


**FACULTY OF ENGINEERING**

**CAIRO UNIVERSITY**

Electronics and Electrical Communication Engineering

# Graduation Project

Digital Implementation of

Physical Coding Sublayer of 200/400 Gbps Ethernet

**Submitted By:**

| | |
|---|---|
| Ahmed Adel Younis | 9213073 |
| Ali Mokhtar El Dahshoury | 9210688 |
| Karim Ayman Mohamed | 9210836 |
| Magdy Ahmed Abbas | 9210899 |
| Mostafa Ibrahim Mohamed | 9211158 |
| Yahia Khaled Abdelfattah | 9211362 |

**Supervised By:** Dr. Rania Osama

**Sponsored By:** Siemens EDA

**June 2025**

# Acknowledgments

We would like to give our heartfelt thanks to everyone who helped, guided, and supported us throughout this year while working on our graduation project.

We are especially grateful to our amazing mentors at **Siemens EDA Company:**

- Eng. Mohamed Shaaban.
- Eng. Ragi Adel.
- Eng. Momen El Desoki.
- Eng. Eman Arafa.
- Eng. Ahmed Emad.
- Eng. Amr Maged.

Your support, technical advice, and kind guidance made a big difference in our learning. Every meeting, comment, and suggestion you gave helped us improve and grow. We truly appreciate the time and effort you gave us.

We would also like to thank our college supervisor, **Dr. Rania Osama**, for her continuous support, encouragement, and helpful feedback. She played an important role in guiding us and helping us stay on the right track.

Thank you all for believing in us and sharing your knowledge. You helped make this project a great learning experience, and we couldn't have done it without you.

With deep appreciation,
**Ethernet Graduation Project Team**

# Abbreviations

| Abbreviation | Full Name |
|---|---|
| **200GMII** | 200 Gb/s Media Independent Interface |
| **AI** | Artificial intelligence |
| **AM** | Alignment marker |
| **BMA** | Barlekamp and Massey Algorithm |
| **CS** | Chien Search |
| **CSE** | Common Sub Expression |
| **CW** | Codeword |
| **EC** | Error Correction |
| **EEE** | Energy-Efficient Ethernet |
| **EE** | Error Evaluator |
| **EOP** | End of Packet |
| **FEC** | Forward Error Correction |
| **FECL** | FEC Lane |
| **FPGA** | Field-Programmable Gate Array |
| **FIFO** | First-in First-out |
| **Gbps** | Giga bit per second |
| **GF** | Galois Field |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **IMA** | Iterative Matching Algorithm |
| **ISO** | International Organization for Standardization |
| **IT** | Information technology |
| **KES** | Key Equation Solver |
| **LFSR** | Linear Feedback Shift Register |
| **MAC** | Media Access Control |
| **MII** | Media Independent Interface |
| **MIMO** | Multiple input Multiple Output |
| **PCS** | Physical coding sublayer |
| **PCB** | Printed Circuit Board |
| **PE** | Processing Element |
| **PHY** | Physical Layer |
| **PMA** | Physical Medium Attachment |
| **PMD** | Physical Medium Dependent |
| **RM** | Rate matching |
| **RS** | Reed Solomon |
| **RX** | Receiver |
| **SC** | Syndrome Calculation |
| **SoC** | System on Chip |
| **SIP** | Symbol In Parallel |
| **Tbps** | Tera bit per second |
| **TX** | Transmitter |

# Table of Contents

# Table of Figures

# List of Tables

# 1. Abstract

The Physical Coding Sublayer (PCS) plays a critical role in the Ethernet physical layer by managing data encoding and error correction, which is especially vital in high-speed networks where retransmissions are prohibitively expensive. Our objective is to implement the PCS of 200/400 Gbps, achieving the required specifications stated by the IEEE 802.3 [1] standard while using minimum FPGA requirements. MATLAB was used in modeling and verifying the PCS implemented using SystemVerilog. A literature review of Reed-Solomon error-correcting codes was carried out to ensure efficient and state-of-the-art technology is used. The resulting implementation complies with the IEEE 802.3 requirements and serves as a functional prototype suitable for commercialization in real-world Ethernet applications.

# 2. Introduction

Due to increasing demand for high-speed IT systems for high data rates such as cloud computing, AI, and 5G technology, where the cloud needs to connect many users with data centers using broadband communications links, resulting in a significant increase in high data traffic. In addition, 5G will require very broadband links to transmit data packets from and to 5G technology users. From this point, the need for broadband Ethernet communication links becomes mandatory to keep up with this evolution in technologies.

To fulfill the demand of the technology, there is a need for developing and implementing the broadband standards of the Ethernet to enable new technologies. The IEEE 802.3 standard defines the PHY layer specifications for Ethernet. The PHY layer is the last layer relevant to the OSI layers:

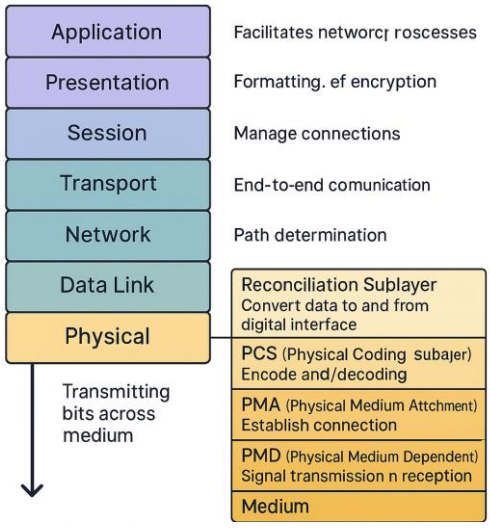



Figure 1: OSI layer with focus on PHY layer

The PHY layer is divided into 4 sublayers: reconciliation, PCS, PMA, and PMD sublayers. The focus will be on the development of the PCS of the latest highest released data rates standards, 400GBASE-R and 200GBASE-R that were released in 2023 specified in IEEE 802.3 [1]. This implementation of PCS can be used as a basis for the next releases. This higher-speed release will lead to smart and efficient utilization of infrastructure and scalability of the network.

In previous standards releases for PCS implementation, there is Reed-Solomon coding included within the layer, and it was an optional separate layer called the FEC layer. The new releases introduce RS coding within PCS so that it becomes mandatory for 200GBASE-R and 400GBASE-R standards.

# 3. Related Work

The PCS is usually implemented for industrial use therefore there's almost no open-source implementations available except for [2] where an ASIC implementation of the whole physical layer of 25Gbps Ethernet, a serial RS encoder and a parallel RS decoder were implemented to achieve the required throughput. The parallel RS decoder was implemented using conventional SC, Chien's search and Forney's algorithm and Euclidean algorithm in KES.

Unlike the PCS, the RS codec implementation has a lot of pervious literature which were intensively surveyed. In [3] a configurable RS (259,239) decoder of 8 or 16 channels with is implemented based on Euclidean algorithm, what's interesting about this implementation is the configurable syndrome cell and a Chien's search cells, they can work as 2 parallel or serial cells to achieve throughput or reduce power consumption depending on the current need of the system.

A comparison between serial and parallel architectures of RS decoder were done in [4], the tradeoff between latency, throughput, power and area were shown. Parallel architecture has more area and power consumption, lower latency and increased throughput. The previous mentioned literature helped shape the implemented PCS as they highlighted the need for a parallel RS codec.

The first mention of a parallel RS architecture codec was in [5], the latest work in [6] added the concept of symbols in parallel (SIP) which generalized the parallel architecture to how many symbols in parallel can the codec process in a clock cycle instead of the usual one symbol per clock cycle in serial architecture, this paper showed the gain in throughput against two different options; the first is increasing the SIP factor and the second increasing the number of channels (number of codecs). The paper concluded that the SIP maximum value should be 16 and any increase in throughput needed after that should be done by increasing the number of channels.

Our implementation of the FEC module will utilize parallel architecture to meet the throughput and latency requirements stated by the standard. The motive behind the architecture and the implementation method of each block in the FEC module will be justified. The module will be pipelined as in [6] with slight modification that removes the need for a controller greatly simplifying the design of the module.

# 4. PCS Literature Review

## 4.1. Physical Coding Sublayer (PCS) for 64B/66B (200/400 GBASE-R)

The Physical Coding Sublayer (PCS) is a critical component shared by both 200GBASE-R and 400GBASE-R physical layer implementations. These standards use 64B/66B encoding, which efficiently carries both data and control information. To reduce transmission overhead and make space for Forward Error Correction (FEC), the encoded data is further transcoded to 256B/257B format. After transcoding, FEC encoding is applied to enhance reliability over high-speed links.

As data moves through the PCS, it is split and distributed across multiple physical lanes. During this distribution, special alignment markers are inserted periodically. These markers allow the receiver to correctly align and reassemble the data from different lanes.

### Key Functions of the PCS

The PCS architecture includes both Transmit and Receive processing paths:

- It acts as a middle layer between the Reconciliation Sublayer and the PMA sublayer, abstracting the physical channel from higher protocol layers.
- The PCS supports two modes: normal mode for regular operation and test-pattern mode for validation purposes.
- It uses a 32-octet wide synchronous data path, with transmission control provided via TXC and RXC signals.
- 200GBASE-R uses 8 PCS lanes, while 400GBASE-R uses 16 PCS lanes. Each lane is an encoded bit stream. All PCS lanes share the same clock but may experience different phase and timing shifts (skew).

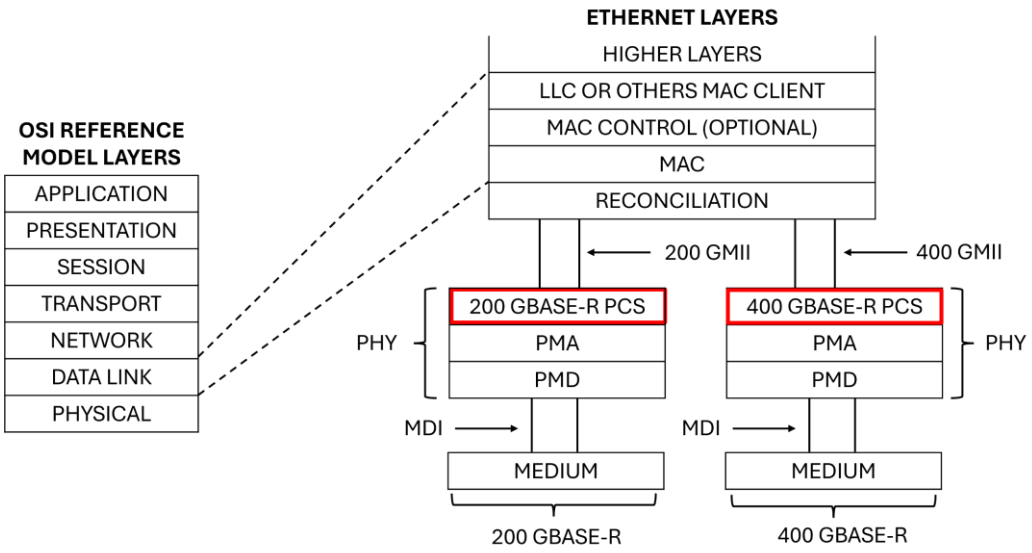



Figure 2: PCS Sublayer in 200/400 GBASE-R

**PCS Transmit Operation**

1. The PCS receives a 256-bit data block from the 200GMII/400GMII interface via TXD [255:0] and TXC [31:0].
2. This block is divided into four 64-bit segments, each encoded into 66-bit blocks using 64B/66B encoding.
3. Each 66-bit block starts with a sync header 01 for data blocks and 10 for control blocks followed by the payload.
4. Groups of four 66B blocks are transcoded into 257-bit blocks to reduce line coding overhead.
5. The blocks are then scrambled using a self-synchronizing scrambler to balance signal transitions and prevent long sequences of 0s or 1s.
6. Alignment markers (4 for 200GBASE-R or 8 for 400GBASE-R) are inserted periodically to support lane alignment at the receiver.
7. The stream is then distributed round-robin into two groups of 514 symbols.
8. Each group is passed through a Reed-Solomon RS (544,514) encoder, adding 30 parity symbols using parallel LFSR-based logic.
9. The two FEC codewords are interleaved and distributed across the PCS lanes (8 lanes for 200G, 16 lanes for 400G).
10. The transmit units are handed off to the PMA sublayer using PMA request primitive.

**PCS Receive Operation**

1. The PCS constantly monitors the signal validity through signal ok.
2. When the signal is good, it locks onto the alignment markers using both common markers (CM) and unique marker (UM) portions.
3. PCS lanes are reordered and deskewed since any TX lane may arrive at any RX lane due to timing shifts.
4. Once aligned, the PCS sets an align status flag and continues processing.
5. The data stream is de-interleaved, and the RS FEC decoder detects and corrects errors.
6. Corrected data is re-interleaved on a 10-bit basis to reconstruct the original stream.
7. The PCS removes alignment markers, descrambles the data, and transcodes it back to 256B/257B format.
8. It then decodes the 66B blocks to 64B, merges them into a 256-bit block, and delivers it to the next sublayer using RXD [255:0] and RXC [31:0] under RX_CLK.
9. The PCS also evaluates end-of-packet and signal ok signals.

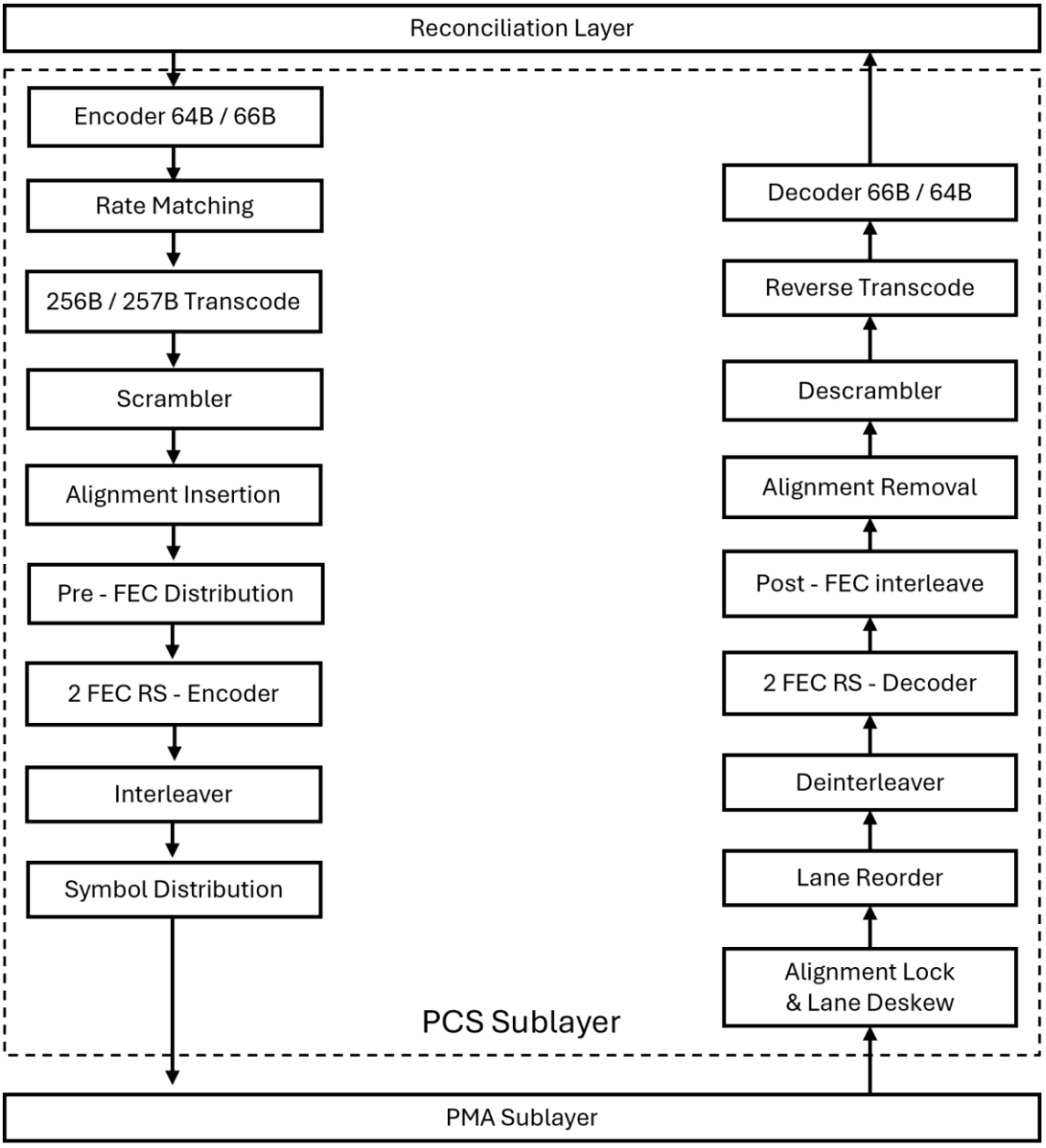



Figure 3: PCS Sublayer Tx & Rx Blocks

## 4.2. Galois Fields

### Overview

A finite field is a set of elements in which we can perform arithmetic operations as addition, subtraction, multiplication or division and the result of this arithmetic operation is an element of the set, addition and multiplication must be commutative, associative and satisfy the distribution laws. Galois fields are the backbone of many errors correcting codes and cryptography algorithms.

Galois fields are represented as GF ($p^m$), where $p^m$ is the number of elements in the field also known as the order of the field [7]. The field's order must be a prime number or the power of a prime number so that the set of elements satisfy the field's conditions mentioned before.

Galois fields have another interesting property, let there be a non-zero element α in a field GF(q), knowing that all non-zero elements in the field are close under multiplication then the powers of α must be non-zero elements and they must be finite as this is a finite field. This indicates that there will be a repetition in the sequence of increasing power of α.

The IEEE standard specified that the FEC module used in the 200/400 GBASE-R is Reed-Solomon (n, k, 2t) codes where n, k and 2t are shown in equation (eqn. (1)). These codes are of GF ($2^{10}$) meaning that they are an extension to the binary Galois field where each symbol is now represented in 10 bits instead of 1 in GF (2) and have error detection capability of 2t and correction capability of t.

$$n = 544,\ k = 514 \text{ and } 2t = n - k = 30 \rightarrow t = 15 \qquad \text{eqn. (1)}$$

Reed Solomon codes are very immune to burst errors due to the fact that they use extended fields GF ( $2^m$) that are extended from the binary Galois field GF (2). This means that a symbol consists of m bits and when a codeword is exposed to a burst error, bits that are close to each other in time will be affected but each m bits correspond to one symbol which costs one correction from the t capability of the code meaning that RS codes are very efficient with burst errors as they correct a group of bits instead of just one bit as other codes.

Polynomials are used to represent symbols, messages and codewords in GF (p) as polynomials form a ring enabling the representation of algebraic operations, has efficient representation and enable error detection and correction mechanisms through their roots. Each property will be discussed in detail next time.

Example of a symbol polynomial in GF ($2^{10}$) is shown in equation (eqn. (2)).

$$a_{m-1}x^m + \cdots + a_1x + a_0\ ,\ \text{where } a_i \text{ is a binary value } \{0,1\} \qquad \text{eqn. (2)}$$

A codeword is composed of n of these symbols while a message consists of k symbols, the process of encoding a message to transform it into a codeword is done by shifting the k message symbols to the left by 2t then appending 2t parity symbols to the end of the message symbols to form the n symbol codewords.

The primitive polynomial is an irreducible polynomial and it defines polynomial multiplication operation as polynomial multiplication modulo p(x), the primitive polynomial p(x) is defined as $x^{10} + x^3 + 1$. α is the called the primitive element or generator of multiplicative group if it's a root of p(x) as it can generate all the non-zero elements of the field.

**Galois Field Arithmetic**

Addition and subtraction in Galois fields are defined as addition module 2 which is equivalent to the XOR operation (carry less addition). Addition operation is straight in Galois field is a straightforward operation as XORing two polynomials of degree m-1 are closed under polynomial addition and the result has the same degree.

Multiplication is more complicated as multiplying two polynomials of degree m-1 might yield a result of degree higher than m-1 which is not an element of the field. Polynomial multiplication in Galois fields is defined as polynomial multiplication modulo p(x), an example is shown in figure (4) assume two polynomials a(x) and B(x) are multiplied in GF ($2^5$). The first step is polynomial multiplication in normal arithmetic but using carry less addition (XOR) resulting in the polynomial $m'(x)$ then the next step is to reduce this polynomial using the primitive polynomial p(x) so that the final result m(x) has a degree of $4\ (m-1)$.

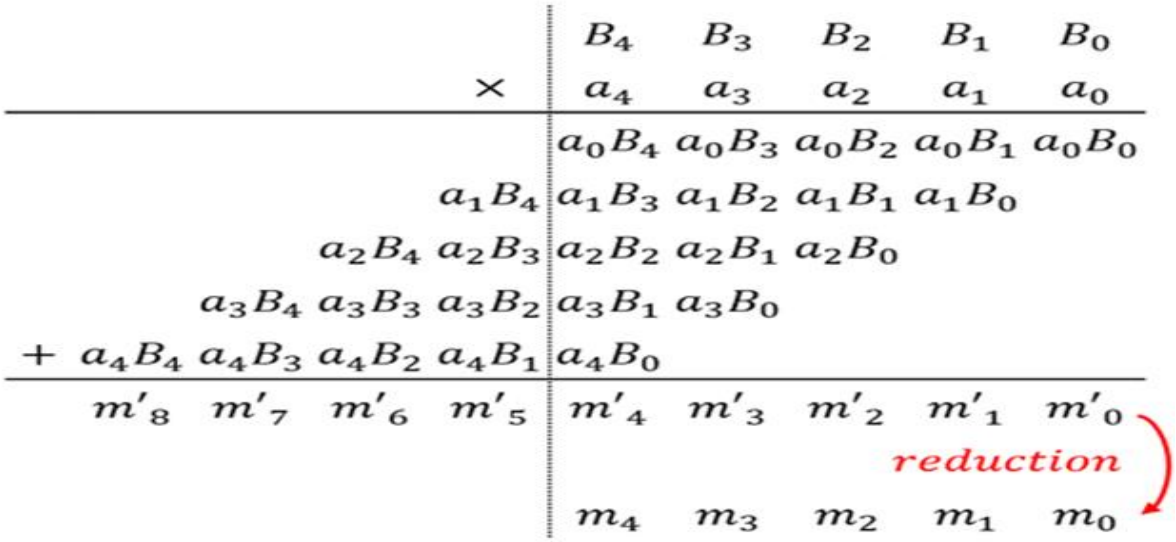



Figure 4: Example of polynomial multiplication in GF ($2^5$)

**Galois Field Multipliers**

There are two types of Galois field multipliers; constant multipliers have a constant value multiplicand and full multipliers with both multipliers have variable values. Each type has different optimization strategies even though full multipliers can be used as constant multipliers it's not recommended as full multipliers use more gates, power, and area and have longer delays. Full multipliers are the subject of focus as constant multipliers were replaced as shown later.

The design of full multipliers depends on the choice of the basis used to represent the polynomials [8], there are a few used basis options depending on the application as the canonical basis, dual basis, normal and standard bases. The result of any operation must be represented using the standard basis so it can be used in other blocks in the decoder flow as shown later, this means that transformation circuits will be required to change from and to standard basis, but they are complicated and affect performance greatly, so only standard basis are considered.

Two architectures are investigated for full GF multipliers; The first architecture is based on the equation (eqn. (3)) which shows that the result (W) of multiplication of two finite field element A and B can be computed in two stages.

$$W = A \cdot B = \sum_{k=m}^{2m-2} d_k p^k + \sum_{k=0}^{m-1} d_k p^k, d_k = \sum_{i=0}^{k} a_i b_i \qquad \text{eqn. (3)}$$

The first stage is the multiplication with carry less addition of the inputs then the resultant is passed through the second stage which is modular reduction using the primitive polynomial p(x) This architecture has the ability to be pipelined which allows the operation of RS decoder at high speeds.

The second architecture uses linear feedback shift register to implement the GF multiplier. This method has a feedback path which makes it hard to pipeline due to the data dependency unlike the first architecture. Other architectures such as serial multipliers and look up exist but serial architectures have very high latency and the performance of look up table deteriorates as the number of elements in the field increase.

**Galois Field Exponential**

GF Exponential operation is one of fundamental operations in cryptography and error correction codes. Conventional implementation of exponential was done using look-up table, but it was used with $m \leq 4$, $GF(2^m)$ so that when operating with large number of bits there is a need for efficient implementation. Most suitable method is square and multiply method for exponential known as binary method [9] which is efficient implementation for hardware and large number of bits.

**Galois Field Inversion**

Inversion in a Galois Field is critical for computing error values. The standard method for computing the inverse of element A in GF $(2^m)$ is based on Fermat's theorem [10]:

$$A^{-1} = A^{2^m - 2}$$

## 4.3. Encoder 64B/66B

The 64B/66B encoder is a mandatory component of the PCS layer in IEEE 802.3-compliant systems. The PCS uses a transmission code to improve transmission characteristics and to support transmission of data and control characters. The encoding guarantees sufficient transition in transmitted bits to ensure clock recovery at the receivers. In addition, it ensures detection of single-bit or multiple-bit errors that occur during transmission of the RECONCILIATION layer. The encoding process adds a 2-bit synchronization header to ensure a transition in transmitted bits to enable the receiver to achieve block alignment on incoming bits from PHY. The 64B/66B encoding scheme is defined in IEEE 802.3. [1].

**Encoding process**

The encoder receives 64-bit data (TXD) from the RECONCILIATION layer, where it is divided into 8 octets, each of which is 8 bits, and 8-bit control bits (TXC) via the media-independent interface (MII). The 64-bit data is encoded into a 66-bit block by adding a synchronization header and formatting the data efficiently for transmission. The data from the RECONCILIATION layer can take the following format:

Table 1: Block format in MII

| **Data block format** | | | | | | | |
|---|---|---|---|---|---|---|---|
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| **Control block format** | | | | | | | |
| $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $S_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| $O_0$ | $D_1$ | $D_2$ | $D_3$ | $Z_4$ | $Z_5$ | $Z_6$ | $Z_7$ |
| $T_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0$ | $T_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0$ | $D_1$ | $T_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0$ | $D_1$ | $D_2$ | $T_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $T_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $T_5$ | $C_6$ | $C_7$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $T_6$ | $C_7$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $T_7$ |

**Control codes**

As per IEEE 802.3 [1], Idle control character /I/ may be transmitted by MII; groups of 8 characters may be added or removed by PCS for rate-matching purposes. Start control character /S/; it indicates the beginning of receiving data characters and must be followed by data characters. If not, the encoder will detect an error in block format and will generate an error block. Order set character /Q/ to extend the ability to send control and status information over the link, such as remote fault and local fault status. The terminate control character /T/ indicates the end of a packet; it must be followed by an idle character within the same 64-bit block; otherwise, the encoder detects an error.

Table 2: Control codes

| Control character | Control character representation in MII (8-bit) | Control codes in PCS (7-bit) |
|---|---|---|
| /I/ | 0x07 | 0x00 |
| /LI/ | 0x06 | 0x06 |
| /E/ | 0xFE | 0x1E |
| /S/ | 0xFB | Implicitly in block type field |
| /T/ | 0xFD | Implicitly in block type field |
| /Q/ | 0x9c | Implicitly in block type field plus O0 |

**Mapping process**

Each incoming 64-bit block and its control bits (8 bits represented in 64 bits in the PCS layer) in addition to the synchronization header, where the sync header is for the data block, where it is for the control blocks. The mapping process includes the addition of a block type field for each corresponding type of control bit block, where the block type field is 8-bit, and this mapping is specified in IEEE 802.3-2018 [1] as follows:

| Input Data | Sync | Block Payload | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit Position: / Data Block Format: | 0 1 | 2 | | | | | | | 65 |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,D_7$ | 01 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| **Control Block Formats:** | | **Block Type Field** | | | | | | | |
| $C_0\,C_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x1E | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $S_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,D_7$ | 10 | 0x78 | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| $O_0\,D_1\,D_2\,D_3\,Z_4\,Z_5\,Z_6\,Z_7$ | 10 | 0x4B | $D_1$ | $D_2$ | $D_3$ | $O_0$ | 0x000_0000 | | |
| $T_0\,C_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x87 | | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,T_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x99 | $D_0$ | | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,T_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0xAA | $D_0$ | $D_1$ | | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,T_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0xB4 | $D_0$ | $D_1$ | $D_2$ | | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,T_4\,C_5\,C_6\,C_7$ | 10 | 0xCC | $D_0$ | $D_1$ | $D_2$ | $D_3$ | | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,T_5\,C_6\,C_7$ | 10 | 0xD2 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,T_6\,C_7$ | 10 | 0xE1 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,T_7$ | 10 | 0xFF | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | |

Table 3: 64B/66B block formats

**Handling invalid cases**

In this section, invalid cases for incoming blocks will be viewed such that if any invalid case appears, the encoder generates an error block that consists of eight error control characters corresponding to each valid case as specified in IEEE 802.3-2018 [1]:

1. Invalid synchronization header "11", or "00".
2. Invalid values of control character are not compatible with the standard.
3. Invalid block format is not included in 64B/66B block format.
4. Invalid sequence of incoming block that is not comply with the state machine of standard.

## 4.4. Rate matching

Rate matching is necessary in the PCS sublayer of Ethernet IEEE 802.3 [1]. It compensates for the rate reduction caused by the alignment marker (AM) insertion process, which is one of the fundamental mechanisms in the PCS to maintain a consistent data rate between the PCS and the MII.

**Functional description**

Rate matching is used to counteract the rate reduction caused by alignment marker (AM) insertion, which forces AMs into the data stream to support PCS lane alignment and efficient reception in RX mode. To maintain a consistent data rate between the PCS and the MII, it is necessary to remove blocks from the data stream incoming from the MII.

Since rate matching occurs after the 64B/66B encoding stage, it removes 66-bit encoded IDLE blocks. Specifically, it is required to remove 1,028 bits for 200 Gbps or 2,048 bits for 400 Gbps, which corresponds to the size of the inserted AMs. These bits can be represented by 16 or 32 IDLE blocks, respectively, before the transcoding operation.

The main goal of rate matching is to remove IDLE blocks equivalent to the size of the AMs, thereby maintaining a consistent data rate over each AM period. Its functionality can be summarized in the following flow chart:

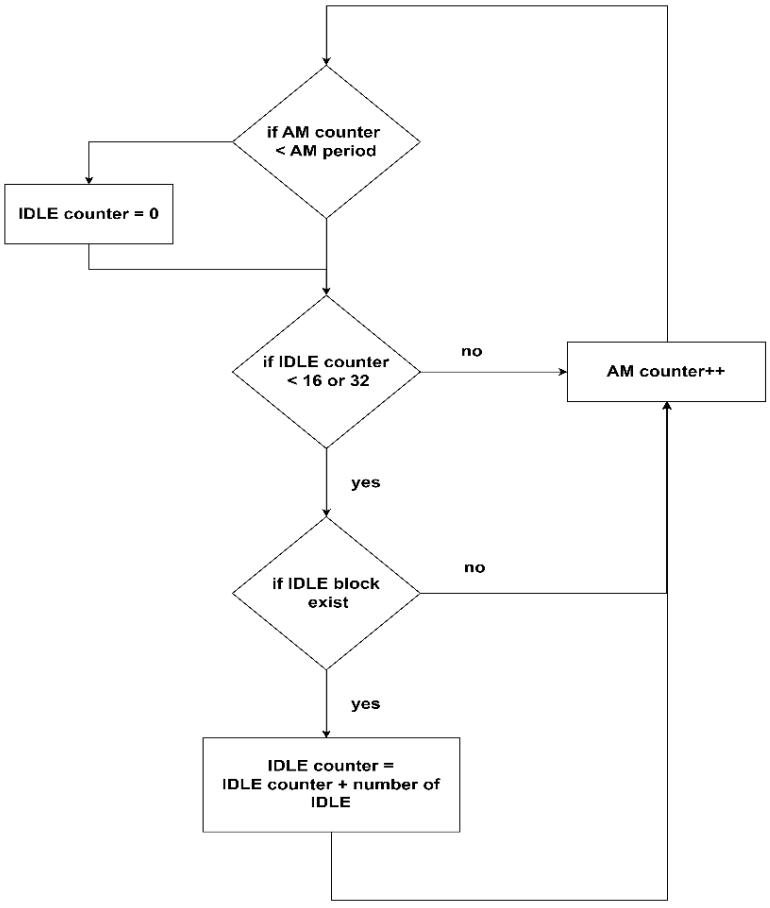



Figure 5: RM functionality flow chart

## 4.5. Tx Transcoder 256B/257B

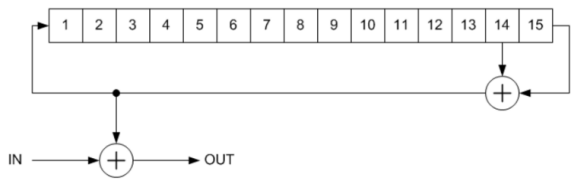
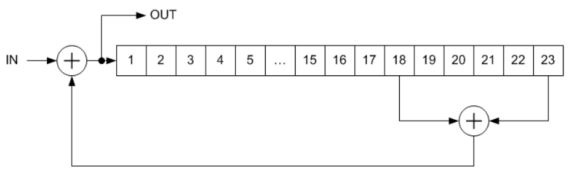
The transcoder constructs a 257-bit block, tx_xcoded, from a group of four 66-bit blocks, tx_coded_j where j = 0 to 3. For each group of four 66-bit blocks, j = 3 corresponds to the most recently received block. Bit 66 in each 66-bit block is the first bit received and corresponds to the first bit of the synchronization header.

## 4.6. Scrambler

Data scrambling plays a vital role in high-speed digital communication systems by maintaining spectral balance, preventing long runs of identical bits, and minimizing electromagnetic interference (EMI). There are two primary scrambling techniques: **Frame Synchronous Scrambling (FSS)** and **Self Synchronous Scrambling (SSS)** [11].

FSS uses a predefined seed (initial state) at the start of each frame to synchronize the transmitter and receiver. This approach ensures that errors in one frame do not affect others. However, it requires frame-by-frame initialization and coordination, making it slightly more complex.

SSS, in contrast, does not require external synchronization or an initial seed. The scrambler state updates dynamically based on the transmitted data, and the receiver uses the incoming scrambled data to perform descrambling. Since it operates continuously without resets, SSS is simpler and more efficient for many applications, therefore SSS is the preferred method in this context.

| Frame synchronous scrambling (FSS) | Self-synchronous scrambling (SSS) |
|---|---|
|  |  |





## 4.7. Alignment Marker Insertion

With the demanding high data rates for transmission & the challenges related to reliably handling these rates on a single through single physical link, the common approach is to split the required data rate into multiple physical links – so called as “PCS Lanes”- at a lower line rate.

The alignment insertion function comes into action to identify, manage these lanes as well as putting context to the synchronization between the Tx & Rx sides by specifying the block boundary pattern between a specific number of codewords.

**Concept of Lanes & Unit Attachment Interfaces:**

In high-speed Ethernet architecture as in our case, achieving reliable transmission over a single electrical or optical link at the full line rate results in practical challenges related to the signal integrity, power & cost constraints. To address this, the IEEE 802.3bs standard defines a multi-lane approach called "multi-lane distribution System (MLD)", where the data stream is split across multiple virtual lanes in terms of the primitives of the attachment unit interface (e.g.: 200/400 GUAI). These virtual lanes allow parallel transmission at a lower per-lane rate, easing implementation while maintaining the total throughput.

**Alignment Markers (AMs) & their properties:**

In order to identify the virtual lanes uniquely through pattern detection bt the **AM lock** function at the Rx, special coded 120-bit blocks are inserted periodically each alignment period to help in [12]:

- Help the receiver identify the lane boundaries accurately.
- Realign the PMA Lanes after transmission skew.
- Ensure lane data is reassembled in the correct order.

AMs are designed with some properties to achieve some physical & logical goals such as:

**Periodic Insertion:**
The AM period is set according to the number of CWs per block that defines the boundary.

200/400G： 81920/163840 x 257 − bit blocks

200G：2048 CWs ，400G：4096 CWs

**Unique Insertion:**
The AMs are not scrambled nor are they part of the encoding scheme specified by the 64/66b encoder to provide unique detection at the Rx. This can be observed from the location of the Insertion function to be after the data scrambling.

**DC Balance / High Transition Density:**

The AMs should be DC balanced (equal number of 1's & 0's) & exhibit high number of transitions to avoid Base-line wander that leads to false detection for long interconnects.

**AM Structure**

The insertion function is composed of 3 operations which are:

1) Map the markers to PCS lanes.
2) Generate the serial marker payload.
3) Handle the rate matching compensation.

**Mapping Function**

The AMs are structured into positions that specify the locking criteria at the Rx into:

**Common Portion (CM):** Common pattern for all lanes for simple pattern matching at locking.

**Unique Portion (UM):** Unique part for each lane.

**Unique Pad (UP):** Unique pad for cancelling the frame overlap in locking by completing the 120-bit AM block per lane.

Table 4: 200 GBASE-R AM Encodings

| PCS Lane | 200 GBASE-R Alignment Marker Encodings | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CM0 | CM1 | CM2 | UP0 | CM3 | CM4 | CM5 | UP1 | UM0 | UM1 | UM2 | UP2 | UM3 | UM4 | UM5 |
| 0 | 0x9A | 0x4A | 0x26 | 0x05 | 0x65 | 0xB5 | 0xD9 | 0xD6 | 0xB3 | 0xC0 | 0x8C | 0x29 | 0x4C | 0x3F | 0x73 |
| 1 | 0x9A | 0x4A | 0x26 | 0x04 | 0x65 | 0xB5 | 0xD9 | 0x67 | 0x5A | 0xDE | 0x7E | 0x98 | 0xA5 | 0x21 | 0x81 |
| 2 | 0x9A | 0x4A | 0x26 | 0x46 | 0x65 | 0xB5 | 0xD9 | 0xFE | 0x3E | 0xF3 | 0x56 | 0x01 | 0xC1 | 0x0C | 0xA9 |
| 3 | 0x9A | 0x4A | 0x26 | 0x5A | 0x65 | 0xB5 | 0xD9 | 0x84 | 0x86 | 0x80 | 0xD0 | 0x7B | 0x79 | 0x7F | 0x2F |
| 4 | 0x9A | 0x4A | 0x26 | 0xE1 | 0x65 | 0xB5 | 0xD9 | 0x19 | 0x2A | 0x51 | 0xF2 | 0xE6 | 0xD5 | 0xAE | 0x0D |
| 5 | 0x9A | 0x4A | 0x26 | 0xF2 | 0x65 | 0xB5 | 0xD9 | 0x4E | 0x12 | 0x4F | 0xD1 | 0xB1 | 0xED | 0xB0 | 0x2E |
| 6 | 0x9A | 0x4A | 0x26 | 0x3D | 0x65 | 0xB5 | 0xD9 | 0xEE | 0x42 | 0x9C | 0xA1 | 0x11 | 0xBD | 0x63 | 0x5E |
| 7 | 0x9A | 0x4A | 0x26 | 0x22 | 0x65 | 0xB5 | 0xD9 | 0x32 | 0xD6 | 0x76 | 0x5B | 0xCD | 0x29 | 0x89 | 0xA4 |

Table 5: 400 GBASE-R AM Encodings

| PCS Lane | 400 GBASE-R Alignment Marker Encodings | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CM0 | CM1 | CM2 | UP0 | CM3 | CM4 | CM5 | UP1 | UM0 | UM1 | UM2 | UP2 | UM3 | UM4 | UM5 |
| 0 | 0x9A | 0x4A | 0x26 | 0xB6 | 0x65 | 0xB5 | 0xD9 | 0xD9 | 0x01 | 0x71 | 0xF3 | 0x26 | 0xFE | 0x8E | 0x0C |
| 1 | 0x9A | 0x4A | 0x26 | 0x04 | 0x65 | 0xB5 | 0xD9 | 0x67 | 0x5A | 0xDE | 0x7E | 0x98 | 0xA5 | 0x21 | 0x81 |
| 2 | 0x9A | 0x4A | 0x26 | 0x46 | 0x65 | 0xB5 | 0xD9 | 0xFE | 0x3E | 0xF3 | 0x56 | 0x01 | 0xC1 | 0x0C | 0xA9 |
| 3 | 0x9A | 0x4A | 0x26 | 0x5A | 0x65 | 0xB5 | 0xD9 | 0x84 | 0x86 | 0x80 | 0xD0 | 0x7B | 0x79 | 0x7F | 0x2F |
| 4 | 0x9A | 0x4A | 0x26 | 0xE1 | 0x65 | 0xB5 | 0xD9 | 0x19 | 0x2A | 0x51 | 0xF2 | 0xE6 | 0xD5 | 0xAE | 0x0D |
| 5 | 0x9A | 0x4A | 0x26 | 0xF2 | 0x65 | 0xB5 | 0xD9 | 0x4E | 0x12 | 0x4F | 0xD1 | 0xB1 | 0xED | 0xB0 | 0x2E |
| 6 | 0x9A | 0x4A | 0x26 | 0x3D | 0x65 | 0xB5 | 0xD9 | 0xEE | 0x42 | 0x9C | 0xA1 | 0x11 | 0xBD | 0x63 | 0x5E |
| 7 | 0x9A | 0x4A | 0x26 | 0x22 | 0x65 | 0xB5 | 0xD9 | 0x32 | 0xD6 | 0x76 | 0x5B | 0xCD | 0x29 | 0x89 | 0xA4 |
| 8 | 0x9A | 0x4A | 0x26 | 0x60 | 0x65 | 0xB5 | 0xD9 | 0x9F | 0xE1 | 0x73 | 0x75 | 0x60 | 0x1E | 0x8C | 0x8A |
| 9 | 0x9A | 0x4A | 0x26 | 0x6B | 0x65 | 0xB5 | 0xD9 | 0xA2 | 0x71 | 0xC4 | 0x3C | 0x5D | 0x8E | 0x3B | 0xC3 |
| 10 | 0x9A | 0x4A | 0x26 | 0xFA | 0x65 | 0xB5 | 0xD9 | 0x04 | 0x95 | 0xEB | 0xD8 | 0xFB | 0x6A | 0x14 | 0x27 |
| 11 | 0x9A | 0x4A | 0x26 | 0x6C | 0x65 | 0xB5 | 0xD9 | 0x71 | 0x22 | 0x66 | 0x38 | 0x8E | 0xDD | 0x99 | 0xC7 |
| 12 | 0x9A | 0x4A | 0x26 | 0x18 | 0x65 | 0xB5 | 0xD9 | 0x5B | 0xA2 | 0xF6 | 0x95 | 0xA4 | 0x5D | 0x09 | 0x6A |
| 13 | 0x9A | 0x4A | 0x26 | 0x14 | 0x65 | 0xB5 | 0xD9 | 0xCC | 0x31 | 0x97 | 0xC3 | 0x33 | 0xCE | 0x68 | 0x3C |
| 14 | 0x9A | 0x4A | 0x26 | 0xD0 | 0x65 | 0xB5 | 0xD9 | 0xB1 | 0xCA | 0xFB | 0xA6 | 0x4E | 0x35 | 0x04 | 0x59 |
| 15 | 0x9A | 0x4A | 0x26 | 0xB4 | 0x65 | 0xB5 | 0xD9 | 0x56 | 0xA6 | 0xBA | 0x79 | 0xA9 | 0x59 | 0x45 | 0x86 |

## Create the AM payload

The AMs mapped for each lane should be interleaved to create a serial stream that shall be round robin distributed across the PCS lanes to provide the locking pattern required at the receiver as specified in **Table (4,5).**

Then, the payload is completed with additional padding to complete 1028/2056 bits resulting from pseudo-random base sequence generated continuously according to the generator polynomial $G(x) = 1 + x^5 + x^9$ that generated 65/133 per cycle to allow instant padding creation plus using 3-bit for signaling the local/remote faults between the PCS Tx/Rx instances known as “status field”.

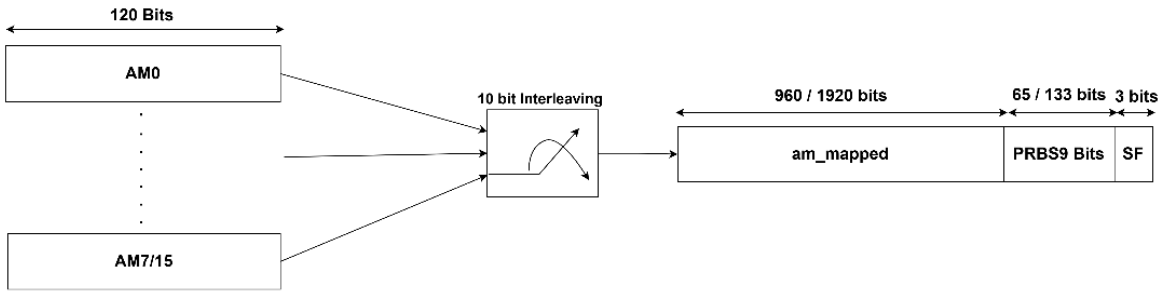



Figure 6: AM Payload Creation

**Handle the rate matching compensation**

The Insertion function should be successful only if it doesn't interfere with the user data delivery at the MAC layer which constrains the function to make the MAC experiences rate-consistent delivery.

This can be handled by aligning with the rate compensation function to utilize the stream gaps it supports by the deletion of the IDLE blocks (IFGs) inserted periodically by the RS layer.

## 4.8. Pre-FEC Distribution

The increasing demand for higher data rates introduces significant challenges related to signal integrity. To address this, 200/400G Ethernet standard implies Reed-Solomon Forward Error Correction (RS-FEC), and the Pre-FEC Distribution block acts as a preparatory stage that arranges the data into a structure efficient for FEC processing & inter-lane distribution.

This block is essential to comprehend how the PCS pipeline is structured to ensure reliability & robustness against different channel conditions.

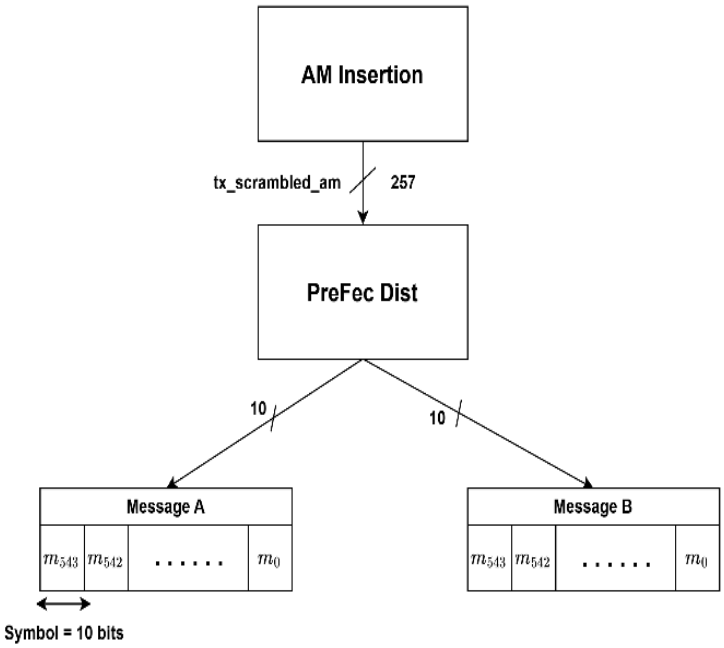



Figure 7: Pre-FEC Original Mechanism

The pre-FEC distribution function operates on a block of 10280 bit derived from [tx_scrambled_am] by splitting it into 2 messages via 10-bit round robin distribution.

This can help in mitigating the burst error that corrupts consecutive bits in the scrambled data to impact only the 2 codewords non-contiguously reducing the probability of uncorrectable errors.

## 4.9. RS FEC Encoder

In high-speed Ethernet communication systems such as the 200/400G, reliable data transmission over potentially noisy optical or electrical channels is mandatory. The PCS of the IEEE802.3 standard emphasized the usage of the FEC to ensure robust data integrity. At the core of the FEC mechanisms lie the RS encoder, a class of block-based error correcting codes capable of detecting, correcting multiple symbols per codeword & handling the burst error as well. Its adoption significantly enhances BER performance without the need for costly re-transmissions, which meets the urging requirements of low latency & throughput of modern data centers & backbone networks.

**Role & Position of the FEC Encoding:**

The IEEE802.3 standard defines the RS Encoding scheme as RS (544,514), where each 544-symbol codeword contains 514 data symbols & 30 parity symbols with each symbol packing a 10-bit frame. This scheme offers corrections of up to 15 symbols per codeword & detection of up to 30 symbols as well.

The RS encoder operates after the data has been distributed by the pre-FEC distribution & interleaving across multiple virtual lanes, ensuring that error bursts are spread across the codewords, hence improving correction efficiency.

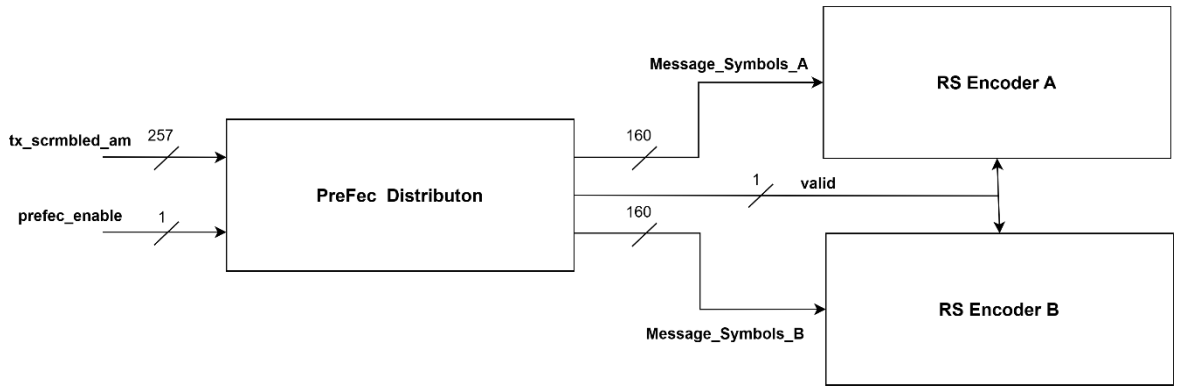



Figure 8: Location of the FEC RS-Encoder

**RS Encoder Serial Architecture:**

The standard specifies the RS (n, k) systematic code which applies over the Galois field GF ($2^m$) where each symbol consists of m bits making it correct up to $t = \frac{n-k}{2}$ and detect up to 2t where n: CW symbols, k = Msg symbols.

The RS (544,514) is based on the generator polynomial G(x) as shown in based on the primitive root ($\alpha$) with the primitive polynomial $P(x) = x^{10} + x^3 + 1$ which specifies the way of generating the non-zero elements of the Galois field.

$$G(x) = \prod_{i=0}^{i=2t-1}(x - \alpha^i) = g_{2t}x^{2t} + g_{2t-1}x^{2t-1} + \cdots + g_1x + g_0 \quad (4)$$

Message symbols are represented by its polynomial where its coefficients show a GF ($2^{10}$) element.

$$m(x) = m_{k-1}x^{n-1} + m_{k-2}x^{n-2} + \cdots + m_0x^{2t} \quad (5)$$

Parity polynomial is used as well to get the parity symbols completing the codeword.

$$p(x) = p_{2t-1}x^{2t-1} + p_{2t-2}x^{2t-2} + \cdots + p_0 \quad (6)$$

The parity polynomial is remainder of the division of m(x) by g(x), a common hardware approach to do such operation is using the linear feedback shift register system shown below.

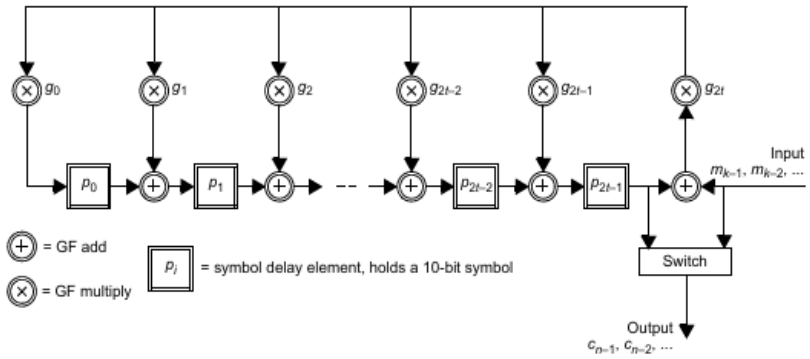



Figure 9: Encoder typical Architecture

The system works through passing the most significant symbols of the message one-by-one till reaching the last symbol where the parity cells $p_i$ will contain the parity symbols. Then a switch is used to concatenate the message symbols with the parity ones to form the systematic block code.

The following system assumes serial-input serial-output structure which addresses high latency with low throughput, but it prioritizes the hardware reuse & the area over them which makes the adoption of parallel architecture a must.

Before going into parallel architecture, one should evaluate this serial approach before scaling it in order to envision the potential of the parallel version of this system.

**Review of RS-encoder Architectures:**

For the sake of clarity, the review is concluded from [13], [14] according to the BCH encoders. Binary BCH encoders have architectures as the RS encoders except for operating over GF (2) so, the GF multipliers can be simply modelled by a short wire or open path (connection/no-connection).

It is worth noting that we still can conclude over the RS encoders since the GF multipliers can be reduced into XORs or no connections as discussed in **GF Math** section.

We will evaluate the architecture in terms of:

1) Timing 2) Power 3) Area4) Scalability (Can be parallel?)

**Conventional Serial Architecture:**

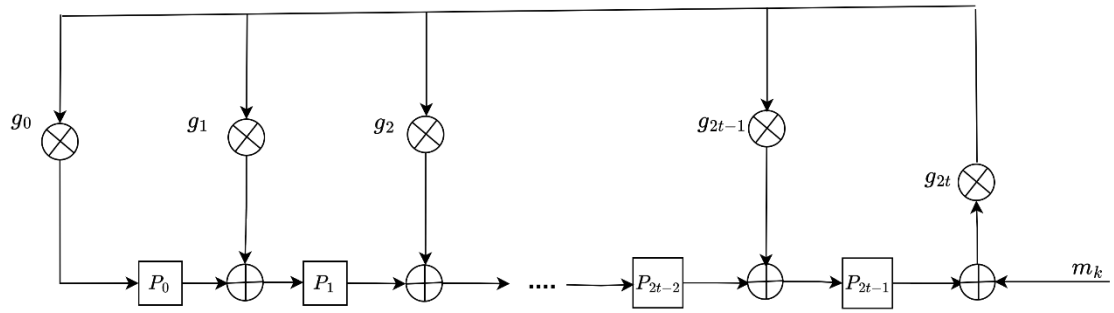



Figure 10: Serial RS Encoder

The serial architecture is simple & promotes area efficiency by hardware reuse over the message symbols over the throughput which may be inefficient especially as system data rates are pushed into the multi-Gbps range, but it suffers from 2 problems: [13].

**Scalability:**

This serial architecture advances the states of the parity cells till reaching the end of the message which limits parallelism since the parity states advance 1 state/cycle.

**Fanout bottleneck:**

A major drawback of this structure is the feedback path from the output of GF multiplier $g_0$ which sees high fan-out due to the input of 2t other multipliers consisting of long XOR chains. This high fanout will cause delays that affect negatively the timing closure, especially for long RS codes as in our case.

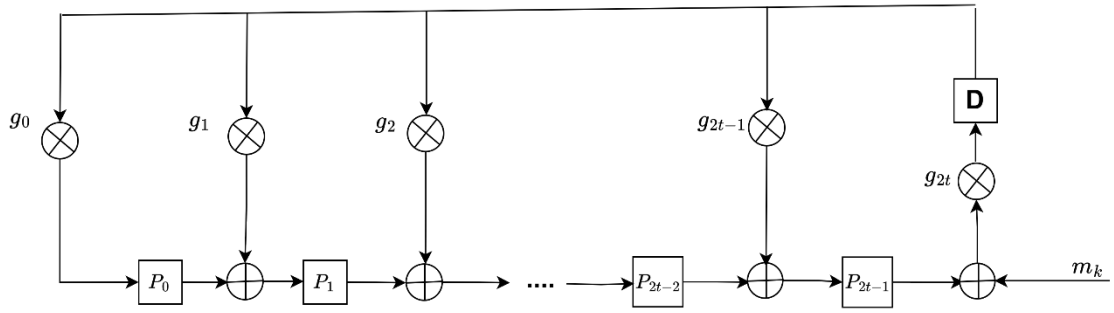



Figure 11: Retimed Version of the Serial Encoder

**Retimed Serial Architecture:**

To address the fanout & timing limitations, retimed versions of the serial encoder have been proposed to find a work-around to redistribute the registers along the feedback path to break the long fanout chains, reducing the critical path delay & enabling higher operating frequencies.

However, this technique suffers from 2 problems:

**Scalability:**

Not only is this architecture an enhanced copy of their serial architecture with the same problem, but it complicates the timing balance between different parts of the pipeline.

**Latency & power:**

Adding extra registers may help to enhance timing but it adds latency, Area & may increase the dynamic power related to the switching activity in the registers [13].

**Unfolded Serial Architecture**

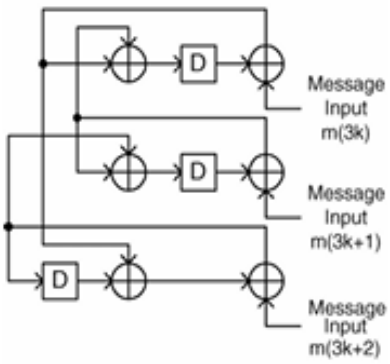



Figure 12: Unfolded RS Encoder Architecture

The unfolding technique is the first method to address the scalability problem by replication of the logic required for **P** cycles of the serial encoder & schedules them to execute simultaneously by rewriting the recursive function of the encoder relating to the next state of the parity registers with the current state of them & the current input.

**This method suffers from several limitations:**

**Area and power overhead:**

Unfolding increases the logic with linear growth in area & increases the switching activity leading to higher dynamic power consumption.

**Timing Complexity:**

The unfolding creates inter-relation between the parallel networks which increases the combinational depth of the logic that scales with long RS codes. This timing problem is worsened by the fact that "retiming" can't be applied to this parallel structure except of special cases explored in [13].

**Incompatible with non-primitive codes:**

This method mainly depends that the states of code can be decimated in time evenly between the hardware units, this is translated into putting a condition on the number of message symbols (K) to be divisible by the unfolding factor (P). This is not the case in the RS (544,514) which can't be scaled by flexible factors.

**State-Transition Architecture:**

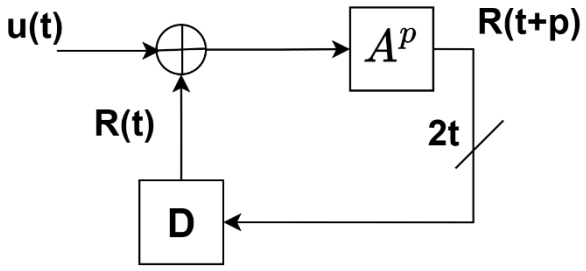



Figure 13: Matrix-based state transition by p states

Matrix-based state transition architectures have been used as a flexible alternative, especially for long codes or large parallel factors. As presented in [14], this method models the LFSR state update as a linear transformation using state & input matrices. By choosing the suitable transformation matrix, we can optimize certain aspects such as: Timing, gate count, both power consumption & gate count as well as choosing the input tap of the LFSR to decrease the feedback fanout problem in the typical serial encoder.[15]

The state transition is done with the help of the state transition matrix A which is constructed to model all the GF multipliers in the feedback path in a recursive manner

$$R_{2tx1}(t+p) = A^{p}_{2tx2t} * (R(t) + U(t)) \qquad (7)$$

A is formulated as follows:

$$A = \begin{bmatrix} G_{n-k-1} & 1 & 0 & \cdots & 0 \\ G_{n-k-2} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ G_1 & 0 & 0 & \cdots & 1 \\ G_0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Where $G_i$ represents the matrix representation of constant GF multiplier by the generator polynomial coefficient $g_i$.

**Iterative Matching Algorithm:**

The optimization technique adopted in the RS codec is the 2D iterative matching algorithm since the state transition matrix may contain many non-zero entries leading to high hardware complexity in terms of area, gate count & power.

To minimize such hazards, the 2D IMA is proposed as heuristic method to minimize the number of logic operations (XORs) leading to power reduction, area & possibly timing by reducing the fan-out as will be discussed.

The core logic of this optimization is to share the common sub-expressions among XOR trees in our case resulting from the GF multipliers as discussed in **GF Math.**

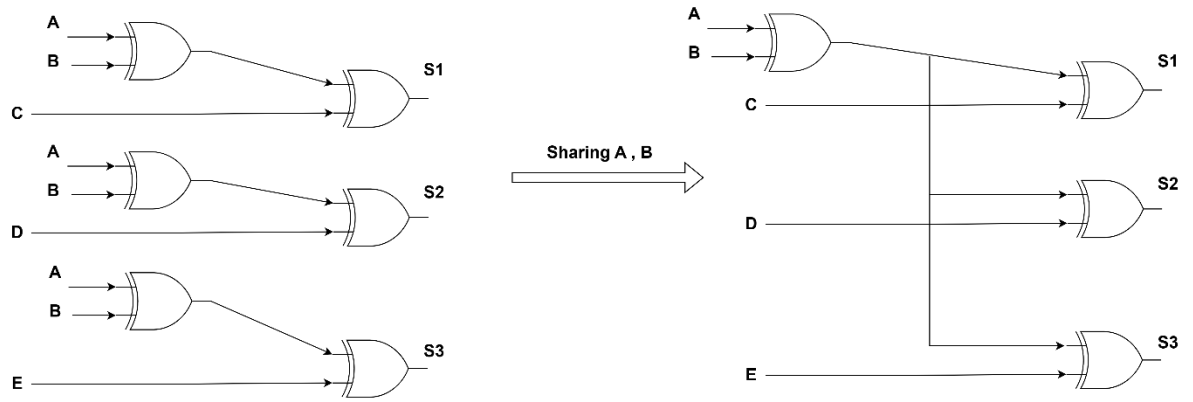



Figure 14: Common Subexpression Logic

**Expectations:**

- The 2D IMA heavily draws on the synthesis tool optimization that sets the possible reuse of the CSE & hence the utilization has potential to Increase if the CSEs aren't structured to balanced XOR Trees.
- If the fanout of the CSE is not high, by producing balanced CSEs that are evenly used in other logic, this may result in timing enhancement as well.
- If the LUTs representing the XORs are reused by the tool by setting some directives, we can decrease dynamic power consumption due to the excessive XOR gates.

## 4.10. Interleaver

The main objective of an **interleaver** is to rearrange the symbols of a codeword in such a way that **burst errors** (errors affecting a sequence of consecutive bits or symbols) do not severely impact the integrity of the entire codeword. By spreading the symbols across time or space, interleaving ensures that when a localized error (such as deep fading in a wireless channel) occurs, it only affects a small part of multiple codewords rather than corrupting one completely. This makes the error correction process much more effective.

**Deep fading** refers to a temporary drop in signal strength due to multipath propagation, interference, or other channel impairments. If a codeword is transmitted without interleaving, such fading could wipe out a contiguous set of symbols, overwhelming the error correction capability of the code. Interleaving mitigates this by dispersing the codeword's symbols across different transmission intervals or channels, reducing the chance that a burst error will destroy a single codeword.

## 4.11. Symbol Distribution

Following the Reed-Solomon (RS) encoding and interleaving stages data must be distributed across multiple Physical Coding Sublayer (PCS) lanes to align with the parallel data path architecture of high-speed Ethernet standards. Specifically, in a 200 GBASE-R PCS, the data is distributed across **8 PCS lanes**, whereas in a 400 GBASE-R PCS, the distribution occurs across **16 PCS lanes**. This Round Robin distribution is performed in units of 10-bit symbols, arranged sequentially from the lowest-numbered to the highest-numbered PCS lane.

## 4.12. Alignment Lock and Deskew

This is the first block in the RX Chain, it's responsible for the synchronization of RX Chain and removal of any skew between the PCS lanes received from PMA.

It interfaces with PMA sublayer through 8/16 lanes in 200/400Gbps Ethernet. Due to the bit multiplexing and electrical and physical characteristics of the wires on the PCB skewing can be introduced between the received PCS lanes which causes the lanes to be misaligned at RX.

This block first achieves lock to each lane independently using the alignment lock process stated by the IEEE standard then it calculates the relative skew between the lanes and eliminates it in the deskew process if the skew between the lane is within the limit stated by the standard.

This block has no previous implementations mentioned in any previous literature, the only requirements on this block are those specified by the IEEE 802.3 standard as throughput and the system's latency, the implementation and architecture shown is the unique work of the authors and is able to achieve these requirements.

**Alignment lock process**

, The block diagram of this process is shown in figure (15), this process is done on each lane independently, its input ports are the unit data of a lane from the PMA sublayer, a control signal "signal_ok" that describes the validity of the data and "restart_lock" signal which is an input from deskew process described in the deskew process.

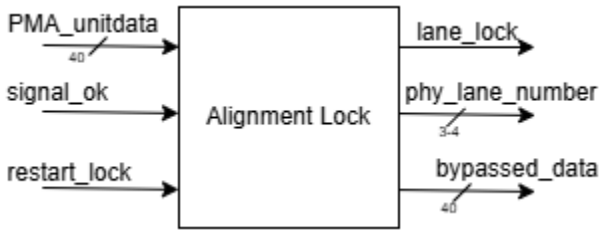



Figure 15: Alignment lock block diagram

The block has three output ports, "lane_lock" which is asserted when a lane is locked, the process of obtaining lock will be explained later, "phy_lane_lock" port which is a 3 or 4 bits signal depending on the speed used is the identified physical lane number of the received data lane, "bypassed_data" is the data bypassed throughout the chain once alignment lock and deskew processes are successfully finished.

The alignment lock process is best described through the finite state machine in figure (16) the process searches the input stream for alignment markers which are 120 bits long having 48 bits called common portion which are identical in all AMs and used to identify them and another 48 bits called unique portion used to identify the lane number of each AM, the remaining bits are unique padding bits which are not used during reception.

Each possible position in the data stream must be checked for AMs as stated by the standard which is handled by the SLIP function. Since the input data stream is processed before error correction, this means that the data will be corrupted by noise meaning that the exact pattern

of the AMs can't be used so the standard defined a criterion for finding a valid AM by comparing the common portion of the candidate block to the actual common portion on a nibble basis, if 9 or more nibbles match then this block is indeed a valid AM, the SLIP function indicates that a valid alignment marker was found by asserting the signal "amp_valid".

Once the first AM is found a counter starts that counts an AM period which is the expected arrival time of the next AM, if this AM is valid and has the same physical lane as the first AM then this lane is locked. A lane then goes out of lock only if "signal_ok" is deasserted or five consecutive invalid AMs are received on this lane or deskew process asserted "restart_lock".

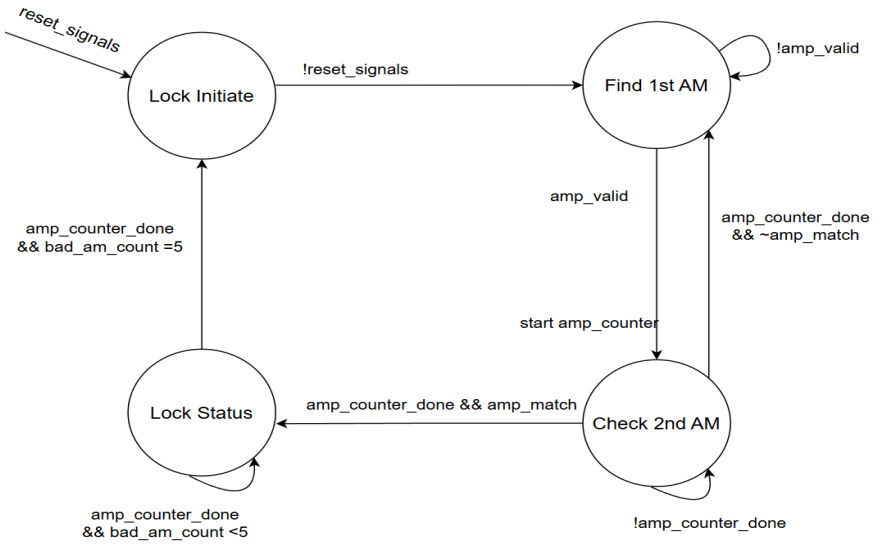



Figure 16: Alignment lock finite state machine

## Deskew process

The block diagram of the deskew process is shown in figure (17). The block operations can be explained using FSM shown in figure(18), it starts its operation whenever a lane or more lock which is indicated by the bus "lane_locks" as it represents the concatenation of "lane_lock" signal of all the lanes. It calculates the skew (relative delay) between the locking of all the lanes relative to the first locked lane, if the skew value is less than the limit all the lanes are deskewed and the physical lane number are unique then "pcs_alignment_valid" is asserted to allow data stream to bypass through the RX chain starting from the AM on each lane otherwise "restart_lock" is asserted to force all the lanes to re-lock. After deskew process is done if 3 consecutive codewords were found uncorrectable from one of the RS decoders, "restart_lock" is asserted to force all lanes to obtain lock again.

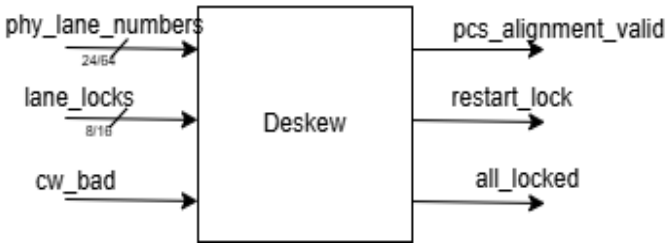



Figure 17: Deskew block diagram

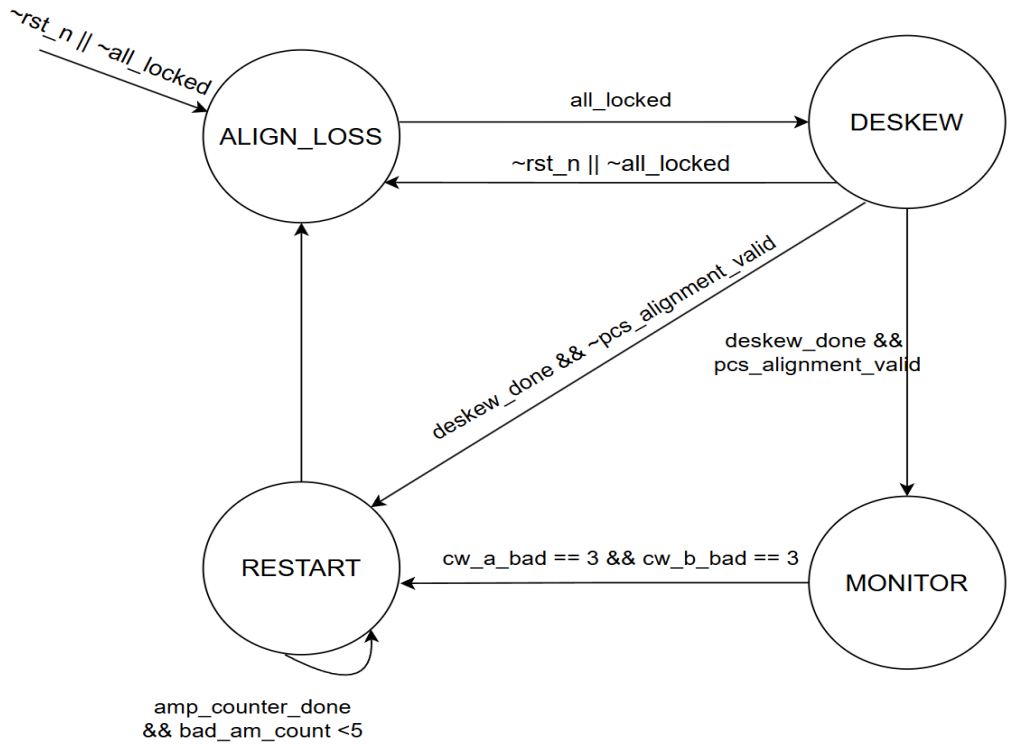



Figure 18: Deskew process FSM

## 4.13. Lane Reorder

In high-speed Ethernet systems (e.g., 200GBASE-R, 400GBASE-R), the data is split across multiple PCS (Physical Coding Sublayer) lanes to handle the data rate. Each PCS Lane transmits a portion of the data to a corresponding physical lane. However, due to skew and serialization issues, the order of arrival may not match the logical order.

Therefore, **Lane Reorder Block** is responsible for:

- Re-aligning the received data based on actual physical lane IDs.
- Reverse the symbol distribution process performed during transmission.

## 4.14. Deinterleave

**Deinterleave** performs the inverse function of the interleaver. Its primary role is to **restore the original symbol order** of the codewords that were altered during interleaving. After transmission over a noisy or fading channel, the received symbols (affected by errors that are now dispersed rather than clustered) are reorganized by the deinterleave to match their initial sequence before decoding. This reordering is essential to allow the **Reed Solomon decoder** to operate effectively, as it expects symbols to be in their original positions.

## 4.15. RS FEC Decoder

The Reed-Solomon decoder extracts the message symbols from the codeword, corrects them as necessary, and discards the parity symbols. The Reed-Solomon decoder shall be capable of correcting any combination of up to $(t = 15)$ symbol errors in a codeword.

The Reed-Solomon decoder shall also be capable of indicating when an errored codeword was not corrected. The probability that the decoder fails to indicate a codeword with $t + 1$ errors as uncorrected is not expected to exceed $10^{-16}$. This limit is also expected to apply for $t + 2$ errors, $t + 3$ errors, and so on.

The RS decoder consists of five blocks as shown in figure (19) and they are:

- **Syndrome Calculation (SC):**
  Identifies whether any errors are present in the received codeword.

- **Key Equation Solver (KES):**
  Determines the error locator and error evaluator polynomials required for error correction.

- **Error locations finder using Chien's search (CS):**
  Locates the positions of errors using the error locator polynomial.

- **Error magnitude evaluation using Froney's algorithm (EE):**
  Calculates the error values corresponding to the error locations found by the Chien search.

- **Error correction:**
  Use the identified error locations and their corresponding values to correct the received codeword **(up to 15 errors).**

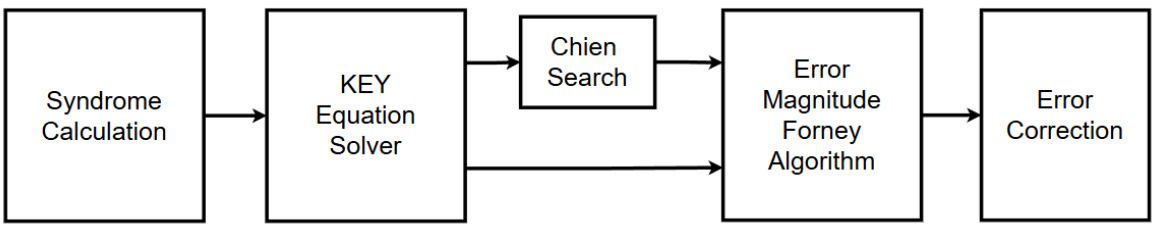



Figure 19: RS decoder Block diagram

### 4.15.1. Syndrome Calculation

Syndrome calculation is the first step in the RS decoding process; its objective is to discover whether an error has occurred or not. The decoder's other stages then find and fix the error or skip the message and remove the parity symbols.

**Implementation theory**

Reed Solomon decoder is implemented can be denoted as RS $(544,514)$ where number of parity symbols is $(2t = 30)$ so that number of calculated syndrome symbols is 30, Let syndrome polynomial is:

$$s(x) = s_1 + s_2 x + s_3 x^2 + \cdots + s_{30}\, x^{29} \tag{8}$$

Each syndrome symbol can be calculated as follows:

$$s_i = r_{543}\, \alpha^{i*543} + r_{542}\, \alpha^{i*542} + \cdots + r_0\, \alpha^0, \qquad \text{where } i \in 1{:}30 \tag{9}$$

If $s(x) = 0$ this means that there is no error in received codeword, otherwise $s(x) \neq 0$ means there is error in received codeword and received codeword need to be corrected if possible.

Since $c(\alpha^i) = 0$ therefore:

$$s_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$$

If $e(x) = 0$ then $s(x) = 0$ otherwise there is error occurred in received codeword.

**Optimization approach**

The main concept of the proposed method is to share common sub-expressions (CSEs) appearing more than once to reduce the hardware complexity, briefly speaking, a CSE is evaluated only once, and then the result is used to replace such sub-expressions.

Applying an efficient, optimized technique called the iterative matching algorithm in addition to a new method that enlarges the search area to find more common sub-expressions by considering all the calculations relevant to syndromes [16].

**Matrix formation for IMA algorithm optimization**

Represent constant Galois Field multiplication in matrix multiplication, where multiplication of received symbol $r_j$ by constant Galois field primitive element $\alpha^{ij}$, where $1 \leq i \leq 2t$ where $2t$ is number of parity symbols, $1 \leq j \leq p$, where $p$ is number of parallel symbols, where received symbols and constant Galois field number is 10-bit symbols of parallel symbols:

$$r_j\, \alpha^{ij} = \begin{bmatrix} r_{j,0} \\ r_{j,1} \\ \vdots \\ r_{j,9} \end{bmatrix} \begin{bmatrix} \alpha_0^{ij} & \alpha_1^{ij} & \cdots & \alpha_9^{ij} \\ \alpha_0^{ij+1} & \alpha_1^{ij+1} & \cdots & \alpha_9^{ij+1} \\ \vdots & \vdots & \cdots & \vdots \\ \alpha_0^{ij+9} & \alpha_1^{ij+9} & \cdots & \alpha_9^{ij+9} \end{bmatrix} = R_j\, A_{ij} \tag{10}$$

Notice: Each element in this matrix is just a single bit; subindex for const Galois field number zero indicates that it is the least significant bit for the corresponding symbol.

Represent single syndrome symbol calculation using multiple constant matrix multiplication, as follows:

$$\delta_i(j+1) = [R_{n-pj-p} \quad R_{n-pj-p} \quad \dots \quad \delta_i(j)] \begin{bmatrix} A_{i,0} \\ A_{i,1} \\ \vdots \\ A_{i,p} \end{bmatrix} \tag{11}$$

Where $\delta_i(j)$ represents $i^{th}$ intermediate syndrome symbol value, where $\delta_i(j+1)$ represents next intermediate value for $i^{th}$ syndrome symbol, it requires $(\mathbf{n/p})$ clock cycle hence intermediate value become actual value of syndrome symbol, where (n: is number of symbol in codeword and equal to **544**) and (p: parallel symbol factor for RS decoder).

$A_{ij}$ Represent matrices for constant multiplication with a dimension matrix $10 \times 10$ matrix, $R_i$ represents the $i^{th}$ received codeword symbol consisting of a 10-bit vector.

Expand matrix to represent calculation for all syndrome symbol values in single matrix as follows:

$$\Delta(j+1) = [\delta_1(j+1) \quad \delta_2(j+1) \quad \dots \quad \delta_{2t}(j)]$$

$$= \begin{bmatrix} R^T_{n-pj-p} \\ R^T_{n-pj-(p-1)} \\ \vdots \\ R^T_{n-pj-1} \\ \delta_1(j)^T \\ \delta_2(j)^T \\ \vdots \\ \delta_{2t}(j)^T \end{bmatrix}^T \begin{bmatrix} A_{1,0} & A_{2,0} & \dots & A_{2t,0} \\ A_{1,0} & A_{2,1} & \dots & A_{2t,1} \\ \vdots & \vdots & \dots & \vdots \\ A_{1,p-1} & A_{1,p-1} & \dots & A_{1,p-1} \\ A_{1,p} & 0 & \dots & 0 \\ 0 & A_{2,p} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & A_{2t,p} \end{bmatrix} = [R_j \quad \Delta(j)] \begin{bmatrix} X_R \\ X_\Delta \end{bmatrix} \tag{12}$$

where $\Delta(j)$ is a binary 1×2tm matrix containing 2t intermediate values of the j-th iteration. $X_R$ is a binary mp×2tm matrix associated with the received symbols, and $X_\Delta$ is a binary 2tm×2tm matrix denoting 2t constant multiplications.

**Iterative matching algorithm**

Main objective is to reduce repetitive calculated sub-expression and find common sub-expression to reduce redundancy in calculation of syndrome symbols [17].

Table 6: IMA Iterations

| iteration | Matrix | Equation |
|---|---|---|
| **1** | $\begin{bmatrix} ① & 1 & ① & ① \\ ① & 0 & ① & ① \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ | $b_0 = a_0 \wedge a_1 \wedge a_2 \wedge a_3$<br>$b_1 = a_0$<br>$b_2 = a_0 \wedge a_1$<br>$b_3 = a_0 \wedge a_1 \wedge a_2$ |
| **2** | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ \boxed{1} & 0 & 0 & \boxed{1} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ \boxed{1} & 0 & 1 & \boxed{1} & 0 \end{bmatrix}$ | $b_0 = a_2 \wedge a_3 \wedge v_1$<br>$b_1 = a_0$<br>$b_2 = v_1$<br>$b_3 = v_1 \wedge a_2$<br>$v_1 = a_0 \wedge a_1$ |
| **3** | $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$ | $b_0 = a_3 \wedge v_2$<br>$b_1 = a_0$<br>$b_2 = v_1$<br>$b_3 = v_2$<br>$v_1 = a_0 \wedge a_1$<br>$v_2 = a_2 \wedge v_1$ |

This example explains CSE search and matrix update for $\alpha^{12}$ constant multiplication for $GF(2^4)$. Each matrix is multiplied by (A) symbol that consists of four bits $(a_3, a_2, a_1, a_0)$ where $a_0$ is least significant bit. For each bit calculation:

- An AND operation is performed between the input symbol bits and the corresponding column of bits in the matrix.
- The results of these AND operations are then combined using an XOR operation to produce the final output bit.

In iteration 2, $a_0 \wedge a_1$ is calculated once and used as a sub-expression to calculate other expressions; hence, we reduce the number of redundancies in the XORing operation. In iteration 3, do further optimization and get a new sub-expression that can include the previous sub-expression; therefore, this optimization will eliminate the existence of a similar XORing operation. This optimization will result in an optimized matrix, where syndrome calculation can be represented in simple matrix multiplication between the input received symbol and the optimized matrix. Hence, we now need to represent a matrix in hardware in an optimized way.

**Matrix generation for hardware implementation**

The matrix is implemented in hardware by mapping its operations into AND, XOR logic gates, removing the need to store the matrix explicitly. This implementation is done using MATLAB:

1. Generate conventional & perform IMA optimization technique to get optimized matrix.
2. Apply iterative matching algorithm on generated conventional matrix, such that it will not stop until there is no remaining common sub-expression not generated.
3. Create file and generate Verilog assign statement that represents the generated optimized matrix.

### 4.15.2. Key Equation Solver

The second block in the RS decoder is responsible for generating the error locator polynomial (λ) and error evaluator polynomial (ω) from the syndromes generated by the syndrome computation module. The error locator polynomial is used to obtain the location of the errors while the error evaluator polynomial is used in calculating the magnitude of each error which is needed in non-binary error correcting code as in this case.

Originally when Reed and Solomon proposed the RS-codec, they suggested that the decoding process would require solving m simultaneous equations where m is the length of the codeword in symbols [18].

Reed Solomon codes are a sub class of BCH codes discovered in 1960, The first decoding algorithm was devised by Peterson then improved several times in the following order by Gorenstein, then Chien then Forney then Berlekamp and Massey and others [7].

There are many algorithms used to find λ and ω, the most popular are the Berlekamp Massey and Euclidean algorithms as they are the most efficient hardware implementations. Also, their scalability as error correction capability increases is the best, which will be shown later in detail. The BMA and Euclidean algorithms are iterative algorithms that find a heuristic solution for λ and ω, if the number of errors in a codeword is less than or equal to the error correction capability, the BMA finds the correct polynomials otherwise it fails and produces incorrect polynomials. This failure is guaranteed to be detected by the error position evaluation block which usually uses Chien's search method if the number of errors in the codeword is below the error detection capability.
The first version of the BMA algorithm was based on shift registers [19] and used Galois field inverters which caused the critical path delay to increase limiting the decoder speed. Modifications to the algorithms were made [20], [21] to replace the GF inverters with multipliers which improved the critical path delay and the area. The Euclidean algorithm is based on finding the greatest common divider between two polynomials and their Bezout coefficient with many modifications done to it similar to BMA.

A survey of the BMA and Euclidean algorithms and different modifications is done to find the suitable algorithm to use. The system specifications specified by the IEEE 802.3 standard is concerned with throughput and latency. The chosen algorithm must have latency and small critical path delay to allow for high frequency operation to achieve the required throughput, Utilization/area requirements is another metric that is also considered.

Only Parallel version of the algorithms is considered to be able to achieve the high throughput and low latency requirements, a comparison between recent modifications of the BMA and Euclidean algorithms in carried out as shown in table (7). Euclidean algorithm was preferred early on due to its homogenous architecture which allowed for low-cost single chips of RS decoders to be implemented and made its layout easier. When the speed requirements increased, the modifications to the Euclidean algorithm fell behind those of BMA in terms of critical path delay making the BMA the algorithm of choice for high-speed applications.

Table 7: Literature Survey of different implementations of BMA and Euclidean algorithm

| Algorithm Name | # Adders | # Multipliers | # Muxes | # Registers | Latency | Critical Path Delay |
|---|---|---|---|---|---|---|
| Modified Euclidean [22] | 4t+2 | 8t+8 | 8t+8 | 4t+t | 2t | Mult + Add +Mux |
| Inversion less BMA (IBMA) [23] | 3t+1 | 5t+3 | 2t+1 | 6t+2 | 2t | 2*Mult+ $\lceil \log_2(t+1) \rceil$ |
| Reformulated IBMA [23] | 3t+1 | 6t+2 | 3t+1 | 6t+2 | 2t | Mult + Add |

From the literature survey it's obvious that the BMA is the algorithm needed in our system due to the low latency and its ability to operate at high operating frequency requirements. The algorithm and the modifications are explained in the next section along with the implementation and architecture results.

It's worth mentioning that other modifications to the BMA may trade off lower area at the cost of higher latency. Even though these modifications are very smart and truly impactful for certain applications, in our case latency is an important metric that can't be increased due to the standard specifications.

### 4.15.3. Chien Search

The Chien Search evaluates the error locator polynomial λ(x), which is provided by the Key Equation Solver (KES) block, to determine the locations of errors. This process involves an exhaustive search over all possible powers of the primitive element ($\alpha^i$), such that $0 \leq i \leq n-1$.

locator polynomial is expressed as : $\boldsymbol{\lambda(x) = \lambda_t X^t + \lambda_{t-1} X^{t-1} + \cdots + \lambda_1 X + \lambda_0}$, where $\left(t = \frac{n-k}{2}\right)$ and it represent the **maximum number of errors** that the RS(n, k) code can correct. Hence, after substituting it with all possible combinations, the error is located at position (**i**) if and only if $\boldsymbol{\lambda(\alpha^i) = 0}$.

Chien Search was introduced by R. T. Chien in 1964 as an efficient means to locate errors by evaluating the error locator polynomial at all possible combinations of primitive element.

Chien Search is the only block in the RS decoder that decide whether the codeword is correctable or not by taking the locator polynomial and start searching in its code space, if the locator polynomial produce roots **equal to** the degree of error locator polynomial, therefore the codeword is correctable else the codeword can't be corrected and the flag will be asserted known as **CW_bad**. This signal will alarm the error magnitude block and error correction block to bypass the codeword without any correction.

- Various hardware implementations have been developed:

1. **Serial architecture:** This architecture is based on computing only one evaluation per cycle, so it is simple architecture but very slow (high latency) as it requires **(n)** clock cycles to complete the overall evaluations and this is impossible for high-speed systems. [in our system, $n = 544$]

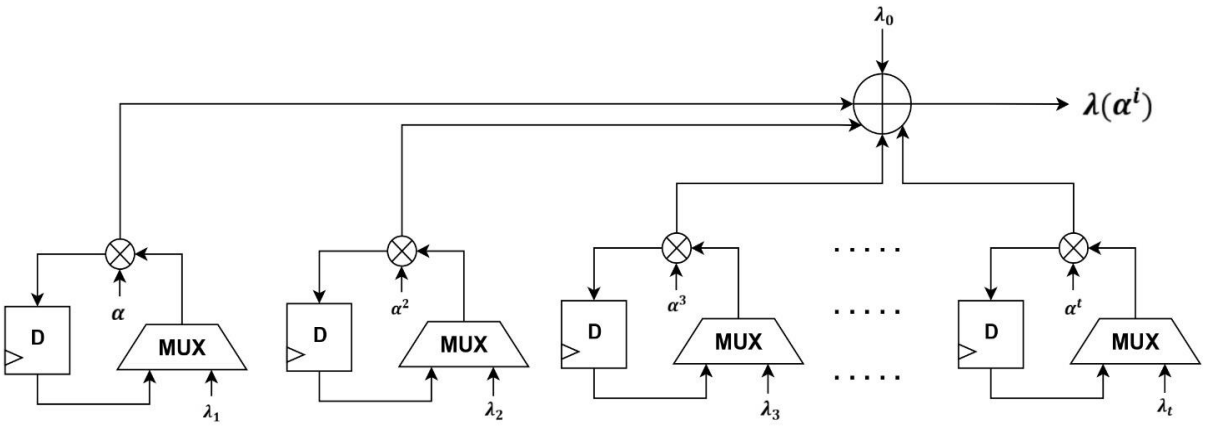



Figure 20: Serial Architecture of Chien Search

2. **Fully parallel architecture:** This design performs all polynomial evaluations simultaneously within a single clock cycle, resulting in extremely high speed. However, it requires a large amount of hardware resources, making it a highly area-intensive solution.

   Therefore, to compromise between speed and hardware efficiency, a **partially parallel architecture** is adopted. In this approach, (**P**) evaluations are performed in each clock cycle, enabling high throughput while significantly reducing area consumption compared to a fully parallel architecture.

### 4.15.4. Forney Algorithm

The final step in the decoding process is to determine the error magnitude values to enable error correction. There are two methods: the "transform decoding process" (which operates in the frequency domain) and the "Forney algorithm" (which operates in the time domain). Although the time-domain method does not require Galois field inversion or a Chien search, it occupies a very large area. The Forney algorithm is preferred due to its lower circuit complexity and smaller area footprint.

**Forney mathematics theory**

After evaluating error locator and error evaluator polynomials in KES, and roots of error locator polynomial were found by Chien search, Forney algorithm start to find error magnitude values to enable error correction for received codeword.

**Formal derivation**

Derivation of Galois field derivative. Let polynomial:

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_t x^t \tag{13}$$

the derivation of this polynomial is

$$f'(x) = f_1 + 2f_2x + 3f_3x^3 + \cdots + tf_tx^{t-1} \quad (14)$$

In GF($2^m$), the addition is XOR (i.e., $\alpha_i + \alpha_i = 0$), so even-powered terms vanish in the derivative:

$$f'(x) = f_1 + f_3x^2 + f_5x^4 + \cdots + f_tx^{t-1} \quad (15)$$

So that derivative simply led to all coefficients of even power of polynomial to reduce to zero.

**Error magnitude evaluation**

Given error locator and evaluator polynomials which defined as:

$$\Omega(x) = \Omega_0 + \Omega_1x + \Omega_2x^2 + \cdots + \Omega_{14}x^{14} \quad (16)$$

$$\lambda(x) = \lambda_0 + \lambda_1x + \lambda_2x^2 + \cdots + \lambda^{15}x^{15} \quad (17)$$

Where degree of error locator polynomial is $2t = 15$, degrees of error evaluator is $2t - 1 = 14$.

Error magnitude values are computed for each corresponding root of error locator polynomial by this equation:

$$e_{il} = \frac{\Omega(X_l^{-1})}{\lambda'(X_l^{-1})} \quad (18)$$

Where:

| | |
|---|---|
| $\Omega(x)$ | error evaluator polynomial with degree $2t - 1$ |
| $\lambda'(x)$ | Derivative of error locator polynomial |
| $X_l^{-1}$ | $l^{th}$ root of error locator polynomial |

Evaluation of this polynomial will require the following operation:

- Derivation of error locator polynomial where it can be defined as:

$$\lambda'(x) = \lambda_1 + \lambda_3x^2 + \lambda_5x^4 + \cdots + \lambda_{15}x^{14} \quad (19)$$

- Exponential Galois field to substitute is in error locator and evaluator polynomials by roots of error locator polynomial which required special circuit for this operation as it is not standard operation.
- Galois field inversion for result of substitution of roots of error locator polynomial in derivative of error locator polynomial.

### 4.15.5. Error Correction

In the context of the physical coding sublayer (PCS), the correction block plays a fundamental role within the forward error correction pipeline. Its main function is to handle error correctio operations based on inputs received from pervious stages. Specifically, the syndrome decoder and FIFO buffer. This block operates on a codeword-by-codeword basis and is designed to manage three types of information per codeword: Error Magnitude, Error location, Error enables signals. These inputs are essential for identifying the specific bits within a codeword that require correction as well as determining the validity and state of the incoming codeword. Upon receiving data from pre sending stage via FIFO buffer, the correction block first checks the codeword statues.

## 4.16. Post-FEC Interleaver

Within the Physical Coding Sublayer (PCS) of the communication system, the Post-FEC Interleave Block plays a crucial role as part of the Forward Error Correction (FEC) chain. This block acts primarily as a data rearrangement and rate adaptation stage, placed immediately after the FEC Decoder modules and just before subsequent processing blocks. The purpose of this module is to ensure a smooth and consistent data stream output that adheres to specific size and timing requirements. Functionally, the Post-FEC Interleave block can be abstracted as a storage and reshaping element. It receives 320-bit wide input data, which originates from two parallel FEC decoders, each producing 160-bit frames. However, the output required from this block is only 257 bits wide, resulting in a need for internal rate and width adaptation, while also ensuring data continuity and integrity. One of the primary challenges in such a configuration arises from the mismatch between the input and output data widths. Since the output frame size is smaller than the incoming data, a direct pass-through would eventually require an infinitely growing buffer space scenario that is practically impossible. To address this, the design divides the Post-FEC Interleave block into four sub-blocks as **interleaver**, **Asynchronous FIFO, Asynchronous FIFO** and **Continuous Data Formatter**. Together, these sub-blocks ensure that the Post-FEC Interleave stage delivers rate-matched, width-adjusted, and format-consistent data to the next stage of the PCS pipeline, overcoming practical implementation challenges such as buffer overflow and CDC hazards.

## 4.17. Alignment Removal

The AM removal block is necessary to counteract the AM insertion process in the **Physical Coding Sublayer (PCS)** of Ethernet, as defined in IEEE 802.3 [1]. AM is not useful for higher layers since it is only used for alignment purposes for PCS lanes, meaning it has no value for the Reconciliation layer. When AM is removed, there is a need to insert a block to preserve the receiving rate.

The AM insertion process occurs every AM period, which is 4096 codewords for 200Gbps and 8192 codewords for 400Gbps as defined in IEEE 802.3 [1], and AM must be added constantly at the beginning of each AM period. The AM removal block can make use of this property so that it can always remove the first 1028 bits (or four 257-bit blocks) received, since AMs are guaranteed to be inserted at a constant position within the AM period. The block removes AM markers and inserts a 257-bit idle block to maintain the receiving rate after removal.

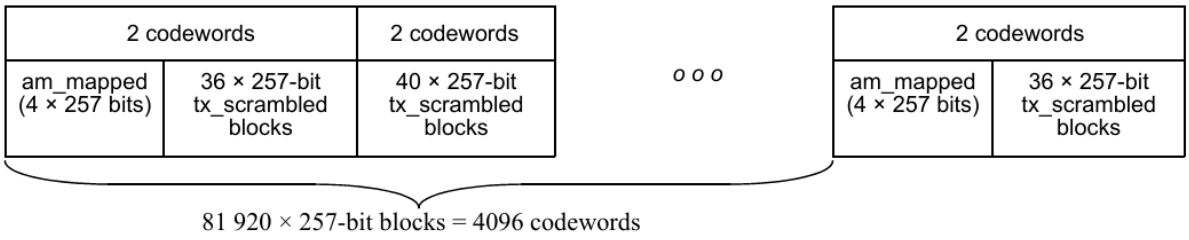



Figure 21: AMs position within AM period [1]

When AM removal inserts 257-bit idle blocks, they must bypass the descrambler so that they remain unchanged. Thus, AM removal signals the descrambler to bypass these blocks without modification. After AM removal and idle insertion, AM removal will bypass all incoming blocks to the descrambler until the start of the next AM period, when it will repeat these operations again. This can be summarized in the following diagram:

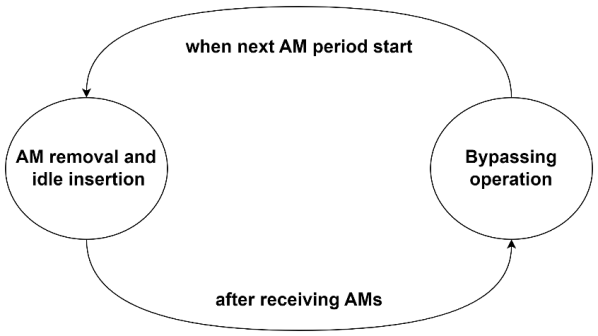



Figure 22: AM operation state diagram

## 4.18. Descrambler

Descrambler is the inverse process to the scrambler. So, it can be designed following the same principles used in the scrambler.

## 4.19. Reverse Transcoder 257B/256B

The Reverse Transcoder block extracts four 66-bit blocks, labeled rx_coded_j (where j ranges from 0 to 3), from each incoming 257-bit block rx_xcoded, starting with bit 0 as the first received and transmitted bit. These blocks are output sequentially from j = 0 to j = 3.

**Note:** If the error indication signal (CW BAD) is available or enabled, and the Reed-Solomon decoder detects uncorrectable errors in a codeword, the PCS receive function must replace all corresponding 66-bit blocks (from the two affected codewords) with error blocks. This is done by setting the sync header of each 66-bit block to 11 during the reverse transcoding process (256B / 257B to 64B / 66B).

## 4.20. Decoder 66B/64B

The 66B/64B decoder is an essential element of the PCS layer in IEEE 802.3-compliant systems. In the receive PCS, it decodes four dependent blocks to generate RXD [255:0] and RXC [31:0], which are forwarded to the 200GMII/400GMII interface. Each block yields one data transfer for the 200GMII/400GMII. The 66B/64B decoding scheme is specified in IEEE 802.3 [1].

To perform the decoding, refer to the 66B/64B block type field format table and the control code table provided below.

| Input Data | Sync | Block Payload | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit Position: / Data Block Format: | 0 1 | 2 | | | | | | | 65 |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,D_7$ | 01 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| **Control Block Formats:** | | **Block Type Field** | | | | | | | |
| $C_0\,C_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x1E | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $S_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,D_7$ | 10 | 0x78 | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| $O_0\,D_1\,D_2\,D_3\,Z_4\,Z_5\,Z_6\,Z_7$ | 10 | 0x4B | $D_1$ | $D_2$ | $D_3$ | $O_0$ | 0x000_0000 | | |
| $T_0\,C_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x87 | | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,T_1\,C_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0x99 | $D_0$ | | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,T_2\,C_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0xAA | $D_0$ | $D_1$ | | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,T_3\,C_4\,C_5\,C_6\,C_7$ | 10 | 0xB4 | $D_0$ | $D_1$ | $D_2$ | | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,T_4\,C_5\,C_6\,C_7$ | 10 | 0xCC | $D_0$ | $D_1$ | $D_2$ | $D_3$ | | $C_5$ | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,T_5\,C_6\,C_7$ | 10 | 0xD2 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | | $C_6$ | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,T_6\,C_7$ | 10 | 0xE1 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | | $C_7$ |
| $D_0\,D_1\,D_2\,D_3\,D_4\,D_5\,D_6\,T_7$ | 10 | 0xFF | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | |

Figure 23: 66B/64B Block formats

Table 8: Control Codes

| Control character | Control character representation in MII (8-bit) | Control codes in PCS (7-bit) |
|---|---|---|
| /I/ | 0x07 | 0x00 |
| /LI/ | 0x06 | 0x06 |
| /E/ | 0xFE | 0x1E |
| /S/ | 0xFB | Implicitly in block type field |
| /T/ | 0xFD | Implicitly in block type field |
| /Q/ | 0x9c | Implicitly in block type field plus O0 |

The decoding process works in reverse. It begins by using the sync header to identify the block type. For data blocks with a sync header of 01, the 8-bit 200G/400GMII control signal from RXC [31:0] is set to 00000000, and the 64-bit data from RXD [255:0] directly corresponds to the data payload.

For control blocks with a sync header of 10, the block type field is referenced using the table below to determine the specific control block format. Then, based on the tables provided, a 64-bit 200G/400GMII RXD and an 8-bit RXC control signal are decoded accordingly.

Table 9: Block format in MII

| Four Decoding Blocks | RXD 256B | RXC 32B |
|---|---|---|
| Data Block 64B | $D_0\ D_1\ D_2\ D_3\ D_4\ D_5\ D_6\ D_7$ | 00000000 |
| Control Blocks 64B | | |
| Idle Block | $C_0\ C_1\ C_2\ C_3\ C_4\ C_5\ C_6\ C_7$ | 11111111 |
| Start Block | $S_0\ D_1\ D_2\ D_3\ D_4\ D_5\ D_6\ D_7$ | 10000000 |
| Order Set Block | $O_0\ D_1\ D_2\ D_3\ Z_4\ Z_5\ Z_6\ Z_7$ | 10000000 |
| Terminate 0 Block | $T_0\ C_1\ C_2\ C_3\ C_4\ C_5\ C_6\ C_7$ | 11111111 |
| Terminate 1 Block | $D_0\ T_1\ C_2\ C_3\ C_4\ C_5\ C_6\ C_7$ | 01111111 |
| Terminate 2 Block | $D_0\ D_1\ T_2\ C_3\ C_4\ C_5\ C_6\ C_7$ | 00111111 |
| Terminate 3 Block | $D_0\ D_1\ D_2\ T_3\ C_4\ C_5\ C_6\ C_7$ | 00011111 |
| Terminate 4 Block | $D_0\ D_1\ D_2\ D_3\ T_4\ C_5\ C_6\ C_7$ | 00001111 |
| Terminate 5 Block | $D_0\ D_1\ D_2\ D_3\ D_4\ T_5\ C_6\ C_7$ | 00000111 |
| Terminate 6 Block | $D_0\ D_1\ D_2\ D_3\ D_4\ D_5\ T_6\ C_7$ | 00000011 |
| Terminate 7 Block | $D_0\ D_1\ D_2\ D_3\ D_4\ D_5\ D_6\ T_7$ | 00000001 |

# 5. Hardware Implementation and Architecture

## 5.1. Galois Fields

### Galois Field Multipliers

The block diagram of the GF multiplier is shown in figure (24), it has 2 input ports each of size m bits and one output port which is the result of the multiplication and of size m bits, it also has a parameter which is the primitive polynomial which is of size m, this can also be a port but in our application it doesn't change so it's defined as a parameter.

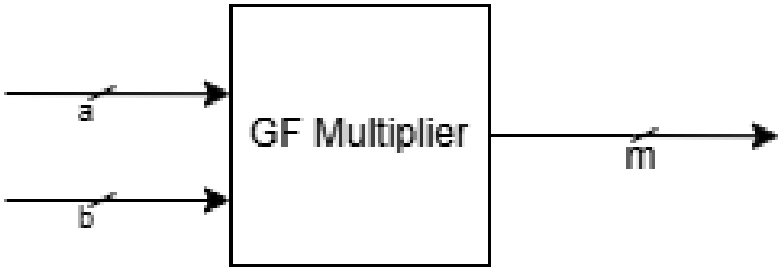



Figure 24: GF multiplier block diagram

Two architecture of GF multipliers are implemented. The first architecture is shown in figure (25) and, the stages can be pipelined by adding a register that registers the polynomial multiplication stage result, The second architecture is shown in figure (26). Each architecture uses AND & XOR gates but has different main principles.

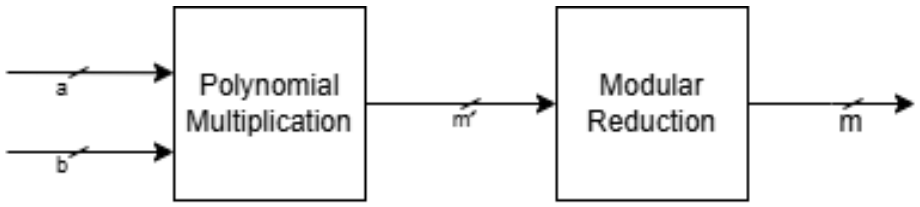



Figure 25: GF multiplier first architecture

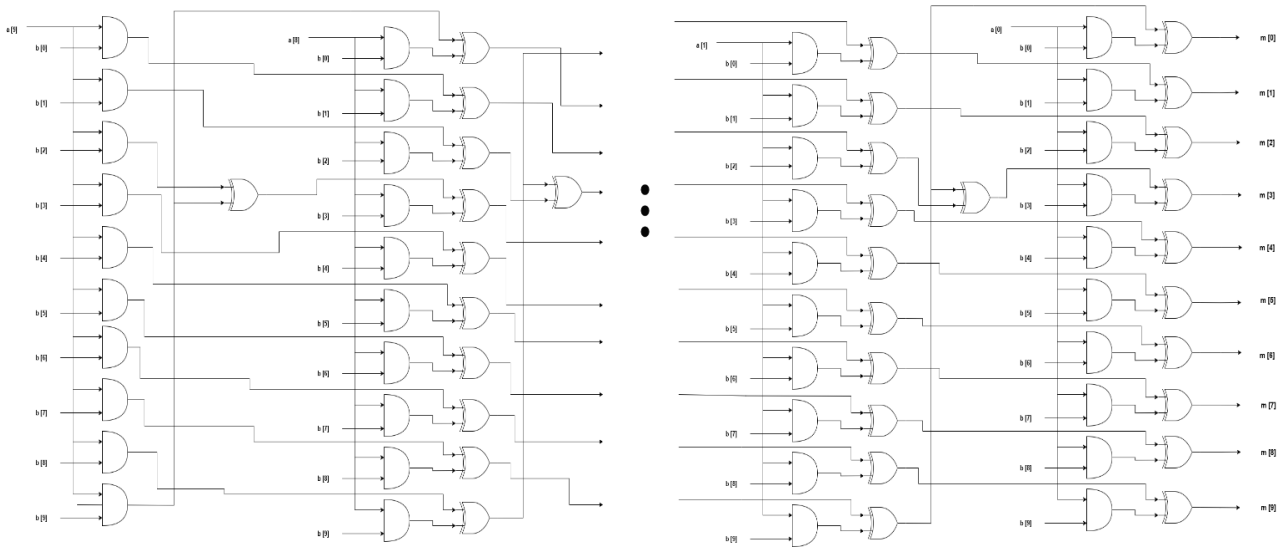


Figure 26: GF multiplier second architecture

**Galois Field Exponential**

**Binary method algorithm**

The LSB-approach of this algorithm is used because there is no need to pre-computation for this algorithm rather than MSB first where there is need for power values of input GF number. The LSB-approach Where exponent computation of GF number is evaluated starting from its LSB bit.

For non-zero element M in $GF(2^m)$ and any exponent integer e that is represented in m-bits:

$$M^e = M^{e \bmod (2^m - 1)} \tag{20}$$

Where LSB-first approach calculates:

$$R = M^e \bmod G \tag{21}$$

Where G is irreducible polynomial and for $GF(2^{10})$ is:

$$G = 1 + x^3 + x^{10} \tag{22}$$

By processing each bit of exponent number starting from LSB to MSB as follows:

- **Initialization step:** $R = 1 \ \& \ C = M$
  Initialize R with multiplication identity, and C is temporary variable to store square terms.

- **Iterate for each bit in exponent number e starting from LSB:**
  If $e_i = 1$ (where i is number of bits that represnt expoenent number) multiply by R by C and store result in R, otherwise keep value of R not changed. Square C in each iteration.

- **Termination:**
  After processing all bits of exponent number e, then R will hold value of $M^e$.

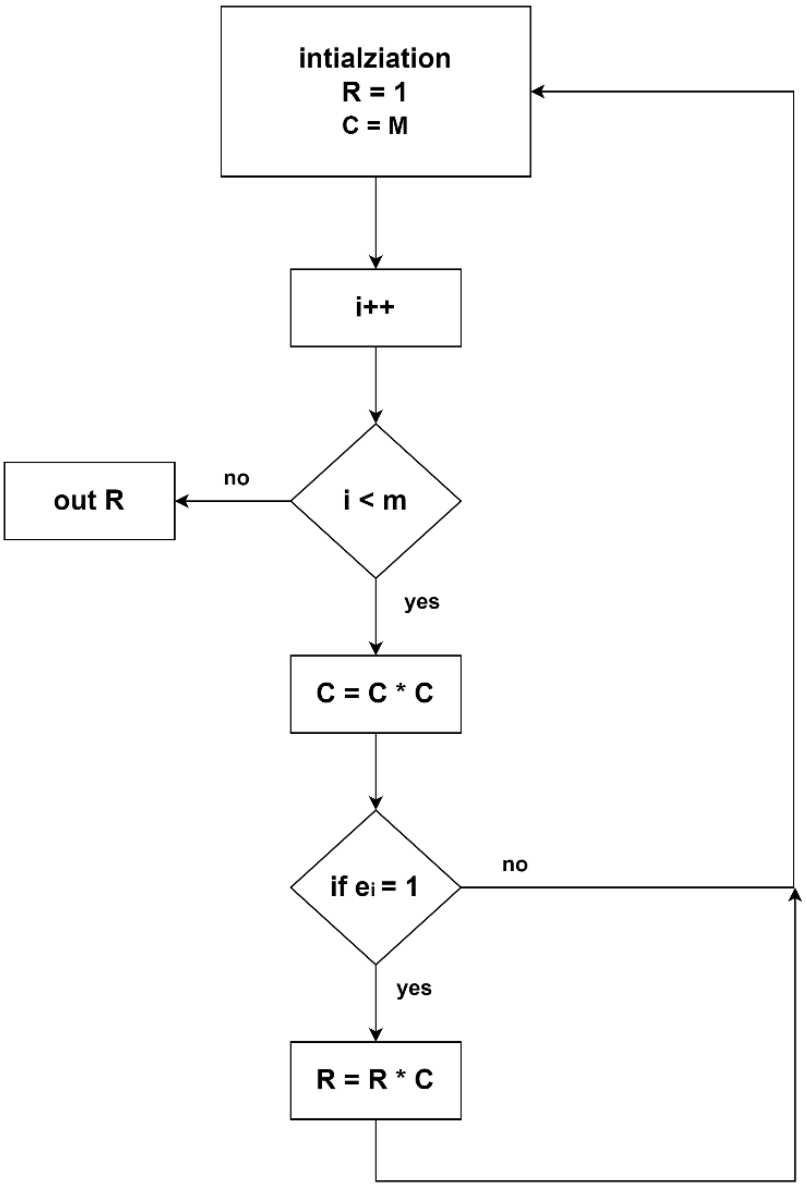



Figure 27: Flow chart of binary method

**Galois Field Inversion**

- In Itoh and Tsujii proposed a very efficient method to compute the inversion in GF $(2^m)$ using the addition chains with at most $2\log_2(m-1)$ field multiplications. An addition chain for computing a positive integer can be given by a sequence of positive integers and a sequence of index pairs such that each term in the sequence is the sum of two previous terms specified by its index pair More precisely, in their algorithm the number of multiplication is reduced to $\log_2(m-1) + HW(m-1) - 1$ (Hamming weight ) and the number of required squaring is $m-1$.

- An addition chain for computing a positive integer can be given by a sequence of positive integers and a sequence of index pairs such that each term in the sequence is the sum of two previous terms specified by its index pair.

Table 10: GF Inversion Addition Chain

| $\mathbf{u_i}$ | $\mathbf{u_i = u_{i-1} + u_j}$ | $\boldsymbol{\beta_{u_i} = A^{2^i-1}}$ |
|---|---|---|
| $1 = u_0$ | $u_0$ | $\beta_1 = A$ |
| $2 = u_1$ | $u_0 + u_0$ | $\beta_2 = (\beta_1)^2 * \beta_1$ |
| $4 = u_2$ | $u_1 = u_1$ | $\beta_4 = (\beta_2)^{2^2} * \beta_2$ |
| $8 = u_3$ | $u_3 + u_3$ | $\beta_8 = (\beta_4)^{2^4} * \beta_4$ |
| $9 = u_4$ | $u_3 + u_0$ | $\beta_9 = (\beta_8)^2 * \beta_1$ |

## 5.2. Encoder 64B/66B

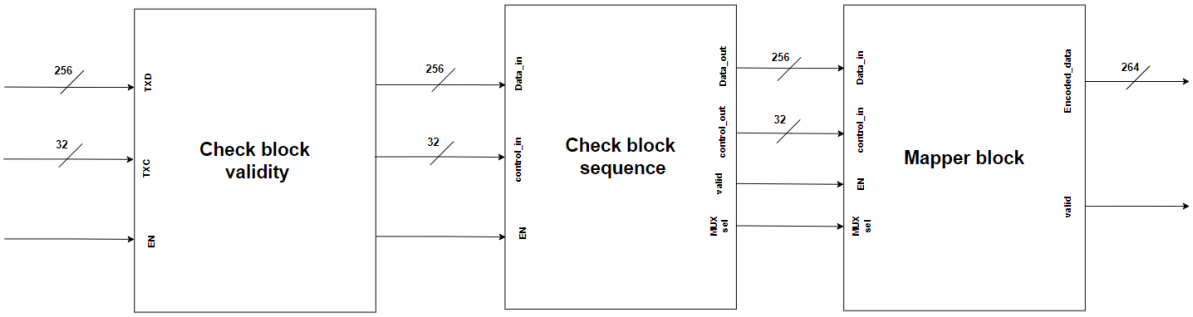



Figure 28: Encoder 64B/66B Architecture

### Check block validity

This stage starts to check block compatibility with the mentioned block format transmitted, determined by the block format in MII. This block checks consistency between control bits and block octets as follows:

Table 11: TXC Block Mapping

| TXC | Expected block |
|---|---|
| 100000000 | 1. Start block: check first character whether match start character or not<br>2. Order set block: first character should be an order set control character, and the last four characters are zeros. |
| 111111111 | 1. IDLE block: where 8 characters should be idle control character.<br>2. Error block: 8 characters must be error character.<br>3. Terminate block: terminate character is in first character and remaining character must be idle control character. |
| 01111111 | The terminate block must be in 2nd character and followed by idle characters. |
| 00111111 | The terminate block must be in 3rd character and followed by idle characters. |
| 00011111 | The terminate block must be in 4th character and followed by idle characters. |
| 00001111 | The terminate block must be in 5th character and followed by idle characters. |
| 00000111 | The terminate block must be in 6th character and followed by idle characters. |
| 00000011 | The terminate block must be in 7th character and followed by idle characters. |
| 00000001 | The terminate block must be in 8th character and followed by idle characters. |

If the incoming input does not match any cases, the encoder generates an error block consisting of 8 error control characters along with their control bits to ensure consistency with the generated error block.

### Check block sequence

The first stage is to check block compatibility, and then the encoder should check the block sequence to see whether it complies with the state machine or not.

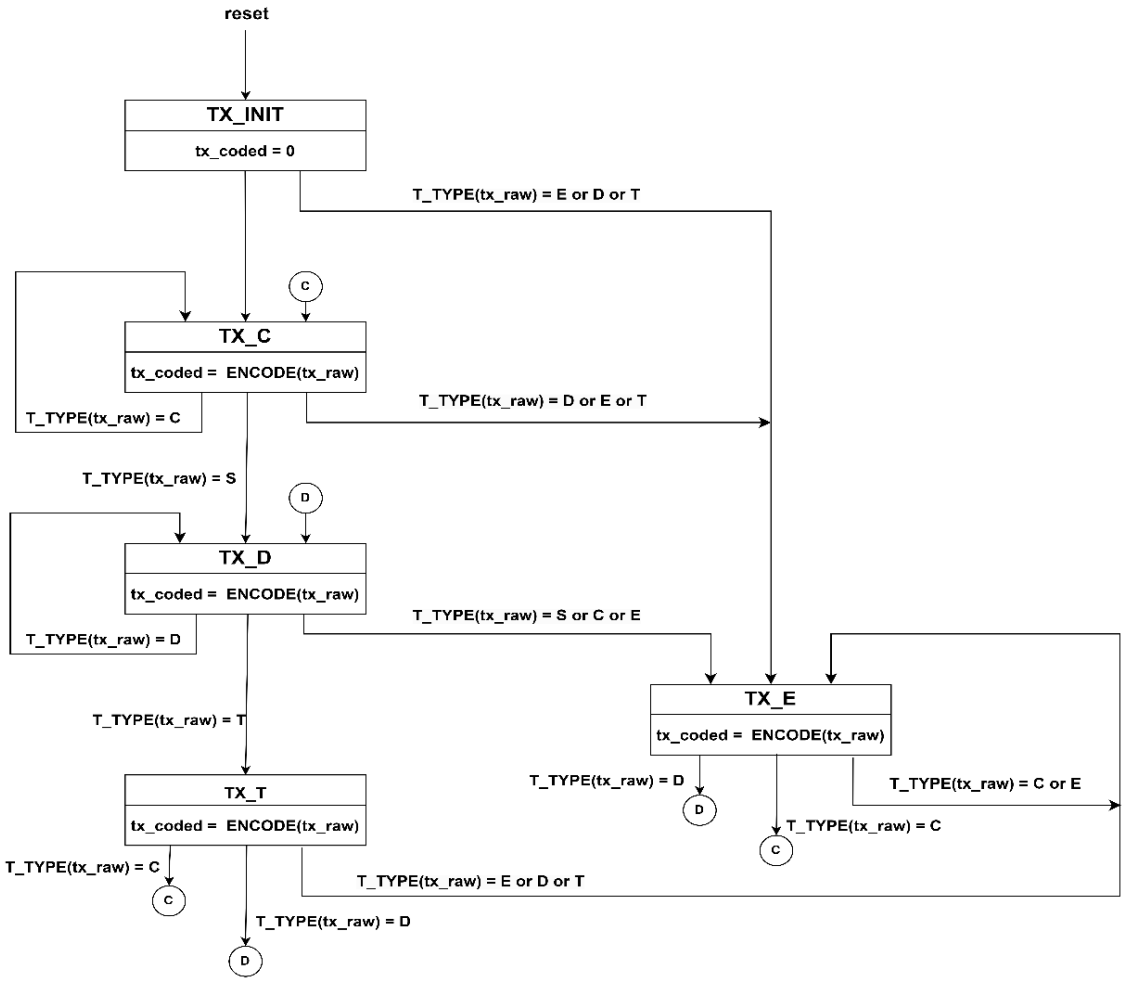



Figure 29: State Machine of Encoder64B/66B

### States description

Table 12: Encoder 64B/66B FSM States

| State name | Description |
|---|---|
| TX_INIT | Initial state of Encoder after reset |
| TX_C | Control state is when an incoming block consists of 8 idle control characters. |
| TX_D | Data states if the incoming block consists of 8 data character and its control bits (TXC = 00000000) |
| TX_T | Terminate state is when incoming block includes a terminate character in any of 8 possible character position. |
| TX_E | Error state when block consist of 8 error control character. |

**Function used in state machine**

| Function name | Description |
|---|---|
| T_TYPE: | This function recognizes the type of incoming 64-bit block with its corresponding 8-bit control bits and specify its type to following types based on their content:<br>1. IDLE block 2. Terminate block 3. Data block<br>4. Start block 5. Order set block 6. Error block |
| ENCODE: | Encode 64-bit block and its 8-bit control bits to 66-bit with proper sync header. |

**Mapper block**

At this stage, the incoming 64-bit block is mapped to a 66-bit block representation with a synchronization header and block type field for control payload blocks. The encoded data takes the format mentioned in table 3 of 64B/66B block formats [24].

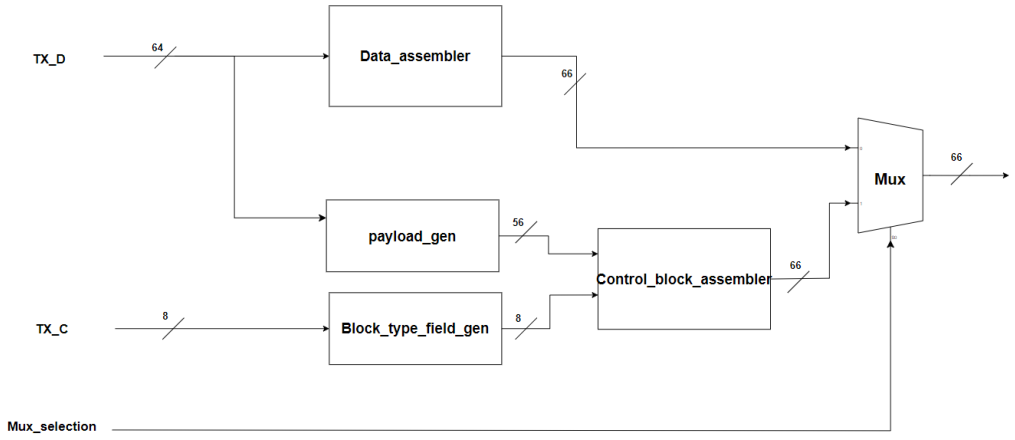



Figure 30: Mapping block unit

The mapper block consists of 4 mapping block units each map single 64-bit block to 66-bit block. Mapping block consists of the following:

**Data assembler**
Combine 64-bit data character with synchronization header "01".

**Payload gen**
The payload gen block generates the corresponding 56-bit block payload, it performs required mapping and padding to create a 56-bit block.

**Block type field gen**
The "block type field gen" block generates an 8-bit block type field according to the type of the block that has been identified in previous stage.

**Control block assembler**
Control Block Assembler assembles the sync header “10, the 8-bit block type field and the 56-bit control code payload into a 66-bit control block.

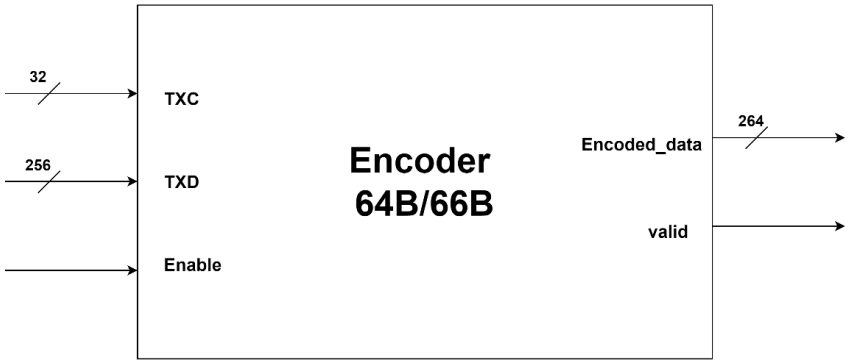



Figure 31: Encoder64B/66B block

Table 13: Encoder 64B/66B Signals and Interface

| Signal name | direction | width | Description |
|---|---|---|---|
| TXD | Input | 256 | Data block transmitted by RECONCILIATION layer it takes formats mentioned in block format table. |
| TX_C | Input | 32 | Control bits where if control bit is one means control block, and when zero it means their data character exists in corresponding octet position. |
| Enable | Input | 1 | Enable signal indicates that data and control bits are ready. |
| Encoded_data | Output | 264 | Encoded data which is result of encoding process it takes formats mentioned in 64B/66B block formats. |
| valid | Output | 1 | Valid signal is raised when there is encoded data. |

**Functional description**

The function of Encoder64B/66B was done over a pipeline where there are three main stages for the encoding process:

- Checking block validity where to check each 64-bit block with its corresponding 8-bit control character is compatible with block formats in standard or not. Generate an error block if the encoder identifies any mismatch between the block and the expected block format.
- Checking block sequence for 4 64-bit blocks where the assumption is the first 64-bit block in the stream is located at the least significant bit. The finite state machine was developed to process four simultaneous 64-bit blocks in a single clock cycle and ends up with the state that is defined by the most significant 64-bit block.
- Mapping each 64-bit block independently to its equivalent format in a 66-bit block defined by the 64B/66B block format. There are two different mapping processes:
  - ❖ Mapping data block done through data assembler block where "01" sync header is added to block payload.
  - ❖ Mapping control blocks involve three blocks. The first is payload generation, where Characters in the payload block are mapped to the appropriate format as defined in the 64B/66B standard. The second is the block type field generation, which inserts suitable block type field bits. The last is the control block assembler, which gathers the payload and block type field in addition to the sync header and generates a 66-bit block.

## 5.3. Rate matching

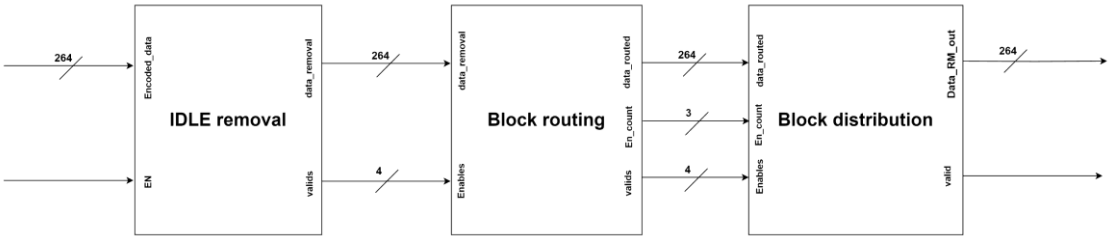



Figure 32: RM architecture

Rate matching operation can be divided into three main blocks:

### IDLE removal

The first stage of rate matching is where the IDLE blocks need to be identified and removed from the data stream. Idle blocks are removed one at a time from individual 66-bit blocks, and we need to trace the number of idle blocks removed over the AM period so as not to remove more than required by AMs. After the AM period, the idle removal counter will reset and start to remove idle blocks for the next AM.

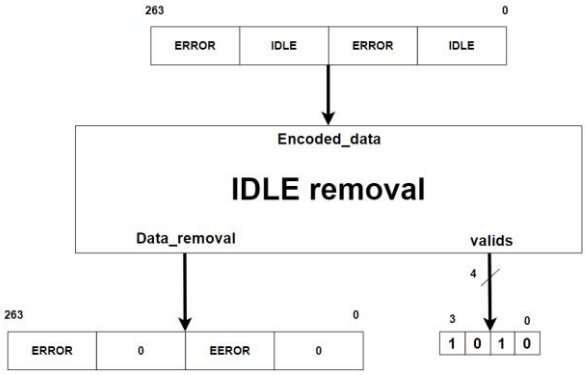



Figure 33: IDLE removal example

Valid data after removal needs to be indicated by a valid signal where each bit is related to each 66-bit idle removal data out.

### Block routing

After the idle removal process, the position of the incoming valid block is not arranged. At this stage, arrange the incoming block to be realigned to their correct sequence. The data stream is assumed to start from the LSB, consistent with the 64B/66B encoding stage as mentioned before in Encoder 64B/66B.

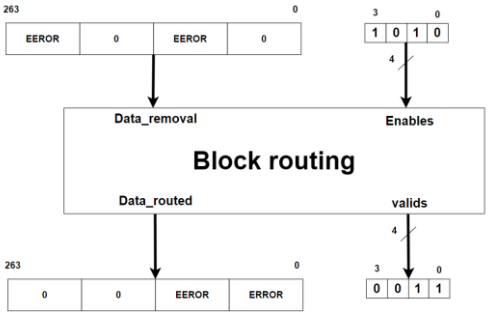



Figure 34: Block routing example

The data removal block was arranged to appear on the least significant data block with corresponding enables for each data block.

**Block distribution**

Block distribution is responsible for storing incoming available valid blocks such that it maintains their ordered transmission on the data output bus while ensuring that the entire data output bus (4 66-bit blocks) is valid to read. This can be done using FIFO for each 66-bit block output with total number of 4 for the entire data output bus.

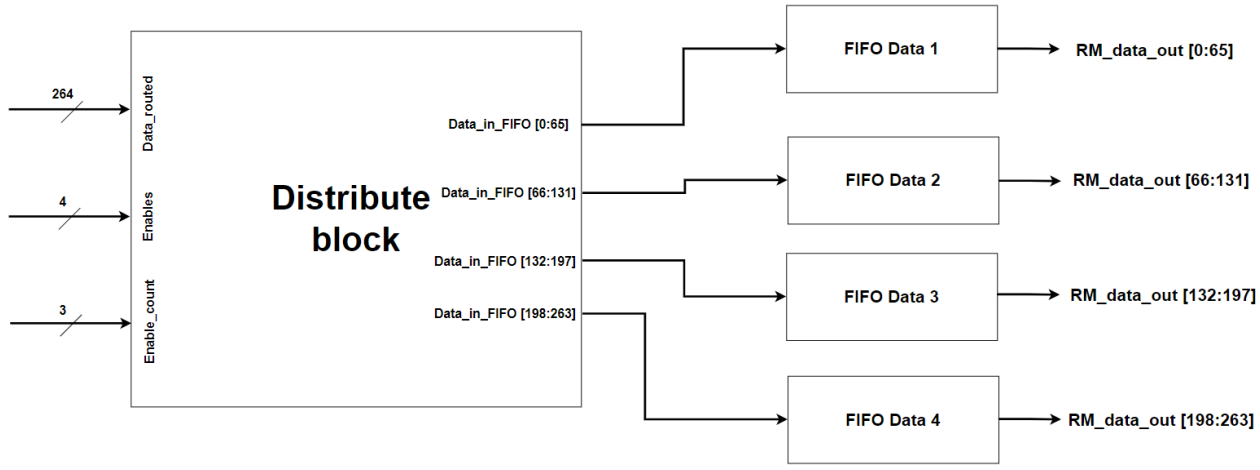



Figure 35: Block distribution architecture

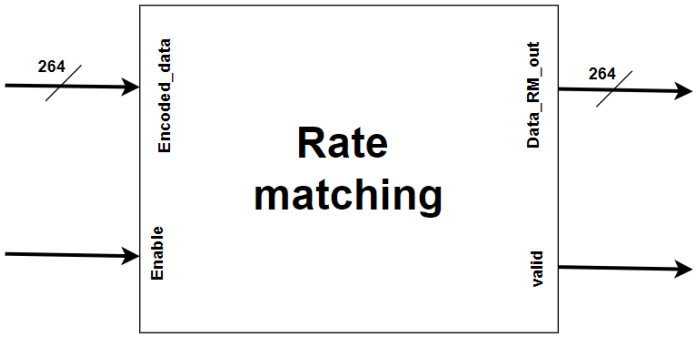



Figure 36: Rate matching signal interface

Table 14: Rate Matching Signals and Interface

| Signal name | Direction | Width | Description |
|---|---|---|---|
| Data_RM_out | Output | 264 | Data output after Rate matching action that is going to transcoder 256B/257B. |
| Enable | Input | 1 | Enable indicates that there is incoming available Encoded data. |
| Encoded_data | Input | 264 | Data incoming from Encoder 64B/66B. |
| valid | Output | 1 | It indicates that there is an available output from RM. |

## 5.4. Tx Transcoder 256B/257B

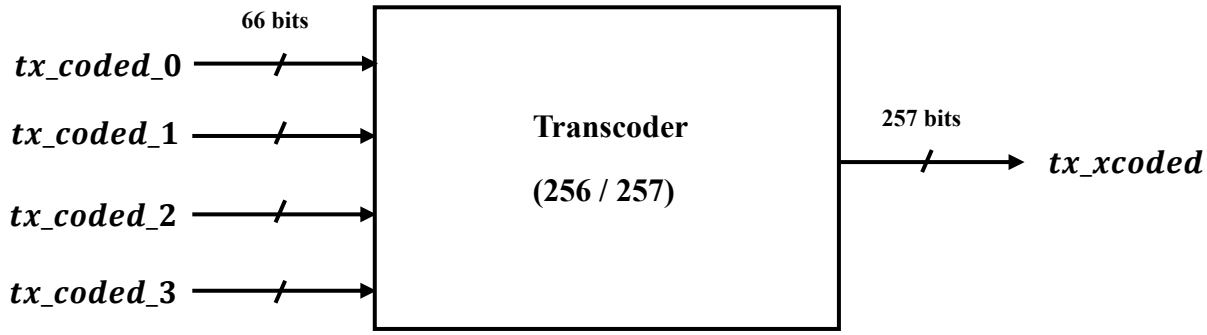
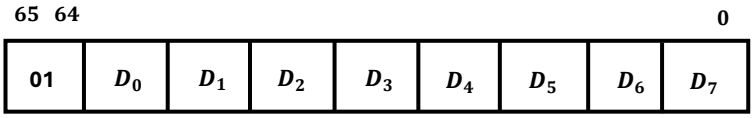



Figure 37: Block diagram of Tx transcoder

**Input block types:**

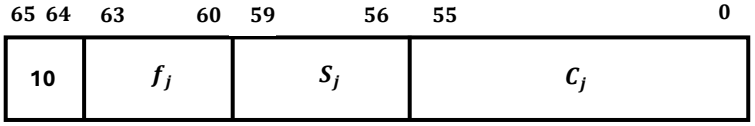
- **Valid inputs:** when synchronization header for all tx_coded_j for j = 0 to 3 is valid.

1) **Data block**: when the synchronization header for $\mathbf{tx\_coded\_j[65:64] = 2'b01}$




Data blocks consist of two synchronization header bits (2'b01) followed by 8 octet data payload (8 * 8 = 64 bits) to form the 66-bit block.

2) **Control block:** when the synchronization header for $\mathbf{tx\_coded\_j[65:64] = 2'b10}$




Control blocks consist of 4 parts as shown above: $\mathbf{c_j}$ contain control characters, ($\mathbf{f_j}$ , $\mathbf{s_j}$) represent first and second nibble of block type field followed by synchronization header bits.

- **Invalid inputs:**
  when sync header for **any** tx_coded_j for j = 0 to 3 is invalid (2'b00 or 2'b11).

- Therefore, in general we have three main cases that may occur:

### a) 4 Data Blocks (sync header is 01):

In this scenario, the most significant bit (MSB) of the output transcoded bus (tx_xcoded) must be set to **1**, representing the synchronization header. The remaining 256 bits of the bus carry the payload data from **tx_coded_j**, where $0 \leq j \leq 3$

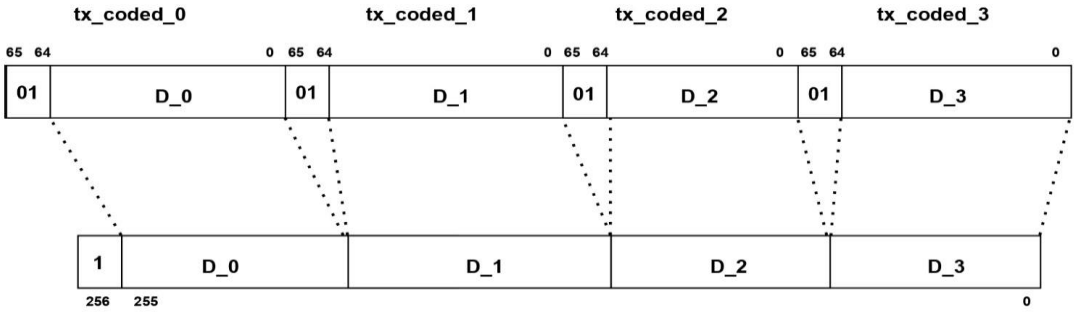



Figure 38: Four consecutive data

## b) Mixing between control blocks and data blocks:

The transcoding process begins by setting the first bit of the output to 0, indicating the synchronization header. The next 4 bits are determined based on the type of each block: a bit value of **0** is assigned for control blocks, while a value of **1** is assigned for data blocks.

When the first control block appears, the second nibble ($S_j$) of that block is intentionally omitted. This behavior is governed by a control parameter (c), which indicates the position (index) of the first control block among the four inputs tx_coded_0 to tx_coded_3. The value of (c) ranges from 0 to 3, depending on the position of the first control block as follows:

- **Case 1**: If the first control block appears at tx_coded_0 → c = 0
- **Case 2**: If the first control block appears at tx_coded_1 → c = 1
- **Case 3**: If the first control block appears at tx_coded_2 → c = 2
- **Case 4**: If the first control block appears at tx_coded_3 → c = 3

The following table presents all possible combinations of mixed data and control blocks

| | $tx\_coded\_0$ | $tx\_coded\_1$ | $tx\_coded\_2$ | $tx\_coded\_3$ |
|---|---|---|---|---|
| 1st combination | Control | Control | Control | Control |
| 2nd combination | Control | Control | Control | Data |
| 3rd combination | Control | Control | Data | Control |
| 4th combination | Control | Control | Data | Data |
| 5th combination | Control | Data | Control | Control |
| 6th combination | Control | Data | Control | Data |
| 7th combination | Control | Data | Data | Control |
| 8th combination | Control | Data | Data | Data |
| 9th combination | Data | Control | Control | Control |
| 10th combination | Data | Control | Control | Data |
| 11th combination | Data | Control | Data | Control |
| 12th combination | Data | Control | Data | Data |
| 13th combination | Data | Data | Control | Control |
| 14th combination | Data | Data | Control | Data |
| 15th combination | Data | Data | Data | Control |

First case with c = 0

Second case with c = 1

Third case with c = 2

Fourth case with c = 3

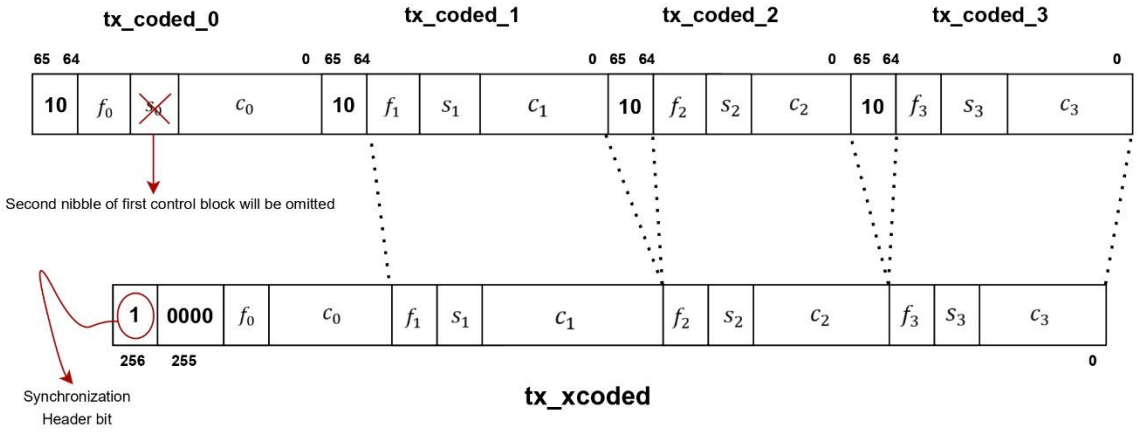



Figure 39: Four consecutive Control blocks example

### c) For invalid cases:

If the synchronization header of any block is either 2'b00 or 2'b11, the second nibble of the first block (**tx_coded_0**) regardless of whether it is a data or control block should be omitted.

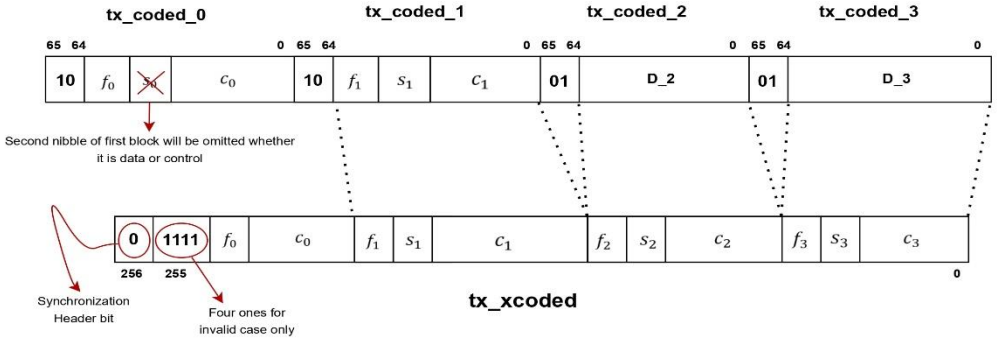



Figure 40: Invalid case example

## 5.5. Scrambler

Two primary methods are used for SSS: **serial scrambling** and **parallel scrambling**. **Serial Scrambling:** Serial scrambling processes data one bit at a time in a sequential manner (serial data stream). Each incoming input bit is XORed with specific tap values (defined by the scrambler polynomial) from the scrambler register and the scrambler state shifts each bit, producing scrambled output bit.

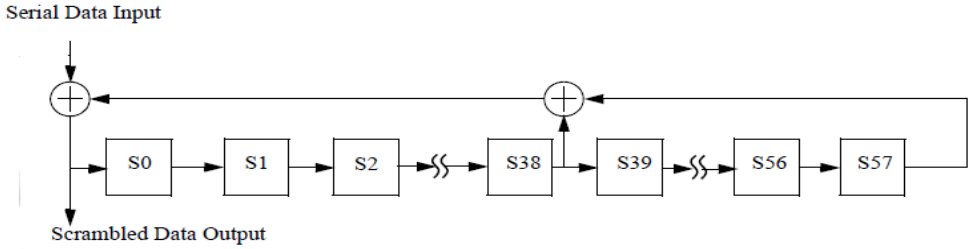



Figure 41: Serial scrambling Architecture with polynomial G(x) = 1 + $x^{39} + x^{58}$

**Parallel Scrambling:**

Parallel scrambling processes multiple bits simultaneously using a polynomial scrambler. It applies the same scrambling logic as serial scrambling but processes the entire data word in parallel. So, our target is to parallelize the structure to advance (N = 257) cycles in just **1 cycle** regardless of the number of the states of the scrambler (L). Therefore, the selected architecture is the parallel version not the serial one.

system scrambler polynomial is $\boldsymbol{G(x) = 1 + x^{39} + x^{58}}$

M L (number of registers "states")

To simplify the implementation, a demonstration version of the scrambler was developed using the polynomial $\mathbf{G(x) = 1 + x^3 + x^5}$, with a 9-bit input (N = 9, M = 3, L = 5). This reduced-complexity setup allows the core concepts to be explored and validated.

By using some manipulations of the sequence space of the scrambler, we can model the system as a MIMO system of such equations: [25].

$$\mathbf{d}_k = \mathbf{A}_p \cdot \mathbf{d}_{k-1} + \mathbf{B}_p \cdot \mathbf{b}_{k-1} \quad , d_k: \text{states of the scrambler}$$

$$\mathbf{b}'_k = \mathbf{C}_p \cdot \mathbf{d}_k + \mathbf{D}_p \cdot \mathbf{b}_k \quad , b_k: \text{input data} , b'_k: \text{scrambled data}$$

By solving the previous state equations, we can model the system by the following equations:

```
scrambled_out_0 = Current_in_0 + reg_new_2 + reg_new_4

scrambled_out_1 = Current_in_1 + reg_new_1 + reg_new_3

scrambled_out_2 = Current_in_2 + reg_new_0 + reg_new_2

scrambled_out_3 = Current_in_0 + Current_in_3 + reg_new_1 + reg_new_2 + reg_new_4

scrambled_out_4 = Current_in_1 + Current_in_4 + reg_new_0 + reg_new_1 + reg_new_3

scrambled_out_5 = Current_in_0 + Current_in_2 + Current_in_5 + reg_new_0 + reg_new_4

scrambled_out_6 = Current_in_0 + Current_in_1 + Current_in_3 + Current_in_6 + reg_new_2 + reg_new_3 + reg_new_4

scrambled_out_7 = Current_in_1 + Current_in_2 + Current_in_4 + Current_in_7 + reg_new_1 + reg_new_2 + reg_new_3

scrambled_out_8 = Current_in_2 + Current_in_3 + Current_in_5 + Current_in_8 + reg_new_0 + reg_new_1 + reg_new_2
```

Figure 42: Equations for the Scrambler output

```
reg_new_0 = Past_in_1 + Past_in_4 + reg_old_0 + reg_old_1 + reg_old_3

reg_new_1 = Past_in_0 + Past_in_2 + Past_in_5 + reg_old_0 + reg_old_4

reg_new_2 = Past_in_0 + Past_in_1 + Past_in_3 + Past_in_6 + reg_old_2 + reg_old_3 + reg_old_4

reg_new_3 = Past_in_1 + Past_in_2 + Past_in_4 + Past_in_7 + reg_old_1 + reg_old_2 + reg_old_3

reg_new_4 = Past_in_2 + Past_in_3 + Past_in_5 + Past_in_8 + reg_old_0 + reg_old_1 + reg_old_2
```

Figure 43: Equations for the internal

From the above equations, it is evident that the number of XOR operations increases with the position of the output bit as **N** grows. To address this, a **recursive substitution approach** can be applied, where the first **L** output bits are generated directly, and then fed back to compute the remaining **(N – L)** output bits. This reformulates the system for a more efficient realization.

Interestingly, a clear pattern emerges that enables a systematic and scalable implementation of a parallel scrambler. This pattern is dependent on the structure of the feedback polynomial and ensures that each output bit requires a **fixed number of XOR operations** (typically two), based on the number of taps in the scrambler polynomial. This results in a more predictable and optimized hardware design [26].

```
scrambled_out_0 = Current_in_0 + reg_new_2 + reg_new_4

scrambled_out_1 = Current_in_1 + reg_new_1 + reg_new_3

scrambled_out_2 = Current_in_2 + reg_new_0 + reg_new_2

scrambled_out_3 = Current_in_3 + scrambled_out_0 + reg_new_1

scrambled_out_4 = Current_in_4 + scrambled_out_1 + reg_new_0

scrambled_out_5 = Current_in_5 + scrambled_out_2 + scrambled_out_1

scrambled_out_6 = Current_in_6 + scrambled_out_3 + scrambled_out_2

scrambled_out_7 = Current_in_7 + scrambled_out_4 + scrambled_out_3

scrambled_out_8 = Current_in_8 + scrambled_out_5 + scrambled_out_4
```

```
reg_new_0 = Past_out_4

reg_new_1 = Past_out_5

reg_new_2 = Past_out_6

reg_new_3 = Past_out_7

reg_new_4 = Past_out_8
```

Figure 44: New Equations for the scrambler output and Internal register

The updated equations illustrate that the register values are influenced by the previous output, leading to a changing internal state over successive clock cycles even when the input remains unchanged. As a result, the scrambler's output also varies, which is undesirable in this context, since the goal is to ensure a consistent and deterministic output for a given input, independent of any external or prior state.

In other words, when the same input is applied with different initial register states, the scrambler produces different outputs—an outcome clearly shown in the figure below.

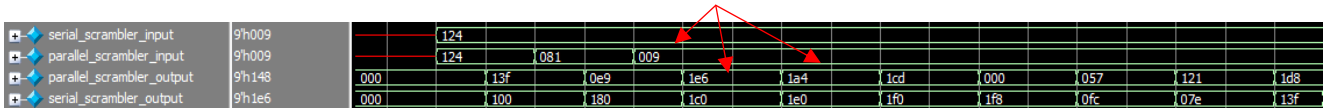



Figure 45: Scrambler output changes over time for the same input

This behavior creates a **one-to-many relationship** between the input and output, where a single input can produce multiple possible outputs depending on the internal state. To resolve this issue, we can **eliminate the feedback path from the output to the registers**. By doing so, the scrambler's output becomes fully determined by the input alone, ensuring a unique and predictable output for each input pattern without altering its intended functionality.

To simplify the design, we select an all-zero initial seed to ensure synchronization between the transmitter and receiver. With this approach, the scrambler registers are always initialized to zero, regardless of the input, allowing them to eliminate the need for internal registers.

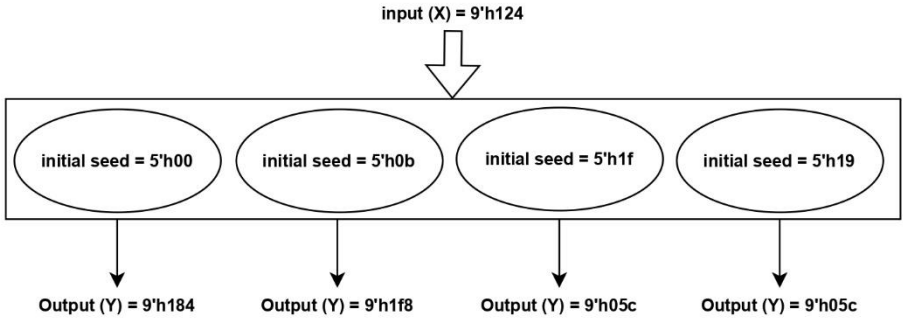



Figure 46: Self-synchronization problem in parallel scrambler

| | |
|---|---|
| $0 \leq i < M$ | $\mathbf{scram\_out[i] = scram\_in[i]}$ |
| $M \leq i < L$ | $\mathbf{scram\_out[i] = scram\_in[i] \wedge scram\_out[i - M]}$ |
| $L \leq i < N$ | $\mathbf{scram\_out[i] = scram\_in[i] \wedge scram\_out[i - M] \wedge scram\_out[i - L]}$ |

## 5.6. Alignment Marker Insertion

**Block Architecture**

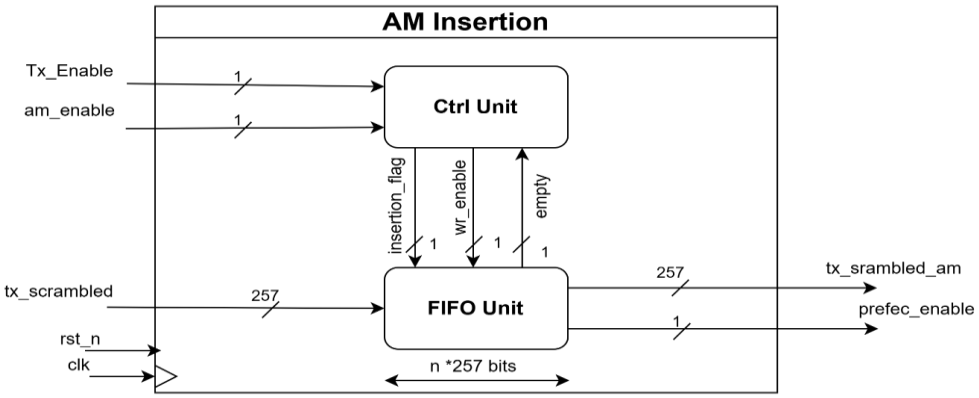



Figure 47: Block Diagram

The Architecture of the AM insertion function is displayed above showing the functions handled evenly between 2 units each focuses on some tasks as below:

**Ctrl Unit:** Tracks the insertion period & decides the FIFO operation at each cycle including the rate matching decisions and the insertion logic according to FSM will be discussed below.

**FIFO Unit:** Handles the data transfer and the insertion of the AMs.

**Ctrl Unit:** This unit is the brain of the insertion logic that follows finite state machine that defines the cases of the insertion along with the data flow utilizing the rate compensation to allow continuous data stream with deterministic block boundary insertion.

**Note:** The initial start of the block should be with the layer start Tx Enable = 1 to start the 1st CW with AM to decrease the initial locking latency at the AM lock function.

Table 15: Alignment Markers Insertion Signals and Interface

| Signal | Direction | Width | Description |
|---|---|---|---|
| Tx_Enable | Input | 1 | PCS Enable to allow Sync between the rate matching & the AM Insertion. |
| am_enable | Input | 1 | Enable signal is used to indicate if rate matching occurs leading to stream gaps with no valid input data at the current cycle. |
| tx_scambled | Input | 257 | Input Scrambled data. |
| insertion_flag | Internal | 1 | A flag raised to identify the AM insertion instant identified by a counter. |
| wr_enable | Internal | 1 | A signal to indicate if the FIFO bus will accept any new data or not. |
| tx_scrambled_am | Output | 257 | The data after inserting the AMs (either buffered data from the input stream or the AMs that are output over n clock cycles). |
| prefec_enable | Output | 1 | Valid Signal for the Pre-FEC Distribution (Should be high all the time to provide continuous stream). |
| SPEED | Parameter | - | 200G or 400G. |
| AM_PERIOD | Parameter | - | 200G: Period = 81920,<br>400G: Period = 163840. |
| PAD_WIDTH | Parameter | - | 200G: PW = 65, 400G: PW = 133 bits. |
| n | Parameter | - | The number of clock cycles to output the AM payload 200G: n = 4, 400G: n = 8. |

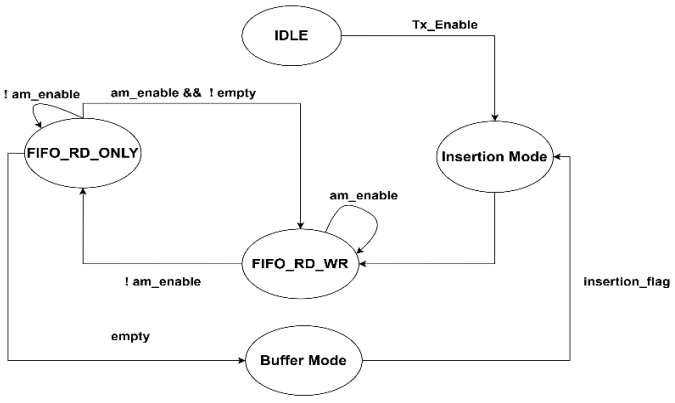



Figure 48: Control FSM

**FIFO Unit:** This Unit is a Normal FIFO that accepts data according to the Ctrl unit that tells it if a rate matching decision occurred or not.

- The FIFO is designed such that it becomes transparent when it is empty showing that it entered the “Buffer Mode”.
- The FIFO should be empty before the next insertion showing complete rate compensation & enough room for AM insertion is available without data loss.

**Insertion Mechanism**

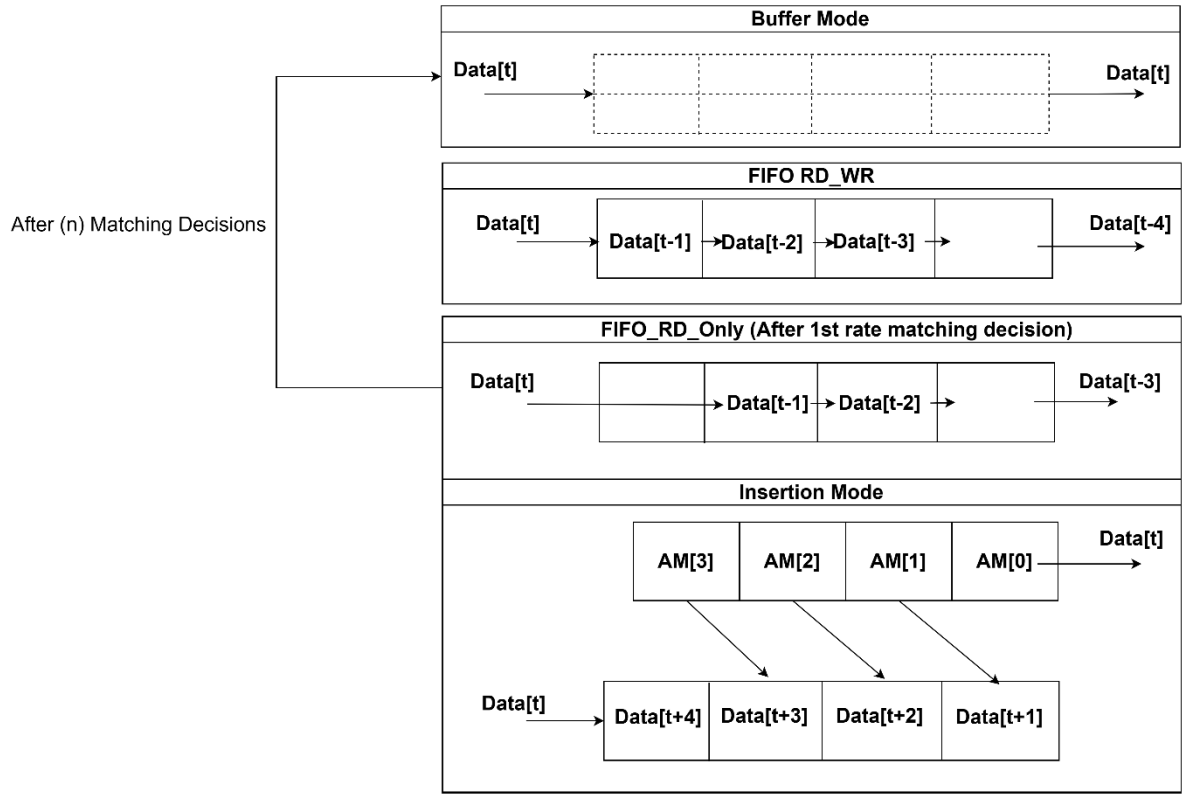



Figure 49: FIFO Scenarios during Insertion

**Note:** From the above scenarios, this architecture can use only (n) hardware cells each of 257-bit instead of $(n + 1)$ cells due to the read-after-write dependency that imposes 1-cycle delay that should be considered by extra cells to avoid any data loss.

## 5.7. Pre-FEC Distribution

**Design Choice:** In order to increase the throughput of the layer, we chose the RS codec to operate on multiple symbols rather than buffering whole codeword(s) then operate on them serially as the typical serial architecture suggested by the standard.

Hence, choosing the RS encoder to operate on multiple symbols specified by the parallel factor as a parameter **"P"** leads to add a new feature for the pre-FEC Function which is **"Data alignment & packing"** which makes it responsible for packing the input 257 bits into $(2 * P)$ bits for encoding & lane distribution.

For $P = 16$ , the pre-FEC should form 320-bit block from each 257-bit "tx_scrambled_am" before 10-bit round robin distribution.

## Block Architecture

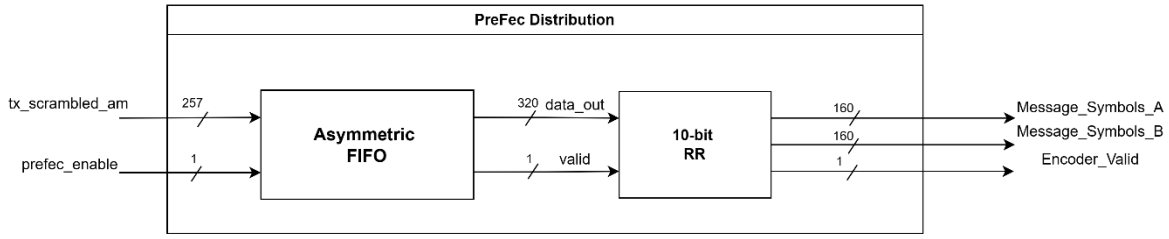



Figure 50: Block Diagram

Table 16: Pre-FEC Insertion Signals and Interface

| Signal | Direction | Width | Description |
|---|---|---|---|
| tx_scrambled_am | Input | 257 | Scrambled data after inserting the AMs. |
| prefec_enable | Input | 1 | flag from the AM Insertion showing valid input. |
| Valid | Internal | 1 | A flag for the RR function to operate on the correctly packed 320 bits. |
| Message_Symbols_(A/B) | Output | 160 | Input Symbols for the Rs Encoder. |
| Encoder_Valid | Output | 1 | Enable for the encoding process to resume encoding on the valid ready symbols from the pre-FEC. |

The pre-FEC function is achieved throughout 2 steps which will be discussed as below.

### Asymmetric FIFO

This module is responsible for the data alignment to produce the encoder input word from the transcoder 257-bit word.

The concept of asymmetric FIFOs is similar to the normal FIFO but with unequal read-to-write port ratios which require another implementation rather than the typical FIFO.

### Objectives

The module is designed for creating non-overlapping messages over the $40^{*}$ cycles but with an output rate = 320 bits/read.

Since the encoder input word = 16 symbols & the total message word/encoder = 514 symbols, the unit should reduce its output rate into 20 bits/read to complete 1 message without any overlapping with the next message.

Note: The encoder messages are a block of 10280-bit formed across 40 * 257-bit blocks.

**Model:**

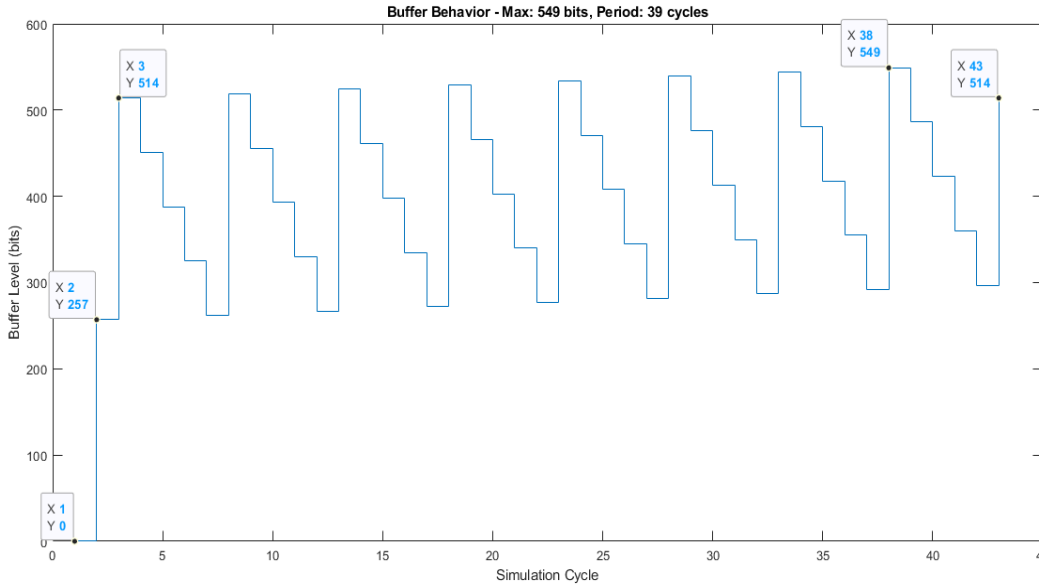



Figure 51: FIFO Read-Write Stats

For the minimum required FIFO capacity, we needed to model the read/write operations along with the rate reductions once per message which resulted in periodic read-write cycles described below.

**10-bit Round Robin:**

This unit is responsible for round robin distribution over a frame of 320 bits to allow for faster encoding process without area overhead of buffering the whole 10280-bit block as the typical function suggests.

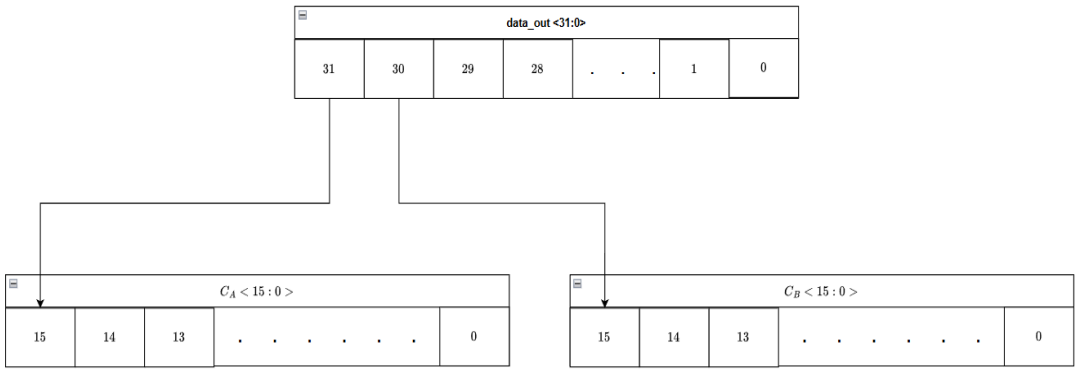



Figure 52: Symbol RR

## 5.8. RS FEC Encoder

### Block Architecture

The Encoder is chosen to operate on a parallel factor "**P**" = **16**

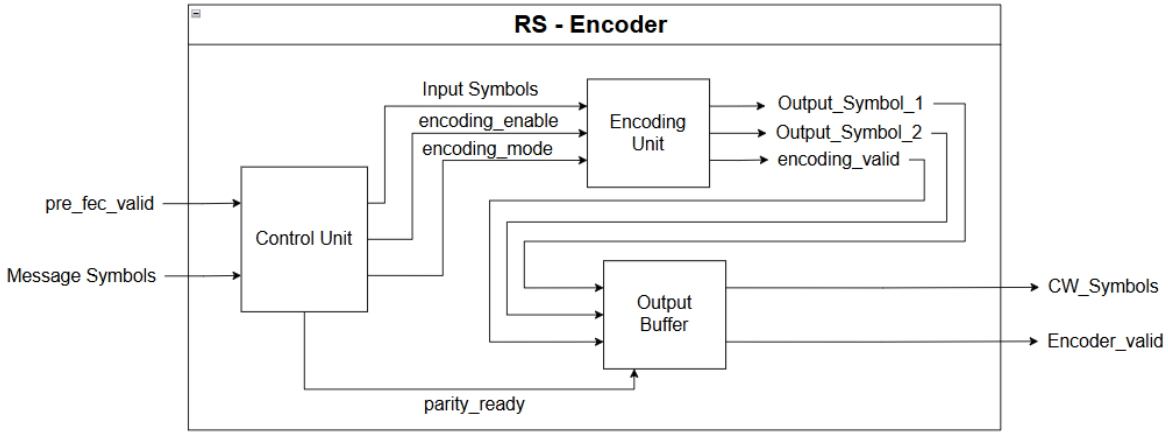



Figure 53: Block Diagram

Table 17: RS FEC Encoder Signals and Interface

| Signal | Direction | Width | Description |
|---|---|---|---|
| Prefec_valid | Input | 1 | flag from the pre-FEC block for a valid 160-bit block. |
| Message_Symbols | Input | 160 | 16 parallel symbols to be processed. |
| CW Symbols | Output | 160 | 16 parallel CW symbols. |
| Encoder Valid | Output | 1 | Valid flag for the interleaver. |
| Encoding_mode | Internal | 1 | Used to identify the state transition mode. |
| Parity_ready | Internal | 1 | Used to identify the finish of the encoding process. |

### Block Details:

**Control Unit:** Since the standard code is the non-primitive RS (544,514), the parallel symbols to be encoded should belong to the same message without overlapping.

With the help of the pre-FEC block, the 16 message symbols will be sent to the encoder for 32 cycles then only 2 symbols will be transferred completing 1 message. Hence, we need to change the state advancement from P = 16 to P = 2 at the end of the message to converge to the same encoding scheme as the serial one.

This unit is responsible for deciding the state advance factor of the encoder according to the following FSM.

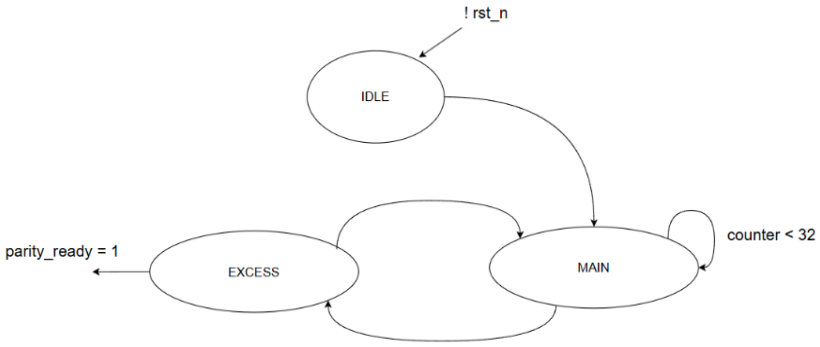



Figure 54: Encoding FSM

### Encoding Unit:

This unit operates the required matrix according to the encoding state – either the "MAIN" matrix $A^{16}$ or the "Excess" matrix $A^{2}$- for generating the parity over 33 clock cycles.

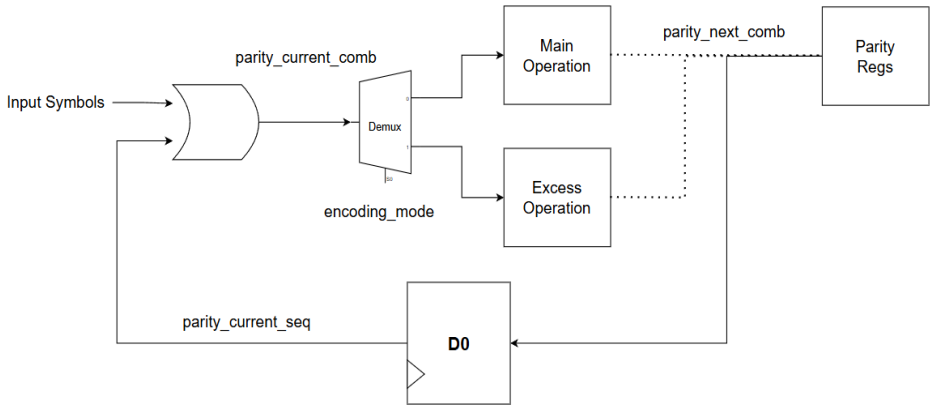



Figure 55: Encoding process

### Output Buffer:

This unit is responsible for bypassing the message symbols of the systematic codeword & inserting the parity symbols at the end of the codeword over 2 clock cycles.

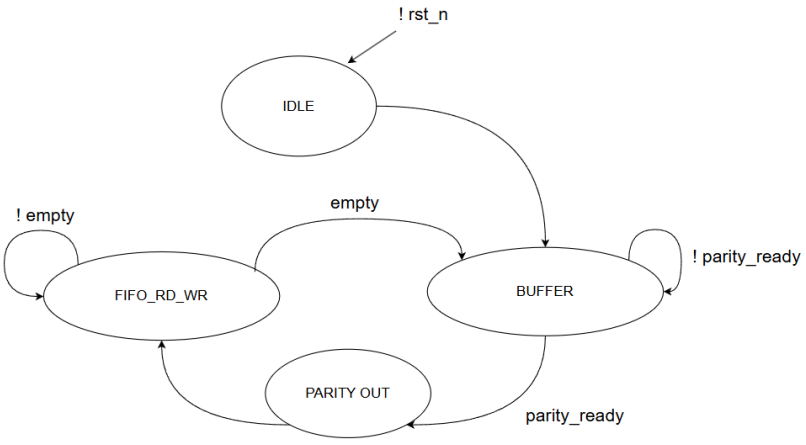



Figure 56: Output Buffer FSM

## 5.9. Interleaver

### Block level Architecture

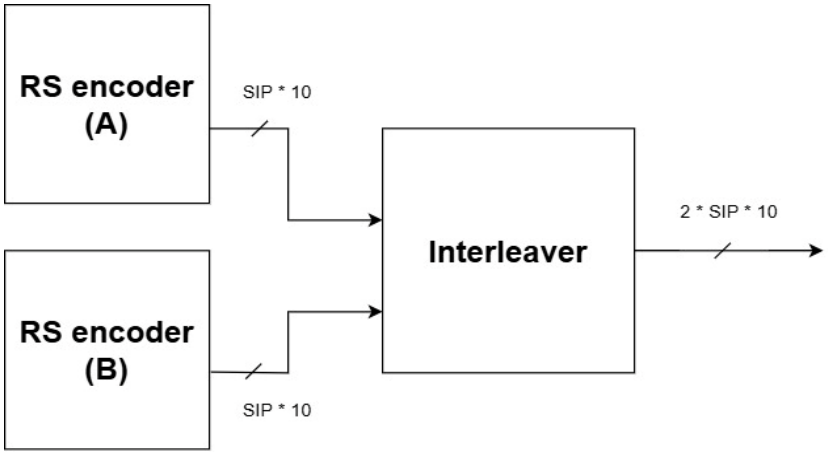



Figure 57: Block diagram of Interleaver

In 200G/400GBASE, interleaver block handles number of symbols (SIP = 16) for each RS encoder as proposed for RS encoder previously.

Here is the high-level functionality of interleaver:

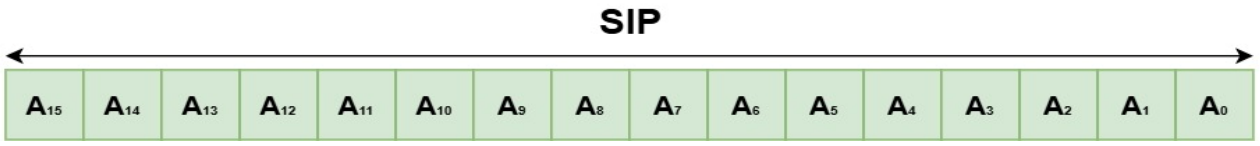



Figure 58: SIP symbols from Encoder (A)

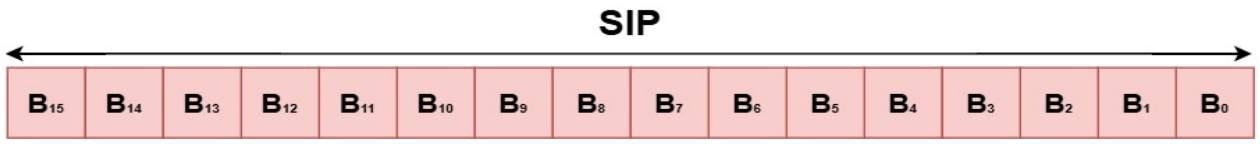



Figure 59: SIP symbols from Encoder (B)

Each encoder generates SIP output symbols, as illustrated in the figures (58,59). The interleaver forms a combined output by distributing symbols from both codewords in a **round-robin** manner. It begins by inserting symbols from Encoder A. After inserting **(PCS_LANES / 2)** symbols from each encoder, the pattern is reversed, where symbols are then interleaved starting with Encoder B. This alternating pattern is demonstrated in the figures (60,61).

[**Note that: PCS_LANES = 8 for 200G, PCS_LANES = 16 for 400G**]

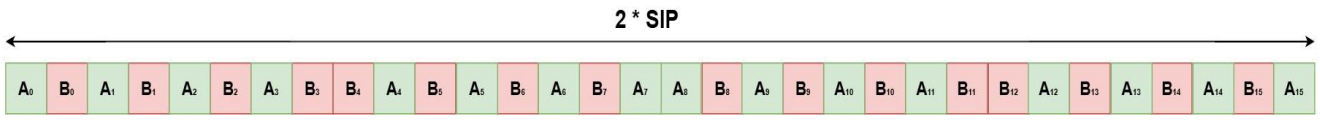



Figure 60: Interleaved data bus for 200G subsystem

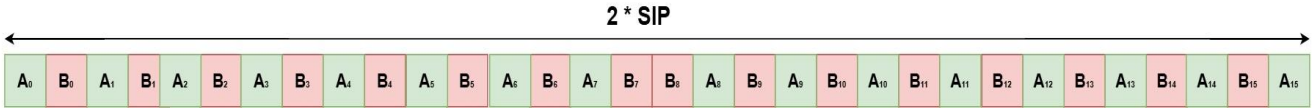



Figure 61: Interleaved data bus for 400G subsystem

Table 18: Interleaver Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| clk | input | 1 | Clock signal. |
| rst_n | input | 1 | Active-low reset signal. |
| enable | input | 1 | Enable signal to start the interleaving process. |
| enc_A | input | 160 (SIP * 10) | SIP symbols from encoder (A). |
| enc_B | input | 160 (SIP * 10) | SIP symbols from encoder (B). |
| tx_out | output | 320 (2 * SIP * 10) | Output interleaved data bus. |
| valid | output | 1 | Output valid signal asserted when the output is ready. |

## 5.10. Symbol Distribution

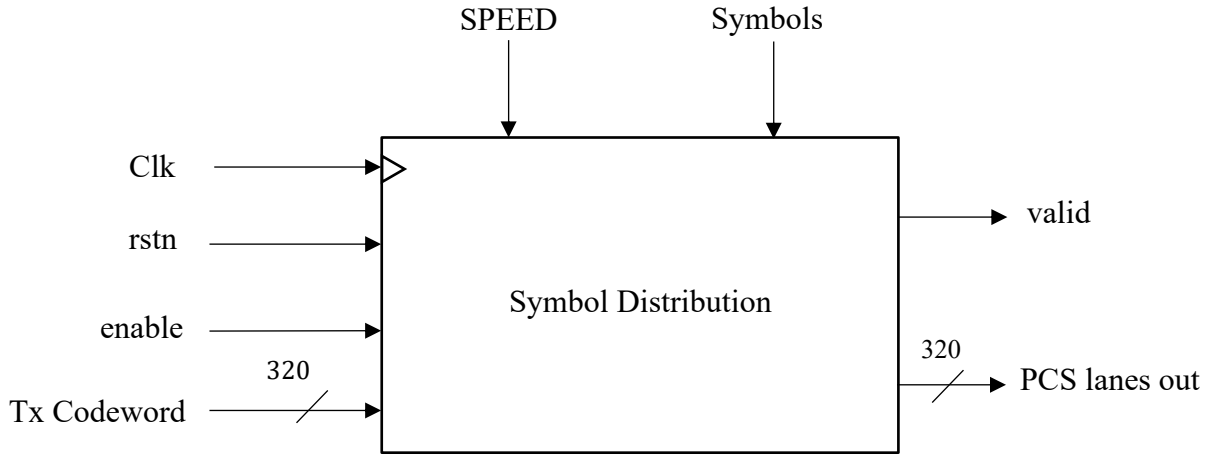



Figure 62: Symbol Distribution Block

The symbol distribution mechanism is designed based on the Symbol in Parallel (SIP) of the RS encoder as previously discussed. The process involves allocating an equal number of bits, denoted as **P**, to each PCS lane per distribution cycle. This is derived as follows:

- **N** be the total number of PCS lanes
  (N = 8 for 200 GBASE-R, N = 16 for 400 GBASE-R).
- **SIP** is the number of 10-bit symbols per RS encoder (SIP = 16).
- Since two RS encoders operate in parallel, the total number of symbols per cycle is **$2 \times SIP = 32$ Symbols,** each symbol is 10 bits wide.

The number of bits per lane (**P**) is calculated as: $\mathrm{P} = \frac{2 \times \mathrm{SIP} \times 10}{\mathrm{N}}$

- For **200 GBASE-R** ($\mathrm{N} = 8$): $\mathrm{P} = \frac{2 \times 16 \times 10}{8} = 40$ bits per lane.
- For **400 GBASE-R** ($\mathrm{N} = 16$): $\mathrm{P} = \frac{2 \times 16 \times 10}{16} = 20$ bits per lane.

This Round Robin distribution ensures balanced load across all lanes and facilitates efficient parallel transmission within the PCS architecture.

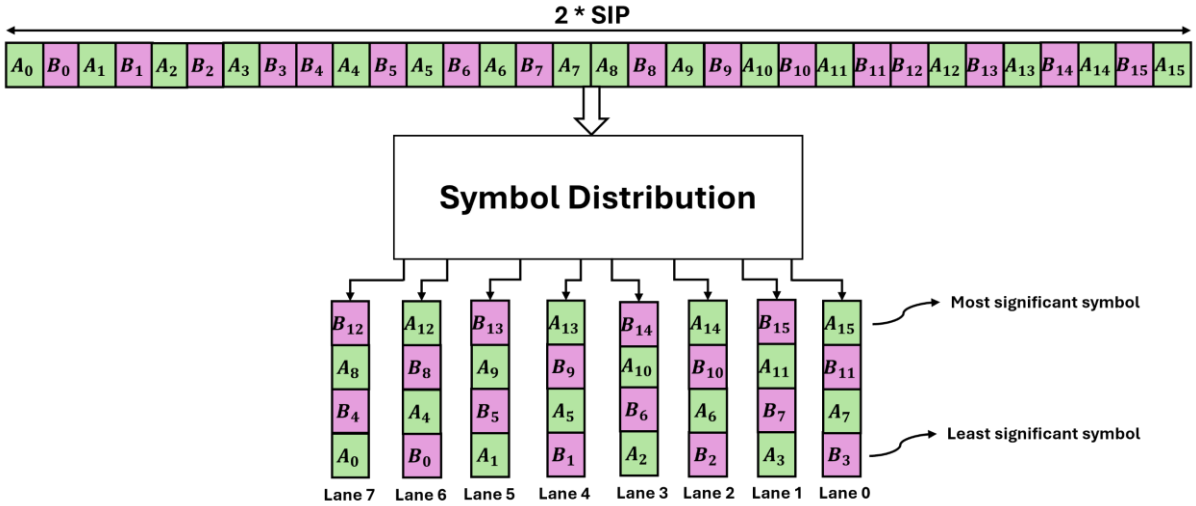



Figure 63: Symbol Round Robin Distribution

Number of symbols per lane $= \frac{\mathrm{P}}{10}$

- For **200 GBASE-R** ($\mathrm{P} = 40$): # Symbos/Lane $= \frac{40}{10} = 4$ Symbols.
- For **400 GBASE-R** ($\mathrm{P} = 20$): # Symbos/Lane $= \frac{20}{10} = 2$ Symbols.

Table 19: Symbol Distribution Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| clk | input | 1 | Clock signal. |
| rst_n | input | 1 | Synch active low reset. |
| enable | input | 1 | Enable signal will activate the block when the interleaving process finishes. |
| tx_codeword | input | 320 | Codeword after interleaving. |
| pcs_lanes_out | output | 320 | The codeword is distributed to the PCS lanes after symbol distribution. |
| valid | output | 1 | Valid Signal After Distribution Done to Allow the Distributed Symbols to the PMA Sublayer (Signal OK). |

## 5.11. Alignment Lock and Deskew

The alignment lock module is responsible for finding valid AMs, obtaining lane lock, finding the physical lane number and delaying data streams until the deskew process is done and making sure all the bypassed data lanes start from their AM.

A direct implementation would have a very long critical path, so a pipelined architecture was implemented where each function is done in a separate module as shown in figure (64), this led to an 8x increase in setup slack. The added latency won't be a problem, as it's only an initial delay due to the continuous transmission nature of PMA sublayer. This allowed the system to operate at a much higher frequency as the timing bottleneck in this module was solved.

The major bottleneck was in the calculation of the physical lane number of an AM as each AM had a different unique portion this would require n comparisons (where n is the number of lanes) and each comparison is logic intense, this was handled by pipelining the comparisons so that each comparison is done on multiple cycles instead of one utilizing the fact that the time required for these results to be ready is an AM period which is much larger than the latency introduce by pipelining.

The deskew module is much simpler than the lane lock module, it checks that all the physical lane numbers of all the lanes are unique and checks that the skew between the lanes is below the maximum limit. A direct implementation yielded good results and implementation options for optimizing power and area were carried out to find the most optimum results.

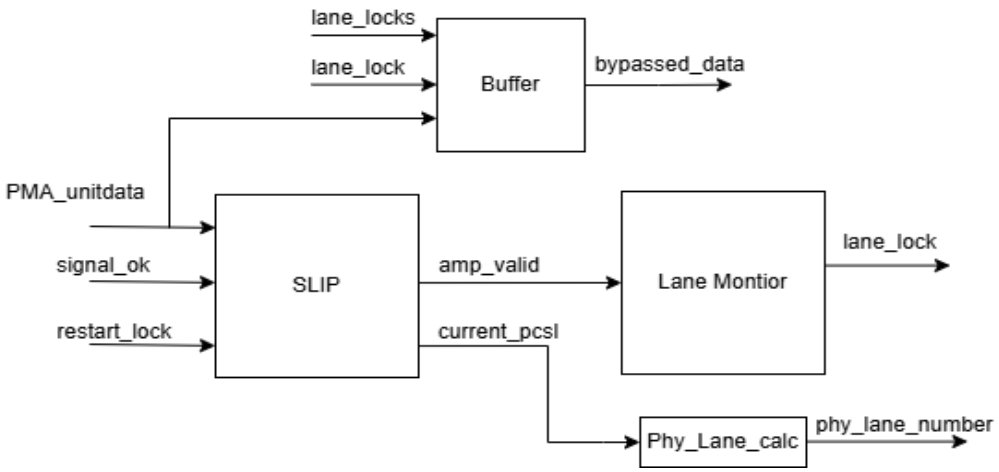



Figure 64: Alignment lock architecture

## 5.12. Lane Reorder

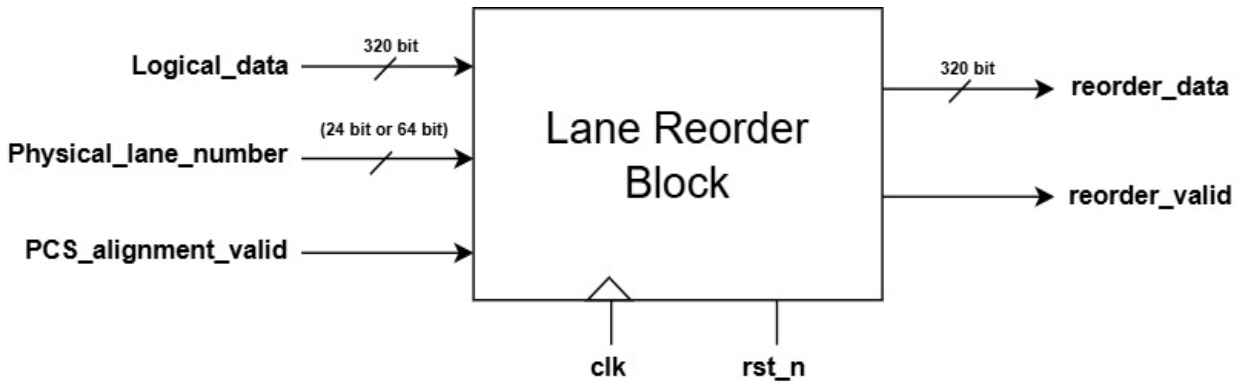



Figure 65: Block diagram of lane reorder

Table 20: Lane Reorder Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| clk | input | 1 | Clock signal. |
| rst_n | input | 1 | Active-low reset signal. |
| pcs_alignment_valid | input | 1 | Enable signal to start the lane reordering process. |
| logical_data | input | 320 | Input logical data bus. |
| physical_lane_number | output | 24/64 | Data bus that contains the physical lane number that corresponds to each portion within logical input data bus. |
| reorder_data | output | 320 | Output reorder data bus. |
| reorder_valid | output | 1 | Output valid signal asserted when the output is ready. |

**Block level Architecture:**

**Logical data** bus consists of (8 * 40 bits) portions of data in case of 200G subsystem or consist of (16 * 20 bits) portions of data in case of 400G subsystem.

**physical lane number** bus which may be 24-bit bus (8 * 3 bits) in case of 200G subsystem or may be 64-bit bus (16 * 4 bits) in case of 400G subsystem, where each portion (3 or 4 bits) of the physical lane number bus must be **unique**.

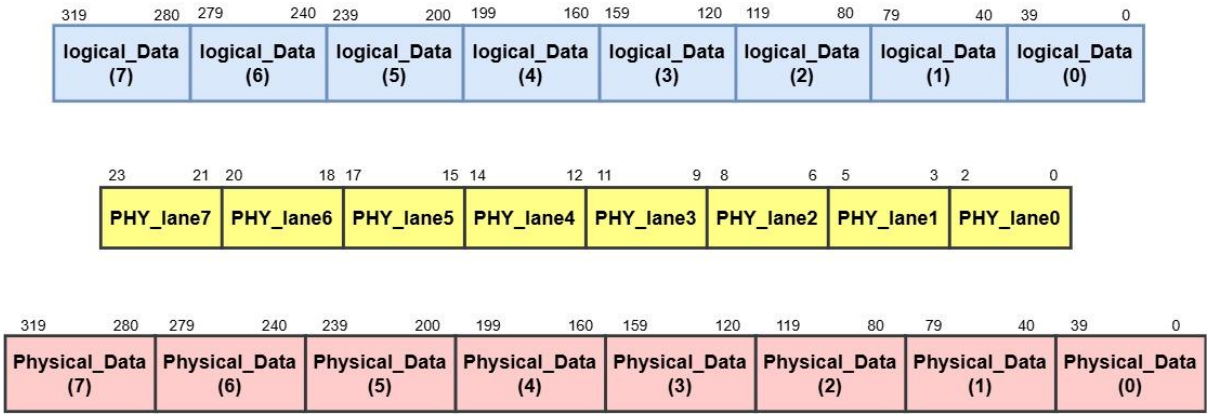



Figure 66: Portions of the data [in case of 200G subsystem]

- **Step 1: Logical-to-Physical Lane Mapping**

Each logical lane carries a segment of the overall data, and the associated physical lane number determines its correct placement in the physical layer. This mapping process restores the original serialized data sequence.

The below example illustrates the process in case of 200G subsystem:

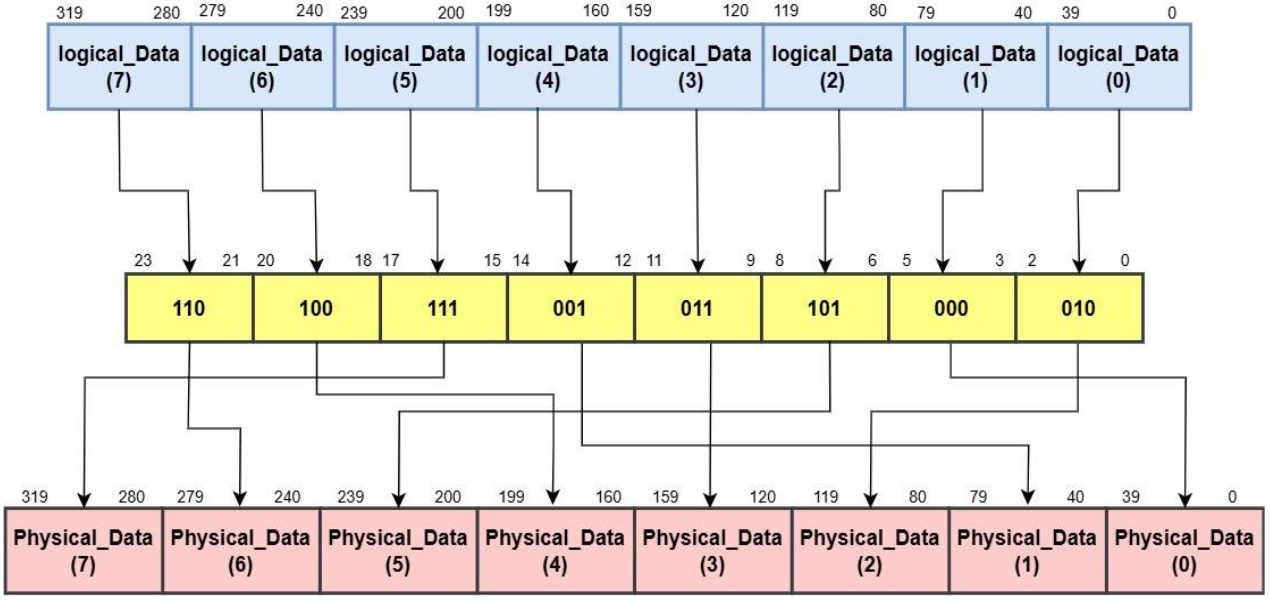



Figure 67: Lane reordering process example for 200G subsystem

- **Step 2: Reverse Symbol Distribution**

In Ethernet PCS, the transmitter distributes symbols across multiple lanes. To recover the original data stream at the receiver, this distribution must be reversed before deinterleaving can occur.

In this step, we undo the transmitter's symbol mapping by taking the physical data bus—produced by the earlier Logical-to-Physical Lane Mapping—and applying a **round-robin** redistribution. This process reconstructs the reordered data bus, which is then forwarded to the **deinterleave** block.

## 5.13. Deinterleave

**Block level Architecture**

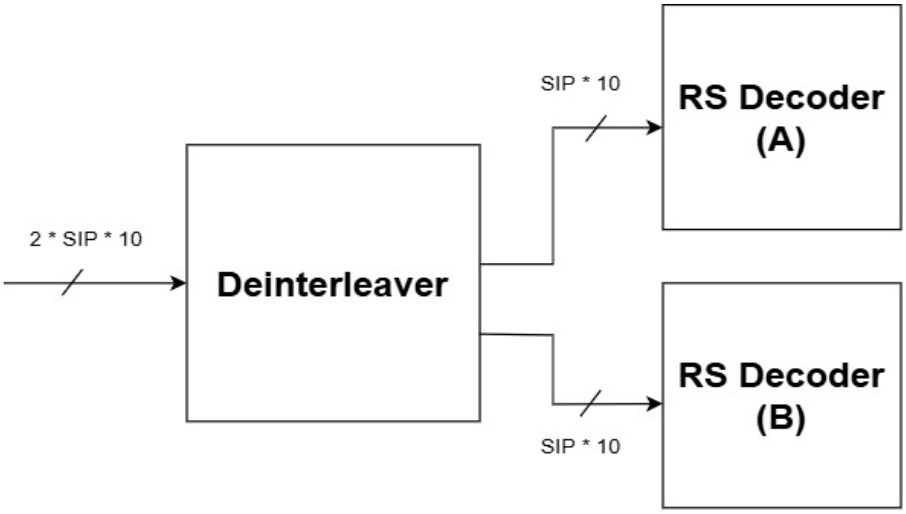



Figure 68: Block diagram of Deinterleave

The de-interleaver receives its input from the lane reorder block and begins distributing the symbols to two decoders, A and B. It follows a **round-robin (RR)** pattern, starting with Decoder A, then B. This continues until **8 symbols** (for a 200G subsystem), or **16 symbols** (for a 400G subsystem) have been assigned from each decoder. After that, the distribution order is reversed: symbols are interleaved starting with Decoder B, then A. This alternating RR process continues until all incoming symbols have been assigned specifically, until **2×SIP** symbols have been distributed across both decoders.

Here is the high-level functionality of Deinterleave:
The incoming symbols "$2 * SIP$" is demonstrated in the figure below:

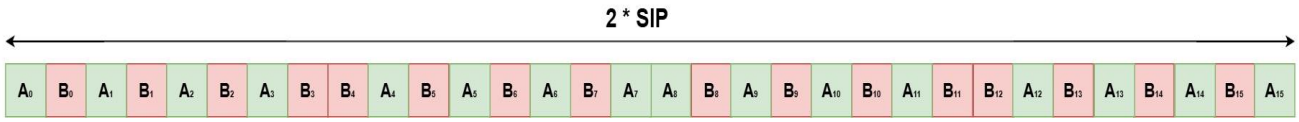



Figure 69: Input data to be Deinterleaved [200G subsystem]

The deinterleave outputs are illustrated in the figure below:

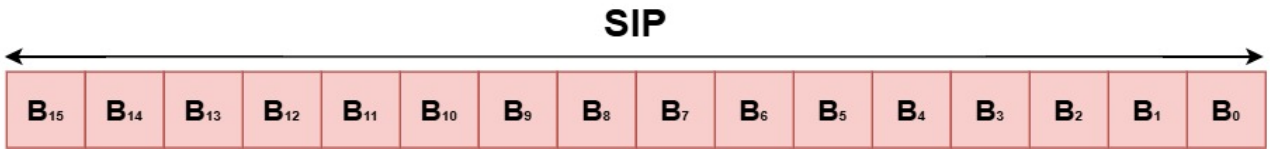



Figure 70: SIP symbols for Decoder (B)

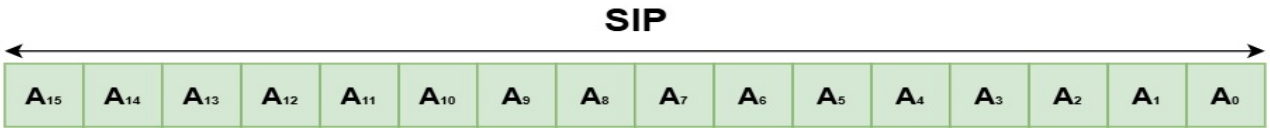



Figure 71: SIP symbols for Decoder (A)

Table 21: Deinterleave Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| **clk** | input | 1 | Clock signal. |
| **rst_n** | input | 1 | Active-low reset signal. |
| **enable** | input | 1 | Enable signal to start the Deinterleaving process. |
| **in_stream** | input | 320 (2 * SIP * 10) | Input data bus to be deinterleaved. |
| **dec_A** | output | 160 (SIP * 10) | SIP symbols for decoder (A). |
| **dec_B** | output | 160 (SIP * 10) | SIP symbols for decoder (B). |
| **valid** | output | 1 | Output valid signal asserted when the output is ready. |

## 5.14. RS FEC Decoder

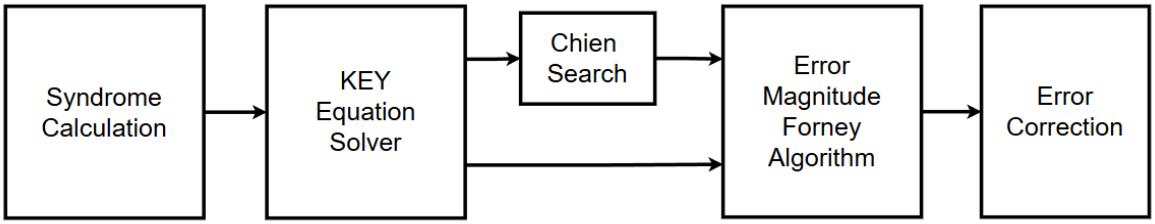



Figure 72: RS decoder Block diagram

### 5.14.1. Syndrome Calculation

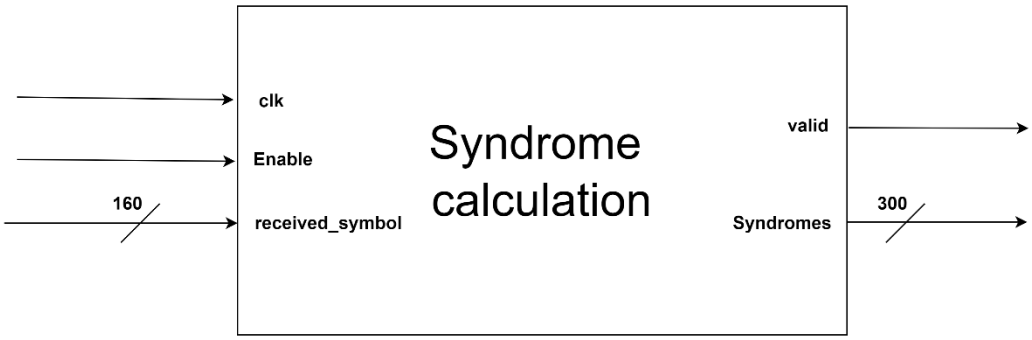



Figure 73: input-output signals of syndrome

Syndrome calculation is implemented in one module "syndrome_unit. sv; this module computes syndrome symbols using the assign statement that represents the optimized matrix generated by the IMA optimization technique.

Table 22: Syndrome Calculation Signals and Interface

| Signal name | direction | width | Description |
|---|---|---|---|
| clk | Input | 1 | Clock signal |
| Enable | Input | 1 | Active-high enable from previous block "lane reorder" block. Starts computation when asserted. |
| received_symbols | Input | 160 | 16 received symbols (each 10-bit) of the codeword. |
| valid | output | 1 | Active-high pulse indicating valid output syndromes. |
| syndromes | output | 300 | 30 syndrome symbols (10-bit each):<br>$s_1 = syndromes[9:0]$<br>...<br>$s_{30} = syndromes\ [299:290]$ |

**Block diagrams**

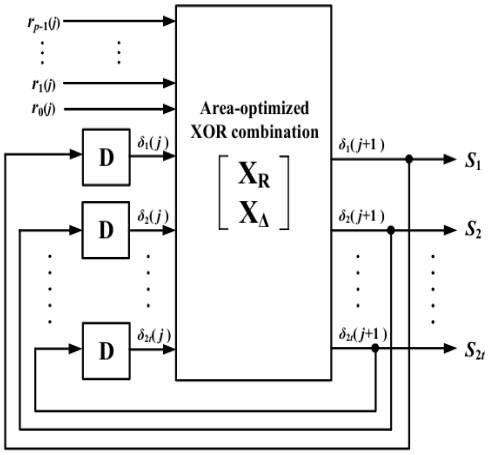



Figure 74: optimized syndrome architecture

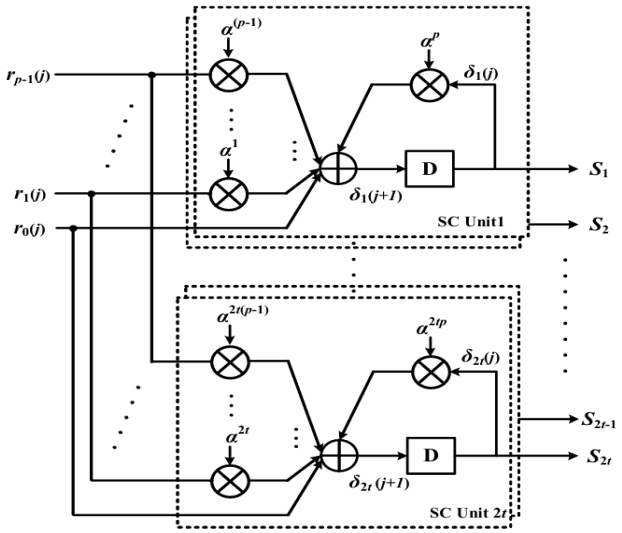



Figure 75: conventional syndrome architecture

## Functional description:

When enable signal is high the following computation:

- Increment counter, Counting is enabled when 'enable' is high, continuing for precisely 34 clock cycles "n/p = 34" before resetting; hence, a valid signal will be asserted high, and the syndrome symbol will be ready for the next stage of the decoder, and the counter will reset to "0" again.
- While counting, intermediate syndrome values calculated using the assign statement generated by the optimization algorithm, when the counter is "n/p = 34" the final value of the syndrome symbol, will be ready.

### 5.14.2. Key Equation Solver

The original Berlekamp Massey algorithm is explained through the flow chart in figure (76), the input to the algorithm is the syndromes computed by SC block. Initially the λ which is the error locator polynomial and B which is the polynomial used to update λ are initialized with 1. Counters as r and k are initialized with 0 and the scaling factor γ is also initialized with 0.

The algorithm iterates for a total of 2*t iterations where t is the error correction capability of the code. The algorithm is based on Massey's LFSR synthesis paper [18] where he was able to prove that synthesizing the shortest LFSR that matches the given set of syndromes will yield the lowest degree error locator polynomial when the number of errors is less than the code's correction capability otherwise the generated polynomial is not guaranteed to be it.

At the start of each iteration the discrepancy, which is a measure of how far the current polynomial is from representing the syndromes, is calculated. If the discrepancy is zero then the polynomials are only shifted otherwise, they are updated with the value calculated in B using the discrepancy and the scaling factor. Updating polynomial B in this algorithm requires the use of GF inverter replacing it with a multiplication allows for operation at higher frequencies due to the lower critical path.

At the last iteration, the algorithm outputs both the error locator and error evaluator polynomials, this means that the computation of both polynomials is done simultaneously allowing for a great reduction in latency.

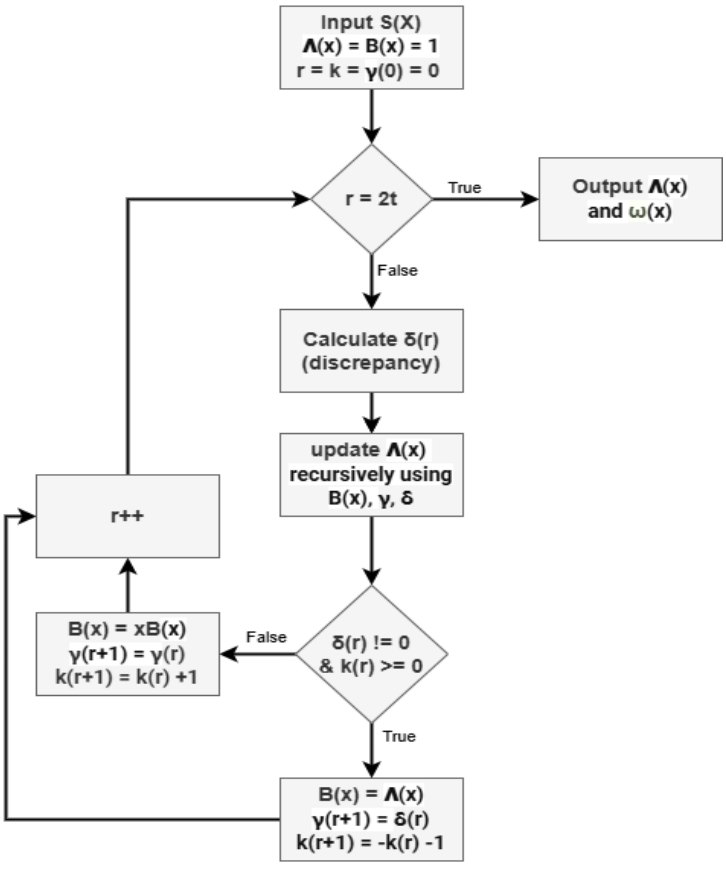



Figure 76: The original BMA flow chart

The reformulated inversion less BMA (RIBMA) [23] is a modification of the original BMA that replaces GF inverter with multiplication decreasing the critical path delay to its lowest possible value which is the delay of an XOR gate and a GF multiplier, in addition the architecture of the module is homogeneous which makes the implementation and timing closure easier.

The RIBMA finds the scaled polynomials of error locator and error evaluator polynomials instead of the original polynomials which allow the division to be replaced by multiplication, since the roots of the error locator polynomials is that of concern the scaling doesn't affect the Chien's search operation. The error evaluator polynomial which is used to evaluate the error magnitudes is affected by this scaling which is accounted for at error evaluation block.

The architecture of the KES based on RIBMA has two main blocks which are the control unit, and the processing element (PE) also called discrepancy block and is shown in figure (77). The control unit shown in figure (78) is responsible for updating the counter r and k, generates MC(r) signals depending on the calculated discrepancy which determines if the polynomials should be updated or not and updates the value of the scaling factor γ.

The PE shown in figure (79) is responsible for calculating the discrepancy and updating the polynomials depending on MC(r) value generated by the control unit. From the figure it's obvious that the critical path is an adder and a multiplier. The GF multiplier can be pipelined to decrease the critical path's delay even more. It wasn't needed in this case as the existing critical path delay was acceptable and achieved the required specifications which will be shown later in the results section.

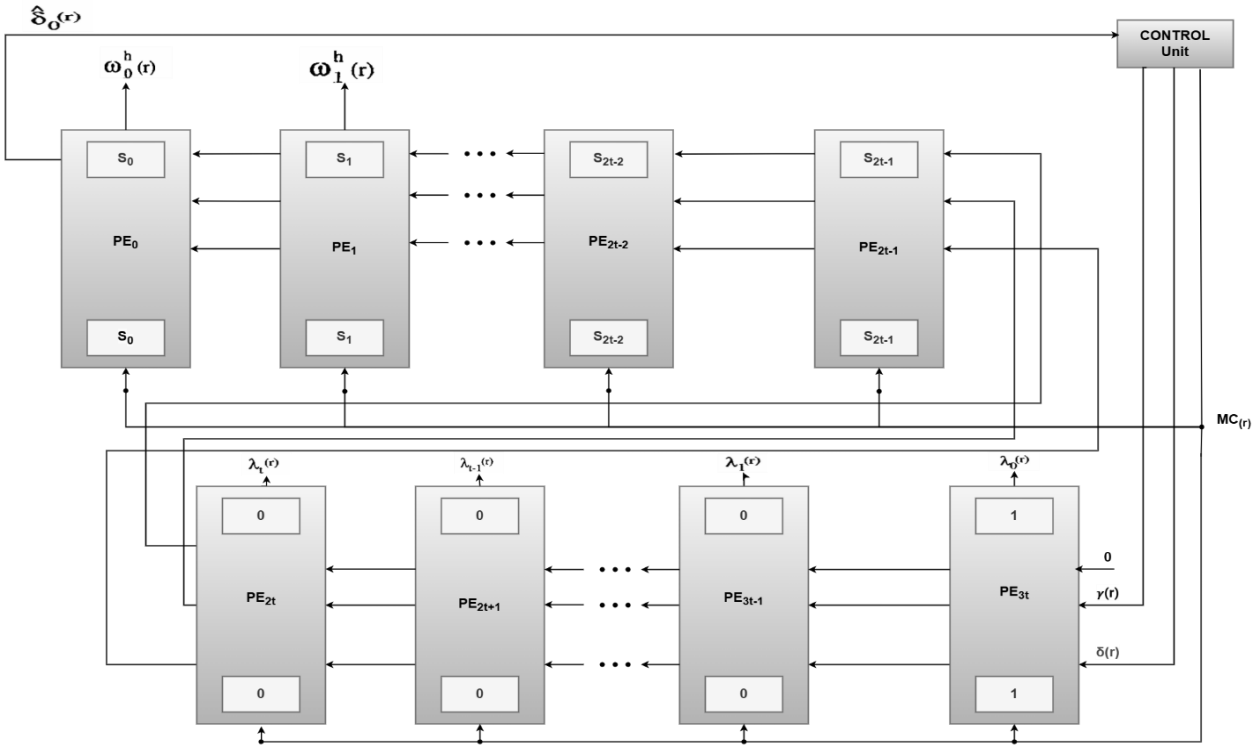
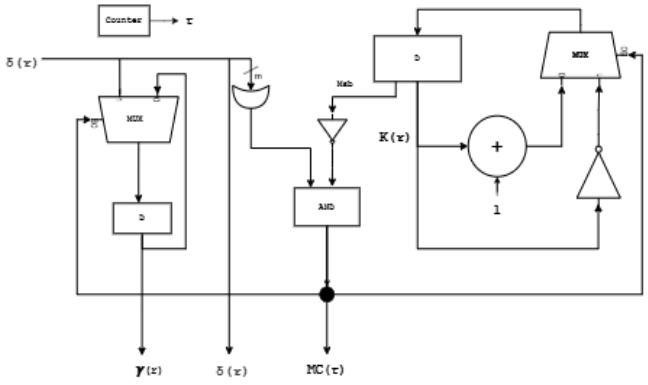



Figure 77: The architecture of the KES using RIBMA



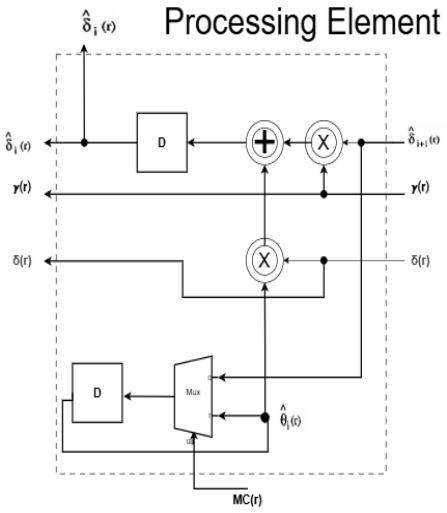




Figure 78: The control unit of the KES using RIBMA

Figure 79: The Processing element of KES using RIBMA

### 5.14.3. Chien Search

#### System Architecture

A partially parallel architecture is adopted with (**P = 17**) evaluation per cycle. So, the latency taken by this parallel architecture is reduced from (**n = 544**) clock cycles to (**n/p = 32**) [27].

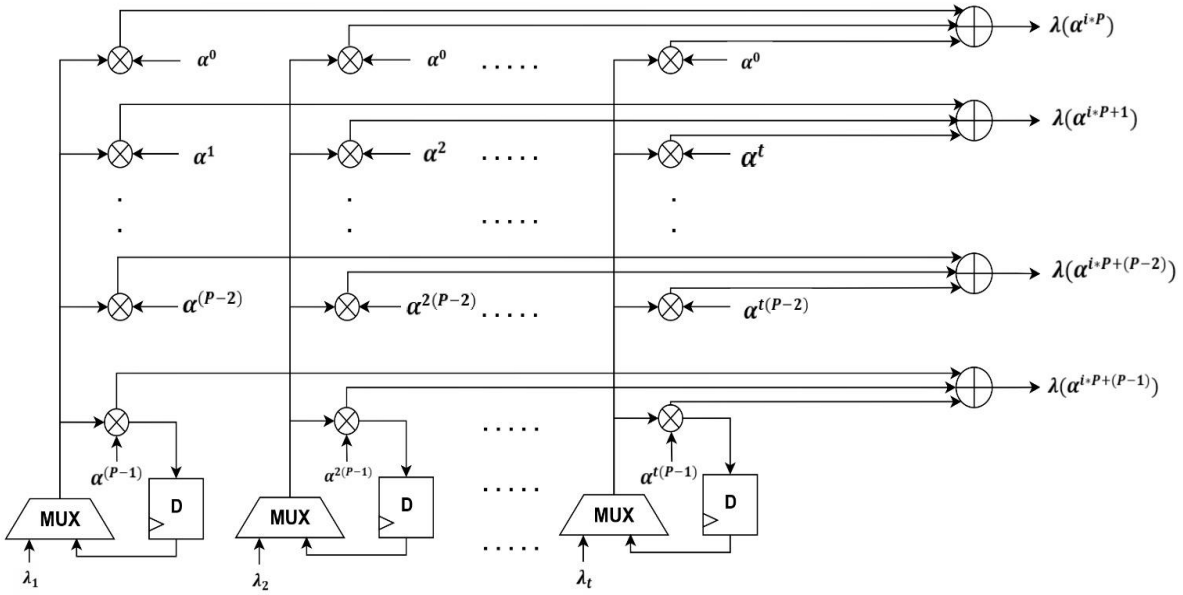



Figure 80: Parallel Architecture of Chien Search

All the multiplexers select coefficients of error locator polynomial in the first clock cycle, then select the registered data afterwards which result from the last (**t = 15**) Galois field (GF) multipliers.

Outputs of MUXs pass through constant GF multipliers and the constant term is ($\alpha^{u*j}$). For simplicity (**u**) represents the row index and it ranges from **zero** to (**P-1**), while (**j**) represents the column index and it ranges from **zero** to (**t**).

Therefore, the above parallel architecture consists of:

1. (**P*t = 255**) constant GF multipliers.
2. (**t = 15**) Multiplexers.
3. (**P = 17**) finite field adders [XORs].

After substituting all the possible combinations ($\alpha^{i*P+u}$) into locator polynomial, the error is located at the index ($i * P + u$) if and only if $\boldsymbol{\lambda(\alpha^{i*P+u}) = 0}$

The top-module of Chien Search architecture controls the pipeline and coordinates evaluation over (n = 544) codeword positions in 32 iterations (with P = 17 evaluations per cycle). For each clock cycle, 17 powers of the primitive element α are evaluated in parallel.

To break down the critical path we pipeline Combinational logic with additional 2 clock cycles beside 32 clock cycles for the main searching operation. Therefore, Chien search block consume total 34 (32+2) clock cycles from **KES_valid** assertion till producing output error locations and the equivalent root for each error location → $\textbf{Error Root} = \boldsymbol{\alpha^{position}}$

Table 23: KES Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| clk | Input | 1 | Clock signal. |
| rst_n | Input | 1 | Active-low reset signal. |
| KES_valid | Input | 1 | Enable signal from KES block to start the Chien search process. |
| Lamda_degree | Input | 1 | Indicates the degree of locator polynomial. |
| error_locator_polynomial | Input | 160 | Bus that contains the coefficients of error locator polynomial from $\lambda_0$ to $\lambda_t$. |
| error_positions | Output | 150 | Bus that contains all the error locations each location in 10 bits. |
| chien_roots | Output | 150 | Bus that contains all the error roots correspond to each location. |
| CW_finish | Output | 1 | Flag indicates the end of Chien Search process. |
| CW_bad | Output | 1 | Flag indicates that the codeword is uncorrectable as number of errors found not equal degree of locator polynomial (**Lamda_degree**). |
| forney_enables | Output | 15 | Enables for Forney Units as Forney block use the concept of hardware resource sharing. So, Forney activates each unit that corresponds to each valid location. |

### 5.14.4. Forney Algorithm

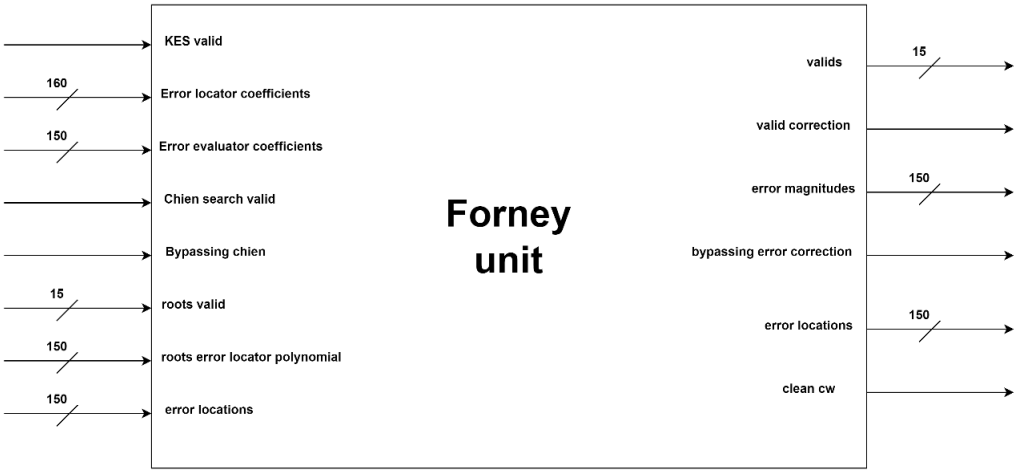



Figure 81: Forney signals interface

Table 24: Forney Signals and Interface

| Signal name | Direction | Width | Description |
|---|---|---|---|
| KES valid | Input | 1 | Indicates whether there is an available error locator and evaluator polynomial coefficient or not. |
| Error locator coefficient | Input | 160 | Represents error locator polynomial coefficients where each 10-bit represents a coefficient and can be specified as: $\text{error locator coeffi}[9:0] = \lambda_0$ $\text{error locator coeffi}[19:10] = \lambda_1$ ... $\text{error locator coeffi}[159:150] = \lambda_{15}$ |
| Chien Search valid | Input | 1 | Indicates that Chien search finished and there are available error locator polynomial roots. |
| Error evaluator coefficient | Input | 150 | Represents error evaluator polynomial coefficients where each 10-bit represents a coefficient and can be specified as: $\text{error evaluator coeffi}[9:0] = \Omega_0$ $\text{error evaluator coeffi}[19:10] = \Omega_1$ ... $\text{error evaluator coeffi}[149:140] = \Omega_{14}$ |
| Bypassing Chien | Input | 1 | When errors in received codeword exceed maximum limit can be corrected, so that this signal indicates that Reed Solomon decoder bypass received codeword without correction. |
| Roots valid Chien | Input | 15 | This signal specifies which 10-bit root is valid to read so for example: $\text{if valid}[0] = 1 \text{ means root error locator}[9:0]$ |
| Roots Chien | Input | 150 | Represents error locator polynomial roots where each root is 10-bit. |

| | | | |
|---|---|---|---|
| Error locations | Input | 150 | Represent error location in received codeword and each location is 10-bit as code word size is 544. |
| valids | Output | 15 | Indicates which error magnitude is valid to read where each bit is corresponding to a single error magnitude value for example: $\text{valids}[0] = 1 \rightarrow \text{error\_mag}[9:0]$ is valid |
| Valid correction | Output | 1 | Indicates that there is error magnitude that can be used to correct received codeword. |
| Error magnitudes | Output | 150 | Error magnitudes values where each error magnitude is 10-bit values. |
| Bypassing error correction | Output | 1 | Indicates that received codeword exceed maximum error correction limit so that received codeword will be bypassed. |
| Clean CW | Output | 1 | Indicates that no error occurred in received codeword. |

**Implementation**

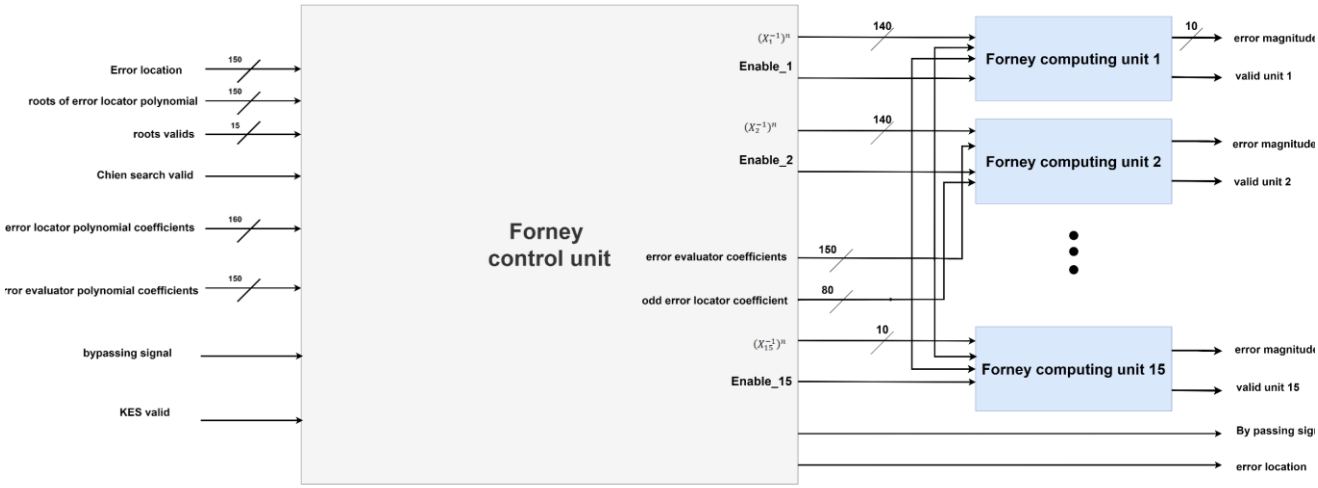



Figure 82: Forney implementation architecture

**Forney control unit**

The Forney control unit decides whether to compute error magnitude or not based on the incoming control signal. There are three cases as follows:

- If the bypassing signal is high, the Forney control unit bypasses this signal to error correction such that error correction will simply bypass the received codeword.
- If the bypassing signal is low and no roots are found by Chien search, this means that no error occurred in the received codeword, so the Forney control unit will raise a clean CW for correction to inform it that there is no need for error correction; just remove the parity symbol to retrieve the original message from the received codeword.

- If the bypassing signal is low and there is any single bit that is high within the valid signal coming from Chien, it means that there is an error that can be corrected. In this case, the Forney control unit will start to calculate for each single incoming root its all-possible power to enable computing substitution of roots in error locator and evaluator polynomials.

The derivation process of error locator polynomials is simply to select the odd power polynomial coefficients as mentioned before. For the Forney control unit, select the number of Forney computing units to be activated relevant to the number of available incoming roots. If the bypass signal is active, error magnitudes are ignored, and the received codeword is passed uncorrected.

**Forney computing unit optimization**

Implementation for this substitution computation can be directly implemented by large number of GF multipliers and adders, while the proposed architecture is hardware re-use. This approach will use just two multiplier-adder accumulate so that the area will be significantly reduced and the tradeoff for this approach is extra latency which will not be big issue as Forney operate after Chien search which require 34 clock cycle to found next error locator polynomial roots, thus there is no big advantage for Forney to complete this stage in one clock cycle.

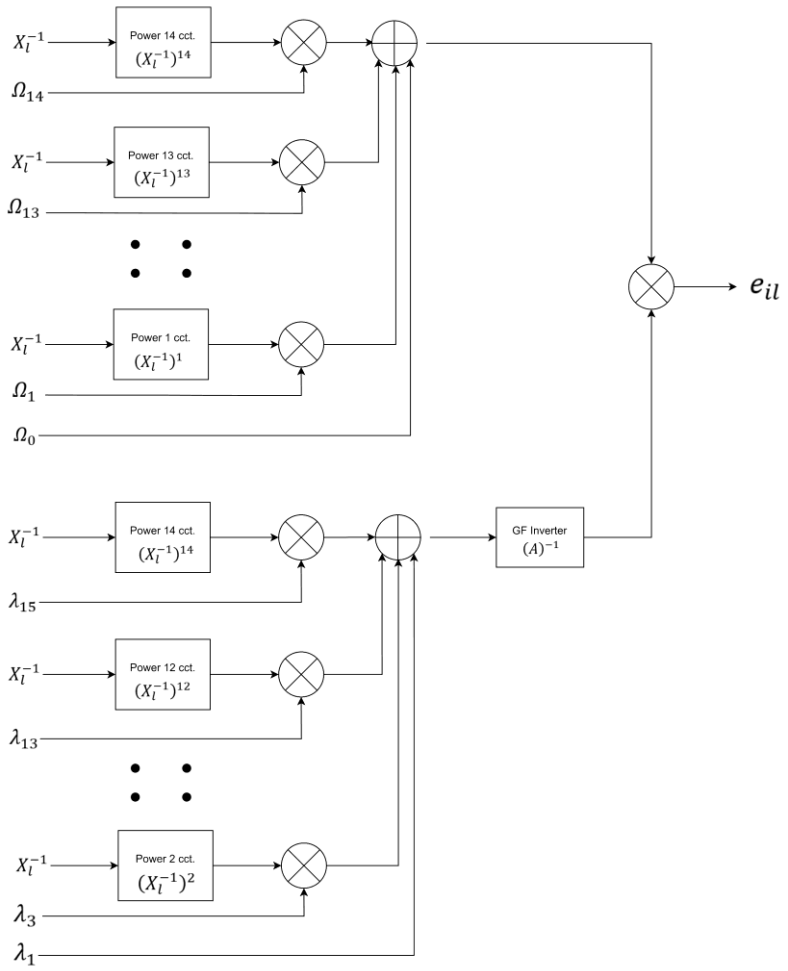



Figure 83: Forney computing unit Direct implementation with parallel multipliers

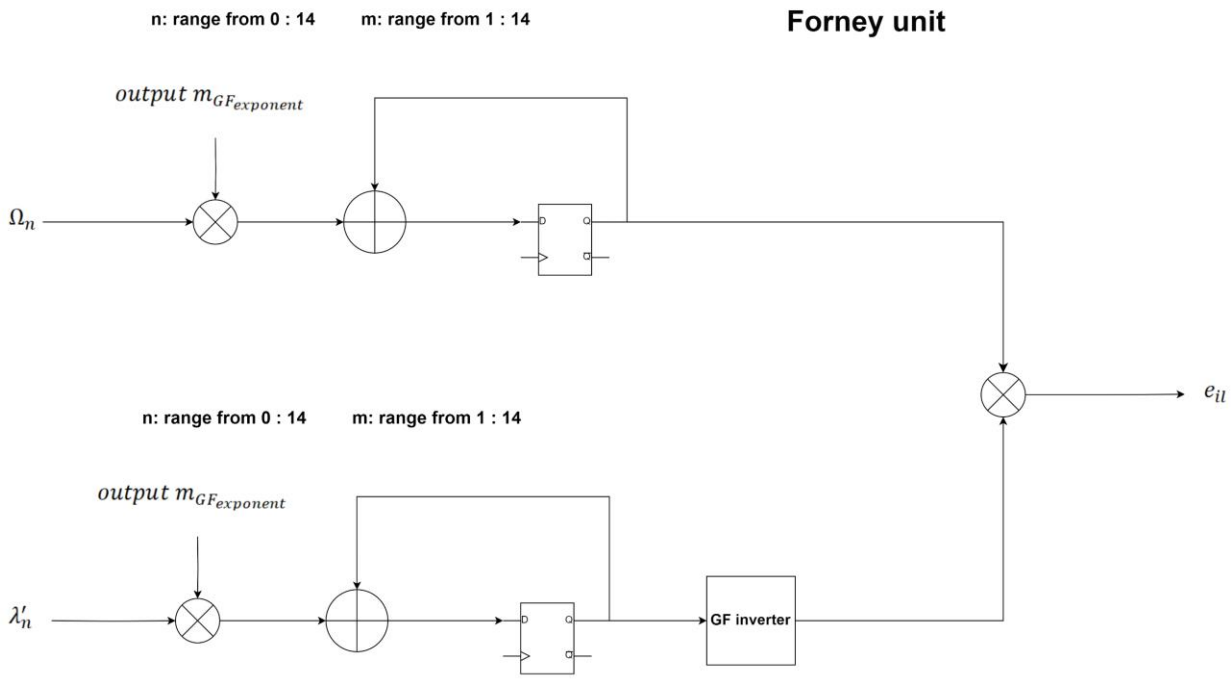



Figure 84: Forney computing unit Area-optimized design with serialized computation

**Forney computing unit implementation**

Evaluation of error magnitude for each available root can be divided into three operations:

- Evaluate substitution root of error locator polynomial in derivation of error locator polynomial using GF multiply-adder accumulate where it requires eight clock cycle to complete evaluation of eight terms as follows:

$$\lambda'\left(X_l^{-1}\right) = \lambda_1 + \lambda_3 X_l^{-1} + \lambda_5 \left(X_l^{-1}\right)^4 + \cdots + \lambda_{15} \left(X_l^{-1}\right)^{14} \tag{23}$$

- After evaluation of substitution of roots in error derivative of error locator polynomial, need to use GF inversion that is designed using pipelined architecture that takes 5 clock cycles so that we get:

$$\frac{1}{\lambda'(X_l^{-1})} \tag{24}$$

- Evaluate substitution root of error locator polynomial in error evaluator polynomial using GF multiply-adder accumulate where it requires 15 clock cycle such that there is 15 GF multiplication-addition operation as follows:

$$\Omega\left(X_l^{-1}\right) = \Omega_0 + \Omega_1 X_l^{-1} + \Omega_2 \left(X_l^{-1}\right)^2 + \cdots + \Omega_{14} \left(X_l^{-1}\right)^{14} \tag{25}$$

After this operation there will be GF multiplication to evaluate the error magnitude:

$$e_{il} = \Omega\left(X_l^{-1}\right) * \frac{1}{\lambda'(X_l^{-1})} \tag{26}$$

### 5.14.5. Error Correction

**Block Diagram**

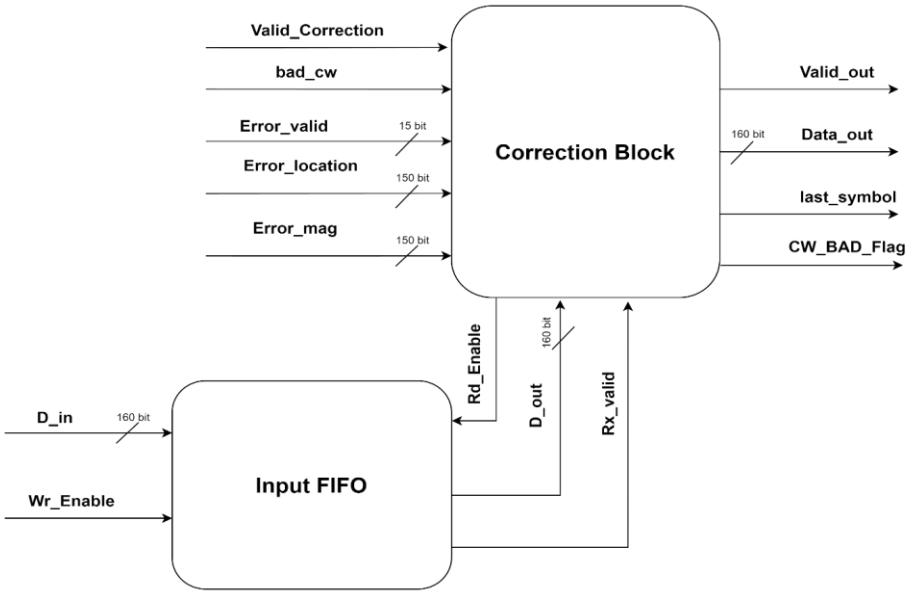



Figure 85: Correction block diagram

Decoder Correction Block as shown consist of Correction and INPUT FIFO. When Data coming to Decoder each frame of data is 160-bit, it goes to 2-blocks as INPUT FIFO and SYNDROMES, which can call that INPUT FIFO store the received codeword and consider main storage. Correction blocks send requests to INPUT FIFO to start sending the stored codeword by using Rd_Enable signal and get Data Frames in D_out and has Enable called Rx_Valid. And this causes using signals that come from FORNEY BLOCK (Valid_correction, Error_valid, Error_mag, Error_location, bad_cw).

Table 25: Error Correction Signals and Interface

| Signal name | Direction | Width | Description |
|---|---|---|---|
| Valid_Correction | Input | 1 | Enable signal of Correction block to start Check. |
| Bad_CW | Input | 1 | Flag indicates that the coming codeword has more than system capability to correct (more than t errors). |
| Error_valid | Input | 15 | Enables for 15 Error than correction can use their information as a Key of room. |
| Error_location | Input | 150 | The concatenation of 15 location (10-bit location for every error). |
| Error_mag | Input | 150 | The concatenation of 15 error magnitudes needed in mask to be ready for XOR operation (10-bit magnitude for every error). |
| RX_valid | Input | 1 | Incoming signal form INPUT FIFO indicates that D_OUT is ready to be received. |

| | | | |
|---|---|---|---|
| D_out | Input | 160 | Data coming for FIFO (Received Frames that entered Decoder). |
| Rd_Enable | Output | 1 | Request signal INPUT FIFO to start sending stored codeword. |
| Valid_out | Output | 1 | Valid signal to next block to appear when data are ready to be used. |
| Data_out | Output | 160 | Output Frame of Data that had XOR operation with MASK to out Clean or have more than 15 error. |
| Last_symbol | Output | 1 | Output signal indicates that the last Data Fram has only 2 symbols (from LSB) and other was not care data. |
| CW_BAD_flag | Output | 1 | Flag indicates that out data was finally clean or still has Errors more than the capability of system (more 15 errors) so if it 0 means that has Clean codeword out, if 1 means it's same input Frame and can't be Clean. |

**FSM of Correction Block**

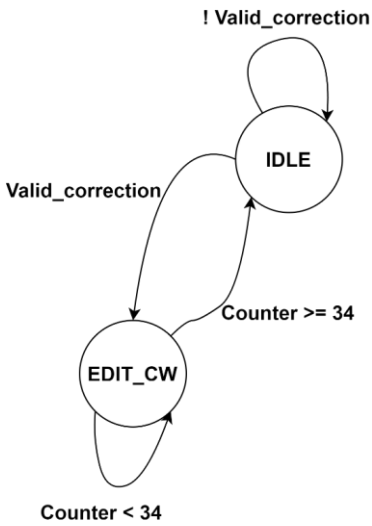



Figure 86: FSM state diagram

This FSM has 2 states (IDLE, EDIT_CW). IDLE state responsible for Freezing and not working and initializing the counters and when Valid_correction be HIGH then go to EDIT_CW state which has 33 iteration , the number of iteration needed came from that original codeword need 34 iteration to out it (each frame 16-symbol and codeword+ Parity are 544-symbol , and symbol is 10-bits). So, by removing parity (30-symbol), the codeword remain became 514-symbol and this can't be divided by 16 so needing to keep it 528-symbol (512-symbol useful + 2-last symbols + 14-dummy symbols).

- **Inside EDIT_CW state in case that Error enables not zero:**

  1) Starting by clearing the mask

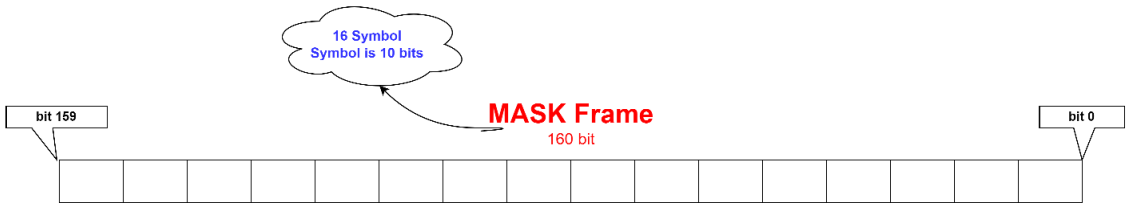



Figure 87: Clearing MASK Frame

  2) Looking at each Error mag, location, enable and detect which one be in this iteration and where to set this location (locations from symbol 0 to 543, but MASK FRAME from symbol 0 to 15), so evaluate in equation that converts the original location to range [0 : 15] to be valid.

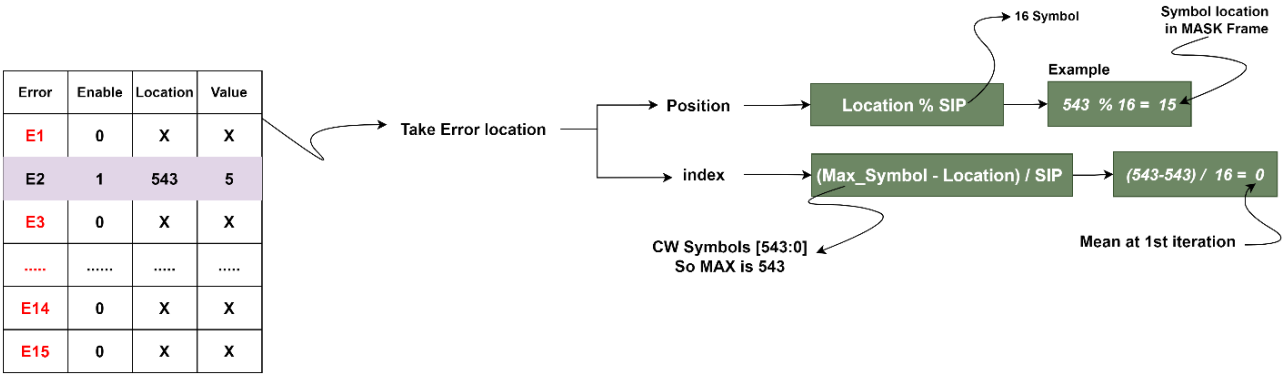



Figure 88: Evaluate Error symbol in MASK Frame

  3) XOR bit wise operation between MASK Frame and Data Frame coming from INPUT FIFO to get Frame Free-Error.

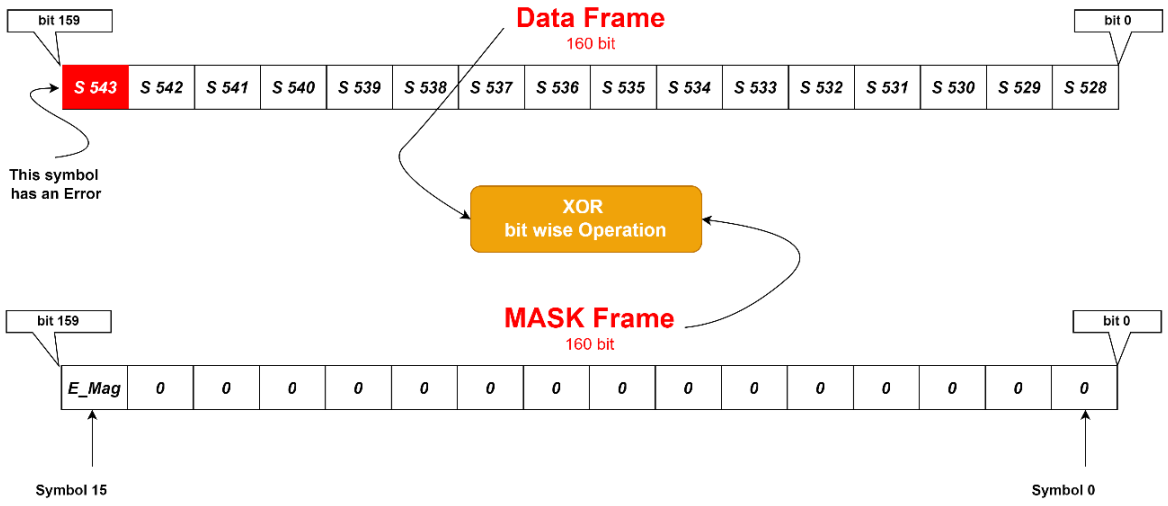



Figure 89: XOR bit wise Operation between Data Frame and MASK Frame

4) Get the results of XOR bit wise operation and set valid to be High.

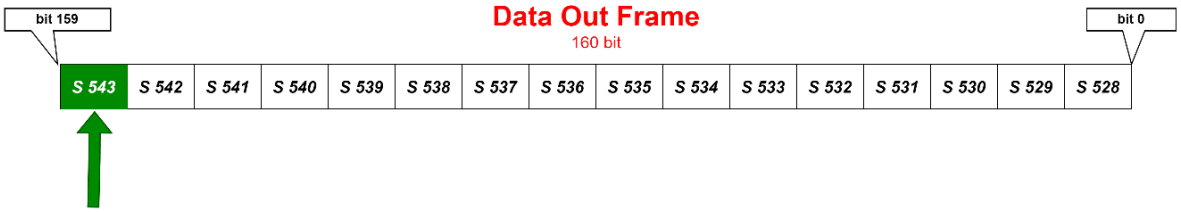



Figure 90: Get Frame of Data out after fixing Error exist in this Frame

- **Inside EDIT_CW state in case that Green Codeword or Bad codeword:**

1) Get Data Frame from FIFO INPUT.
2) Bypassing the Frame without editing it.
3) Incase Errors greater than T symbols (T = 15), set CW_BAD_Flag to be HIGH, if free errors set CW_BAD_Flag to be LOW.

## 5.15. Post-FEC Interleaver

### General Block Diagram

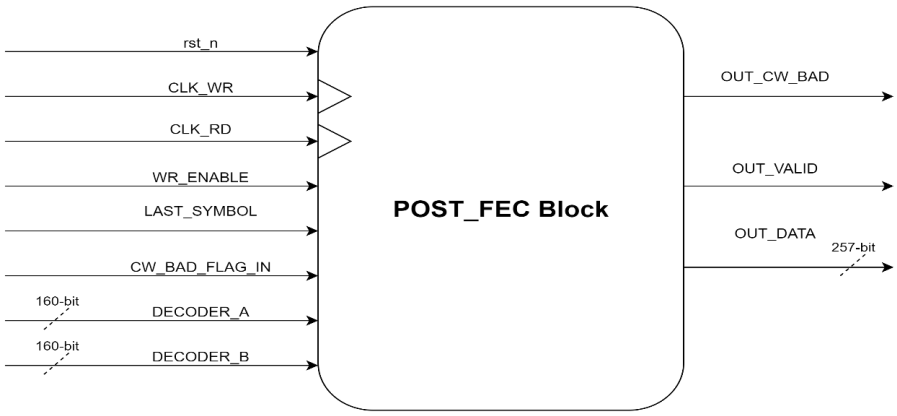



Figure 91:General Block Diagram to POST_FEC_BLOCK

### Interleaver Block Diagram:

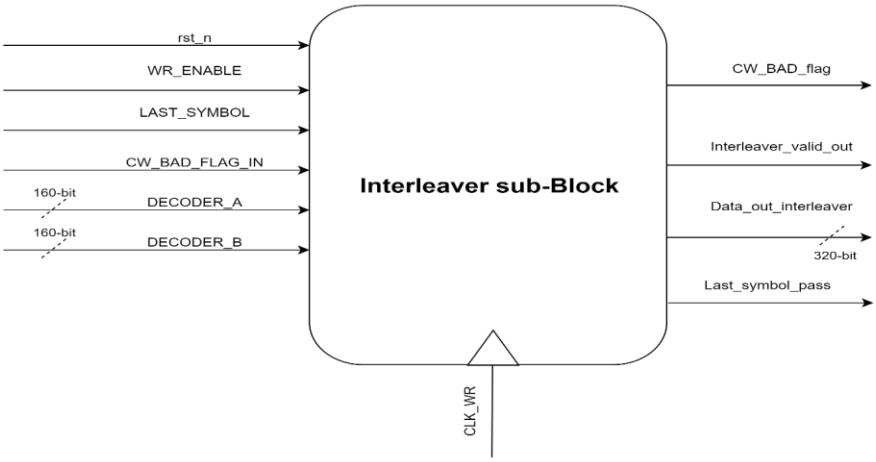



Figure 92: interleave Block Diagram

This sub-block collects 160-bit frames from each decoder and combines them into a single 320-bit frame. This interleaving process ensures data is arranged in a consistent and deterministic pattern, helping reduce burst error sensitivity and improving downstream processing alignment.

**Asynchronous FIFO Block Diagram:**

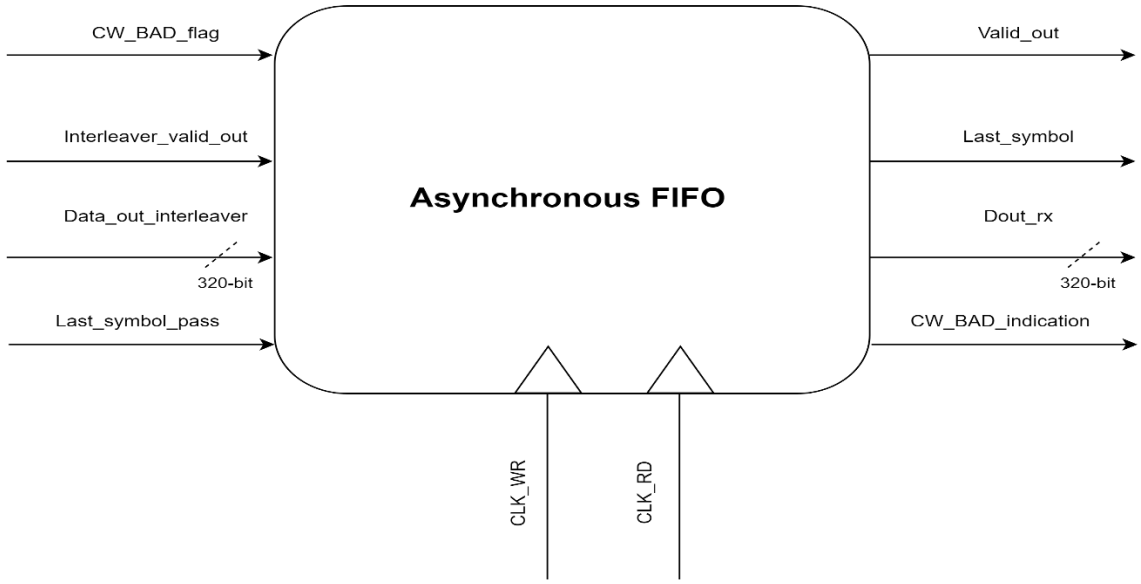



Figure 93: Asynchronous FIFO Block Diagram

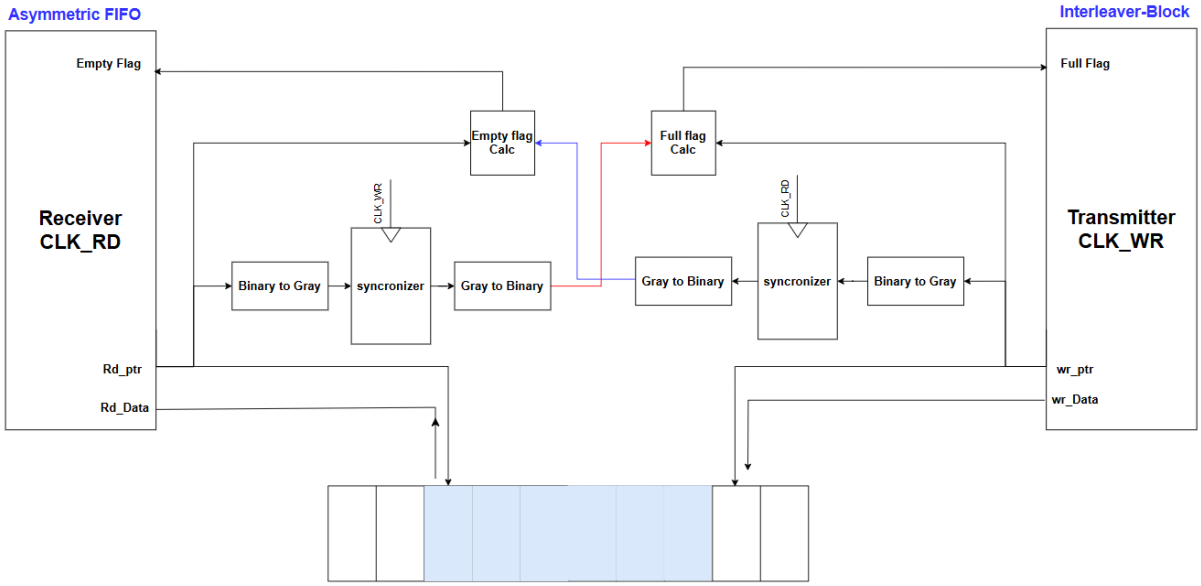



Figure 94: Asynchronous FIFO Architecture

The **Asymmetric FIFO block** is a key component used to resolve **Clock Domain Crossing (CDC)** issues between two asynchronous domains in the system: the **write domain (Transmitter)** operating at **300 MHz**, and the **read domain (Receiver)** running at **400 MHz**. Its architecture enables safe and reliable data transfer while also performing width adaptation, making it asymmetric in nature. In this design:

- Data is written to the FIFO in the **write domain (CLK_WR)** and read in the **read domain (CLK_RD)**.
- To safely transfer control signals (such as read/write pointers) across domains, the design utilizes **gray code conversion**. Binary pointers are first converted to gray code, then synchronized using dedicated synchronizers, and finally reconverted back to binary in the target domain.
- **Full and Empty flags** are calculated using synchronized pointers, ensuring safe FIFO status monitoring without metastability.
- The bottom part of the block (highlighted in blue) represents the actual memory buffer (FIFO storage) that bridges the two domains.

**Asymmetric FIFO Block Diagram:**

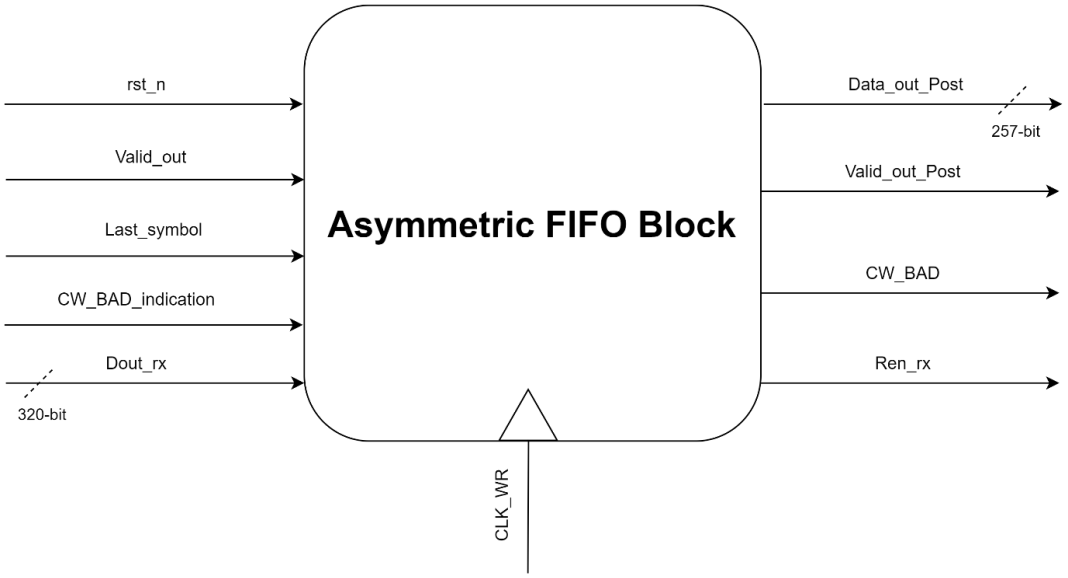



Figure 95: Asymmetric FIFO Block Diagram

The **Asymmetric FIFO block** represents a critical stage in the POST_FEC system pipeline, serving as both a **data width converter** and a buffer to decouple the writing and reading processes. Unlike traditional FIFOs that operate with equal-width input and output data, this FIFO is intentionally designed to handle **asymmetric widths**, transitioning from a **320-bit input** to a **257-bit output**.

**Key Function:**

- The primary role of this block is to reshape the FEC-decoded frame (320 bits) into a format suitable for the following transmission stage, which expects 257-bit output.
- Internally, this is achieved using a **large buffer memory** where new 320-bit frames are written sequentially. Upon each write, the content of the buffer is **shifted left**, creating a continuous stream-like structure. As data accumulates, the reading logic extracts the **most significant 257 bits** from the valid written region.

**Buffer Behavior:**

- Assume the buffer has a physical capacity of 1000 bits. If 500 bits are written into it (as an example), the output logic will take the **highest 257 bits** from that 500-bit valid region, rather than the full buffer. This ensures only valid data is processed and passed downstream.

**Symbol Awareness:**

- Due to the post-correction phase, not all bits in a 320-bit frame are necessarily valid. The valid portion typically corresponds to **4 symbols (i.e., 40 bits)** extracted from two 20-bit symbols from **Decoder A** and **Decoder B** respectively.
- The Asymmetric FIFO block is designed to be **symbol-aware**, meaning it knows the exact location and validity of these symbols. This enables it to maintain the correct alignment and positioning of **valid symbols** when passing data forward.
- Moreover, it ensures the propagation of the **Codeword Bad signal**, which is critical to downstream blocks for distinguishing between clean and corrupted data segments.

**Continuous Data Formatter Block Diagram:**

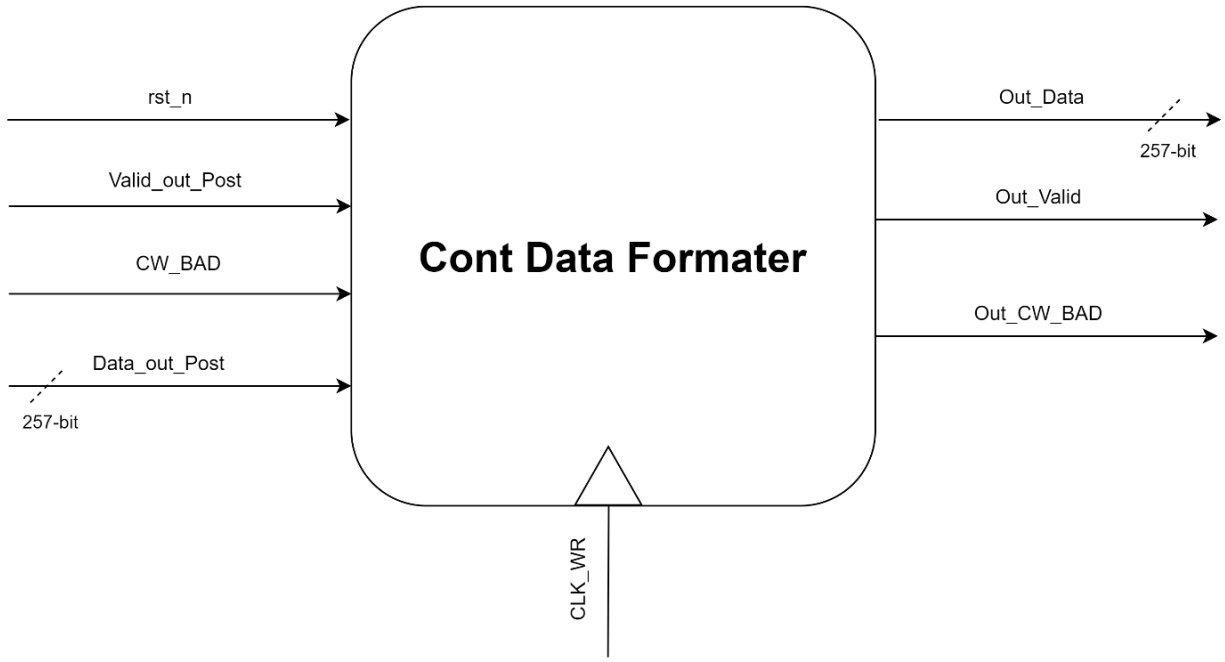



Figure 96: Continuous Data Formatter Block Diagram

The **Continuous Data Formatter** is the final stage in the POST_FEC chain. It reorganizes the output of the Asymmetric FIFO to ensure that **complete and uninterrupted codewords** are transmitted. This guarantees **data continuity and alignment**, preventing errors in downstream modules that rely on receiving fully formed codewords.

Table 26: Post-FEC Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| CLK_WR | INPUT | 1-bit | Clock from Write Domain with Freq 300 MHZ to Enter the Data. |
| CLK_RD | | 1-bit | Clock from Read Domain with Freq 400 MHZ to Read the Stored Data. |
| rst_n | | 1-bit | Synchronous active low reset. |
| WR_ENABLE | | 1-bit | Enable to Write Frames from Decoders to POST_FEC. |
| LAST_SYMBOL | | 1-bit | Flag indicates that the current frame has only 20-bit valid from 160-bit Frame from Decoder A and another 20-bit valid from Frame of Decoder B. |
| CW_BAD_FLAG_IN | | 1-bit | Flag indicates that the current Codeword still has errors and Decoder couldn't fix it because of Number errors more than 15. |
| DECODER_A | | 160-bit | Output Frame of Data from Decoder A. |
| DECODER_B | | 160-bit | Output Frame of Data from Decoder B. |
| OUT_CW_BAD | OUTPUT | 1-bit | Propagate Flag to remining blocks in RX chain that indicates that this codeword couldn't be fixed in decoder. |
| OUT_VALID | | 1-bit | Valid signals for output Data from POST FEC Block. |
| OUT_DATA | | 257-bit | Output Frame of 257-bit From POST FEC Block. |

## 5.16. Alignment Removal

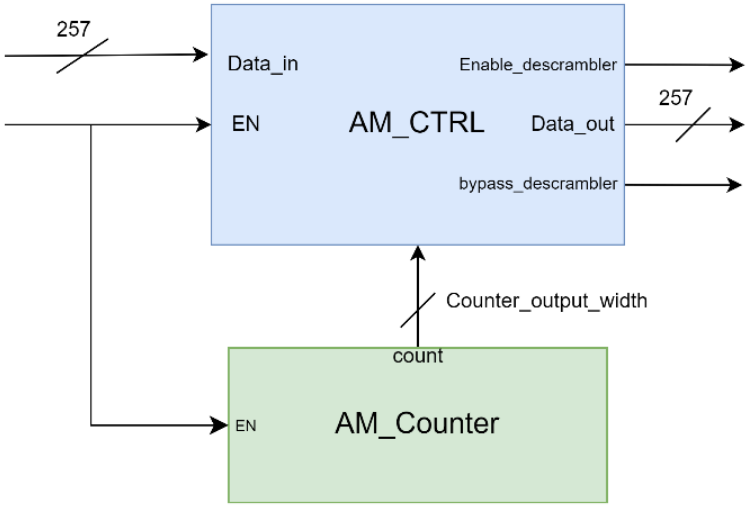



Figure 97: AM removal architecture

The architecture consists of two main building blocks as follows:

**AM CTRL**

This block is responsible for the insertion of idle blocks to ensure that the reverse transcoder output will be four 66-bit idle blocks and managing bypassing of this block by using a bypassing signal for the descrambler. It controls the descrambler's enable when there is valid data that can be read from AM removal, whether during idle insertion or normal bypassing operation of AM removal.

**AM counter**

The AM counter is responsible for counting the number of input blocks and resets to zero when entering a new AM period. It informs the AM CTRL of the beginning of an AM period when the counter is zero.

Table 27: Alignment Removal Signals and Interface

| Signal name | direction | width | Description |
|---|---|---|---|
| Data_in | input | 257 | Incoming data from post-FEC. |
| EN | input | 1 | Enable signal from post-FEC indicating whether current bus data is valid to read or not. |
| Enable_descrambler | output | 1 | Valid signal informing the descrambler that valid data exists on AM removal Data_out. |
| Data_out | output | 257 | Output data from AM removal block. |
| Bypass_descrambler | output | 1 | Bypass signal enabling the descrambler to bypass incoming blocks without processing. |

## 5.17. Descrambler

Modelling the descrambling logic as MIMO system, we can see the following results [25]:

$$\hat{\mathbf{d}}_k = \widehat{\mathbf{A}}_p \cdot \hat{\mathbf{d}}_{k-1} + \widehat{\mathbf{B}}_p \cdot \tilde{\mathbf{b}}_{k-1} \quad , \ \hat{d}_k: \text{states of the descrambler}$$

$$\hat{\mathbf{b}}_k = \widehat{\mathbf{C}}_p \cdot \hat{\mathbf{d}}_k + \widehat{\mathbf{D}}_p \cdot \tilde{\mathbf{b}}_k \quad , \ \tilde{b}_k: \text{srambled data} \ , \hat{b}_k: \text{descrambled data}$$

After solving the previous state equations, a **recursive substitution approach** can be applied to minimize the number of XOR operations as performed in scrambler, so we can model the system by those equations:

```
descrambled_output_0 = Scrambled_in_0 + reg_2 + reg_4
descrambled_output_1 = Scrambled_in_1 + reg_1 + reg_3
descrambled_output_2 = Scrambled_in_2 + reg_0 + reg_2
descrambled_output_3 = Scrambled_in_0 + Scrambled_in_3 + reg_1
descrambled_output_4 = Scrambled_in_1 + Scrambled_in_4 + reg_0
descrambled_output_5 = Scrambled_in_0 + Scrambled_in_2 + Scrambled_in_5
descrambled_output_6 = Scrambled_in_1 + Scrambled_in_3 + Scrambled_in_6
descrambled_output_7 = Scrambled_in_2 + Scrambled_in_4 + Scrambled_in_7
descrambled_output_8 = Scrambled_in_3 + Scrambled_in_5 + Scrambled_in_8
```

```
reg_new_0 = past_in_4
reg_new_1 = past_in_5
reg_new_2 = past_in_6
reg_new_3 = past_in_7
reg_new_4 = past_in_8
```

Figure 98: Equations for the Descrambler output and internal register

The equations show that in the descrambler, the internal registers depend on the input, in contrast to the scrambler, where they depend on the output.

This behavior creates a **one-to-many relationship** between the input and output, where a single input can produce multiple possible outputs depending on the internal state. To resolve this issue, we can **eliminate the feedback path from the input to the registers** (same concept as in the scrambler). By doing so, the descrambler's output becomes fully determined by the input alone, ensuring a unique and predictable output for each input pattern without altering its intended functionality. To simplify the design, we select an all-zero initial seed to ensure synchronization between the transmitter and receiver.

- Output equations for the Descrambler after removing the registers will be:

| | |
|---|---|
| $0 \leq i < M$ | $\mathbf{descram\_out[i] = descram\_in[i]}$ |
| $M \leq i < L$ | $\mathbf{descram\_out[i] = descram\_in[i] \wedge descram\_in[i-M]}$ |
| $L \leq i < N$ | $\mathbf{descram\_out[i] = descram\_in[i] \wedge descram\_in[i-M] \wedge descram\_in[i-L]}$ |

## 5.18. Reverse Transcoder 257B/256B

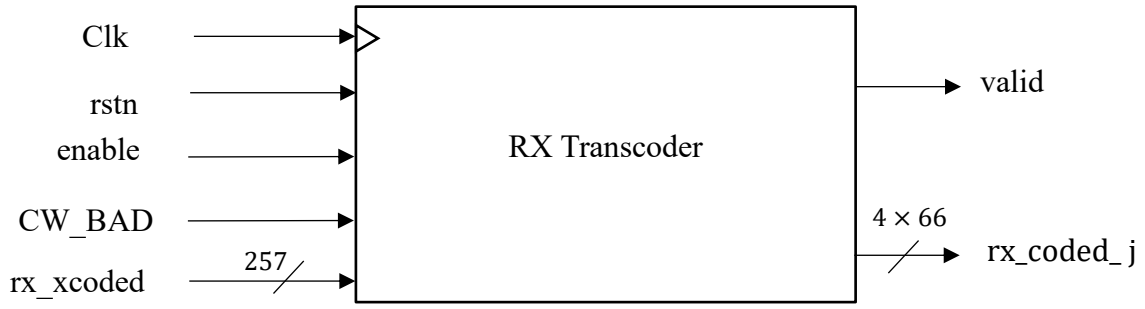


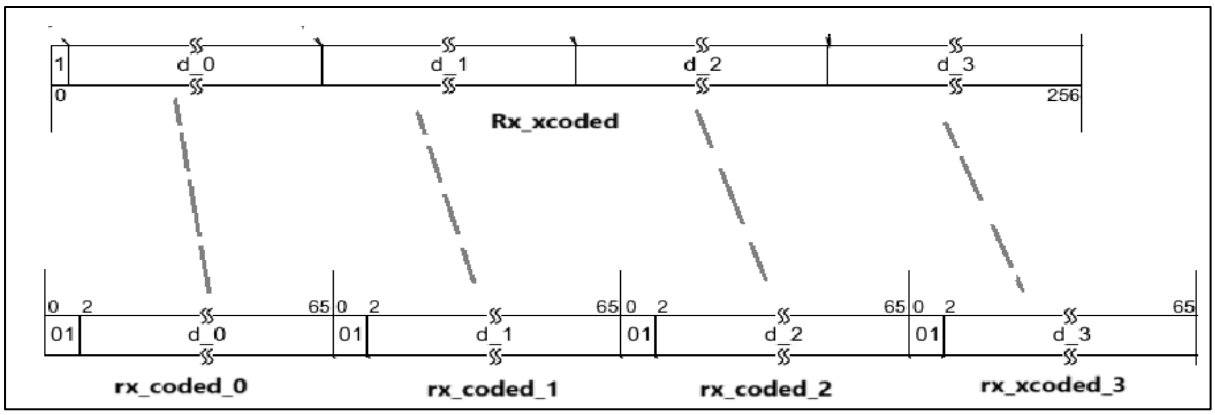

Table 28: Reverse Transcoder Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| rx_xcoded | input | 257 | Descrambled 257 Bit Block Data. |
| enable | input | 1 | Enable signal will activate the block when the descrambler process finishes. |
| CW_BAD | input | 1 | High Priority Active High Flag when Reed Solomon decoder determines that a codeword contains errors that were not corrected. |
| valid | output | 1 | Valid signal indicates that reverse transcoding finishes and enable decoder 66B/64B. |
| rx_coded_j | output | 4 × 66 | 4 reverse transcoded blocks 66B each. |

**The 4 Blocks are Data (rx_xcoded [256] = 1)**

If 1st bit in 257-bit block was 1 that mean the 4 blocks will be data with sync header “01”.

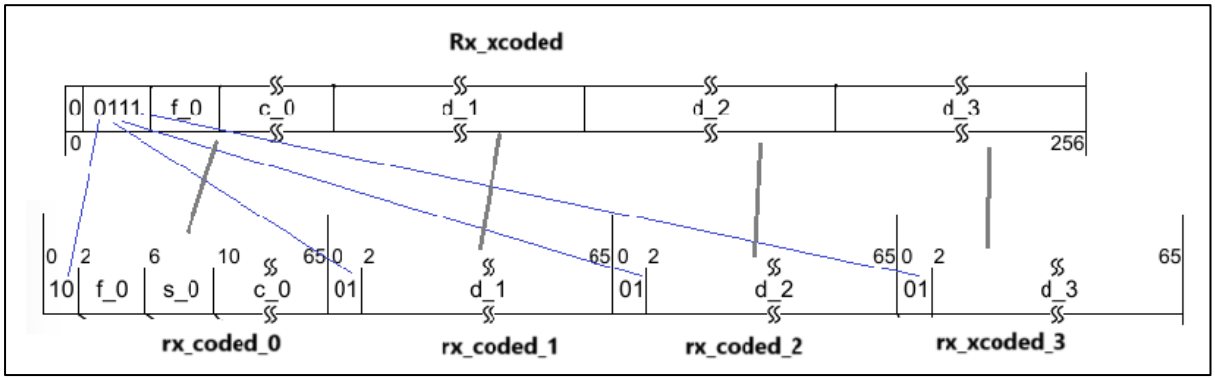
Example:




**The 4 Blocks are Mix between data and control**

For data block its sync header will be “01”, and for control block sync header will be “10”

Example: 1st block is control and the 3 remaining are data (rx_xcoded [256:252] = 5’b00111)




To know S_0 which is the 2nd nibble in the 1st control block:

Using the cross-reference table below (Block Type Field):

- 0x1 ↔ 0xE, 0x7 ↔ 0x8, 0x4 ↔ 0xB, 0x9 ↔ 0x9, 0xA ↔ 0xA, till the end of table.

For example, if f_0 in rx_xcoded equals 0xE, then S_0 of rx_coded_0 should be set to 0x1.

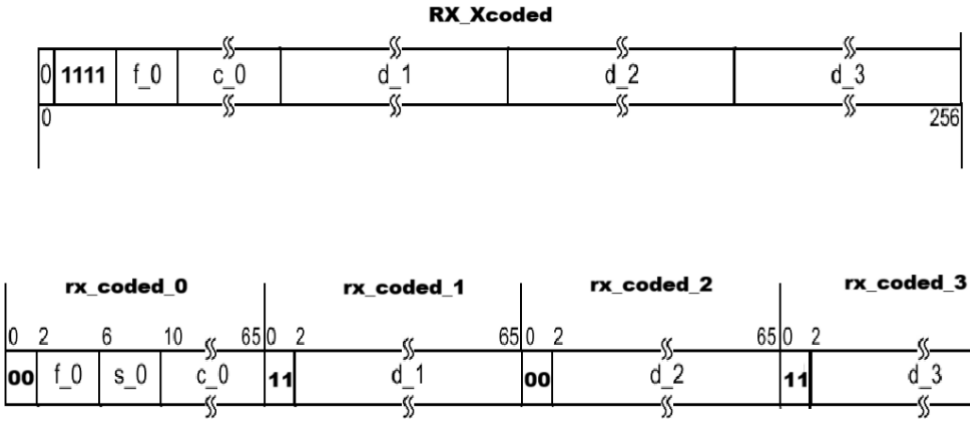
If Rx_coded_0[4:2] does not match any entry in the table, then Rx_coded_0[9:6] is set to 0000 (an invalid sync header), and in that case, rx_coded_0 [65:64] is set to 2'b11.

| Input Data | Sync | Block Payload | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bit Position: / Data Block Format: | 0 1 | 2 | | | | | | | 65 |
| $D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7$ | 01 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| **Control Block Formats:** | | **Block Type Field** | | | | | | | |
| $C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7$ | 10 | 0x1E | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ $C_7$ |
| $S_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7$ | 10 | 0x78 | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| $O_0 D_1 D_2 D_3 Z_4 Z_5 Z_6 Z_7$ | 10 | 0x4B | $D_1$ | $D_2$ | $D_3$ | $O_0$ | 0x000_0000 | | |
| $T_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7$ | 10 | 0x87 | | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ $C_7$ |
| $D_0 T_1 C_2 C_3 C_4 C_5 C_6 C_7$ | 10 | 0x99 | $D_0$ | | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ $C_7$ |
| $D_0 D_1 T_2 C_3 C_4 C_5 C_6 C_7$ | 10 | 0xAA | $D_0$ | $D_1$ | | $C_3$ | $C_4$ | $C_5$ | $C_6$ $C_7$ |
| $D_0 D_1 D_2 T_3 C_4 C_5 C_6 C_7$ | 10 | 0xB4 | $D_0$ | $D_1$ | $D_2$ | | $C_4$ | $C_5$ | $C_6$ $C_7$ |
| $D_0 D_1 D_2 D_3 T_4 C_5 C_6 C_7$ | 10 | 0xCC | $D_0$ | $D_1$ | $D_2$ | $D_3$ | | $C_5$ | $C_6$ $C_7$ |
| $D_0 D_1 D_2 D_3 D_4 T_5 C_6 C_7$ | 10 | 0xD2 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | | $C_6$ $C_7$ |
| $D_0 D_1 D_2 D_3 D_4 D_5 T_6 C_7$ | 10 | 0xE1 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $C_7$ |
| $D_0 D_1 D_2 D_3 D_4 D_5 D_6 T_7$ | 10 | 0xFF | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ |

Figure 99: 64B/66B Block formats

### Invalid block combinations (rx_xcoded [256:252] = 5'b01111)

If the first bit of the 257-bit block is 0, followed by four consecutive 1s, and the S_0 nibble of the first block will be 0000, then set the sync header of the first and third 66-bit blocks to 2'b00, while setting the sync headers of the remaining blocks to 2'b11.



## 5.19. Decoder 66B/64B

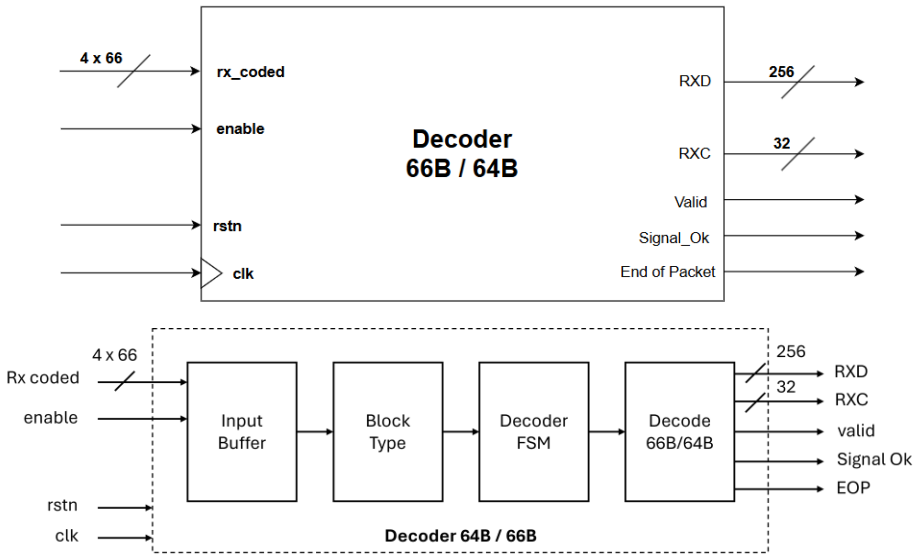
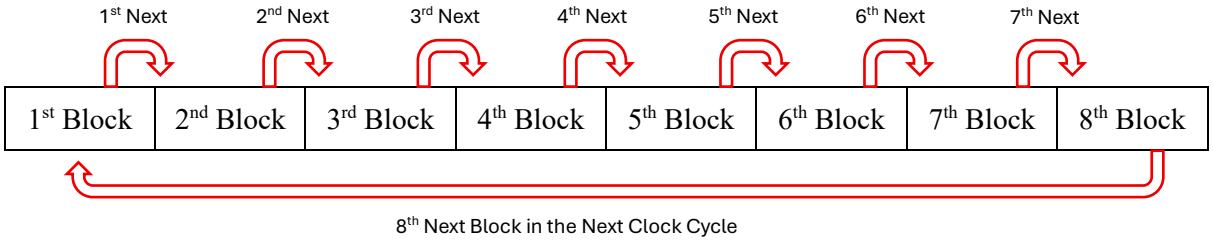



Table 29: Decoder 66B/64B Signals and Interface

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| rx_coded_j | input | 4 × 66 | 4 reverse transcoded blocks 66B each. |
| enable | input | 1 | Enable signal will activate the block when the descrambler process finishes. |
| RXD | output | 256 | Decoded 256B for 4 block 64B each transmitted to RECONCILIATION layer it takes formats mentioned in block format table. |
| RXC | output | 32 | Control 32B for 4 blocks 8B each. where if control bit, is one means control block, and when zero it means their data character exists in corresponding octet position. |
| valid | output | 1 | Valid signal indicates that the decoding process finishes and enables the RECONCILIATION layer. |
| Signal Ok | output | 1 | Signal Ok to Guarantee That RXD Decoded Output Without Errors. |
| EOP | output | 1 | A valid end of packet occurs when a block containing a /T/ is followed by a control block that does not contain a /T/ or an /E/. |

**Input Buffer Block**

It buffers 8 blocks of 66 Bit instead of the usual 4. This is because each block needs to know the next one to help detect the end of a packet. Since the first 4 blocks alone aren't enough, another 4 are added in the next clock cycle, making a total of 8 blocks over 2 clock cycles.




### Decoder Block Type

It includes a function R_TYPE that independently checks each 66-byte block against the block type field table. If the block is not found on the table, it returns an ERROR block. Otherwise, it outputs the corresponding 64-byte block type and classifies it as one of the following:

1) Data 2) IDLE 3) Terminate 4) Start 5) Order Set 6) Error

**Decoder FSM:** Checks the sequence of the four blocks and the next four to verify compliance with the state machine.

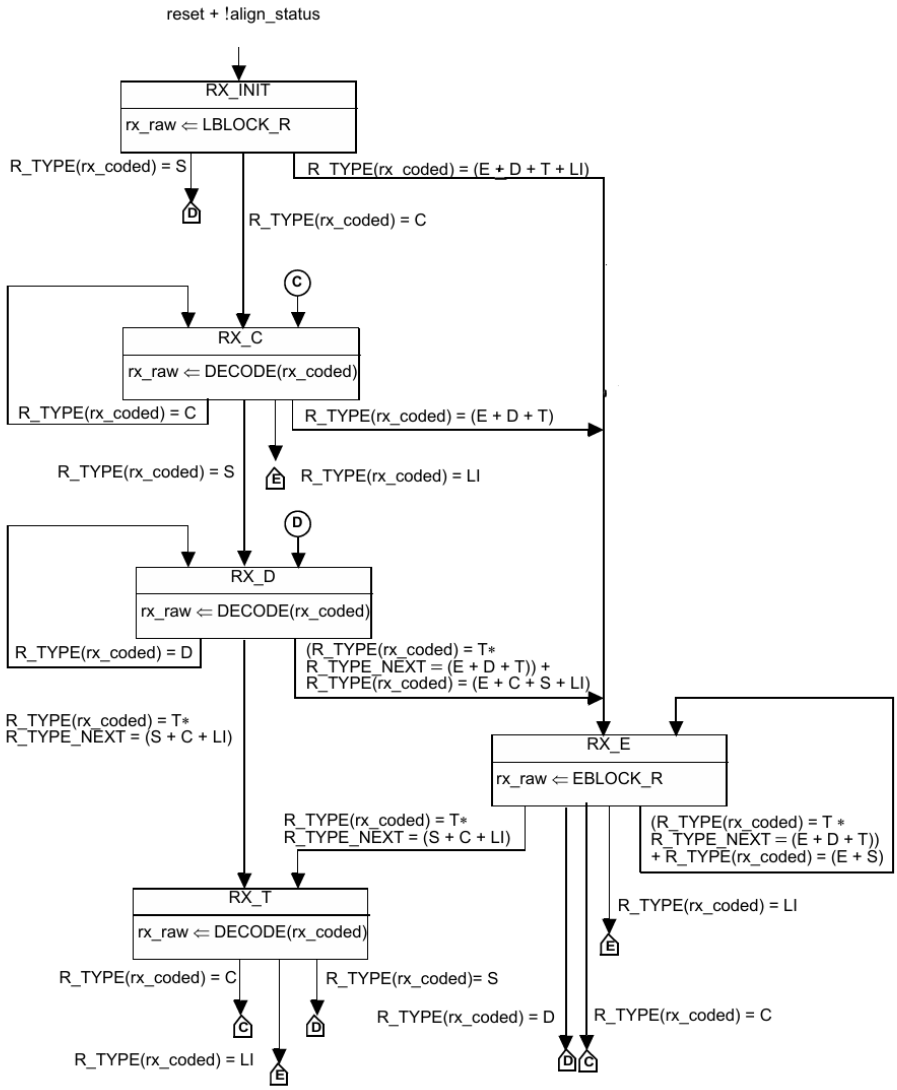



Figure 100: State Machine of Decoder 66B/64B

Table 30: Decoder 66B/64B FSM States

| State Name | Description |
|---|---|
| RX_INIT | Initial state of Decoder after reset. |
| RX_C | The Control state occurs when a 66B incoming block has a sync header of 10 and contains eight 7-bit idle control characters. |
| RX_D | The Data state occurs when a 66B incoming block has a sync header of 01 and contains 8 data characters. |
| RX_T | The Terminate state occurs when the incoming 66B block contains 8-bit data characters then zeros followed by 7-bit idle control characters. |
| RX_E | The Error state is triggered when the block contains an error character or is not listed in the block type field table. |

**Decoding process**

Finally, we decode the four 66B blocks, check the state of each based on the block type table, and convert them to their corresponding 64B format. We then extract and combine the RXD [63:0] and RXC [7:0] values for each block to form RXD [255:0] and RXC [31:0]. If any error characters are detected in RXD or RXC, the Signal OK signal is set low; otherwise, it is set high. We also evaluate the eop (end of packet) signal by checking if a Terminate block is followed only by control blocks, with no Data block appearing afterward using the next block type function. Once decoding is complete, the valid signal is asserted from the last block to the reconciliation layer.

# 6. Results and Analysis

## 6.1. Encoder 64B/66B

After implementation on the Zynq UltraScale+ MPSoC
(device number: xczu17eg-ffve1924-3-e), the area, timing, and power summary are displayed

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.916 ns | Worst Hold Slack (WHS): | 0.028 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1629 | Total Number of Endpoints: | 1629 | Total Number of Endpoints: | 1644 |

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1509 | 423403 | 0.36 |
| FF | 1643 | 846806 | 0.19 |
| IO | 556 | 668 | 83.23 |
| BUFG | 1 | 940 | 0.11 |

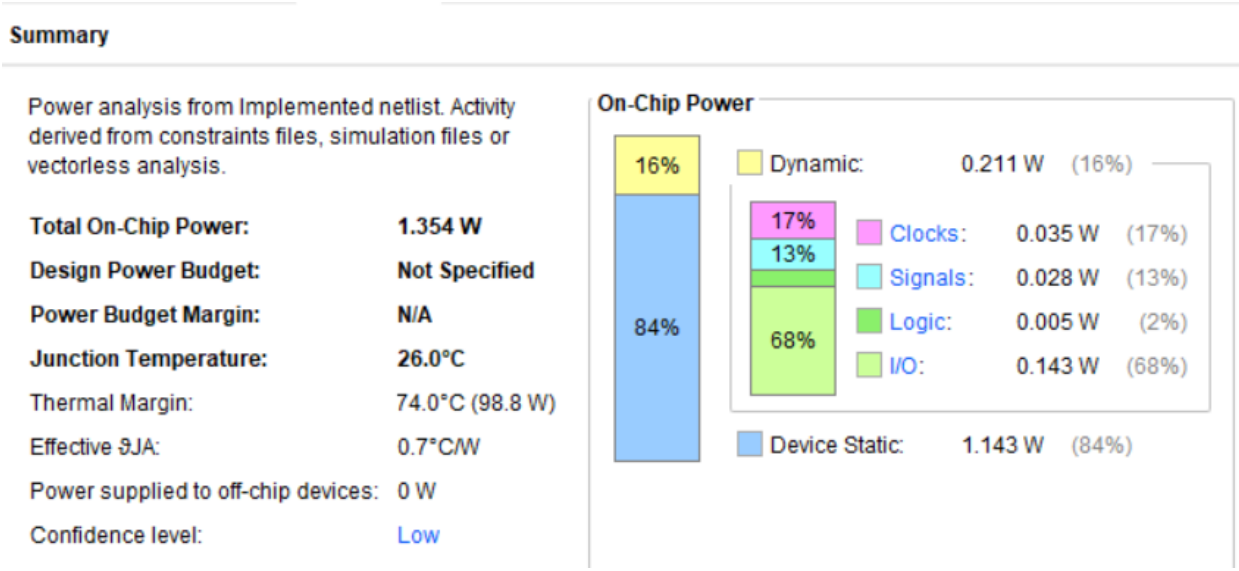



Figure 101: Encoder64B/66B Implementation Results

## 6.2. Rate Matching

After implementation on the Zynq UltraScale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the area, timing, and power summary are displayed.

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1410 | 423403 | 0.33 |
| FF | 2202 | 846806 | 0.26 |
| IO | 532 | 668 | 79.64 |
| BUFG | 1 | 940 | 0.11 |

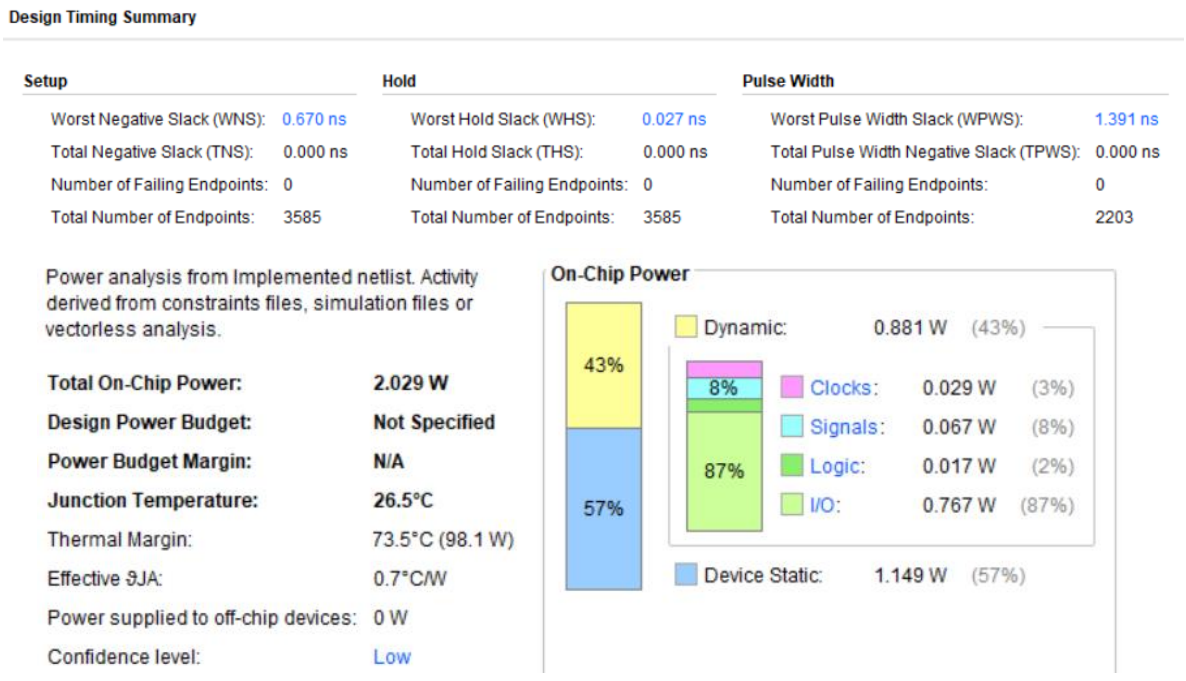



Figure 102: Rate Matching Implementation Results

## 6.3. Tx Transcoder 256B/257B

- Transcoder design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics including resource utilization, timing performance, and power consumption are summarized as follows:


| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | LUT Flip Flop Pairs (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tx_transcoder | 212 | 523 | 92 | 212 | 201 | 525 | 244 | 244 | 3 | 1 | 33 |

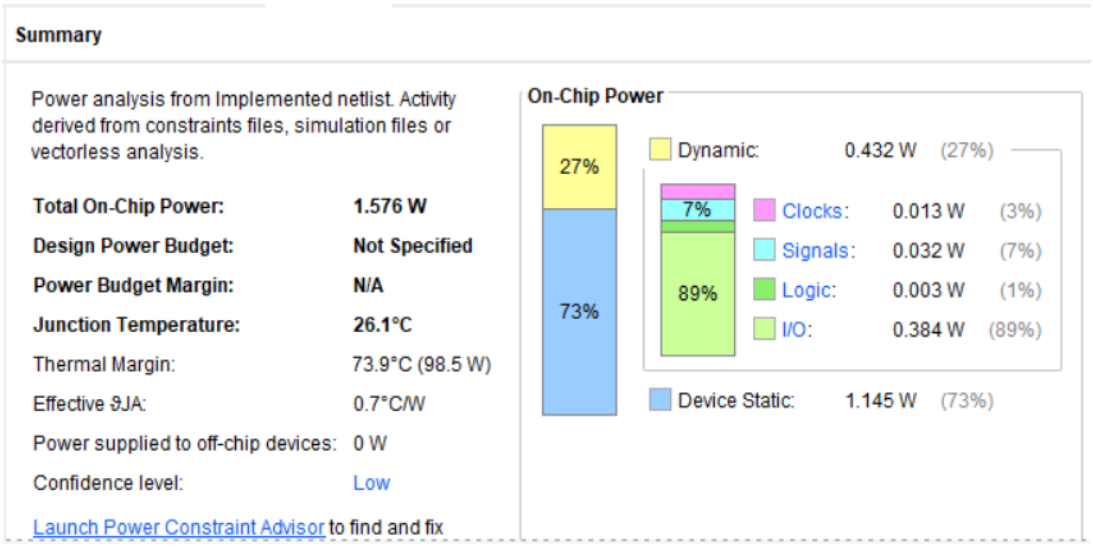



Figure 103: Tx Transcoder 256B/257B Implementation Results

## 6.4. Scrambler

Scrambler design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics — including resource utilization, timing performance, and power consumption — are summarized as follows:

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.244 ns | Worst Hold Slack (WHS): | 0.059 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 515 | Total Number of Endpoints: | 515 | Total Number of Endpoints: | 517 |

**All user specified timing constraints are met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | LUT Flip Flop Pairs (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| N Parallel_Scrambler | 247 | 516 | 74 | 247 | 142 | 518 | 240 | 240 | 3 | 1 | 34 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.849 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.3°C** |
| Thermal Margin: | 73.7°C (98.3 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

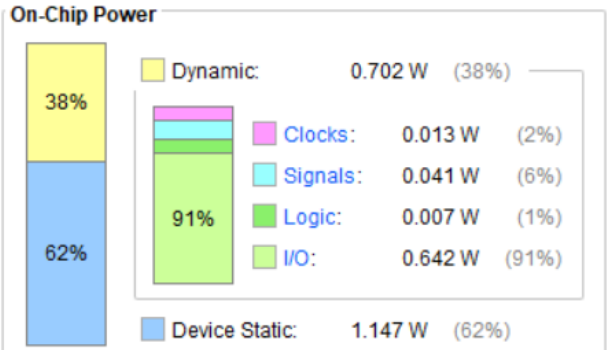



Figure 104: Scrambler Implementation Results

## 6.5. Alignment Marker Insertion

The following results are related to the target device: FPGA Zynq Ultra Scale+ part name: (xczu17eg-ffve1924-3-e).

- Our speed goal is still 300 MHz which the design surpasses so that the design can work up to 421 MHz.
- The hold slack is acceptable since the route delays on the system level will be enough to offer more delays.

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.959 ns | Worst Hold Slack (WHS): | 0.021 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 4475 | Total Number of Endpoints: | 4475 | Total Number of Endpoints: | 2408 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1960 | 423403 | 0.46 |
| FF | 2407 | 846806 | 0.28 |
| IO | 518 | 668 | 77.54 |
| BUFG | 1 | 940 | 0.11 |

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **2.013 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.5°C** |
| Thermal Margin: | 73.5°C (98.1 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

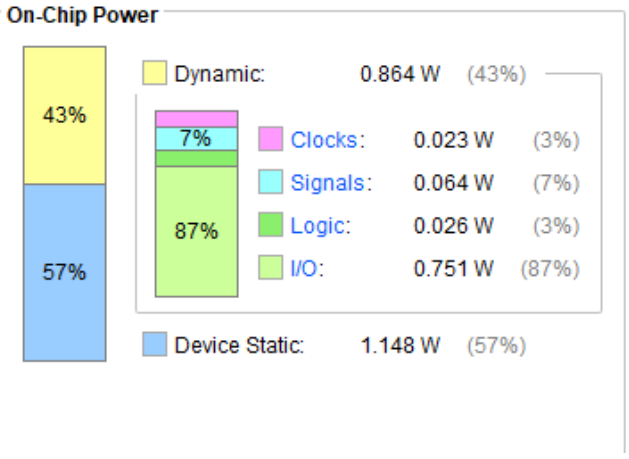



Figure 105: Alignment Markers Insertion Implementation Results

## 6.6. Pre-FEC Distribution

- The following results are related to the target device: FPGA Zynq Ultra Scale+ part name: (xczu17eg-ffve1924-3-e).

- Our speed goal is still 300 MHz which the design surpasses so that the design can work up to 367 MHz.

- The hold slack is acceptable since the route delays on the system level will be enough to offer more delays.

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.611 ns | Worst Hold Slack (WHS): | 0.029 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 2258 | Total Number of Endpoints: | 2258 | Total Number of Endpoints: | 1257 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 3769 | 423403 | 0.89 |
| FF | 1256 | 846806 | 0.15 |
| IO | 582 | 668 | 87.13 |
| BUFG | 1 | 940 | 0.11 |

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **2.443 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.8°C** |
| Thermal Margin: | 73.2°C (97.7 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

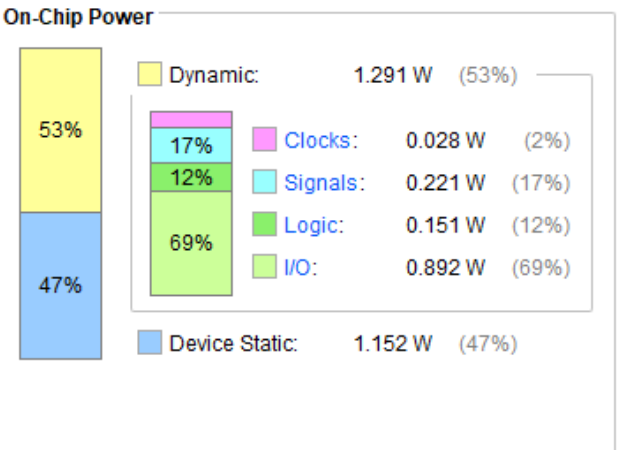



Figure 106: Pre-FEC Implementation Results

## 6.7. RS FEC Encoder

- The following comparative results are held between the conventional parallel encoder using the matrix-based representation & the parallel encoder using the 2D IMA.
- The following results are related to the target device: FPGA Zynq Ultra Scale+ part name: (xczu17eg-ffve1924-3-e).

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.231 ns | Worst Hold Slack (WHS): | 0.036 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 2227 | Total Number of Endpoints: | 2227 | Total Number of Endpoints: | 1280 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 4614 | 423403 | 1.09 |
| FF | 1279 | 846806 | 0.15 |
| IO | 324 | 668 | 48.50 |
| BUFG | 1 | 940 | 0.11 |

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **2.372 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.7°C** |
| Thermal Margin: | 73.3°C (97.8 W) |
| Effective ϑJA: | 0.7°C/W |

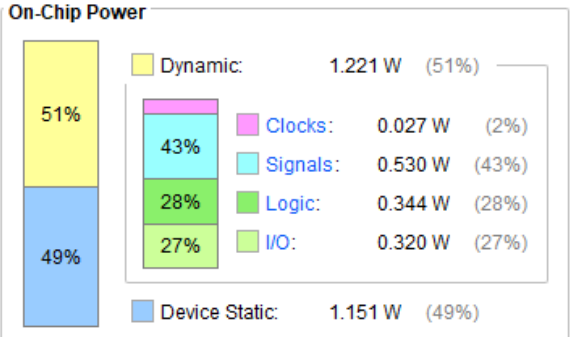



Figure 107: Conventional RS-FEC Encoder Implementation Results

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.427 ns | Worst Hold Slack (WHS): | 0.038 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 2227 | Total Number of Endpoints: | 2227 | Total Number of Endpoints: | 1280 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 4765 | 423403 | 1.13 |
| FF | 1279 | 846806 | 0.15 |
| IO | 324 | 668 | 48.50 |
| BUFG | 1 | 940 | 0.11 |

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **2.365 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.7°C** |
| Thermal Margin: | 73.3°C (97.8 W) |
| Effective ϑJA: | 0.7°C/W |

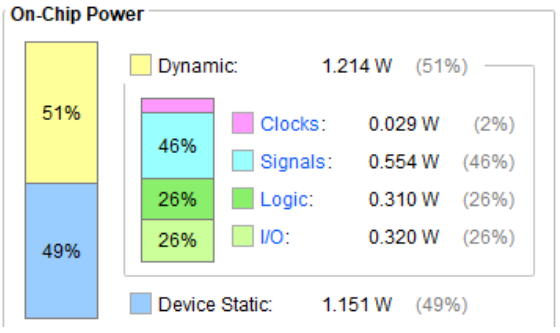
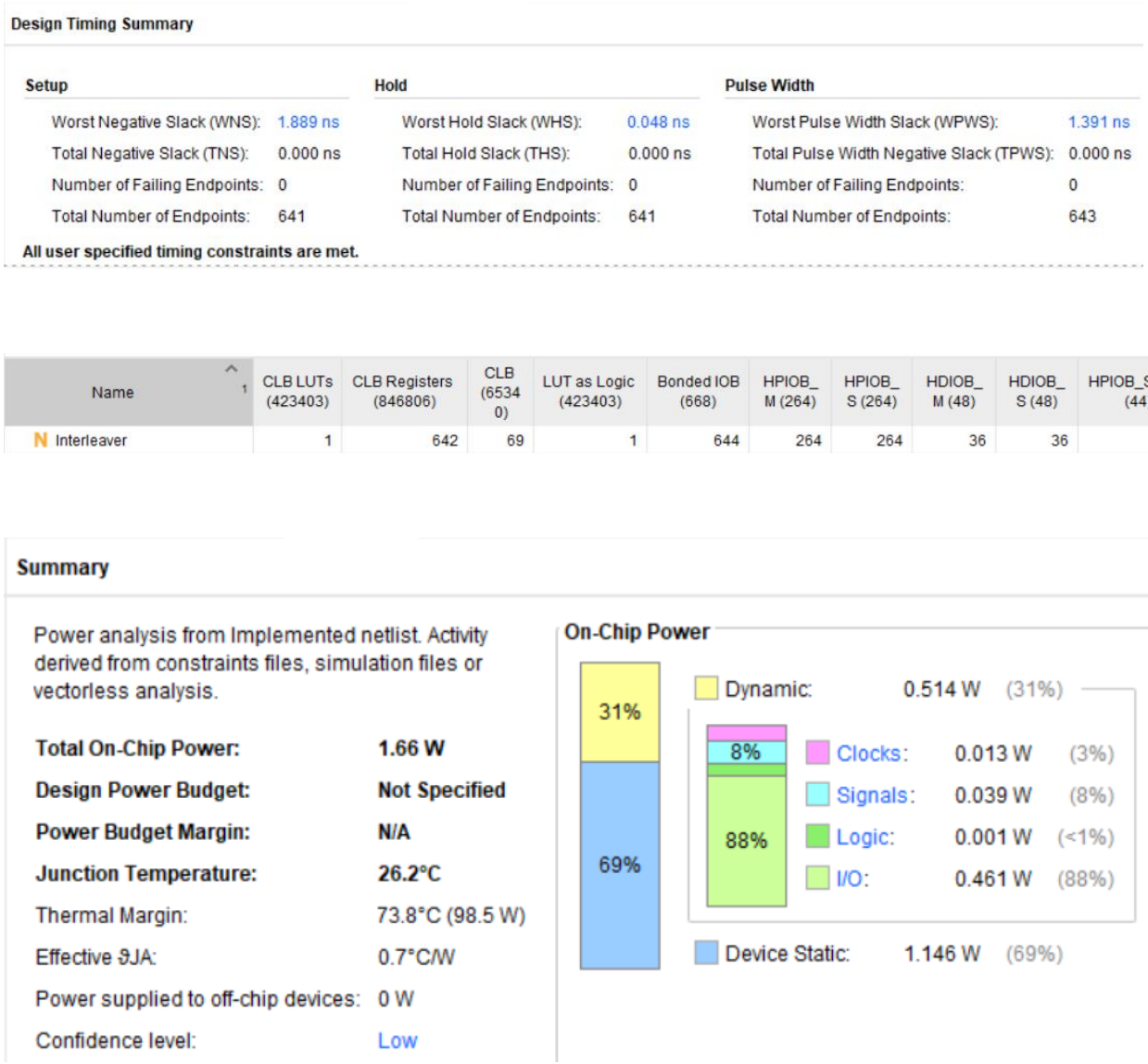



Figure 108: Optimized RS-FEC Encoder Implementation Results

## 6.8. Interleaver

- Interleaver design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics including resource utilization, timing performance, and power consumption are summarized as follows:



**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.889 ns | Worst Hold Slack (WHS): | 0.048 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 641 | Total Number of Endpoints: | 641 | Total Number of Endpoints: | 643 |

**All user specified timing constraints are met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|
| N Interleaver | 1 | 642 | 69 | 1 | 644 | 264 | 264 | 36 | 36 | 44 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.66 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.2°C** |
| Thermal Margin: | 73.8°C (98.5 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

**On-Chip Power**



Figure 109: Interleaver Implementation Results

As demonstrated by the results above, the interleaver meets the timing requirements and is area-efficient, utilizing only a single LUT.

## 6.9. Symbol Distribution

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at 300 MHZ.

### Symbol Distribution Latency

- It takes **1 Clock Cycle** to execute output to PCS Lanes.

### Symbol Distribution Timing

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.950 ns | Worst Hold Slack (WHS): | 0.030 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 641 | Total Number of Endpoints: | 641 | Total Number of Endpoints: | 643 |

**All user specified timing constraints are met.**

### Symbol Distribution Utilization

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 161 | 423403 | 0.04 |
| FF | 321 | 846806 | 0.04 |
| IO | 644 | 668 | 96.41 |
| BUFG | 1 | 940 | 0.11 |

### Symbol Distribution Power

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.615 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.2°C** |
| Thermal Margin: | 73.8°C (98.5 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

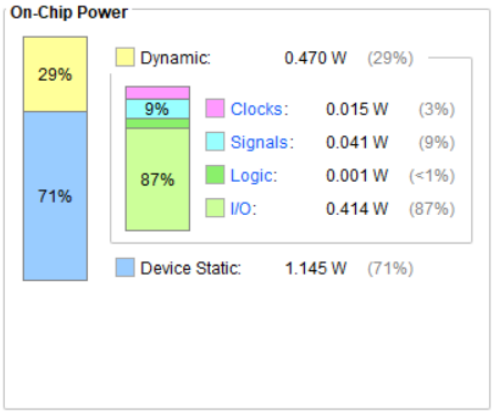



Figure 110: Symbol Distribution Implementation Results

## 6.10. Alignment Lock and Deskew

The module is implemented on the Zynq Ultra Scale+ 104 board, and it achieves the required timing and throughput specifications, the total utilization is 6% which is to be expected as the block requires a lot of logic to perform its functions. The implementation results are shown in figure (111).

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.259 ns | Worst Hold Slack (WHS): | 0.013 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 8043 | Total Number of Endpoints: | 8043 | Total Number of Endpoints: | 7484 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 21440 | 423403 | 5.06 |
| FF | 7483 | 846806 | 0.88 |
| IO | 648 | 668 | 97.01 |
| BUFG | 1 | 940 | 0.11 |

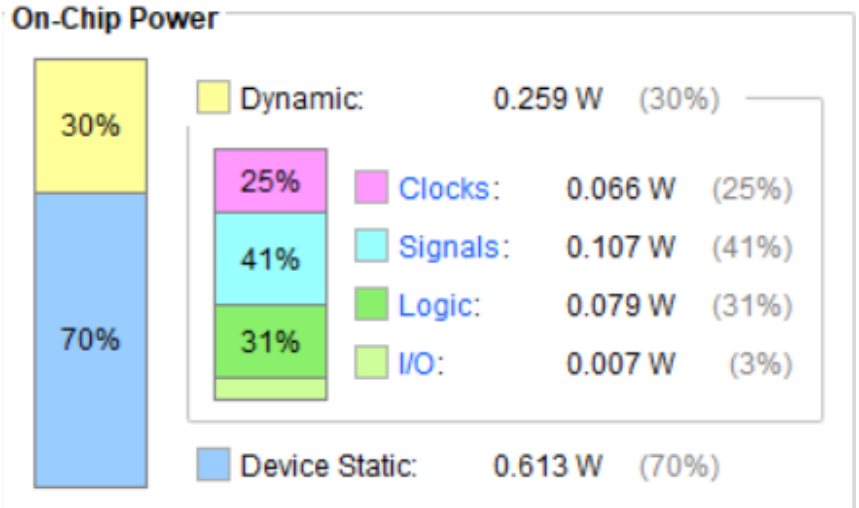



Figure 111: Alignment Lock and Deskew Implementation Results

## 6.11. Lane Reorder

Lane reorder design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics — including resource utilization, timing performance, and power consumption — are summarized as follows:

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.705 ns | Worst Hold Slack (WHS): | 0.098 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 641 | Total Number of Endpoints: | 641 | Total Number of Endpoints: | 675 |

**All user specified timing constraints are met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | LUT Flip Flop Pairs (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| N lane_reorder | 2103 | 674 | 430 | 2103 | 317 | 668 | 264 | 264 | 48 | 48 | 44 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.838 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.3°C** |
| Thermal Margin: | 73.7°C (98.3 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

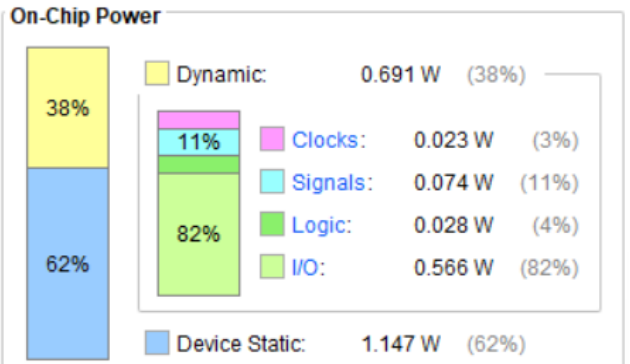



Figure 112: Lane Reorder Implementation Results

## 6.12. Deinterleave

Deinterleave design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics including resource utilization, timing performance, and power consumption are summarized as follows:

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 2.093 ns | Worst Hold Slack (WHS): | 0.031 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 641 | Total Number of Endpoints: | 641 | Total Number of Endpoints: | 643 |

**All user specified timing constraints are met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|
| N Deinterleaver | 1 | 642 | 69 | 1 | 644 | 264 | 264 | 36 | 36 | 44 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.614 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.2°C** |
| Thermal Margin: | 73.8°C (98.5 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

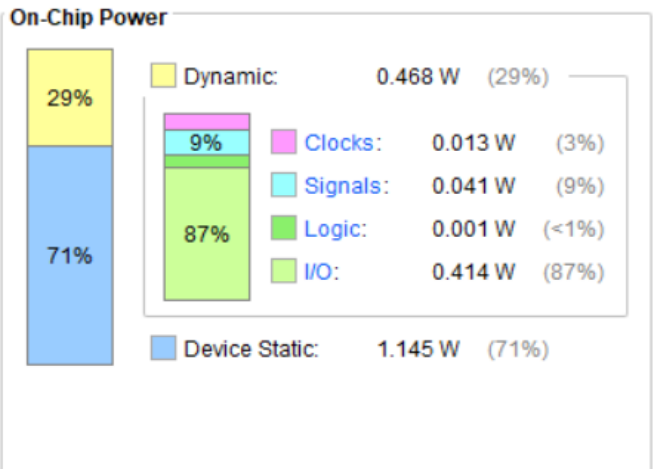



Figure 113: Deinterleave Implementation Results

As demonstrated by the results above, Deinterleave meets the timing requirements and is area-efficient, utilizing only a single LUT.

## 6.14. RS FEC Decoder

### 6.14.1. GF Multiplier

The modules are implemented on Zynq Ultra Scale+ 104 board with the non-pipelined results shown in figures (114) while the pipelined results are shown in figures (115).

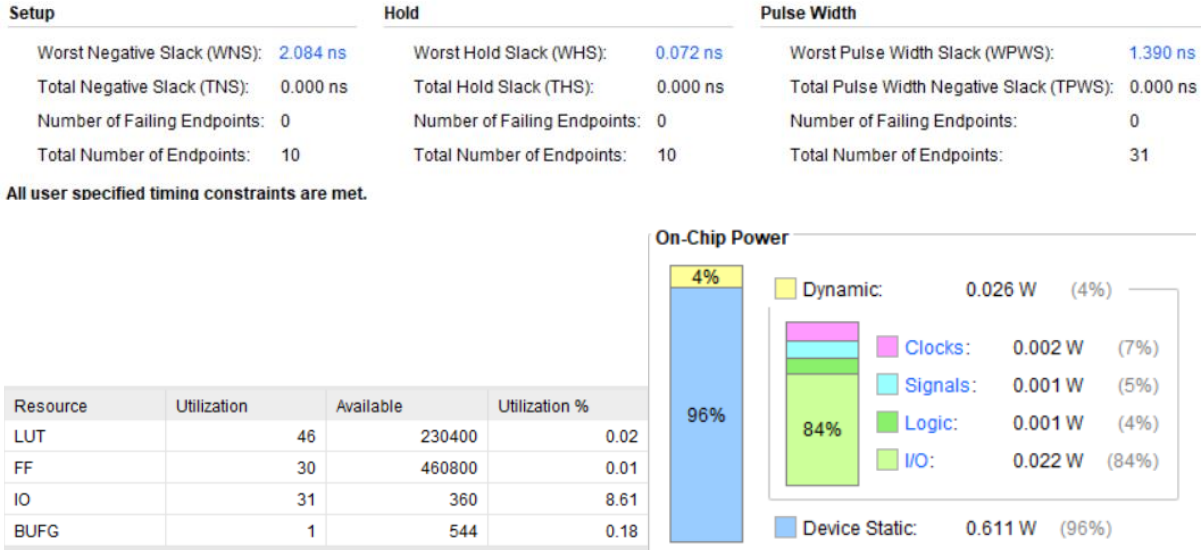



Figure 114: GF Multiplier Implementation Results

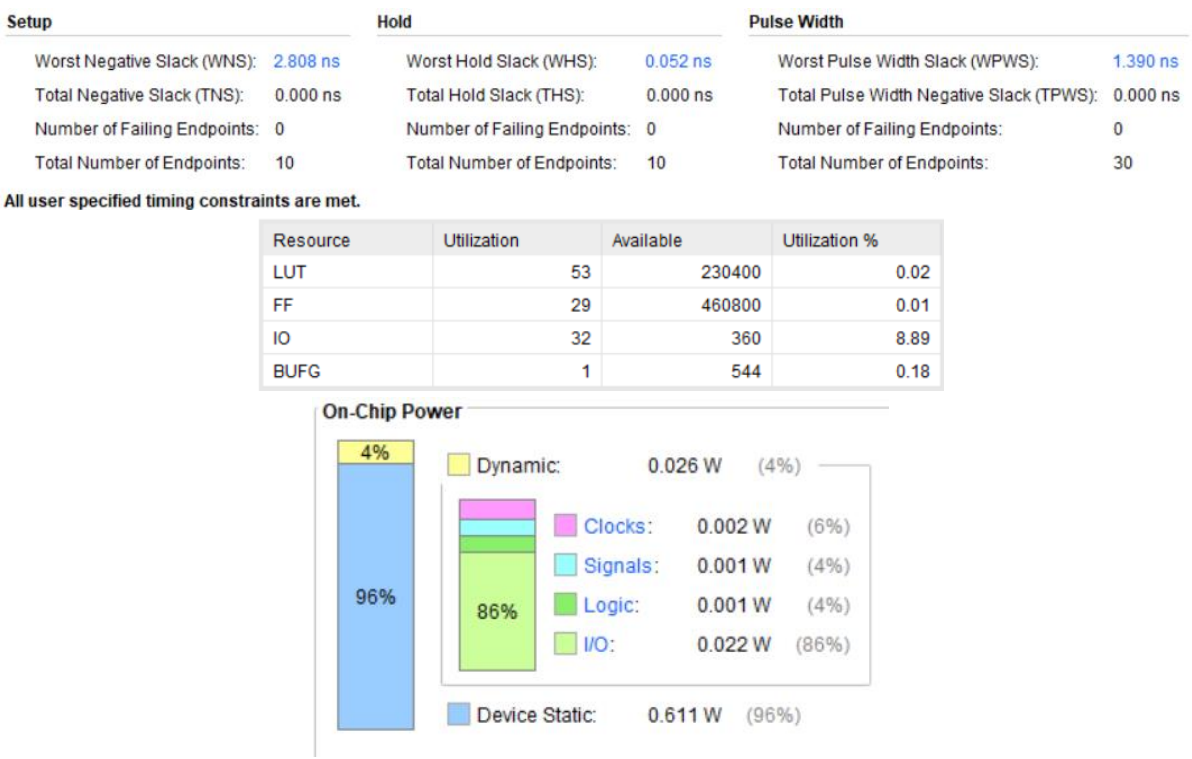


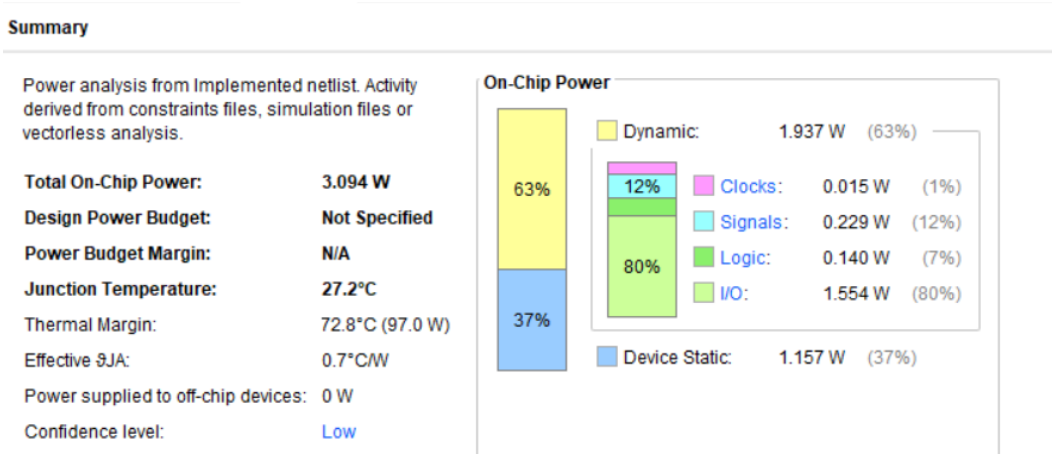

Figure 115: Pipelining Implementation Results

### 6.14.2. Syndrome Calculation

After implementation on the Zynq Ultra Scale+ MPSoC
(device number: xczu17eg-ffve1924-3-e), the area, timing, and power summary are displayed

**Optimized Syndrome Calculation Implementation:**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 2505 | 423403 | 0.59 |
| FF | 439 | 846806 | 0.05 |
| IO | 464 | 668 | 69.46 |
| BUFG | 1 | 940 | 0.11 |



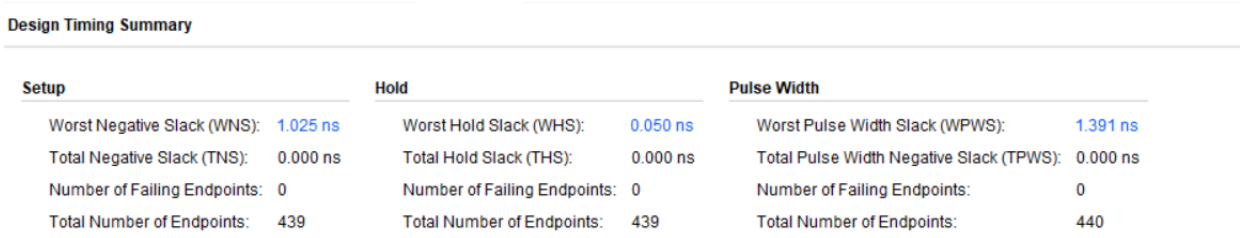




Figure 116: Syndrome Implementation Results

### 6.14.3. Key Equation Solver

The module is implemented on Zynq Ultra Scale+ 104 board. The timing report, utilizations summary and power consumption report of the KES implementation are shown in figures (117) respectively. The implementation met the constraints with the non-pipelined GF multiplier therefore the pipelined GF multiplier was not used.

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.442 ns | Worst Hold Slack (WHS): | 0.042 ns | Worst Pulse Width Slack (WPWS): | 1.390 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1460 | Total Number of Endpoints: | 1460 | Total Number of Endpoints: | 978 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 5126 | 230400 | 2.22 |
| FF | 977 | 460800 | 0.21 |
| IO | 308 | 360 | 85.56 |
| BUFG | 1 | 544 | 0.18 |

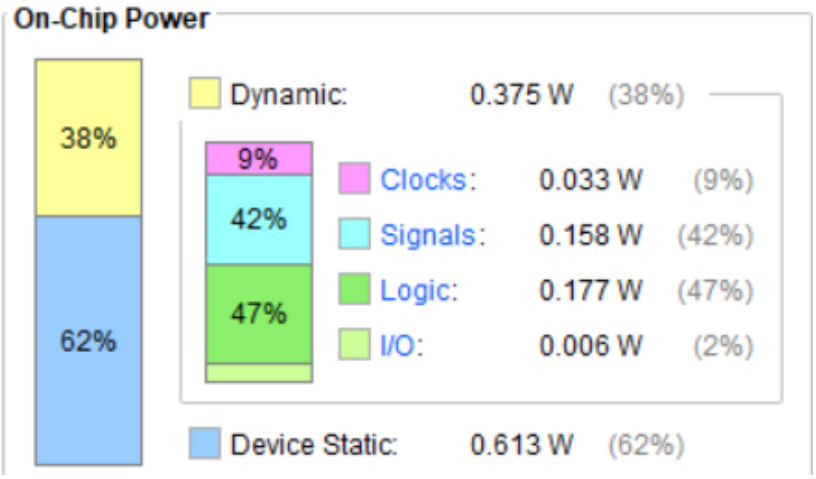



Figure 117: Key Equation Solver Implementation Results

### 6.14.4. Chien Search

The Chien Search design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **300 MHz** (corresponding to a 3.333 ns clock period). The resulting hardware metrics — including resource utilization, timing performance, and power consumption — are summarized as follows:

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.069 ns | Worst Hold Slack (WHS): | 0.042 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1167 | Total Number of Endpoints: | 1167 | Total Number of Endpoints: | 780 |

**All user specified timing constraints are met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | LUT Flip Flop Pairs (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Chien_Search | 10007 | 779 | 1521 | 10007 | 695 | 484 | 215 | 215 | 12 | 12 | 30 |
| LOCATION_CHK (Chie... | 9348 | 404 | 1508 | 9348 | 364 | 0 | 0 | 0 | 0 | 0 | 0 |
| ROOTS (Chien_error_r... | 492 | 11 | 287 | 492 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.543 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.1°C** |
| Thermal Margin: | 73.9°C (98.6 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

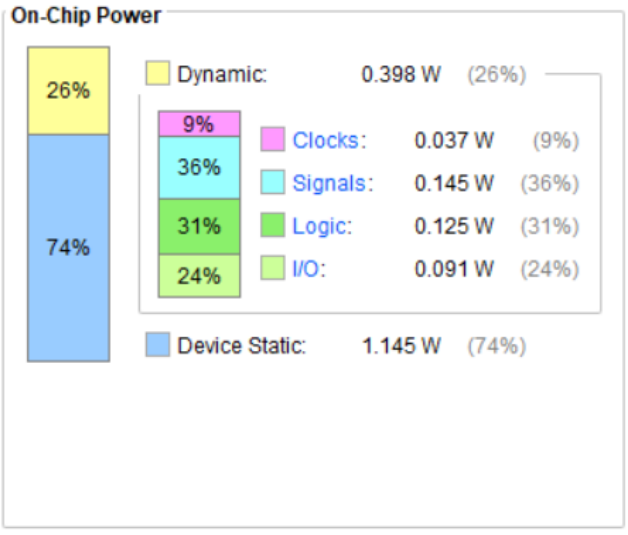



Figure 118: Conventional Chien Search Implementation Results

To validate the effectiveness of the proposed architecture, a comparison was conducted against an alternative parallel Chien Search implementation based on the **Iterative Matching Algorithm (IMA)**. In the IMA approach, each finite field multiplier (FFM) is modeled using a matrix representation. Bit-level optimization is then applied iteratively to minimize the number of required XOR gates by identifying and reusing **common subexpressions (CSEs)** [28].

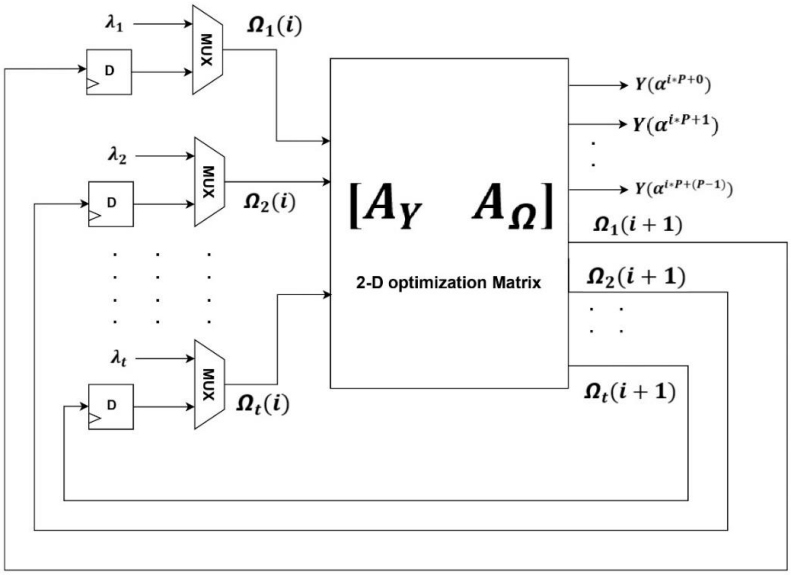



Figure 119: Chien Search Architecture using

The following are the implementation results of the Chien Search architecture based on the Iterative Matching Algorithm (IMA):

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | -8.354 ns | Worst Hold Slack (WHS): | 0.023 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | -3607.463 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 673 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1135 | Total Number of Endpoints: | 1135 | Total Number of Endpoints: | 786 |

**Timing constraints are not met.**

| Name | CLB LUTs (423403) | CLB Registers (846806) | CLB (65340) | LUT as Logic (423403) | LUT Flip Flop Pairs (423403) | Bonded IOB (668) | HPIOB_M (264) | HPIOB_S (264) | HDIOB_M (48) | HDIOB_S (48) | HPIOB_SNGL (44) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| N Chien_Search | 12739 | 785 | 1986 | 12739 | 706 | 484 | 215 | 215 | 12 | 12 | 30 |
| LOCATION_CHK (Chie... | 12164 | 424 | 1975 | 12164 | 376 | 0 | 0 | 0 | 0 | 0 | 0 |
| ROOTS (Chien_error_r... | 384 | 11 | 282 | 384 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

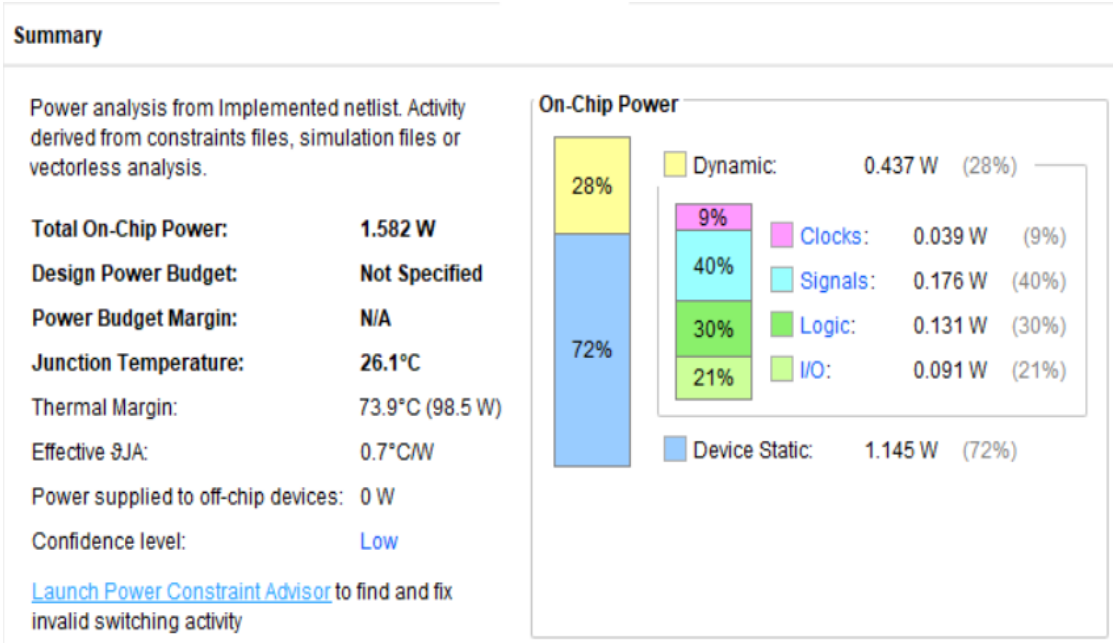



Figure 120: Chien Search using IMA Implementation Results

Based on the results presented above, it can be concluded that the proposed parallel Chien Search architecture utilizing GF multipliers outperforms the IMA-based parallel implementation in key performance metrics.
A summary of the comparison is provided in the table below.

Table 31: Chien Search Comparison for Different Architecture

| Points of Comparison | Chien Search using GF multipliers | Chien Search using IMA |
|---|---|---|
| **Latency** | 34 clock cycle | 34 clock cycle |
| **Setup slack** | 0.069 ns | -8.354 ns |
| **Hold slack** | 0.042 ns | 0.023 ns |
| **Utilization (Area)** | 10,007 look up tables (LUTs) | 12,739 LUTs |
| **Power** | 1.543 watt | 1.582 watt |

### 6.14.5. Forney Algorithm

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at **300 MHZ.**

**Optimized (Resource Sharing) Forney Results:**

**Optimized Forney Latency**

- It requires an **initial latency of 22 clock cycles** to generate error magnitudes followed by a **1 clock cycle** pipeline delay.

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 2.324 ns | Worst Hold Slack (WHS): | 0.033 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1865 | Total Number of Endpoints: | 1865 | Total Number of Endpoints: | 1171 |

**All user specified timing constraints are met.**

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 22429 | 423403 | 5.30 |
| FF | 13523 | 846806 | 1.60 |
| IO | 868 | 668 | 129.94 |
| BUFG | 1 | 940 | 0.11 |

**Summary**

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

| | |
|---|---|
| **Total On-Chip Power:** | **2.099 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.5°C** |
| Thermal Margin: | 73.5°C (98.0 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

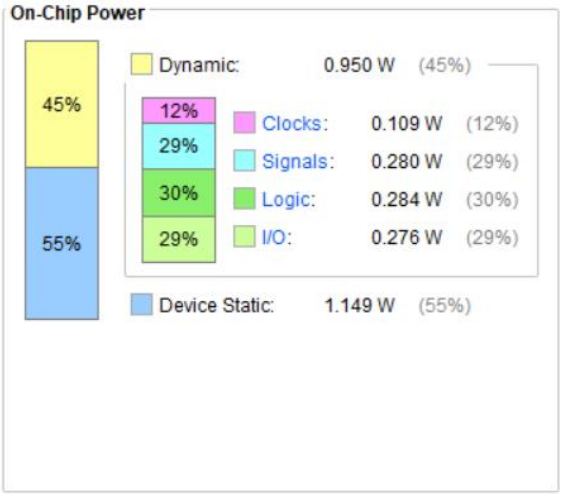



Figure 121: Optimized Forney Implementation Results

## Conventional Forney Results:

## Optimized Forney Latency

- It requires an **initial latency of 8 clock cycles** to generate error magnitudes followed by a **1 clock cycle** pipeline delay.

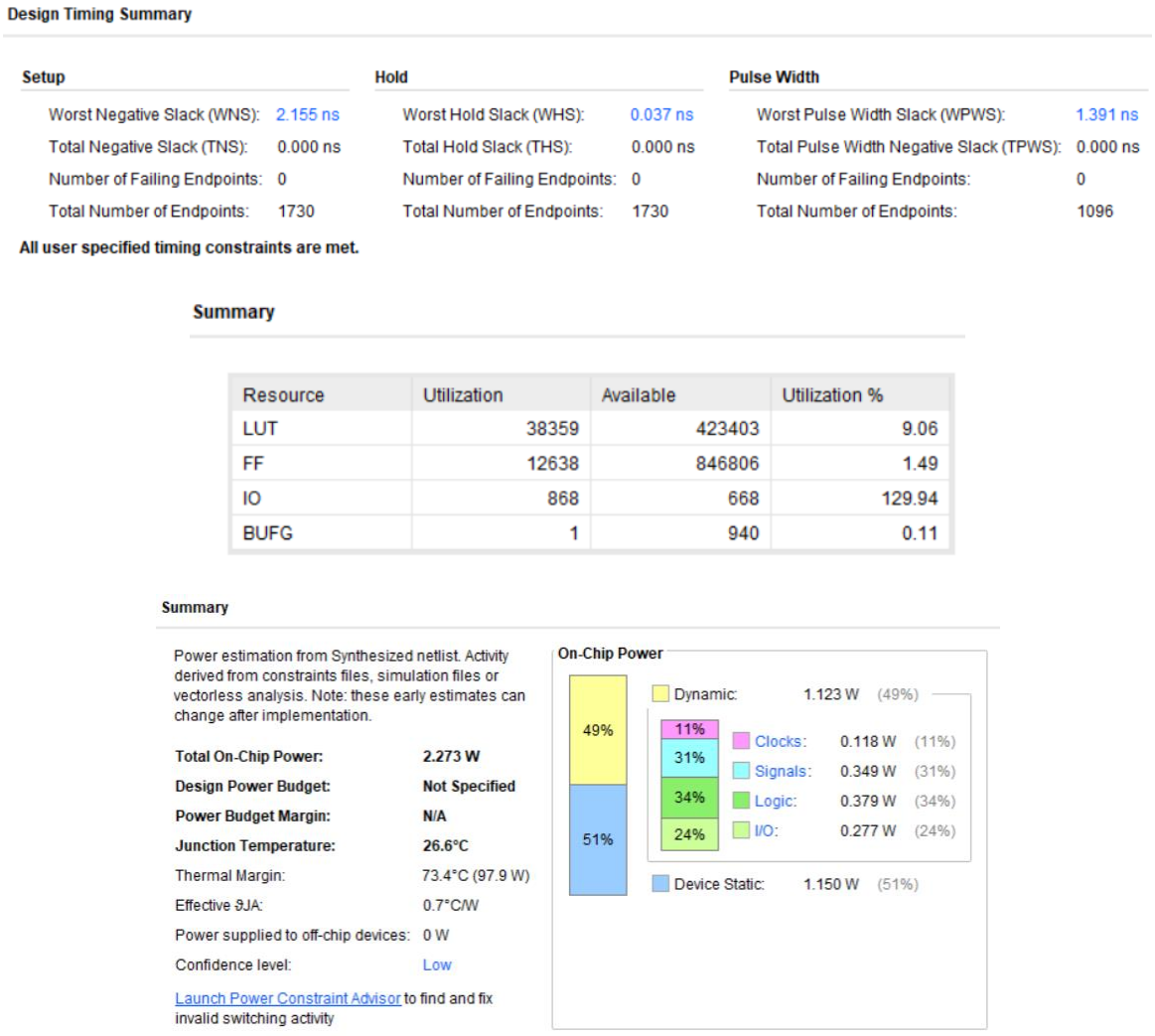



Figure 122: Conventional Implementation Results

Table 32: Forney Algorithm Comparison for different Architecture

| Points of Comparison | Forney Algorithm using Resource Sharing | Conventional Forney |
|---|---|---|
| **Initial Latency** | 22 clock cycle | 8 clock cycle |
| **Setup slack** | 2.324 ns | 2.155 ns |
| **Hold slack** | 0.033 ns | 0.037 ns |
| **Utilization (Area)** | 5.3 % | 9.06 % |
| **Power** | 2.099 watt | 2.273 watt |

### 6.14.6. Error Correction

**Test cases:**

```
ENCODED_FRAME  = 834 927 130 935 647 99 285 560 980 988 161 993 980 497 819 145
RECEIVED_FRAME = 834 927 130 935 647 99 285 560 980 988 161 993 980 497 819 145
HDL_OUT        = 834 927 130 935 647 99 285 560 980 988 161 993 980 497 819 145
MATLAB_MODEL   = 834 927 130 935 647 99 285 560 980 988 161 993 980 497 819 145
```

Figure 123: Green Test case

Generate Codeword and pass it through Encoder (called Encoded_Frame) and pass through channel (called Received_Frame), here free-error channel so HDL_out and MATLAB model was the same.

```
ENCODED_FRAME  = 535 1017 223 108 112 65 414 459 374 781 642 790  955 996 196 142
RECEIVED_FRAME = 535 1017 223 108 112 65 414 459 374 781 642 1001 955 996 196 142 * This Frame Has 1 Errors
HDL_OUT        = 535 1017 223 108 112 65 414 459 374 781 642 790  955 996 196 142
MATLAB_MODEL   = 535 1017 223 108 112 65 414 459 374 781 642 790  955 996 196 142
```

Figure 124: Able to fix Test case

In this case the channel generates less than 16 errors. So, Decoder can detect errors and correct them. As shown in figure (125), in 1$^{st}$ frame 790 sent as encoded symbol and received as 1001 (has error from channel). So, Decoder Detect and back it to 790 again and have successful correction.

```
ENCODED_FRAME  = 791 319 183 347 215 522 928 644 103 400 55 513 442 1021 831 497
RECEIVED_FRAME = 791 319 200 347 215 522 928 644 103 400 55 513 442 822  831 497 * This Frame Has 2 Errors
HDL_OUT        = 791 319 200 347 215 522 928 644 103 400 55 513 442 822  831 497
MATLAB_MODEL   = 791 319 200 347 215 522 928 644 103 400 55 513 442 822  831 497
```

Figure 125: Unable to fix Test case

In this case the channel generates more than 15 but less than 30 (because T = 15, so if 2T errors it will consider wrong codeword). As shown Received_Frame has 2 Errors Symbols (200, 822) and here the Decoder couldn't fix these 2 errors and will bypass them and raise flag indicate Codeword bad.

The following results depend on FPGA Zynq Ultra Scale+ (xczu17eg-ffve1924-3-e)

**Latency:** Correction Block takes 34-clk cycle.

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.329 ns | Worst Hold Slack (WHS): | 0.077 ns | Worst Pulse Width Slack (WPWS): | 1.391 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 677 | Total Number of Endpoints: | 677 | Total Number of Endpoints: | 510 |

**All user specified timing constraints are met.**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 3043 | 423403 | 0.72 |
| FF | 524 | 846806 | 0.06 |
| IO | 645 | 668 | 96.56 |
| BUFG | 1 | 940 | 0.11 |

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

| | |
|---|---|
| **Total On-Chip Power:** | **1.278 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **25.9°C** |
| Thermal Margin: | 74.1°C (98.8 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

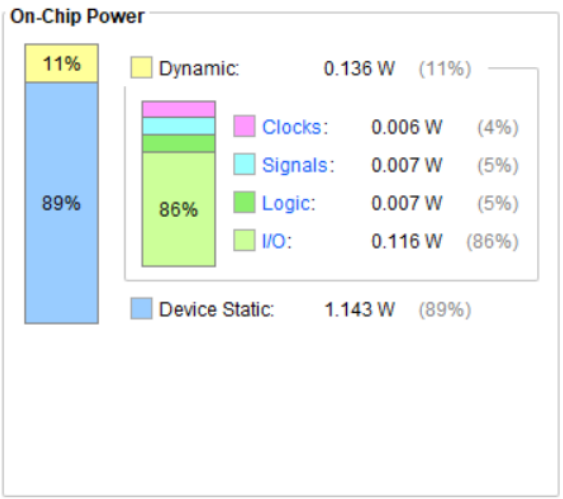



Figure 126: Error Correction Implementation Results

## 6.15. Post-FEC Interleaver

**Latency:**

Interleaver works with CLK_WR (300MHZ) to interleave one codeword (33 Frame of 320-bit) takes 35 cycles.

- **Asynchronous FIFO** works as a bridge between tx domain (300MHZ) and rx domain(400MHZ). So, take only 1-2 cycles (CLK_RD) to get one Frame of Data_rx. So, around 35 cycles of CLK_RD to get Codeword.
- **Asymmetric FIFO** works with CLK_RD (400MHZ) and takes 45 cycles to get out Codeword.
- **Data Formatter** works with CLK_RD (400MHZ) and takes 45 cycles to get out codeword.

After these 4 sub-blocks latency, POST_FEC has pipeline architecture. So, this latency is considered as initial delay.

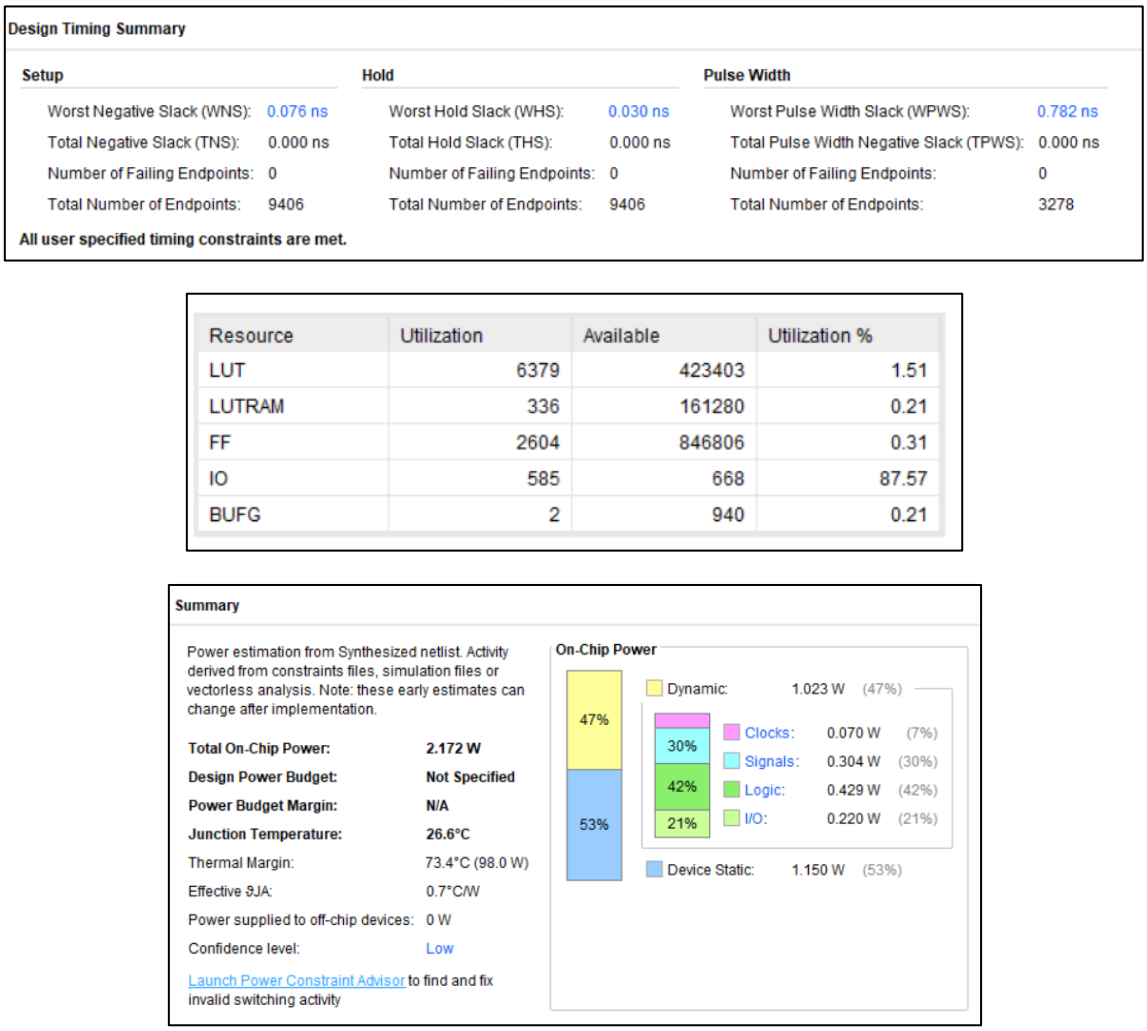
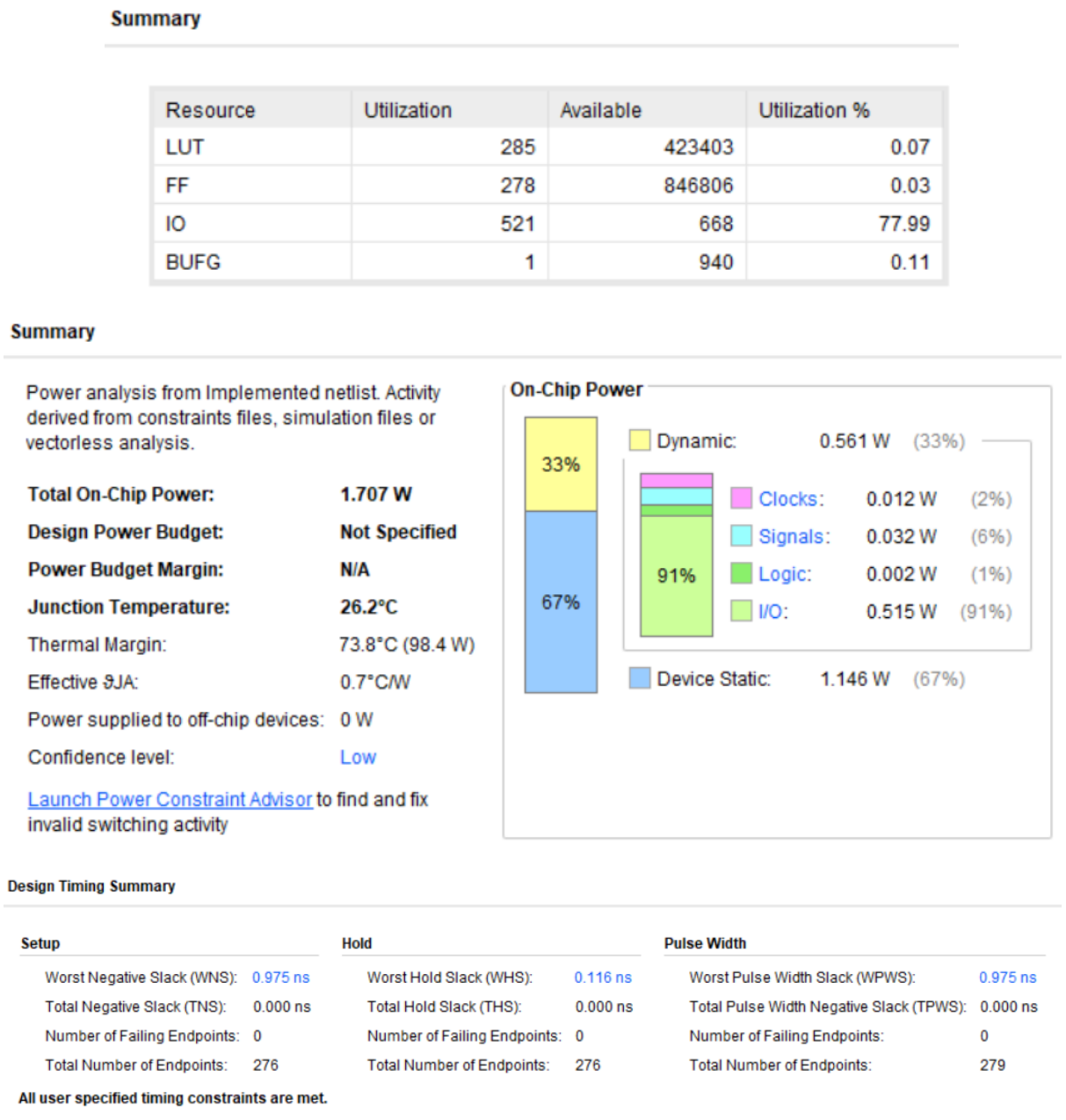



Figure 127: Post-FEC Implementation Results

## 6.16. Alignment Markers Removal

After implementation on the Zynq UltraScale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following summaries are displayed: area utilization, timing performance, and power consumption at **400 MHz.**



**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 285 | 423403 | 0.07 |
| FF | 278 | 846806 | 0.03 |
| IO | 521 | 668 | 77.99 |
| BUFG | 1 | 940 | 0.11 |

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **1.707 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.2°C** |
| Thermal Margin: | 73.8°C (98.4 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

**On-Chip Power**

33%
67%
91%

| | | |
|---|---|---|
| Dynamic: | 0.561 W | (33%) |
| Clocks: | 0.012 W | (2%) |
| Signals: | 0.032 W | (6%) |
| Logic: | 0.002 W | (1%) |
| I/O: | 0.515 W | (91%) |
| Device Static: | 1.146 W | (67%) |

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.975 ns | Worst Hold Slack (WHS): | 0.116 ns | Worst Pulse Width Slack (WPWS): | 0.975 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 276 | Total Number of Endpoints: | 276 | Total Number of Endpoints: | 279 |

**All user specified timing constraints are met.**

Figure 128: Alignment Removal Implementation Results

## 6.17. Descrambler

Descrambler design was implemented on a Zynq Ultra Scale+ FPGA (device: xczu17eg-ffve1924-3-e), operating at a clock frequency of **400 MHz** (corresponding to a 2.5 ns clock period). The resulting hardware metrics — including resource utilization, timing performance, and power consumption — are summarized as follows:

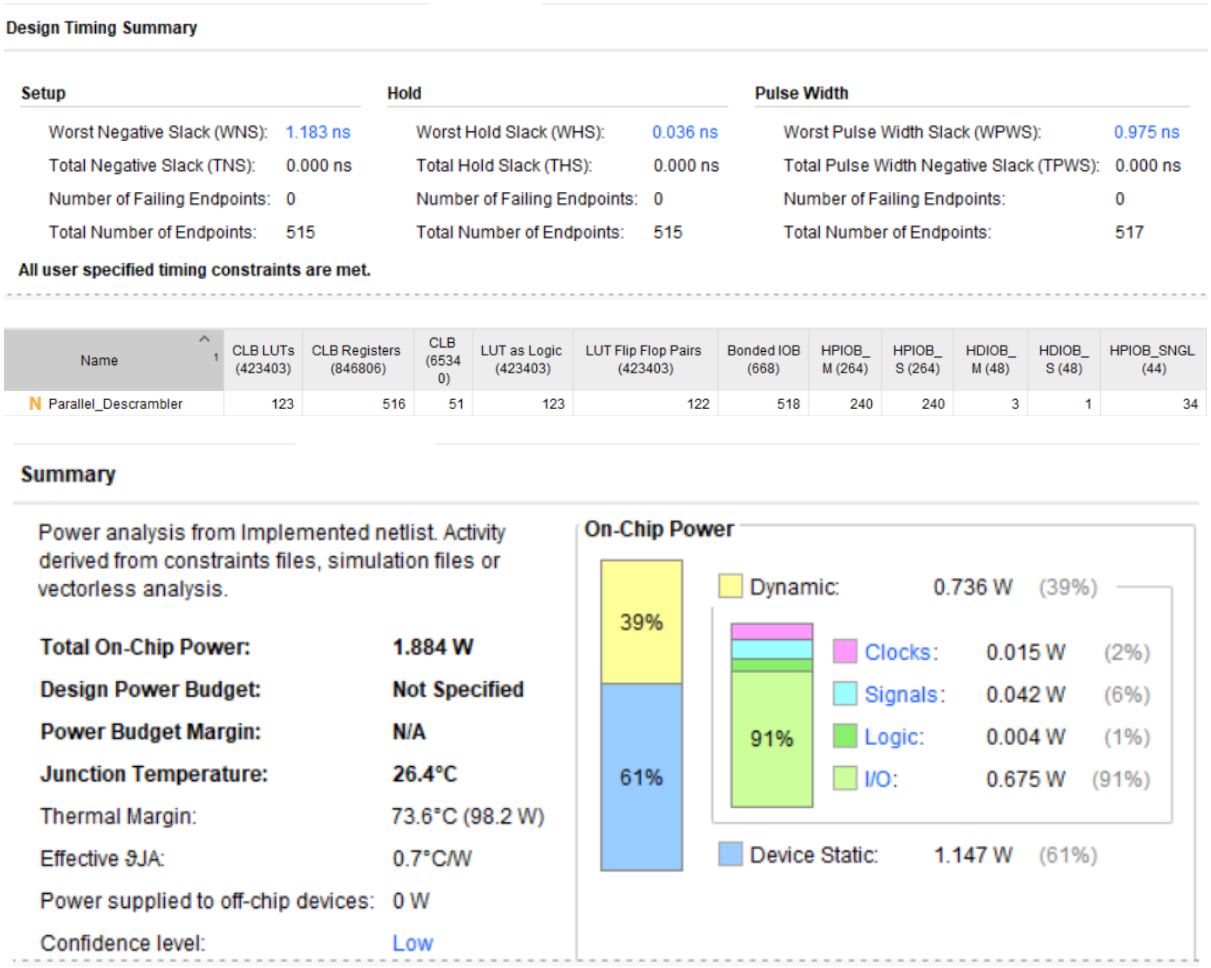



Figure 129: Descrambler Implementation Results

## 6.18. Reverse Transcoder 257B/256B

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at **400 MHZ.**

### Reverse Transcoder Latency

- It takes **1 Clock Cycle** to execute 4 reverse transcoded blocks 66B each.

### Reverse Transcoder Timing

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.042 ns | Worst Hold Slack (WHS): | 0.030 ns | Worst Pulse Width Slack (WPWS): | 0.975 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 525 | Total Number of Endpoints: | 525 | Total Number of Endpoints: | 525 |

**All user specified timing constraints are met.**

### Reverse Transcoder Utilization

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 229 | 423403 | 0.05 |
| FF | 524 | 846806 | 0.06 |
| IO | 526 | 668 | 78.74 |
| BUFG | 1 | 940 | 0.11 |

### Reverse Transcoder Power

**Summary**

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

| | |
|---|---|
| **Total On-Chip Power:** | **1.914 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.4°C** |
| Thermal Margin: | 73.6°C (98.2 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

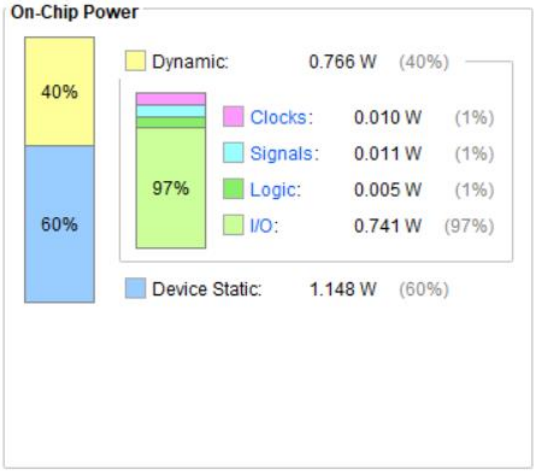



Figure 130: Reverse Transcoder Implementation Results

## 6.19. Decoder 66B/64B

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at **400 MHZ.**

### Decoder 66B/64B Latency

- It requires an **initial latency of 6 clock cycles** to generate RXD, RXC, Signal_Ok, and EOP, followed by a **1 clock cycle** pipeline delay.

### Decoder 66B/64B Timing

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.466 ns | Worst Hold Slack (WHS): | 0.018 ns | Worst Pulse Width Slack (WPWS): | 0.975 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 2806 | Total Number of Endpoints: | 2806 | Total Number of Endpoints: | 1931 |

**All user specified timing constraints are met.**

### Decoder 66B/64B Utilization

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1351 | 423403 | 0.32 |
| FF | 1731 | 846806 | 0.20 |
| IO | 558 | 668 | 83.53 |
| BUFG | 1 | 940 | 0.11 |

### Decoder 66B/64B Power

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **2.115 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26.5°C** |
| Thermal Margin: | 73.5°C (98.0 W) |
| Effective ϑJA: | 0.7°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

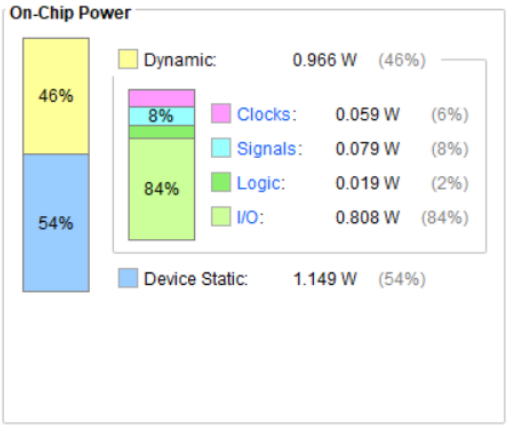



Figure 131: Decoder 66B/64B Implementation Results

## 6.20. Tx Chain

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at **300 MHZ.**

### TX Chain Latency

- It requires an **initial latency of 19 clock cycles** to execute data to the PMA Sublayer, followed by a **1 clock cycle** pipeline delay.

### TX Chain Timing

| Name | Waveform | Period (ns) | Frequency (MHz) |
|---|---|---|---|
| clk | {0.000 1.666} | 3.333 | 300.030 |

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.330 ns | Worst Hold Slack (WHS): | 0.019 ns | Worst Pulse Width Slack (WPWS): | 1.198 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 19019 | Total Number of Endpoints: | 19019 | Total Number of Endpoints: | 11211 |

**All user specified timing constraints are met.**

### TX Chain Utilization

**Summary**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 19729 | 423403 | 4.66 |
| LUTRAM | 1 | 161280 | 0.00 |
| FF | 11209 | 846806 | 1.32 |
| IO | 612 | 668 | 91.62 |
| BUFG | 1 | 940 | 0.11 |

### TX Chain Power

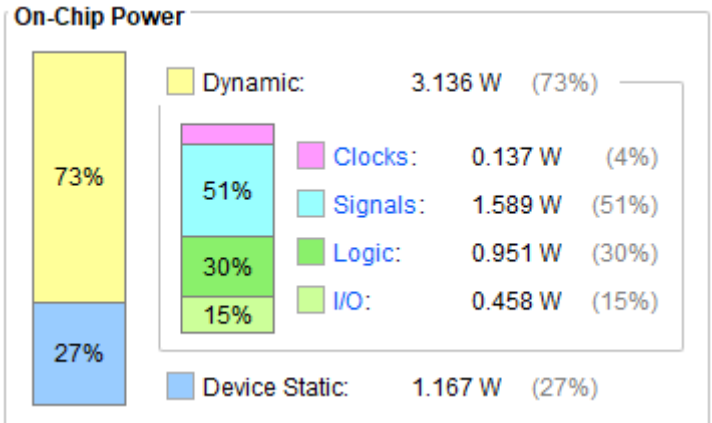



Figure 132: Tx Chain Implementation Results

## 6.21. Rx Chain

- After implementation on the Zynq Ultra Scale+ MPSoC (device number: xczu17eg-ffve1924-3-e), the following results are displayed: Latency, area utilization, timing performance, and power consumption at **clk1 = 300 MHZ & clk2 = 400 MHZ.**

### Rx Chain Latency

- It requires an **initial latency of 133 clock cycles** to execute data to the Reconciliation layer, followed by a **1 clock cycle** pipeline delay.

### Rx Chain Timing

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.000 ns | Worst Hold Slack (WHS): | 0.006 ns | Worst Pulse Width Slack (WPWS): | 0.782 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 72526 | Total Number of Endpoints: | 72526 | Total Number of Endpoints: | 40776 |

**All user specified timing constraints are met.**

### Rx Chain Utilization

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 105420 | 423403 | 24.90 |
| LUTRAM | 40 | 161280 | 0.02 |
| FF | 40690 | 846806 | 4.81 |
| BRAM | 2 | 796 | 0.25 |
| IO | 359 | 668 | 53.74 |
| BUFG | 3 | 940 | 0.32 |

### Rx Chain Power

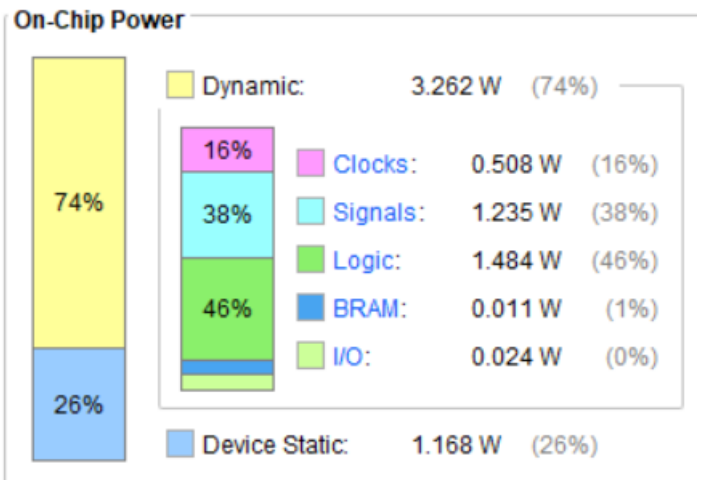



Figure 133: Rx Chain Implementation Results

# 7. Conclusion and Future Work

The implemented Physical coding sublayer System operates correctly when implemented with a **300 MHz** clock speed. The system served as a functional prototype and the design can be developed in ASIC flow to be used in real life products.

Future work can be done to improve our current implementation by supporting the EEE (Energy Efficient Ethernet) feature which can decrease the total power consumed in the system when it operates in Idle mode specified by the IEEE 802.3 standard and other interesting features. The next release of the standard possibly in 2026 will include higher speeds as 800 Gbps/1.6Tbps which might require different methodologies to support the higher throughput requirements.

# 8. References

[1] IEEE, “IEEE Standard for Ethernet,” IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018), pp. 1–7025, Jul. 2022, doi: https://doi.org/10.1109/IEEESTD.2022.9844436.

[2] E. Salem, A. Zekry, and R. M. Tawfeek, “ASIC Design and Implementation of 25 Gigab it Ethernet Transceiver with RS_FEC,” International Journal of Electrical and Electronic Engineering & Telecommunications, pp. 148–155, 2020, doi: https://doi.org/10.18178/ijeetc.9.3.148-155.

[3] H. Kenawy, “Design and Implementation of Configurable Reed Solomon Decoder Using Euclidean Algorithm,” Academia.edu, 2010. https://www.academia.edu/79202076/Design_and_Implementation_of_Configurable_Reed_Solomon_Decoder_Using_Euclidean_Algorithm .

[4] H. Ahmed, H. Salah, T. Elshabrawy, and H. Fahmy, “A low energy high speed Reed-Solomon decoder using Decomposed Inversion less Berlekamp-Massey Algorithm,” pp. 406–409, Nov. 2010, doi: https://doi.org/10.1109/acssc.2010.5757588.

[5] T. K. Matsushima, T. Matsushima, and S. Hirasawa, “Parallel architecture for high-speed Reed-Solomon codec,” vol. 2, pp. 468–473, Nov. 2002, doi: https://doi.org/10.1109/its.1998.718439 .

[6] L. Liu, L. Chi, S.-Y. Huang, and K.-Y. Yeh, “Compiler of Reed–Solomon Codec for 400-Gb/s IEEE 802.3bs Standard,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 8, pp. 2776–2780, Dec. 2022, doi: https://doi.org/10.1109/tcad.2022.3232994 .

[7] S. Lin and D. Costello, Error control coding: fundamentals and applications. Upper Saddle River: Pearson/Prentice-Hall, 2004. doi: https://doi.org/10.1109/tit.2005.844056

[8] S. Wicker and V. Bhargava “Reed-Solomon Codes and Their Applications | IEEE eBooks | IEEE Xplore,” Ieee.org, 2025. https://ieeexplore.ieee.org/book/5264570

[9] D. Knuth.: The art of computer programming, vol. II: Seminumerical algorithms, (Addison-Wesley, MA, 1969) .

[10] B. Rashidi, R. Farashahi, and Sayed Masoud Sayedi, “High-performance and high-speed implementation of polynomial basis Itoh–Tsujii inversion algorithm over GF($2^m$),” IET Information Security, vol. 11, no. 2, pp. 66–77, Apr. 2016, doi: https://doi.org/10.1049/iet-ifs.2015.0461

[11] Nokia Bell Labs Journals & Magazine | Parallel scrambling techniques for digital multiplexers. (1986, October 1). IEEE Xplore. https://ieeexplore.ieee.org/document/6767485

[12] M. haghighat, “nicholl_02_1108,” Scribd, 2025. https://www.scribd.com/document/589708936/nicholl-02-1108 .

[13] K. Parhi, “Eliminating the fanout bottleneck in parallel long BCH encoders,” 2004 IEEE International Conference on Communications (IEEE Cat. No.04CH37577), pp. 2611–2615 Vol.5, 2004, doi: https://doi.org/10.1109/icc.2004.1313004 .

[14] X. Zhang, “High-Speed and Low-Complexity Parallel Long BCH Encoder,” 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Oct. 2020, doi: https://doi.org/10.1109/iscas45731.2020.9180783.

[15] N. Jun, W. Zhi-Gong, H. Sheng, and N. Jie, “Optimized design for high-speed parallel BCH encoder,” vol. 22, pp. 97–100, Sep. 2005, doi: https://doi.org/10.1109/iwvdvt.2005.1504560 .

[16] Y. Lee, I. Park, and H. Yoo, “Area-optimized Syndrome Calculation for Reed Solomon Decoder,” JSTS Journal of Semiconductor Technology and Science, vol. 18, no. 5, pp. 609–615, Oct. 2018, doi: https://doi.org/10.5573/jsts.2018.18.5.609 .

[17] M. Potkonjak, “Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms Subexpression for Exploring Elimination,” Common IEEE Trans. Computer-Aided Design Integr. Circuits Syst., vol. 15, no. 2, pp. 151-165, Feb. 1996.

[18] I. 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, Available: https://www.jstor.org/stable/2098968

[19] J. Massey, “Shift-register synthesis and BCH decoding,” IEEE Transactions on Information Theory, vol. 15, no. 1, pp. 122–127, Jan. 1969, doi: https://doi.org/10.1109/tit.1969.1054260.

[20] N. R.-C. Chang and C. B. Shung, “A (208,192;8) Reed-Solomon decoder for DVD application,” vol. 2, pp. 957–960, Nov. 2002, doi: https://doi.org/10.1109/icc.1998.685153.

[21] J. Park, K. Lee, C. Choi, and H. Lee, “High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm and Its Folded Architecture,” JSTS:Journal of Semiconductor Technology and Science, vol. 10, no. 3, pp. 193–202, Sep. 2010, doi: https://doi.org/10.5573/jsts.2010.10.3.193.

[22] H. Lee, M. Yu, and L. Song, “VLSI design of Reed-Solomon decoder architectures,” 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353), vol. 5, pp. 705–708, 2025, doi: https://doi.org/10.1109/iscas.2000.857589.

[23] D. Sarwate and N. Shanbhag, "High-speed architectures for Reed-Solomon decoders," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 5, pp. 641-655, Oct. 2001, doi: 10.1109/92.953498.

[24] Z. Qinglun, L. Chunyan, and W. Yong, “Hardware implementation of 64B/66B encoder/decoder for 10-Gigabit ethernet,” 2006 International Conference on Communication Technology, Nov. 2006. doi:10.1109/icct.2006.341665

[25] B. Lee and S. Kim, Scrambling Techniques for Digital Transmission. Springer Science & Business Media, 2012. https://doi.org/10.1007/978-1-4471-3231-8?nosfx=y

[26] C. Yang, C. Zhang, W. Xie, Z. Wang and X. Xie, “Design of the critical controller in physical coding sublayer based on the 10Gbase-KR protocol,” vol. 37, pp. 1–2, Oct. 2017, doi: https://doi.org/10.1109/edssc.2017.8126494.

[27] N. Hu, N. Wang, N. Zhang, and N. Xiao, “Low Complexity Parallel Chien Search Architecture for RS Decoder,” 1993 IEEE International Symposium on Circuits and Systems, Jul. 2005, doi: https://doi.org/10.1109/iscas.2005.1464594.

[28] Y. Lee, H. Yoo, and I. Park, “Low-Complexity Parallel Chien Search Structure Using Two-Dimensional Optimization,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 58, no. 8, pp. 522–526, Aug. 2011, doi:https://doi.org/10.1109/tcsii.2011.2158709