VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT
CAPSTONE PROJECT

DESIGN AND IMPLEMENT THE FORWARD ERROR CORRECTION ALGORITHM ON FPGA

MAJOR: COMPUTER ENGINEERING

COMMITTEE: **COMPUTER ENGINEERING 06**
SUPERVISOR: **Assoc. Prof. Dr. Pham Quoc Cuong**
REVIEWER: **M. Eng. Pham Kieu Nhat Anh**

STUDENT 1: **Nguyen Nhat Khai - 2111506**
STUDENT 2: **La Thi Kieu Ngan - 2114149**

HO CHI MINH CITY – May 2025

VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT
CAPSTONE PROJECT

DESIGN AND IMPLEMENT THE FORWARD ERROR CORRECTION ALGORITHM ON FPGA

MAJOR: COMPUTER ENGINEERING

COMMITTEE: **COMPUTER ENGINEERING 06**
SUPERVISOR: **Assoc. Prof. Dr. Pham Quoc Cuong**
REVIEWER: **M. Eng. Pham Kieu Nhat Anh**

STUDENT 1: **Nguyen Nhat Khai - 2111506**
STUDENT 2: **La Thi Kieu Ngan - 2114149**

HO CHI MINH CITY – May 2025

ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

KHOA:KH & KT MÁY TÍNH
BỘ MÔN: KỸ THUẬT MÁY TÍNH

NHIỆM VỤ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

Chú ý: Sinh viên phải dán tờ này vào trang nhất của bản thuyết trình

HỌ VÀ TÊN: Lã Thị Kiều Ngân — MSSV: 2114149
HỌ VÀ TÊN: Nguyễn Nhật Khải — MSSV: 2111506
NGÀNH: Kỹ thuật Máy tính — LỚP: MT21KTTN

1. Đầu đề luận văn/ đồ án tốt nghiệp:
Thiết kế và hiện thực giải thuật FEC (Forward error correction) trên FPGA *(Design and implement the FEC algorithm on FPGA)*

2. Nhiệm vụ (yêu cầu về nội dung và số liệu ban đầu):
- Study the FEC algorithm and testing dataset/models (Nghiên cứu giải thuật FCC và các mô hình/tập dữ liệu kiểm thử)
- Design the accelerator architecture for the algorithm (Thiết kế lkiến trúc lõi tăng tốc giải thuật)
- Desgin the SoC system for sender and receiver with the FEC algorithm (Thiết kế hệ thống SoC để gửi và nhận với giải thuật FEC)
- Implement the proposed system (Hiện thực hệ thống)
- Conduct tests and evaluate the system (Kiểm thử và đánh giá)

3. Ngày giao nhiệm vụ: 06/01/2025

4. Ngày hoàn thành nhiệm vụ: 22/5/2025

5. Họ tên giảng viên hướng dẫn: **Phần hướng dẫn:**
1) Phạm Quốc Cường

Nội dung và yêu cầu LVTN/ ĐATN đã được thông qua Bộ môn.

Ngày 06 háng 01 năm 2025

CHỦ NHIỆM BỘ MÔN
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

GIẢNG VIÊN HƯỚNG DẪN CHÍNH
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):
Đơn vị:
Ngày bảo vệ:
Điểm tổng kết:
Nơi lưu trữ LVTN/ĐATN:

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 15 tháng 5 năm 2025

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người hướng dẫn)

1. HỌ VÀ TÊN: Lã Thị Kiều Ngân MSSV: 2114149
Nguyễn Nhật Khải MSSV: 2111506
NGÀNH: Kỹ thuật Máy tính LỚP: MT21KTM

2. Đề tài:
Thiết kế và hiện thực giải thuật FEC (Forward error correction) trên FPGA *(Design and implement the FEC algorithm on FPGA)*

3. Họ tên người hướng dẫn: Phạm Quốc Cường

4. Tổng quát về bản thuyết minh:
Số trang: Số chương:
Số bảng số liệu Số hình vẽ:
Số tài liệu tham khảo: Phần mềm tính toán:
Hiện vật (sản phẩm)

5. Những ưu điểm chính của LV/ ĐATN:
- Sinh viên hoàn thành tốt nhiệm vụ đặt ra
- Hệ thống hoạt động ổn định
- Sinh viên làm việc nghiêm túc, báo cáo đạt yêu cầu
- Các kết quả thử nghiệm là ấn tượng và được trình bày rõ ràng

6. Những thiếu sót chính của LV/ĐATN:
- Báo cáo còn quá dài và chưa tận dụng hết khả năng của thiết bị để đạt được kết quả cao hơn nữa

7. Đề nghị: Được bảo vệ ☒ Bổ sung thêm để bảo vệ ☐ Không được bảo vệ ☐
8. Các câu hỏi SV phải trả lời trước Hội đồng:
a. Đâu là rào cản cho việc gia tăng hiệu suất của hệ thống?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm: 9.5/10

Ký tên (ghi rõ họ tên)

Phạm Quốc Cường

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 13 tháng 5 năm 2025

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người ~~hướng dẫn~~/phản biện)

1. Họ và tên SV: Nguyễn Nhật Khải, Lã Thị Kiều Ngân
   MSSV: 2111506, 2114149 Ngành (chuyên ngành): Kỹ thuật Máy tính
2. Đề tài: Thiết kế và hiện thực giải thuật FEC (Forward error correction) trên FPGA

3. Họ tên người ~~hướng dẫn~~/phản biện: Phạm Kiều Nhật Anh

4. Tổng quát về bản thuyết minh:
   Số trang: 152 Số chương: 7
   Số bảng số liệu: 31 Số hình vẽ: 51
   Số tài liệu tham khảo: 39 Phần mềm tính toán:
   Hiện vật (sản phẩm)
5. Những ưu điểm chính của LV/ ĐATN:
   - Luận văn hoàn thành tốt các yêu cầu được đặt ra
   - Demo hoạt động khá tốt

6. Những thiếu sót chính của LV/ĐATN: (Không)

7. Đề nghị: Được bảo vệ ☑ Bổ sung thêm để bảo vệ ☐ Không được bảo vệ ☐
8. Các câu hỏi SV phải trả lời trước Hội đồng:
a.

b.

c.

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm : 9.2 /10

Ký tên (ghi rõ họ tên)

PHẠM KIỀU NHẬT ANH

This thesis is dedicated to our parents and our instructors at HCMUT.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

COMMITMENT

We declare that the project ***"Design and Implement the Forward Error Correction Algorithm on FPGA"*** is the result of our own research, understanding, and implementation. This project is carried out by two members of our group, under the guidance of Assoc. Prof. Dr. Pham Quoc Cuong. All results in this project are original, carefully calculated, honestly documented, and entirely neither copied nor reused from any other similar project. All references consulted are properly cited and comprehensively listed in the References section.

We take full responsibility for this declaration, as well as for the content of the project. We bear full responsibility before the Faculty of Computer Science and Engineering and the Ho Chi Minh City University of Technology – Vietnam National University, Ho Chi Minh City, for any dishonesty in the information presented in this project.

Sincerely,

Group of Authors

Nguyen Nhat Khai
La Thi Kieu Ngan

ACKNOWLEDGEMENT

First and foremost, we would like to express our deepest gratitude to our supervisor, Assoc. Prof. Dr. Pham Quoc Cuong. He has been there, always providing his heartfelt support and guidance. He has given us invaluable inspiration and suggestions in our quest for knowledge during our time at the university. Without his assistance and dedicated involvement in every step throughout the process, we would have never accomplished the goals set for this project.

Our heartfelt thanks go to the lecturers of the Faculty of Computer Science and Engineering, in particular, and the Ho Chi Minh City University of Technology, VNUHCM, in general, for their constant imparting of knowledge over the past four years. Their encouragement and support have greatly contributed to the completion of this project.

We would like to extend our heartfelt gratitude to the seniors at Marvell Technology Vietnam, who offered their time and expertise without reserve in supporting every detail of our project. Their thoughtful attention became a wellspring of motivation, driving us to strive with purpose and bring the project to completion with care and devotion.

On a personal note, I (Nguyen Nhat Khai) wish to express my profound gratitude to my parents and younger sister, whose unwavering support has been a steady foundation of strength throughout every step of my life. I would also like to extend the warmest thank you to Pham Van Nhat Vu and Nguyen Thanh Hien, friends who have always ensured I had everything to stay focused on the project. Saving the most precious words for last, I dedicate my whole heart to thanking you, Pham Le Bao Han. You have been, are, and will always be the bright moon untouched by worldly dust, silently lit a lone light for me in the midst of an eternal night.

On my side, I (La Thi Kieu Ngan) would like to express my deepest appreciation to my beloved family for their unconditional love, support, and encouragement. Their belief in me has been a powerful force in motivating me to persevere in the face of challenges. I am also incredibly grateful to Nguyen Trong Anh, my best friend, who has always been there to provide unwavering support and offer thoughtful solutions whenever I felt stuck. Their presence has been a source of immense comfort and inspiration throughout this journey.

ABSTRACT

In today's world of technology, data transmission and the assurance of its reliability have become essential concerns in many industries. Among the myriad solutions that have been and are being applied, forward error correction (FEC) in general, and Reed–Solomon (RS) codes in particular, stand out as effective techniques with exceptionally powerful error correction capabilities. This realization illuminated the great significance of FEC and became the driving force behind the start of this project.

Given the importance of a capstone project, this work not only explores the concepts of FEC and RS codes but also brings these algorithms to life on an field-programmable gate array (FPGA) platform. Along with that, a data transmission system will be introduced, which will then integrate the implemented RS code algorithms to demonstrate their error detection and correction capabilities. Through preliminary evaluations, the Kria KV260 Vision AI starter kit, with its part number xck26-sfvc784-2lv, was designated as the platform of the entire project.

This work has successfully covered most of the concepts related to FEC and the techniques used in implementing RS code algorithms. Based on these studies, two models respectively for the noisy channel and the error probability analysis are constructed before identifying $RS(255,239)$ as the foundational algorithm. Subsequent design steps resulted in the RS encoder and decoder, which are capable of handling continuous data and processing up to 128 bits simultaneously. Although some hardware limitations significantly affected the synthesis process, the design still achieved data rates of up to 10 Gbps, greatly surpassing previous works in the same field.

The data transmission system between two Kria KV260 Vision AI starter kits, with the integration of the monitoring website and RS algorithms, is the final mark of this work. The system gathers real raw images from the camera through interaction with the website, allows for the simultaneous display of the original, noisy, and decoded images on a single page, with a response time that remains impressively under 0.5 seconds.

1

INTRODUCTION

The content of this chapter provides an overview of the project, outlining the current trends and advancements in data transmission systems utilizing FEC algorithms. It highlights the significance of these trends in improving data reliability and error correction. Based on this context, a set of objectives and the scope of the project will be defined to guide the execution process.

1.1. PROJECT'S STATEMENT

Nowadays, the demand for efficient and reliable digital data transmission in communication systems has grown significantly due to the emergence of large-scale and high-speed data networks. However, along with the evolution of data networks, errors are increasingly likely to occur due to numerous technical and environmental factors during the process of transmission. Consequently, a primary concern today is controlling transmission errors caused by channel noise so that reliable reproduction of transmitted information can be obtained.

Looking back to 1948, C. E. Shannon, in his landmark paper [1], demonstrated that by using proper techniques, effects of noisy channel can be reduced to any desired level without sacrificing the rate of information transmission. Shannon's work laid the foundation for the theoretical development of FEC. Since then, researchers in this field have continuously developed new theories, encoding and decoding methods, as well as other coding techniques to maximize the reliability of data transmission. Currently, the use of FEC has become an integral part in the design of modern digital communication systems. To address it, prestigious global organizations also defined technical standards that

align with various application types and channel characteristics, as exemplified by Institute of Electrical and Electronics Engineers (IEEE) 802 standards [2].

Behind every FEC implementation lie thoughtful efforts to effectively balance technology capabilities with the system's intended purpose. The choice of implementation, whether in software, microcontrollers, FPGAs, or application-specific integrated circuit (ASIC)s, carries its own unique strengths and compromises, reflects the specific demands of the system. FPGAs, in particular, offer a unique advantage by providing flexibility through their ability to be reconfigured for different tasks. Along with their robust processing capabilities, FPGAs provide the necessary speed to master demanding tasks, ensuring efficient performance across complex systems. These remarkable properties enable rapid adjustments to the system, creating an undeniable competitive edge, especially during the development phase of any application.

Drawing influence from the concepts above, FEC and FPGA have been chosen as the two key pillars that shape the foundation of this project. Additionally, inspired by the legacy of the Voyager program[1], where the RS codes became a classic application, this project selects image transmission over a noisy channel as the central scenario for the demonstration. This project aims to showcase how cutting-edge advancements in FEC and FPGA technology can be applied to overcome real-world challenges, underscoring their critical contribution to achieving reliable data transmission and improving communication systems. This focus ensures that the project remains both grounded in historical significance and relevant to current technological demands.

1.2. PROJECT'S OBJECTIVES

Based on a preliminary research phase into FEC and a general understanding of the project specifics in this field, six objectives have been carefully established for this project, which are presented below:

The first objective of the project is to *gain a comprehensive understanding of FEC and RS codes*. More specifically, this project focuses on studying FEC in its entirety, aiming to provide an overview of FEC, along with its techniques and applications in current practice. In contrast, for RS codes, the project delves into them in depth, exploring their underlying theories, construction, as well as encoding and decoding techniques used in modern applications.

[1] Launched by National Aeronautics and Space Administration (NASA) in 1977, the Voyager program explored the outer planets and is still transmitting data back to Earth from interstellar space. Initially equipped with only basic coding hardware, Voyager was upgraded with a special RS error correction code before reaching Uranus to improve data transmission reliability.

The second objective of the project is to *propose a complete system design* that demonstrates the error detection and correction capabilities of RS codes. The proposed system must ensure the basic properties of a communication system and be feasible within the project's permitted facility conditions.

The third objective of the project is to analyze and propose, based on the researched issues, to *accurately identify the models and algorithms that need to be implemented*. In detail, a system design on the FPGA platform needs to be proposed to demonstrate the error detection and correction capabilities of RS codes. Moreover, identifying a specific RS code to implement requires evaluating various options based on their efficiency as well as practical applicability.

The fourth objective of the project is to *implement the determined RS code algorithms*. The task consists of two internal components, including the design and implementation of an encoder and decoder on an FPGA platform for the specific RS code that has been identified. In this, the focus is on optimizing the design to achieve maximum performance under certain system constraints.

The fifth objective of the project is to *build the complete system design*, incorporating all necessary components and ensuring their proper integration. It involves setting up environments for two kits to communicate with each other, with each kit deploying the final RS code design. In addition, since the data chosen for demonstration are digital images, a simple website must also be implemented to support basic image operations.

The sixth objective of the project is to *evaluate the effectiveness of the implemented design*, with comprehensive evaluations conducted on both the system and the RS code algorithms. Specifically, the system will be evaluated by comparing error rates before and after implementing the RS code algorithms. Meanwhile, the efficiency of the RS code algorithms will be evaluated using specific metrics detailed in the project. Additionally, a software solution will also be implemented for comparison with the aforementioned hardware solution, demonstrating the efficiency of FPGA in performing complex computations.

1.3. PROJECT'S SCOPE

About the *overall goal*, the project focuses on the design and implementation of FEC algorithms, specifically a carefully chosen RS code, optimized for FPGA platforms deployment.

Regarding the *expected outcome*, the project aims to successfully deploy a complete hardware implementation of the RS encoder and decoder within the proposed data transmission system.

In terms of *execution time*, the project includes two phases. The first phase of the project, the computer engineering project, spans 15 weeks, followed by the 15-week capstone project.

With respect to the project's boundaries, the tasks involve designing, implementing, and testing algorithms related to the RS encoding and decoding process. Conversely, techniques in other areas, such as the actual channel, signal processing, and digital image compression, fall beyond the project's scope and will not be implemented.

1.4. PROJECT'S OUTLINE

The project report is structured into seven distinct chapters, including the current one. Each chapter plays a specific role in presenting the various aspects of the project. To aid in the comprehensive understanding, a short overview of each chapter is delineated as follows.

- The first chapter, *Introduction*, outlines the current problems, the obstacles encountered, and provides a brief overview of the project.
- The second chapter, *Background and Related work*, provides an understanding of the theoretical aspects of FEC and RS codes. A brief introduction to current FPGA trends, as well as the general status of studies and works in this field, is also discussed in this chapter.
- The third chapter, *Proposed architecture*, provides an overview of the proposed system architecture, along with the two key models aimed at identifying the implemented RS code.
- The fourth chapter, *Reed–Solomon algorithms design*, is the ultimate work of art of this project, where the design of the $RS(255, 239)$ code unfolds step by step, piece by piece.
- The fifth chapter, *Design verification and Synthesized results*, performs the correctness verification and collects the results obtained after the synthesis step, allowing for comparison with similar works.
- The sixth chapter, *System implementation*, is where the design is deployed on FPGA platform as a real system, carrying out actual missions, and evaluated using realistic metrics.
- The seventh chapter, *Conclusion*, is the final review of achievements, shortcomings, and future aspirations that carry the project forward.

2

BACKGROUND AND RELATED WORK

This chapter delves into the foundational theories, uncovering the essential principles that form the core of this project. More specifically, the first two sections set the stage with FEC and finite fields, naturally guiding the discussion toward RS codes in the third section, the algorithm central to this project. In the fourth section, an overview of FPGAs and the challenges they present in current practice is provided. In the fourth section, an overview of FPGAs and some related issues in current practice is provided, and the chapter concludes with the fifth section, where some powerful and feasible hardware for the project is introduced.

2.1. FORWARD ERROR CORRECTION

The transmission of digital data over a channel was a challenge for humanity in the mid-20th century, when the foundations of digital communication were being laid and technology was still in its infancy. However, in the modern digital era, this is no longer a major challenge, and attention in this field has shifted to how fast, how reliably, and how efficiently that data can be delivered. The historical development of this field shows that the human need for data reliability has consistently advanced faster than the evolution of transmission channels, making the need for technical solutions to address this gap inevitable. Among them, FEC has emerged as one of the best approaches to addressing data reliability, in case errors on the transmission line are significant, or in case propa-

gation delay constraints matter. With the idea of adding some redundant data, which can then be used to detect and correct errors without the need for retransmission, FEC has continuously proven its effectiveness over the decades.

2.1.1. DATA TRANSMISSION SYSTEM

Data transmission, in theory, is the process of transferring data from an information source to a designated destination through a channel. However, this process is far more complex than a straightforward connection. Instead, it involves multiple stages and components, which are designed to counteract noise that may occur on the channel. According to S. Lin and J. Li [3], a typical data transmission system can be represented as Figure 2.1.

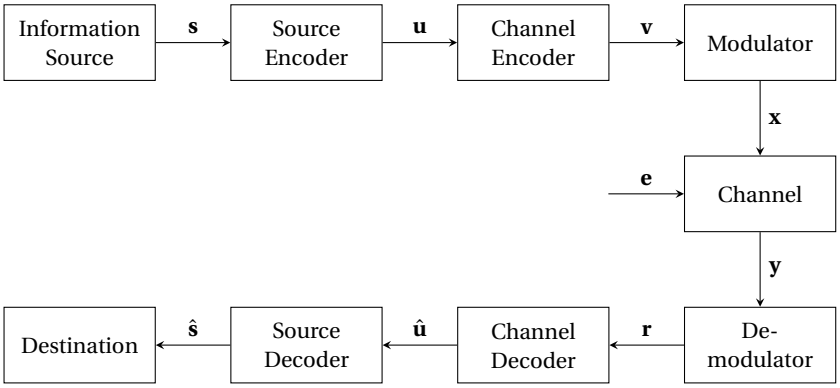


Figure 2.1: A typical data transmission system

In any communication system, there must be an *information source* that is responsible for producing data, called *source information* **s**. Source information may take a variety of different forms, either continuous or discrete. In radio broadcasting, for example, the source information generally is a continuous audio source, such as voice or music. Or, in communication between computers, discrete binary sequences or ASCII characters often considered as source information [4].

However, the original source information is not appropriate for transmitting over the communication channel since it is not only in an unsuitable form but also contains a large amount of redundant data. The *source encoder* performs the task of converting this source information into an *information sequence* **u**, which is a sequence of binary digits. Another function of the source encoder is to minimize the amount of data required to represent the source information, in ways that it can be used to accurately reconstruct the original data.

The *channel encoder*, in its turn, transforms the information sequence into a discrete encoded sequence **v**. This transformation process, known as encoding, is exactly the process of adding redundant data to combat the channel's noise. Since the characteristics of channel errors vary not only in channel properties but also in terms of environmental factors, encoding techniques today are also very diverse. The design and implementation of channel encoder is one of the major topics of this project.

The encoded sequence at the output of the channel encoder is still not suitable for transmission, since it is still discrete binary digits in general. The *modulator* is responsible for converting these binary digits into a waveform of specific duration, which is suitable for transmission over a physical medium, such as combining them with a high-frequency carrier signal, for example.

The *channel*, in practical terms, is a system of physical medium through which information is transferred. Examples of such channels include copper wires, optical fibers, wireless communication channels, or even buses within a computer. These channels, of course, are subjected to various types of noise **e**, which can cause data at the destination no longer the same as the information source. However, collecting realistic error behavior from such a channel is not feasible for this project, so it will be modeled as a probabilistic device.

After traveling through the channel, data will be passed through the *demodulator*, *channel decoder*, and *source decoder* in sequence before reaching its *destination*. Each of these blocks performs the reverse function of the modulator, channel encoder, and source encoder, respectively. In particular, the demodulator processes the received waveform and produces a received sequence **r** of discrete binary digits. Then, the channel decoder takes the received sequence and attempts to reconstruct the information sequence. This process is referred to as decoding, and its output binary sequence $\hat{\mathbf{u}}$, called estimated information sequence. Finally, the source decoder is responsible for recovering the estimated source information $\hat{\mathbf{s}}$ using estimated information sequence from the channel decoder. Ideally, the estimated information sequence and estimated source information are exact replicas of the information sequence and source information, respectively, regardless of any possible errors on the channel.

In order to focus on FEC algorithms, which are implemented in the channel encoder and decoder, the data transmission system described in Figure 2.1 can be simplified into the system shown in Figure 2.2. Firstly, components upstream of the channel encoder will be considered as the only digital source (or generator), and the simplification techniques within the source encoder will also be ignored. Secondly, components between channel encoder and decoder

will be combined with the channel, which is implemented as a probabilistic device, as mentioned earlier. Finally, components downstream of channel decoder will be considered as the only digital sink (or monitor), and it is also the destination of transmitted data.

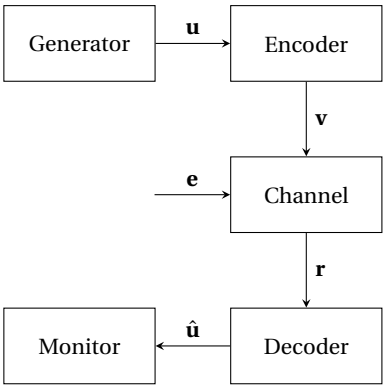


Figure 2.2: A simplified data transmission system

2.1.2. ERROR CHARACTERISTICS AND CHANNEL MODELS

Generally, the characteristic of channel errors can be specified based on their memory effects. For the first type, noise independently affects each transmitted bit meaning each bit has a probability p of being received incorrectly and a probability $1 - p$ of being received correctly. Such memoryless channels can be considered as *random-error channels*, and the reliability of channels completely depend on the error probability p.

However, that is not the typical error behavior of channels, especially when the transmission medium has complex effects. Or, in the case of other components on the physical medium having memory properties, which is common in today's technologies, it will also make the effects of noise no longer independent. One of the most basic and commonly used models for simulating such channels is shown in Figure 2.3, which is the *two-state Gilbert–Elliott model*, developed by E. N. Gilbert [5] and E. O. Elliott [6].

In this model, there are two distinct states: S_1, a good state with a low probability of errors in transmission, and S_2, a bad state where such errors occur more frequently, or in other words, $p_1 \ll p_2$. Most of the time, the channel is in the good state S_1, which means $q_{11} \approx 1$. However, it occasionally switches to bad state S_2 due to changes in the channel's transmission characteristics. And, because the probability q_{22} is also relatively high, the channel tends to remain in state S_2 for a certain period of time afterwards, causing transmission errors

often occur in clusters or bursts. Such channels with memory are called *burst-error channels*, and the two-state Gilbert–Elliott model is also one of the best ways to model such channels. Additionally, error behaviors on the channel can also be a combination of random and burst errors, and such channels are often considered *compound channels*.

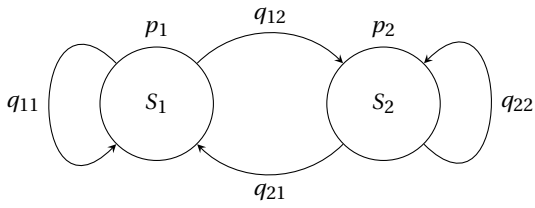


Figure 2.3: The two-state Gilbert–Elliott model

In a more complex scenario, channel errors can make it too difficult to determine whether a bit is 0 or 1. In such cases, the receiver may decide to erase these bits rather than make an uncertain decision, causing the received sequence to lose information at specific, known locations. Such channels are called *binary erasure channels*, and are further subdivided into random and burst erasure, similar to the previously described error channels. Within the scope of this project, the erasure channels are only mentioned to provide a comprehensive view and are not discussed further.

2.1.3. ERROR-CORRECTING CODES

In order to deal with channel errors that can vary from characteristics to probability, a series of error-correcting code (EEC)s have emerged. Basically, each EEC consists of two algorithms, corresponding to the encoding and decoding processes.

Firstly, in the encoding process, the information sequence is often divided into blocks with the same, specified length k. Such a block is called a *message* and can be represented by a binary k-tuple $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ with $u_i \in \{0,1\}, 0 \leq i < k$. And, after redundant bits are added by the encoder, it becomes a binary n-tuple $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ with $v_i \in \{0,1\}$, $0 \leq i < n$, called a *codeword*. Figure 2.4 provides a visual representation of a simple codeword structure. Note that since block codes typically work with messages and codewords, $\mathbf{u}$ and $\mathbf{v}$ are commonly used to denote them, respectively, instead of the entire sequence. And the encoding process, of course, is tied to adding redun-

dant data to the message, so $k < n$ and the ratio $R = k/n$ is referred to as the *code rate*, which can be interpreted as the number of information bits compared to the total codeword length. And the trade-off is that, as the ratio R increases, the efficiency of data transmission improves, but the efficiency of error correction may decrease. Therefore, finding a balance point is crucial for EECs in general.

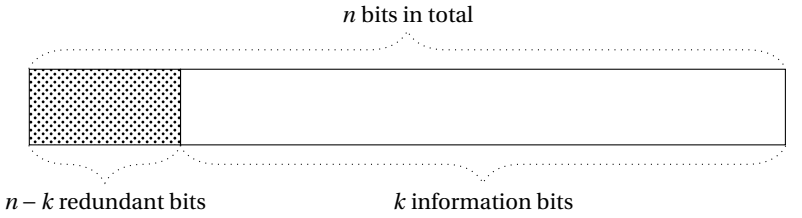


Figure 2.4: A simple codeword structure

Additionally, if the encoding process is a one-to-one mapping from a message to a codeword, or a codeword depends solely on its corresponding message at the same time unit, in other words, the EEC will be referred to as a *block code*. As of now, a large number of block codes have been developed and proven effective in practical applications, with *Golay codes*, *Hamming codes*, *Bose–Chaudhuri–Hocquenghem (BCH) codes*, and RS codes being the most typical examples. Conversely, if a codeword depends not only on the messages of the same time unit but also on messages of the previous time unit, this EEC is called a *convolutional code*. *Turbo codes* can be regarded as the most solid example of the development of convolutional codes. However, this does not mean that convolutional codes will always be more efficient. The use of messages from previous time units requires storing this data, and the computation is, of course, much more complex than that of block codes. The reality is that both block codes and convolutional codes have their place in specific applications, and are evolving together. For this project, block codes will be the only topic covered.

On the other hand, the decoding process, in the most basic scenario, accepts the received codeword as a binary n-tuple, meaning its data as completely quantized into two levels: 0 and 1. The decoder in this case is known as a *hard-decision* decoder. In another scenario, the received codeword is quantized into more than two levels, or even not quantized at all, the decoder in this case must process these discrete values and is referred to as a *soft-decision* decoder. Given the same code, soft-decision decoding is likely to be more accurate compared to hard-decision decoding because it makes use of more information. However, it comes, of course, at the cost of a larger computational load, resulting in

more complex decoding, higher latency, as well as higher power consumption, and so on. Although the continued advancement of fabrication technology has significantly reduced these trade-offs, hard-decision decoding is still desirable in communication systems that require fast decoding process and low power consumption.

Focusing on the hard-decision decoding process, its goal is to reconstruct the estimated message using the received codeword. One of the most popular and efficient approaches is *maximum likelihood decoding*, where the decoder chooses an estimated $\hat{\mathbf{v}}$ from the set of 2^k possible codewords such that the difference between $\mathbf{r}$ and $\hat{\mathbf{v}}$ is minimal, or in other words, the number of differing bits between $\mathbf{r}$ and $\hat{\mathbf{v}}$ is the smallest.

To achieve this, two new definitions are introduced, including the *Hamming distance d*, which represents the number of positions where two codewords differ, and the *minimum Hamming distance* $d_{\min}$, which is the smallest distance between any two codewords. Clearly, if the received codeword $\mathbf{r}$ contains fewer than $d_{\min}$ errors, or if $d(\mathbf{r},\mathbf{v}) < d_{\min}$, in other words, it is guaranteed that the error will be detected. Conversely, error detection is no longer guaranteed, as errors can make the received codeword identical to another. According to another perspective, if $d(\mathbf{r},\mathbf{v}) \leq \lfloor (d_{\min}-1)/2 \rfloor$, the maximum likelihood decoding strategy completely can determine the unique $\hat{\mathbf{v}} = \mathbf{v}$. On the contrary, the decoding process neither ensures a unique $\hat{\mathbf{v}}$ nor prevents $\hat{\mathbf{v}} \neq \mathbf{v}$, leaving the outcome completely uncertain.

2.1.4. ADDITIONAL CODING TECHNIQUES

Although EECs have been one of the most effective ways to combat channel noise, they cannot solve the problem on their own when the error behavior becomes too complex. To be more precise, it is necessary to use an EEC with stronger error correction capabilities, which can significantly increase the complexity of the encoding and decoding process. Apart from the construction of new generation EECs, there are other coding techniques that can be useful when dealing with such error behaviors. To provide a comprehensive perspective on modern FEC, this project will cover several well-known techniques, focusing on their underlying concepts and theories.

To begin with, *concatenation* is a powerful technique for constructing long powerful codes from good short codes, enhancing the reliability of information transmission. First introduced in 1965 by G. D. Forney [7], it has since been widely used in the field of communication due to its superiority. One reason for the need of concatenated code is that the error behavior of channels is often

a combination of several simple behaviors, of which compound channel is an example. Meanwhile, each EEC is often effective at correcting a specific error behavior. For example, with Hamming codes, these are random errors, while, in the case of RS codes, these are burst errors. Typically, a concatenated coding system consists of two linear component codes, which are often performed in serial. Such a coding system can be represented as shown in Figure 2.5.

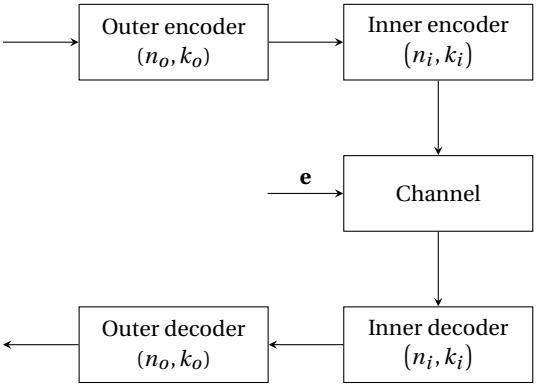


Figure 2.5: A concatenated coding system

Another powerful coding technique that is used more and more today is *interleaving*. Returning to the concept of burst errors in communication systems, noise is often localized in time, meaning it may introduce multiple errors within a small range of transmitted codewords. And, instead of using a strong EEC, which may be wasted due to the majority of transmitted codewords are error free, distributing these errors over codewords in the transmitted sequence is also a good option. If designers can analyze the channel effectively and determine a suitable *interleaving degree*, the number of errors in each deinterleaved vector can be kept small enough for the decoder to correct them using a short, efficient EEC. A simple interleaving strategy with interleaving degree of 2 is described in Figure 2.6.

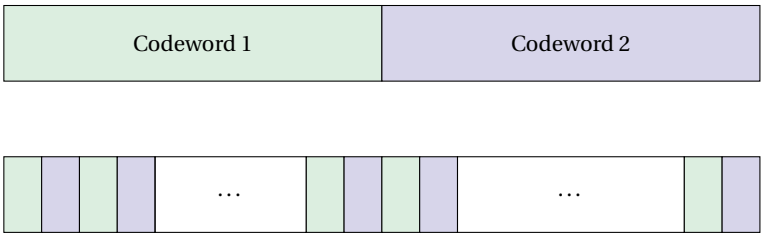


Figure 2.6: An interleaving strategy

2.1.5. PERFORMANCE EVALUATION METRICS

Generally, the performance of a coded system is measured by its *error probability* and its *coding gain* compared to an uncoded system. For the first type of error probability, *block-error probability*, commonly known as block-error rate (BLER), is the probability that the estimated codeword is different from the transmitted codeword. Similarly, *symbol-error probability* and *bit-error probability*, or symbol-error rate (SER) and bit-error rate (BER), are the probability that a decoded information symbol (or bit) is in error. The three parameters provide a comprehensive view of the post-FEC data reliability. However, because of its inherent clarity, BER remains the most popular metric for measurement and evaluation today.

Another widely used metric is coding gain, which is defined as the reduction in signal-to-noise ratio (SNR) required to achieve a specific error probability compared to an uncoded system. Generally, the coding gain can be calculated using Equation 2.1, where $(E_b/N_0)_{\text{uncoded}}$ and $(E_b/N_0)_{\text{coded}}$ are the SNRs required by the uncoded and coded communication system, respectively.

$$\begin{aligned}\zeta &= 10\log_{10}\frac{(E_b/N_0)_{\text{uncoded}}}{(E_b/N_0)_{\text{coded}}} \\ &= 10\log_{10}(E_b/N_0)_{\text{uncoded}} - 10\log_{10}(E_b/N_0)_{\text{coded}}\end{aligned} \tag{2.1}$$

However, obtaining the *SNR-per-bit* E_b/N_0 depends on the specific modulation technique, which is beyond the scope of this project. Thus, the simplest one, which is binary phase shift keying (BPSK) will be used for all calculations and analyses in this study. The corresponding relation relationship between BER and E_b/N_0 [8] is calculated as:

$$\text{BER} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{2.2}$$

where $Q(x)$ is the Q-function (or the tail distribution function) of the standard normal distribution, and its expression is as follows:

$$Q(x) = \frac{1}{\sqrt{2\pi}}\int_x^\infty e^{-u^2/2}\mathrm{d}u \tag{2.3}$$

In summary, a coded communication system should be designed to minimize error probabilities and maximize coding gain, under certain system constraints such as power, bandwidth, as well as encoding and decoding latency.

2.2. FINITE FIELD ARITHMETIC

Across the tapestry of human history, it seems every great advancement is born of a theory, illuminating the path toward its realization. And with FEC, finite fields are critical. First articulated by Évariste Galois, a young French mathematician in the early 19th century, finite fields possessed powerful properties from the start. However, it was only in the mid-20th century that their true potential was realized, particularly in computing and, more specifically, in FEC, where they became crucial in solving problems within a defined set of elements.

2.2.1. GROUP AND FINITE GROUP

Starting with group, one of the foundational concepts of field theory, defined as an algebraic system with an associative binary operation $*$. Let G be a set of elements, it becomes a group under binary operation $*$ if and only if the following conditions are satisfied.

Starting with group, one of the foundational concepts of field theory, defined as an algebraic system with an associative binary operation $*$. Let G be a set of elements, it becomes a group under binary operation $*$ if and only if the following conditions are satisfied.

(1) The binary operation is *associative*, meaning for any a, b, and c in G,

$$a * (b * c) = (a * b) * c$$

(2) G contains a unique *identity element* e, such that for any element a in G,

$$a * e = e * a = a$$

(3) For any element a in G, there exists an unique *inverse* a', such that:

$$a * a' = a' * a = e$$

Note that a group can be considered as a *commutative group* if, for any elements a and b in G, the following property holds:

$$a * b = b * a$$

The number of elements in a group G is called the *order* of the group, denoted by $|G|$. Groups are classified as *finite* or *infinite* based on their order: finite if the order is finite, and infinite otherwise. Binary operations $*$, by their nature, have no way to limit the number of elements, thus specialized groups with specific binary operations are required to maintain a limited number of elements in finite fields. In the context of FEC, the two most well-known finite groups are defined under two binary operations: *modulo-m addition* and *modulo-m multiplication*.

Firstly, consider $G = \{0, 1, \ldots, m-1\}$ is a set of nonnegative integers less than m. The modulo-m addition $\boxplus$ is defined as a binary operation for which Equation 2.4 is satisfied.

$$i \boxplus j = i + j \bmod m \tag{2.4}$$

Clearly, for any $i, j \in G$, $i \boxplus j$ is also a nonnegative integer between 0 and $m-1$. Besides, G is a group under modulo-m addition where: (1) modulo-m addition is associative, as it is derived from real addition; (2) the identity element of G is 0; and (3) for any nonzero element $i \in G$, $m-i$ is also in G and is the unique inverse of i, which is widely known as the *additive inverse* of i.

Secondly, let p is a prime, consider a set of nonzero integers less than p, $G = \{1, \ldots, p-1\}$. The binary operation modulo-m multiplication, represented as $\boxdot$, is defined to fulfill the conditions of Equation 2.5.

$$i \boxdot j = i \cdot j \bmod m \tag{2.5}$$

One can easily demonstrate here that G is a finite group under modulo-m multiplication, with 1 being the identity element. Similar to the additive one, this class of groups is referred to as the *multiplicative group*.

2.2.2. FIELD AND FINITE FIELD

Extending the concept of a group provides access to the definition of a field, an algebraic system with two defined binary operations. More generally, a field is a set of elements where addition, subtraction, multiplication, and division (excluding division by zero) can be performed without leaving the set.

Let F represent a set of elements with two binary operations, called *addition* "+" and *multiplication* "$\cdot$". F qualifies as a field under these operations if and only if it satisfies the following conditions.

(1) F is a commutative group under addition, where the identity element with respect to addition is called the *zero element* (or *additive identity*) of F.

(2) The set $F^* = F \setminus \{0\}$, consisting of the nonzero elements of F, forms a commutative group under multiplication. The identity element of multiplication is called the *unit element* (or *multiplicative identity*) of F.

(3) For any element a, b, and c in F, the *distributive law*, meaning that

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

In order to completely form the four field operations, let $-b$ denote the additive inverse of an element b, and b^{-1} the multiplicative inverse of a nonzero

element b. The two remaining operations, namely *subtraction* "$-$" and *division* "$\div$" or "$/$", can be formed as Equation 2.6 and Equation 2.7, respectively.

$$a - b = a + (-b) \tag{2.6}$$

$$a \div b = a \cdot \left(b^{-1}\right) \tag{2.7}$$

Similar to a group, the number of elements within a field is also referred to as its *order*, and if the number of elements in such a field is finite, it takes the form of what is known as a *finite field*. Furthermore, the smallest integer λ that satisfies Equation 2.8 is known as the *characteristic* of the field F. In this project, the theorem is restated without proof, stating that the characteristic of a finite field must be a prime.

$$\sum_{i=1}^{\lambda} 1 = \underbrace{1 + 1 + \ldots + 1}_{\lambda} = 0 \tag{2.8}$$

Leaving the analytical perspective behind, this project focuses more on the construction of such a finite field. Consider the set $GF(p) = \{0, 1, \ldots, p-1\}$ as the collection of nonnegative integers less than a prime p. As proven in subsection 2.2.1, $GF(p)$ and $GF(p) \setminus \{0\}$ are commutative groups under modulo-p addition and modulo-p multiplication, respectively. Given the definitions of modulo-p addition and multiplication, and the fact that real addition and multiplication obey the distributive law, it follows that modulo-p addition and multiplication also satisfy the distributive law. Therefore, $GF(p)$ is a finite field with p elements under modulo-p addition and multiplication, with the zero element is 0 and the unit element is 1. Since p is a prime, the field $GF(p)$ is often referred to as a *prime field*, as it represents the simplest type of finite field.

In FEC, broadly and in block codes, specifically, prime fields and their concepts hold a profound importance, serving as the mathematical foundation upon which these coding algorithms and techniques are built. One of the most significant prime fields in coding theory is $GF(2)$, also called the *binary field*, as illustrated in Table 2.1. Due to its unique characteristics, $GF(2)$ serves as the sole focus of this project, rather than the broader field $GF(p)$.

$\boxplus$	0	1
0	0	1
1	1	0

(a) Modulo-2 addition

$\boxdot$	0	1
0	0	0
1	0	1

(b) Modulo-2 multiplication

Table 2.1: The finite field under modulo-2 addition and multiplication

2.2.3. POLYNOMIALS OVER BINARY FIELD

Mathematically, a *polynomial* is an expression consisting of variables and coefficients. In coding theory, it often involves a polynomial expressed in the form $f(X) = f_0 + f_1 X^1 + f_2 X^2 + \ldots + f_n X^n$, where n is a specified finite number. Even within a finite field, a polynomial retains its inherent properties, including addition (or subtraction), multiplication, as well as division.

Specifically, let $f(X)$ and $g(X)$ be two polynomials of degrees n and m over $GF(2)$, respectively. The addition of $f(X)$ and $g(X)$ is given by Equation 2.9, where $f_i + g_i$, is carried out in modulo-2 addition.

$$f(X) + g(X) = \sum_{i=0}^{\max\{n,m\}} \left(f_i + g_i\right) X^i \tag{2.9}$$

Besides, the multiplication between $f(X)$ and $g(X)$ is calculated as presented in Equation 2.10, and the $f_i \cdot g_i$, of course, is performed in modulo-2 multiplication.

$$f(X) \cdot g(X) = \sum_{i=0}^{n} \sum_{j=k-i}^{m} \left(f_i \cdot g_j\right) X^k \tag{2.10}$$

Division of two polynomials is more complex, as it cannot be performed in a directly way. Instead, their relationship is expressed in Equation 2.11, where $f(X)$, $g(X)$, $q(X)$, and $r(X)$ are the dividend, divisor, quotient, and remainder, respectively.

$$f(X) = q(X) \cdot g(X) + r(X) \tag{2.11}$$

Currently, although a range of methods for polynomial division is available, the long-division remains the most appropriate for fundamental calculations.

It is important to note that the properties of polynomials described above are applicable not only to the binary field but also to its extensions, which will be later introduced in subsection 2.2.4. In this case, calculations between coefficients must, of course, be performed in the corresponding extension of the binary field.

Finally, in coding theory, there exist polynomials with special properties that require further explanation. An *irreducible* polynomial $p(X)$ of degree m over a finite field is one that is not divisible by any polynomial of degree less than m within the same field. An irreducible polynomial $p(X)$ is said to be *primitive* if the smallest positive integer n for which $p(X)$ divides $X^n + 1$ is $n = 2^m - 1$. A list of widely used primitive polynomials over $GF(2)$, which have the minimum number of terms, is provided in Table 2.2.

Degree	Primitive polynomials	Degree	Primitive polynomials
2	$1+X+X^2$	8	$1+X^2+X^3+X^4+X^8$
3	$1+X+X^3$	9	$1+X^4+X^9$
4	$1+X+X^4$	10	$1+X^3+X^{10}$
5	$1+X^2+X^5$	11	$1+X^2+X^{11}$
6	$1+X+X^6$	12	$1+X+X^4+X^6+X^{12}$
7	$1+X^3+X^7$	13	$1+X+X^3+X^4+X^{13}$

Table 2.2: A list of primitive polynomials over $GF(2)$

2.2.4. EXTENSION OF BINARY FIELD

In the world of computing, where all information is represented using the binary digits 0 and 1, it is often preferable to expand the binary field rather than use a larger $GF(p)$ field. Aligning with the above trend, most existing block codes today are constructed using the binary field $GF(2)$ and its extension field, denoted as $GF(2^m)$, where m is a positive integer.

The construction of an extension field begins with a primitive polynomial $p(X) = p_0 + p_1X^1 + p_2X^2 + \ldots + p_mX^m$ of degree m over $GF(2)$, and an element α, which is a root of $p(X)$, or $p(\alpha) = 0$ in other words. Determine a set of $GF(2) = \{0,1\}$ and powers of α as:

$$F = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^j, \ldots\right\} \tag{2.12}$$

Since α is a root of $p(X)$, which divides $X^{2^m-1}+1$, it follows that α is also a root of $X^{2^m-1}+1$, implying that $\alpha^{2^m-1} = 1$. As a consequence, F consists of exactly m elements, presented as follows:

$$F = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\right\} \tag{2.13}$$

To obtain α, for $0 \leq i < 2^m - 1$, first divide X^i by the primitive polynomial $p(X)$. Using the polynomial division defined in section Equation 2.11, the corresponding expression can be written as:

$$X^i = q_i(X) \cdot p(X) + r_i(X) \tag{2.14}$$

where $q_i(X)$ and $r_i(X)$ are the quotient and and remainder polynomials over the binary field $GF(2)$, respectively. An important note is that the degree of $r_i(X)$ is no greater than $m-1$, since the degree of the divisor in this case is m. Furthermore, it is stated without proof that X and $p(X)$ are relatively prime,

which leads to X^i not being divisible by $p(X)$. Finally, substituting X in Equation 2.14 by α, the resulting relationship, as presented in Equation 2.15, is satisfied for all non-zero elements of F. Such a representation is also known as the polynomial form of α^i.

$$\begin{aligned}\alpha^i &= q_i(\alpha)\cdot p(\alpha) + r_i(\alpha)\\ &= q_i(\alpha)\cdot 0 + r_i(\alpha)\\ &= r_{i,0} + r_{i,1}\alpha + r_{i,2}\alpha^2 + \ldots + r_{i,m-1}\alpha^{m-1}\end{aligned} \tag{2.15}$$

The set F is indeed a finite field of characteristic 2 and order 2^m, since it satisfies the formation conditions of a finite field, which is outlined in subsection 2.2.2. In this project, the above problem will not be proven, but only the properties that can be derived later will be stated.

Let α^i and α^j be two elements in the finite field $GF(2^m)$, generated by the primitive polynomial $p(X)$. The addition over $GF(2^m)$ is defined as shown in Equation 2.16, where $r_{i,k} + r_{j,k}$ is carried out over $GF(2)$. On the other hand, the multiplication between α^i and α^j is defined as in Equation 2.17.

$$\alpha^i + \alpha^j = \left(r_{i,0} + r_{j,0}\right) + \left(r_{i,1} + r_{j,1}\right)\alpha + \ldots + \left(r_{i,m-1} + r_{j,m-1}\right)\alpha^{m-1} \tag{2.16}$$

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

Clearly, starting from α, the entire set of elements can be obtained, and as such, it is referred to as the *primitive element* of $GF(2^m)$. Besides, based on the definition of the additive and multiplicative groups, 0 serves as the zero element, while 1 is defined as the unit element of $GF(2^m)$. Finally, the last two operations over $GF(2^m)$ include additive inverse $-\alpha^i$ and multiplicative inverse α^{-i}, which can be expressed in turn as follows:

$$-\alpha^i = \alpha^i \tag{2.18}$$

$$\alpha^{-i} = \alpha^{2^m-1-i} \tag{2.19}$$

2.3. REED–SOLOMON CODES

Every field, in its rise and evolution, carries the names of distinguished contemporary scientists, with their contributions forever etched in its history. In the case of the EEC, this was no different. In fact, many scientists in this field have given birth to algorithms that are forever entwined with their names. I. S. Reed and G. Solomon [9] are one such case. Introduced in the 1960s, RS codes were swiftly recognized and widely applied due to their exceptional features. After more than 60 years, RS codes remain the core technology in a wide array

of information transmission techniques, playing a crucial role in applications ranging from common connection standards to high-speed data transmission in data centers, and even in the most advanced space exploration efforts.

2.3.1. BASIC DEFINITIONS

Before exploring the structure and properties of RS codes, one must begin by defining the foundational terms and concepts underlying them. Starting with a finite field of order q, the RS code over $GF(q)$ has the following parameters:

- Field's order $q = p^m$, refers to the total number of elements within the constructed field, and thus, q must be a prime power.
- Field's characteristic p, is a prime number and is typically chosen to be 2.
- Field's degree m, is a positive integer and also the degree of the primitive polynomial that constructs the field.

Similar to other block codes, RS codes operate on data blocks of fixed length. However, RS codes do not treat a data block as individual bits but rather as individual elements of a finite field, which is most commonly $GF(2^m)$. This is one of the reasons why RS codes offer significantly better burst error correction capability compared to other block codes with the same code rate.

After completing the construction of the finite field, let $0 < t < q$ be a positive integer less than q, the remaining parameters of the RS code are defined as follows:

- Codeword length $n = q - 1$, represents the total number of symbols within a codeword. Note that the symbol here is exactly an element of the finite field that constructs the RS code. Since a symbol over $GF(2^m)$ includes m bits, there are $m(q-1)$ bits within a codeword in total.
- Message length $k = q - 2t - 1$, indicates the number of information symbols within a codeword, which corresponds to a portion of data that needs to be transmitted.
- Parity length $n - k = 2t$, refers to the number of symbols added by the encoding process, which are essential for combating the channel noise.
- Error-correcting capability t, determines the maximum number of symbol errors that the RS code can effectively correct within a codeword.
- Error-detecting capability $2t$, refers to the maximum number of symbol errors that the RS code can completely detect but is unable to correct within a codeword.

It is noted, without proof, that the minimum Hamming distance of the RS code is $d_{\min} = n - k + 1$, which serves as the basis for determining the RS code parameters, as previously outlined.

In practice, codeword length n and message length k are often used as the two key parameters to characterize each code, denoted as $RS(n,k)$, since, in theory, they contain enough information to determine all the remaining parameters. In less common cases, typically used in specialized studies, the field's order q and error-correcting capability t can also be used to characterize a code, which is specifically called a *q-ary t-symbol-error-correcting* RS code.

2.3.2. ENCODING TECHNIQUES

As of now, more than one approach has been introduced to perform the encoding process for RS codes, each offering distinct methods and advantages depending on the specific application requirements. Rather than providing an exhaustive list of all encoding methods, this project limits itself to discussing only a few, emphasizing those that are the most prominent, including *non-systematic* and *systematic* encoding.

To begin with, the generator polynomial, which is widely regarded as the central component of the encoding process, will be introduced and discussed in detail. Let α be a primitive element of the finite field $GF(q)$, the generator polynomial $g(X)$ is defined as a polynomial whose roots are and exactly are $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$. Since $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ are distinct elements in $GF(q)$, the generator polynomial is expressed in the form shown in Equation 2.20.

$$\begin{aligned} g(X) &= (\alpha + X)\left(\alpha^2 + X\right)\ldots\left(\alpha^{2t} + X\right) \\ &= g_0 + g_1 X + g_2 X^2 + \ldots + g_{2t-1} X^{2t-1} + g_{2t} X^{2t} \end{aligned} \tag{2.20}$$

RS codes generated using such a generator polynomial are called *primitive RS codes*, and this is also the only method for constructing the generator polynomial discussed in this project.

NON-SYSTEMATIC ENCODING

In essence, the encoding process involves finding a codeword $v(X)$ that satisfies the condition of being divides by both the message $u(X)$ and the elements $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$. In other words, $v(X)$ is divides by both $u(X)$ and $g(X)$. This leads to the most direct encoding approach, known as non-systematic encoding, where $v(X)$ is defined as in Equation 2.21.

$$v(X) = u(X)\, g(X) \tag{2.21}$$

SYSTEMATIC ENCODING

From Equation 2.21, it can be seen that the message $u(X)$ has been completely transformed and integrated into the codeword. This leads to the consequence that, at the decoding process, which is much more complex than the encoding one, an additional computational step must be performed to recover the transmitted data. To solve the above problem, a new approach to the encoding process, referred to as systematic encoding, is introduced, using equation Equation 2.22.

$$v(X) = X^{n-k}u(X) + \left(X^{n-k}u(X) \bmod g(X)\right) \tag{2.22}$$

Clearly, by shifting the message part and placing it at the highest coefficients of the codeword, it becomes easy to identify the information part to be transmitted, which is the most common structure of a codeword.

2.3.3. DECODING STEPS

The codeword produced by the encoding process is subsequently transmitted through a noisy channel, with the effect of the noise is demonstrated as:

$$r(X) = v(X) + e(X) \tag{2.23}$$

Note that the polynomial $e(X)$ here is referred to as *error polynomial* (or *error pattern*) and is defined in Equation 2.24, where e_i, $0 \leq i < n$, are elements in $GF(q)$:

$$e(X) = e_0 + e_1X + e_2X^2 + \ldots + e_{n-1}X^{n-1} \tag{2.24}$$

Compared to encoding, the decoding process is a much more complicated process. It consists of several steps that occur sequentially, with each step performing a distinct and essential task that cannot be separated from the others. The decoding process of a q-ary t-symbol-error-correcting RS code can be divided into the following steps:

(1) Calculate the syndrome S.
(2) Find the error-location polynomial $\sigma(X)$.
(3) Determine the error locations β.
(4) Compute the error values δ.
(5) Perform the error correction.

Similar to encoding processes, this project does not seek to list all currently available decoding algorithms. Instead, it presents the most commonly used methods for each step, arranged in the sequence in which they take place.

2.3.4. SYNDROME CALCULATION

At the receiver, neither the transmitted codeword $v(X)$ nor the error pattern $e(X)$, as shown in Equation 2.23, is known. Consequently, upon receiving $r(X)$, the decoder's first task is to determine whether there are any transmission errors are present in $r(X)$. Such a process is called error detection, and to accomplish it, the syndrome calculation step must be performed.

Starting from the property of a codeword mentioned in subsection 2.3.2, which accepts $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ as roots. The syndrome calculation essentially involves checking whether $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ are roots of the received polynomial $r(X)$ or not. For a q-ary t-symbol-error-correcting primitive RS code, the syndrome of a received polynomial $r(X)$ is given by a $2t$-tuple over $GF(q)$ as shown in Equation 2.25, where each of them is calculated using Equation 2.26, with all additions and multiplications carried out in $GF(q)$.

$$\mathbf{S} = (S_1, S_2, \ldots, S_{2t}) \tag{2.25}$$

$$S_i = r\left(\alpha^i\right) = r_0 + r_1\alpha^i + r_2\alpha^{2i} + \ldots + r_{n-1}\alpha^{n-1} \tag{2.26}$$

It is evident that if $S = 0 = (0, 0, \ldots, 0)$, the received polynomial $r(X)$ has all elements $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ as roots, satisfying the property of a codeword. Therefore, it can be assumed that no errors occurred during transmission. Otherwise, $r(X)$ fails to qualify as a codeword, indicating that an error occurred during transmission and necessitating error correction.

It should also be noted that even if $S = 0$, errors may still be present, as they can map one codeword to another. However, according to the Hamming distance definition, this scenario only happens when at least $n - k + 1$ symbols are in error, which is highly improbable.

Here, the syndrome calculation concludes its journey. Yet, to set the stage for what follows, attention turns to a few definitions closely tied to the syndrome.

Initially, assume that the error polynomial contains $v \leq t$ errors, which are located at $X^{j_1}, X^{j_2}, \ldots, X^{j_v}$, with $0 \leq j_1 < j_2 < \ldots < j_v < n$. The error polynomial can be rewritten in the form expressed in Equation 2.27, where $e_{j_1}, e_{j_2}, \ldots, e_{j_v} \in GF(q)$ are the values of errors at $X^{j_1}, X^{j_2}, \ldots, X^{j_v}$, respectively.

$$e(X) = e_{j_1}X^{j_1} + e_{j_2}X^{j_2} + \ldots + e_{j_v}X^{j_v} \tag{2.27}$$

From Equation 2.23, Equation 2.26, Equation 2.27, and the fact that α^i, with $0 \leq i \leq 2t$, are roots of $v(X)$, syndrome calculation for S_i can be rewritten as in Equation 2.28.

$$\begin{aligned} S_i = r\left(\alpha^i\right) = v\left(\alpha^i\right) + e\left(\alpha^i\right) = e\left(\alpha^i\right) \\ = e_{j_1}\alpha^{ij_1} + e_{j_2}\alpha^{ij_2} + \ldots + e_{j_v}\alpha^{ij_v} \end{aligned} \tag{2.28}$$

To make it easier to observe, for $1 \leq l \leq v$, let $\beta_l = \alpha^{j_l}$ and $\delta_l = e_{j_l}$. Equation 2.28 now has the form of Equation 2.29, which is also known as *power-sum symmetric functions*, with elements β_l and δ_l denoted as the *error-location numbers* and *error values*, respectively.

$$S_i = \sum_{l=1}^{v} \delta_l \beta_l^i \tag{2.29}$$

Finally, as can be recognized, Equation 2.25 is written in vector form, which is not ideal for computation and therefore needs to be converted to polynomial form, as seen in Equation 2.30.

$$S(X) = S_1 + S_2 X + \ldots + S_{2t-1} X^{2t-1} \tag{2.30}$$

2.3.5. ERROR-LOCATION POLYNOMIAL

Once it is clear that the received codeword contains errors, the next step is to determine the location and value of the errors, or in other words, to identify all β and δ. As shown in Equation 2.29, there is a strong relationship between β, δ, and the calculated syndrome value. The task now, as one might expect, is to bring them to light.

To achieve this, the error-location numbers will first be determined using a polynomial of the form shown in Equation 2.31, which is widely known as the *error-location polynomial*.

$$\begin{aligned} \sigma(X) = \left(1 + \beta_1 X\right)\left(1 + \beta_2 X\right)\ldots\left(1 + \beta_v X\right) \\ = \sigma_0 + \sigma_1 X + \sigma_2 X^2 + \ldots + \sigma_v X^v \end{aligned} \tag{2.31}$$

However, up to now, no algorithm powerful enough to directly solve the upper form of the error-location polynomial shown in Equation 2.31, and it must instead be solved indirectly through the lower form. This project will present two well-known algorithms, which are most likely to be implemented on FPGA, including the *Berlekamp–Massey algorithm* and the *Euclidean algorithm*.

BERLEKAMP–MASSEY ALGORITHM

Beginning with the Berlekamp–Massey (BM) iterative algorithm with the goal to find the error-location polynomial. This project focuses on explaining the basic concepts of the BM algorithm rather than proving them. Expanding

the product of Equation 2.31, the v equalities in Equation 2.32 can be found, relating the coefficients of the error-location polynomial and the error-location numbers, widely known as *elementary-symmetric functions*.

$$\begin{aligned}
\sigma_0 &= 1 \\
\sigma_1 &= \beta_1 + \beta_2 + \ldots + \beta_v \\
\sigma_2 &= \beta_1\beta_2 + \beta_1\beta_3 + \ldots + \beta_{v-1}\beta_v \\
&\vdots \\
\sigma_v &= \beta_1\beta_2\ldots\beta_v
\end{aligned} \tag{2.32}$$

Afterward, taking the error-location number part Equation 2.29 and combining it with Equation 2.32, the $2t$ identities in Equation 2.33 are derived and are referred to as *Newton's identities*. Additionally, note that the last $2t-v$ equalities of Newton's identities are also known as *generalized Newton's identities*.

$$\begin{aligned}
\sigma_0 S_1 + \sigma_1 &= 0 \\
\sigma_0 S_2 + \sigma_1 S_1 + 2\sigma_2 &= 0 \\
&\vdots \\
\sigma_0 S_v + \sigma_1 S_{v-1} + \sigma_2 S_{v-2} + \ldots + \sigma_{v-1} S_1 + v\sigma_v &= 0 \\
\sigma_0 S_{v+1} + \sigma_1 S_v + \sigma_2 S_{v-1} + \ldots + \sigma_v S_1 &= 0 \\
&\vdots \\
\sigma_0 S_{2t} + \sigma_1 S_{2t-1} + \sigma_2 S_{2t-2} + \ldots + \sigma_v S_{2t-v} &= 0
\end{aligned} \tag{2.33}$$

Building on the idea of finding a minimum-degree polynomial whose coefficients satisfy the $2t$ Newton's identities, E. R. Berlekamp [10] and J. L. Massey [11] developed an algorithm, now named after them, to solve the above problem. The algorithm demonstrates that error-location polynomial $\sigma(X)$ can be computed iteratively in $2t$ steps, with the minimum-degree polynomial at the μth step, $0 \le \mu \le 2t$, whose coefficients satisfy the first μ Newton's identities.

$$\sigma^{(\mu)}(X) = \sigma_0^{(\mu)} + \sigma_1^{(\mu)} X + \sigma_2^{(\mu)} X^2 + \ldots + \sigma_{l^{(\mu)}}^{(\mu)} X^{l^{(\mu)}} \tag{2.34}$$

To prepare for the next minimum-degree polynomial, it is necessary to first determine whether the current minimum-degree polynomial satisfies the first $\mu+1$ Newton's identities. This can be determined through the μth *discrepancy*, which can be calculated as expressed in Equation 2.35.

$$d^{(\mu)} = \sigma_0^{(\mu)} S_{\mu+1} + \sigma_1^{(\mu)} S_\mu + \sigma_2^{(\mu)} S_{\mu-1} + \ldots + \sigma_{l^{(\mu)}}^{(\mu)} S_{\mu+1-l^{(\mu)}} \tag{2.35}$$

In the case of $d^{(\mu)} = 0$, the current minimum-degree polynomial is determined to satisfy the first $\mu+1$ Newton's identities, making it the minimum-degree polynomial for the next iteration as well.

$$\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) \tag{2.36}$$

In the opposite case, specifically when $d^{(\mu)} \neq 0$, a new minimum-degree polynomial must be determined in the following sequence. Return to the steps preceding the μth step and identify a step ρ where the discrepancy $d^{(\rho)} \neq 0$, and $\rho - l^{(\rho)}$ has the largest value, with $l^{(\rho)}$ representing the degree of the minimum-degree polynomial at the ρth step. In this context, Equation 2.37 presents the resulting minimum-degree polynomial for the $(\mu+1)$th step, with its degree determined using Equation 2.38.

$$\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d^{(\mu)}/d^{(\rho)} X^{\mu-\rho} \cdot \sigma^{(\rho)}(X) \tag{2.37}$$

$$l^{(\mu+1)} = \max\left\{l^{(\mu)}, \mu - \rho + l^{(\rho)}\right\} \tag{2.38}$$

Repeat the above correction process until reaching the $2t$ step, at which point the final error-correction polynomial will be reached, as shown below:

$$\sigma(X) = \sigma^{(2t)}(X) \tag{2.39}$$

Additionally, it is important to note that the initial condition for the iterative process is provided in Equation 2.40, serving as starting point for computations.

$$\begin{aligned} \sigma^{(-1)}(X) = 1, \quad l^{(-1)} = 0, \quad d^{(-1)} = 1, \\ \sigma^{(0)}(X) = 1, \quad l^{(0)} = 0. \end{aligned} \tag{2.40}$$

INVERSIONLESS BERLEKAMP–MASSEY ALGORITHM

Although the BM algorithm presented earlier is nearly complete and perfect, challenges may arise when implementing the original algorithm using hardware description language (HDL) in general. The main issue is that, at certain steps, the algorithm needs to revisit previous steps to retrieve data, which requires the corresponding hardware design to incorporate a large number of memory elements along with a complex mechanism to manage and utilize this data correctly. An effective way to address this issue is by using the *inversionless Berlekamp–Massey algorithm*.

Consider the original BM algorithm, if $d^{(\mu)} \neq 0$ at the μth step, it is necessary to return to a designated previous step ρ to retrieve $d^{(\rho)}$ and $\sigma^{(\rho)}(X)$. The construction ideas of the inversionless Berlekamp–Massey (IBM) algorithm are opposite to those of the original one. This means that, at each step μ, it auto-

matically checks whether it will become step ρ in the future and preemptively saves the two parameters, $d^{(\rho)}$ and $\sigma^{(\rho)}(X)$, for future use if the conditions are satisfied. The following is a simple proof that demonstrates the relationship between BM and IBM algorithms.

Start by initializing three additional parameters which include *auxiliary error-location polynomial* $B(X)$, *auxiliary discrepancy* γ, and auxiliary error-location polynomial's degree l_B. In terms of meaning, $B(X)$ and γ are used to store values of the error-location polynomial and discrepancy, at step μ when it is determined that it will become step ρ in the future. Meanwhile, l_B, corresponding degree of the polynomial, is a redundant parameter used only to determine the algorithm's condition and not sent to the output. At the beginning of the calculation sequence, initial values for these parameters are:

$$B^{(0)}(X) = 1, \quad \gamma^{(0)} = 1, \quad l_B^{(0)} = 0. \tag{2.41}$$

It is easy to predict that success or failure of the IBM algorithm's construction depends on the ability to determine if step μ will become step ρ in the future. Keep in mind that step $\rho < \mu$ is determined based on two factors, including $d^{(\rho)} \neq 0$ and $\rho - l^{(\rho)}$ reaching the maximum value. Therefore, the two corresponding conditions for determining whether step μ will become step ρ in the future are $d^{(\mu)} \neq 0$ and $\mu - l^{(\mu)} > \mu' - l^{(\mu')}$ for all step $\mu' < \mu$.

Without losing generality, assume that $l_B^{(\mu)}$ represents the degree of the auxiliary error-location polynomial at step $\mu' < \mu$, where $\mu' - l_B^{(\mu)}$ is currently the maximum value. Clearly, $\mu - l^{(\mu)}$ becomes the maximum value if and only if $\mu - l^{(\mu)} > \mu' - l_B^{(\mu)}$. Through mathematical analysis, the following expression can be obtained:

$$\begin{aligned} \mu - l^{(\mu)} > \mu' - l_B^{(\mu)} &\Leftrightarrow l^{(\mu)} < l_B^{(\mu)} + \mu - \mu' \\ &\Rightarrow l^{(\mu)} \leq l_B^{(\mu)} \end{aligned} \tag{2.42}$$

Summing up, the two condition for determining whether step μ will become step ρ in the future are $d^{(\mu)} \neq 0$ and $l^{(\mu)} \leq l_B^{(\mu)}$, with the corresponding algorithm provided in pseudo code as follows:

Input: $S = (S_1, S_2, \ldots, S_{2t})$.
Initialize: $\sigma^{(0)}(X) = 1$, $B^{(0)}(X) = 1$, $\gamma^{(0)} = 1$, $l^{(0)} = 0$, $l_B^{(0)} = 0$.
for $\mu = 0, 1, 2, \ldots, 2t-1$ **do**
  Calculate: $d^{(\mu)} = \sum_{i=0}^{l^{(\mu)}} \sigma_i^{(\mu)} \cdot S_{\mu-i+1}$.
  Update: $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d^{(\mu)}/\gamma^{(\mu)} X \cdot B^{(\mu)}(X)$.

if $d^{(\mu)} \neq 0$ **and** $l^{(\mu)} \leq l_B^{(\mu)}$ **then**
Update: $B^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$, $\gamma^{(\mu+1)} = d^{(\mu)}$.
Update: $l^{(\mu+1)} = l_B^{(\mu)} + 1$, $l_B^{(\mu+1)} = l^{(\mu)}$.
else
Update: $B^{(\mu+1)}(X) = X \cdot B^{(\mu)}(X)$, $\gamma^{(\mu+1)} = \gamma^{(\mu)}$.
Update: $l^{(\mu+1)} = l^{(\mu)}$, $l_B^{(\mu+1)} = l_B^{(\mu)} + 1$.
end-if
end-for
Output: $\sigma^{(2t)}(X)$, $B^{(2t)}(X)$, $\gamma^{(2t)}$, $l^{(2t)}$.

Algorithm 2.1: Inversionless Berlekamp–Massey algorithm

Note that while updating the error-location polynomial $\sigma^{(\mu+1)}(X)$, $d^{(\mu)} = 0$ and $d^{(\mu)} \neq 0$ are not computed separately as in the original algorithm. This is because when $d^{(\mu)} = 0$, the second addend always results in 0, leading to the fact that $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$.

EUCLIDEAN ALGORITHM

In addition to the BM algorithm for finding error-location polynomial in decoding RS codes, another effective method is based on the Euclidean algorithm for finding the greatest common divisor (GCD) of two polynomials over the same field [12]. Below, this project presents a simple but effective proof of the algorithm's fundamental concept.

Looking back at the error-location polynomial $\sigma(X)$ in Equation 2.31 and the syndrome polynomial $S(X)$ in Equation 2.30, it becomes evident that coefficients of their multiplication can be expressed as in Equation 2.43. Here, $Z(X)$ represents the error-value evaluator, a polynomial with a maximum degree of $\nu - 1$, which is discussed in detail in subsection 2.3.7.

$$\begin{aligned} Z(X) &= \left(S_1 + S_2 X + \ldots + S_{2t-1} X^{2t-1}\right) \\ &\quad \cdot \left(\sigma_0 + \sigma_1 X + \sigma_2 X^2 + \ldots + \sigma_\nu X^\nu\right) \\ &= \sigma_0 S_1 + (\sigma_0 S_2 + \sigma_1 S_1) X + (\sigma_0 S_3 + \sigma_1 S_2 + \sigma_2 S_1) X^2 \\ &\quad + \ldots + (\sigma_0 S_\nu + \sigma_1 S_{\nu-1} + \ldots + \sigma_{\nu-1} S_1) X^{\nu-1} \end{aligned} \tag{2.43}$$

Undoubtedly, the first ν terms of $\sigma(X) S(X)$ provide the error-value evaluator $Z(X)$, while the next $2t - \nu$ terms give the generalized Newton's identities as expressed in Equation 2.33. Mathematically, the three polynomials $\sigma(X)$, $S(X)$, and $Z(X)$ are connected by the relationship shown in Equation 2.44, widely recognized as the *key-equation*.

$$Z(X) \equiv \sigma(X) S(X) \pmod{X^{2t}} \tag{2.44}$$

According to Y. Sugiyama and colleagues, the key equation can be represented in the form of Equation 2.45, which is suitable for implementing the Euclidean algorithm to simultaneously find the error-location polynomial $\sigma(X)$ and the error-value evaluator $Z(X)$.

$$Z(X) = \tau(X) X^{2t} + \sigma(X) S(X) \tag{2.45}$$

At the ith iteration, where $i \leq 1$, the computation can be performed using the formulas in Equation 2.46. Note that here, $q^{(i)}(X)$ represents the quotient at the ith step, which is the result of dividing $Z^{(i-2)}(X)$ by $Z^{(i-1)}(X)$. Also, since the goal of the algorithm is to find $\sigma(X)$ and $Z(X)$, $\tau^{(i)}(X)$ can be omitted.

$$\begin{aligned} Z^{(i)}(X) &= Z^{(i-2)}(X) + q^{(i)}(X) Z^{(i-1)}(X) \\ \sigma^{(i)}(X) &= \sigma^{(i-2)}(X) + q^{(i)}(X) \sigma^{(i-1)}(X) \\ \tau^{(i)}(X) &= \tau^{(i-2)}(X) + q^{(i)}(X) \tau^{(i-1)}(X) \end{aligned} \tag{2.46}$$

Finally, the initial values and stop condition of the algorithm are presented in Equation 2.47 and Equation 2.48, respectively.

$$\begin{aligned} Z^{(-1)}(X) &= X^{2t}, \quad \sigma^{(-1)}(X) = 0, \quad \tau^{(-1)}(X) = 1, \\ Z^{(0)}(X) &= S(X), \quad \sigma^{(0)}(X) = 1, \quad \tau^{(0)}(X) = 0. \end{aligned} \tag{2.47}$$

$$\deg\left(Z^{(\rho)}(X)\right) < \deg\left(\sigma^{(\rho)}(X)\right) \leq t \tag{2.48}$$

2.3.6. ERROR-LOCATION DETERMINATION

As mentioned earlier, neither BM, IBM nor Euclidean algorithms can directly determine the error-location numbers β_i but are limited to computing the coefficients σ_i, where $0 \leq i \leq v$, of the error-location polynomial. As a result, in general, this step involves determining all roots of a v-degree equation whose coefficients are in $GF(q)$.

Nowadays, it is widely recognized that there are numerous methods to solve a v-degree equation. However, since the work is done with a finite field, containing a predetermined number of elements, this process can be simplified to simply checking each element to determine whether it is a root. Such a process was generalized by R. Chien [13] in his research.

Clearly, from Equation 2.31, if $\beta_1^{-1}, \beta_2^{-1}, \ldots, \beta_v^{-1} \in GF(q)$ are roots of $\sigma(X)$, then $\beta_1, \beta_2, \ldots, \beta_v$ are error-location numbers, which indicates that the received symbol at the corresponding locations is erroneous. Mathematically, for all

values of i such that $0 \leq i < q-1$, the expression shown in Equation 2.49 must be evaluated and its result must be checked for equality to 0.

$$\sigma\left(\alpha^i\right) = \sigma_0 + \sigma_1\alpha^i + \sigma_2\left(\alpha^i\right)^2 + \ldots + \sigma_v\left(\alpha^i\right)^v \tag{2.49}$$

Moreover, at this step, if the number of received roots does not align with the degree of error-location polynomial, it signifies that the process of error-location determination has faltered. The cause of this phenomenon, naturally, lies in the fact that number of errors in the received codeword has surpassed the error-correction capacity of the RS code. In such a case, one may choose to accept the received codeword as it stands, or proceed with further steps to modify it into a codeword that can contain more errors. Each approach carries its own merits, and the choice should be guided by the specific application, taking into account the error-detection and error-correction capabilities of other algorithms that may be present within the system.

2.3.7. ERROR-VALUE EVALUATION

Once the task of determining the error-location numbers has been successfully completed, the remaining task, which follows naturally, is to determine the error values. To date, *Forney algorithm* [14] and *Horiguchi–Koetter algorithm* [15] continues to stand as the most effective methods, with a brief proof provided below.

FORNEY ALGORITHM

From $S(X)$ introduced in Equation 2.30, $\sigma(X)$ in Equation 2.31, and the key-equation shown in Equation 2.44, the form of *error-value evaluator* $Z(X)$ is proved as follows:

$$\begin{aligned}
Z(X) = \sigma(X)\,S(X) &= \prod_{i=1}^{v}\left(1-\beta_i X\right)\cdot\sum_{j=1}^{\infty} S_j X^{j-1} \\
&= \prod_{i=1}^{v}\left(1-\beta_i X\right)\cdot\sum_{j=1}^{\infty} X^{j-1}\sum_{l=1}^{v}\delta_l\beta_l^j \\
&= \prod_{i=1}^{v}\left(1-\beta_i X\right)\cdot\sum_{l=1}^{v}\delta_l\beta_l\sum_{j=1}^{\infty} X^{j-1}\beta_l^{j-1} \\
&= \prod_{i=1}^{v}\left(1-\beta_i X\right)\cdot\sum_{l=1}^{v}\frac{\delta_l\beta_l}{1-\beta_l X} \\
&= \sum_{l=1}^{v}\delta_l\beta_l\prod_{i=1,i\neq l}^{v}\left(1-\beta_i X\right)
\end{aligned} \tag{2.50}$$

Afterward, taking the derivative of the error-location polynomial $\sigma(X)$ given by Equation 2.31, the calculation process is illustrated in Equation 2.51.

$$\sigma'(X) = \frac{\mathrm{d}}{\mathrm{d}X}\prod_{i=1}^{\nu}\left(1-\beta_i X\right) = \sum_{l=1}^{\nu}\beta_l \prod_{i=1,i\neq l}^{\nu}\left(1-\beta_i X\right) \tag{2.51}$$

For $1 \leq k \leq \nu$, by substituting the variable X in the Equation 2.50 and Equation 2.51 with β_k^{-1}, Equation 2.52 follows, illuminating the way to uncover the error-values.

$$\frac{Z\left(\beta_k^{-1}\right)}{\sigma'\left(\beta_k^{-1}\right)} = \frac{\delta_k\beta_k \prod_{i=1,i\neq l}^{\nu}\left(1-\beta_i\beta_k^{-1}\right)}{\beta_k \prod_{i=1,i\neq l}^{\nu}\left(1-\beta_i\beta_k^{-1}\right)} = \delta_k \tag{2.52}$$

The Forney algorithm presented above is the best continuation of the Euclidean algorithm since the Euclidean algorithm produces error-location polynomial and error-value evaluator simultaneously. However, before reaching this step, it is mandatory for the BM algorithm to find the error-value evaluator itself through Equation 2.44 because its output solely provides the error-location polynomial. Besides error-location polynomial, the IBM algorithm also produces a pair of auxiliary parameters, which might be useful for the process of error-value evaluation. What enables this is the Horiguchi–Koetter algorithm, which builds directly upon the Forney algorithm.

HORIGUCHI–KOETTER ALGORITHM

To start, define the two polynomials, $\omega(X)$ as the *evaluator polynomial*, and $A(X)$ as the *scratch poynomial*, in accordance with Equation 2.53 and Equation 2.54, respectively.

$$\omega^{(\mu)}(X) \equiv \sigma^{(\mu)}(X)\,S(X) \pmod{X^{\mu}} \tag{2.53}$$

$$A^{(\mu)}(X) \equiv B^{(\mu)}(X)\,S(X) - X^{\mu} \pmod{X^{\mu+1}} \tag{2.54}$$

The goal of proving Horiguchi–Koetter algorithm is to clarify the correctness of Equation 2.55, for all $0 \leq \mu \leq 2t$. Once achieved, Forney algorithm can be employed to simplify it into a more practical computational form. Before diving deeper into the expression, the algorithm's validity will first be established.

$$\sigma^{(\mu)}(X)\,A^{(\mu)}(X) - \omega^{(\mu)}(X)\,B^{(\mu)}(X) = \gamma^{(\mu)}X^{\mu-1} \tag{2.55}$$

Let $c \in \{0,1\}$ be the condition that step μ will become step ρ in future, the IBM algorithm can be rewritten in matrix form as in Equation 2.56. Note further

that the calculation of values for $\omega^{(\mu+1)}(X)$ and $A^{(\mu)}(X)$, based on $\omega^{(\mu)}(X)$ and $A^{(\mu)}(X)$ respectively, is not the focus of this project and is adopted from the research results of R. E. Blahut [15].

$$\begin{bmatrix} \sigma^{(\mu+1)}(X) & \omega^{(\mu+1)}(X) \\ B^{(\mu+1)}(X) & A^{(\mu+1)}(X) \end{bmatrix} = \begin{bmatrix} 1 & d^{(\mu)}/\gamma^{(\mu)}X \\ c & (1-c)X \end{bmatrix} \begin{bmatrix} \sigma^{(\mu)}(X) & \omega^{(\mu)}(X) \\ B^{(\mu)}(X) & A^{(\mu)}(X) \end{bmatrix} \tag{2.56}$$

Besides, the initial conditions for proving process include $\gamma^{(0)} = 1$, and:

$$\begin{bmatrix} \sigma^{(0)}(X) & \omega^{(0)}(X) \\ B^{(0)}(X) & A^{(0)}(X) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & X^{-1} \end{bmatrix} \tag{2.57}$$

In preparing for the proof by induction, at $\mu = 0$, Equation 2.55 becomes:

$$\begin{aligned} \sigma^{(0)}(X) A^{(0)}(X) - \omega^{(0)}(X) B^{(0)}(X) &= 1 \times X^{-1} - 0 \times 1 \\ &= X^{-1} = \gamma^{(0)} X^{-1} \end{aligned} \tag{2.58}$$

Building on this, assuming Equation 2.55 holds for $\mu = k$, prove that it is also satisfied for $\mu = k+1$ as follows:

$$\begin{aligned} & \sigma^{(k+1)}(X) A^{(k+1)}(X) - \omega^{(k+1)}(X) B^{(k+1)}(X) \\ &= \begin{vmatrix} \sigma^{(k+1)}(X) & \omega^{(k+1)}(X) \\ B^{(k+1)}(X) & A^{(k+1)}(X) \end{vmatrix} = \begin{vmatrix} 1 & d^{(k)}/\gamma^{(k)}X \\ c & (1-c)X \end{vmatrix} \begin{vmatrix} \sigma^{(k)}(X) & \omega^{(k)}(X) \\ B^{(k)}(X) & A^{(k)}(X) \end{vmatrix} \\ &= \left[(1-c)X - d^{(k)}/\gamma^{(k)}Xc\right] \left[\sigma^{(k)}(X) A^{(k)}(X) - \omega^{(k)}(X) B^{(k)}(X)\right] \\ &= \left[(1-c) - d^{(k)}/\gamma^{(k)}c\right] X \cdot \gamma^{(k)} X^{k-1} = \gamma^{(k+1)} X^{k} \end{aligned} \tag{2.59}$$

At this point, Equation 2.55 can be concluded. With $\mu = 2t$, clearly, $\omega^{(2t)}(X)$ and $Z(X)$ are identical. By combining this with Equation 2.52 and the fact that $\sigma^{(2t)}\left(\beta_k^{-1}\right)$ for all $1 \le k \le \upsilon$, the final result obtained in Equation 2.60.

$$\delta_k = \frac{\omega^{(2t)}\left(\beta_k^{-1}\right)}{\sigma'^{(2t)}\left(\beta_k^{-1}\right)} = \frac{\gamma^{(2t)} \beta_k^{-2t+1}}{B^{(2t)}\left(\beta_k^{-1}\right) \sigma'^{(2t)}\left(\beta_k^{-1}\right)} \tag{2.60}$$

2.3.8. ERROR CORRECTION

After successfully determining all the error-location numbers and error values, the final step involves applying Equation 2.61 to uncover the estimated codeword. From there, by discarding the redundant data portion, a fully estimated message emerges, complete and ready.

$$\hat{v}(X) = r(X) + \hat{e}(X) \tag{2.61}$$

2.4. FPGA-BASED ARCHITECTURE

In modern computing, microprocessors and microcontrollers are typically used to handle a wide range of tasks. While these platforms are highly versatile and suitable for general-purpose applications, they are often not optimized for specific, high-performance tasks. This limitation has led to the development of specialized hardware solutions like ASICs and FPGAs. Unlike ASICs, which are expensive to design and fixed in functionality, FPGAs provide a more cost-effective and flexible alternative. They can be reprogrammed to meet the changing requirements of a given task, offering the adaptability needed for applications that demand customization, high speed and low latency. As a result, FPGAs are particularly well-suited for implementing specialized functions, such as error correction algorithms in communication networks.

2.4.1. FIELD-PROGRAMMABLE GATE ARRAY

The FPGA industry began to take shape in the 1980s, emerged from programmable read-only memory (PROM) and programmable logic device (PLD)s, which laid the foundation for the reconfigurable and versatile capabilities of modern FPGAs. Generally, an FPGA [16] is an integrated circuit which can be hardware-programmed using HDLs, such as Verilog, SystemVerilog, or very high-speed integrated circuit hardware description language (VHDL). A typical FPGA architecture, which contains thousands of configurable logic gates that can be interconnected to perform a wide range of logic functions as illustrated in Figure 2.7, is composed of three key components.

For the first ones, *input/output buffer (IOB)s* act as interfaces between the FPGA and external devices, handling voltage level conversion, signal conditioning, and protocol support. They ensure proper communication with other components, such as processors, peripherals, as well as sensors, by adapting to different communication protocols specific to the device or system.

Secondly, *configurable logic block (CLB)s* form the core of the FPGA's logic processing, which contain various components, including combinational logic such as look-up table (LUT)s, multiplexers, and logic gates, as well as sequential logic, typically flip-flop (FF)s. Typical of their operations, LUTs store the truth table of Boolean functions in static random-access memory (SRAM), allowing the FPGA to implement any combinational logic, while multiplexers route data within the block, enabling complex logic functions and arithmetic operations. On the other hand, FF store state information for sequential logic, allowing the FPGA to perform time-dependent tasks such as memory storage, counters, and finite state machines, and especially enabling synchronous design.

Not to be overlooked, *programmable interconnects*, implemented through switch matrix (SM), connect the CLBs and other FPGA components, enabling flexible data routing. The programmable routing resources allow the FPGA to adapt its interconnections based on the design, providing the flexibility to implement a wide range of circuits and functions.

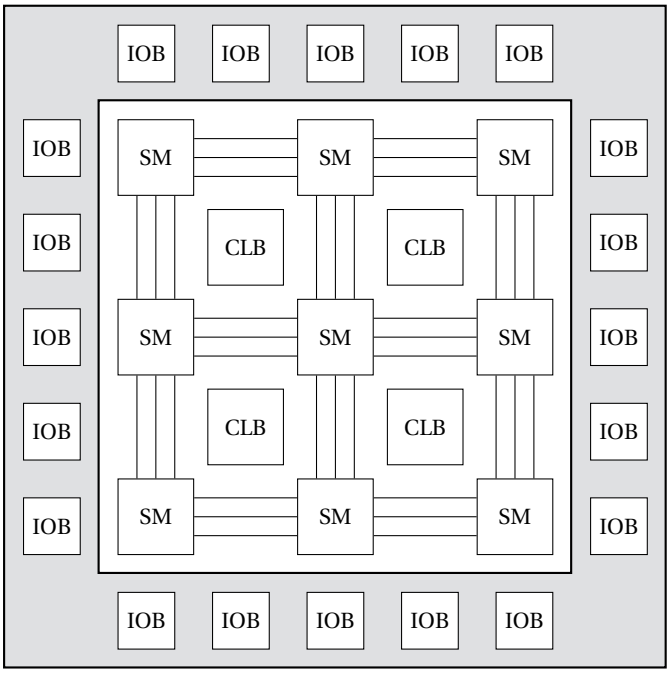


Figure 2.7: Basic FPGA architecture

Ultimately, seamlessly integrating these elements enables the FPGA to dynamically adapt to various design requirements. This synergy ensures optimal performance and customization for diverse applications, securing FPGAs a prominent role in today's technology landscape.

2.4.2. SYSTEM-ON-CHIP

Although FPGAs allow rapid prototyping, enabling ideas to be implemented in silicon with minimal development time, they cannot be the only component within the system due to their specific limitations. As hardware systems grow increasingly complex, integrating components such as memory, microprocessors, and programmable logic has become crucial. To meet these demands, system-on-chip (SoC) technology was developed to consolidate diverse hardware resources into a single chip.

Xilinx further advanced this concept with the introduction of multiprocessor system-on-chip (MPSoC), which integrates programmable logic and a dedicated processor on a single chip, connected through advanced extensible interface (AXI)s. As shown in Figure 2.8, MPSoC are divided into two main regions known as programmable logic (PL) and processing system (PS). The PL, functioning as an FPGA, excels at parallel processing, while the PS, typically an advanced reduced instruction set computing machine (ARM) Cortex-A53 processor, is optimized for sequential, general-purpose tasks. This combination enables an efficient co-design workflow, blending hardware and software development seamlessly [17].

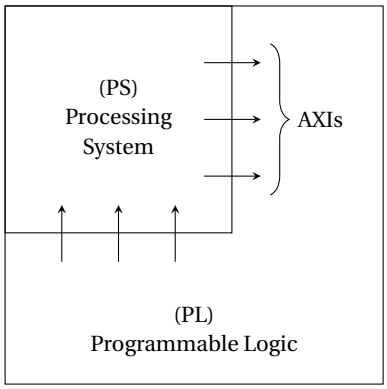


Figure 2.8: Basic MPSoC architecture

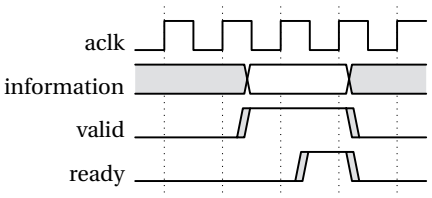
2.4.3. AXI4 PROTOCOL

AXI [18] is part of ARM's advanced microcontroller bus architecture (AMBA) suite, a set of interconnect protocols designed for high-speed, on-chip data transfer. AXI4 is the latest version of the AXI protocol and is utilized in modern embedded systems, including the Kria system-on-module (SoM) used in this project, to connect the PS with custom logic in the PL part of the system. AXI4 offers a flexible and high-bandwidth interface, enabling parallel handling of multiple transactions, which enhances performance in applications that require intensive data processing. In AXI4, there are five key channels that facilitate different aspects of data communication:

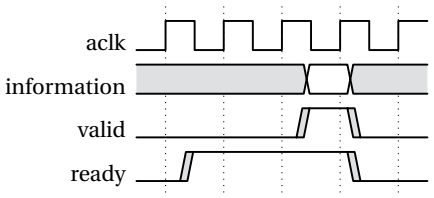
- *Read address channel* (AR) involves the process where the master sends the address of the data to be read from the slave.
- *Read Data Channel* (R) refers to the process of sending requested data from slave to master through the read data channel after receiving the address.

- *Write Address Channel* (AW) operates as the pathway for the master sending the address where the data should be written to the slave.
- *Write Data Channel* (W) is used by the master to send write data to the slave through this channel.
- *Write Response Channel* (B) is where the slave sends a response back to the master to indicates the completion of a write operation, providing status information.

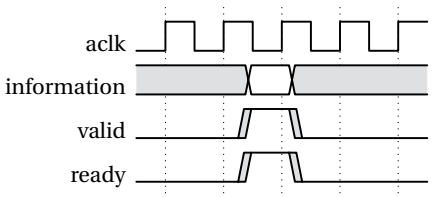
All five channels in the AXI4 protocol use the same *valid*/*ready* handshake mechanism to transfer both data and control information. This bidirectional flow control system allows the master and slave devices to manage the rate at which data and control signals are transferred. The source generates the *valid* signal to indicate that data or control information is ready, while the destination generates the *ready* signal to indicate its readiness to accept the data. The transfer only occurs when both the *valid* and *ready* signals are high. Figure 2.9a, Figure 2.9b, and Figure 2.9c show three examples of handshake sequences, showing cases where the *ready* and *valid* signals are either not asserted simultaneously or are asserted simultaneously.



(a) *ready* active before *valid*



(b) *valid* active before *ready*



(c) *valid* active with *ready*

Figure 2.9: Handshake sequences

AXI4 READ PROCESS

For any communication protocol, reading and writing are always the two most basic and important operations, forming the foundation for data transfer.

AXI4 is no exception, as these two processes serve as the basis for communication between PS and PL. As a starting point, the read process is first examined in detail. Generally, the channel architecture of such a read process can be depicted in Figure 2.10, with two channels consisting of read address and read data between the master and slave interfaces.

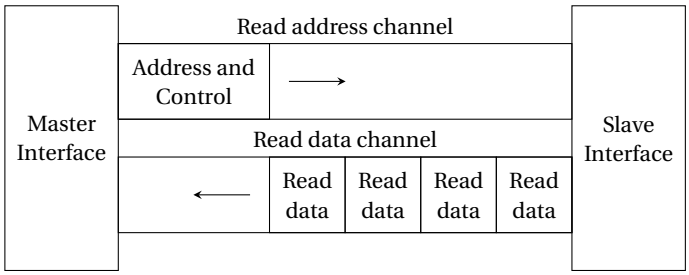


Figure 2.10: Channel architecture of read process

Figure 2.11 shows an example of a read burst of four transfers. To put it specifically, at the beginning of the read process, the master asserts the *arvalid* signal along with the read address *araddr* and related signals through the read address channel to indicate that it has valid control information. This signal remains asserted until the slave acknowledges it by asserting the corresponding *arready* signal.

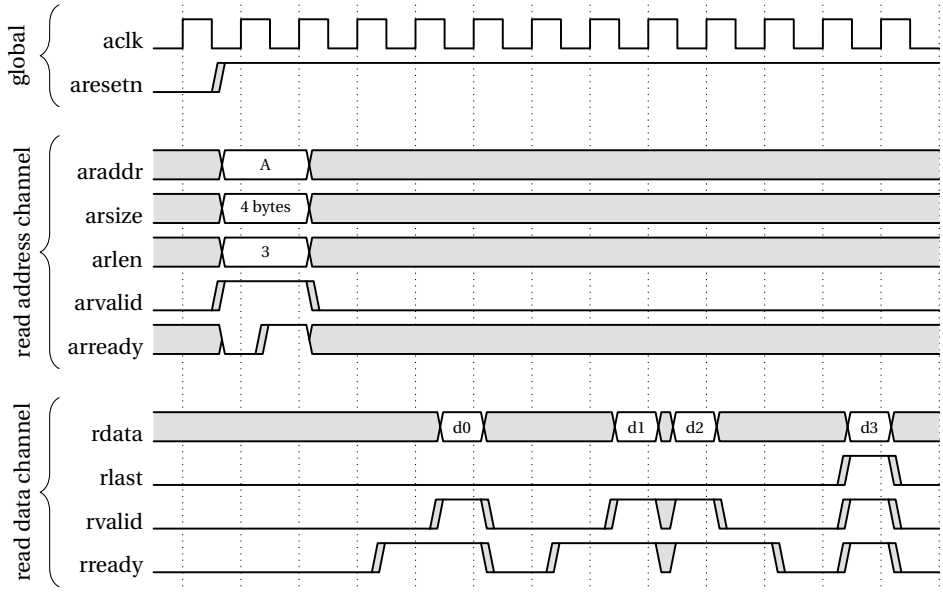


Figure 2.11: Example of read process

Once the slave receives a valid address, it asserts the *rvalid* signal to indicate that valid read data is available. The *rvalid* signal must remain asserted until the master acknowledges the data by asserting the *rready* signal. While *rvalid* is asserted, the slave transmits the read data. The slave waits for the master to assert *rready* before proceeding to the next piece of data.

This handshake continues until the final data item in the sequence. Upon the last data transfer, the slave asserts the *rlast* signal to indicate that it is the last data item being transmitted. This marks the completion of the read transfer. For multi-beat transactions, the *rlast* signal ensures that the master recognizes the end of the entire read sequence.

AXI4 WRITE PROCESS

Compared to the read process, the write process is more complicated because, in addition to two channels, write address and write data, there is also the appearance of the write response channel. The addition is necessary because both the write address and write data channels flow from master to slave, and the presence of the write response channel ensures communication occurs in both directions. The channel architecture of write process can be presented as shown in Figure 2.12.

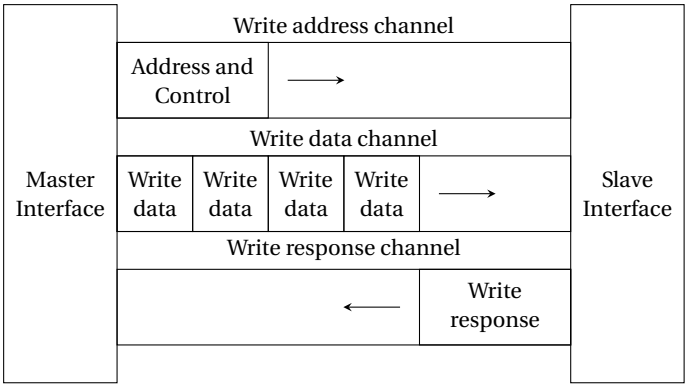


Figure 2.12: Channel architecture of write process

An example of a write burst, consisting of four transfers in a sequential order, is shown in Figure 2.13. At the beginning, the master asserts the *awvalid* signal along with the write address *awaddr* and related signals through the write address channel to indicate that it has valid control information. This signal remains asserted until the slave acknowledges it by asserting the corresponding *awready* signal. Once *awready* is asserted, the address handshake is completed, and the write address phase ends.

During the write data phase, the master sends the write data to the slave through the write data channel. The master asserts the *wvalid* signal to indicate that the data on the *wdata* signal is valid. This signal remains high until the slave acknowledges it by asserting the *wready* signal. The master continues this handshake for each beat of the burst. For the last data item, the master asserts the *wlast* signal to indicate that it is the final data transfer in the burst. The *wlast* signal ensures the slave recognizes the end of the write data phase.

Finally, during the write response phase, the slave uses the write response channel to indicate the status of the write transaction. It asserts the *bvalid* signal to signal that a valid response is available. The master acknowledges this by asserting the *bready* signal. Once the slave detects *bready*, it deasserts *bvalid*, completing the write transaction. This handshake ensures that the master and slave are synchronized and that the write process is concluded successfully.

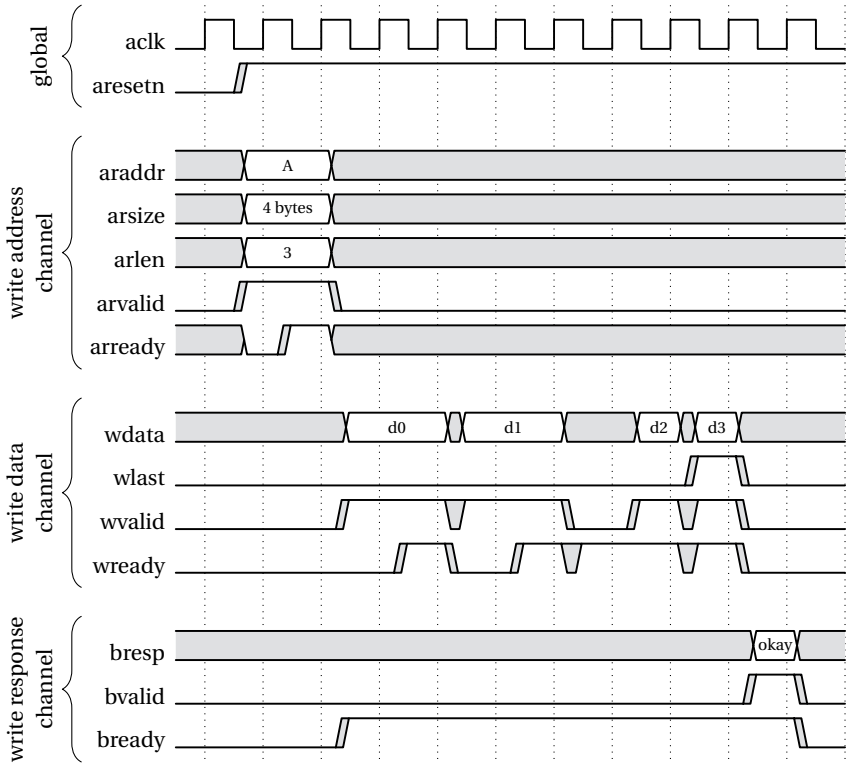


Figure 2.13: Example of write process

2.4.4. PYNQ FRAMEWORK

Python productivity for Zynq (PYNQ) is an open-source framework developed by Xilinx to simplify the use of adaptive computing platforms. By leveraging the Python programming language and Jupyter notebooks, PYNQ provides a user-friendly environment that bridges the gap between software and hardware design. This framework supports a range of devices, including Zynq-7000 SoCs, Zynq UltraScale+ MPSoCs, Kria SoMs, as well as Alveo accelerator cards. PYNQ aims to make the power of programmable logic accessible to a broader audience, from system architects to software engineers, without the need for extensive hardware design knowledge.

One of the core concepts of PYNQ is the use of overlays, which function similarly to hardware libraries. These overlays abstract complex hardware configurations, allowing developers to interact with programmable logic through high-level Python application programming interface (API)s. Although creating a new overlay requires FPGA design skills, PYNQ's "build once, reuse many times" approach ensures that these overlays can be easily adapted and reused across multiple applications. This modularity significantly enhances productivity and streamlines the development process.

PYNQ uses Python as the primary interface for programming both the embedded processors and the overlays. This integration with Python enhances productivity by providing an intuitive and high-level environment for system design and debugging. PYNQ also supports integration with C libraries and lower-level optimizations for cases where higher performance is required. This dual approach allows developers to achieve both ease of use and high performance in their projects.

Furthermore, PYNQ features a web-based architecture that is built on the Jupyter notebook interface, making it accessible from any modern web browser. Within the PYNQ environment, the ARM processor in Zynq devices hosts a web server that provides access to interactive tools such as terminals, dashboards, and Jupyter notebooks. These tools enable users to run Python code, manage configurations, and visualize results efficiently, all within a unified and interactive platform.

Through the integration of software-defined workflows and hardware acceleration, PYNQ empowers developers to prototype and deploy sophisticated applications in various domains, including artificial intelligence (AI), internet of things (IoT), robotics, and data acceleration. The framework's ability to make adaptive computing more accessible and efficient represents a significant advancement in the field of embedded systems.

2.5. HARDWARE CAPABILITY

With the rapid advancement of current technology, numerous kits that are fully integrated with PS and PL have been developed to meet the needs of research and product development. Within the capabilities of this project's facility, access to and analysis of some of them were carried out with the aim of finding a solution for the project.

2.5.1. ZYNQ ULTRASCALE+ MPSOC ZCU106 EVALUATION KIT

Zynq UltraScale+ MPSoC ZCU106 evaluation kit [19] is a development platform designed by Xilinx to accelerate high-performance applications and system integration.

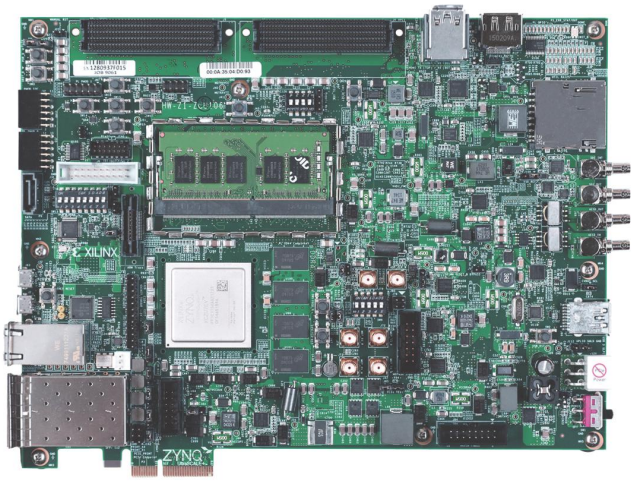


Figure 2.14: Zynq UltraScale+ MPSoC ZCU106 evaluation kit

Ideal for developers engaged in advanced computing and data processing, the kit provides a seamless approach to prototyping and deploying complex designs. It integrates high-performance programmable logic and powerful processing capabilities, thanks to the Xilinx Zynq UltraScale+ MPSoC. Additionally, with a quad-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 processor, and a Mali-400 MP2 graphics processing unit (GPU), the kit is suitable not only for general-purpose tasks and real-time operations, but also for efficient graphics processing. In summary, the kit is well-suited for computationally intensive workloads and system-level integration tasks.

The most prominent feature of the kit is that it is *optimized for quick application prototyping*. It provides a powerful combination of pre-configured development tools, including the Vivado Design Suite and Vitis Unified Software Platform, along with flexible input/output options such as universal serial bus (USB) 2.0/3.0, high-definition multimedia interface (HDMI), DisplayPort, and an RJ-45 Ethernet port, making it an ideal choice for developers working on advanced designs.

In addition, the kit's *programmable capabilities* are also excellent. With outstanding specifications as shown in Table 2.3, it becomes one of the worthy choices for projects that need to handle large workloads in the PL, such as machine learning, data processing, as well as high-speed communication systems.

Resource	Quantities
System Logic Cells	504K
LUTs	230400
FFs	460800
IOBs	360
Block random access memory (RAM)	312
Ultra RAM	96
digital signal processing (DSP) slices	1728

Table 2.3: Resource of Zynq UltraScale+ MPSoC ZCU106 evaluation kit

2.5.2. KRIA KV260 VISION AI STARTER KIT

Kria KV260 Vision AI starter kit [20] is a powerful development platform designed by Xilinx to accelerate AI and vision-based applications.

Tailored for developers working on edge computing and embedded AI solutions, Kria KV260 Vision AI starter kit provides an accessible and efficient way to prototype and deploy advanced designs. Equipped with the Xilinx Zynq UltraScale+ MPSoC, the kit integrates high-performance programmable logic with robust processing capabilities. It features a quad-core ARM Cortex-A53 processor for general-purpose tasks, a dual-core ARM Cortex-R5 processor for real-time applications, and a Mali-400 MP2 GPU for efficient graphics processing. This versatile architecture makes the KV260 ideal for computationally intensive AI workloads and latency-sensitive tasks.

As one of the latest product lines, Kria KV260 Vision AI starter kit offers outstanding features, some of which are highlighted below.

Firstly, as a *vision-ready* platform, the kit is designed for vision-based applications, seamlessly integrating up to 8 camera interfaces, 3 mobile industry processor interface (MIPI) sensor interfaces, and USB cameras, along with a built-in image signal processor (ISP) for improved image quality.

Flexibility in connectivity marks the second feature of the kit. USB 3.0/2.0 interfaces for peripheral devices, along with a 1 Gb Ethernet port for fast networking, make it one of the most suitable kits for communication-related applications.

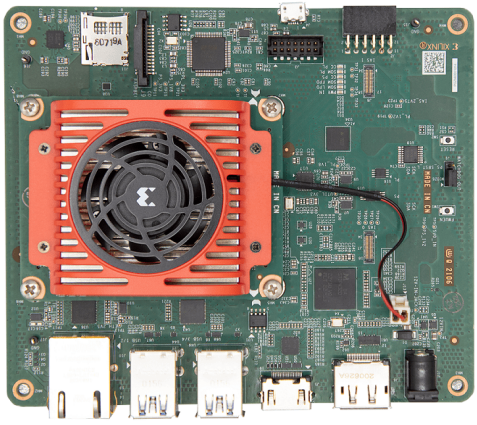


Figure 2.15: Kria KV260 Vision AI starter kit

For an FPGA-based project, *programmable capabilities* are essential, marking the kit's third outstanding feature. As shown in Table 2.4, the kit demonstrates its superiority, enabling complex designs, particularly in the communication field, to be fully realized.

Resource	Quantities
System Logic Cells	256K
LUTs	117120
FFs	234240
IOBs	189
Block RAM	144
Ultra RAM	64
DSP slices	1248

Table 2.4: Resource of Kria KV260 Vision AI starter kit

While both Zynq UltraScale+ MPSoC ZCU106 and Kria KV260 Vision AI are powerful and well-suited for the proposed system, the Kria KV260 Vision AI seems to be the more fitting choice, thanks to its vision-ready feature. Additionally, Kria KV260 Vision AI appears to be the safer choice, given the facilities it offers, which align well with the needs of this project. Therefore, after careful consideration, Kria KV260 Vision AI starter kit will be chosen to serve the project.

2.6. RELATED WORKS OVERVIEW

Thanks to the extremely powerful burst error correction capabilities of RS codes and the lightning growth of FPGA platforms, originating from data transmission systems, applications and technical standards with the participation of RS codes are becoming more and more common. Some featured works with typical RS codes were investigated, and initial data were obtained as follows.

As closely related fields to data transmission, video broadcasting and satellite are some of the territories where RS codes prove their usefulness. With its equivalent technical standard confirmed, $RS(204,188)$ was implemented quite efficiently on Xilinx Virtex-6 FPGA by S. M. Dilek in his work [21]. The design allows operation at clock frequency up to 427.50 MHz, achieving data rate performance up to 3.42 Gbps.

Data storage technology is another area where RS codes have found their place, especially in applications such as digital versatile disc (DVD) storage and flash memory. Representative in this field, C. Kim and his colleagues successfully developed $RS(255,247)$ on Altera Stratix II FPGA, achieving 1.07 Gbps data rate at 209.00 MHz clock frequency [22].

More than anything, the field of data transmission and general purpose intellectual property (IP) has seen the most significant development, especially with the two widely used $RS(255,223)$ and $RS(255,239)$ codes. [23] provides such an $RS(255,239)$ architecture on Xilinx Virtex-4 FPGA, which can be applied to the current IEEE 802.16 network standard, achieving an aggregate data rate efficiency of 1.22 Gbps data rate at 152.59 MHz clock frequency. On the other hand, the $RS(255,223)$ code implemented by J. You and his colleague can be considered an example of general-purpose IP [24]. Their work was implemented on an Altera Cyclone II FPGA, with a maximum operating clock frequency of 120.00 MHz, resulting in a data rate of 0.96 Gbps.

In addition to studies by many authors from various sources, well-known names in the FPGA field have also started to develop their own IP cores for

RS codes. Particularly, AMD has introduced RS encoder and decoder IP cores [25][26], something that offer a data rate at $m \times f$ Mbps, where f is the chosen clock frequency of the design.

However, a common weakness in most of the above works is that the design processes only a single symbol per clock cycle, causing data rates to hover around 1 Gbps and making it hard to achieve appreciably higher throughput. This is a critical limitation, as ASIC-based implementations have easily achieved this, enabling not only parallel processing of multiple symbols but also the integration of multiple RS codec instances to push the data rates up to tens or even hundreds of Gbps. Typical of the above statement, Perrone and his colleagues, through their research, produced an $RS(255,239)$ architecture that integrates both of the above factors, achieving an effective data rate of 142 Gbps at 555 MHz clock frequency, implemented using 90 nm complementary metal-oxide semiconductor (CMOS) process [27].

Achieving data rates in the hundreds of Gbps on FPGA platforms is truly a major challenge. This project, therefore, focuses on breaking the 1 Gbps barrier for RS encoders and decoders in general by leveraging the processing of multiple symbols per clock cycle as the main, overarching concept.

3

PROPOSED ARCHITECTURE

This chapter establishes the foundation for designing and implementing both the system and the RS code. Beginning with the presentation of the system architecture of a data transmission system that will serve as the platform for implementing the RS code. Then, the noisy channel model is introduced in the second section to simulate real-world noise effects. Finally, through an in-depth analysis of error probabilities and a thorough review of existing RS codes, the specific code to be implemented is identified.

3.1. SYSTEM ARCHITECTURE

As mentioned in the introduction, FEC, or more particular RS codes, represent the central ambition, the ultimate goal that this project aims for. For that reason, the system this project strives to create is designed to subtly showcase the feasibility and efficiency of RS code algorithms, particularly in encoding and decoding.

3.1.1. SYSTEM OVERVIEW

Considering that the overarching theme of the project is a communication system, it would fail to convey its full purpose if confined to a single kit. As a result, the design was built on two kits, with one acting as the sender and the other, naturally, as the receiver. Besides, regarding the communication protocol between the two kits, this project will design it as a direct peer-to-peer system, utilizing the built-in Ethernet ports on both kits.

Additionally, as mentioned earlier, digital images were chosen as the experimental data, guiding the decision to use a camera to capture this data directly. Naturally, a tool is also needed to observe the digital images during processing. To fulfill this, a simple website will be proposed, offering basic functions such as uploading and comparing images, which are essential to the project's flow.

In summary, a communication system consisting of two Kria KV260 Vision AI starter kits, along with a camera and a simple website, will form the entire system that this project is built on. Figure 3.1 is constructed to show both the overall design of the system, the functional components, and the detailed data flow between them, which will be presented in more detail in subsection 3.1.2.

3.1.2. COMPONENT AND DATA FLOW DESCRIPTIONS

The system begins with data generation, where the *camera* captures images and sends them to the *image to binary converter*, which is implemented on the PS of the sender kit. Here, the raw images are processed and converted into a sequence of binary digits, making them suitable for the further encoding algorithm. Additionally, digital images captured by the camera will also be sent to a *website access module* and then to the website, serving as the original reference for later comparisons.

The sequence of binary digits, which is the output of the image to binary converter, is now transferred from the PS to the PL through the AXI4 communication interface to reach the *RS encoder*. At this stage, the encoder performs its function by adding parity portions to the sequence, forming a new sequence of binary digits.

The encoded sequence is then sent back to the PS, also via AXI4. The important thing here is that errors are added within the *noisy channel module* implemented in the PS before actually being sent. In reality, this process will never happen. This action is intended solely to simulate the possibility of errors occurring on the transmission line.

After being injected with errors, the encoded sequence will be sent to the receiver kit via the Ethernet connection. Here, assuming that data transmission over the Ethernet connection is reliable, meaning that errors either do not occur or are negligible. As a result, the sequence received by the receiver kit will be exactly the error-injected sequence, which is, of course, completely distinct from the actual information of digital images.

Here, since the communication used is Ethernet, which has its own conventions, the received data is not immediately suitable for processing. The *data*

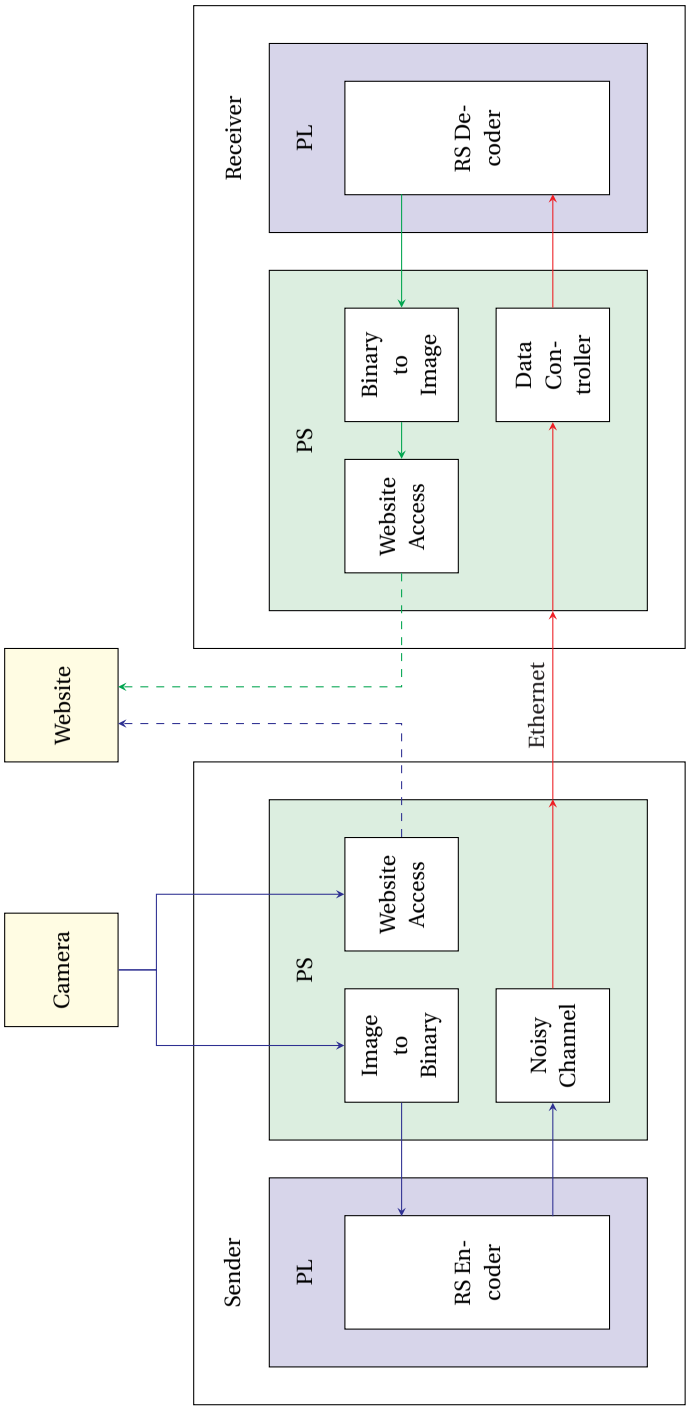


Figure 3.1: Proposed system architecture

controller, which is implemented on the PS will handle the conversion task to return the received data to a suitable form for the subsequent decoding step.

The data, after being processed by the data controller will now be sent to PL through AXI4, and then to the *RS decoder* to perform the decoding algorithm and retrieve the original information of the digital image. As reflected in the earlier theories, this step stands as the most challenging and serves as one of the central focuses of this project.

The decoded data is then sent back to another *binary to image converter* implemented in the PS, continuing through AXI4. Here, it is converted back into another image, which can be called the estimated image. The estimated image is then be passed directly to the *website access module* in the receiver, where it is continuously sent to the website and prepared for display.

Finally, the two digital images, the original and the estimated, will be fully displayed on the *website*. Comparisons are then made to demonstrate that RS algorithms can almost perfectly reconstruct information from digital images.

3.1.3. TECHNICAL REQUIREMENTS

As stated, the system's purpose is to be dedicated to RS algorithms, so the technical requirements must remain consistent with this overarching goal. At the system level, the following key requirements are first established.

- The camera must have a resolution of at least video graphics array (VGA) standard, in order to ensure that the digital image has sufficient detail and a large enough data size.
- The Ethernet connection between the two kits must be fully utilized, meaning it must operate stably at a 1 Gbps data rate.
- The website must deliver responsive performance, supporting concurrent image uploads and comparisons without noticeable delays, even under moderate workloads.

Continuing on, the focus will shift to the two kits that are at the heart of the system. The technical requirements are first established for the modules implemented in the PS, specifically as follows.

- For the website access modules, it is essential to uniquely perform the function of uploading digital images to the website, without altering the image information.
- The image to binary and binary to image converters must accurately transform images into binary strings and vice versa.

- Similarly, the data controller module is restricted to performing only the function of converting the information format.
- The noisy channel module must accurately implement the error injection functionality for the data, ensuring that the injected error aligns with the model proposed in section 3.2.

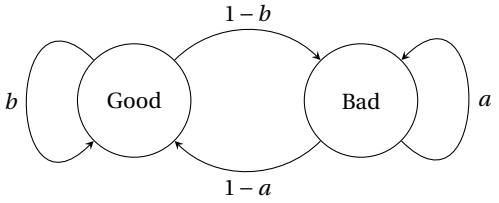
Finally, and most importantly, are the RS encoder and RS decoder modules implemented in the PL of the two kits. The technical requirements outlined here are aimed at optimizing the design to the fullest.

- The implemented RS code must be the one identified in section 3.3, which has been proven to be the most efficient.
- The RS encoder and decoder interfaces must align with the communication protocol between the PS and PL, specifically AXI4.
- The RS encoder and decoder must be capable of performing at least at 1 Gbps data rate to take full advantage of the hardware capabilities. Note that this is only the lower limit of the data rate, and the implemented system requires careful analysis to meet it while optimizing processing capabilities.
- The clock frequency of them must be the same and must meet, at a minimum, the default parameter of 100 MHz. This parameter can be increased in the case of a large computational load, but trade-offs, especially regarding power, must be considered.
- Lastly, the resources utilized and the latency are key parameters that must be carefully optimized and thoroughly benchmarked against similar products in order to effectively demonstrate the system's efficiency.

3.2. NOISY CHANNEL MODEL

As mentioned earlier, identifying the error characteristics of the noisy channel is a critical factor in selecting an appropriate EEC algorithm and, more specifically, an appropriate RS code. Given that obtaining such data from real-world channels like Ethernet or Wi-Fi is unfeasible for this project's scope, the channel will instead be modeled using the Gilbert–Elliott model.

For the purpose of this project, to demonstrate the effectiveness of RS code, it becomes essential to build a model that best captures the essence of burst errors. Therefore, within the original model depicted in Figure 2.3, the two parameters p_1 and p_2 are fixed at 0 and 1, respectively. Meaning in good state S_1, data transmission is completely error-free and, conversely, in bad state S_2, data transmission is undeniably erroneous.



3

Figure 3.2: Proposed Gilbert–Elliott model

Besides, according to the principles of Markov chain theory [28], $q_{11} + q_{12} = q_{21} + q_{22} = 1$, reflects the rule that all transition probabilities from a state must sum to 1. As a result, Gilbert–Elliot model takes on a new form, as illustrated in Figure 3.2, with its corresponding transition matrix presented in Equation 3.1.

$$\mathbf{P} = \begin{bmatrix} b & 1-b \\ 1-a & a \end{bmatrix} \tag{3.1}$$

With the model constructed as described above, the probability of a bit being in error is equivalent to the probability of the system being in bad state, denoted as π_B, value that can be determined through a process as presented in Equation 3.2. Clearly, the probability of an error occurring on the modeled channel depends entirely on how the state transition probabilities a and b are chosen. Figure 3.3 shows the influence of a and b on π_B, offering a visual perspective for understanding the relationship between the parameters.

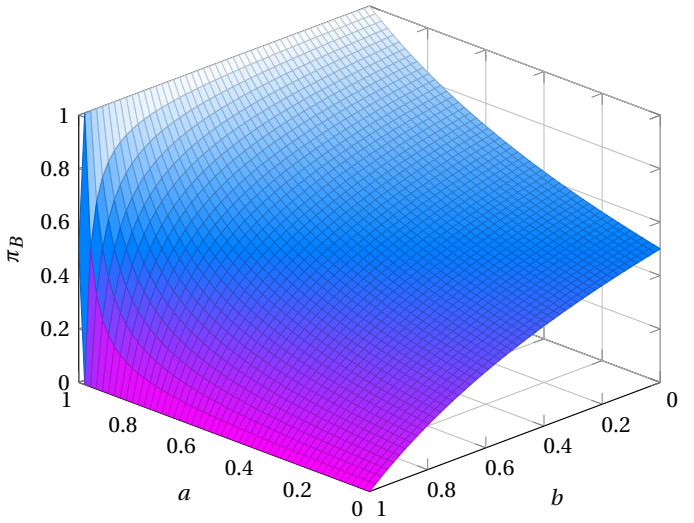


Figure 3.3: Influence of state transition on error probability

$$\begin{cases} \pi_G = b\pi_G + (1-a)\,\pi_B \\ \pi_B = a\pi_B + (1-b)\,\pi_G \\ \pi_G + \pi_B = 1 \end{cases} \Rightarrow \begin{cases} \pi_G = \dfrac{1-a}{1-a+1-b} \\ \pi_B = \dfrac{1-b}{1-a+1-b} \end{cases} \tag{3.2}$$

Recent work by C. Y. Liu [29] has stated the connection between the Gilbert–Elliot model and the decision feedback equalization (DFE) architecture, leading to an approximate determination of the bad-to-bad transition probability a through the corresponding parameters of the system. Since the DFE architecture lies beyond the scope of this project, values of a are chosen based on notable recent research [29][30]. Accordingly, three cases are considered, including $a = 0.375$, $a = 0.5$, and $a = 0.75$.

In specific simulation scenarios, the probability of bit errors on the channel, commonly referred to as the pre-FEC BER, is often treated as a given parameter. It not only reflects the capabilities of the channel but also implies the objectives of the simulation scenario. In such a case, the remaining parameter, the good-to-good transition probability b, can be calculated as shown in Equation 3.3.

$$b = 1 - (1-a)\,\frac{\pi_B}{1-\pi_B} \tag{3.3}$$

To demonstrate effectiveness of the model, a simulation was conducted with the desired pre-FEC BER set to 10^{-4}. Starting from the good state, 10^9 modeled state transitions were performed and the results are presented in Table 3.1. Evidently, the burst errors are simulated effectively, with the number of burst-$(n+1)$ errors appears approximately equal to a times the burst-n errors.

	$a = 0.375$	$a = 0.5$	$a = 0.75$
Single error	38955	25334	6174
Burst-2 error	14592	12352	4670
Burst-3 error	5430	6167	3528
Burst-4 error	2053	3250	2625
Burst-5 error	810	1586	1936
Burst-6 error	282	779	1478
Burst-7 error	127	357	1142
Burst-8 error	45	210	817
Other errors	30	187	2531
Total bit error	99919	100197	100217

Table 3.1: Proposed noisy channel model's simulation results

3.3. REED–SOLOMON CODE IDENTIFICATION

Once the channel construction is complete, the next step is to perform preliminary analysis to determine the EEC algorithms and other coding techniques required for the system. For this project, although RS codes were the only implemented algorithm, analysis was still necessary to identify a suitable choice.

3.3.1. ERROR PROBABILITY ANALYSIS

To determine the suitability of a particular RS code, it is crucial to know the error probability before and after applying the RS algorithm, necessitating a model for relatively accurate analysis. The concept explored in this section draws inspiration from the trellis model introduced by R. Barrie and colleagues [30], simplified and adapted to align with the aspirations of this project.

BIT TRELLIS

The goal of error probability analysis is to determine the number of bits and symbols with errors in a codeword of a given length, based solely on the state transition probabilities between the good (G) and bad (B) states. First of all, the state transitions of the channel model built in section 3.2, when represented over time, would appear as shown in Figure 3.4. Note that in this project, k_b represents the k_bth bit in the transition sequence.

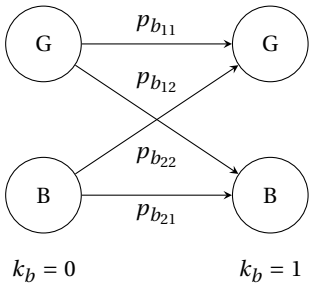


Figure 3.4: Bit transitions over time

Clearly, the steady-state probability π_i, can be calculated using Equation 3.4, where $i, i' \in \{G, B\}$. Here, to maintain consistency with the notation used at higher levels, $p_{b_{11}}$, $p_{b_{12}}$, $p_{b_{21}}$, and $p_{b_{22}}$ represent the state transition probabilities corresponding to b, $1-b$, $1-a$, and a, respectively. As a result, π_G and π_B can be obtained using Equation 3.2.

$$\pi_i = \sum_{i'} p_{b_{i'i}} \pi_{i'} \tag{3.4}$$

SYMBOL TRELLIS

Before addressing codewords, the trellis model must be developed to the symbol level, which includes a specific number of transitions at the bit level. This particular state transition can be visually represented in greater detail through Figure 3.5.

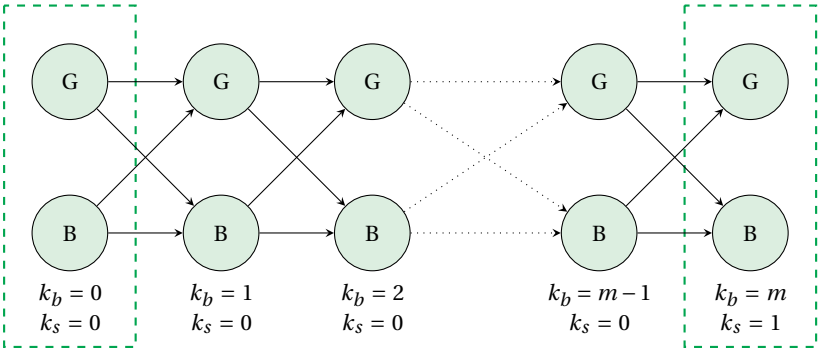


Figure 3.5: Bit transitions within a symbol

The goal at the symbol level is not to determine the probability of a symbol's last bit being in state good or bad, but rather to identify the exact number of bit errors that have occurred. To achieve this, the most traditional approach is to enumerate all possible error patterns in the trellis, leading to an overwhelming number of possible cases. With the idea of grouping the probability of all trellis paths having the same number of bit errors, the above situation can be solved efficiently.

Specifically, let $PrB_{k_b}^{j_b}(i)$ represent the probability of arriving at state i at the k_bth bit, after traversing all trellis paths containing exactly j_b bit errors. Then, at time k_b+1, the corresponding probability is determined by Equation 3.5, where $j = 0$ if i = G, and $j = 1$ if i = B. And so on, the algorithm is repeated until the length of a symbol is reached, at which point a set of $PrB_s^{j_b}$ (G) and another set of $PrB_s^{j_b}$ (B) are obtained, with $0 \leq j_b \leq m$.

$$PrB_{k_b+1}^{j_b}(i) = \sum_{i'} PrB_{k_b}^{j_b-j}\left(i'\right) \cdot p_{i'i} \tag{3.5}$$

CODEWORD TRELLIS

If bits are the constituent unit of a symbol, then symbols, in turn, are the constituent unit of a codeword, making the process of building the trellis model at the codeword level somewhat similar to the previous one. Figure 3.6 illustrates these state transitions.

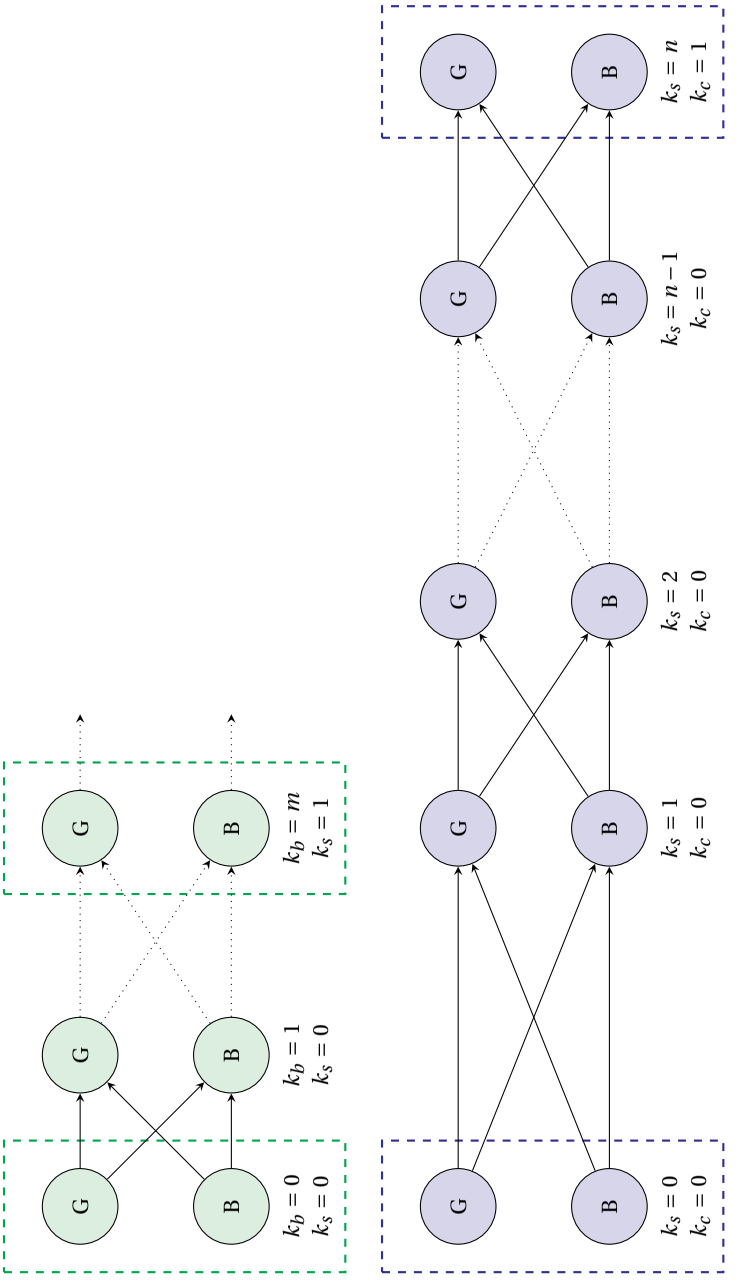


Figure 3.6: Symbol transitions within a codeword

At the codeword level, the task involves determining the exact number of bit errors as well as identifying the symbol errors within a codeword. Let $PrS_{k_s}^{j_s,j_b}(i)$ represent the probability of arriving at state i at the k_sth symbol, after traversing all trellis paths with exactly j_s symbol errors and j_b bit errors. Assuming the trellis paths from time k_s to k_s+1 account for j bit errors, the corresponding probability at time k_s+1 is calculated using Equation 3.6.

$$PrS_{k_s+1}^{j_s,j_b}(i) = \sum_{j}\sum_{i'} PrS_{k_s}^{j_s-\min\{1,j\},j_b-j}\left(i'\right)\cdot PrB_s^{j}(i) \mid PrB_0^{j}\left(i'\right) = 1 \tag{3.6}$$

BIT ERROR RATE DETERMINATION

For a codeword with n symbols, the computation is repeated n times, yielding two sets, $PrS_n^{j_s,j_b}$ (G) and $PrS_n^{j_s,j_b}$ (B), where $0 \le j_s \le n$ and $0 \le j_b \le nm$. By definition, a codeword is considered erroneous after decoding if the number of symbol errors exceeds the error correction capability of the RS code or, mathematically, $j_s > t$. To add to that, the corresponding probability is calculated precisely as shown in Equation 3.7.

$$P\left(j_s > t\right) = \sum_{j_s=t+1}^{n}\sum_{j_b=1}^{nm}\sum_{i} j_b \cdot PrS_n^{j_s,j_b}(i) \tag{3.7}$$

In the realm of reality, once the decoder is implemented, the probability of error transforms into something much more complicated. Specifically, when the number of symbol errors surpasses the decoder's error capability, it may worsen the errors or, in other words, increase the post-FEC BER. The opposite scenario can also completely emerge since, by removing the redundant data, some error bits can be simultaneously eliminated, resulting in a slightly lower post-FEC BER. However, a detailed analysis of the above problem proves too complex, so this project adopts the post-FEC BER as an approximation of the bit error probability within a decoded codeword.

$$\text{post-FEC BER} \approx \sum_{j_s=t+1}^{n}\sum_{j_b=1}^{nm}\sum_{i} j_b \cdot PrS_n^{j_s,j_b}(i) \tag{3.8}$$

3.3.2. ANALYSIS OF CURRENT REED–SOLOMON CODES

With the noisy channel and the post-FEC BER evaluation model established, the effectiveness of any RS code can be assessed under any specific error strategy. For this project, the RS codes found in [2], which are the most popular RS codes in the communication field today, are used to perform the evaluations. Specifically, it includes $RS(255,223)$, $RS(255,239)$, $RS(528,514)$, and

RS(544,514) codes. Each of them will be analyzed using three strategies, corresponding to three values of a defined in section 3.2, with pre-FEC BER values ranging from 10^{-2} to 10^{-5}.

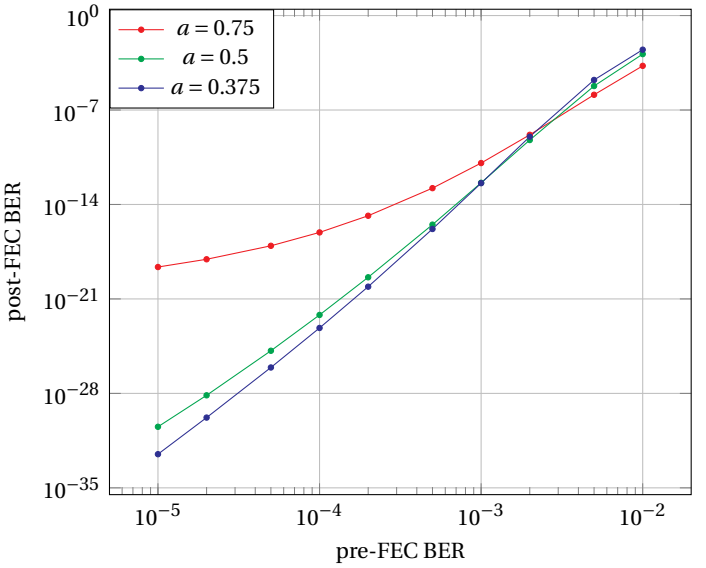


Figure 3.7: Performance analysis of *RS*(255,223) code

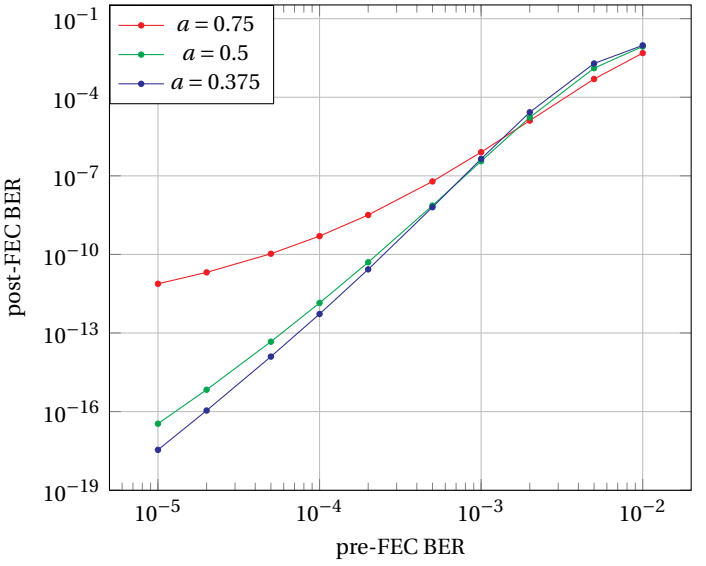


Figure 3.8: Performance analysis of *RS*(255,239) code

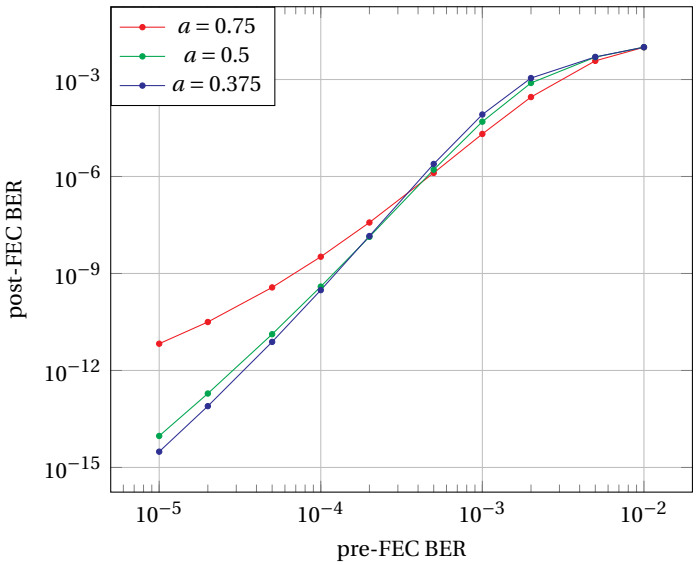


Figure 3.9: Performance analysis of $RS(528,514)$ code

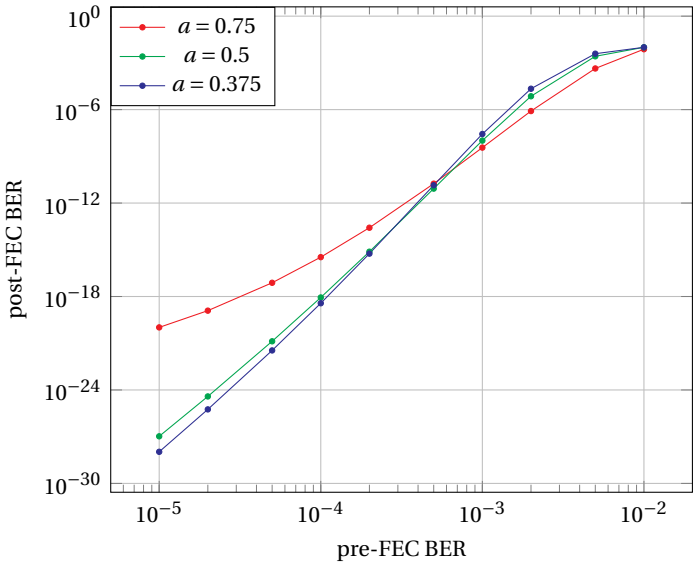


Figure 3.10: Performance analysis of $RS(544,514)$ code

From the analysis results obtained above, it is clear that the shapes of the lines are relatively similar, thus this project chooses the pre-FEC BER = 10^{-4} to perform the evaluations. Given the ambition, the error probability is expected to decrease by 10^8 times after applying RS algorithms, resulting in a

post-FEC BER = 10^{-12}. Take note that the proposed parameters are crafted in alignment with the current standard for 1 Gb Ethernet [2]. According to the analysis results, $RS(255,223)$ and $RS(544,514)$ are the two codes that yield the best results when all strategies produce satisfactory outcomes. $RS(255,239)$ also performs well when two out of three strategies are satisfied. In the remaining case, $RS(528,514)$ does not meet any of the strategies required.

Besides the relationship between pre-FEC BER and post-FEC BER, the code rate, which is k/n ratio, and the construction field's degree are also important parameters to consider. Firstly, in terms of the field's degree, $RS(528,514)$ and $RS(544,514)$ are two units larger than the others, which means their construction field is four times larger. Besides, it is easy to see that the code rates of $RS(255,239)$ and $RS(544,514)$ are quite similar, while the above parameter for $RS(255,223)$ is too low. As for $RS(528,514)$, its code rate is much higher than the others, but this comes at the cost of poor error correction, as mentioned above.

	RS $(255,223)$	RS $(255,239)$	RS $(528,514)$	RS $(544,514)$
Field's degree m	8	8	10	10
Codeword length n	255	255	528	544
Message length k	223	239	514	514
Code rate k/n	0.8745	0.9373	0.9735	0.9449

Table 3.2: Parameters of current RS codes

Based on the above evaluations, this project chose $RS(255,239)$, which performed the best compared to the others, as the only RS code to be implemented in the project.

4

REED–SOLOMON ALGORITHMS DESIGN

This chapter provides a comprehensive perspective on the design process of RS algorithms using HDL. In a more thorough exploration, the chapter starts by utilizing background knowledge to construct the identified $RS(255,239)$ code. Following this, all operators in the finite field, which serve as elementary components of the RS algorithms, will be elaborated for design in the second section. To conclude, the final two sections will, respectively, address the transition from theoretical concepts to actual circuit design for the RS encoder and decoder, fulfilling the project's final objective.

4.1. REED–SOLOMON CODE CONSTRUCTION

For the implementation of an algorithm, no matter what, on a hardware platform, defining fixed parameters plays a paramount role as it maximizes resource efficiency and minimizes effort required during the design process. This project, of course, is not only no exception but also one of the significant cases that show the outstanding effectiveness of this work.

As in section 3.3, where $RS(255,239)$ is the identified code, and with the corresponding theory for determining code parameters introduced in subsection 2.3.1, its fixed parameters include:

- Codeword length $n = q - 1 = 255$ symbols.
- Message length $k = q - 2t - 1 = 239$ symbols.
- Parity length $n - k = 2t = 16$ symbols.

- Error-correcting capability $t = 8$ symbols.
- Error-detecting capability $2t = 16$ symbols.
- Field's order $q = p^m = 256$.
- Field's characteristic $p = 2$.
- Field's degree $m = 8$.

The focus within these parameters begins with $m = 8$, the key element for constructing the finite field $GF\left(2^8\right)$, which exactly is the extension of the binary field $GF(2)$, serving as the basic foundation for RS code algorithms. Using the corresponding primitive polynomial $p(X) = 1+X^2+X^3+X^4+X^8$ from Table 2.2, step by step, the construction of such a finite field is revealed, with each detail laid out in subsection 2.2.4. For the convenience of upcoming calculations, the complete set of elements of the finite field $GF\left(2^8\right)$ is presented with clarity in Appendix A.

After successfully defining the primitive polynomial and element set of the finite field, the next and also the final fixed parameter to consider is the generator polynomial $g(X)$ of the RS code. Recall from the theory that, while there are multiple approaches to form the generator polynomial, the most favored one is through Equation 2.20, leading to a primitive RS code. Given the large amount of computation required, only the original expression and its concluding result are offered in Equation 4.1.

$$\begin{aligned} g(X) &= (\alpha + X)\left(\alpha^2 + X\right)\ldots\left(\alpha^{16} + X\right) \\ &= \alpha^{136} + \alpha^{240}X + \alpha^{208}X^2 + \alpha^{195}X^3 + \alpha^{181}X^4 \\ &\quad + \alpha^{158}X^5 + \alpha^{201}X^6 + \alpha^{100}X^7 + \alpha^{11}X^8 + \alpha^{83}X^9 + \alpha^{167}X^{10} \\ &\quad + \alpha^{107}X^{11} + \alpha^{113}X^{12} + \alpha^{110}X^{13} + \alpha^{106}X^{14} + \alpha^{121}X^{15} + X^{16} \end{aligned} \tag{4.1}$$

Summarizing, in addition to RS code parameters, the fixed parameters required for the algorithm include the primitive polynomial $p(X)$, the generator polynomial $g(X)$, and the full set of elements within the finite field $GF\left(2^8\right)$. This implies that, during the later design process, the above parameters can be directly accessed, eliminating the need for recalculations.

4.2. FINITE FIELD ELEMENTS

As introduced in the theory sections, the extension of binary field is constructed based on the two groups, which include the modulo-2 addition and multiplication. Thus, it possesses all their inherent properties, and most importantly, the four foundational operators are the addition, multiplication, and their respective inverses. Clearly, from these four operators, all available oper-

ations, no matter if they are addition, subtraction, multiplication, division, or exponentiation, can be effortlessly executed. However, the additive inverse, as defined in Equation 2.18, is an identity operator, leaving just three remaining operators to be designed. This section, consequently, is responsible for designing these operators on the hardware platform.

In the specific case of this project, finite field operators are but a modest fragment within the much larger complexity of the overall challenge. Alongside that, the field's degree $m = 8$, within the context of mathematics, is far from being a large number. Therefore, the most suitable strategy for designing them is through the corresponding bit-parallel combinational logic circuits.

4.2.1. INVERTER

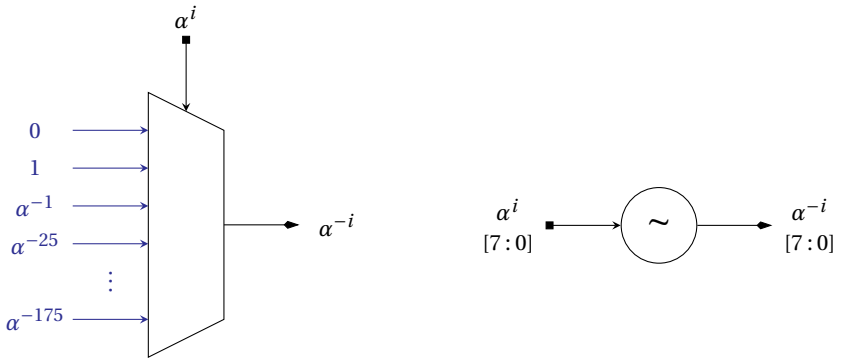
To begin, the multiplicative inverse, which is also the only remaining unary operation, is first considered through its corresponding hardware element, the inverter. Even though Equation 2.19 provides a powerful solution in theory, determining α^i from a binary 8-tuple, performing the calculation, and then retracing it backward on hardware platform becomes an impractical task, or more precisely, consumes a significant amount of resources. Instead, a more appropriate approach is to take advantage of the group property, which states that each element has one and exactly one inverse.

Accordingly, the inverter operates as a direct one-to-one mapping from a binary 8-tuple input to an equally formatted output. One such circuit used to realize this function, in modern digital designs, is an 8-to-8 LUT, which can be easily implemented via either array-based or multiplexer-based design styles.

Visually represented in Figure 4.1a, its equivalent architecture can also be interpreted as a 256-to-1 multiplexer[1]. Within it, input data of the multiplexer, except for the first entry, is ordered as the multiplicative inverses of ascending binary 8-tuples. To make it easier to understand, in addition to the first input being 0, the second input is the multiplicative inverse of binary 8-tuple 10000000, which corresponds to 1 (see Table A.1). Similarly, the third input is the inverse multiplicative of binary 8-tuple 01000000 = α, and the fourth input is the inverse multiplicative of binary 8-tuple 11000000 = α^{25}, and so on.

Ultimately, to facilitate the development of complex architectures later, an equivalent symbolic representation for it is introduced in Figure 4.1b, which also marks the end of this section.

[1] In this chapter's architecture diagrams, fixed parameters are presented in blue, input signals are marked by an arrow with a square tail, while output signals by an arrow with a diamond head.



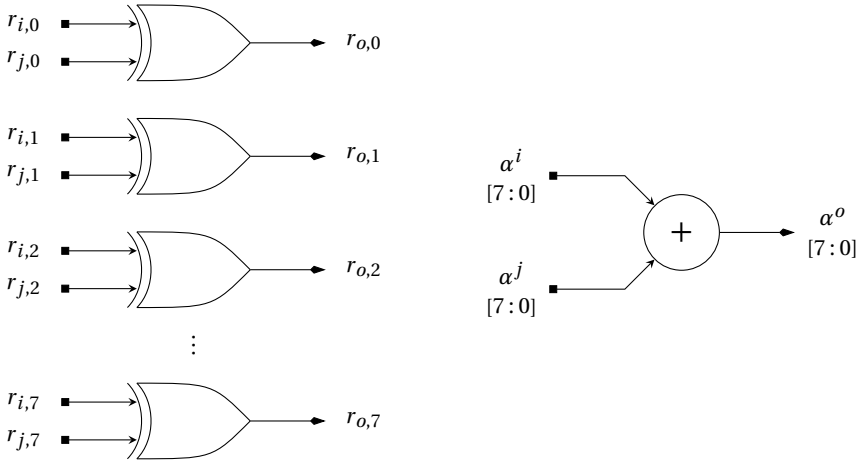
(a) Functional architecture (b) Symbolic representation

Figure 4.1: $GF(2^8)$ inverter

4.2.2. ADDER

Compared to elements that need to be implemented, it can be said that the adder is the easiest one, since its operation is built upon the modulo-2 addition group. Such a definition is proven in Equation 2.16, and since the internal computations follow Table 2.1a, which is equivalent to the *exclusive-or* truth table, the hardware implementation can be performed in a directly way.

Without loss of generality, consider the calculation $\alpha^i + \alpha^j = \alpha^o$, along with the value of α^i, for $0 \leq i < 2^m - 1$, constructed according to Equation 2.15, the corresponding functional architecture and symbolic representation can be directly obtained, as illustrated in Figure 4.2.



(a) Functional architecture (b) Symbolic representation

Figure 4.2: $GF(2^8)$ adder

It should be added that, for convenience in representing more complex designs in later sections, a symbolic representation for operators can include more than two inputs. In such a case, suppose the element accepts ξ binary 8-tuple inputs, with $\xi \geq 2$, the actual number of elements is defined as $\xi - 1$, while the critical path delay includes $\lceil \log_2(\xi) \rceil$ corresponding element units. Although the obtained post-implementation result may differ slightly, it is enough to emphasize the importance of minimizing inputs for all finite field elements during the course of design.

4.2.3. MULTIPLIER

When looking at RS algorithms as a whole, addition and multiplication are not only the two basic operators but also the most commonly used ones. However, as previously introduced, the adder can be directly implemented with minimal resources, so the cost is now concentrated back to the multiplier. That is why designing an optimized multiplier plays a pivotal role in optimizing the RS algorithm designs in general.

As a representative case, take $\alpha^i \cdot \alpha^j = \alpha^o$. It is clear that Equation 2.17 cannot be applied, as, from the perspective of hardware, where data is represented entirely in binary, transforming a binary 8-tuple to α^i is a major challenge. Along with that, the idea of using LUTs also lead to an undesirable outcome, given that as many as $2^8 \times 2^8 = 65536$ input combinations must be managed, which is beyond what a basic element can reasonably have.

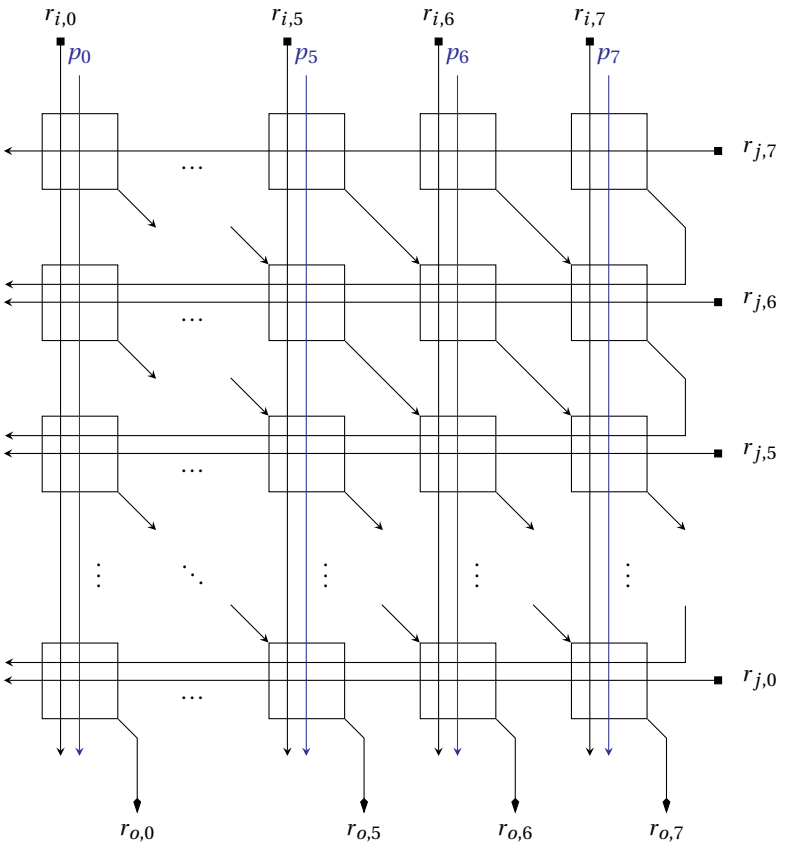
Referring back to the definition in Equation 2.14, where both α^i and α^j are defined under modulo $p(X)$, it follows that $\alpha^i \cdot \alpha^j$ is also under modulo $p(X)$. Let $R_p[\xi(X)]$ denote that $\xi(X)$ is defined under modulo $p(X)$, the above statement can be reformulated mathematically as shown below.

$$\begin{aligned} \alpha^i \cdot \alpha^j &= R_p\left[r_i(X) \cdot r_j(X)\right] \\ &= R_p\left[\ldots R_p\left[X \cdot R_p\left[X \cdot R_p\left[X \cdot r_{i,7} r_j(X)\right]\right.\right.\right. \\ &\qquad \left.\left.\left. + r_{i,6} r_j(X)\right] + r_{i,5} r_j(X)\right] + \ldots\right] + r_{i,0} r_j(X) \\ &= R_p[r_o(X)] = \alpha^o \end{aligned} \tag{4.2}$$

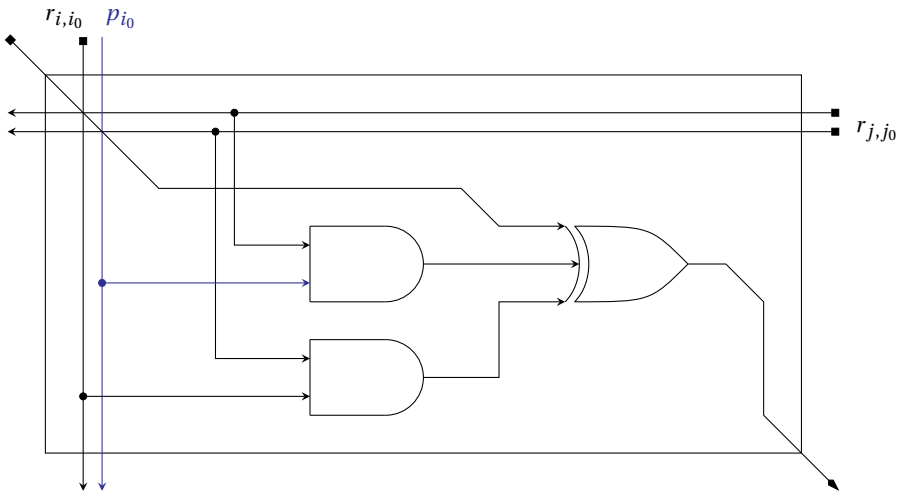
A hardware architecture corresponding to the above expression is shown in Figure 4.3a, consisting of interconnections among $8 \times 8 = 64$ standard cells, with one of which is shown in Figure 4.3b. In the standard cell, one bit from α^i and another from α^j are multiplied modulo-2 together using Table 2.1b, which also corresponds to the truth table of the *and* gate. The result is then added, also in modulo-2, to two other terms and passed through the diagonal path, which is equivalent to the action of multiplication by X. One of the two

terms mentioned above is the output from the earlier standard cell, which is multiplied by X and can be in an overflow state, so it needs an additional term to address this condition. Specifically, when the overflow occurs, or the most significant bit (MSB) is 1, in other words, a feedback pattern is created, and whether it is added to the sum is decided by coefficients of $p(X)$.

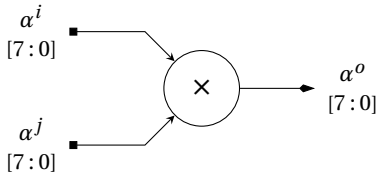
It should be emphasized that Figure 4.3b only shows the logical relationships between the signals and is not, in any way, a post-implementation architecture. Design complexity would be reduced considerably, provided the fact that that one of the signals is locked as a fixed parameter. For example, if $p_{i_0} = 1$, the second input of the *exclusive-or* gate will be directly connected to the overflow feedback signal, and even in the case where $p_{i_0} = 0$, this input will be completely removed, simplifying the *exclusive-or* gate to a two-input gate. This also explains the significance of defining fixed parameters, as well as choosing a primitive polynomial with the minimum number of terms.



(a) Functional architecture



(b) Standard cell architecture



(c) Symbolic representation

Figure 4.3: $GF(2^8)$ multiplier

Finally, as with earlier elements, this section concludes with a symbolic representation for the multiplier is provided in Figure 4.3c to support the subsequent design stages.

4.2.4. HALF-MULTIPLIER

The earlier exploration points out how crucial fixed parameters are in optimizing the design, especially when faced with a component as complex as the multiplier. Along with that, in practical RS algorithms, multiplication does not always require both factors to be variables. Therefore, a simplified version of it, referred to as a half-multiplier or a scaler, can be developed, in which one of the two factors in the multiplication is fixed. No generality is lost by taking α^i as a fixed parameter, and a corresponding architecture for a standard cell in such case is then depicted in Figure 4.4.

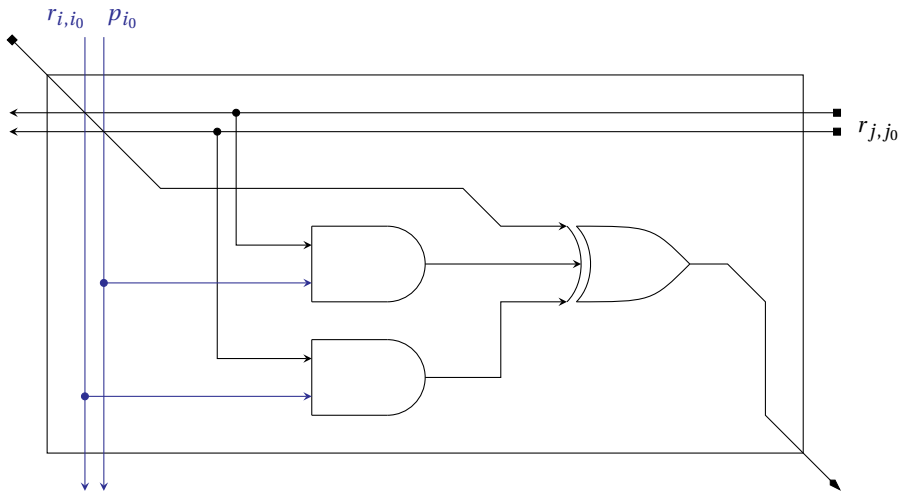


Figure 4.4: Standard cell architecture of $GF(2^8)$ half-multiplier

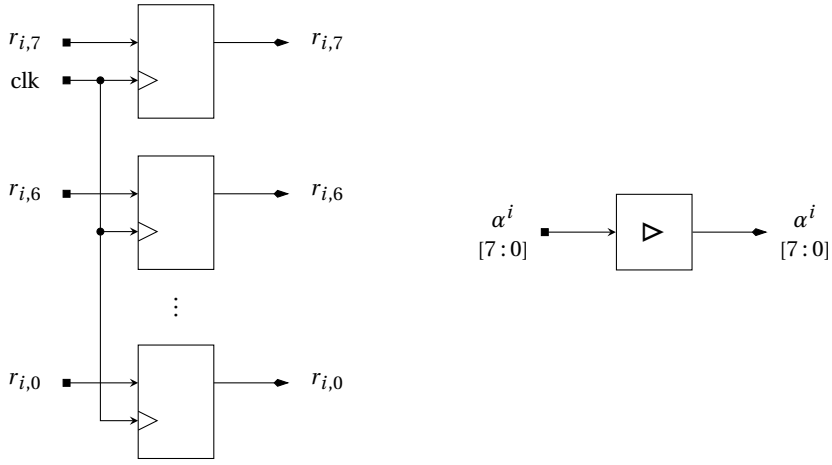
Clearly, since one input of each *and* gate is fixed, they no longer exist in the actual hardware. At worst, when both fixed parameters entering a standard cell are 1, the cell's hardware equivalent consists of a three-input *exclusive-or* gate. If either parameter is 0, the gate simplifies to two inputs. Finally, when both parameters are 0, which is also the most ideal scenario, the input is directly forwarded to the output without passing through any logic gates. It is evident that the stated required resources are far fewer than those of a standard multiplier, making this element a preferred choice for design optimization.

Apart from this important difference, the remaining aspects, including functional architecture and symbolic representation, will remain unchanged compared to the multiplier, which was previously introduced in Figure 4.3a and Figure 4.3c, respectively.

4.2.5. REGISTER-SET

Given the high complexity of algorithms, there is no doubt that the calculations must be performed over multiple clock cycles, leading to the need to store values of $GF(2^8)$ elements.

Clearly, a set of 8 FFs in parallel is suitable for storing each element, so this section focuses only on standardizing their symbolic representation. Note that the symbolic representation focuses only on expressing the element's value, while other naturally existing signals such as clock and reset are intentionally left out. A detailed and complete version of this representation can be found in Figure 4.5.



(a) Functional architecture (b) Symbolic representation

Figure 4.5: $GF(2^8)$ register-set

4.3. REED–SOLOMON ENCODER DESIGN

For all RS-code-based applications, the encoder stands as one of the two fundamental and indispensable components. Despite its considerable divergence from the decoder in terms of algorithmic complexity and resource consumption, the importance compared to its counterpart on the other side of the frontline is almost no different. This section provides a transition process from classical theoretical concepts to their corresponding realization in modern hardware-based architecture.

4.3.1. OVERALL ARCHITECTURE

Considering encoding techniques, the theory section brings two separate methods, including non-systematic and systematic encoding. However, given the enormous and obvious advantages of the systematic approach, selecting the appropriate algorithm becomes an easy decision.

Looking back to Figure 2.2, encoder is part of the transmitter, located between the generator and the channel, thus its interface must align with these components appropriately. First of all, in essence, since the generator and the encoder both belong to the transmitter, interfaces along this path are allowed to be flexibly adjusted. However, the reverse direction is not so easy because the channel always requires a fixed interface, which is commonly a power of 2. Finally, it can be said that the system always expects to work with the same data interface to increase compatibility. The current Ethernet standard [2] has also

proven that, while interface for input and output data of not only the encoder but also the decoder is set to be equal.

Since the technical requirements specify that, at the default frequency of 100 MHz, the system must operate stably at a data rate of at least 1 Gbps, it can be deduced that the minimum number of bits must be perform per clock cycle can be calculated as below.

$$\text{bits per clock cycle} \geq \frac{\text{data rate}}{\text{frequency}} \div \frac{k}{n} = \frac{1 \times 10^9}{100 \times 10^6} \div \frac{239}{255} \approx 10.6695\,(\text{bits}) \quad (4.3)$$

Combined with the condition that the number of bits for the data interface must be a power of 2, it follows that permissible values are $16, 32, 64, \ldots$ bits, which correspond to $2, 4, 8, \ldots$ $GF\left(2^8\right)$ symbols, respectively. For the purpose of the design process, let s refer to how many symbols are at the encoder's data interface, with $s = \{2, 4, 8, 16\}$, satisfying all project constraints.

To better understand what is meant by data interface, it is necessary to affirm that there are only two possible cases: either all s symbols in the interface carry meaningful information, or none of them do. Obviously, to eliminate even the slightest waste, it is mandatory to ensure that the entire channel is filled with data at all times, or, in other words, to keep the output data interface with s symbols at every clock cycle. On the other hand, the input data interface is more open, some clock cycles may not carry any data, due to the undeniable fact that message's length is always smaller than its corresponding codeword.

Concluding the discussion, the overall architecture of the encoder is clearly captured in Figure 4.6.

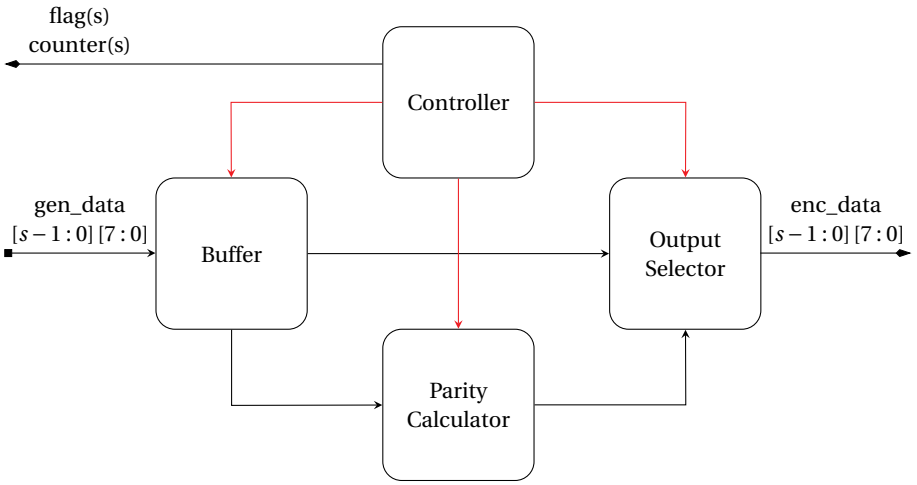


Figure 4.6: Encoder overall architecture

A brief overview of the architecture shown in Figure 4.6, it consists of four main distinct modules, each serving a specific purpose as follows.

- Firstly, *controller* is the module that is responsible for coordinating the encoder's operations through control signals[2], as well as producing required flags according to the design specifications.
- Secondly, the *buffer* is used for two purposes, one is to store sufficient data for the calculation process, and the other is to serve as a pipeline input buffer for the entire encoder.
- Thirdly, *parity calculator* is the most important module in the encoder, where the systematic encoding technique takes place.
- Finally, the *output selector* is responsible for ensuring that all codewords are sent in the conventional order[3], and that the encoder output maintains a continuous data flow. Additionally, as it is the last module in the processing chain, a pipeline output buffer must also be integrated into it.

All of above modules will be presented in detail in the following sections of the project, arranged in a logical sequence for ease of understanding.

4.3.2. CONTROLLER

Functionally, the controller does not serve as the central module of the overall encoding algorithm. Rather, it acts as a bridge, mapping the real-world problem of continuous data into an problem, where data is treated in fixed-size blocks. To carry this out, a designated *master counter* is used, whose operation is set up as follows.

Input: master_counter$_i$
Initialize: master_counter$_0 = 0$
if master_counter$_i > n - s$ **then**
    Update: master_counter$_{i+1}$ = master_counter$_i + s - n$
else
    Update: master_counter$_{i+1}$ = master_counter$_i + s$
end-if
Output: master_counter$_{i+1}$

Algorithm 4.1: Master counter operation

[2] Control signals are highlighted in red throughout all architecture diagrams of this chapter.

[3] In conventional transmission schemes, the higher the symbol's corresponding degree, the sooner it will be sent.

The above update, in essence, assigns to the master counter the number of symbols of the latest codeword, that have started being processed by the encoder, at the current clock cycle. There is nothing to argue about, the above information is more than enough to determine which symbols the current cycle is actually processing, and from there, the problem between continuous and fixed-size block data is completely solved.

It should also be added that the master counter is also the only master signal of the entire encoder module. Every other control signal is either derived directly from the master counter or synchronized to it when previous clock cycle values are involved. From a different angle, this setup functions as the *one master multiple slaves* mechanism, which is one of the best approaches for block-code-based applications in general and RS-code-based applications in particular.

For this project, deriving specific values for each control signal is considered typical logic operations. Therefore, this work will not be discussed further to focus more on the core RS algorithms of the project.

4.3.3. BUFFER

It can be said that the buffer is the simplest module in terms of implementation, because it is, in fact, just a collection of register-sets wired together. The main challenge lies in and only in determining the exact number of register-sets required. Here, that number is directly specified as $2s-1$, including s register-sets acting as pipeline input buffer, and the remaining $s-1$ used to store symbols that have not been calculated yet. A corresponding architecture for this module is proposed in Figure 4.7.

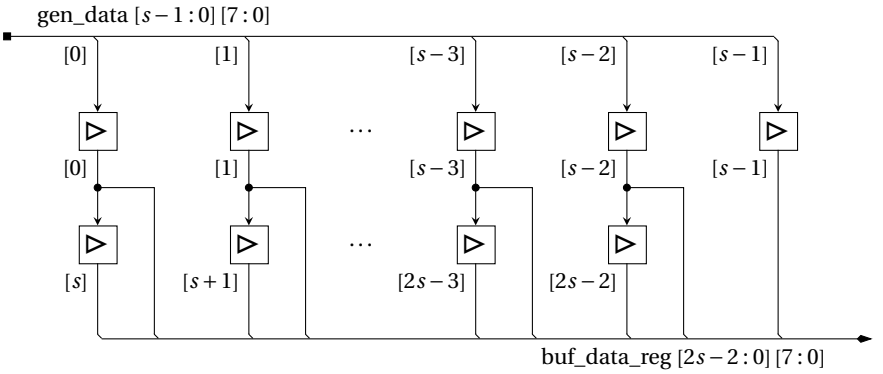


Figure 4.7: Encoder buffer architecture

The only aspect that needs to be clarified in this module is that the clock signal for these register-sets is no longer the system clock. Instead, the clock gating technique is applied to shut off the system clock to these register-sets while the *stall flag* is set, in order to ignore input data during this time. About the problem of when this stall flag is set, clearly, it is after the message has been completely transmitted, and the encoder is pushing out parity symbols.

4.3.4. PARITY CALCULATOR

As introduced earlier, *parity calculator* is truly the soul of the encoder, in which the most important calculation of the encoding process will be performed. The theory of encoding techniques in subsection 2.3.2 has proven that the systematic encoding algorithm, specifically Equation 2.22, directly divide the codeword into two portions: the message at $k = 239$ indices with the highest degree, and the parity, of course, at the remaining $n - k = 16$ indices. This module, therefore, is responsible for calculating that parity portion.

First of all, let $R_g[\xi(X)]$ represent the remainder of dividing $\xi(X)$ by $g(X)$, the calculation of parity portion can be rewritten as in Equation 4.4. Clearly, this is the perfect format for the linear feedback shift register (LFSR) architecture to flex its power. Here, the proposed architecture based on LFSR model is presented in Figure 4.8, followed by an in-depth analysis.

$$\begin{aligned} X^{16}u(X) \bmod g(X) = R_g\big[\ldots R_g\big[X \cdot R_g\big[X \cdot R_g\big[X \cdot X^{16}u_{238}\big] \\ + X^{16}u_{237}\big] + X^{16}u_{236}\big] + \ldots\big] + X^{16}u_0 \end{aligned} \tag{4.4}$$

Move on to the real-world problem, one that, like many others, involves handling continuous data. That data is then immediately stored to the buffer without undergoing any processing. Therefore, a mechanism to select appropriate symbols to process in each specific clock cycle is indispensable. It is not too hard to guess, since all necessary data is already correctly ordered in the buffer, the only missing piece is a variable to identify the starting index, which is implemented as a control signal named *offset*.

Continuing on, when it comes to the standard LFSR architecture, most people know that it is used to process a single bit or symbol each clock cycle. That said, it is too slow for an architecture that receives and produces s symbols per clock cycle, which is what the project is pursuing. To achieve so, the original architecture is extended by connecting s combinational logic computations in sequence before letting these results reach the shift registers. Despite the significant increase in resource usage, as well as a substantially longer critical path, the proposed architecture still offers impressive efficiency, thanks to well-

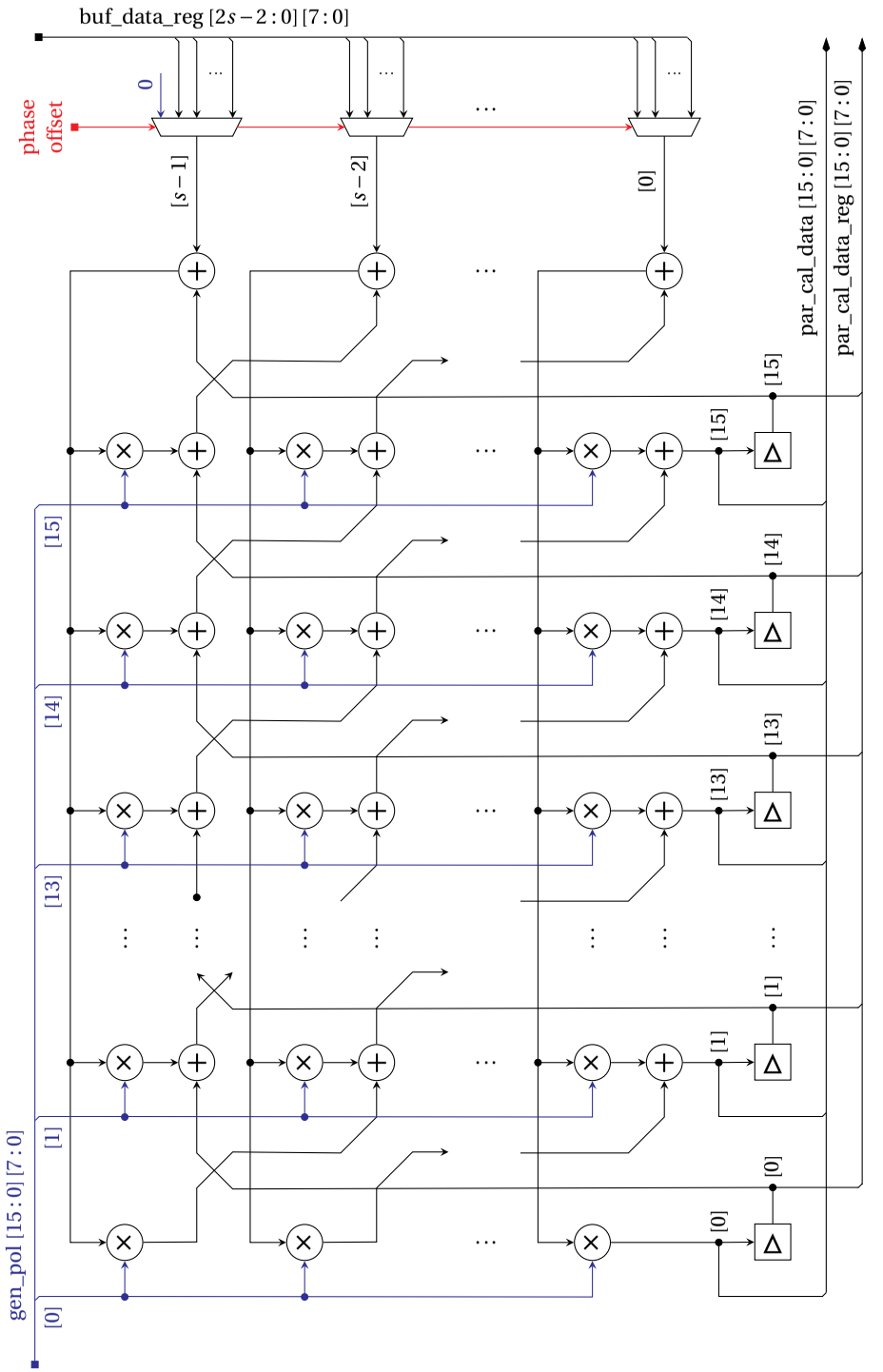


Figure 4.8: Parity calculator architecture

optimized foundational elements and the typically low-frequency demands of FPGA-based applications.

The final problem that needs to be handled arises from the fact that $k = 239 \equiv s-1 \bmod s \; \forall s \in \{2,4,8,16\}$, thus for each codeword, the architecture must handle fewer than s symbols in at least one clock cycle. Detailed evaluations in the timing domain were conducted before the final conclusion was reached that processing $s-1$ remainder symbols as soon as they are received will yield the greatest efficiency. To implement that idea in hardware, an additional *phase* is used, whose value depends entirely on the master counter and follows the set of values shown in Table 4.1. With such definition, setting the highest input symbol of the LFSR architecture to 0 whenever the phase is *first* is enough to solve the problem.

Phase	Description
idle	Idling and waiting for new message
first	Processing the $s-1$ remainder symbols
normal	Processing s symbols as normal

Table 4.1: Parity calculator phase descriptions

4.3.5. OUTPUT SELECTOR

If the controller is the big gate that brings continuous data into the RS algorithms, then output selector is the path that guides these fixed-size block data back into the continuous world of data. Looking back at the previous definition, when considering $s \in \{2,4,8,16\} \leq 2t$, it implies that the output data can only fall into one of the following four specific scenarios.

(1) These symbols belong exclusively to the message portion of a codeword.

(2) All these symbols fall within the parity portion of exactly one codeword.

(3) Some symbols with the highest indices are message symbols, while the remaining are parity symbols of the same codeword.

(4) Parity symbols occupy the highest indices, with the rest representing message symbols of the subsequent codeword.

Completely handling the above four cases requires just two control signals, one is a *phase* signal, whose corresponding description can be found in Table 4.2, and the other is used to determine number of message symbols out of the total s, which is referred to as *request*.

Phase	Description
idle	Default state of the control signal
message	The highest index of output data is a message symbol
parity	The output data's highest index lies a parity symbol

Table 4.2: Output selector phase descriptions

Finally, since either the message or parity portion exists under fixed-size block data in the encoder, two additional offsets are mandatory to indicate the two starting indices, which are referred to as *message offset* and *parity offset*, respectively. These above control signals are completely enough to start the process in Algorithm 4.2 to return the encoded data to the continuous form, which will actually be sent to the channel.

Input: input_data: message_data, parity_data,
Input: control_signals: phase, request, message_offset, parity_offset
if phase = *message* **then**
  Assign: output_data$[s-1:s-\text{request}]$
    = message_data$[\text{message_offset}+\text{request}-1:\text{message_offset}]$
  Assign: output_data$[s-1-\text{request}:0]$
    = parity_data$[\text{parity_offset}+s-\text{request}-1:\text{parity_offset}]$
else-if phase = *parity* **then**
  Assign: output_data$[s-1:\text{request}]$
    = parity_data$[\text{parity_offset}+s-\text{request}-1:\text{parity_offset}]$
  Assign: output_data$[\text{request}-1:0]$
    = message_data$[\text{message_offset}+\text{request}-1:\text{message_offset}]$
end-if
Output: output_data

Algorithm 4.2: Output selecting mechanism

Hardware implementation, as the final remaining step, is not a complicated problem either. Using the correct combination of multiplexers according to the defined algorithm is the only key to solving the problem.

4.3.6. ENCODER WRAP-UP

Summarizing the entire design, the encoder is encapsulated in a single module, with the full meaning of its interfaces presented in Figure 4.9 and Table 4.3, respectively. This information is also one of the most important for later system integration stage.

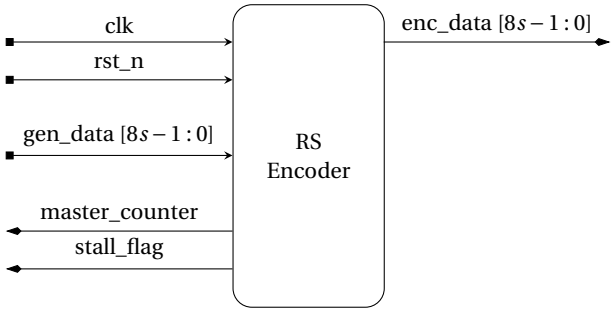


Figure 4.9: Encoder module

Name	Side	Width	Description
clk	Input	1 bit	Clock signal
rsn_n	Input	1 bit	Active low reset signal
gen_data	Input	$8s$ bits	Contains s consecutive symbols from the message data **u**
enc_data	Output	$8s$ bits	Contains s consecutive symbols from the codeword data **v**
master_ counter	Output	8 bit	Number of symbols from the latest codeword currently being processed
stall_flag	Output	1 bit	Indicate whether the encoder is available to receive a new gen_data

Table 4.3: Encoder interface descriptions

In addition, based on the design sections discussed earlier, the statistical number of elements used for each internal module are depicted in Table 4.4. Also, with the design's critical path including $2s$ adders, s half-multipliers, and several other multiplexers, the latency of the proposed encoder architecture is determined to be exactly one clock cycle.

Module	Inverters	Adders	Multipliers	Half-multipliers	Register-sets
Buffer	0	0	0	0	$2s-1$
Parity calculator	0	$32s$	0	$16s$	16
Output selector	0	0	0	0	s

Table 4.4: Finite field element usage in encoder

4.4. REED–SOLOMON DECODER DESIGN

Thanks to the brilliant minds of I. S. Reed and G. Solomon, the concept of a high-performance block code was introduced to both the scientific community and humanity as a whole. However, it was only when the success of the decoding process was validated by later scientists that RS code had a chance to find its way into practical applications. The decoder presented in this section is the hardware embodiment of all quintessence in the entire decoding process.

4.4.1. OVERALL ARCHITECTURE

If a comparison must be made, greater is probably not enough to describe the complexity of decoders compared to encoders. The decoding process must go through the full five steps as described in subsection 2.3.3, which results in internal complexity, latency, as well as resource usage that is many times higher than that of the encoding algorithm.

Under the time limit of the project, it is impossible to implement all of them to draw comparisons on the efficiency of each algorithm. Therefore, only one algorithm regarded with the highest confidence is selected for implementation at each step, and together they form an architecture like the one described in Figure 4.10.

- Firstly, *controller and synchronizer* is made up of two components, whose functions are closely linked. The controller part, as with the encoder, controls decoder's operations and generates required flags and counters. While the synchronizer part is unique to the decoder, it introduces a new concept tasked with implementing a *codeword self-synchronization scheme*.
- Secondly, the primary role of the *buffer* is to store data for the encoder's error detecting and correcting process. It also serves as a pipeline input buffer due to its earliest position in the processing flow.
- Thirdly, the *syndrome calculator* does what its name suggests, using the only method mentioned in the background section.
- Fourthly, to find the error-location polynomial, the IBM algorithm is chosen and implemented in the *BM algorithm solver*.
- Fifthly, the *Error locator and evaluator* is responsible for simultaneously performing the Chien search and Horiguchi–Koetter algorithm, fully taking advantage of the hardware's parallel processing capabilities.
- Finally, the *error corrector* will, if possible, correct errors and navigate these data to the output after passing through a pipeline output buffer.

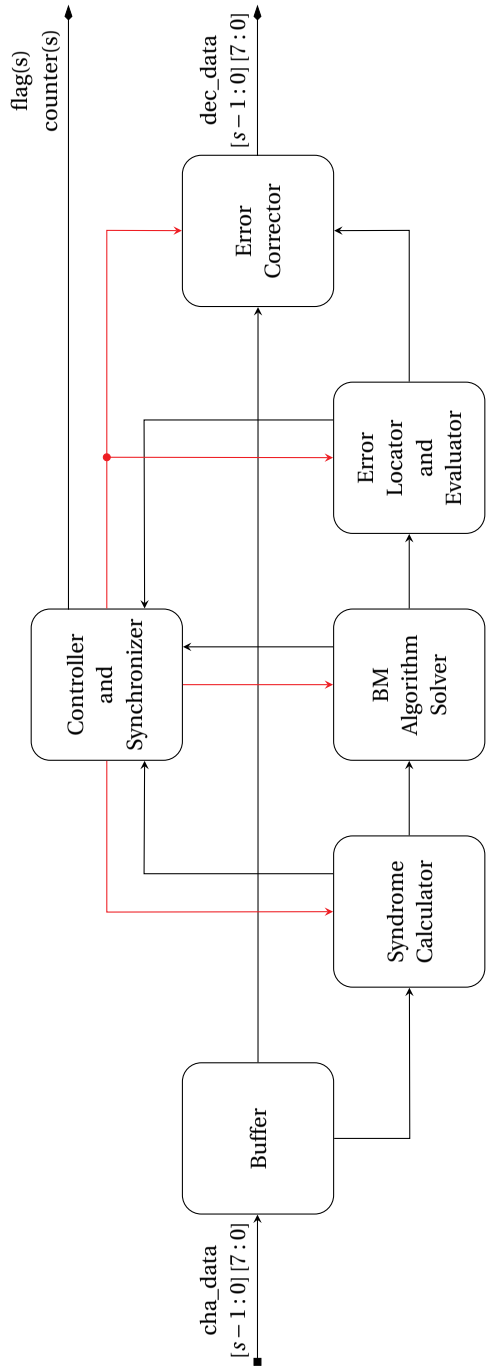


Figure 4.10: Decoder overall architecture

Note that all issues related to data rate, input and output interface, as well as the continuity of data outside the decoder, are inherited intact from the encoder side. Specifically, the input and output data interfaces continue to operate at s symbols per clock cycle, with $s \in \{2,4,8,16\}$. The input data received from the channel is guaranteed to be fully filled, while the output data towards the monitor can be empty for a certain number of clock cycles. Also, the interface facing the channel consists of only input data, while the opposite side is relaxed to permit other values such as flags and counters to be sent.

4.4.2. BUFFER

Buffer design for the decoder is not much different from that of the previously defined encoder module, as it continues to be a group of register-sets connected together. However, the number of register-sets required is dramatically larger, specifically $(\lceil 255/s \rceil + 15)\, s$, where $\lceil 255/s \rceil + 15$ represents total latency of the three modules syndrome calculator, BM algorithm solver, and error locator and evaluator, which will be accurately recorded in subsection 4.4.8.

Figure 4.11 below is the definitive finding of the above analysis. It should also be noted that the register-sets, from left to right, in the bottom row, correspond to the output symbols at indices $(\lceil 255/s \rceil + 14)\, s$ to $(\lceil 255/s \rceil + 15)\, s - 1$, which are omitted due to space limitations.

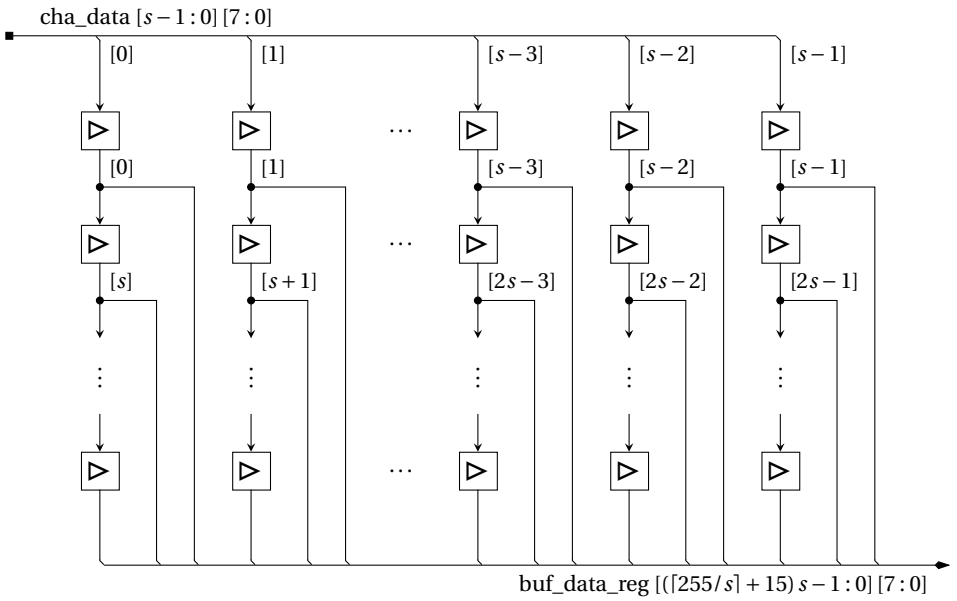


Figure 4.11: Decoder buffer architecture

4.4.3. SYNDROME CALCULATOR

As the decoding process begins, the syndrome calculator carries out the first one in the sequence of tasks. Building on the theoretical basis defined in subsection 2.3.4, syndrome calculation is clarified as the evaluation of the received polynomial at $2t = 16$ distinct points. Designing a module for polynomial evaluation can be presented directly due to its fundamentality, but only after the following issues are thoroughly addressed.

Primarily, accepting s symbols per clock cycle from the received polynomial, the syndrome calculator must, to minimize hardware resource usage, attempt to process this data as much as possible upon arrival. A strategy similar to the encoder side is used, meaning that $n \bmod s = s-1$ symbols will be processed in the first clock cycle and s symbols in all remaining ones. Equation 4.5 provides an approach that, while seemingly more sophisticated, is better suited for evaluating a polynomial on hardware, commonly known as the Horner's method [31]. Immediately following, Equation 4.6 provides the adjusted formula to execute the correct number of symbols per clock cycle as described.

$$S_i = r\left(\alpha^i\right) = \alpha^i \cdot \left(\ldots\left(\alpha^i \cdot \left(\alpha^i \cdot r_{254} + r_{253}\right) + r_{252}\right) + \ldots\right) + r_0 \tag{4.5}$$

$$\begin{aligned} S_i = r\left(\alpha^i\right) = \alpha^{si}\Big[\ldots\Big[\alpha^{si}\Big[\alpha^{si}\Big[\alpha^{(s-2)i} r_{254} + \alpha^{(s-3)i} r_{254-1} + \ldots + r_{254-s+2}\Big] \\ + \alpha^{(s-1)i} r_{254-s+1} + \alpha^{(s-2)i} r_{254-s} + \alpha^{(s-3)i} r_{254-s-1} + \ldots + r_{254-2s+2}\Big] \\ + \alpha^{(s-1)i} r_{254-2s+1} + \alpha^{(s-2)i} r_{254-2s} + \alpha^{(s-3)i} r_{254-2s-1} + \ldots + r_{254-3s+2}\Big] \\ + \ldots] + \alpha^{(s-1)i} r_{s-1} + \alpha^{(s-2)i} r_{s-2} + \alpha^{(s-3)i} r_{s-3} + \ldots + r_0 \end{aligned} \tag{4.6}$$

The next problem comes from the timing constraints for calculating according to Equation 4.6, as it takes $\lceil 255/s \rceil$ clock cycles to complete. Clearly, with $s \in \{2, 4, 8, 16\}$, it takes only 255 clock cycles for the decoder to completely collect s received codewords, but up to 256 clock cycles to do the syndrome calculation for them, which will lead to a timing conflict. A possible solution is that some additional hardware must be generated, and a control signal named *phase* with a detailed description in Table 4.5 must be used for control purposes.

Finally, since this module is located right after the buffer, where data still remains in continuous form, converting it into fixed-size block format also is essential. An *offset*, which is responsible for pointing to the starting index of the data, of course, is a mandatory control signal. In addition, although not explained in detail by the algorithm, it can be said that there is a difference in how data is selected for calculation during the *overlap* phase, so the phase itself must also be part of this process.

Phase	Description
idle	Default state of the control signal
first	Processing the $s-1$ first received symbols
normal	Processing s symbols as normal
overlap	Processing both s last symbols of previous codeword and $s-1$ first symbols of the consecutive one

Table 4.5: Syndrome calculator phase descriptions

Of course, on a hardware platform, there is no reason to carry 16 evaluations out sequentially when parallelizing is a more optimal solution. Consequently, the architecture includes 16 evaluator sub-blocks, similar to that shown in Figure 4.12, will be deployed to simultaneously evaluate the received polynomial. It should be added that the above approach may not be strictly optimal, yet it successfully meets the requirements within an acceptable resource budget.

4.4.4. BERLEKAMP–MASSEY ALGORITHM SOLVER

It remains unclear whether by chance or design, but the module positioned in the middle of the executing chain holds the central algorithm of the decoding process. Although theo theory section has provided a viable solution for hardware implementation, additional refinements could still enhance its efficiency. Specifically, from Algorithm 2.1, the following properties can be derived.

At the beginning, during initialization stage, the algorithm defined that $l^{(0)} + l_B^{(0)} = \mu = 0$. After that, assuming $l^{(\mu)} + l_B^{(\mu)} = \mu$ holds true, Equation 4.7 is also considered valid. As a result, the basic induction outlined above confirms that $l^{(\mu)} + l_B^{(\mu)} = \mu$ for all $0 \leq \mu < 2t$. At this point, omitting $l_B^{(\mu)}$ can be done without affecting the algorithm's outcome.

$$l^{(\mu+1)} + l_B^{(\mu+1)} = \begin{cases} l^{(\mu)} + \left(l_B^{(\mu)} + 1\right) = \mu + 1 & \text{if } d^{(\mu)} \neq 0 \text{ and } l^{(\mu)} \leq l_B^{(\mu)} \\ \left(l^{(\mu)} + 1\right) + l_B^{(\mu)} = \mu + 1 & \text{otherwise} \end{cases} \tag{4.7}$$

The second problem to be addressed is the auxiliary error-location polynomial, whose degree can be up to $2t-1$ in the worst case where $v = 1$, despite having at most only t non-zero coefficients.

Consider step μ with $l_B^{(\mu)} = t-1$, suppose the error-location polynomial has not yet obtained the final result, or in other words, the $2t$ Newton identities in Equation 2.33 remain unsatisfied. Let $\mu' > \mu$ be the smallest step with $d^{(\mu)} \neq 0$. Equation 4.8 then determines the sufficient condition for the branching condi-

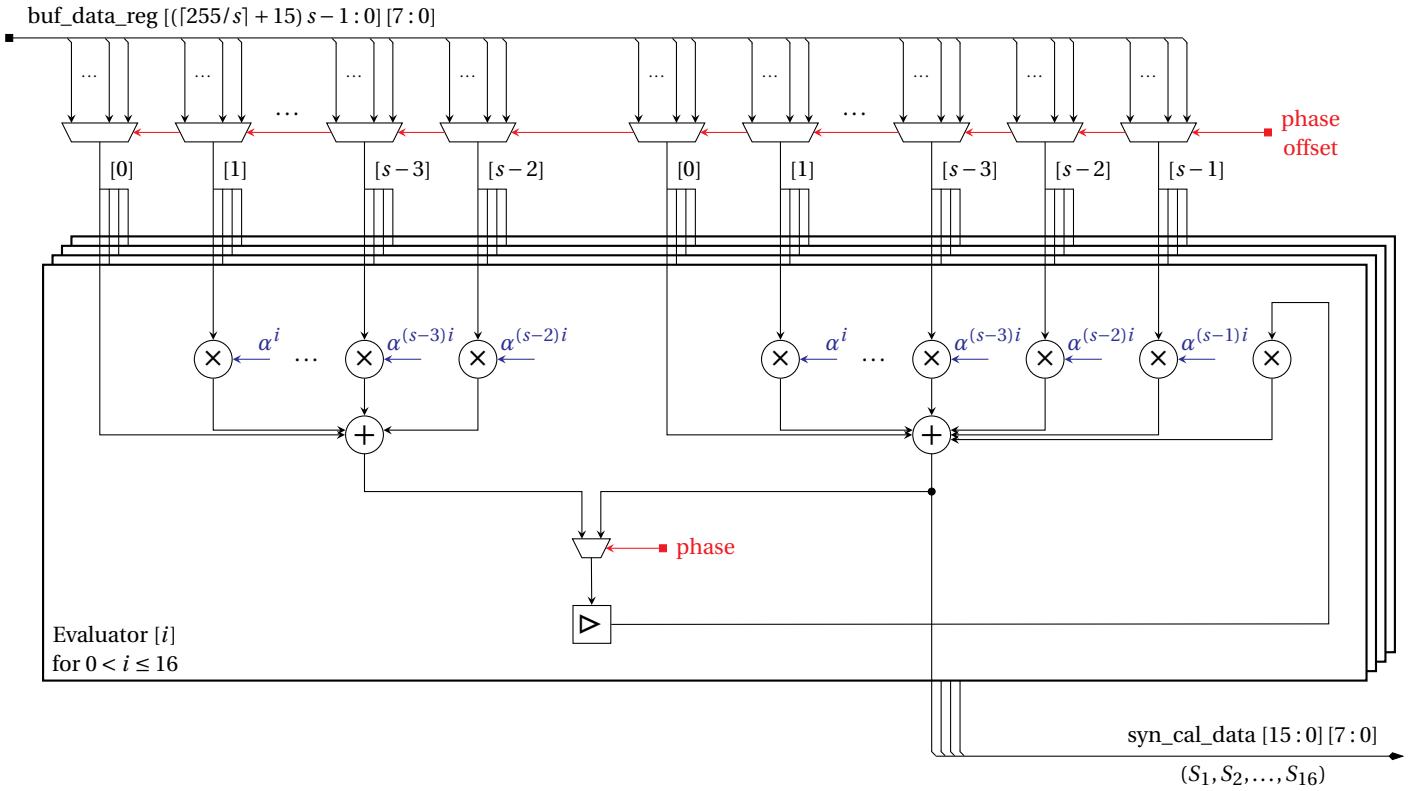


Figure 4.12: Syndrome calculator architecture

tion. Note that in the calculations, $l^{(\mu')} = l^{(\mu)}$ because there is no step that gives a non-zero discrepancy between μ' and μ.

$$\begin{aligned} l^{(\mu')} - l_B^{(\mu')} &= l^{(\mu')} - \left(\mu' - l^{(\mu')}\right) = 2l^{(\mu')} - \mu' = 2l^{(\mu)} - \mu' \\ &= 2\mu - 2l_B^{(\mu)} - \mu' = 2\mu - 2(t-1) - \mu' \leq 0 \end{aligned} \tag{4.8}$$

With $d^{(\mu)} \neq 0$ and $l^{(\mu')} \leq l_B^{(\mu')}$, the value of $l^{(\mu'+1)}$ is given by:

$$\begin{aligned} l^{(\mu'+1)} &= l_B^{(\mu')} + 1 = \mu' - l^{(\mu')} + 1 = \mu' - l^{(\mu)} + 1 \\ &= \mu' - \mu + l_B^{(\mu)} + 1 = \mu' - \mu + (t+1) + 1 > t \end{aligned} \tag{4.9}$$

4

Obviously, this is unacceptable since degree of the error-location polynomial is $v \leq t$. From that point onward, it is asserted that at step μ with $l_B^{(\mu)} = t-1$, the error-location polynomial reach its final claim.

Timing constraints form the final issue, as in the worst case, the BM algorithm solver is limited to 15 clock cycles to execute its task. Thus, the initial values are initialized at step 1 rather than 0, as suggested by theory. To summarize the whole discussion, an official algorithm will be determined right after.

Input: $S = (S_1, S_2, \ldots, S_{2t})$.
if $S_1 = 0$ **then**
  Initialize: $\sigma^{(1)}(X) = 1 + S_1 X$, $B^{(1)}(X) = \mathrm{X}$. $\gamma^{(1)} = 1$, $l^{(1)} = 0$.
else
  Initialize: $\sigma^{(1)}(X) = 1 + S_1 X$, $B^{(1)}(X) = 1$. $\gamma^{(1)} = S_1$, $l^{(1)} = 1$.
end-if
for $\mu = 1, 2, \ldots, 2t-1$ **do**
  Calculate: $d^{(\mu)} = \sum_{i=0}^{l^{(\mu)}} \sigma_i^{(\mu)} \cdot S_{\mu-i+1}$.
  Update: $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d^{(\mu)} / \gamma^{(\mu)} X \cdot B^{(\mu)}(X)$.
  if $d^{(\mu)} \neq 0$ **and** $2l^{(\mu)} \leq \mu$ **then**
    Update: $B^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$, $\gamma^{(\mu+1)} = d^{(\mu)}$, $l^{(\mu+1)} = \mu - l^{(\mu)} + 1$.
  else-if $l^{(\mu)} \neq \mu - t + 1$ **then**
    Update: $B^{(\mu+1)}(X) = X \cdot B^{(\mu)}(X)$.
  else
    terminate
  end-if
end-for
Output: $\sigma^{(t+v)}(X)$, $B^{(t+v-1)}(X)$, $\gamma^{(t+v-1)}$, $l^{(t+v-1)}$.

Algorithm 4.3: Berlekamp–Massey algorithm for hardware implementation

Figure 4.13 provides a visual depiction of the BM algorithm solver architecture, showcasing the four most important sub-blocks, which are tightly interconnected to carry out the algorithm. Arranged from top to bottom and left to right, they sequentially calculate the discrepancy, update the error location polynomial, auxiliary error location polynomial, and auxiliary discrepancy of the algorithm. Of course, there are a few other sub-blocks in play for tasks such as updating the error-location polynomial's degree, counting the current steps, and so on, but they all operate on logic and basic arithmetic and do not affect the use of finite field elements.

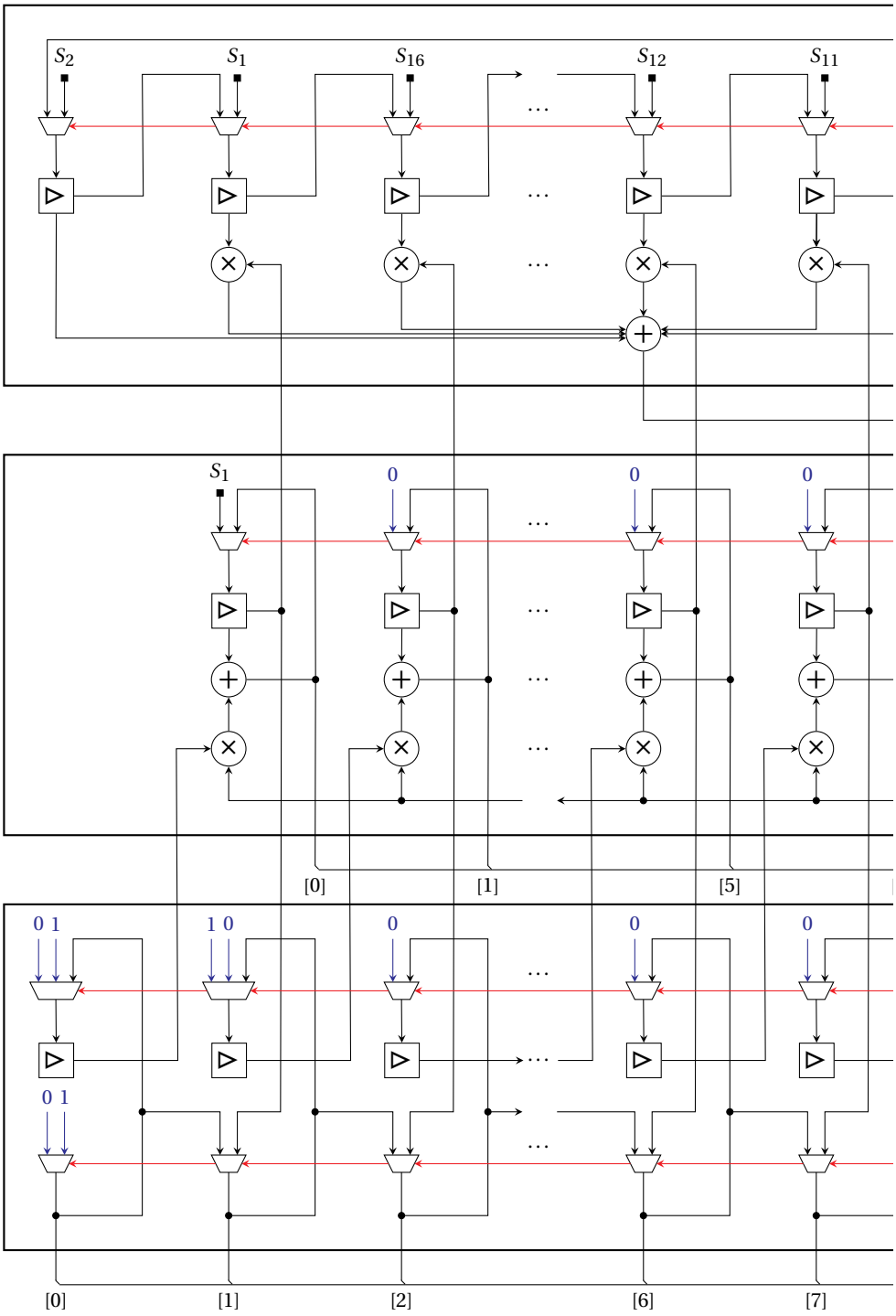
However, no matter how well designed it is, it still is powerless without proper control signals. Without a doubt, the most important of these is the *phase* that is directly derived from the master counter and regulates the entire module's behavior. In addition, several other internal control signals formed during the execution process are equally important in solving the problem itself. Much like the previous sections, a set of possible values for phase, along with descriptions for the internal control signals are shown in Table 4.6 and Table 4.7, respectively.

Phase	Description
idle	Idling and waiting for a newly codeword's syndrome
prepare	Collecting the finished syndrome and preparing to store it at the next posedge clock edge
execute	Executing the BM algorithm as normal
overlap	Both executing the BM algorithm for previous codeword and preparing for the next one

Table 4.6: Berlekamp–Massey algorithm solver phase descriptions

Signal	Description
step_counter	Representing the current value of μ
zero_flag	Indicating whether $S_1 = 0$ or not for initialization step
update_flag	Being the outcome produced by performing the logical operation $d^{(\mu)} \neq 0$ & $2l^{(\mu)} \leq \mu$
terminate_ flag	Being set when and only when *update_flag* is deasserted and $l^{(\mu)} = \mu - t + 1$ simultaneously

Table 4.7: Berlekamp–Massey algorithm solver internal control signals


S_2
S_1
S_{16}
S_{12}
S_{11}
S_1
0
0
0
[0]
[1]
[5]
0 1
1 0
0
0
0
0 1
[0]
[1]
[2]
[6]
[7]

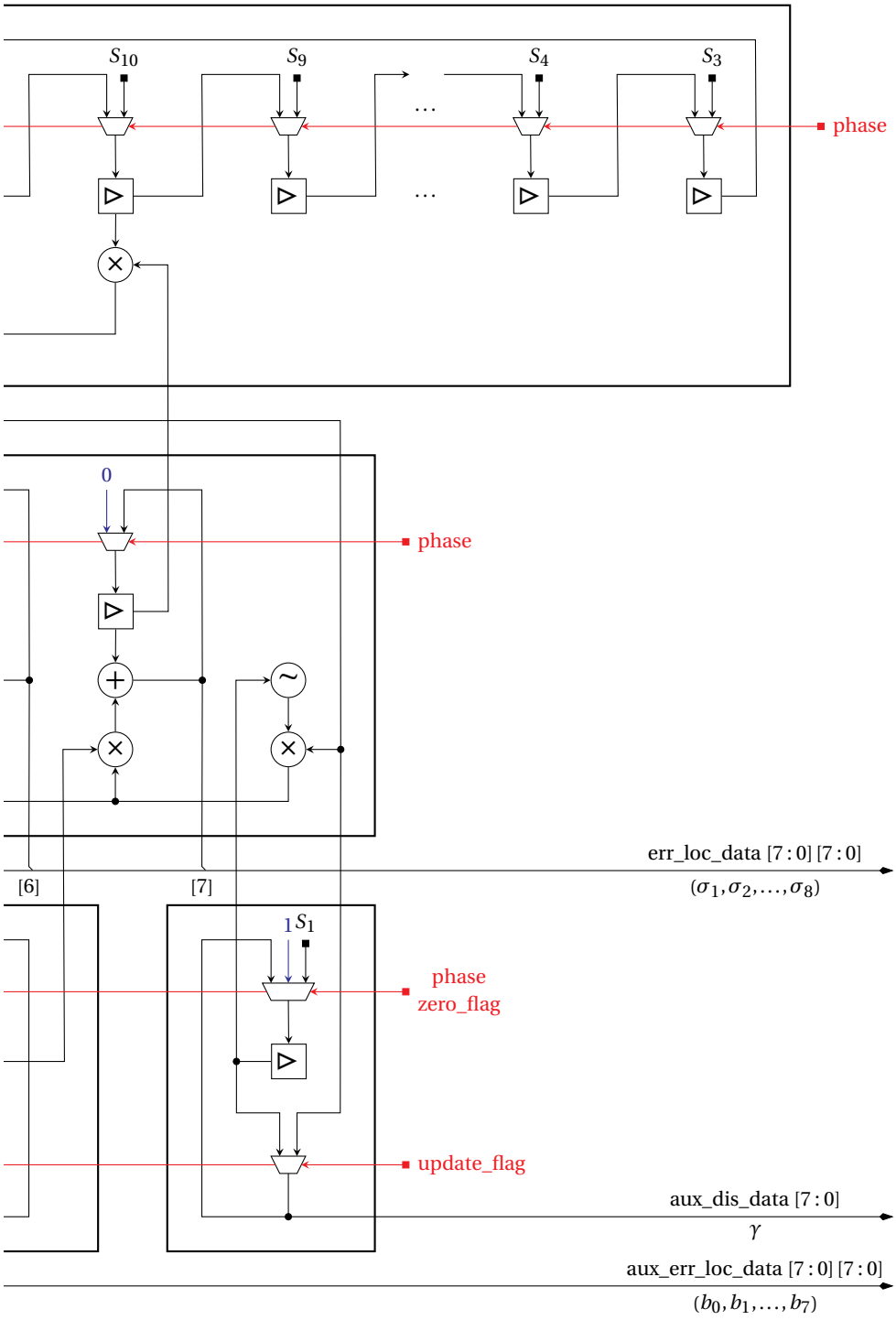


Figure 4.13: Berlekamp–Massey algorithm solver architecture

A final explanation is provided that to implement the algorithm's termination when the corresponding flag is set. The only thing required is to prevent the update of the register-sets in the two sub-blocks, including the auxiliary error-location polynomial updater and the step counter, since these are the only two that can still remain variable. Additionally, even though the final result is always 0, stopping the update of the register-sets in the discrepancy calculator sub-block can be considered to save energy. For the remaining sub-blocks, no further processing is necessary since their results are fixed as expected.

4.4.5. ERROR LOCATOR AND EVALUATOR

In the field of FEC in general, latency is an important aspect and can be traded off with a certain amount of resource. With the need to perform a large number of calculations, though not too complex, this module becomes the ideal place to realize the above idea. First of all, extreme cases will be detailed to clarify where the proposed strategy stands.

For the worst case, the error-location polynomial is evaluated at one position per clock cycle, resulting in exactly n cycles to complete. Then, up to t more clock cycles are needed to evaluate the entire error values. To sum up, there is an $n+t$ clock cycle latency with only one calculation sub-block for each task. Conversely, the entire computation can also be performed in a single clock cycle, of course, with a huge amount of resources, including n calculation sub-blocks for the error locator and k for the error evaluation step. The proposed approach, based on the decoder's specification that it inputs and outputs s symbols per clock cycle, will therefore also handle a comparable amount of computation. For convenience in the following, a list of values for the corresponding *phase* controlling this module will be provided in Table 4.8.

Phase	Description
idle	Idling and waiting for the next data set from the BM algorithm solver module
prepare	Gathering $\sigma^{(t+\nu)}(X)$, $B^{(t+\nu-1)}(X)$, $\gamma^{(t+\nu-1)}$, $l^{(t+\nu-1)}$ and preparing to store them at the next posedge clock edge
execute	Executing the Chien search and Horiguchi–Koetter algorithm as normal
overlap	Both executing the Chien search for $s-1$ last symbols of previous codeword and preparing for the next one

Table 4.8: Error locator and evaluator phase descriptions

ERROR LOCATOR

Although both syndrome calculator and error locator perform the same task of evaluating a polynomial, with a small degree and the need to be processed in one clock cycle, Horner's method is no longer an effective approach, while the naive surprisingly gives better results. Abstractly, the naive method evaluates the error-location polynomial at α^i, for $0 \leq i < n$, as follows.

$$\sigma\left(\alpha^i\right) = \sum_{l=0}^{v} \sigma_l \alpha^{il} \tag{4.10}$$

The order of evaluation operations also needs to be carefully managed. As the decoder receives and produces symbols in descending degree order, that is, from 254 down to 0, evaluation must progress sequentially from $\alpha^{-254} = \alpha^1$ to $\alpha^{-1} = \alpha^{254}$ before finally circling back to $\alpha^{-0} = 1$. This same order is followed during the error evaluation step discussed in the following section.

The last thing left is to reshape the original formula to best suit hardware-saving requirement. For generality, assume the evaluations at the jth clock cycle, with $0 \leq j < \lceil n/s \rceil - 1$, give results in a set of $\sigma\left(\alpha^{js+i}\right)$, for $0 < i \leq s$. Analysis of that result would proceed as follows.

$$\sigma\left(\alpha^{js+i}\right) = \sum_{l=0}^{v} \sigma_l \alpha^{jsl+il} = \sum_{l=0}^{v} \sigma_l \alpha^{jsl} \alpha^{il} \tag{4.11}$$

The calculation for the subsequent clock cycle would be as shown in Equation 4.12. Clearly, $\sigma_l \alpha^{jsl} \alpha^{sl}$ is the previously calculated factor from Equation 4.11, and the only need is to multiply it with α^{il}, for which the corresponding hardware already exists.

$$\sigma\left(\alpha^{(j+1)s+i}\right) = \sum_{l=0}^{v} \sigma_l \alpha^{(j+1)sl+il} = \sum_{l=0}^{v} \sigma_l \alpha^{jsl} \alpha^{sl} \alpha^{il} \tag{4.12}$$

In summary, the proposed architecture is detailed in Figure 4.14, along with some important notes. Firstly, the only data feedback to the register-sets belongs to the sth execute-evaluator, as theoretically expected. Secondly, the architecture includes s execute-evaluator sub-blocks in parallel, which output both the complete result and the results specific to odd-degree positions for error evaluation later. Thirdly, the timing conflict problem also occurs similarly to the syndrome calculator module, leading to the need for $s-1$ additional overlap-evaluation sub-blocks to produce complete results. Finally, there are a few other hidden sub-blocks that perform simple logic tasks, such as checking if the result is zero and raising the corresponding *error flag*s, as well as counting the current number of symbol errors in the codeword.

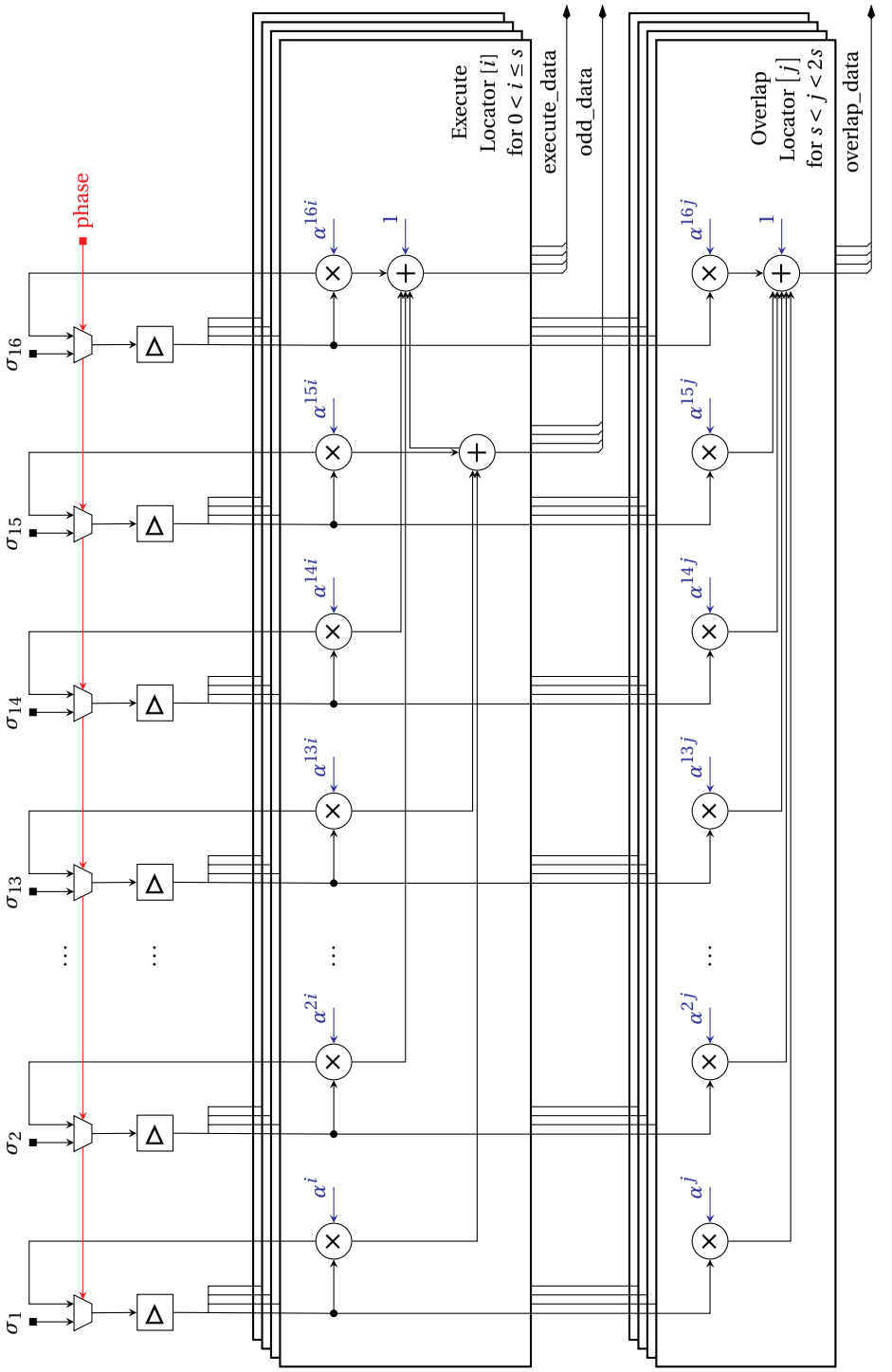


Figure 4.14: Error locator architecture

ERROR EVALUATOR

Looking back at the original formulation of the Horiguchi–Koetter algorithm in Equation 2.60 and the equivalent output of the final BM algorithm in Algorithm 4.3, it becomes evident that a mismatch exists and needs to be resolved.

To revisit some theory, although the algorithm terminates halfway through step $t+\nu$ and gives $\sigma^{(t+\nu)}(X)$, $\gamma^{(t+\nu-1)}$ as outputs, it is completely identical to the final results $\sigma^{(2t)}(X)$ and $\gamma^{(2t)}$. The only element that has not reached the final result is the auxiliary error-location polynomial. Since the algorithm produces $B^{(t+\nu-1)}(X)$, while it is not until $B^{(2t)}(X)$ that the final result is reached, multiplying it by $X^{t-\nu+1}$ is necessary. Additionally, computing the derivative in hardware also requires a bit of tweaking. Specifically, it can be brought to a more suitable form as done in Equation 4.13.

$$\begin{aligned}\sigma'(X) &= \sigma_1 + 2\sigma_2 X + 3\sigma_3 X^2 + \ldots + t\sigma_t X^{t-1} \\ &= X^{-1}\sigma_{odd}(X)\end{aligned} \tag{4.13}$$

Summarizing, the new version of the Horiguchi–Koetter algorithm is defined in the following form.

$$\begin{aligned}\delta_k = \frac{\gamma^{(2t)}\beta_k^{-2t+1}}{B^{(2t)}\left(\beta_k^{-1}\right)\sigma'^{(2t)}\left(\beta_k^{-1}\right)} &= \frac{\gamma^{(t+\nu-1)}\beta_k^{-2t+1}}{\beta_k^{-t+\nu-1}B^{(t+\nu-1)}\left(\beta_k^{-1}\right)\cdot\beta_k^{1}\sigma_{odd}^{(t+\nu)}\left(\beta_k^{-1}\right)} \\ &= \frac{\gamma^{(t+\nu-1)}\beta_k^{-t-\nu+1}}{B^{(t+\nu-1)}\left(\beta_k^{-1}\right)\sigma_{odd}^{(t+\nu)}\left(\beta_k^{-1}\right)}\end{aligned} \tag{4.14}$$

Figure 4.15 provides a vivid visualization of Equation 4.14. Except for the $\sigma_{odd}^{(t+\nu)}(X)$, which has been calculated in the previously Chien search algorithm, the remaining calculations are divided into three sub-blocks as follows.

- Firstly, at the top left is the evaluators, which are responsible for evaluating the auxiliary error-location polynomial $B^{(t+\nu-1)}(X)$.
- Following this, sub-blocks located at the top right are tasked with calculating the numerator portion of the Horiguchi–Koetter formula.
- At last, the error values are fully determined in the remaining calculators.

Each region, as noted, is made up of s parallel sub-blocks, which simultaneously perform computations for s different elements of the finite field.

Finally, although it works in parallel with the error locator, this algorithm does not require additional hardware to handle overlaps, due to the fact that the overlapped symbols are the last, or parity symbols, in other words, so evaluating them is of no practical value.

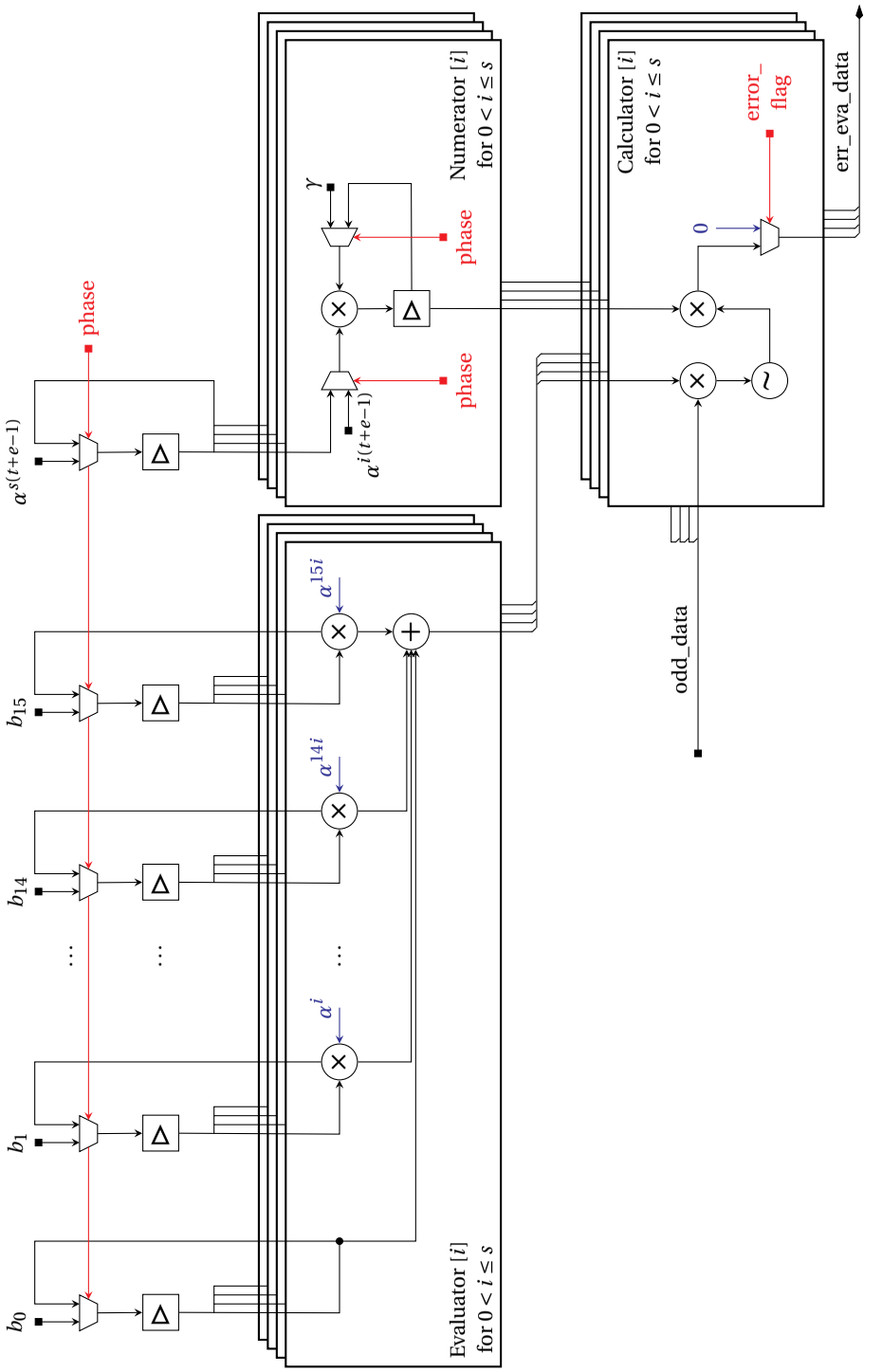


Figure 4.15: Error evaluator architecture

4.4.6. ERROR CORRECTOR

Serving as the last module in the processing chain, the error corrector's task is a bit more onerous than its theoretical equivalent. Specifically, for the error correction process to take place, it must find a way to align continuous data from the buffer with distinct error values for each symbol of the codeword.

Not too different in design ideas, two control signals are proposed to handle this problem, including an *correct offset* that points to the first position among the s symbols to be corrected, and a *correct flag* that indicates whether the current clock cycle needs to be corrected or not. Shortly, a corresponding pseudocode will be provided.

Input: input_data: received_data, error_values
Input: control_signals: correct_flag, correct_offset
if correct_flag **then**
    Assign: estimated_data = received_data
      + (error_values $\ll 8 \times$ correct_offset)
else
    Assign: estimated_data = received_data
end-if
Output: estimated_data

Algorithm 4.4: Error correcting mechanism

The second task, covered in this module, involves selecting the appropriate piece of data and sending it through the pipeline output buffer before output. Upon retrieving the four cluster types of s symbols predefined in subsection 4.3.5, it becomes clear that only three of them are brought to the decoder output, as the case consisting entirely of parity symbols is ignored.

However, with the choice of $s \in \{2, 4, 8, 16\}$, it is necessary to take full advantage of the property that $n \equiv k \bmod s$ to optimize the design. In detail, for two consecutive codewords, the total number of message symbols in the third and fourth cases is exactly s. Consequently, the dynamic shift, or dynamic selection from another perspective, can be implemented using a much more hardware-efficient method, masking.

Specifically, two more control signals will be used, namely *phase*, with corresponding descriptions in Table 4.9, and a *mask counter* to monitor the number of leftover message symbols from the previous codeword. As a final point, combining its operations completely, as shown in Algorithm 4.5, is the final key to completing this module.

Input: input_data: previous_data, next_data
Input: control_signals: phase, mask_counter
if phase = *masking* **then**
    Assign: output_data[$s-1$: mask_counter]
      = previous_data[$s-1$: mask_counter]
    Assign: output_data[mask_counter -1 : 0]
      = next_data[mask_counter -1 : 0]
else-if phase = *suspending* **then**
    Assign: output_data = previous_data
else
    Assign: output_data = next_data
end-if
Output: output_data

Algorithm 4.5: Data outputting mechanism

Phase	Description
idle	Default state of the control signal
normal	Processing an s-message-symbol cluster as normal
updating	Handling an s-symbol cluster, with message symbols at higher indices, followed by parity
masking	Handling an s-symbol cluster, with parity symbols at higher indices, followed by message
suspending	Ignoring input cluster as all of them are parity symbols
synchronizing	Acting similarly to the *normal* phase, but also taking the role of synchronizing correct offset and mask counter

Table 4.9: Error corrector phase descriptions

4.4.7. CONTROLLER AND SYNCHRONIZER

In terms of meaning, this module consists of two main functional sub-blocks, including the controller and the synchronizer. The controller, however, completely inherits the principles from the encoder, with a *master counter* maintained to control the operation of the remaining four modules performing RS algorithms. Therefore, further details are not necessary, and this section will concentrate solely on the conceptual design and corresponding implementation of the synchronizer.

First of all, it is necessary to understand several concepts about synchronization in general and codeword self-synchronization scheme in particular. Unlike the encoder, which can easily communicate with its data source, the generator, what the decoder actually receives is just a completely unstructured sequence of bits from the channel. In such a case, the controller cannot correctly determine the index of the symbol currently being processed, in other words, it cannot provide the master counter with the correct value, thus the decoder, of course, will not be able to function as expected. A synchronization scheme uses specific strategies to locate the starting point of the codeword and inform the master counter accordingly.

Obviously, adding more wires to the channel, or using one of them to transmit this synchronization signal, even a single one, is not possible as it negatively affects the data rate of the entire transmission process. Sending fixed chunks of data for synchronization is also not a good idea, because an unreliable transmission channel cannot guarantee that it will arrive intact. For that reason, the only way left is to make the decoder able to realign autonomously by leveraging the structural properties of the codewords themselves, without relying on external indicators. Such a mechanism is referred to as a codeword self-synchronization scheme.

With the noisy channel model and state transition probabilities both known, it is possible to predict parameters such as the probability at which a codeword is in error, uncorrectable, or even undetectable by RS algorithms. However, once the implementation process, in general, has been performed, the probability of an undetectable codeword becomes meaningless, while the probability of uncorrectable is no longer strictly accurate due to the minimum Hamming distance of the RS code being only $n-k+1$. Therefore, only the probability that a codeword is in error will be analyzed using Equation 4.15, with the meanings of the symbols referenced from subsection 3.3.1.

$$\text{codeword\_error\_rate} \approx \sum_{j_s=1}^{n} \sum_{j_b=1}^{nm} \sum_{i} PrS_n^{j_s,j_b}(i) \tag{4.15}$$

Analyzing the results presented in Figure 4.16 shows that, for every specified error rate, there is a corresponding probability of a codeword being error and uncorrectable. If this number is much larger than expected, or in other words, if it is close to one, it is safe to assume that there is a very high chance of misalignment between sender and receiver occurring. Specifically, in the proposed scheme, a window consisting of a specific number of codewords will be used. In case the number of erroneous codewords within it is higher than a certain threshold, a conclusion will be made that the window did not correctly identify

the starting point of the codeword. The window will then be shifted exactly s symbols and continue tracking until a window with the correct starting point is determined. Since n and $s \in \{2,4,8,16\}$ are relatively prime, every possible window is guaranteed to be scanned.

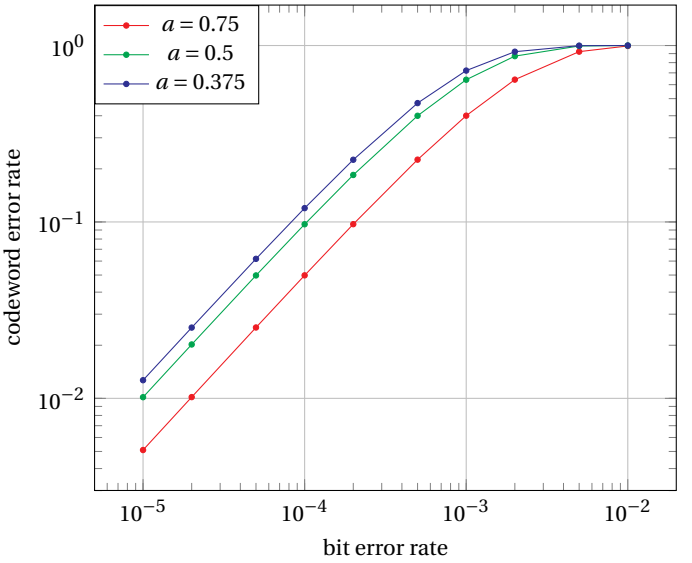


Figure 4.16: Codeword error rate of $RS(255,239)$ code

Summarizing, the codeword self-synchronization scheme is represented by a finite state machine in Figure 4.17, with Table 4.10 explaining the roles of each state. Note that even when the state machine reaches the *active* state, misalignment may continue to be occur due to alignment changes at the sender side, or other potential problems encountered during transmission. To add to that, because of the fact that message's length is always shorter than its equivalent codeword, a *valid flag*, with the opposite idea of stall on the encoder side is used. In particular, it is responsible for announcing the unavailability of data at certain specific clock cycles.

Finally, because setting values for the two parameters, window's size and threshold, has an important impact on the system's performance, specifically the time required for the synchronization task, they must be carefully considered. For optimal implementation, the size of a window will be chosen to be a multiple of s, namely $2s$. Also, according to the results from Figure 4.16, a window will be considered misaligned if all the codewords within it are erroneous, which is far beyond what the channel is capable of injecting.

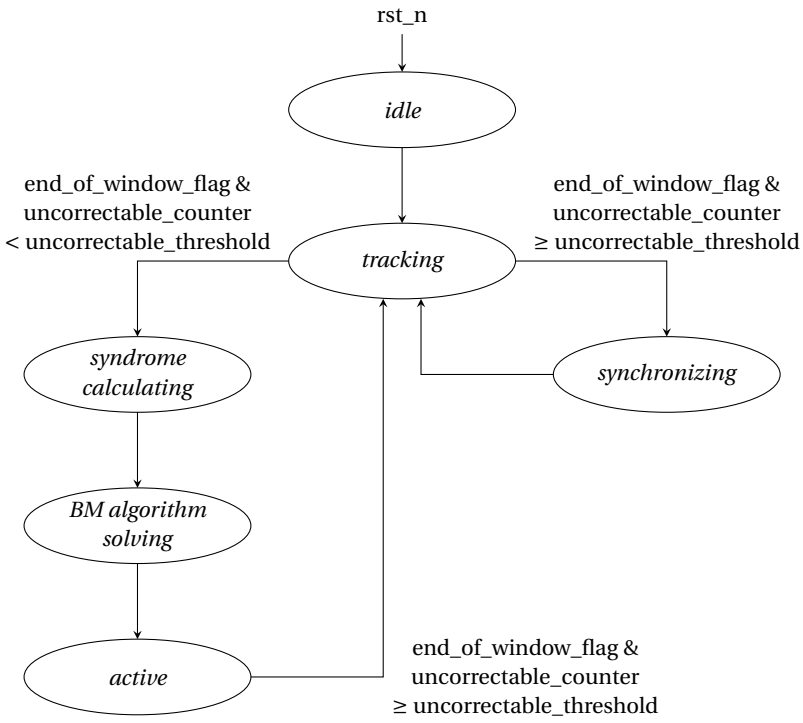


Figure 4.17: Codeword self-synchronization scheme

State	Description
idle	Default state of the control signal
tracking	Tracking the correctness of the starting index of the current window
synchronizing	Shifting the window by s symbols by stopping the master counter for a clock cycle
syndrome calculating	Calculating syndrome for the first correctly aligned codeword, with the valid flag disabled
BM algorithm solving	Solving the BM algorithm for the first correctly aligned codeword, with the valid flag disabled
active	Indicating data availability by enabling valid flag to function normally

Table 4.10: Synchronizer state descriptions

4.4.8. DECODER WRAP-UP

Wrapping up the entire decoder design into a single module, the final results are shown in Figure 4.18, with the meaning of its interface signals presented immediately afterward.

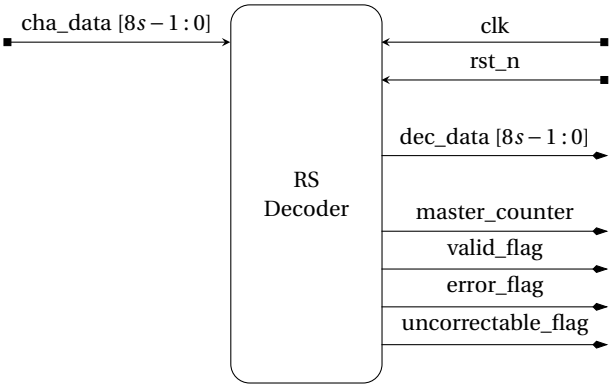


Figure 4.18: Decoder module

Name	Side	Width	Description
clk	Input	1 bit	Clock signal
rsn_n	Input	1 bit	Active low reset signal
cha_data	Input	8s bits	Contains s consecutive symbols from the received codeword data $\mathbf{r}$
dec_data	Output	8s bits	Contains s consecutive symbols from the estimated message data $\hat{\mathbf{v}}$
master_ counter	Output	8 bit	Number of symbols from the latest codeword currently being processed
valid_flag	Output	1 bit	Indicate whether the dec_data of current clock cycle is meaningful or not
error_flag	Output	1 bit	Assert in one clock cycle if the received codeword is error
uncorrect- able_flag	Output	1 bit	Assert in one clock cycle if the received codeword is uncorrectable

Table 4.11: Decoder interface descriptions

Summing up the designed modules, the total number of finite field elements used is listed in Table 4.12. Note that besides finite field elements, there are still

a lot of multiplexers, along with various other logic gates as well as FFs that serve these logic tasks.

Module	Inverters	Adders	Multipliers	Half-multipliers	Register-sets
Buffer	0	0	0	0	$15s+256$
Syndrome calculator	0	$32s-32$	0	$32s-32$	16
BM algorithm solver	1	16	17	0	33
Error locator and evaluator	s	$47s-16$	$3s$	$47s-16$	$s+33$
Error corrector	0	$3s$	0	0	$3s$

Table 4.12: Finite field element usage in decoder

It is clear from Table 4.12 that the number of elements used increases linearly with s. From a digital designer's perspective, this is the necessary trade-off for improving data rate. With the confirmed linear growth rate, these numbers can be considered technically acceptable.

Finally, in terms of overall timing, the decoder experiences a latency of $\lceil 255/s \rceil + 17$ clock cycles, a considerable timing cost compared to its encoder counterpart on the other end. In addition, the critical timing path, by design, can involve several cases. One such path starts from the register sets of the discrepancy sub-block and leads to the output of the error-location polynomial sub-block. This path traverses through five adders and three multipliers, all within the BM algorithm solver module. Another path, which could be critical, begins from the register sets in the error locator and evaluator module, and ends before data is stored to the output pipeline buffer. This path passes through one inverter, five adders, two multipliers, and one half-multiplier. Taking these paths into careful consideration and being willing to add additional pipeline stage buffers in case timing constraints are not met is the ultimate step toward perfecting the decoder's design.

5

DESIGN VERIFICATION AND SYNTHESIZED RESULTS

This chapter stands as the final milestone in completing the HDL design phase of the project. To begin with, the initial section is dedicated to verifying the correctness of the design, ensuring that all implemented modules behaves as intended. After a solid groundwork has been laid, the synthesis and implementation steps will follow, providing an honest reflection of the results achieved. Finally, comparison with existing works in the same field will be conducted to determine the rightful standing of the proposed architecture.

5.1. DESIGN VERIFICATION

A design, no matter how robust on paper, will hardly be accepted if it has not undergone a rigorous verification process. Especially in systems operating at data rates reaching several gigabits per second, particularly those employing RS codes, cannot afford to overlook this stage. This section is the tangible realization of the above inevitable need.

5.1.1. TEST ENVIRONMENT

When discussing a test environment, it denotes a collection of interrelated components working together to perform a single task, checking the correctness of the device under test (DUT). This environment can either be built manually or by following certain standards. Given the extremely strict timing requirements of this project, opting for manual construction could be a more ap-

propriate choice. Such an environment can be found in Figure 5.1, with the component's specific role provided shortly thereafter.

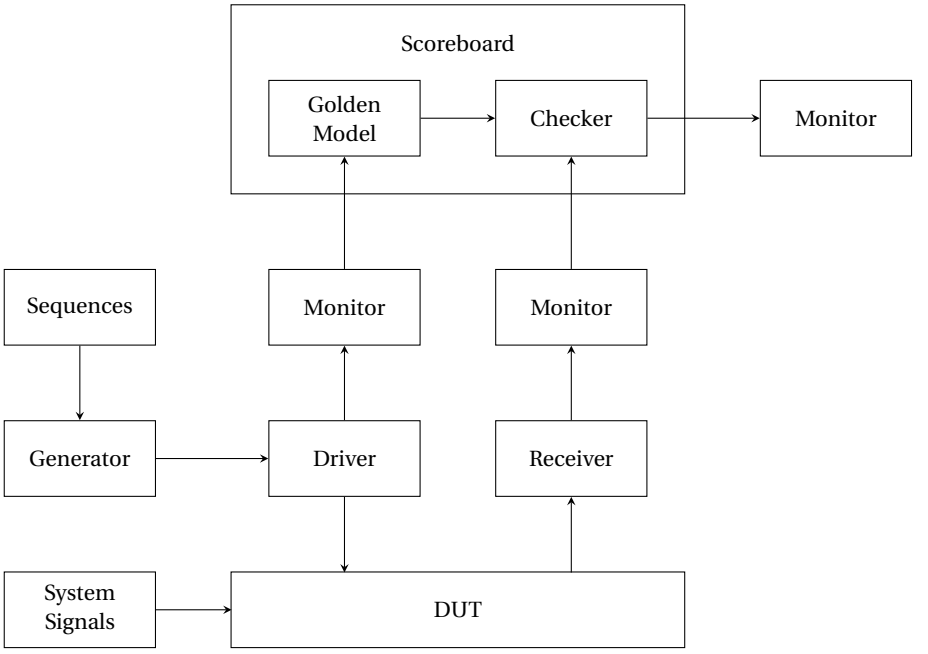


Figure 5.1: Test environment

First and foremost, the *system signals* responsible for generating the two most important signals of any design: one is the clock, the heartbeat of every digital system, and the other is the reset, a soft sigh that clears the whole mind before the next beat begins.

Closely related to the component just mentioned, the *generator*'s task is to generate the entire remaining input for the DUT. Input data here is usually composed of two main types, either random data for modules with simple behavior, or from fixed text files for ones with complex computational volume. Such input data packages, referred to as *sequences*, are combined in various ways to create the desired test scenario, and to be able to be applied in a systematic manner.

The *driver* and *receiver* are the only two components that communicate directly with the DUT, which perform two completely opposite functions, writing to and reading from the DUT, respectively. In each such read and write action, the corresponding values are displayed on the *monitor* to serve the purposes of verification and, if necessary, debugging.

Appearing as one of the most important modules of the verification process, the *golden model* is a parallel operating component, with the same functionality and input data as the DUT, but on a platform recognized for its near-perfect reliability. When it comes to mathematical computations for block codes in general, matrix laboratory (MATLAB), which stands out with its extensive support and dependable built-in functions, is an ideal platform. Besides, for modules with minimal computational demands, the original SystemVerilog can still be used with confidence.

Finally, the two results from the DUT and the golden model will be compared with each other at the *checker*, and the final outcomes will also be clearly displayed on the monitor, completely ending the test environment.

5.1.2. TEST PLANS AND OBSERVED RESULTS

With the designed test environment and depending on the inherent characteristics of each element or module, detailed test plans for each of them was constructed. With a comprehensive verifying strategy and positive results obtained in Table 5.1, it is completely possible to confidently affirm that the design fully meets the requirements of the $RS(255,239)$ encoder and decoder.

5.2. SYNTHESIZED RESULTS

In order to perform the synthesis, this project uses AMD Vivado™ ML edition 2023.1, with the target board being the Kria KV260 Vision AI starter kit. With the limited number of IOBs available on the kit, the design can only be synthesized with $s \in \{2,4,8\}$, while the case $s = 16$, although satisfying all functionalities, must still be excluded.

5.2.1. DESIGN CONSTRAINTS

No matter which platform the design is implemented on, the first and most important step is to define a set of constraints that are specific and suitable for the design. For this project, timing is the sole focus of the constraints, while the others follow the synthesis tool's default configuration.

First of all, regarding the clock frequency, there are two frequencies that will be used, including 100 MHz, corresponding to a 10 ns clock period, and 166.6667 MHz, corresponding to a 6 ns clock period. The first one, as mentioned before, is the default frequency, while the later can be considered as the maximum clock frequency. Another value that cannot be ignored is clock uncertainty. For this project, it will, however, be kept in the default configuration

No.	Module	Test name	Test description	Expected behavior	Results
01	System signals	Clock sensitive	Transmit different input data to any register, separated by a clock period	Output data after each positive clock edge is identical to the input	☑
02		Clock frequency	Rerun the test case no. 01 at multiple different clock frequencies	Functional behaviors stay the same across all clock frequencies	☑
03		Reset sensitive	Assert the active low reset signal	All register values, as well as outputs of every module are reset to zero	☑
04	Inverter	Zero inverse	Provide the inverter with the zero element	Result in the zero element instead of invalid as theorized	☑
05		Unit inverse	Provide the inverter with the unit element	Result in the unit element as theorized	☑
06		Normal inverse	Repeatedly provide the inverter with a random element α^i, for $0 < i < 255$	Result in α^{255-i}, with its binary representation shown in Appendix A	☑
07	Adder	Normal add	Repeatedly assign random elements for both inputs	Yield the bitwise exclusive-or of the inputs	☑
08	Multiplier	Multiple by zero	Sequentially drive the zero element for one input, then for both	Return the zero element for all cases	☑
09		Multiple by unit	Sequentially drive the unit element for one of the two inputs	Return the remaining input element	☑
10		Sub-limit multiple	Repeatedly drive the two inputs with α^i and α^j, such that $i + j < 255$	Result in the binary representation of α^{i+j}	☑

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
11		Over-limit multiple	Repeatedly drive the two inputs with α^i and α^j, such that $i + j \geq 255$	Result in the binary representation of $\alpha^{i+j \bmod 255}$	☑
12	Encoder's controller	Master counter test	Deliver the system clock and reset signal to the controller only, the reset signal is initially high, then falls low after some clock cycles	Behaves the same as the SystemVerilog golden model, built according to Algorithm 4.1	☑
13		Stall signal test		The stall signal is set if and only if $239 \leq$ master_counter < 255	☑
14	Encoder's buffer	Rejecting data test	Transmit random input data into the buffer while stall signal is kept at high	The buffer's data remains unchanged regardless of input	☑
15		Accepting data test	Transmit random input data into the buffer while stall signal is kept at low	The entire buffer shifts upward by $8s$ indices and accepts s symbols at the lowest positions	☑
16	Parity calculator	*first*-phase test	Fix the phase to *first*, randomize an offset, then generate $s-1$ data symbols at positions matching the offset, and bring all of them to the DUT	The obtained result is equivalent to $s-1$ single steps of the standard LFSR architecture from the MATLAB golden model	☑
17		*normal*-phase test	Fix the phase to *normal*, randomize an offset, then generate s data symbols at positions matching the offset, and bring all of them to the DUT	The obtained result is equivalent to s single steps of the standard LFSR architecture from the MATLAB golden model	☑

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
18	Output selector	Output data test	Randomize all possible input data following Algorithm 4.2	Match the behavior of corresponding SystemVerilog golden model	☑
19	Encoder	Single message encode	Generate a message and send it to the encoder over $\lceil 239/s \rceil$ clock cycles, s symbols each, with a zero symbol added in the final send	Continuously gain 239 message symbols, 16 parity symbols and an additional zero element after 1 clock cycle latency	☑
20		Two messages encode	Generate two consecutive messages followed by two additional zero symbols, then continuously send them to the encoder while the stall signal is low and zeros otherwise	Receive, after 1 clock cycle latency, two consecutive codewords in $\lceil 510/s \rceil$ clock cycles, s symbols each, with two zero symbols added in the final reception	☑
21		s messages encode	Redo the test case no. 20 with s messages and no added zero element	Receive s consecutive codewords in exactly 255 clock cycles	☑
22		Multiple messages encode	Repeatedly rerun test case no. 20, using a large number of messages	Equivalent to the output results from MATLAB golden model built for both encoding and processing data string to simulate continuous data	☑
23	Decoder's buffer	Buffering data test	Transmit random input data into the decoder's buffer	The entire buffer shifts upward by $8s$ indices and accepts s symbols at the lowest positions	☑

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
24	Syndrome calculator	Single codeword calculate	Form a codeword and an additional zero symbol then send them to the DUT over $\lceil 255/s \rceil$ clock cycles, with s symbols each	Result in 16 zero symbols as soon as data transmission is complete	☑
25		Two codewords calculate	Form two codewords and send them to the DUT over $\lceil 510/s \rceil$ clock cycles, s symbols each, with two zero symbol added in the final send	Result in 16 zero symbols when each codeword completes its data transmission, or at $\lceil 255/s \rceil$th and $\lceil 510/s \rceil$th clock cycles in other words	☑
26		s codewords calculate	Form s codewords and send them to the DUT over exactly 255 clock cycles, with s symbols each	Result in 16 zero symbols upon the completion of each codeword's data transmission	☑
27		Multiple received codewords calculate	Form a large number of codewords, then randomly and independently inject 0% to 10% symbol errors into, and finally send them to the DUT	Capture syndromes after each codeword finishes transmission are identical to the corresponding result from the MATLAB golden model	☑
28	BM algorithm solver	Zero-syndrome solve	Pass the zero-syndrome signal exclusively to the DUT	After 15 clock cycles of latency, except for $B(X)$ returning X^7, all remaining outputs return zero	☑
29		Nonzero-syndrome solve	Pass a randomly nonzero-syndrome signal exclusively to the DUT	After 15 clock cycles of latency, the entire output values must match those from the MATLAB golden model	☑

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
30		*s* syndromes solve	Rerun the test case no. 26 to collect times at which *s* complete syndromes are received, and then randomly pass *s* syndromes to the DUT accordingly	After every transmission 15 clock cycles of latency, all output values are identical to those from the MATLAB golden model	☑
31		*overlap*-phase test	Rerun the test case no. 30 but with fixed $s = 16$ for overlap occurrences		☑
32	Error locator and evaluator	Sub-limit evaluate	Use MATLAB to determine a polynomial $\sigma(X)$ whose degree is less than t with the same number of roots, then provide it, along with other randomly input values to the DUT	Output at every clock cycle is identical to the corresponding MATLAB golden model, which is string-processed to simulate evaluating *s* positions each clock cycle	☑
33		At-limit evaluate	Rerun the test case no. 32, but with an $\sigma(X)$ of degree t and exactly t roots		☑
34		Over-limit evaluate	Rerun the test case no. 32, but with an $\sigma(X)$ has fewer roots than its degree		☑
35		*overlap*-phase test	Rerun the test case no. 30 to collect the times when *s* outputs from BM algorithm are received, then randomize all inputs to the DUT accordingly	Outputs for each codeword are one-to-one equivalent to those obtained when operating individually	☑

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
36	Error corrector	Error correct	Randomize all possible input data following Algorithm 4.4	Match the behavior of corresponding SystemVerilog golden model	✓
37		Output data test	Randomize all possible input data following Algorithm 4.5	Match the behavior of corresponding SystemVerilog golden model	✓
38	Decoder's controller	Master counter test	Deliver the system clock and reset signal to the controller only, the reset signal is initially high, then falls low after some clock cycles	Repurpose the SystemVerilog golden model from test case no. 12	✓
39		Valid signal test		The valid signal is asserted when $0 <$ master_counter $\leq 17s-15$ or $17s+1 <$ master_counter ≤ 255	✓
40	Decoder without synchronizer	Single codeword decode	Generate a codeword followed by an additional zero symbol, then pass all of them to the decoder over $\lceil 255/s \rceil$ clock cycles, with s symbols each	After $\lceil 255/s \rceil + 17$ clock cycles latency, $240-s$ message symbols are consecutively output, the remaining $s-1$ symbols arrive along with the zero symbol after $2t/s$ more clock cycles	✓
41		Two codewords decode	Generate two codewords then pass all of them to the decoder over $\lceil 510/s \rceil$ clock cycles, s symbols each, with two added zero symbols in the final pass	Collect all output data whenever the valid flag is set, resulting in two complete messages followed by two zero elements	✓
42		s codewords decode	Rerun the test case no. 41 with s codewords and no added zero element	Receive s consecutive messages in exactly 255 clock cycles	✓

Continued on the next page ...

Continued from the previous page ...

No.	Module	Test name	Test description	Expected behavior	Results
43		Multiple codewords decode	Repeatedly rerun test case no. 41, using a large number of codewords	Equivalent to the output results from MATLAB golden model built for both decoding and processing data string to simulate continuous data	☑
44		Multiple received codewords decode	Repeatedly rerun test case no. 43, but with codewords randomly and independently injected with 0% to 10% symbol errors		☑
45	Decoder with synchronizer	Aligned test	Pass data into the decoder such that the meaning of the master counter is preserved	The decoder uses a single codeword window for tracking purposes, then produces data continuously	☑
46		s-symbol faster test	Pass data into the decoder such that the master counter is s symbols faster than the actual	The decoder uses two codeword windows for tracking purposes, then produces data continuously	☑
47		Any-symbol faster test	Pass data into the decoder such that the master counter is faster than the actual by ξ symbols, with $0 \leq \xi < 255$	The number of codeword windows required for tracking purposes is calculated by $1 + (\xi + 255\,(\xi \bmod s))\,/\,s$	☑
48		Alignment change test	Ensure the *active* state is in progress and functioning correctly, then try to change the sender's alignment by performing a reset	The receiver must acknowledge the alignment change, successfully regain synchronization, and continue function correctly	☑

Table 5.1: Test plans and observed results

of the synthesis tool. Experimental results indicate that this parameter is fixed at 0.035 ns for both clock frequencies targeted by this project.

The next consideration redirects to input/output delay constraints, one of the prerequisites for integrating the design into a larger system in general. With the design fully including the pipeline input/output buffer, setting these constraints becomes more obvious than ever. Table 5.2 displays the final results of this step, with each value being obtained experimentally to be appropriate for the FPGA platform being synthesized.

Clock frequency	100 MHz	166.6667 MHz
Minimum input delay	4.00 ns	3.50 ns
Maximum input delay	8.00 ns	4.00 ns
Minimum output delay	−2.00 ns	−2.00 ns
Maximum output delay	2.00 ns	−1.50 ns

Table 5.2: Input/output delay constraint

The final remaining issue concerns the most unique signal in the entire design, the asynchronous active low reset signal. Specifically, due to its asynchronous nature, any timing constraints must be relaxed, which necessitates setting it as a false path in the constraints file.

5.2.2. RESOURCE UTILIZATION

With the design and constraints ready, synthesis was performed fully automatically, and the results shown in Table 5.3 was obtained, for the resource utilization of the three versions and two clock frequencies of the design. Note that a more detailed breakdown of resource allocation for each internal module, as well as details on the type of resources utilized, is provided in Appendix B.

In the synthesis results, in addition to the LUT and IOB that have been introduced previously, the synthesis tool also introduces some new primitives, specifically as follows [32]:

- MUXF, a 2-to-1 look-up table multiplexer, which includes two variants, MUXF7 and MUXF8, is a special type of 2-to-1 multiplexer, useful in extending, and can be considered as LUTs as needed.
- D flip-flop with clock enable and asynchronous clear (FDCE) and D flip-flop with clock enable and asynchronous preset (FDPE), as their names suggest, are two types of D-type FFs that serve this project.

Module name	Clock frequency	Data interface	Data rate	LUT	MUXF	FDCE FDPE	IOB
	MHz	bits	Gbps				
Encoder	100	16	1.6	441	112	190	43
Decoder				2267	1	2925	45
Encoder		32	3.2	806	0	241	75
Decoder				3450	0	3234	77
Encoder		64	6.4	1602	32	340	139
Decoder				5866	16	3847	141
Encoder	166.6667	16	2.6667	440	112	190	43
Decoder				2260	1	2925	45
Encoder		32	5.3333	806	0	241	75
Decoder				3443	0	3234	77
Encoder		64	10.6667	1594	0	340	139
Decoder				5815	16	3847	141

Table 5.3: Synthesized resource utilization

The synthesis results demonstrate that the design reaches data rates up to 10 Gbps, which is a truly impressive achievement on an FPGA platform. That number can even be easily doubled, provided the target board has enough IOBs, specifically 269, or with some minor improvements such as separating the clock domains for receiving and processing data. This project, however, stopped at 10 Gbps as a final milestone in the design process.

5.2.3. EFFECTIVENESS COMPARISON

With rapid advancements, particularly in data transmission, more and more RS-code-based applications are becoming available on the market and on academic research papers. Initially, two prerequisite criteria for selecting a work for comparison purposes are that the work must be based on an FPGA platform and that the implemented algorithm must be $RS(255,239)$.

Besides, to aid in visualization, an informal metric called efficiency is introduced, with its formula given in Equation 5.1, fulfilling all the necessary conditions for comparison.

$$\text{efficiency} = \frac{\text{data rate}}{\text{LUT} + \text{MUXF} + \text{FF}} \left(\text{Mbps/primitives}\right) \tag{5.1}$$

Design	Device	Clock frequency	Data rate	LUT MUX-F	FF	Efficiency
Proposed	xck26-sfvc784-2lv [33]	100	1.6	553	190	2.15
		100	3.2	806	241	3.06
		100	6.4	1634	340	3.24
		166.67	2.67	552	190	3.60
		166.67	5.33	806	241	5.09
		166.67	10.67	1594	340	5.52
Dayal's [23]	xc4vlx15-12sf363	152.59	1.22	~ 246	150	3.08
Gianni's [34]	xc3s200	173.41	1.38	353	170	2.64

Table 5.4: Encoder design comparison

Design	Device	Clock frequency	Data rate	LUT MUX-F	FF	Efficiency
Proposed	xck26-sfvc784-2lv [33]	100	1.6	2268	2925	0.31
		100	3.2	3450	3234	0.48
		100	6.4	5882	3847	0.66
		166.67	2.67	2261	2925	0.51
		166.67	5.33	3443	3234	0.80
		166.67	10.67	5831	3847	1.10
Dayal's [23]	xc4vlx15-12sf363	152.59	1.22	3484	745	0.29
Mhaske's [35]	xc6lx16-3csg324	162.72	1.34	4018	2786	0.20
Tiwari's [36]	xc2vp50-5-ff1148	100.30	0.80	3463	994	0.18
	xc3s500e-4-fg320	100.00	0.80	3408	1043	0.18
Gianni's [34]	xc3s200	173.41	1.38	4224	1101	0.26

Table 5.5: Decoder design comparison

The results from Table 5.4 and Table 5.5 show the amazing performance of the proposed design measured against its counterparts in the same domain. Most noteworthy is the data rate of the proposed design, which can be considered evidently superior, as it breaks the approximately 1 Gbps threshold of its peers. In addition, decoder latency is another notable aspect. Although it is mentioned only in the work of P. Gianni and his colleagues [34], the 275 clock cycles latency far exceeds that of any version of the proposed design.

6

SYSTEM IMPLEMENTATION

In fact, RS codes, by themselves, are not enough to form a complete data transmission system. Other essential functional modules, such as those for packaging, compression, additional coding techniques, voltage balancing, digital signal processing, and even hundreds of others are required to construct a marketable system today. Given the current scale of the project, such a system is entirely impossible, and the RS code, therefore, will be treated as the only functional module in the journey of data.

This chapter will examine the designed RS codec as an IP and detail the process of building a complete data transmission system to integrate it into. For demonstration purposes, only the encoder and decoder with 32 bits on the data interfaces are selected and deployed on realistic hardware. The section afterward focuses on the actual implementation of the project in terms of communication between the sender and the receiver. The chapter will end after the third section, where a simple website is developed to provide a visual representation of how RS algorithms actually work.

6.1. REED–SOLOMON CODEC IMPLEMENTATION

As depicted in the proposed system from Figure 3.1, in addition to the IP core designed on the PL, both the sender and receiver contain functional blocks on the PS with the task of performing simple data processing that serves the IP core. Since implementing PS functional components, with the help of PYNQ, became a relatively painless task, the design effort was now concentrated on the communication between these two regions.

6.1.1. BLOCK DESIGN INTEGRATION

Given the nature of a continuous data processing application, the AXI direct memory access (DMA) is almost the perfect choice to serve this purpose, with its proposed architecture shown in Figure 6.1. Here, the design setup leverages Vivado's *run block automation* and *connection automation* to simplify the configuration of the Zynq processing system and ensure proper integration of AXI DMA with the necessary system components.

Starting with the Zynq processing system IP, the software interface around the Zynq processing system in general. The purpose of the configurations on this block is to allow for bidirectional communication between it and the PL, which results in two AXI4 interfaces being enabled, including M_AXI_HPM0-_FPD master and S_AXI_HP0_FPD slave port. The first one is used for initiating AXI transactions from the PS to the PL, specifically for controlling the AXI DMA and transferring data between the two. On the other hand, the later one allows the PL, through the AXI DMA, to access the double data rate (DDR) memory within it. Note that these configurations are automatically generated during the run block automation process for the Kria KV260 Vision AI starter kit.

After that, the AXI DMA, which is employed for high-speed data transfer between the previously introduced DDR memory and the IP core implemented in the PL, is taken into consideration. According to the data interface of the IP core, the *stream data width* is set to 32 bits. This width applies to all M-_AXI_MM2S and M_AXI_S2MM interfaces for memory-mapped read and write operations, as well as M_AXIS_MM2S and S_AXIS_S2MM for streaming data to and from the IP core. In addition to the most important one mentioned, some other minor configurations will also be set up to ensure the efficient operation of the whole process.

The two other blocks, namely AXI SmartConnect and AXI Interconnect, are responsible for bridging the Zynq processing system and the AXI DMA. The first block handles the adaptation of AXI transactions from the AXI DMA master interface to the slave interface of the Zynq processing system, while the second one, conversely, connects the master interface of the Zynq processing system to the AXI DMA's slave interface for configuration and data handling. Given that both the Zynq processing system and the AXI DMA operate with 32-bit data widths and the full AXI4 protocol, these two intermediary blocks mainly facilitate routing without requiring protocol or data width conversion.

Finally, the focus shifts to the RS codec implemented on the PL, which will be encapsulated within an *AXI4 stream wrapper* to enable seamless data communication with the AXI DMA and the Zynq processing system. Here, the wrap-

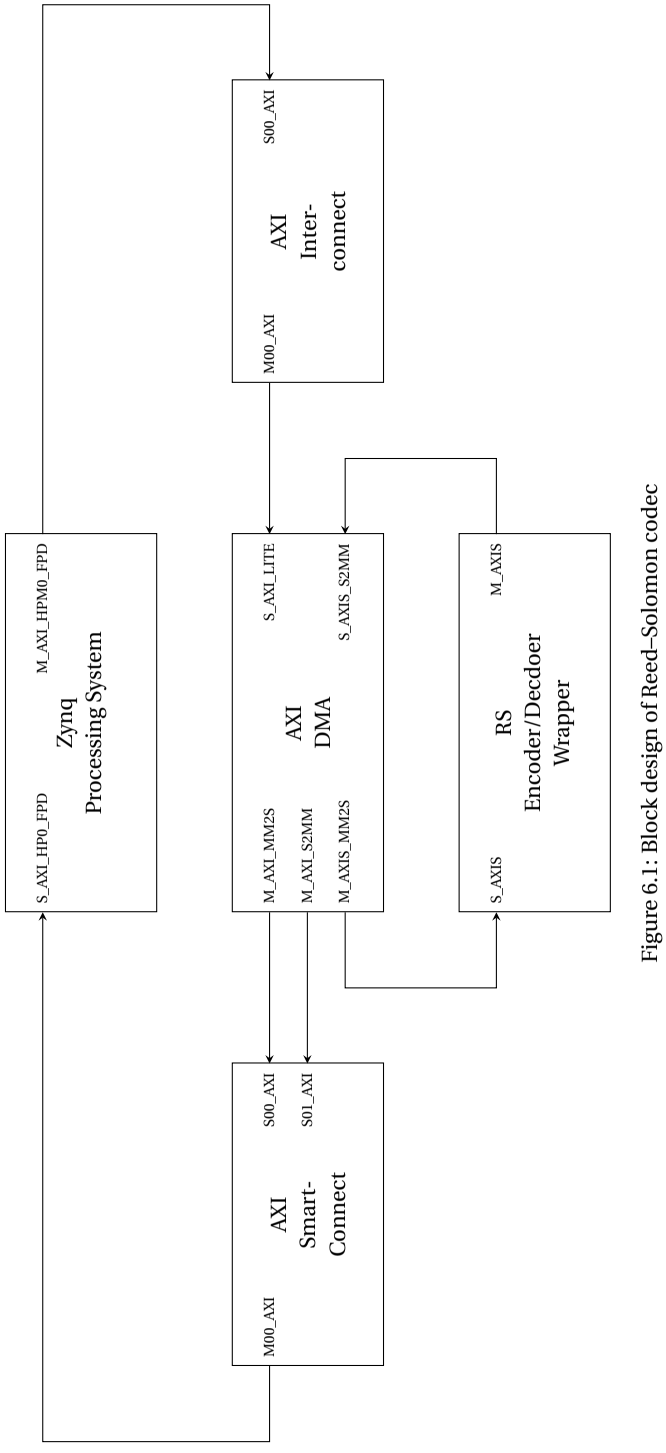


Figure 6.1: Block design of Reed–Solomon codec

per consolidates the input interface signals, including *tvalid*, *tready*, *tdata*, and *tlast*, into a single S_AXIS port, and the output interface signals into an M_AXIS port. Internally, it also handles the handshake mechanism and ensures proper signaling of data packets for reliable stream transactions. The S_AXIS port is then configured to accept input data streams from the AXI DMA's MM2S channel, while the M_AXIS port sends the processed output back to the AXI DMA's S2MM channel. This wrapping approach enables the original SystemVerilog module to interface with standard AXI4 stream peripherals without modifying its core processing logic.

6.1.2. IMPLEMENTED RESULTS

The completed architecture is then implemented on the target board, the Kria KV260 Vision AI starter kit. Here, the clock frequency is set to the standard 100 MHz instead of the maximum. This setting is chosen because it produces a pair of RS codec operating at 3.2 Gbps data rate, currently exceeding the Ethernet connection speed supported by the kit.

Although in reality nowadays, with the high demand of bidirectional communication, a device can be integrated with both encoder and decoder, this project is aimed only at simple communication between a fixed sender and receiver. Consequently, the resource utilization and power consumption will be statistically separated for each side. The final results of such an implementation are briefly provided in Table 6.1 and Table 6.2, showing the actual resource consumption of the kit to implement the entire system, instead of just the IP core as in subsection 5.2.2.

Incidentally, a full breakdown of resource utilization is supplemented in Appendix B, which serves as a reference for future studies requiring deeper analysis of resource usage.

Side	LUTs	FFs	Block RAMs	Ultra RAMs	DSP slices	Others
Sender	4890 (4.18%)	5296 (2.26%)	2 (1.39%)	0 (0.00%)	0 (0.00%)	1083 (2.35%)
Receiver	7322 (6.25%)	5997 (2.56%)	2 (1.39%)	0 (0.00%)	0 (0.00%)	1212 (2.91%)
Total available	117120	234240	144	64	1248	-

Table 6.1: Implemented resource utilization

Side	Static	Dynamic				
	PL	Clocks	Signals	Logic	Block RAM	PS
	W	W	W	W	W	W
Sender	0.301 (10%)	0.011 (< 1%)	0.014 (< 1%)	0.023 (1%)	< 0.001 (< 1%)	2.546 (88%)
Receiver	0.302 (10%)	0.014 (< 1%)	0.042 (1%)	0.054 (2%)	< 0.001 (< 1%)	2.546 (86%)

Table 6.2: Implemented power consumption

The resource utilization statistics reveal that the IP core occupies a significant proportion of the total resources, reflecting the optimization of block design process in this regard. Additionally, the total resource consumption is not large, being less than 5% of the overall kit, showing its potential for integration into other existing applications if they share the same ideals to pursue. Finally, the system's power consumption is well within the expected range, staying below 0.45 W, or less than 15% in ratio, across all PL elements, showcasing efficiency of the implementation process.

6.2. DATA TRANSMISSION SYSTEM IMPLEMENTATION

Once the individual sender and receiver implementations are finalized, the following mission is to integrate them into a unified data transmission system. Such a system, with specific equipment, is depicted in Figure 6.3, and its detailed explanations are provided below.

Revisiting the overall objective of the system, it involves two Kria KV260 Vision AI starter kits interconnected via Ethernet, with a website hosted on one of them. The problem arises right from its inception, when there is a concern that monitoring of results through the kit will be limited. Consequently, a small network, consisting of two kits connected indirectly through a router, was born. From this setup, other devices, most importantly the work laptops, can appear in the above network via Wi-Fi connection supported by the router, and take on some tasks that support the monitoring and presenting results to reviewers. For this project, the Home Wi-Fi Zte H196A router [37] was chosen, which supports Wi-Fi broadcasting standards up to 802.11ac and offers two gigabit Ethernet local area network (LAN) connections, perfectly fitting the project's needs.

Everything else now looks no different from the architecture proposed in subsection 3.1.1. The data starts its journey, upon request from the website,

from a C260 web camera [38], which easily produces VGA resolution images as required by the project's technical requirements. That data is received by the sender kit via the USB 3.0 port, and goes through its adventure as mentioned in subsection 3.1.2.

Finally, cannot fail to mention the website, where the visual images will be displayed. Specifically, it will be implemented as an hyper text transfer protocol (HTTP) service, receiving image upload requests from both the sender and the receiver. Note that a more detailed description of the behavior and possible interactions with the website is presented in section 6.3.

6.3. MONITORING WEBSITE DEVELOPMENT

With a unidirectional data transmission system as defined, clear differences emerge in how the sender and receiver interact with the website. In a more detailed explanation, the sender's workload is much heavier, as it must not only receive image capture requests from reviewers but also send two images to the website, twice the number sent by the receiver. As a logical outcome, the sender is assigned the role of hosting the website, while the receiver communicates its results via provided services when necessary.

6.3.1. INTERFACE DESIGN

With the ultimate goal of visually displaying the results during the RS codec's process of operation, a minimalist interface is provided by the website. On the top row, the website provides a live view screen, allowing reviewers to preview the image composition before capturing the shot using the button next to it. The three frames in the bottom row are where the most important display task is done, showing the original, the noise, and the decoded image, respectively.

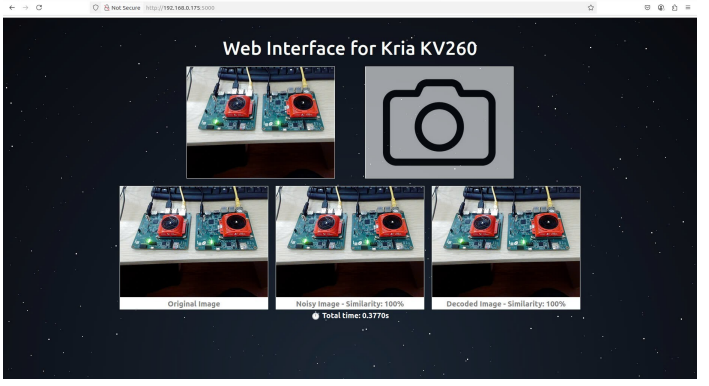


Figure 6.2: Visual representation of website interface

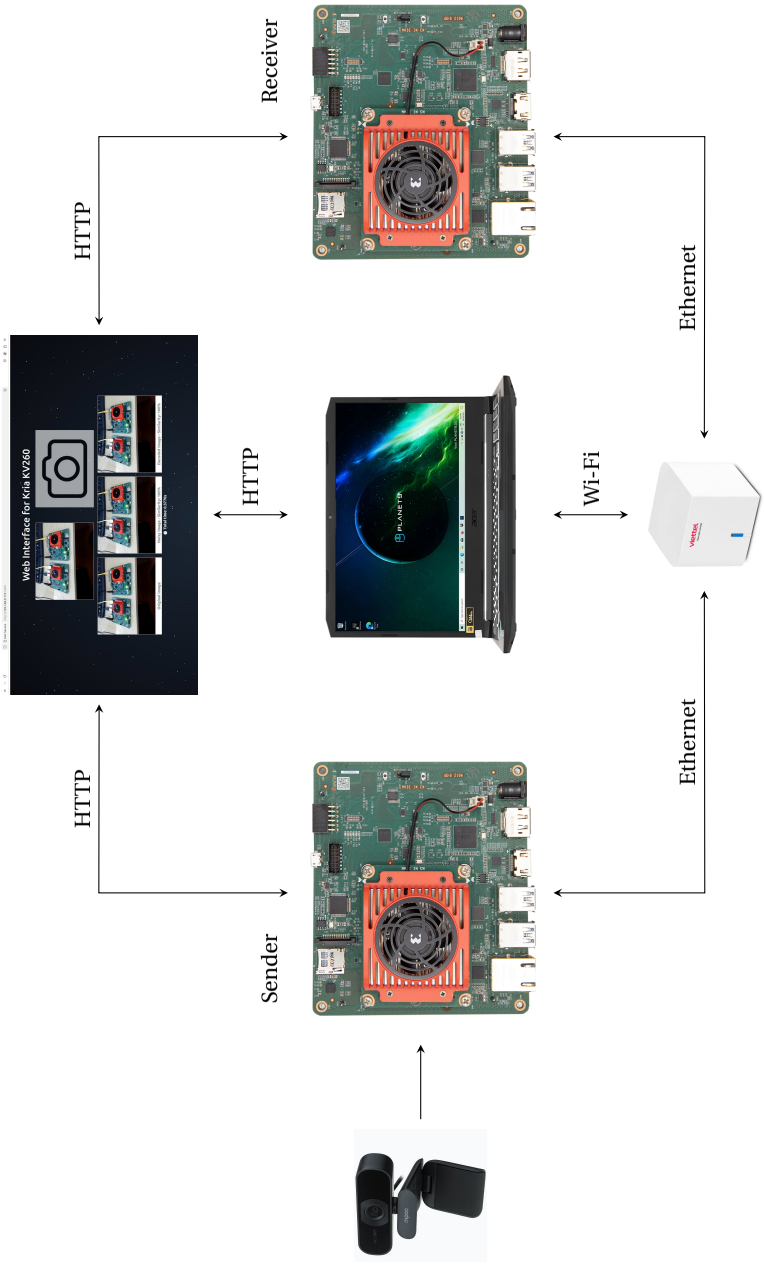


Figure 6.3: Implemented data transmission system

For noisy and decoded images, a value representing their similarity to the original image will be shown below each one, providing a more intuitive understanding of the issue. At the bottom of all of them is the time spent on the entire process, that is, the total time, which will be explained in detail in section 6.4.

In addition to the above, other metrics of the operation process, through different ways, will be collected and displayed on the terminal, and are excluded to keep the website simple, clean, and focused on essential functionalities.

6.3.2. ACTIVITY DIAGRAMS

Initially, with the simple website that only involves a single interaction from reviewers, and three additional initiated by the system, activity diagrams are enough to clearly describe every nook and cranny of the system.

At the sender side in Figure 6.4, the FPGA system is first triggered to boot. Afterward, the website starts working with the two main threads that handle its core functions as follows.

The sender side workflow is shown on the right, starting with a state where the system awaits a capture request from reviewers. Simultaneously, the captured image will be posted directly to the website while also undergoing preliminary processing, encoding, and finally, intentional error injection to obtain a noisy image. The noisy image is now concurrently posted to the website at the second frame and sent to the receiver side to complete its final task.

Situated on the left side, a service is available to support the receiver in posting the decoded image to the website. Specifically, it will be in a waiting state and will promptly perform its duty as soon as a request to post an image from the receiver is received.

As described in Figure 6.5, the activity diagram on the receiver side is somewhat simpler. Specifically, once started, it will fall into a state of waiting for data from the sender. Once the data is received, it will be identified as an image, which, of course, can be noisy, and the decoding process will follow. Finally, the decoded image will be indirectly posted to the website through the service presented on the sender side.

Finally, there is one important issue that cannot be left unmentioned. In the implemented system, each image, although it contains a relatively large amount of data, can still be considered an independent sequence of data rather than a streaming one. Based on the actual needs of the application, the data sent to the RS encoder and decoder will be done at certain times, which will ensure that they can correctly recognize which are the first symbols. This means

that the synchronizer, mentioned in the decoder design, will be disabled, and instead, a manual synchronization process, as described above, will be used. The above replacement, although somewhat underpowered, still ensures the key concepts of the design in general.

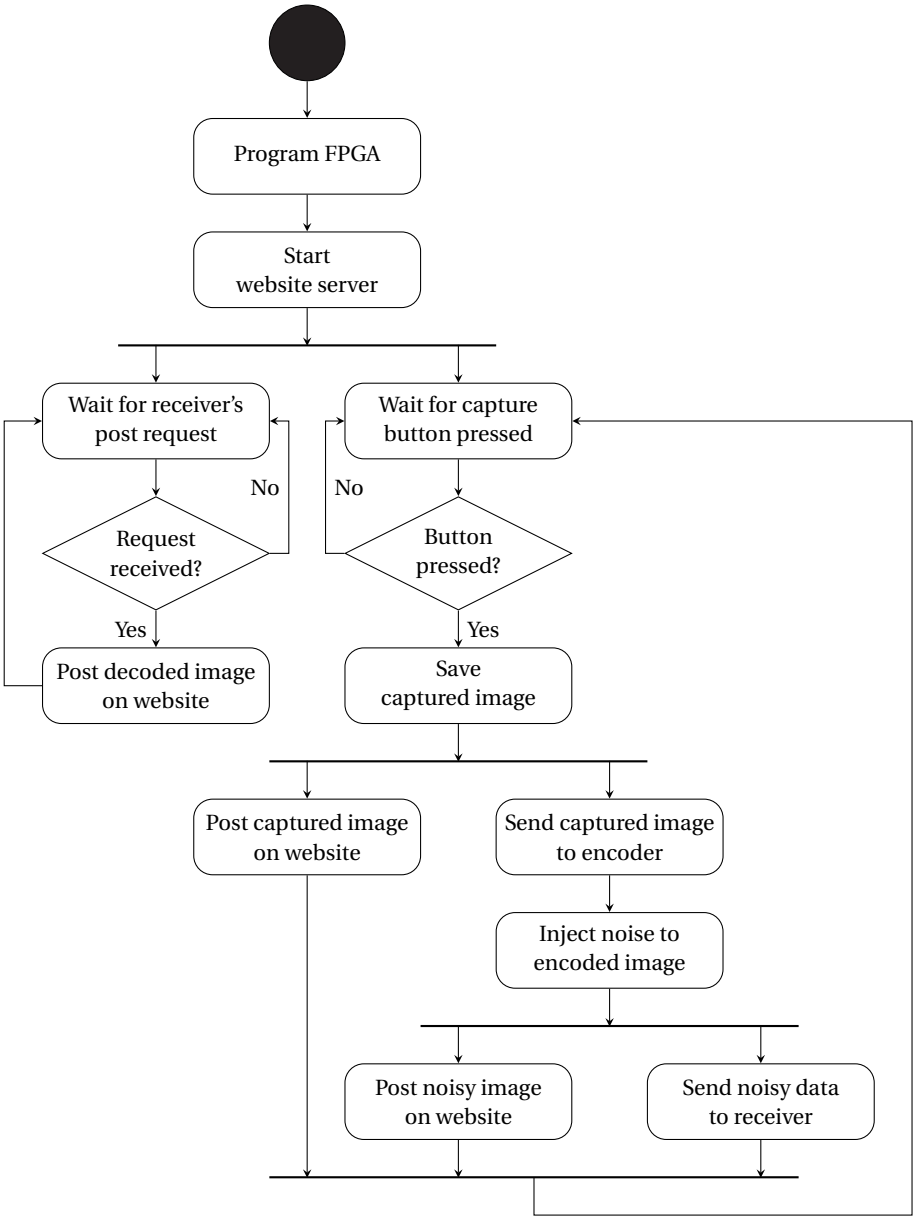


Figure 6.4: Sender's activity diagram

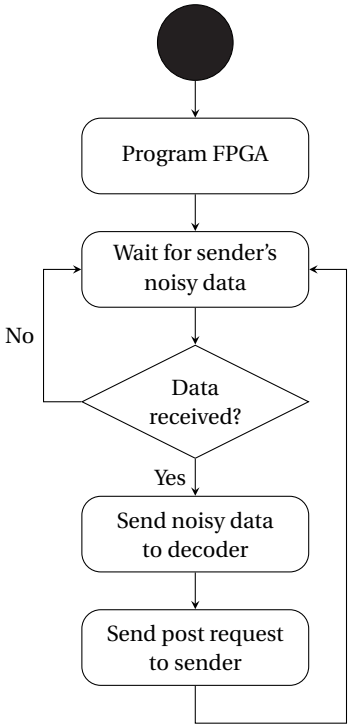


Figure 6.5: Receiver's activity diagram

6.4. SYSTEM PERFORMANCE EVALUATION

With the role of quantifying performance metrics, validating system behavior, and especially identifying strengths and weaknesses, system performance evaluation serves as the final step before any system is put into operation. Such a performance evaluation process will be presented in this section to demonstrate the robustness of the RS algorithms implemented on the PL, as well as the system's ability to effectively support their operation.

6.4.1. EVALUATION METRICS

To begin with, according to the theory outlined in subsection 2.1.5, BER and SNR are the two most important metrics for evaluating the efficiency of FEC in general and RS codes in particular. Specifically, by collecting the BER before and after applying RS algorithms, all other values can be inferred. Note that Equation 2.3 can be rewritten in the form shown in Equation 6.1, with $\Phi(x)$ being the cumulative standard normal distribution function, with subsequent calculations referencing statistical probability theory [39].

$$\begin{aligned} Q(x) &= 1 - \Phi(x) \\ \Leftrightarrow x &= \Phi^{-1}(1 - Q(x)) \end{aligned} \tag{6.1}$$

Additionally, for a system working with raw images, pixel-error rate (PER) is also a metric that should be taken into account. Similar to BER, two values, respectively before and after applying RS algorithms will be collected to indicate the difference level of noisy and decoded images compared to the original one.

From the system's viewpoint, the selected metrics mainly include the time required for multiple activities during its runtime. Among them, metrics that are the most meaningful are those that directly evaluate the effectiveness of the designed RS codec designs, which encompass the following.

- *Encoding time* refers to the time required to encode a raw image that adheres to the VGA standard, which has a resolution of 640×480 pixels, amounting to 7372800 bits or 900 kB. This metric accounts for the total measured time, including the AXI communication process, which will be provided alongside the theoretically calculated encoding time.
- Similarly, *decoding time* is the time required for the decoding process, under conditions similar to the concept just defined.

Besides the key metrics mentioned above, similar ones concerning the processing time of components in the PSs and transmission between the two kits are also deserving attention, specifically:

- Firstly, *injecting time* is the time for intentionally injecting errors into the encoded data, which, naturally, never happens in real-world applications.
- On the other hand, *packaging time* is defined as the time spent on the packaging and unpacking tasks performed on the PSs of both sides.
- As well, *sending time* denotes the duration from the moment that data leaves the sender until it arrives at the receiver.

Finally, it is impossible not to mention the time required to perform the website's communication tasks. For the purposes of this project, however, these are not metrics to be concerned with. In fact, this issue was addressed primarily to avoid causing a frustrating delay for reviewers, rather than being thoroughly optimized, as with the metrics presented. To demonstrate this, *total time*, defined as the time from when reviewers press the capture button on the website until all tasks of processing RS algorithms, as well as collecting and calculating metrics are completed, will be the only metric collected.

6.4.2. EVALUATION SCENARIOS

As the system was crafted with the intention of monitoring the operation of RS algorithms rather than enabling wide-ranging interactions, it has a fixed functionality and a single mode of operation. Therefore, from the system's perspective, that is also the only scenario in which the evaluation can be performed. Since this scenario is directly related to the implemented system, all defined timing metrics must be collected. Besides, in order to guarantee data integrity and consistency, multiple independent executions need to be performed and accurately measured.

In addition to the design implemented in PL, an equivalent function in PS will be used to evaluate the effectiveness in terms of timing consumption of RS algorithms. To ensure fairness, an existing Python library will be used instead of building the functions themselves, which could easily affect the processing time in a favorable direction. In this scenario, the only two measured metrics are encoding and decoding time, and both are compared against their counterparts from the deployed system.

The final aspect that needs to be evaluated concerns different rates of transmission errors, which are also the most important scenarios to evaluate the robust error correction capabilities of RS codes. In these evaluation scenarios, a total of tens distinct cases of pre-FEC BER, ranging between 10^{-2} and 10^{-5}, will be conducted to uncover the system's capabilities. As mentioned before, error injection is performed following the model defined in section 3.2, with some minor adjustments to maximize the pixel errors' observability.

6.4.3. EXPERIMENTAL RESULTS

After successfully setting up the whole system, it will first be operated at a default pre-FEC BER of 10^{-4} for a randomly taken image to collect runtime timing measurements. Starting with serving the first scenario, the corresponding results are detailed in Table 6.4, with the average and variance for each metric provided immediately afterwards.

Moving on, the second scenario requires measuring the execution time of the two RS encode and decode algorithms that are implemented as a software solution. Ten measurements were also performed as in the previous scenario and the corresponding results can be seen in Table 6.3. Note that the two algorithms above are implemented and measured as standalone components on PS without being integrated into the system, primarily due to the complexity and inefficiency of the development approach.

	Encoding time	Decoding time	Injecting time	Packaging time	Sending time	Total time
	ms	ms	ms	ms	ms	s
	4,4222	4,4057	137,3355	0,8369	30,1189	0,4110
	4,4134	4,4034	144,8338	0,9112	19,6540	0.3920
	4,3931	4,3957	142,7081	0,8380	15,1393	0,4020
	4,4131	4,4220	141,5646	1,2376	19,5036	0,4630
	4,4034	4,4093	139,0622	0,7930	14,5502	0,3980
	4,4298	4,3707	142,4828	1,6770	15,2030	0,5790
	4,4167	4,4017	135,5250	0,8669	14,4482	0,3810
	4,4601	4,3972	144,1586	0,8516	14,4606	0,4380
	4,4837	4,4172	139,4978	0,8447	19,9535	0,3690
	4,7083	4,3900	138,4649	0,9213	19,4972	0,5300
Average	4,4544	4,4013	140,5633	0,9778	18,2529	0,4363
Variance	0,0087	0,0002	9,3384	0,0760	23,3705	0,0047

Table 6.4: Experimental timing results for hardware solution

	Encoding time	Decoding time
	s	s
	20.9470	43.9659
	21.0780	44.2161
	21.0676	44.1660
	20.9069	43.9954
	20.8418	44.2164
	20.9596	43.9869
	21.0022	43.9841
	20.9126	44.2471
	21.0297	43.9830
	20.6711	43.9426
Average	20.9417	44.0704
Variance	0.0146	0.0153

Table 6.3: Experimental timing results for software solution

Once the evaluation of timing metrics has been done, the efficiency of the RS algorithms becomes a primary area of interest. Respectively, Figure 6.6 and Figure 6.7 present the measured post-FEC BER and PER, with missing values indicating that a zero rate or infinite ratio was received.

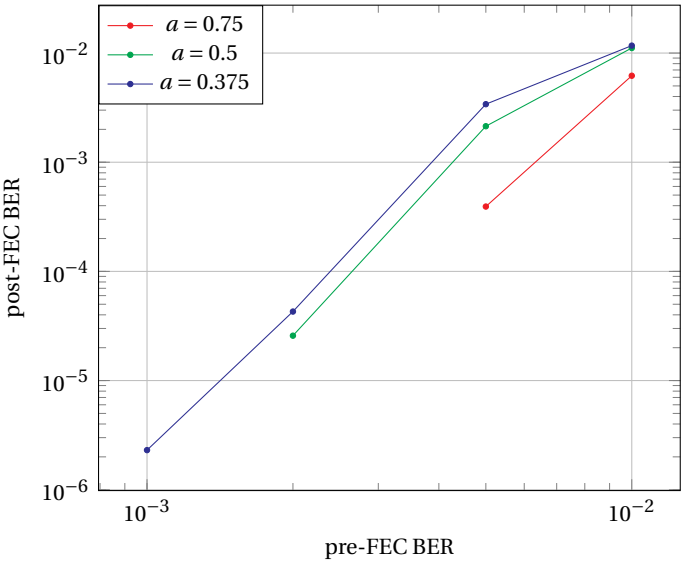


Figure 6.6: Experimental bit-error rate results

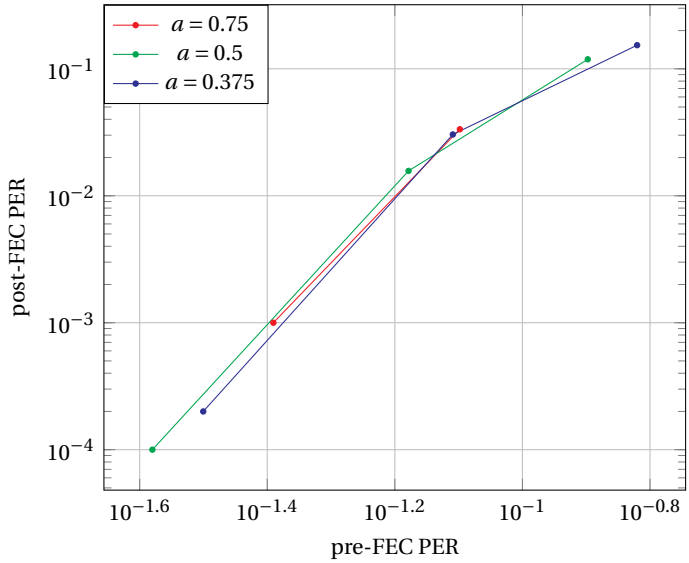


Figure 6.7: Experimental pixel-error rate results

6.4.4. PERFORMANCE ANALYSIS

First of all, it should be stated that the measured values related to encoding and decoding time are particularly stable, with coefficients of variation all less than 0.1%. Afterward, it was found that the execution time of each of the two RS algorithms is less than 4.5 ms, remarkably small for processing 900 kB of data. The above result is not just superior but far outpaces the software solution, exceeding it by thousands-fold, as the massive computations of RS code algorithms are truly a nightmare when deployed on this platform.

However, according to theoretical calculations, 2.4589 ms and 2.4597 ms are the times required for the RS encoder and decoder IPs to perform their tasks, respectively. Comparing the obtained results, it is crystal clear that only approximately 55% of the time is actually spent on the execution of the RS algorithms, while the remaining time is on AXI communication. Judging by the percentage, this number is not yet high and is still open to further optimization.

The remaining timing issues revolve mainly around packaging, as it is an essential step in the data transmission process within the system. According to the collected results, completion is guaranteed in less than 1 ms. Besides, injecting time, although the most time consuming, is a block that does not exist in any actual system and can completely be omitted. Finally, sending time and total time are the factors most affected by the quality of the network connection during the demonstration and, therefore, will not be analyzed in detail. Instead,

the only recognized result is the total time, which came in at an impressive time of less than 1 second.

In the end, the pair of metrics post-FEC BER and post-FEC PER comes together to showcase the steadfast reliability of RS codes in error correction. Although the error correction performance at pre-FEC BER = 10^{-2} is not really impressive, it quickly improves when the error rate starts to decrease slightly. And finally, it is possible to eliminate any transmission errors if the channel ensures that the error rate is less than 10^{-3}.

7

CONCLUSION

Summing up the two phases over the past 30 weeks, the project journey approaches the final chapter, and its outcomes stand as a meaningful testament to what was pursued and achieved. However, through a balanced and comprehensive lens, along with the achievements, the project also leaves behind traces of what could not yet be resolved and carries even unmet aspirations. In closing, this chapter serves as both a final summary of the project's journey and a genuine gratitude to those who have walked alongside its thoughtful and lengthy path.

7.1. ACHIEVED ACCOMPLISHMENTS

From the outset, the project opens up a comprehensive landscape of FEC as a whole and then honing in on the specific intricacies of RS codes. Its related problems walk a systematic path from fundamental concepts to conclusive outcomes, rather than just presenting an unconvincing formula. Alongside this, existing projects in the same field have been explored and assessed, shaping the foundation from which the guiding concepts of this project emerged.

In its second achievement, the project has successfully constructed two significant models, the noisy channel and error probability analysis, offering a key that unlocks nearly every facet of any RS code. With their inherent flexibility, these models stand ready to be effortlessly applied to future works with little need for adjustment.

The project's third strength lies in its thoughtful progression, as it completely shifts theoretical concepts into the way hardware interprets and operates them,

thus paving the way by which RS codes begin to take form beyond the page. In its depth, the project also unveils the architecture of each constituent component, where the design aligns almost one-to-one with the underlying theory.

The fourth milestone, perhaps the most important, is the successful construction of two architectures, $RS(255, 239)$ encoder and decoder, which are designed to handle two crucial practical challenges, namely continuous data and synchronization between them. The designs have been fully verified using 48 test cases, with golden models for the main algorithms built in MATLAB, covering nearly every aspect and every possible scenario that might arise during operation. In their finest version, the designs allow operation at a maximum clock frequency of 166.6667 MHz, enabling data rates exceeding the 10 Gbps mark, far outperforming similar designs in the same field.

Completing the journey, the project's final achievement finds its footing on the Kria KV260 Vision AI starter kit FPGA platform, with a system that satisfies all technical requirements from the proposed process and allows the integration of RS codec as IPs, ready for broader application. The system was then successfully deployed and evaluated, delivering RS algorithm results in under 4.5 ms and completing the full processing of 900 kB of raw image data in under 0.5 second, placing it in a league of its own compared to the software solution.

Together, the five achievements represent the strongest evidence of success in achieving the set of six objectives, which were set out since the project was initiated and affirmed in section 1.2. Such results speak for themselves, signaling the project's successful realization.

7.2. LIMITATIONS AND CHALLENGES

Though the accomplishments are both meaningful and aligned with defined goals, the project, in the context of current technological developments, is still equally important for bringing its limitations to light.

Firstly, while the 10 Gbps data rate achieved by the project is a noteworthy accomplishment in the context of similar efforts, it appears modest when contrasted with the most advanced technologies that shape the present.

Secondly, the project is largely limited to the FPGA platform, while today, ASICs are generally superior in high-performance applications. At the same time, comparing the solution only to FPGA-based works is a major disadvantage, as current leading technologies are not developed on this platform.

System construction and integration were the final limitation of the project, as hardware limitations somewhat hindered efficiency. The evaluation data,

900 kB, although substantial, is still insufficient to explore cases with post-FEC BER less than 10^{-6}. Additionally, the current system, though functional, has not yet explored the full spectrum of design possibilities that could lead to a more optimal implementation.

7.3. INSPIRATION FOR FUTURE WORKS

These honest reflections on the project's limits do not diminish its value, but rather serve as lasting sparks of inspiration reminders that this project is just one step in the longer journey for FEC and RS codes to reach even greater heights. The project, while drawing to a close, points toward untraveled paths with rich potential, waiting to be explored and carried further.

In the current era, where ASICs have begun to touch Tbps architectures, the demand for rapid advancements in FPGA data rates stands as one of the urgent needs. This reality unveils a promising path for the evolution, one that revolves around the ever-important goal of increasing the data rate.

The error detection and correction capability of RS codes specifically, and FEC in general, represents another potential avenue for the project. As data rates rise, so too does the challenge of transmission errors, making the need for more powerful error correction codes, or additional coding techniques to unlock performance breakthroughs.

Even if RS codes are no longer the primary concern, the continued development of the data transmission system remains an open road. Optimizing AXI transmission time, mastering firmware to directly use the Ethernet channel, as well as enriching the website with new features, broader interoperability, and diverse data forms, all stand as vivid examples of this unfolding opportunity.

A

ELEMENT SET OF $GF(2^8)$

Power form	Base-n representation [LSB : MSB]		Power form	Base-n representation [LSB : MSB]	
	$n = 2$	$n = 16$		$n = 2$	$n = 16$
0	00000000	00	α^{127}	00110011	cc
1	10000000	10	α^{128}	10100001	58
α	01000000	20	α^{129}	11101000	71
α^2	00100000	40	α^{130}	01110100	$e2$
α^3	00010000	80	α^{131}	00111010	$c5$
α^4	00001000	01	α^{132}	00011101	$8b$
α^5	00000100	02	α^{133}	10110110	$d6$
α^6	00000010	04	α^{134}	01011011	ad
α^7	00000001	08	α^{135}	10010101	$9a$
α^8	10111000	$d1$	α^{136}	11110010	$f4$
α^9	01011100	$a3$	α^{137}	01111001	$e9$
α^{10}	00101110	47	α^{138}	10000100	12
α^{11}	00010111	$8e$	α^{139}	01000010	24
α^{12}	10110011	dc	α^{140}	00100001	48
α^{13}	11100001	78	α^{141}	10101000	51
α^{14}	11001000	31	α^{142}	01010100	$a2$

Continued on the next page ...

Continued from the previous page ...

Power form	Base-n representation [LSB : MSB]		Power form	Base-n representation [LSB : MSB]	
	$n = 2$	$n = 16$		$n = 2$	$n = 16$
α^{15}	01100100	62	α^{143}	00101010	45
α^{16}	00110010	$c4$	α^{144}	00010101	$8a$
α^{17}	00011001	89	α^{145}	10110010	$d4$
α^{18}	10110100	$d2$	α^{146}	01011001	$a9$
α^{19}	01011010	$a5$	α^{147}	10010100	92
α^{20}	00101101	$4b$	α^{148}	01001010	25
α^{21}	10101110	57	α^{149}	00100101	$4a$
α^{22}	01010111	ae	α^{150}	10101010	55
α^{23}	10010011	$9c$	α^{151}	01010101	aa
α^{24}	11110001	$f8$	α^{152}	10010010	94
α^{25}	11000000	30	α^{153}	01001001	29
α^{26}	01100000	60	α^{154}	10011100	93
α^{27}	00110000	$c0$	α^{155}	01001110	27
α^{28}	00011000	81	α^{156}	00100111	$4e$
α^{29}	00001100	03	α^{157}	10101011	$5d$
α^{30}	00000110	06	α^{158}	11101101	$7b$
α^{31}	00000011	$0c$	α^{159}	11001110	37
α^{32}	10111001	$d9$	α^{160}	01100111	$6e$
α^{33}	11100100	72	α^{161}	10001011	$1d$
α^{34}	01110010	$e4$	α^{162}	11111101	fb
α^{35}	00111001	$c9$	α^{163}	11000110	36
α^{36}	10100100	52	α^{164}	01100011	$6c$
α^{37}	01010010	$a4$	α^{165}	10001001	19
α^{38}	00101001	49	α^{166}	11111100	$f3$
α^{39}	10101100	53	α^{167}	01111110	$e7$
α^{40}	01010110	$a6$	α^{168}	00111111	cf
α^{41}	00101011	$4d$	α^{169}	10100111	$5e$
α^{42}	10101101	$5b$	α^{170}	11101011	$7d$
α^{43}	11101110	77	α^{171}	11001101	$3b$

Continued on the next page ...

Continued from the previous page ...

Power form	Base-n representation [LSB : MSB]		Power form	Base-n representation [LSB : MSB]	
	$n = 2$	$n = 16$		$n = 2$	$n = 16$
α^{44}	01110111	ee	α^{172}	11011110	$b7$
α^{45}	10000011	$1c$	α^{173}	01101111	$6f$
α^{46}	11111001	$f9$	α^{174}	10001111	$1f$
α^{47}	11000100	32	α^{175}	11111111	ff
α^{48}	01100010	64	α^{176}	11000111	$3e$
α^{49}	00110001	$c8$	α^{177}	11011011	bd
α^{50}	10100000	50	α^{178}	11010101	ba
α^{51}	01010000	$a0$	α^{179}	11010010	$b4$
α^{52}	00101000	41	α^{180}	01101001	69
α^{53}	00010100	82	α^{181}	10001100	13
α^{54}	00001010	05	α^{182}	01000110	26
α^{55}	00000101	$0a$	α^{183}	00100011	$4c$
α^{56}	10111010	$d5$	α^{184}	10101001	59
α^{57}	01011101	ab	α^{185}	11101100	73
α^{58}	10010110	96	α^{186}	01110110	$e6$
α^{59}	01001011	$2d$	α^{187}	00111011	cd
α^{60}	10011101	$9b$	α^{188}	10100101	$5a$
α^{61}	11110110	$f6$	α^{189}	11101010	75
α^{62}	01111011	ed	α^{190}	01110101	ea
α^{63}	10000101	$1a$	α^{191}	10000010	14
α^{64}	11111010	$f5$	α^{192}	01000001	28
α^{65}	01111101	eb	α^{193}	10011000	91
α^{66}	10000110	16	α^{194}	01001100	23
α^{67}	01000011	$2c$	α^{195}	00100110	46
α^{68}	10011001	99	α^{196}	00010011	$8c$
α^{69}	11110100	$f2$	α^{197}	10110001	$d8$
α^{70}	01111010	$e5$	α^{198}	11100000	70
α^{71}	00111101	cb	α^{199}	01110000	$e0$
α^{72}	10100110	56	α^{200}	00111000	$c1$

Continued on the next page ...

Continued from the previous page ...

Power form	Base-n representation [LSB : MSB]		Power form	Base-n representation [LSB : MSB]	
	$n = 2$	$n = 16$		$n = 2$	$n = 16$
α^{73}	01010011	ac	α^{201}	00011100	83
α^{74}	10010001	98	α^{202}	00001110	07
α^{75}	11110000	$f0$	α^{203}	00000111	$0e$
α^{76}	01111000	$e1$	α^{204}	10111011	dd
α^{77}	00111100	$c3$	α^{205}	11100101	$7a$
α^{78}	00011110	87	α^{206}	11001010	35
α^{79}	00001111	$0f$	α^{207}	01100101	$6a$
α^{80}	10111111	df	α^{208}	10001010	15
α^{81}	11100111	$7e$	α^{209}	01000101	$2a$
α^{82}	11001011	$3d$	α^{210}	10011010	95
α^{83}	11011101	bb	α^{211}	01001101	$2b$
α^{84}	11010110	$b6$	α^{212}	10011110	97
α^{85}	01101011	$6d$	α^{213}	01001111	$2f$
α^{86}	10001101	$1b$	α^{214}	10011111	$9f$
α^{87}	11111110	$f7$	α^{215}	11110111	fe
α^{88}	01111111	ef	α^{216}	11000011	$3c$
α^{89}	10000111	$1e$	α^{217}	11011001	$b9$
α^{90}	11111011	fd	α^{218}	11010100	$b2$
α^{91}	11000101	$3a$	α^{219}	01101010	65
α^{92}	11011010	$b5$	α^{220}	00110101	ca
α^{93}	01101101	$6b$	α^{221}	10100010	54
α^{94}	10001110	17	α^{222}	01010001	$a8$
α^{95}	01000111	$2e$	α^{223}	10010000	90
α^{96}	10011011	$9d$	α^{224}	01001000	21
α^{97}	11110101	fa	α^{225}	00100100	42
α^{98}	11000010	34	α^{226}	00010010	84
α^{99}	01100001	68	α^{227}	00001001	09
α^{100}	10001000	11	α^{228}	10111100	$d3$
α^{101}	01000100	22	α^{229}	01011110	$a7$

Continued on the next page ...

Continued from the previous page ...

Power form	Base-n representation [LSB : MSB]		Power form	Base-n representation [LSB : MSB]	
	$n = 2$	$n = 16$		$n = 2$	$n = 16$
α^{102}	00100010	44	α^{230}	00101111	$4f$
α^{103}	00010001	88	α^{231}	10101111	$5f$
α^{104}	10110000	$d0$	α^{232}	11101111	$7f$
α^{105}	01011000	$a1$	α^{233}	11001111	$3f$
α^{106}	00101100	43	α^{234}	11011111	bf
α^{107}	00010110	86	α^{235}	11010111	be
α^{108}	00001011	$0d$	α^{236}	11010011	bc
α^{109}	10111101	db	α^{237}	11010001	$b8$
α^{110}	11100110	76	α^{238}	11010000	$b0$
α^{111}	01110011	ec	α^{239}	01101000	61
α^{112}	10000001	18	α^{240}	00110100	$c2$
α^{113}	11111000	$f1$	α^{241}	00011010	85
α^{114}	01111100	$e3$	α^{242}	00001101	$0b$
α^{115}	00111110	$c7$	α^{243}	10111110	$d7$
α^{116}	00011111	$8f$	α^{244}	01011111	af
α^{117}	10110111	de	α^{245}	10010111	$9e$
α^{118}	11100011	$7c$	α^{246}	11110011	fc
α^{119}	11001001	39	α^{247}	11000001	38
α^{120}	11011100	$b3$	α^{248}	11011000	$b1$
α^{121}	01101110	67	α^{249}	01101100	63
α^{122}	00110111	ce	α^{250}	00110110	$c6$
α^{123}	10100011	$5c$	α^{251}	00011011	$8d$
α^{124}	11101001	79	α^{252}	10110101	da
α^{125}	11001100	33	α^{253}	11100010	74
α^{126}	01100110	66	α^{254}	01110001	$e8$

Table A.1: $GF\left(2^8\right)$ generated by $p(X) = 1 + X^2 + X^3 + X^4 + X^8$

B

RESOURCE UTILIZATION

B.1. SYNTHESIZED RESOURCE

Module name	Clock frequency	Data interface	LUT	MUX-F	FDCE FDPE
	MHz	bits			
Controller	100	16	11	0	9
Encoder's buffer			1	0	24
Parity calculator			236	0	131
Output selector			193	112	26
Controller and synchronizer			67	0	26
Decoder's buffer			0	0	2288
Syndrome calculator			336	0	131
BM algorithm solver			1075	0	267
Error locator and evaluator			721	1	158
Error corrector			68	0	55
Controller	100	32	11	0	9
Encoder's buffer			1	0	56
Parity calculator			453	0	132

Continued on the next page ...

Continued from the previous page ...

Module	Data interface	Clock frequency	LUT	MUX-F	FDCE FDPE
Output selector			341	0	44
Controller and synchronizer			73	0	28
Decoder's buffer			0	0	2528
Syndrome calculator			757	0	132
BM algorithm solver			1075	0	267
Error locator and evaluator			1358	0	174
Error corrector			187	0	105
Controller	100	64	12	0	9
Encoder's buffer			1	0	120
Parity calculator			952	32	133
Output selector			637	0	78
Controller and synchronizer			76	0	30
Decoder's buffer			0	0	3008
Syndrome calculator			1715	16	133
BM algorithm solver			1075	0	267
Error locator and evaluator			2587	0	206
Error corrector			413	0	203
Controller	166.6667	16	11	0	9
Encoder's buffer			1	0	24
Parity calculator			235	0	131
Output selector			193	112	26
Controller and synchronizer			67	0	26
Decoder's buffer			0	0	2288
Syndrome calculator			336	0	131
BM algorithm solver			1068	0	267
Error locator and evaluator			721	1	158
Error corrector			68	0	55
Controller	166.6667	32	11	0	9
Encoder's buffer			1	0	56

Continued on the next page ...

Continued from the previous page ...

Module	Data interface	Clock frequency	LUT	MUXF	FDCE FDPE
Parity calculator			453	0	132
Output selector			341	0	44
Controller and synchronizer			73	0	28
Decoder's buffer			0	0	2528
Syndrome calculator			757	0	132
BM algorithm solver			1068	0	267
Error locator and evaluator			1358	0	174
Error corrector			187	0	105
Controller			12	0	9
Encoder's buffer			1	0	120
Parity calculator			944	0	133
Output selector			637	0	78
Controller and synchronizer	166.6667	64	76	0	30
Decoder's buffer			0	0	3008
Syndrome calculator			1715	16	133
BM algorithm solver			1068	0	267
Error locator and evaluator			2543	0	206
Error corrector			413	0	203

Table B.1: Synthesized resource utilization by modules

Module name	Clock frequency	Data interface	LUT1	LUT2	LUT3	LUT4	LUT5	LUT6	MUX-F7	MUX-F8	FDCE FDPE
	MHz	bits									
Encoder	100	16	1	4	30	10	83	313	80	32	190
Decoder			0	109	408	423	432	895	1	0	2925
Encoder		32	1	7	55	64	140	539	0	0	241
Decoder			0	258	491	630	523	1548	0	0	3234
Encoder		64	1	8	132	61	333	1067	24	8	340
Decoder			0	510	773	720	713	3150	16	0	3847
Encoder	166.6667	16	1	4	13	24	51	347	80	32	190
Decoder			0	106	407	411	444	892	1	0	2925
Encoder		32	1	7	55	64	140	539	0	0	241
Decoder			0	255	475	618	535	1560	0	0	3234
Encoder		64	1	8	140	61	309	1075	0	0	340
Decoder			0	506	738	674	725	3172	16	0	3847

Table B.2: Synthesized resource utilization by types

B.2. IMPLEMENTED RESOURCE

Primitive	Description	Sender side	Receiver side
BUFGCE	General clock buffer with enable	0	1
BUFGPS	High-fanout clock buffer for PS	1	1
CARRY8	Fast carry logic with look ahead	37	45
FDCE	DFF with clock enable and clear	292	927
FDPE	DFF with clock enable and preset	1	1
FDRE	DFF with clock enable and reset	4771	4860
FDSE	DFF with clock enable and set	232	209
LUT1	1-bit look-up table	184	154
LUT2	2-bit look-up table	479	776
LUT3	3-bit look-up table	1321	1616
LUT4	4-bit look-up table	689	1244
LUT5	5-bit look-up table	856	1131
LUT6	6-bit look-up table	1361	2401
MUXF7	CLB multiplexer to connect LUT6s	3	19
MUXF8	CLB multiplexer to connect LUT7s	0	8
RAMD32	32×1 dual port synchronous RAM	672	672
RAMS32	32×1 static synchronous RAM	98	98
SRL16E	16-bit shift register look-up table	223	255
SRL32E	32-bit shift register look-up table	49	113
RAMB-36E2	36-kbit configurable synchronous block RAM	2	2

Table B.3: Implemented resource utilization by types

REFERENCES

[1] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[2] IEEE Computer Society, 802.3-2018 – IEEE Standard for Ethernet, Available at: https://ieeexplore.ieee.org/document/8457469, 2018.

[3] S. Lin and J. Li, *Fundamentals of Classical and Modern Error-Correcting Codes*. Cambridge University Press, 2021.

[4] J. G. Proakis and M. Salehi, *Digital Communications*, 5th. McGraw-Hill Education, 2007.

[5] E. N. Gilbert, "Capacity of a Burst-Noise Channel," *The Bell System Technical Journal*, vol. 39, no. 5, pp. 1253–1265, 1960.

[6] E. O. Elliott, "Estimates of Error Rates for Codes on Burst-Noise Channels," *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.

[7] J. G. David Forney, "Concatenated codes," M.I.T. Press, Cambridge, Tech. Rep. 440, 1965.

[8] Y. Djemamar, I. Saida, and A. Zeroual, "Performance Analysis of QPSK, 4QAM, 16QAM and 64QAM with Binary and Gray Constellation Codes over AWGN Channel," *International Conference on Advanced Information Technology, Services and Systems*, 2015.

[9] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[10] E. R. Berlekamp, *Algebraic Coding Theory*, revised. World Scientific, 1984.

[11] J. L. Massey, "Shift-Register Synthesis and BCH Decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[12] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A Method for Solving Key Equation for Decoding Goppa Codes," *Information and Control*, vol. 27, pp. 87–99, 1975.

[13] R. T. Chien, "Cyclic Decoding Procedures for Bose–Chaudhuri–Hocquenghem Codes," *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357–363, 1964.

[14] J. G. David Forney, "On Decoding BCH Codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 549–557, 1965.

[15] R. E. Blahut, *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.

[16] D. Das, *VLSI Design*, 2nd. Oxford University Press, 2015.

[17] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson, and R. W. Stewart, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, 2019.

[18] Arm Developer, AMBA AXI Protocol Specification, Available at: https://developer.arm.com/documentation/ihi0022, 2010.

[19] Xilinx, Inc., Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit, Available at: https://www.xilinx.com/products/boards-and-kits/zcu106.html.

[20] Xilinx, Inc., Kria KV260 Vision AI Starter Kit, Available at: https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit.html.

[21] S. M. Dilek, B. Ors, and M. Kartal, "Reed–Solomon Decoder Hardware Implementation for DVB-S Receiver," *Signal Processing and Communications Applications Conference*, 2013.

[22] C. Kim, S. Rhee, J. Kim, and Y. Jee, "Product Reed–Solomon Codes for Implementing NAND Flash Controller on FPGA Chip," *International Conference on Computer Engineering and Applications*, 2010.

[23] P. Dayal and R. K. Patial, "Implementation of Reed–Solomon Codec for IEEE 802.16 network using VHDL code," *International Conference on Reliability Optimization and Information Technology*, 2014.

[24] J. You and S. Wu, "Design and Realization of Reed–Solomon Codec Based on FPGA Technique," *International Conference on Mechatronic Science, Electric Engineering and Computer*, 2011.

[25] AMD Technical Information Portal, Reed–Solomon Encoder Product Guide, Available at: https://docs.amd.com/v/u/en-US/pg025_rs_encoder.

[26] AMD Technical Information Portal, Reed–Solomon Decoder Product Guide, Available at: https://docs.amd.com/v/u/en-US/pg107-rs-decoder.

[27] G. Perrone, J. Valls, V. Torres, and F. M. Garcia-Herrero, "High-throughput one-channel RS(255, 239) Decoder," *Euromicro Conference on Digital System Design*, 2018.

[28] S. M. Ross, *Introduction to Probability Models*, 10th. Academic Press, 2009.

[29] C. Y. Liu, "100+ Gb/s Ethernet Forward Error Correction (FEC) Analysis," *DesignCon*, 2019.

[30] R. Barrie, M. Yang, and A. C. Carusone, "Statistical BER Analysis of Concatenated FEC in Multi-Part Links," *DesignCon*, 2023.

[31] W. G. Horner, "A New Method of Solving Numerical Equations of All Orders, by Continuous Approximation," *Philosophical Transactions of the Royal Society of London*, vol. 109, pp. 308–335, 1819.

[32] AMD Technical Information Portal, UltraScale Architecture Libraries Guide, Available at: https://docs.amd.com/r/en-US/ug974-vivado-ultrascale-libraries.

[33] AMD Technical Information Portal, Kria K26 SOM Data Sheet, Available at: https://docs.amd.com/r/en-US/ds987-k26-som.

[34] P. Gianni, G. Di, F. Corteggiano, M. Del, and C. Argentina, "Implementacion en FPGA de un Codigo Reed–Solomon RS(255, 239)," *Escuela Argentina de Microelectronica, Tecnologia y Aplicaciones*, 2007.

[35] S. D. Mhaske, U. Ghodeswar, and G. G. Sarate, "VLSI Implementation of Low Complexity Reed–Solomon Decoder," *International Conference on Communication and Signal Processing*, 2014.

[36] B. Tiwari and R. Mehra, "Design and implementation of Reed–Solomon Decoder for 802.16 network using FPGA," *IEEE International Conference on Signal Processing, Computing and Control*, 2012.

[37] Viettel Digital, Home Wi-Fi Zte H196A, Available at: https://viettel-digital.com/product/zte-h196a.

[38] Rapoo, C260 Web Camera, Available at: https://www.rapoo.com/product/c260.

[39] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, 7th. Wiley, 2018.