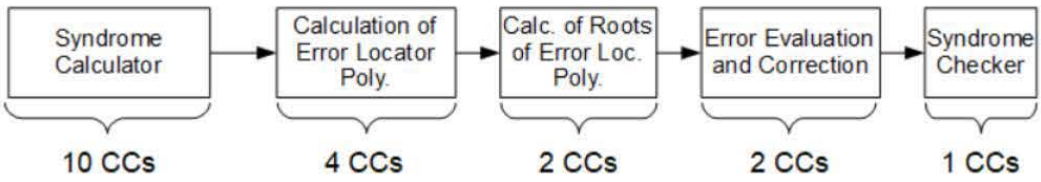


Figure 3.1: Chronological order of the processes during RS(208,204)

rather complex and the design of the multiplication circuit is very important as multiplication operation is performed many times during the decoder. The design of the multiplication circuitry is explained in Section 3.1. Section 3.2-3.5 presents the designs and algorithms used for implementing the various stages of the decoder. Implementation results of the decoder is given in Section 3.7 while communication performance of the decoder is given in Section 3.8 .

## 3.1 GF($2^8$) Multiplier Design

The choice for the multiplier design is rather important due to its repetitive usage throughout the decoder. Both complexity and latency of the decoder are important parameters. The simplest design for the multiplier is memory based tables where $m$ bit binary symbols are matched to power of the *alpha* value they represent. Multiplication is performed by summing the powers of the *alpha* values and $m$ bit binary result is found from another table that stores the $m$ bit values for each power of *alpha*s. These memory based tables are called logarithm and anti-logarithm tables and there are methods to construct them [24]. However, this design has one major drawback; each memory unit can be called only once each clock cycle. Therefore, number of tables implemented must be equal to the number of multiplications performed in each clock cycle. Implementing a lot of tables are very costly in terms of area, routing complexity and latency.

Therefore, we used another multiplier design which is based on the Karatsuba Algorithm [25]. The Karatsuba Algorithm aims to reduce the complexity of the

multiplication by using the method "divide and conquer". This algorithm reduces the complexity of multiplication from $O(n^2)$ to $O(n^{log_2 3})$. A slight modification to original Karatsuba Algorithm enables it to be used for GF multiplication operations, as shown in [26]. We unrolled the algorithm in [26] and used it as our GF multiplier.

Using the Karatsuba Algorithm based multiplier and unrolled design has another very important benefit. There are some multiplication operations in Reed-Solomon chain in which one of the multipliers is constant. For example, for the multiplications during the syndrome calculation; one multiplier comes from the input signal which is variable and the other multiplier comes from the polynomial which is constant. By taking advantage of this constant multiplier, we can reduce the complexity of the multiplication operation where one of the multipliers is constant.

Table 3.1 shows the complexity values of the multiplication circuit with two variable multipliers and several multiplication circuits where one variable is constant. As it can be seen from the Table 3.1, multiplication circuits with one constant multiplier are four to eight times simpler.

There are 255 different multiplication circuits with one constant input. Although all of the possible multipliers are not used in a single decoder design, significant amount of them are used; therefore, there are still a lot of design work to be done. Instead of designing all of the circuits by hand, I have developed a Python script that writes the VHDL code of the required multipliers when the constant input and prime polynomial are provided.

## 3.2 Syndrome Calculator

Syndrome calculator block is the part of the decoder where the syndromes are calculated. The main algorithm of syndrome calculation is given in Section 2.4.1, Equation 2.13. There are not many things that can be improved from Equation

Table 3.1: Multipliers Synthesized using TSMC 40nm Library at 400 MHz and $V_{DD} = 0.81V$

| First Multiplier | Second Multiplier | ASIC Synthesis | | | |
|---|---|---|---|---|---|
| | | Cells | Power (uW) | Area (um^2) | Slack (ps) |
| Variable | Variable | 137 | 202 | 405 | 107 |
| | 00000010 | 9 | 23 | 56 | 1352 |
| | 00000100 | 13 | 30 | 63 | 1483 |
| | 00001000 | 13 | 29 | 68 | 1235 |
| | 00010000 | 15 | 36 | 75 | 1468 |
| | 00100000 | 17 | 41 | 78 | 1251 |
| | 01000000 | 18 | 43 | 82 | 1038 |
| | 10000000 | 20 | 47 | 87 | 1214 |
| | 01110101 | 18 | 43 | 82 | 1207 |
| | 11001110 | 19 | 53 | 91 | 935 |
| | 11010111 | 22 | 62 | 97 | 891 |
| | 11111110 | 20 | 53 | 90 | 663 |
| | 00111110 | 20 | 55 | 92 | 936 |

2.13; however, folding the equation to reduce the latency of the calculation while increasing the implementation complexity is one of the changes that can be made on the equation. Our implementation of RS(208,204) aims to reach very high throughputs. Therefore, we preferred to reduce the latency of the syndrome calculator as much as possible and folded the equation as much as necessary to reach the latency of 10 clocks. While we used 10 clock version of syndrome decoder for this implementation, we are also aware that latency parameter depends on the requirements and parameters of the decoder and it can change if requirements of the decoder change. In order to avoid rewriting the syndrome calculator after every change, we coded a Python script the parameters prime polynomial, number of received symbols, number of received data symbols and desired latency of the syndrome calculator block are given; Python script can write the VHDL code of the entire syndrome calculator block. Thus, we automatize the VHDL coding of the block in this way. The inputs and output of the Python based automation script is shown in Figure 3.2.

Input and output ports of the block is given in Figure 3.3. '*clk*' refers to the

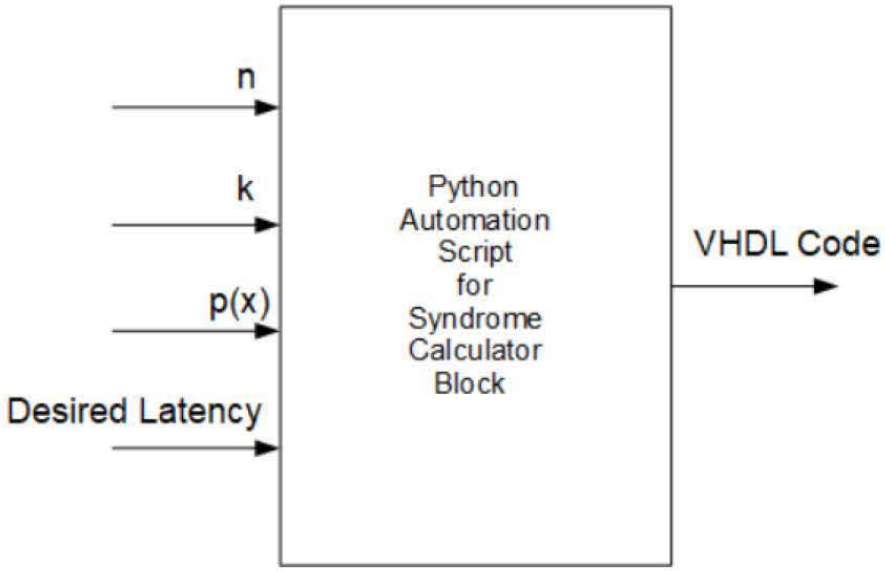


Figure 3.2: Inputs and outputs of Python Automation Script

clock signal which is synchronized throughout the decoder, it is a one bit signal. 'reset' refers the reset signal which is synchronized throughout the decoder and resets the decoder into its original state. '$clk_{en}$' is a one bit wide input signal that provides the information that received signal at the receiver port is valid or not; if '$clk_{en}$' port is at low '(0)', it means received signal is not valid and we should not make calculations on it, if the port is high '(1)', it suggests that the received signal is valid and syndromes can be calculated for that signal. '$r(x)$' is the received signal and port width is equal to the $n \times m$, which is equal to $208 \times 8 = 1664$ bit for our design RS(208,204). $Synd.Pol(S(x))$ is the main output signal of the block, denoted as $S(x) = s_0 + s_1x^1 + s_2x^2 + s_3x^3$. $S(x)$ is 4 symbols long and carries the syndrome information to the other blocks.

## 3.3 Calculation of the Error Locator Polynomial

Calculation of the error locator polynomial has been the most challenging part of RS decoders since their existence and; therefore, many methods have been

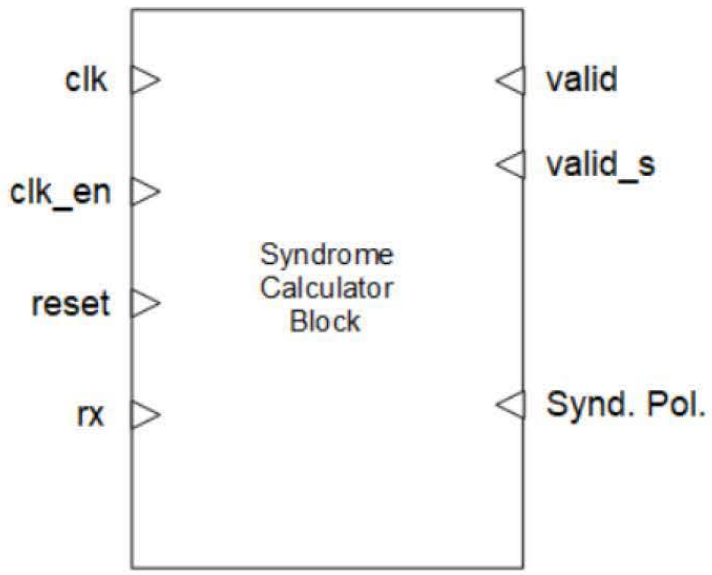


Figure 3.3: Inputs and outputs of Syndrome Calculator Block

developed to calculate the polynomial. We choose to implement this block with one of the current state of the art algorithms: enhanced parallel Inversionless Berlekamp-Massey algorithm (ePIBMA) [11]. Inputs and outputs of this block are given in Figure 3.4.

'$clk$', '$clk_{en}$', '$reset$' and '$valid$' signals are the same for all of the blocks. Therefore, they will not be explained again to avoid repetition.Input signal '$Synd.Pol.$' is the output of the Syndrome Calculation block and carries the same properties. The output of this block is error locator polynomial which is polynomial whose roots are the location of the errors. Therefore, the degree of the polynomial is at most 2.

Python based automated VHDL creator is coded for this block as well as the following blocks. In order to avoid repetition, it will not be mentioned in this section or the following sections.

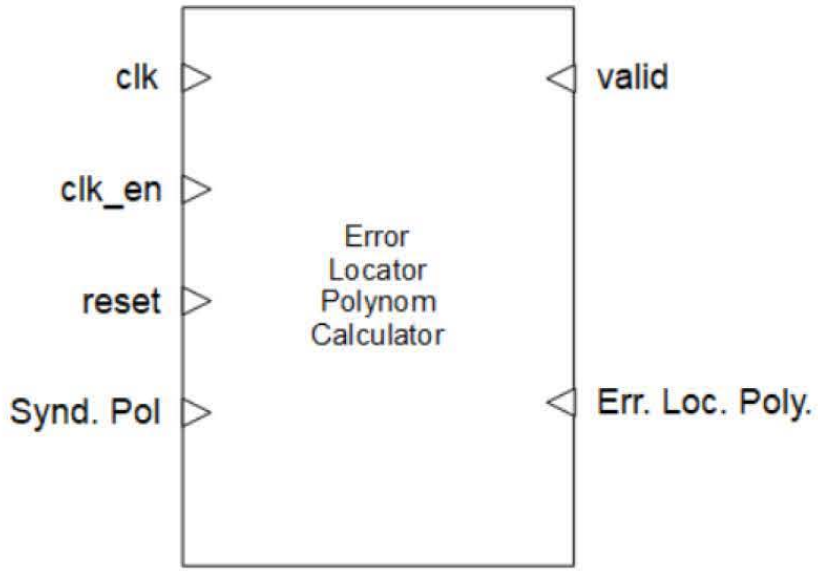


Figure 3.4: Inputs and outputs of Calculation of Error Locator Polynomial Block

## 3.4 Calculation of the Roots of the Error Locator Polynomial

Error locator polynomial, $\Lambda(x)$, is a polynomial that is used to calculate the locations of the erroneous symbols as explained in Section 2.4.2. Most of RS decoders use the method called Chien Search for this step [12]. Chien search is a brute force method that tries every possible error location and calculates if the result is equal to zero. If the result is equal to zero, it means that location is one of the roots of the $\Lambda(x)$. When the degree of the $\Lambda(x)$ is high, which is the case for most of RS decoders in literature, Chien search is very cost efficient compared to solving the $\Lambda(x)$ for the roots. Therefore, Chien Search is almost used exclusively for this step of the decoder. However, complexity of Chien Search depends on the number of symbols; thus, the complexity of Chien Search does not change because RS(208,204) has a low error correction capability. Chien Search for RS(208,204) would require 416 multipliers and 3328 XoR gates [11]. In our decoder RS(208,204), error correction capability of the decoder is two and;

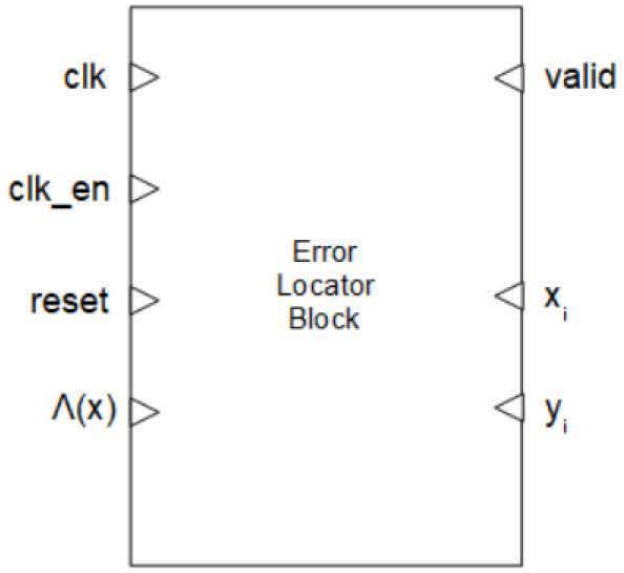


Figure 3.5: Input and output ports of calculation of the roots of the error locator polynomial block

consecutively, degree of the $\Lambda(x)$ is at most two. Solving the $\Lambda(x)$ is a lot more efficient than calculating the result in a brute force manner. Our solution uses 6 multipliers, 72 XoR gates and one look-up table for the inversion operation which is significantly much simpler than using Chien Search.

If there are no errors, the decoder detects that the received signal is a valid codeword at the syndrome calculation stage. Therefore, if there are no errors, $\Lambda(x)$ is never calculated. Thus, there are only two possible degrees that $\Lambda(x)$ can have, which are one or two. We use similar but different approaches for different degrees of $\Lambda(x)$. For the following discussions, we will use the definition $\Lambda(x) = ax^2 + bx + c$.

We can easily detect the degree of the $\Lambda(x)$ by examining the coefficient of the second order term $a$. If $a$ is equal to zero, the degree of $\Lambda(x)$ is equal to one, otherwise it is equal to two. We will start by finding the location of the error when there is only one. Calculation process is shown in Equation 3.1. Calculation

is quite simple, since $a$ is equal to zero, $\Lambda(x)$ is equal to $bx + c$. We add $c$ to both sides of the equation because adding same values in GF($2^8$) results in zero, left hand side is equal to $bx$ and right hand side is equal to $c$. To find $x_1$ where $x_1$ is the root of the equation, we only need to divide $c$ by $b$. Implementation wise, division in GF($2^8$) is very costly. Therefore, we used a look-up table for taking the inverse of the symbols and multiplied the inverse of $b$ with $c$. The size of the look-up table used for taking the inverse is $255 \times 8 = 10.2$ kbits.

$$\begin{aligned} ax^2 + bx + c &= 0 \\ bx + c &= 0 \\ bx + c + c &= 0 + c \\ bx &= c \\ x_1 &= \frac{c}{b} \\ x_1 &= c \times b^{-1} \end{aligned} \tag{3.1}$$

Finding the roots of $\Lambda(x)$ when degree of $\Lambda(x)$ is equal to two is much challenging. It is shown by Berlekamp and co. [27] that square and square root operations are linear operations in GF($2^8$). Adding the two same values in GF($2^8$) results in zero. Therefore, during the squaring operation, the middle term $2 \times a \times b$ disappears as shown in Equation 3.2.

$$(a + b)^2 = a^2 + a \times b + a \times b + b^2 = a^2 + b^2 \tag{3.2}$$

Before taking advantage of the linearity property of the square operation in GF($2^8$), we need to manipulate the error locator polynomial according to our needs. These manipulations are shown in Equation 3.3.

$$
\begin{aligned}
ax^2 + bx + c &= 0 \\
x^2 + \frac{b}{a}x + \frac{c}{a} &= 0 \\
x^2 + \frac{b}{a}x &= \frac{c}{a}
\end{aligned}
\tag{3.3}
$$

At this point we use the change of variable $y = \frac{a}{b}x$.

$$
\begin{aligned}
\frac{b}{a}^2 y^2 + \frac{b}{a} \times \frac{b}{a}y &= \frac{c}{a} \\
y^2 + y &= \frac{a \times c}{b^2}
\end{aligned}
\tag{3.4}
$$

We use the change of variable again such that $d = \frac{a \times c}{b^2}$.

$$
y^2 + y = d \tag{3.5}
$$

Solving Equation 3.5 to find the root $y_1$ and $y_2$ is very costly. We can store the roots for each $d$ value in a look-up table and get the roots from there; however, the size of such a look-up table would be enormous when we want to store the roots for every possible $d$ value, which leads us to taking advantage of the linearity of the square operation in GF($2^8$). The left hand size of Equation 3.5, $y^2 + y$ is linear due to the linearity of both square and addition operations. If we treat $d$ as a 8 dimensional vector and store the roots for these dimensions, we can calculate the root $y_1$ from the stored results. For our decoder RS(208,204) with $p(x) = x^8 + x^4 + x^3 + x^2 + 1$, we calculated the solutions using a software code written in Python and stored them in a look-up table. Stored values can be seen in Table 3.2.

For example, let us calculate the $y_1$ when $d = 01000101$. We check the Table 3.2 for which bits of the $d$ are equal to '1′ and perform summation in GF($2^8$) of those values to find $y_1$. These calculations are shown in Equation 3.6.

Table 3.2: Solution Table for y-d Pairs

| $d$ | $y_1$ |
|---|---|
| 00000001 | 11010110 |
| 00000010 | 11101000 |
| 00000100 | 11101010 |
| 00001000 | 00101101 |
| 00010000 | 11101110 |
| 00100000 | - |
| 01000000 | 00100101 |
| 10000000 | 01010000 |

$$\begin{aligned} y_{1_1} &= 11010110 \\ y_{1_2} &= 11101010 \\ y_{1_3} &= 11101000 \\ y_1 &= y_{1_1} + y_{1_2} + y_{1_3} \\ y_1 &= 11010100 \end{aligned} \tag{3.6}$$

Finding the second root $y_2$ from $y_1$ is really simple: $y_2 = y_1 + 1$. The solution can be verified easily using Equation 3.5. Steps are shown in Equation 3.7

$$\begin{aligned} y_2^2 + y_2 &= d \\ (y_1 + 1)^2 + y_1 + 1 &= d \\ y_1^2 + 1^2 + y_1 + 1 &= d \\ y_1^2 + 1 + y_1 + 1 &= d \\ y_1^2 + y_1 &= d \end{aligned} \tag{3.7}$$

Instead of the memory block presented in Table 3.2, it is possible to use a memory block that represents $y_1$ values for every possible d value. This approach

would spare us from calculating the linear combination of the solution; however, it would require a much bigger look-up table. Although look-up tables are good solutions to many implementation problems we encounter, excessive use of them increases the area usage; thus, make the design harder to route. Therefore, we preferred to use smaller look-up tables or solutions without a look-up table when we can.

We find the roots of the equation using the calculated value $y$ and coefficients of $\Lambda(x)$, namely $a$ and $b$.

$$\begin{aligned} y &= \frac{a}{b}x \\ \frac{b}{a}y_1 &= x_1 \\ \frac{b}{a}y_2 &= x_2 \end{aligned} \tag{3.8}$$

We have already implemented a look-up table for calculating the reverse of the symbols and we can calculate the locations of the erroneous symbols $x_1$ and $x_2$by taking the inverse of the roots $y_1$ and $y_2$.

Using the values calculated up to this point, we will evaluate the errors and correct them as a last duty of the decoder, which is explained in Section 3.5.

## 3.5 Error Evaluation and Error Correction

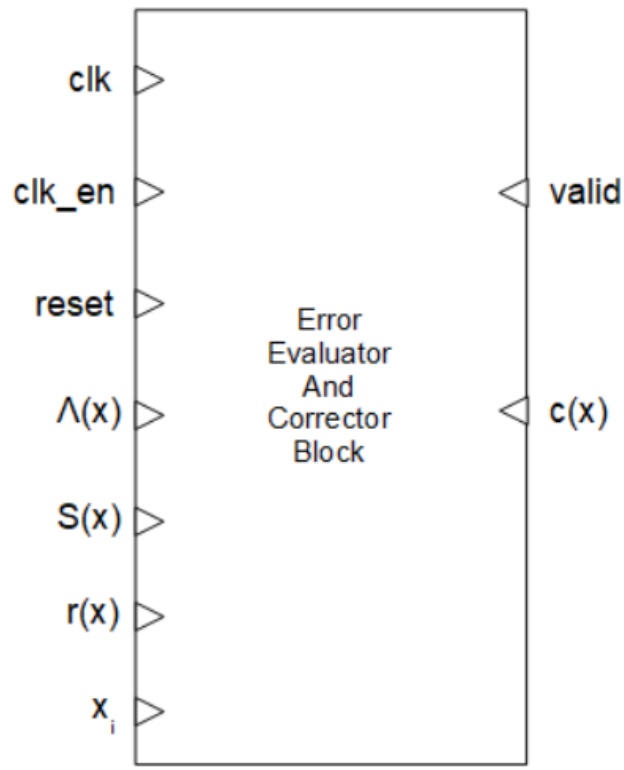


Figure 3.6: Input and output ports of error evaluation and correction block

Evaluating the errors at the locations we previously calculated is carried out by using the Forney Algorithm(FA) [13]. FA is used pretty much exclusively at this step for every decoder. Equation 3.9 shows the formulation of the FA.

$$e_i = \frac{\Omega({x_i}^{-1})}{\Lambda'({x_i}^{-1})} \tag{3.9}$$

$\Omega(x)$ is calculated according to Equation 3.10. Three multiplications and one summation operation are enough to calculate the $\Omega(x)$.

$$\begin{aligned}
\Omega(x) &= [S(x) \times \Lambda(x)]\ (mod\, x^t) \\
\Omega(x) &= (s_0 + s_1 x + s_2 x^2 + s_3 x^3) \times (ax^2 + bx + c))\ (mod\, x^t) \\
\Omega(x) &= s_0 \times c + (b \times s_0 + c \times s_1) \times x
\end{aligned} \tag{3.10}$$

Calculation of $\Lambda'(x)$, where " ' " denotes the derivation, is shown in Equation 3.11. We take the derivative of $\Lambda(x)$ like a regular derivation is carried out which makes the derivation equal to $2ax + b$. In GF($2^8$), '2' is equal to '0'; therefore, the end result is $\Lambda'(x) = b$.

$$\begin{aligned} \Lambda(x) &= ax^2 + bx + c \\ \Lambda'(x) &= 2ax + b \\ \Lambda'(x) &= b \end{aligned} \tag{3.11}$$

Since $\Lambda'(x)$ is constant, calculating it for the different locations is not needed. We calculate $\Omega({x_i}^{-1})$ values for both of the locations and multiply them with $b^{-1}$. Taking the inverse of $b$ is performed using the look-up table.

After the calculation of the location and error values at those locations, correcting the errors is quite easy. Error values should be added to symbols that are at the calculated error locations. The calculations are shown in Equation 3.12.

$$\begin{aligned} c(\ell_1) &= r(\ell_1) + e_1 \\ c(\ell_2) &= r(\ell_2) + e_2 \\ c(\ell_i) &= r(\ell_i) \quad (\ell_i \neq \{\ell_1, \ell_2)\} \end{aligned} \tag{3.12}$$

After the correction of the errors, the decoder performs some additional calculations to check if the reached codeword is valid. Section 3.6 explains which check steps are performed at which steps of the decoder and what is the reason for performing those checks.

## 3.6 Correction Check

RS(208,204) contains some additional calculations compared to regular RS decoders. These operations are performed during various stages of the decoding process; however, they are mentioned in this section together.

The reason for performing these additional calculations is explained using Figure 3.7. On Figure 3.7, only two codewords ($c_1$ and $c_2$) out of $2^{204\times 8}$ possible codewords are shown for the sake of simplicity. Circles around the $c_1$ and $c_2$ represent the decode-able area for each codeword and received signals are represented by $r_i$.


Figure 3.7: Representative Decoding Space

Assume $c_1$ is sent from the encoder, different cases will be examined for different received signals:

**Case 1: $r_1$ is received**

When the received signal is two or less symbols away from the sent codeword, RS(208,204) can correctly decode and reach to the sent codeword. In this case, when $r_1$ is received, the output of the decoder will be $c_1$, which is the correct result.

**Case 2: $r_2$ is received**

When the received signal is more than two symbols away from the sent codeword, output of the decoder cannot be correct as RS(208,204) has only a two symbols correcting capability. However, if the received signal is in the solvable space of another valid codeword, the output of the decoder will be that codeword and decoder would think that it solved the signal correctly. In this case, when $r_2$ is received, the output of the decoder will be $c_2$, which is not the correct output; however, on the receiver side, there is no way of knowing that this is not the correct result. Therefore, mistakes performed because of the situations like Case 2 cannot be corrected.

**Case 3: $r_3$ is received**

When the received signal is more than two symbols away from the sent codeword and it is not inside the decode-able area of any other codeword either; decoder tries to decode the received signal; however, output will not be a valid codeword and; therefore, it will not be correct. In this case, when $r_3$ is received, output of the decoder will be some point which is not a valid codeword. In such cases, the frame error is not corrected and number of bit errors might increase or decrease depending on the output signal; however, increase in the number of bit errors is more likely.

There is nothing to correct in "Case 1" and there is nothing that can be corrected in the receiver side in "Case 2". Therefore, the aim of performing the checks is to avoid the further corruption of the received data when "Case 3" happens. Regularly used RS decoders do not have these anti-corruption measures mainly because of two reasons. First, most RS decoders have a higher error correction capability, which means their decode-able area circles are much bigger; thus, the number of received signals that fall into the "Case 3" type areas is very low. Therefore, it is not worth to add more circuit elements and increase the latency to avoid a case that happens very rarely. Second, RS(208,204) is designed to be used in product structure; corrupting some symbols during vertical decoding would harm the horizontal decoding which will happen during the next iteration and vice versa. If the output of RS decoder is the final step of the system and if it will not be used as an input for any other components, it is not worth to add

extra components to avoid slight increase in BER.

RS(208,204) uses three different check measures to avoid the mentioned problem. These checks are explained in the following sections.

### 3.6.1 Solution Check

During the calculation of the roots of the error locator polynomial, explained in Section 3.4, the solution table for y-d pairs is implemented, Table 3.2. There are no solutions presented in Table 3.2 when $d = 00100000$. There is no solution presented for $d = 00100000$ because there is none available. None of the valid "$d$" value can have "1" in their 3rd most significant bit. However, sometimes 3rd significant bit of "$d$" is "1" in which case the decoder knows that the received signal cannot be decoded correctly. Thus, instead of carrying on with decoding and spending power for a futile effort, RS(208,204) terminates decoding and outputs the received signal without change. Implementation cost of this check is almost zero as it can be implemented with a single gate checking whether the 3rd most significant bit is "0" or "1". However, terminating the decoder early can have substantial power savings.

### 3.6.2 Location Check

During the calculation of the location of the errors step, the location of the erroneous symbols is calculated. Since RS(208,204) consists of 208 symbols, all the calculated locations should be in range [0,207]. If any of the error locations is bigger than 207, which means calculation is wrong and "Case 3" applies. Therefore, decoding process is terminated and the received signal is fed into the output without change. Implementation of this check can be performed with two MUXes. Much like aforementioned "Solution Check" this check is very cheap, as well, and it also saves considerable amount of power by terminating the decoder early.

### 3.6.3 Syndrome Update

If the output of the decoder is a valid codeword, syndromes of that codeword should be equal to zero. Calculating the syndromes from scratch is not a viable option due to vastly increased latency and implementation complexity. However, after the calculation of error locations and corresponding error magnitudes, syndrome values can be updated easily. To update the syndromes, we use Equation 2.12. We implement 4 memory blocks to store the values of location multipliers for $S_1$, $S_2$, $S_3$ and $S_4$. Multiplying the error values with corresponding memory entry and adding it to initial syndrome yield the updated syndrome values. If the values are not equal to zero, this means that the output is not a valid codeword and the received signal is given as the output.

Syndrome check could catch all the "Case 3" type inputs; thus, it encapsulates both solution and location checks. Performing only the syndrome check would yield the same performance. However, syndrome update is much more costly than other checks; thus, it should be avoided if possible. Therefore, we perform the solution and location checks to see if the possible faulty outputs can be caught by using those cheaper methods.

## 3.7 Implementation Results

Implementation of RS(208,204) carried out using the Genus tool and $28nm$ library of TSMC ($tcbn28hpcbwp12t30p140ssg0p72v125c_ccs$). The decoder runs at 200 MHz, which resulted in 32.64 Gb/s net throughput. Total power consumption of the decoder is 8.965 mW which is a minuscule amount and it uses 0.059 $mm^2$ area. Other important parameters are given in Table 3.3. As it can be seen from Table 3.3, RS(208,204) is very energy efficient.

Table 3.3: Implementation results of RS(208,204)

| | |
|---|---|
| Technology ($nm$) | 28 |
| Throughput ($Gb/s$) | 32.64 |
| Area ($mm^2$) | 0.059 |
| Power ($mW$) | 8.965 |
| Area Eff. ($Gb/s/mm^2$) | 564.068 |
| Pow. Den. ($W/mm^2$) | 0.152 |
| Energy Eff. ($pJ/bit$) | 0.269 |
| Frequency ($MHz$) | 200 |

## 3.8 Communication Performance

The communication performance of RS(208,204) is calculated using the Monte Carlo simulations using Python software. BER and FER of RS(208,204), when BPSK modulation and demodulation is used and channel is AWGN, is presented in Figure 3.8. The first generation fiber optic FEC RS(255,239) has a 6.2 dB coding gain at BER $10^{-15}$ while the second generation fiber optic FEC RS(2720,2550) has an 8 dB coding gain at BER $10^{-15}$. Compared to earlier FEC solutions, RS(208,204) has an abysmal communication performance with less than 3 dB coding gain at BER $10^{-15}$. The reason for this very poor communication performance is the very low error correction capability of the code. Fortunately, we are not planning on using RS(208,204) alone, we are planing to use it in an iterative manner with product RS(208,204) decoder, which is explained in Section 4.

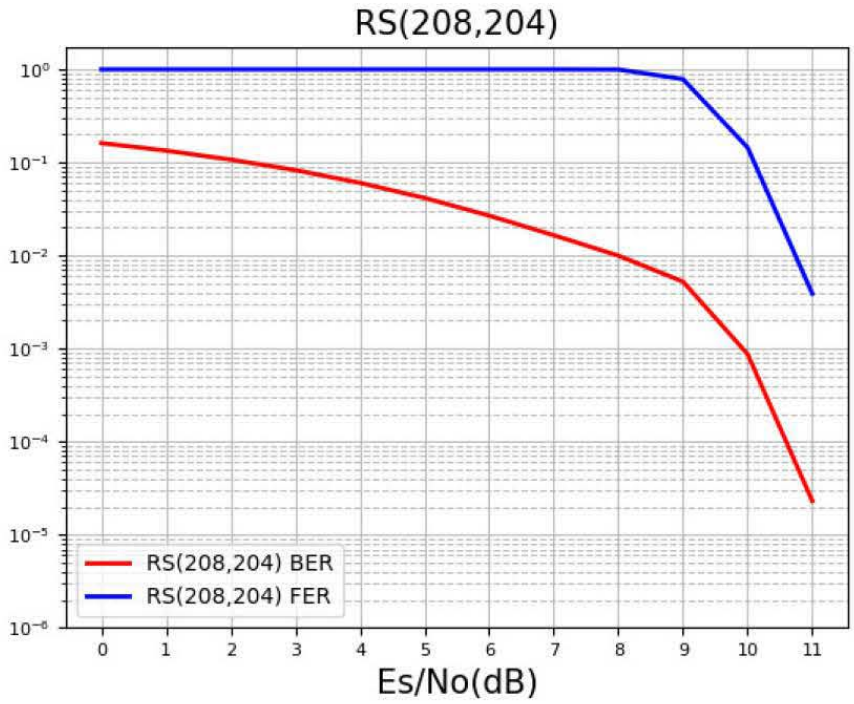


Figure 3.8: Communication performance of RS(208,204) tested on software.

## 3.9 Summary of the Chapter

Error correcting codes with very high code rates are very rare and RS(208,204) is no exception. Therefore, decoding methods, algorithms and architectures for such decoders are not well explored. Our implementation of RS(208,204) explores some original methods. For example, calculating the roots of the error locator polynomial instead of finding them by trial-and-error method is our unique approach. Furthermore, applying checks to avoid worsening the communication performance is another unique approach of ours.

The main reason why these high rate error correcting codes are unexplored is that they have abysmal communication performance as mentioned. However, thanks to such high code rate, RS(208,204) consumes only $8.965mW$, which makes it excellent for using as building block to other decoders rather than deploying it as a stand-alone decoder. Implementation of product RS(208,204) (PRS(208,204)) tries to take advantage of the implementation simplicity of RS(208,204) by using it both in parallel and serially. Chapter 4, explains the implementation of the PRS(208,204) and its parameters.

# Chapter 4

# Implementation of Product RS(208,204)

Product RS(208,204) (PRS(208,204)) decoder uses the algorithms and architectures developed during the implementation of RS(208,204) as a building block and decodes the symbols in an iterative manner. Product structure of PRS(208,204) is shown in Figure 4.1 where small squares represent the symbols. The total of 43264($208 \times 208$) symbols are placed in the PRS structure, 41616($204 \times 204$) of those symbols represents data and the rest are overhead symbols.

PRS(208,204) works in an iterative manner as mentioned. First, we fill the product structure with 43264 symbols and calculate the syndromes for both vertical and horizontal code words. These syndrome values are stored, then we start decoding vertically from the calculation of error locator polynomial step. After the vertical decoding is performed, both vertical and horizontal syndromes are updated according to changed symbols. After vertical decoding, horizontal decoding is performed. Then consecutive vertical and horizontal decoding is performed two more times, making it total of three times. PRS(208,204) does not utilize pipelining; thus, new input is only taken after the output is out of the decoder.

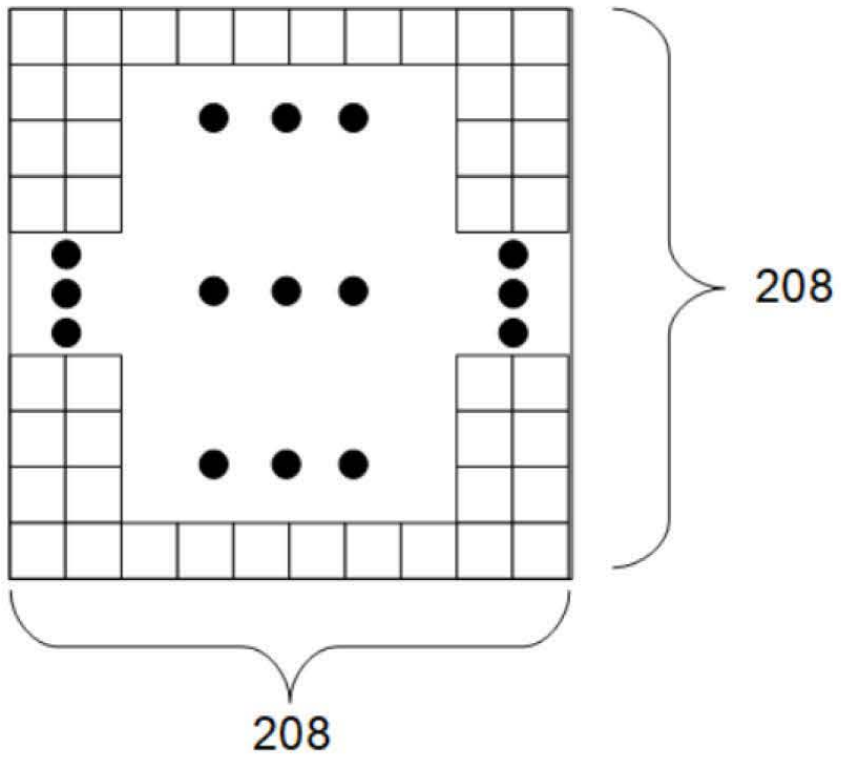


Figure 4.1: Product Structure of PRS(208,204)

PRS(208,204) is implemented using the 28nm TSMC library, running at 200 MHz frequency. The total latency of the decoder is 64 CCs, at every 64 CCs $332,928(204 \times 204 \times 8)$ information bits are decoded. When decoder runs at 200 MHz, net throughput of the decoder is 1.040 Tb/s satisfying the 1 Tb/s condition.

Implementation methodology and steps are explained in the following sections.

## 4.1 Syndrome Calculation

The first step of PRS(208,204) is syndrome calculation, same as RS(208,204). However, in this instance, instead of calculating the syndrome vector for a single decoder, 416 syndrome vectors are calculated for 416 decoders. 208 of them are from vertical decoders while 208 them are from horizontal ones. These syndrome values for all of the decoders are stored in registers for later use.

We use the same syndrome implementation with folded architecture, which we also used for RS(208,204) and explained in Section 3.2. Latency of the syndrome calculation is 10 for PRS(208,204), as well. Therefore, the complexity and area of the block increased 416 times.

In a regular RS decoder, if all of the syndrome values are equal to zero, the received signal is a valid codeword and it is given to the output without further calculation. The same approach applies to PRS decoder as well. However, PRS decoder having all zero syndrome values is highly unlikely especially when the channel SNR is low. Therefore, optimization should prioritize the cases where some of RS decoders inside the PRS having zero syndromes. If a syndrome value of a single RS decoder inside the PRS is equal to zero, we consider these vertical or horizontal symbols as correct and we froze those symbols, which means during the iterations, those symbols cannot be changed. With the frozen and non-frozen symbols, we start calculation of the following blocks in an iterative manner, which will be explained in Section 4.2.

## 4.2 Iterations Block

The calculation of the error polynomial, calculation of the roots of the error polynomial, calculation of the error locations, error evaluation, correction and syndrome update are components of a single iteration block of PRS(208,204) decoder. These processes are performed 6 times in total during PRS, starting with vertical decoders and rotating between the vertical and horizontal decoders.

Algorithms and architectures of each block are the same as RS(208,204) decoder except for the syndrome update block. Therefore, only the syndrome update block is explained for PRS(208,204) decoder.

### 4.2.1 Syndrome Update

For RS decoder, the reason for deploying syndrome is to see if the resulting signal is a valid codeword. The same reason applies for PRS decoders, as well. However, the syndrome update in PRS has a more important role. Thanks to the syndrome update block, syndrome values after each iteration can be calculated in a power and area efficient way. Thus, in this way, syndromes are calculated only at the start of the decoder and syndrome calculation is not part of the iterative circuitry. Syndrome does not have to be calculated but only updated after each iteration, which saves time and energy.

The syndrome update of PRS is much more complex than the syndrome update of RS codes. Although algorithm and methodology for updating the syndrome values are the same; updates should be performed for both vertical and horizontal decoders and they have to be performed simultaneously, which is the reason syndrome update of PRS is more complex. A simple example of $10 \times 10$ PRS decoder is demonstrated in Figure 4.2. After the horizontal iteration, erroneous symbols were found for RS decoders at 5th and 8th rows. Decoder at 5th row had erroneous symbols at 4th and 8th symbols while the decoder at 8th row had erroneous symbols at 2nd and 8th symbols, shown with blue on Figure 4.2. The syndromes of the 4th and 8th horizontal decoders are updated according to the methodology explained in Section 3.6.3. Moreover, the of the 2nd, 4th and 8th vertical decoders should also be updated with the same methodology because some of their symbols are also changed after the horizontal iteration. This simple method of updating the syndromes instead of re-calculating them from the beginning allows PRS to become much simpler in implementation terms.

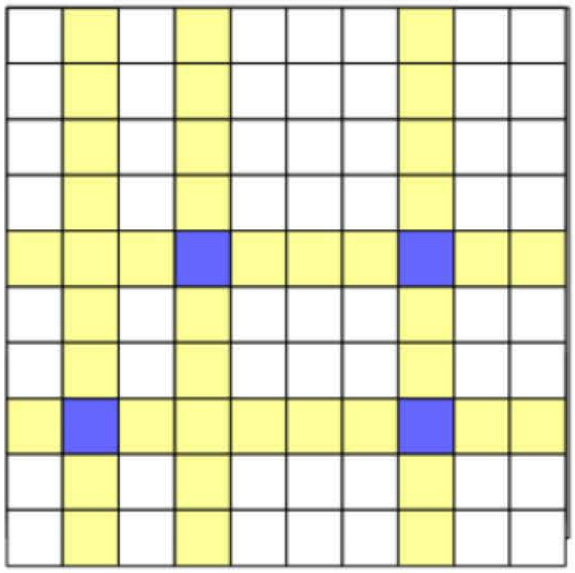

Figure 4.2: Example for syndrome update for PRS

## 4.3 Implementation Results

Implementation of the PRS(208,204) carried out using the Genus tool and $28nm$ library of TSMC ($tcbn28hpcbwp12t30p140ssg0p72v125c_ccs$). Decoder runs at 200 MHz and puts out 1040.4 Gb/s net throughput. The total power consumption of the decoder is 6.245 W, which is not ideal; however, an acceptable power consumption considering the size of the design. Other important parameters are given in Table 4.1.

## 4.4 Communication Performance

Measuring the communication performance of PRS via Monte Carlo simulations is a very hard task due to the complex structure and high throughput demand of PRS. Even the state-of-the-art FPGAs do not have enough look-up tables (LUT) to run PRS and testing system together. However, there is an existing work [1],

Table 4.1: Implementation Results of PRS(208,204)

| | |
|---|---|
| Technology ($nm$) | 28 |
| Throughput ($Gb/s$) | 1040.400 |
| Area ($mm^2$) | 13.341 |
| Power ($W$) | 6.245 |
| Area Eff. ($Gb/s/mm^2$) | 77.985 |
| Pow. Den. ($W/mm^2$) | 0.468 |
| Energy Eff. ($pJ/bit$) | 6.002 |
| Frequency ($MHz$) | 200 |

which analyzes the performance of PRS codes when they are concatenated with polar decoders [28] where polar decoders are inner and PRS decoders are outer decoders with an interleaver in between. [1] shows that polar successive cancellation list of 4 decoder with 512 block length and 416 data length concatenated with PRS(208,204) achieves up to 11.5 dB coding gain at $10^{-15}$ BER compared to uncoded transmission. Polar decoder can also run at 1 Tb/s rates [29]. The work also examines other concatenation schemes that involve other PRS type decoders.

As another communication performance parameter, minimum distance of PRS(208,204) is equal to 9.

## 4.5 Summary of the Chapter

In this chapter, the design and architectures of PRS(208,204) decoder is explained and its ASIC implementation results are shown.

Using an out-of-the-box very high rate decoder RS(208,204), we build PRS(208,204) decoder, which is the final product of this thesis. An iterative PRS(208,204) decoder can provide 1.04 Tb/s net throughput while spending around 6 W of power when the synthesis of the decoder is performed using the Genus tool of Cadence using the 28 nm TSMC library.

# Chapter 5

# Conclusion

A decoder for product Reed-Solomon codes has been designed and implemented in ASIC, which provides more than 1 Tb/s net throughput. The decoder is developed with the use-case of fiber optical communications in mind. Therefore, the requirements of the decoder are chosen accordingly: high throughput, good communication performance and low power usage.

In Chapter 4, the main work of the thesis is presented. In order to implement this rather complicated design, bottom to top design methodology is utilized. Firstly, the building blocks for operations in GF are developed and tested. Using these operation in GF, RS(208,204) is developed. Due to the unusually high code rate of RS(208,204), some novel algorithms and architectures, that are better suited for high rate RS decoders, are developed and used. An automated Python script is coded, which codes RS decoder in VHDL when the necessary parameters are given; such script reduces the workload enormously in case some parameters of the decoder changes. Using RS(208,204) as a component code, PRS(208,204) is developed and implemented on 28nm ASIC. PRS decoder can provide 1 Tb/s net throughput with 6 pJ/bit energy efficiency. When PRS(208,204) is used as a outer code to the polar decoder, analytical work shows that 11.5 dB coding gain at $10^{-15}$ BER is achieved.

Using RS decoder as an outer decoder allows us to take advantage of the correlation between the output bits of the polar decoder. Both, architecture of the successive cancellation polar decoder and product decoder, are well structured and ordered. Therefore, they are rather easy to implement even if the decoder is massive, like fiber optical decoders. These two advantages make polar-PRS concatenated decoder a candidate pair for using in fiber optical communications. Their energy efficiency and communication performance combination is good enough to be in the same league with the current state-of-the-art fiber optical decoders.

As a future work, clock line could be improved. It is possible to shot down the parts of the decoder when corresponding syndrome values of that part is equal to zero. However, meddling with the clock line is a meticulous work and requires expertise and time. Considering the expected power gain of such improvement is not big, we preferred not to commit such time to it. Using BCH decoder in a product structure instead of RS decoder is another possibility that should be considered and examined. Using BCH decoder loses the advantage of the correlation between the output bits of polar decoder; however, the simplicity of BCH decoder compared to RS decoder might be advantageous.

# Bibliography


[1] Y. Ertugrul, "Polar Reed-Solomon concatenated codes for optical communications," Master's thesis, Bilkent University, July 2020.

[2] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, Jun 1960.

[3] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, pp. 68–79, Mar 1960.

[4] R. E. Blahut, *Theory and Practice of Error Control Codes*. Addison Wesley, 1984.

[5] E. Berlekamp, "Bit-serial reed - solomon encoders," *IEEE Transactions on Information Theory*, vol. 28, pp. 869–874, Nov 1982.

[6] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, pp. 122–127, Jan 1969.

[7] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A method for solving key equation for decoding Goppa Codes," *Information and Control*, vol. 1, pp. 87–89, 1975.

[8] L. R. Welch and E. R. Berlekamp, "Error correction for algebraic block codes," *United States Patent*, 1986.

[9] H. M. Shao, T. K. Truong, L. J. Deutsch, J. H. Yuen, and I. S.Reed, "A VLSI design of a pipeline reed-solomon decoder," *IEEE Transactions on Computers*, vol. C-34, pp. 393–403, May 1985.

[10] D. Sarwate and N. Shanbhag, "High-speed architectures for Reed-Solomon decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 641–655, Oct 2001.

[11] Y. Wu, "New Scalable Decoder Architectures for Reed–Solomon Codes," *IEEE Transactions on Communications*, vol. 63, pp. 2741–2761, Aug 2015.

[12] R. Chien, "Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 10, pp. 357–363, Oct 1964.

[13] G. Forney, "On decoding BCH codes," *IEEE Transactions on Information Theory*, vol. 11, pp. 549–557, Oct 1965.

[14] ITU, "Forward error correction for submarine systems," *ITU-T G.975*, 2000.

[15] ITU, "Forward error correction for high bit-rate dwdm submarine systems," *ITU-T G.975.1*, 2004.

[16] L. Seungbeom, L. Hanho, C. Chang-Seok, S. Jongyoon, and K. Je-Soo, "40-gb/s two-parallel reed-solomon based forward error correction architecture for optical communications," *2008 IEEE Asia Pacific Conference on Circuits and Systems, Macao*, Nov 2008.

[17] S. Lee and H. Lee, "A high-speed pipelined degree-computationless modified euclidean algorithm architecture for reed-solomon decoders," *IEICE Transactions on fundamentals of Electronics, Communiations, and Computer Sciences*, vol. E91-A, no. 3, pp. 830–835, Mar 2008.

[18] H. Lee, C.-S. Choi, Jongyoon Shin, and Je-Soo Ko, "100-Gb/s three-parallel Reed-Solomon based foward error correction architecture for optical communications," in *2008 International SoC Design Conference*, (Busan, Korea (South)), pp. I–265–I–268, IEEE, Nov 2008.

[19] J.-I. Park, Jewong Yeon, Seung-Jun Yang, and H. Lee, "An ultra high-speed time-multiplexing Reed-Solomon-based FEC architecture," in *2012 International SoC Design Conference (ISOCC)*, (Jeju Island, Korea (South)), pp. 451–454, IEEE, Nov 2012.

[20] W. Ji, W. Zhang, X. Peng, and Z. Liang, "16-channel two-parallel reed-solomon based forward error correction architecture for optical communications," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, (Singapore, Singapore), pp. 239–243, IEEE, Jul 2015.

[21] G. Perrone, J. Valls, V. Torres, and F. M. Garcia-Herrero, "High-Throughput One-Channel RS(255,239) Decoder," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, (Prague), pp. 110–114, IEEE, Aug 2018.

[22] C. Foughstedt and P. Larsson-Edefors, "Energy-efficient high-throughput VLSI architectures for product-like codes," *Journal of Lightwave Technology*, vol. 37, Jan 2019.

[23] C. Foughstedt, A. Sheikhand, A. G. Amat, G. Liva, and P. Larsson-Edefors, "Energy-efficient soft-assisted product decoders," *2019 Optical Fiber Communications Conference and Exhibition*, April 2019.

[24] J. Torres-Jimenez, N. Rangel-Valdez, A. L. Gonzalez-Hernandez, and H. Avila-George, "Construction of logarithm tables for galois fields," *International Journal of Mathematical Education in*, 2010.

[25] P. C., F. P., and R. P., "Efficient multiplier architectures for galois fields gf," *IEEE Transactions on Computers*, vol. 47, pp. 162–170, 1998.

[26] J. Samanta, J. Bhaumik, and S. Barman, "Modified Karatsuba multiplier for key equation solver in RS Code," *Radioelectronics and Communications Systems*, vol. 58, pp. 452–461, Oct 2015.

[27] E. Berlekamp, H. Rumsey, and G. Solomon, "On the solution of algebraic equations over finite fields," *Information and Control*, vol. 10, pp. 553–564, Jun 1967.

[28] E. Arikan, "Channel polarization: A method for constructing capacity-achieving for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, Jul 2009.

[29] A. Süral, E. G. Sezer, E. Kolagasioglu, V. Derudder, and K. Bertrand, "Tb/s polar successive cancellation decoder 16nm ASIC implementation," *CoRR*, vol. abs/2009.09388, 2020.