View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M University

EFFICIENT DECODER FOR OPTICAL TRANSPORT NETWORKS ACHIEVING NEAR CAPACITY PERFORMANCE

A Thesis

by

SUSHRUT KAUL

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Krishna Narayanan
Committee Members,	Gwan Choi
	Scott Miller
	Duncan Walker
Head of Department,	Miroslav M. Begovic

August 2019

Major Subject: Electrical and Computer Engineering

Copyright 2019 Sushrut Kaul

ABSTRACT

Today's optical transport networks (OTNs) support a plethora of services such as video streaming, cloud computing, social networking and many more. To make such a wide assortment of services possible, a tremendous amount of data needs to be carried over the internet backbone supported by these optical transport networks. In order to cope with this increase in traffic, data rate on OTNs has increased significantly. Product codes (PC) are a class of codes that provide good coding gain at reasonable decoding complexity and, hence, have been a popular choice for OTNs in recent times.

The key goal of this thesis is to implement a decoder for a Product Code (PC) on a Virtex-7 Field Programmable Gate Array(FPGA). The product code of choice for this project is based on a (1023,993) BCH code as a component code. The conventional decoder for BCH codes has a computationally expensive step for finding the roots of error locator polynomial. The BCH decoder implemented as a part of this project is optimized to speed up the decoding process while at the same time also simplifying the hardware complexity of the design. The implementation is parallelized and pipelined to achieve high throughputs. This provides a hardware platform to evaluate the performance of product codes at low bit error rates that is infeasible using software simulations.

DEDICATION

To my mother for her ongoing love and support and to my father who could not see this thesis completed.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Krishna Narayanan, for guiding and supporting me, not only for this project but during the entire course of my graduate studies. I am grateful to Dr. Gwan Choi for extending a helping hand and sharing his expertise with me. I would also like to thank Dr. Duncan Walker and Dr. Scott Miller for serving on my thesis committee.

I would also like to express my gratitude towards all the current and past members of my research group for their support and valuable inputs.

I would like to extend a special thanks to my friends and family for their constant support and motivation. I would also like to thank Dr. Parimal Saraf for his unwavering belief in me during the course of my graduate studies.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Krishna Narayanan, Dr. Scott Miller and Dr. Gwan Choi of the Department of Electrical and Computer Engineering and Dr. Duncan Walker of the Department of Computer Science and Engineering.

The idea for the implementation of product code decoder with an optimized BCH code decoder as the component code decoder was given by Dr. Krishna Narayanan. All other work conducted for the thesis was completed by the student independently.

Funding Sources

This project was supported by National Science Foundation(NSF) Grant 1611285 titled "Advanced Coding Techniques for Next Generation Optical Communications"

NOMENCLATURE

OTN	Optical Transport Networks
LDPC	Low Density Parity Check
PC	Product Codes
FPGA	Field Programmable Gate Array
IP	Intellectual Property
BCH	Bose Chaudhari Hocquenghem
XST	Xilinx Synthesis Tool
HDL	Hardware Description Language
DUT	Design Under Test
RTL	Register Transfer Level
CLB	Configurable Logic Block
MT	Mersenne Twister

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

LIST OF TABLES xi

1. INTRODUCTION, MOTIVATION AND OBJECTIVE 1

    1.1 Thesis contributions 2

2. BACKGROUND AND RELATED WORK 4

    2.1 Introduction to product codes 4
        2.1.1 Product code construction 5
        2.1.2 Minimum distance and code rate for product codes 5
    2.2 Iterative decoder for product codes 7
        2.2.1 Product code decoding example 7
    2.3 Stopping set conditions 9
    2.4 Introduction to BCH codes 11
        2.4.1 Algebraic decoding of BCH codes 11
        2.4.2 Syndrome computation 13
        2.4.3 Error locator polynomial 14
        2.4.4 Solving error locator polynomials of degree 2 and 3 14
            2.4.4.1 Error locator polynomials of degree 2 14
            2.4.4.2 Error locator polynomials of degree 3 16

3. IMPLEMENTATION DETAILS 17

    3.1 Field Programmable Gate Array 17
        3.1.1 FPGA design flow 18

3.2 BCH implementation details 19
3.2.1 Data generator block 20
3.2.2 Syndrome computer block 21
3.2.2.1 Example of syndrome computation and simulation diagram 23
3.2.3 Error locator finder block 24
3.2.3.1 Example of finding the coefficients of error locator polynomial 25
3.2.4 Error locator solver block 28
3.2.4.1 Second degree solver 28
3.2.4.2 Third degree solver 28
3.2.5 Complete BCH decoder implementation 29
3.2.6 BCH decoder performance comparisons 32
3.3 Product code implementation details 35
3.3.1 Product code 10000 feet view 35
3.3.2 Product code detailed architecture 35

4. SUMMARY AND CONCLUSIONS 39

4.1 Challenges 39
4.2 Future work 39

REFERENCES 41

LIST OF FIGURES

FIGURE Page

2.1 The Product Code construction 4

2.2 A product codeword with (7,4,3) BCH code as the component code. The red dots represent the errors introduced due to noise. 8

2.3 Product codeword after one round of row decoding. The red dots represent the errors that could not be corrected. The blue dots shown in the figure represent the errors that have been corrected by the decoder. 8

2.4 Product codeword after one round of column decoding. The blue dots represent the errors that have been corrected by the decoder. 9

2.5 Stopping set condition. The figure depicts a minimum stopping set for the product code based on (7,4,3) BCH code. 10

2.6 BCH decoder module. 12

3.1 Generic FPGA architecture. 17

3.2 BCH decoder design flow. 19

3.3 BCH decoder implemented design. The figure depicts a high level design for the BCH decoder implemented in Verilog. 20

3.4 Data Generator simulated design. This figure shows the simulation output for data generator block. 21

3.5 Syndrome computation architecture. The received vector is divided into 33 streams of 31 bits each. 22

3.6 Simulation of syndrome computation block 24

3.7 Schematic for error locator finder. This figure shows elaborated design for the error locator finder block in Vivado 25

3.8 Format Conversion from polynomial format to exponential format in Matlab 26

3.9 $\Lambda 2$ computation in Matlab. 27

3.10 Vivado simulation for error locator finder block. The simulation shows the coefficients of the error locator polynomial. 28

3.11 Vivado simulation for error locator solver block. 29

3.12 Block diagram of the complete BCH decoder. 30

3.13 Utlization report of a BCH decoder. 31

3.14 Power utilization of a BCH decoder. 32

3.15 Timing summary of a BCH decoder. 32

3.16 BCH decoder in Matlab. This decoder uses the in-built gftuple function. 33

3.17 BCH decoder in Matlab. This decoder does not use the in-built gftuple function. 33

3.18 Bit Error Rate(BER) vs cross over probability(p). 34

3.19 10000 ft view of product code decoder architecture. 35

3.20 Detailed product code decoder architecture. 36

3.21 Product code decoder state diagram. 37

3.22 Product code decoder utilization report. 38

LIST OF TABLES

TABLE Page

2.1 A_i for standard basis. Here, A_i is one root of equation $x^2 + x + \alpha^i = 0$ where $i \in 0, 1, 2, ..., 9$ 15

1. INTRODUCTION, MOTIVATION AND OBJECTIVE

Modern optical transport networks (OTNs) support a wide array of applications ranging from video streaming and cloud computing to social networking applications like Whatsapp and Facebook. The popularity of these applications has led to an unprecedented increase in the amount of data being carried over the internet backbone. Internet protocol (IP) traffic has seen a five fold increase from 2011. CISCO, a leading networking equipment company, predicts that global consumer IP traffic will reach 233 exabytes/month by 2021. In order to cope with this increase in traffic, the data rate on OTNs that constitute the internet backbone has increased commensurately.

With increase in the data rates, effects of optical impairments such as non-linearities, chromatic dispersion, noise due to amplified spontaneous emission become more pronounced. To ensure reliable data delivery at such high data rates is a challenging task and good forward error correction schemes that deliver large coding gains with minimum redundancy are needed.

Over the past few decades, a number of coding schemes like turbo codes, Low Density Parity Check (LDPC) codes and their spatially-coupled cousins have been shown to achieve near capacity performance for several applications. However, these existing coding schemes are not very attractive for optical transport networks because of the differences in the inherent nature of OTNs from other applications. A few such differences are noted below :

- The decoders for next generation OTNs must achieve throughput of the order of 100 Gbps - 10 Tbps.
- The target code rate is 0.942 and the bit error rate is 10^{-15} which is very small. This requires that the coding scheme of choice should maintain large gains at very high rates and should not exhibit floors above 10^{-15}.
- At the mentioned rates, memory bandwidth is a big bottleneck in the implementation of the existing capacity achieving coding schemes. For example, a high-rate message passing decoder needs considerable memory bandwidth to pass messages from one iteration to another.

- Soft decision decoding is not commonplace in OTNs as soft decision decoders do not work well with hard decision channels. The trend is to use hard decision decoders at very high rates with very few bits of soft decision at lower code rates.

As is evident from the above mentioned constraints, popular coding solutions such as LDPC, polar codes, turbo codes are not attractive solutions to OTNs. It turns out that Product Codes (PC) are well suited for Optical Transport Networks. Product codes are a class of codes that provide good coding gain at reasonable decoding complexity and hence, have been a popular choice in recent times. FPGA based implementation provides a hardware platform to evaluate the performance of product codes at low bit error rates that is infeasible using software simulations. Inherent parallelism of hardware designs allows us to instantiate several decoders in parallel as each row/column can be thought off as an independent component codeword and can be decoded in parallel with several other component codewords. The number of such instantiations possible is directly proportional to the complexity of the individual component decoder. A part of this work is also dedicated to implementing an efficient algorithm for Bose Chaudhuri Hocquenghem(BCH) codes decoding which avoids the computationally complex Chein Search algorithm.

1.1 Thesis contributions

The main contributions of this work can be summarized as follows :

- Designing a hardware based architecture for an error correction coding scheme based on product codes that can provide high throughput of the order of 10 Gbps.

- Implementing a highly pipelined and fast decoder for the underlying BCH code with AXI complaint custom IPs.

- Implementing a decoder for a $(1023, 993)$ turbo product-code on a Virtex-7 FPGA after functionally verifying each of the custom IPs using a simulator.

- Instantiating multiple BCH decoders to improve the overall throughput of the design. It is possible to instantiate 50-60 such decoder blocks for faster and parallel product code

decoding.

The implementation of the mentioned coding scheme on FPGA enables us to study the error floors which are hard to study using software simulations. The hardware implementation provides significant speed up over the software based implementations saving a lot of simulation time as well.

2. BACKGROUND AND RELATED WORK

2.1 Introduction to product codes

Product codes were introduced by Elias(1954) [1]. Given two linear block codes, C_1 and C_2, with dimensions (N_1, K_1) and (N_2, K_2) respectively, a product code is a :

- Collection of codewords where each codeword is a matrix.
- The rows of this matrix are codewords of the component code C_1.
- The columns of this matrix are codewords of the component code C_2.

Essentially, a product code can be thought of as a transformation from K_1K_2 information bits to N_1N_2 coded bits. The product code construction is shown in 2.1.

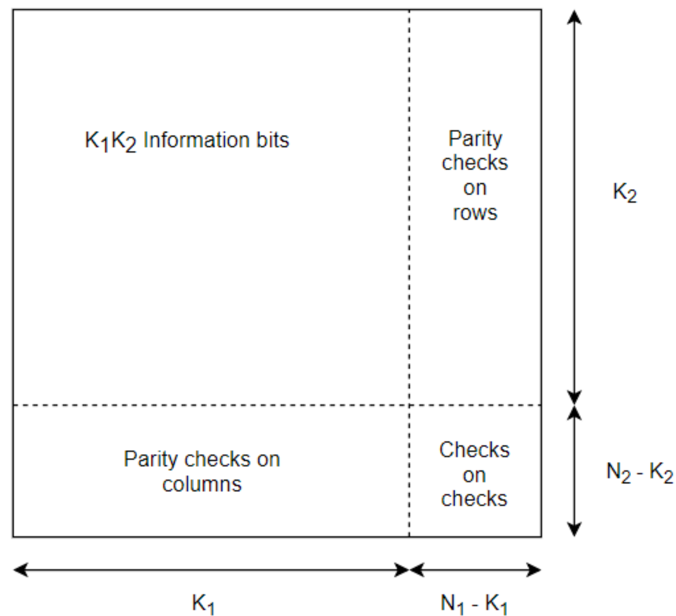

Figure 2.1: The product code construction. Rows are codewords from (N_1, K_1) code C_1 and columns are codewords from (N_2, K_2) code C_2.

2.1.1 Product code construction

Given two systematic linear block codes, $C_1\ (N_1, K_1)$ and $C_2\ (N_2, K_2)$, a product code,P, with C_1 and C_2 as the component codes is a (N_1N_2, K_1K_2) code. A codeword of product code, P, can be constructed in the following way :

- The information bits are arranged in form of a sub array with K_2 rows and K_1 columns.
- Each row is encoded using the code C_1 and each column is encoded using the code C_2. There are $(N_1 - K_1)$ parity checks on rows formed using code C_1 and $(N_2 - K_2)$ parity check on columns form using code C_2 as shown in figure 2.1.
- The bottom right rectangle contains parity on parity checks. It can be formed by encoding the rows to get the parity checks on the rows using code C_1 and then encoding these checks on row parities using code C_2. Another way to get these parity on parity checks is to encode the columns first using code C_2 and then encoding the checks on the column parities using code C_1. Either method will lead to the same set of $(N_1 - K_1)(N_2 - K_2)$ checks.

2.1.2 Minimum distance and code rate for product codes

The minimum distance, d_{min}, of a block code, C is the minimum Hamming distance between two distinct codewords of C.

$$d_{min} = min\{d_H(c_1, c_2) :\ c_1,\ c_2\ \in C\, and\, c_1 \neq c_2\}$$

where d_H is the Hamming distance. The minimum distance of a product code, d^*, can be derived from the minimum distance of the component codewords. Given two linear block codes, C_1 and C_2, with minimum distances d_1 and d_2 respectively, it can be shown that the minimum distance of the product code with C_1 and C_2 as the component codes is given by,

$$\boxed{d^* = d_1 d_2} \tag{2.1}$$

The component codes, C_1 and C_2, selected for this work are both (1023,993) BCH codes having a minimum distance,d , of 7. The minimum distance of the product code with (1023,993) BCH code as the component code is given by,

$$\boxed{d^* = d_1 d_2 = d^2 = 49} \tag{2.2}$$

Code rate is a very useful parameter when quantifying the redundancy in a coded stream. The code rate of a linear block code, R, is the ratio of information bits to the coded bits given by

$$R = K/N$$

where K is the number of information bits and N is the length of code word obtained after encoding the K information bits. For product codes, the information bits K_P is given by K_1K_2. The number of coded bits, N_P is given by N_1N_2. The code rate for the product code, R_P is given by

$$R_P = K_P/N_P.$$

For this work, C_1 and C_2 are both (1023,993) codes. So, the rate of the chosen coding scheme is given by,

$$\boxed{R_P = (993)^2/(1023)^2 = 0.942} \tag{2.3}$$

This is in line with the requirements of next generation OTNs.

2.2 Iterative decoder for product codes

Product codes can be decoded using an iterative decoder. The decoding process involves the following steps :

- Decode rows of the received matrix with the decoder for the row code, C_1. Correct the errors in each row and continue the decoding process. If a decoding failure occurs at any row, move to the next row.
- Decode columns of the received matrix with the decoder for the column code, C_2. Correct the errors in each row and continue the decoding process. If a decoding failure occurs at any column, move to the next column.
- Repeat the cascaded decoding process for a fixed number of iterations.

2.2.1 Product code decoding example

Consider the decoding process for a product code with (7,4,3) BCH code as the component code. The (7,4,3) BCH code represents 4 bits of information using 7 coded bits. The third parameter in the specification of the coding scheme (7,4,3) is the minimum distance of the code, d_{min}. We can use this minimum distance to give an upper bound on the error correction capability, t, of the code. The error correction capability is given by

$$t \leq (d_{min} - 1)/2$$

(7,4,3) BCH code has an error correction capability, t, of 1 which means that it can correct upto one bit error in the rows or columns. Figure 2.2 shows a sample received product code word after noise has been added. The complete decoding process is shown in the following figures.

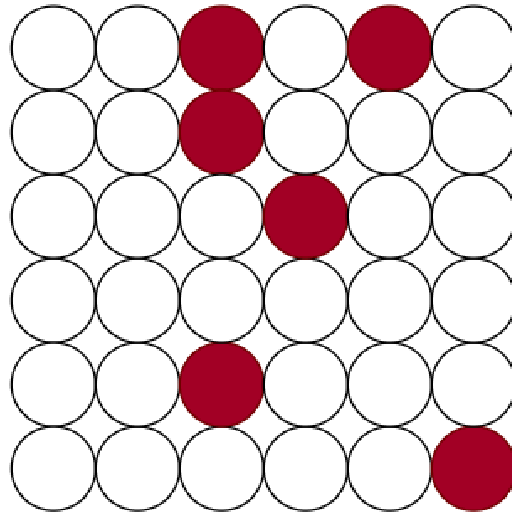

Figure 2.2: A product codeword with (7,4,3) BCH code as the component code. The red dots represent the errors introduced due to noise.

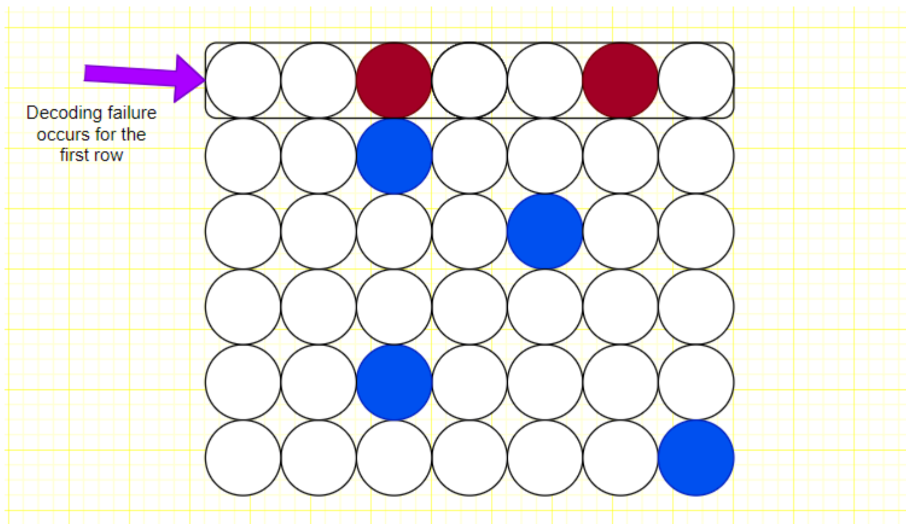

Figure 2.3: Product codeword after one round of row decoding. The red dots represent the errors that could not be corrected. The blue dots shown in the figure represent the errors that have been corrected by the decoder.

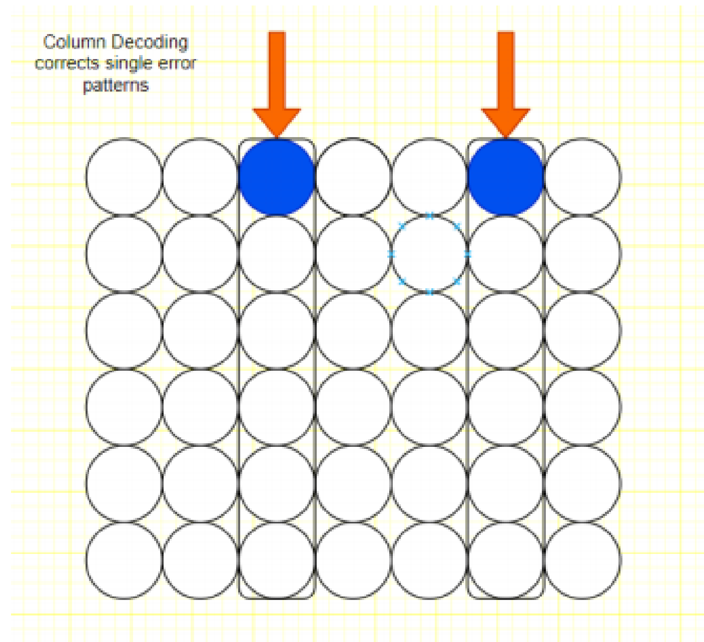

Figure 2.4: Product codeword after one round of column decoding. The blue dots represent the errors that have been corrected by the decoder.

The first row has 2 errors which is more than the error correction capability of the component code. So, this leads to a decoding failure. However, all the single errors are corrected by the decoder. This is shown in Fig 2.3. The blue circles in the figure represents the bit errors that the decoder has been successful in correcting. The first row, however, still contains two red circles representing the bit errors which could not be corrected by the decoder as a decoding failure has occurred.

After decoding all the rows of the matrix, we decode all the columns. As we can see in Fig 2.3 , we have two single error code words which can be corrected by the decoder. The column decoding process is shown in Fig 2.4

2.3 Stopping set conditions

A stopping set is an arrangement of errors such that each row has x errors and each column has x errors such that $x \geq t$, where t is the error correction capability of the code. Consider a product

code, P, with component code C having an error correction capability t. The cardinality of the minimum stopping set is given by

$$s_{min} = (t+1)^2.$$

For our example, component code has an error correction capability of 1. So, the minimum stopping set consists of 4 errors. This is depicted in Fig 2.5

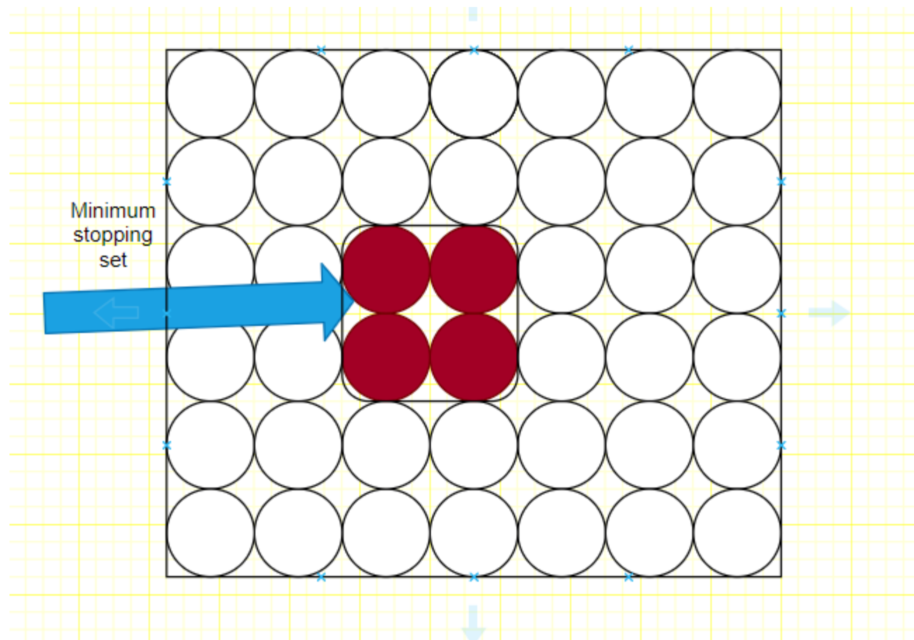

Figure 2.5: Stopping set condition. The figure depicts a minimum stopping set for the product code based on (7,4,3) BCH code.

2.4 Introduction to BCH codes

The decoder for product codes relies on the component code decoder to decode individual rows and columns. To achieve high throughput for the product code decoder, it is imperative to optimize the component code decoder. The optimizations need to be done in such a way that a balance is maintained between the decoding latency and the area requirements. Optimizing the area is crucial as lesser resource utilization would allow us to instantiate multiple such decoder modules which can then work in parallel. Usually, decoding latency and area have an inverse relationship and this trade-off has to be exploited carefully. In this work, we have selected a (1023,993) BCH code as the component code. This section is dedicated to a brief discussion and introduction of BCH codes.

BCH codes are a class of cyclic error correction codes. A BCH code of length n which can correct atleast t errors can be constructed as follows :

- Find the smallest m such that $\mathbb{GF}(q^m)$ has a primitive n^{th} root of unity β.
- Find the minimal polynomial in $\mathbb{GF}(q)$ for $2t$ consecutive powers of β i.e $\beta^b, \beta^{b+1}, \cdots, \beta^{b+2t-1}$. If b is 1, the BCH code is said to be narrow-sense.
- The least common multiple of these minimal polynomials gives the generator polynomial $g(x)$ for the code.

Two Galois fields are involved in the construction of BCH codes. The coefficients of the generator polynomial are derived from the smaller field $\mathbb{GF}(q)$. The roots of the generator polynomial are present in the larger field $\mathbb{GF}(q^m)$. Binary BCH codes are the codes where the coefficients of the generator polynomial and the elements of the codeword are binary in nature. For this work, we will be working with (1023,993) binary BCH code. The primitive polynomial for this BCH code is $x^{10} + x^3 + 1 = 0$.

2.4.1 Algebraic decoding of BCH codes

Algebraic decoding of BCH codes follows the following steps :

- The first step in algebraic decoding of BCH codes involves finding the syndromes from the received vector. A codeword from the BCH code will always have zero syndromes. Non zero syndrome values is an indication that an error has occured.

- The second step involves finding the error locator polynomial. The roots of the error locator polynomial provide an indication of where the errors are. Some of the popular algorithms for finding the error locator polynomial are Peterson's algorithm, Berlekamp and Massey algorithm, Euclidean algorithm etc.

- The third step is to find the roots of the error locator polynomial. This is usually done using the Chien search algorithm. Chien search involves an exhaustive search over the elements of the field to determine the roots. This method is computationally very expensive for larger fields. The decoder implemented for this project follows the approach proposed in [1].

- The final step involves inverting the roots of the elements in the field which gives the error locations in the received vector.

An outline of a typical BCH decoder is shown in 2.6.

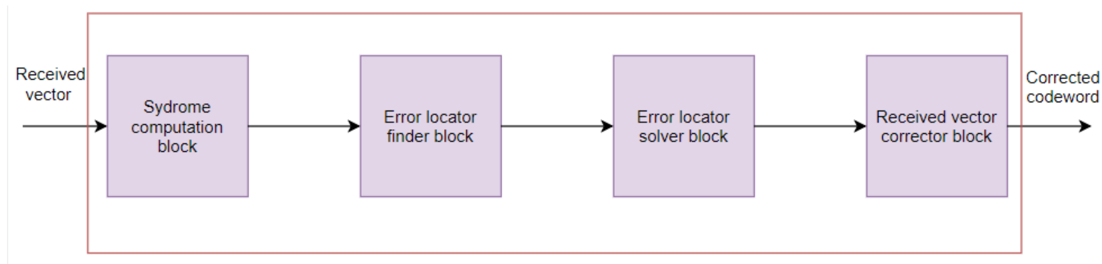

Figure 2.6: BCH decoder module.

2.4.2 Syndrome computation

For a BCH code, the generator polynomial, $g(x)$ has $2t$ roots namely $\alpha, \alpha^2, \alpha^3 \cdots \alpha^{2t}$ where α is a primitive element of the field. If follows that,

$$g(\alpha) = g(\alpha^2) = g(\alpha^3) = \cdots = g(\alpha^{2t}) = 0. \tag{2.4}$$

Any codeword from the BCH code can be expressed as,

$$c(x) = m(x)g(x). \tag{2.5}$$

This shows that for any codeword from the BCH code,

$$c(\alpha) = c(\alpha^2) = \cdots = c(\alpha^{2t}) = 0. \tag{2.6}$$

Define the received polynomial, $r(x) = c(x) + e(x)$, where $e(x) = \sum_{j=0}^{n-1} e_j x^j$. For a received polynomial $r(x)$, the kth syndrome S_k is defined as $S_k \coloneqq r(\alpha^k)$ and is given by,

$$S_k = r(\alpha^k) = c(\alpha^k) + e(\alpha^k) = e(\alpha^k) = \sum_{j=0}^{n-1} e_j \alpha^{jk},\ k = 1, 2, \cdots, 2t. \tag{2.7}$$

Suppose the received vector, r, has v errors at locations $i_1, i_2, i_3, \cdots, i_v$. Since we are dealing with binary BCH codes, the error values at these locations will be 1. This implies,

$$S_k = \sum_{m=1}^{v} (\alpha^{i_m})^k. \tag{2.8}$$

Let $Y_m = \alpha^{i_m}$.Then,

$$S_k = \sum_{m=1}^{v} Y_m^k,\ k = 1, 2, \cdots, 2t \tag{2.9}$$

Here $Y_1, Y_2, \cdots, Y_v$ are called error locators.

2.4.3 Error locator polynomial

The error locator polynomial, $\Lambda(x)$, is defined as

$$\boxed{\Lambda(x) := \prod_{m=1}^{v}(1 - Y_m x) = \Lambda_v x^v + \Lambda_{v-1}x^{v-1} + \cdots + \Lambda_1 x + \Lambda_0} \tag{2.10}$$

Clearly, $x = Y_m^{-1}$ is the root of the error locator polynomial which implies that the roots of the error locator polynomial are reciprocals of the error locators.

Peterson's algorithm provides a way to compute the error locator polynomial using the syndromes. For small number of errors, we can provide explicit formulas for coefficients of the error locator polynomial. Since (1023,993) BCH code is a 3 error correcting code, we list here the formulas for the coefficients of error locator polynomial for 3 error correcting BCH code.

$$\boxed{\Lambda_1 = S_1,\ \Lambda_2 = (S_1^2 S_3 + S_5)/(S_1^3 + S_3),\ \Lambda_3 = (S1^3 + S_3) + S_1\Lambda_2} \tag{2.11}$$

2.4.4 Solving error locator polynomials of degree 2 and 3

As mentioned in the previous section, Chien search algorithm which involves an exhaustive search is computationally complex for large fields. For a field of size q, Chien search requires q polynomial evaluations. The approach followed in this work uses the closed form solutions presented in [2] to find the roots of the error locator polynomial.

2.4.4.1 Error locator polynomials of degree 2

An error locator polynomial of degree 2 is of the form,

$$\boxed{\Lambda(y) = \Lambda_2 y^2 + \Lambda_1 y + \Lambda_0} \tag{2.12}$$

This second degree equation can be converted to the form $x^2 + x + k = 0$ by making the substitution $y = (\Lambda_1/\Lambda_2)x$. For any k in $\mathbb{GF}(2^m)$, one can construct the solution from the solutions on any $\mathbb{GF}(2)$-basis for $\mathbb{GF}(2^m)$. This is because $x^2 + x$ is a $\mathbb{GF}(2)$-linear function on $\mathbb{GF}(2^m)$ as

$x^2 + x + j = 0$ and $y^2 + y + k = 0$ together imply that,

$$(x + y)^2 + (x + y) + (j + k) = 0 \tag{2.13}$$

This is helpful because we can decompose k in terms of a basis for the field and compute the root of the polynomial as the linear combination of the roots pre- computed for the basis elements and stored in a look-up table. This greatly reduces the complexity of implementing the standard solver for second degree error locator polynomials. During the design phase, we choose the standard basis for $\mathbb{GF}(2^{10})$ which is 1,α,α^2,$\cdots$, α^9. Let A_i be the root of the equation $x^2 + x + \alpha^i = 0$ for i ranging from 0 to 9. These roots are fixed constants and can be computed during the design phase. It is important to note that for some values of k, roots may not be in $\mathbb{GF}(2^m)$. In general, roots are present in $\mathbb{GF}(2^m)$ if and only if trace-2, $T_2(k)$ is 0 where $T_2(k)$ is defined as,

$$T_2(k) := \sum_{i=0}^{9} k^{2^i} \tag{2.14}$$

It can be proved that exactly half of the k values satisfy this equation. Table 2.1 shows the values of one root A_i for standard basis in $\mathbb{GF}(2^m)$.

A_i	Element
A_0	α^{341}
A_1	α^{255}
A_2	α^{515}
A_3	α^{549}
A_4	α^{1020}
A_5	α^{903}
A_6	α^{75}
A_7	N/A
A_8	α^{14}
A_9	α^{607}

Table 2.1: A_i for standard basis. Here, A_i is one root of equation $x^2 + x + \alpha^i = 0$ where $i \in 0, 1, 2, ..., 9$

Given one root of the equation $x^2 + x + k = 0$, finding the other root of this equation is trivial. If A_{r_1} is one root of the equation, then $A_{r_1} + 1$ also satisfies the equation. This is because,

$$(A_{r_1} + 1)^2 + (A_{r_1} + 1) + k = A_{r_1}^2 + 1 + A_{r_1} + 1 + k = A_{r_1}^2 + A_{r_1} + k \tag{2.15}$$

But since A_{r_1} is the root of the equation, $A_{r_1}^2 + A_{r_1} + k = 0$. This shows that $A_{r_1} + 1$ is a root of the equation as well.

2.4.4.2 Error locator polynomials of degree 3

An error locator polynomial of degree 3 is of the form,

$$\Lambda_3 y^3 + \Lambda_2 y^2 + \Lambda_1 y + \Lambda_0 = 0. \tag{2.16}$$

The error locator polynomial of the form shown in equation 2.16 can be written as[2],

$$y^3 + (\Lambda_2/\Lambda_3)y^2 + (\Lambda_1/\Lambda_3) + \Lambda_0/\Lambda_3 = 0. \tag{2.17}$$

This can then be converted into a standard equation of the form $x^3 + x + k = 0$ by making the substitution $y = (\Lambda_2/\Lambda_3) + x((\Lambda_2/\Lambda_3)^2 + \Lambda_1/\Lambda_3)^{1/2}$.

Setting $x = w + 1/w$, we get,

$$w^3 + 1/w^3 + k = 0. \tag{2.18}$$

Substitute $w^3 = z$. This gives us, $z^2 + kz + 1 = 0$. This quadratic equation can be solved using the method presented in the previous section.

In this section, a brief overview of BCH codes and algebraic decoding of BCH codes was presented. Computationally, it is beneficial to avoid Chien search and a more efficient solution was reviewed [2]. In the next section, we will present the implementation details of the BCH decoder.

3.1 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are re-configurable semiconductor devices. Basically, FPGAs are based around a matrix of configurable logic blocks and these configurable logic blocks are then connected via programmable interconnects. The programming of these interconnects is done depending upon the logic that needs to be realized on the FPGA, hence making the FPGA re-configurable. FPGAs offer considerable speed up over software based approaches. Unlike processors, FPGAs are truly parallel in nature as each independent task/function is assigned to dedicated section of the chip which can function autonomously. A generic FPGA architecture is shown in Fig 3.1.

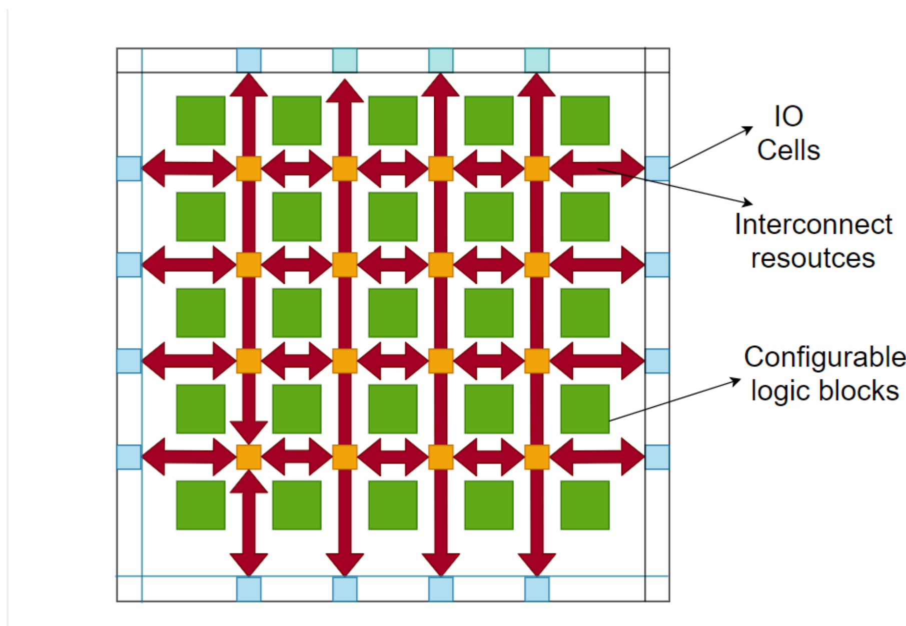

Figure 3.1: Generic FPGA architecture.

3.1.1 FPGA design flow

A typical FPGA design flow is shown in Fig 3.2. The first step in the FPGA design flow is to describe the hardware structure using a Hardware Description Language (HDL). The most commonly used HDLs are VHDL and Verilog. The hardware description for this work is written in Verilog. The next step involves simulating the design to verify that behavior described in the first step matches the expected behavior as per project requirements. This involves writing testbenches for the Design Under Test (DUT). The high level description of hardware needs to be mapped into a gate-level netlist. This translation from RTL description to gate-level netlist is done in step 3 which is Synthesis. Synthesis is done using a tool called synthesizer. In this project, Xilinx Synthesis Tool (XST) is used for converting HDL based design to gate-level netlist. After generating the gate-level netlist, the design needs to be mapped into technology specific logic units like the Configurable Logic Blocks (CLBs) on the FPGA. This is done by combining all netlists and constraints into one large netlist which is stored in a native generic database file(.ngd). The resources specified in the native generic database file are compared with available resources on the target FPGA and the large netlist present in the .ngd file is broken down into sub-blocks which can fit the FPGA logic blocks. These sub-blocks are then mapped to available FPGA logic units by the placement tool. The signals between these logic blocks on the FPGA are routed by the routing tool in such a way that the timing constraints are met. Finally, a bitstream in generated which contains the programming information for the FPGA. This includes programming the interconnect resources and Input/Output (I/O) blocks. This bitstream is then programmed on the FPGA.

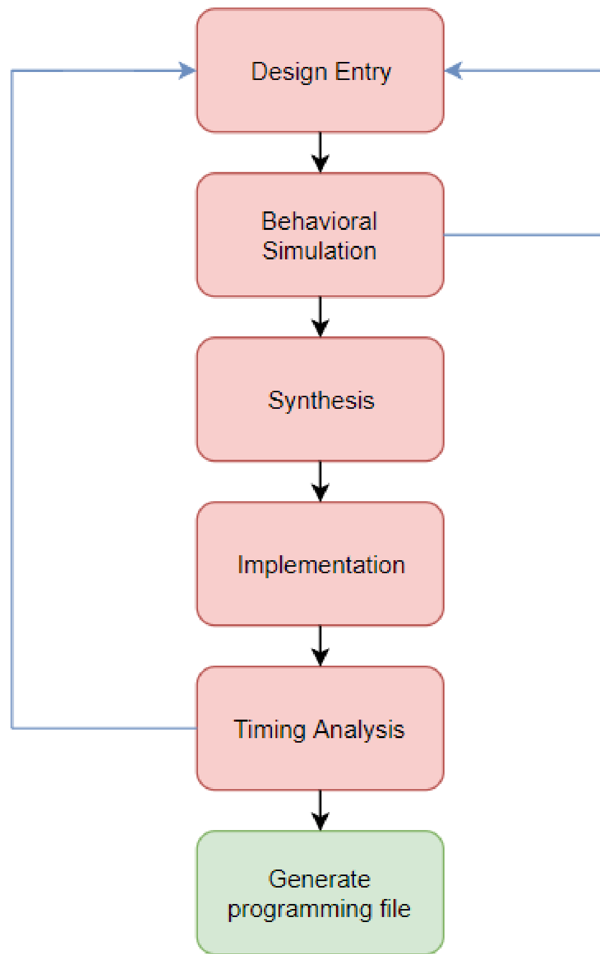

Figure 3.2: BCH decoder design flow.

3.2 BCH implementation details

The implementation of the BCH decoder is broken down into a number of custom Intellectual Property (IP) blocks as shown in fig 3.3. The IP blocks are as follows :

- Data Generator block
- Syndrome computer block
- Error locator finder block
- Error locator solver block

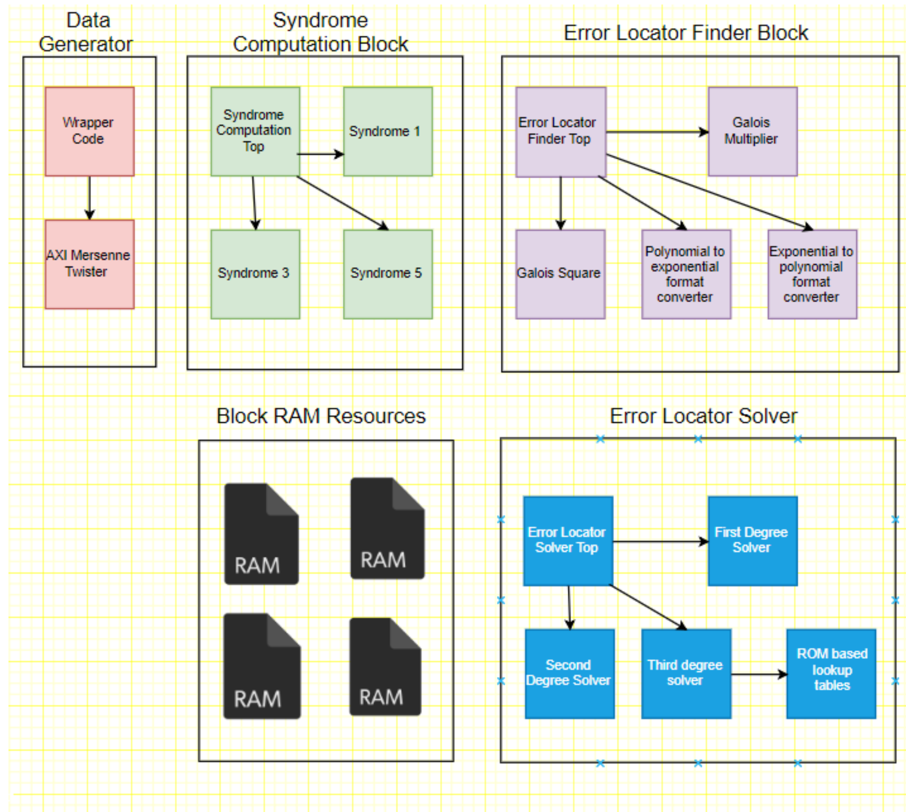

Figure 3.3: BCH decoder implemented design. The figure depicts a high level design for the BCH decoder implemented in Verilog.

3.2.1 Data generator block

This block generates pseudo random data for decoding by the BCH decoder. The pseudo random number generator used for this work is Mersenne twister (MT). Mersenne twister has a period of $2^{19937} - 1$. The MT block generates a 64 bit random number at every clock cycle. The simulated output of Mersenne Twister is shown in figure 3.4.

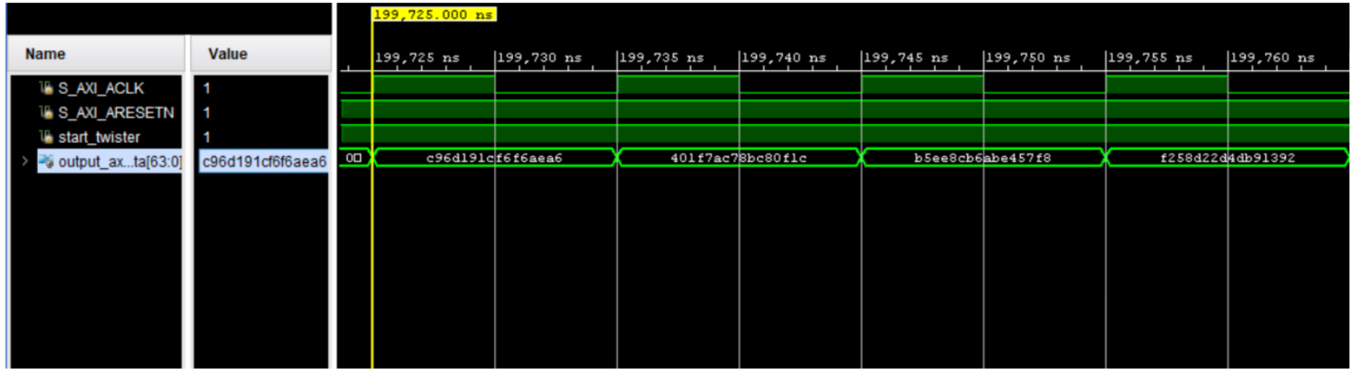

Figure 3.4: Data Generator simulated design. This figure shows the simulation output for data generator block.

We are interested in generating errors with a specific probability. To get errors in a bit stream with probability p,we follow the following steps :

- We break down the 64 bit random number generated by the Mersenne twister into four 16 bit vectors.
- Compute scaling factor, S, given by,

$$\boxed{S = \lceil p \times 65535 \rceil} \tag{3.1}$$

- If the 16 bit number is less than S, map it to a 1. Otherwise, map it to a zero.

Following the above mentioned approach, we get 4 bits every clock cycle and 1 pseudo random data sequence of 1023 bits with a specific error probability, p, is generated in 256 clock cycles.

3.2.2 Syndrome computer block

Syndrome computation in the first step in the decoding process for BCH codes. Syndromes are an indication of one or more errors in the received vector. For binary BCH codes,

$$S_{2i} = S_i^2 \tag{3.2}$$

As a consequence of equation 3.2, we only need to compute the odd numbered syndromes. The syndrome computation architecture implemented in this project is shown in fig 3.5

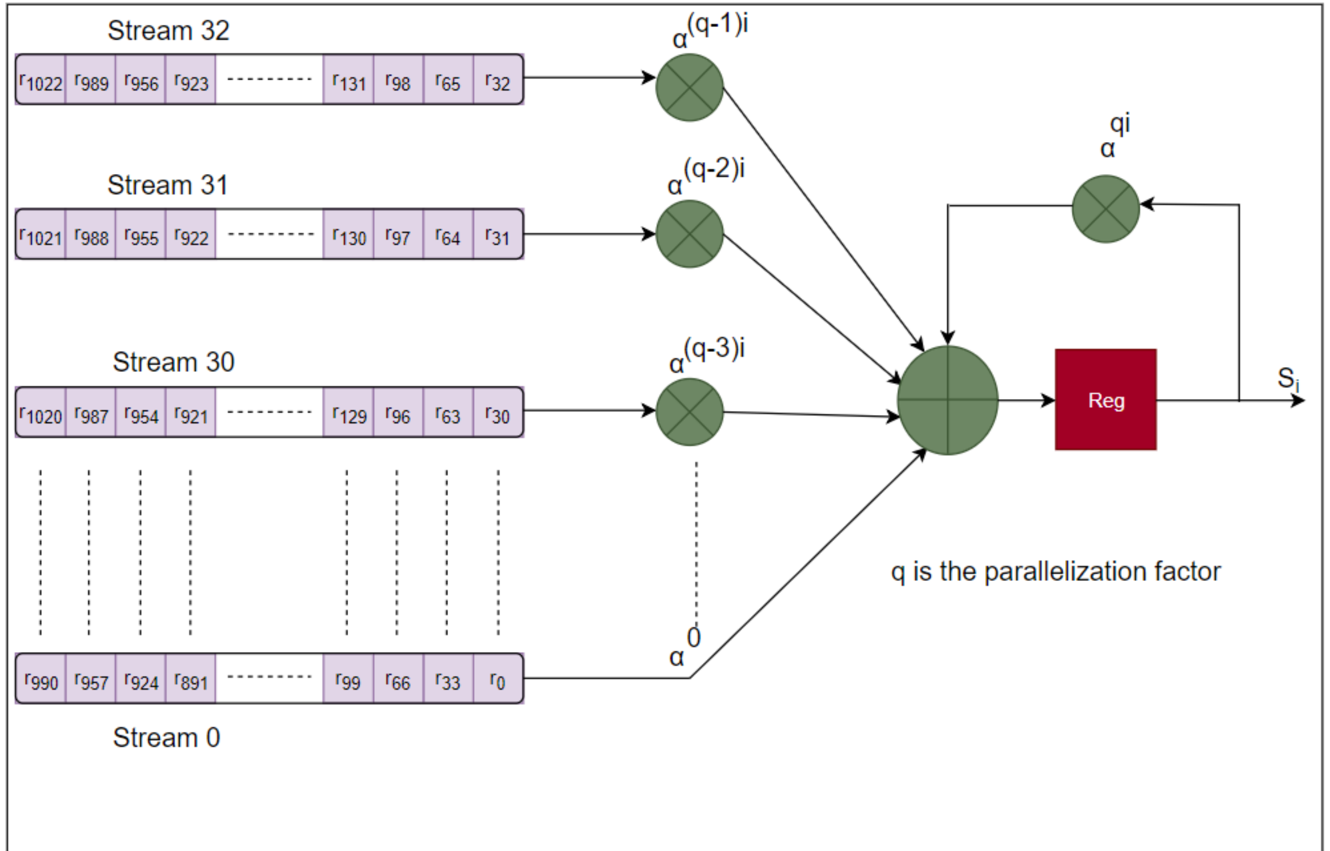

Figure 3.5: Syndrome computation architecture. The received vector is divided into 33 streams of 31 bits each.

Syndrome computation block takes a 1023 bit vector as an input. This vector is broken down into q streams. Here, q is the parallelization factor of the syndrome computation block. In contrast to the sequential circuit for computing the syndromes which needs 1023 clock cycles, this architecture needs only s clock cycles for computing the syndromes which is given by,

$$\boxed{s = 1023/q} \tag{3.3}$$

In this work, we have chosen a parallelization factor of 33 which allows us to compute the syndromes in 31 clock cycles.

3.2.2.1 Example of syndrome computation and simulation diagram

We consider a received vector of 1023 bits, r. Suppose r in polynomial format is given by,

$$r(x) = 1 + x + x^2$$

Since we are working with a 3 error correcting BCH code, we need 6 syndromes. However, since it is a binary code, we compute only the odd numbered syndromes S_1, S_3, S_5. The syndromes are given by,

$$S_1 = r(\alpha) = 1 + \alpha + \alpha^2$$
$$S_3 = r(\alpha^3) = 1 + \alpha^3 + \alpha^6$$
$$S_5 = r(\alpha^5) = 1 + \alpha^5 + \alpha^{10} = 1 + \alpha^5 + \alpha^3 + 1 = \alpha^3 + \alpha^5$$

Here, S_1, S_3, S_5 are 10 bit vectors. Converting the syndromes from polynomial form to vectors, we get,

$$S_1 = [0\,0\,0\,0\,0\,0\,0\,1\,1\,1]$$

$$S_3 = [0\,0\,0\,1\,0\,0\,1\,0\,0\,1]$$

$$S_5 = [0\,0\,0\,0\,1\,0\,1\,0\,0\,0]$$

Finally converting these 10 bit vectors to decimal, we get,

$$S_1 = 7_{10}$$

$$S_3 = 73_{10}$$

$$S_5 = 40_{10}$$

Syndromes shown in figure 3.6 match the results we computed in this section.

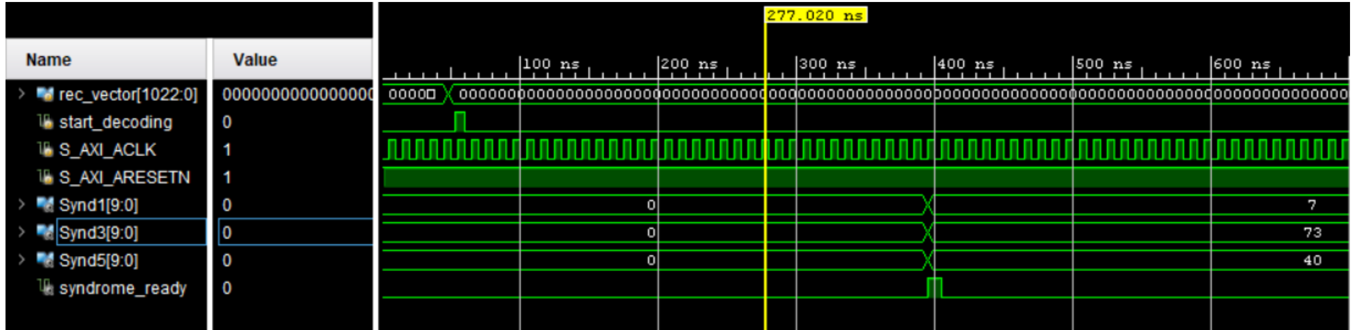

Figure 3.6: Simulation of syndrome computation block

3.2.3 Error locator finder block

The error locator finder block finds the coefficients of the error locator polynomial. There are many ways to compute the error locator polynomial. For small error correction capability, t, Peterson's algorithm provides closed form formulas for computing the coefficients of the error locator polynomial. Since we are working with 3 error correcting (1023,993) BCH code, we use the results presented in 2.11. The schematic for the error locator finder block is shown in figure 3.7. Coefficients of error locator polynomial give us some indication of number of errors that have occurred. If Λ_3 and Λ_2 are zero but Λ_1 is not zero, then a one error has occurred in the received vector. On the other hand, if Λ_3 is zero but Λ_2 and Λ_1 are non zero, then we can conclusively say that 2 errors have occurred. If all three coefficients are non-zero, then ≥ 3 errors have occurred. In general,multiplication of elements in the Galois field is simpler in exponential format (simple addition of powers) whereas addition of Galois field elements in simpler in polynomial format (simple xor). Error locator finder block uses the polynomial to exponential format and exponential to polynomial format converters to do the computations efficiently.

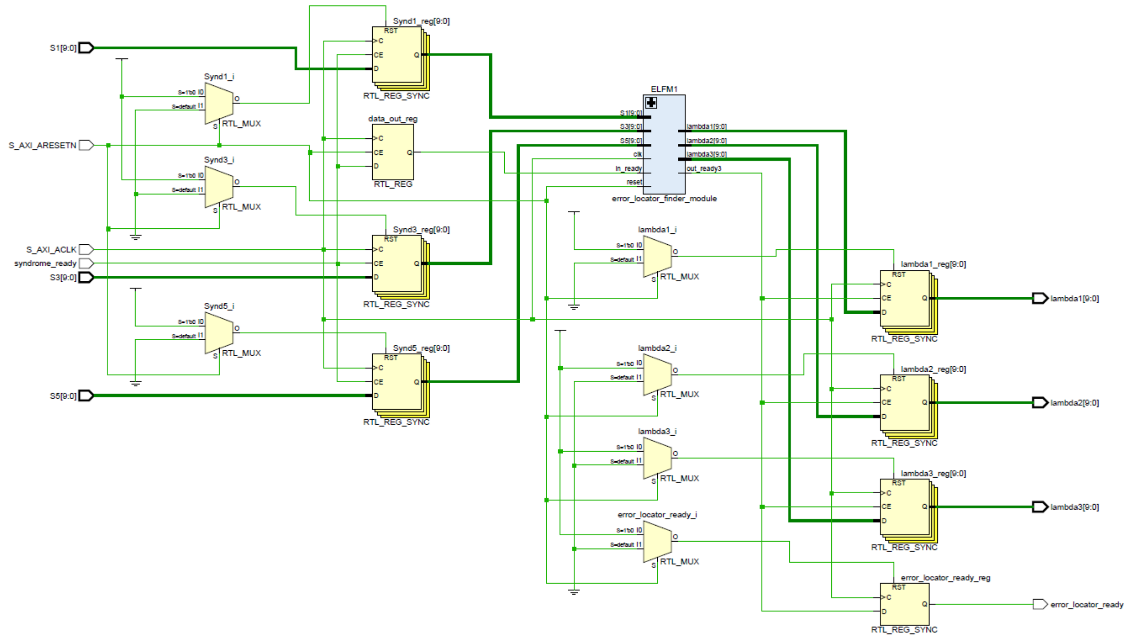

Figure 3.7: Schematic for error locator finder. This figure shows elaborated design for the error locator finder block in Vivado

3.2.3.1 Example of finding the coefficients of error locator polynomial

In this section, we will find the error locator polynomial for received vector, r, given by,

$$r(x) = 1 + x + x^2$$

In section 3.2.2.1, we computed the syndromes for this received vector. The computed syndromes were as follows :

$$S_1 = 7_{10}, S_3 = 73_{10}, S_5 = 40_{10}$$

The error locator finder block takes these syndrome values as input and gives us the coefficients of the error locator polynomial as output. The formulas for coefficients of error locator polynomial for a 3 error correcting code are given by,

$$\Lambda_1 = S_1$$

$$\Lambda_2 = (S_1^2 S_3 + S_5)/(S_1^3 + S_3) \tag{3.4}$$

$$\Lambda_3 = (S_1^3 + S_3) + S_1 \Lambda_2 \tag{3.5}$$

Using these results,

$$\Lambda_1 = S_1 = 7$$

The terms $S_1^2 S_3$ and S_1^3 in the formula for the computation of Λ_2 are easier to compute in exponential format. Converting $S_1,\ S_3$ and S_5 to exponential format, we get,

$$S_{exp_1} = 956,\ S_{exp_3} = 157,\ S_{exp_5} = 314 \tag{3.6}$$

This format conversion is shown in figure 3.8.

```
Command Window
New to MATLAB? See resources for Getting Started.
>> prim_poly = [1 0 0 1 0 0 0 0 0 0 1];
>> S1 = [1 1 1 0 0 0 0 0 0 0];
>> [S1,S1_EXP] = gftuple(S1,prim_poly);
>> S1_EXP

S1_EXP =

   956

>> S3 = [1 0 0 1 0 0 1 0 0 0] ;
>> [S3,S3_EXP] = gftuple(S3,prim_poly);
>> S3_EXP

S3_EXP =

   314

>> S5 = [0 0 0 1 0 1 0 0 0 0] ;
>> [S5,S5_EXP] = gftuple(S5,prim_poly);
>> S5_EXP

S5_EXP =

   157

fx >> |
```

Figure 3.8: Format Conversion from polynomial format to exponential format in Matlab

Substituting the exponential format of S_1, S_3 and S_5 in equation 3.4 , we get,

$$\Lambda_2 = ((\alpha^{956})^2\alpha^{314} + \alpha^{157})/((\alpha^{956})^3 + \alpha^{314}) \implies \Lambda_2 = (\alpha^{2226} + \alpha^{157})/(\alpha^{2868} + \alpha^{314}) \quad (3.7)$$

Since $\alpha^{1023} = 1$, we take the modulus of the powers by 1023. This gives us,

$$\Lambda_2 = (\alpha^{180} + \alpha^{157})/(\alpha^{822} + \alpha^{314}) \quad (3.8)$$

Matlab provides a gfadd function with the Communications toolbox that allows us to add elements in exponential format. This is shown below in figure 3.9

```
Command Window
New to MATLAB? See resources for Getting Started.
>> p = 2 ;
>> m = 10 ;
>> prim_poly = [1 0 0 1 0 0 0 0 0 0 1] ;
>> field = gftuple((-1:p^m-2)',prim_poly,p);
>> numerator_exp = gfadd(180,157,field) ;
>> denominator_exp = gfadd(822,314,field);
>> lambda2_exp = numerator_exp + 1023 - denominator_exp ;
>> lambda2_exp

lambda2_exp =

   957

>> [lambda2,lambda2_exp] = gftuple(lambda2_exp,prim_poly) ;
>> bi2de(lambda2)

ans =

    14
```

Figure 3.9: $\Lambda 2$ computation in Matlab.

As shown in figure 3.9, $\Lambda 2$ is 14.

Substituting in equation 3.5, we have,

$$\Lambda_3 = (\alpha^{822} + \alpha^{314}) + \alpha^{956}\alpha^{957} \tag{3.9}$$

Solving 3.9, we get $\Lambda_3 = 8$. Vivado simulation results are shown in figure 3.10.

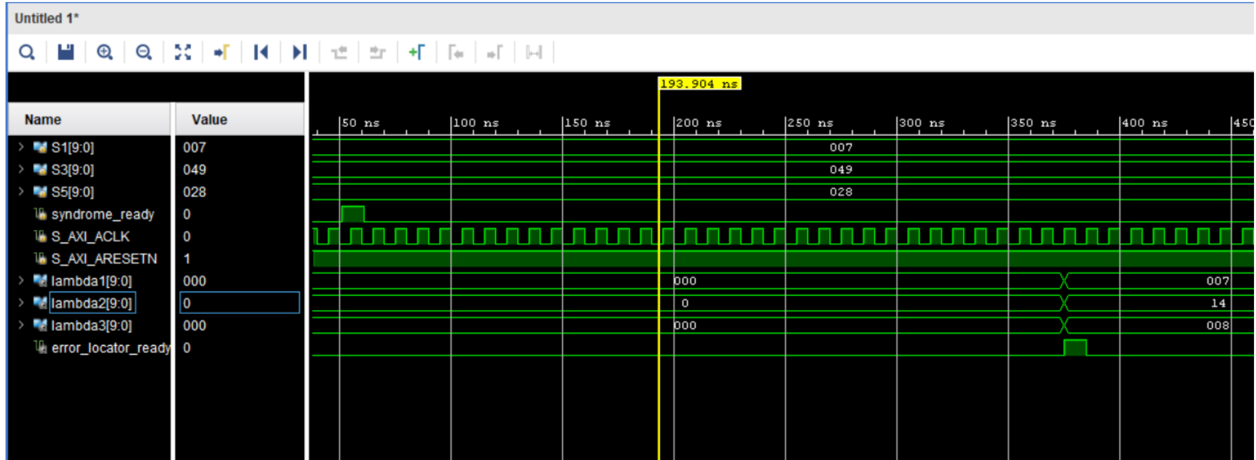

Figure 3.10: Vivado simulation for error locator finder block. The simulation shows the coefficients of the error locator polynomial.

3.2.4 Error locator solver block

3.2.4.1 Second degree solver

As we saw in section 2.4.4.1, α^7 has no roots in the field $\mathbb{GF}(1024)$. To find the roots of a standard second degree equation of the form $x^2 + x + k = 0$, we decompose the element k in terms of the standard basis. If the coefficient of α^7 after the decomposition is 1, we declare that no roots can be found in the field. Otherwise, using Table 2.1, we do a simple xor of the roots of the standard basis elements which have a coefficient 1.

3.2.4.2 Third degree solver

As we saw in section 2.4.4.2, any error locator polynomial of degree 3 can be converted into a standard equation of the form $x^3+x+k = 0$. During the design phase, we compute the roots for all

the values of k that have 3 roots in the field $\mathbb{GF}(2^m)$. This number is of the order of 128 elements for the field $\mathbb{GF}(1024)$. We create a Read Only Memory (ROM) based look-up-table to store these roots. Simulation of the error locator polynomial block for 3 errors is shown in figure 3.11. In this simulation, three errors were introduced in the received vector at locations 0,1,2. As can be seen in the figure, the error locator block correctly reports that the errors are at locations 0,1,2. The error locator block also returns the number of errors that have been corrected in the received stream. A decoding complete signal notifies that the error locator solver block has finished its computation.

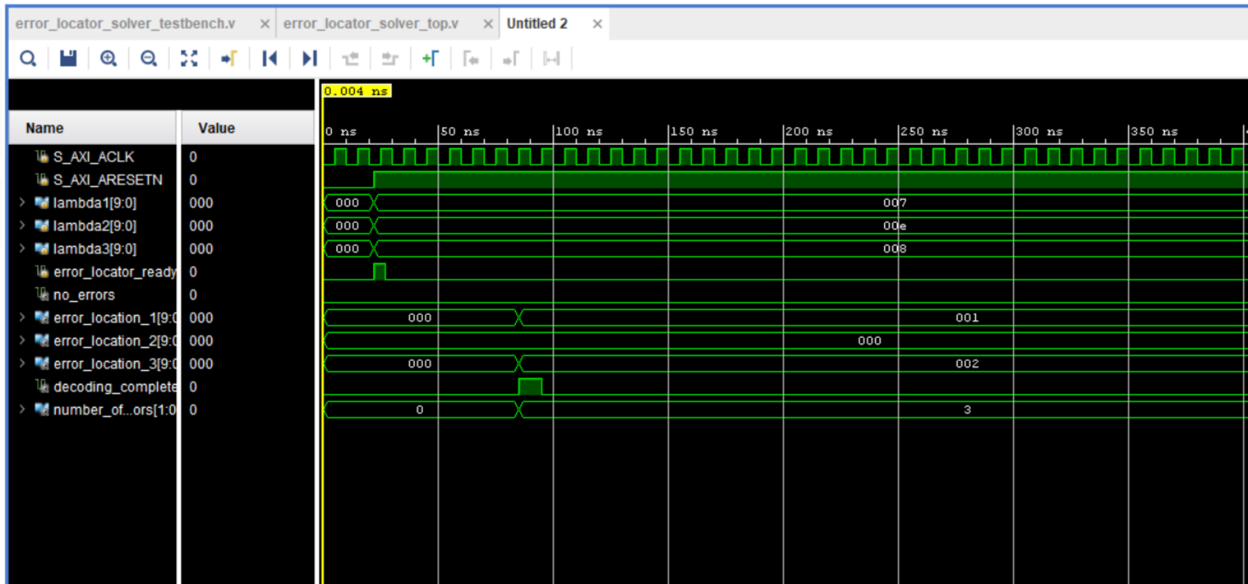

Figure 3.11: Vivado simulation for error locator solver block.

3.2.5 Complete BCH decoder implementation

The block diagram of the complete BCH decoder is shown in 3.12. Vc-709 board does not provide hard IP processors. To work around that issue, we instantiate a Microblaze soft IP processor. Microblaze functions as a AXI Master for all the other custom IP blocks which act as a slave. This means that Microblaze can initiate AXI transactions with any of these modules. This is very useful when debugging the design.

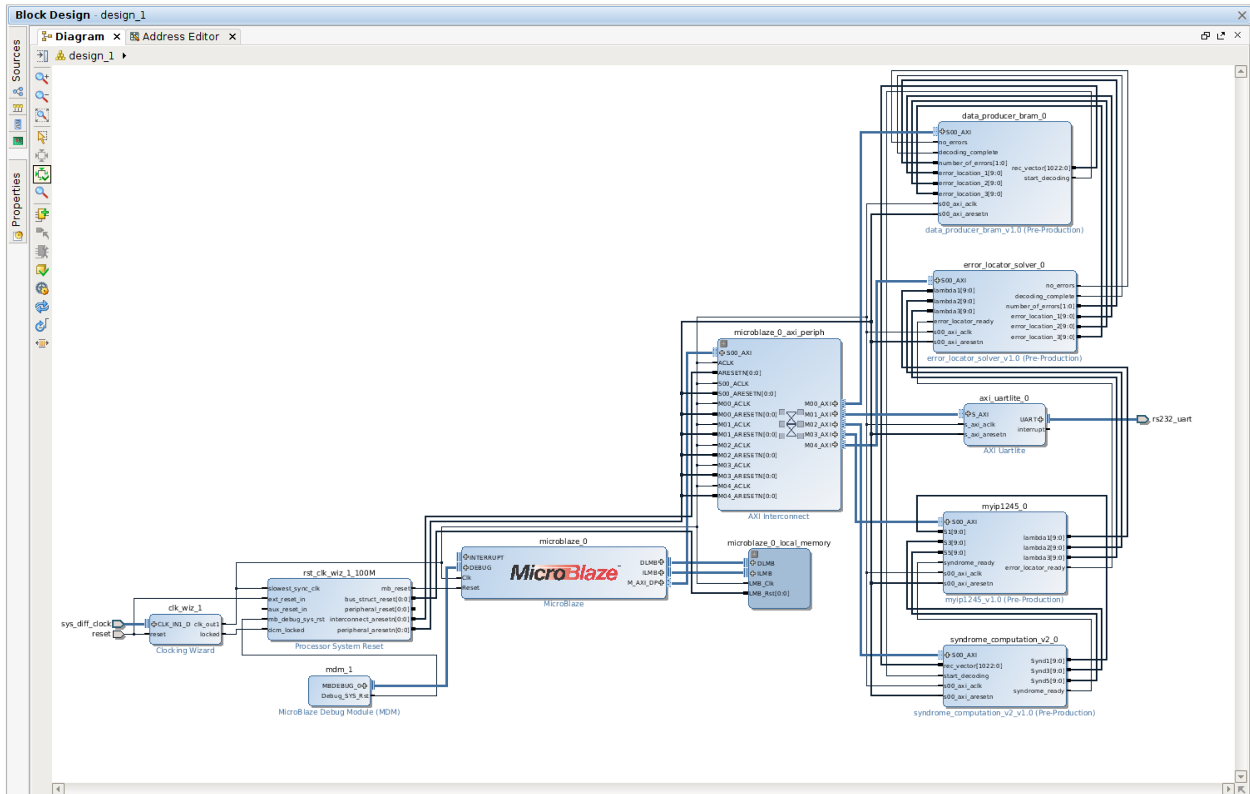

Figure 3.12: Block diagram of the complete BCH decoder.

Resource utilization and power utilization are very useful parameters for characterizing the performance of a decoder on the FPGA. Lower resource utilization enables us to instantiate multiple decoders running in parallel leading to higher throughput. Utilization report for a (1023,993) BCH decoder is shown in figure 3.13.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	12879	0	433200	2.97
LUT as Logic	12743	0	433200	2.94
LUT as Memory	136	0	174200	0.07
LUT as Distributed RAM	64	0		
LUT as Shift Register	72	0		
Slice Registers	7935	0	866400	0.91
Register as Flip Flop	7934	0	866400	0.91
Register as Latch	0	0	866400	0.00
Register as AND/OR	1	0	866400	<0.01
F7 Muxes	2986	0	216600	1.37
F8 Muxes	1188	0	108300	1.09

Figure 3.13: Utlization report of a BCH decoder.

As a general rule of thumb, 80 percent LUT utilization is considered to be an upper limit and it is generally not advisable to go beyond this threshold. Assuming a maximum LUT utilization of 80 percent, we can instantiate N decoder modules, where N is given by,

$$N = 80/2.41 \approx 33$$

.

Each decoder module gives a throughput of $\approx$ 200 Mbps. With 33 such decoder modules, a throughput of 6.6 Gbps can be achieved on an FPGA.

The total power utilization of the decoder is 0.509W. This is shown in the figure 3.14.

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.026	7	---	---
Slice Logic	0.006	26816	---	---
LUT as Logic	0.006	12743	433200	2.94
CARRY4	<0.001	411	108300	0.37
LUT as Distributed RAM	<0.001	64	174200	0.03
Register	<0.001	7935	866400	0.91
F7/F8 Muxes	<0.001	4174	433200	0.96
LUT as Shift Register	<0.001	72	174200	0.04
Others	0.000	737	---	---
Signals	0.008	13820	---	---
Block RAM	0.012	34	1470	2.31
MMCM	0.122	1	20	5.00
I/O	0.004	5	850	0.58
Static Power	0.331			
Total	0.509			

Figure 3.14: Power utilization of a BCH decoder.

The timing summary of the design is shown in the figure 3.15.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.545 ns	Worst Hold Slack (WHS):	0.064 ns	Worst Pulse Width Slack (WPWS):	1.100 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:	21180	Total Number of Endpoints:	21180	Total Number of Endpoints:	8259

All user specified timing constraints are met.

Figure 3.15: Timing summary of a BCH decoder.

3.2.6 BCH decoder performance comparisons

We implemented 3 versions of BCH decoder. The first version of the BCH decoder was implemented in Matlab and used Matlab's in-built gftuple function. This decoder was run over a set of 1000 codewords and timed using Matlab's tic-toc. The results are shown in figure 3.16.

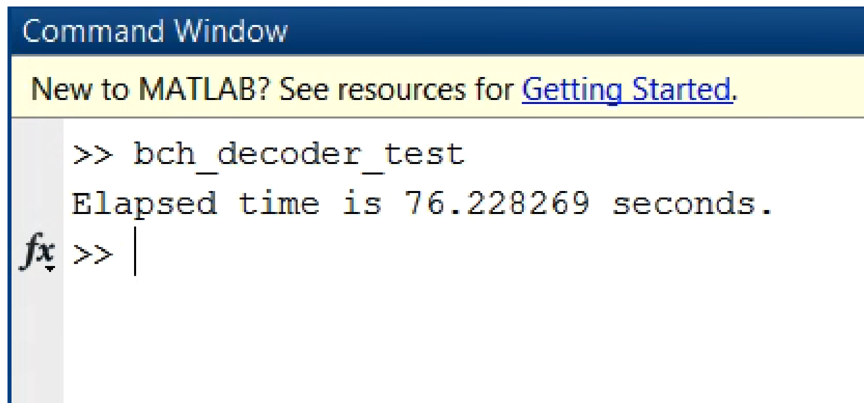

Figure 3.16: BCH decoder in Matlab. This decoder uses the in-built gftuple function.

This second flavor BCH decoder that was implemented without using Matlab's in-built gftuple function. This decoder was also run over a set of 1000 codewords. The results of Matlab's tic-toc are shown in figure 3.17.

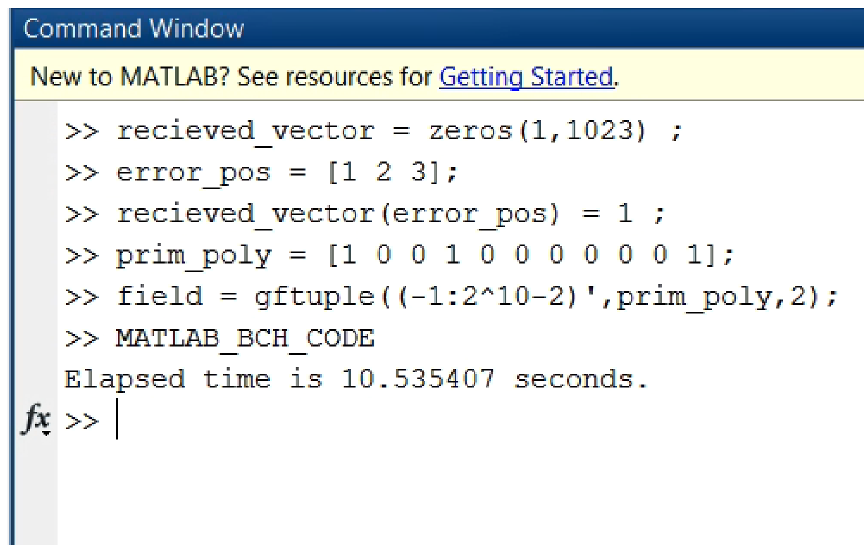

Figure 3.17: BCH decoder in Matlab. This decoder does not use the in-built gftuple function.

While the two versions presented above were software based, the third variant of the decoder was implemented in Verilog. This decoder completed decoding 1000 codewords in 75000 clocks.

We are using a 75 MHz clock. This means that a single clock period, T, is given by,

$$T = 1/(75 \times 10^6) = 13.33\,\text{ns}$$

So, the time taken for decoding 1000 codewords is $75000T = 999750\,\text{ns} = 10^{-3}\,\text{s}$. This implies that we can decode 10^7 codewords in 1s.

This hardware based BCH decoder was then run over a set of 100 million codewords for each probability p, where p ranges from 0.003 to 0.001 in steps of 0.001. The Bit Error Rate(BER) vs cross over probability curve generated is shown in figure 3.18. Generally, it is possible to mathematically compute the bit error rate for BCH code. However, computing the probability of miscorrections and decoding failure is much harder to compute analytically. Figure 3.18 presents graphs for miscorrection probability and decoding failure probability as well.

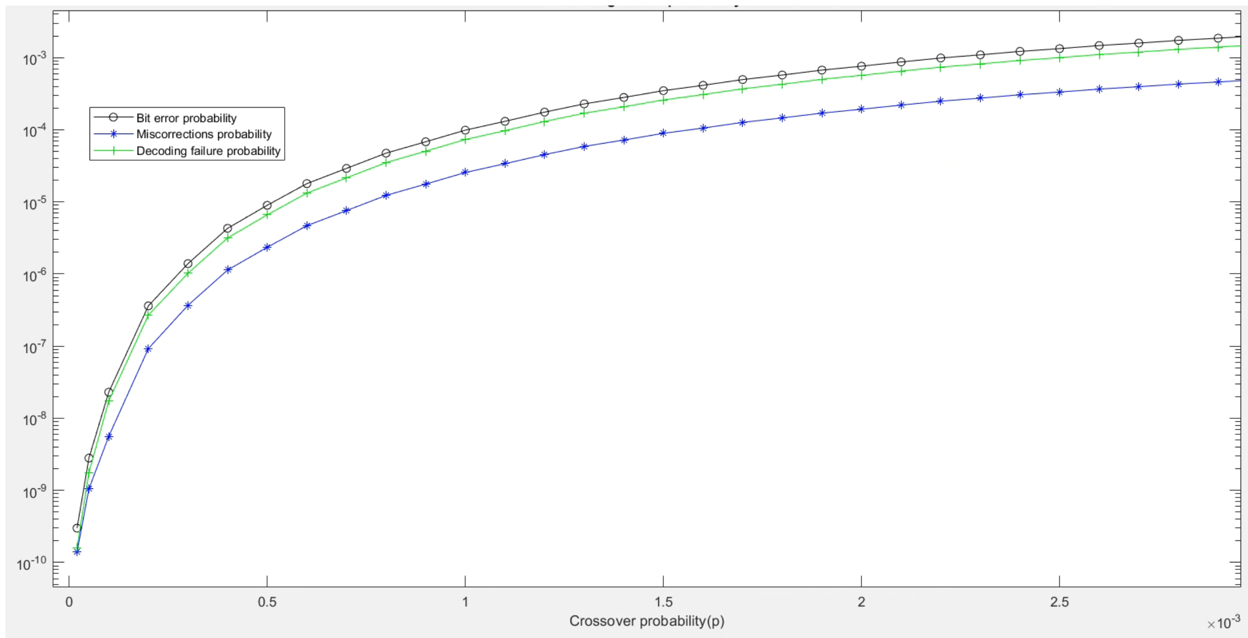

Figure 3.18: Bit Error Rate(BER) vs cross over probability(p).

3.3 Product code implementation details

3.3.1 Product code 10000 feet view

A schematic of product code decoder architecture is presented in figure 3.19.

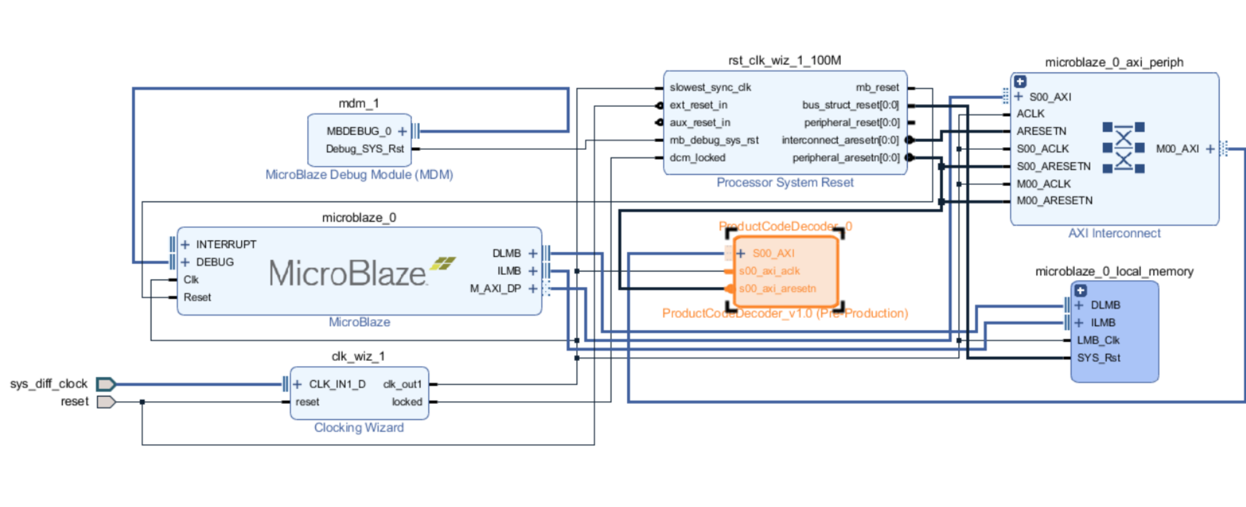

Figure 3.19: 10000 ft view of product code decoder architecture.

Microblaze is a soft IP processor and acts as an AXI master for the decoder module. Microblaze initiates the decoding by writing to the registers on decoder block. Microblaze also handles the task of gathering data from the registers of the decoder when decoding is complete and displays the results for the user. The highlighted block in the figure 3.19 represents the product code decoder. We discuss a more detailed version of this decoder in the following sections.

3.3.2 Product code detailed architecture

Detailed product code architecture is shown in figure 3.20

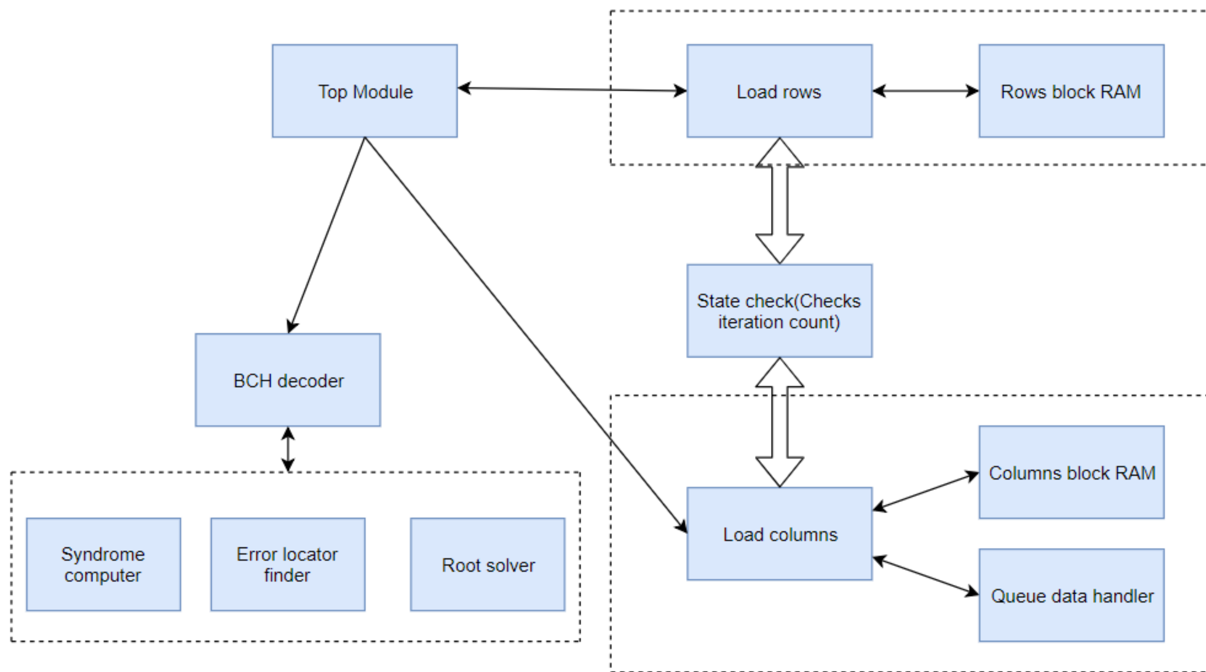

Figure 3.20: Detailed product code decoder architecture.

Initially, the decoder starts in IDLE state. Microblaze initiates the decoding operation by writing to the registers of the product code decoder. This write operation triggers the Mersenne twister random number generator and the state of the decoder changes to LOADING. In the loading stage, Twister generates 64 bits every clock cyle which get mapped to 4 bits. This way we get a complete 1023 bit recieved vector in 256 clock cycles. As soon as the first vector becomes ready for decoding, it is pushed to the BCH decoder. The BCH decoder returns the error locations and the number of errors if the decoding succeeds. Otherwise, a decoding failure is declared. Separate block RAM instances are maintained to store the corrected rows and columns after the decoding operation. The corrected vector is then written to the rows block RAM. After loading the rows, we derive column information from the corrected row and make subsequent writes to the column block RAM. A queue based architecture is used for loading the column information in block RAM. As the loading gets completed, the state of the decoder is changed to column decoding stage. In

this stage, a column is fetched from the column BRAM and forwarded to the BCH decoder for decoding. The corrected column codeword is then written to the column block RAM. The row information is derived from the corrected column and writes are performed on the row block RAM. After the column decoding stage is complete, we transition to an intermediate stage where a check is performed on the iteration count for the rows and column decoding. If the iteration count is 0 for both rows and columns, we transition back to the loading state and start the complete cycle again. Product code state diagram is shown in figure 3.21.

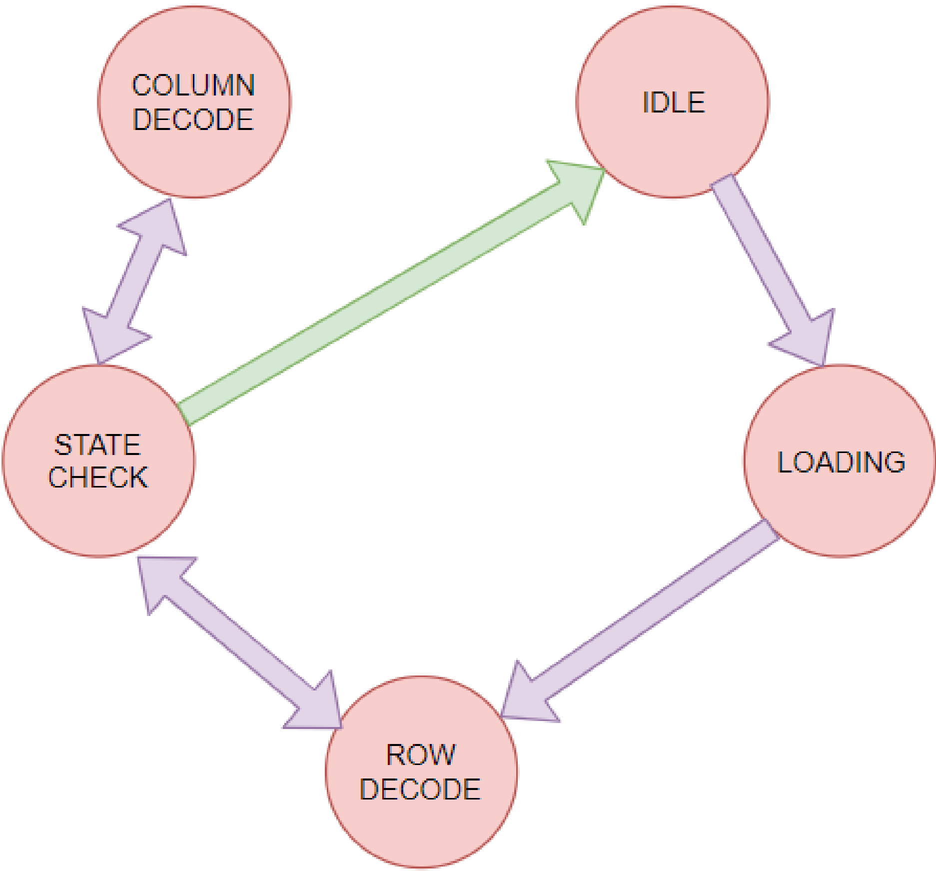

Figure 3.21: Product code decoder state diagram.

A utilization report for the product code decoder is presented in figure 3.22. The results presented here are simulation results of the Product code decoder in Xilinx Vivado.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	21829	0	134600	16.22
LUT as Logic	21829	0	134600	16.22
LUT as Memory	0	0	46200	0.00
Slice Registers	10813	0	269200	4.02
Register as Flip Flop	10813	0	269200	4.02
Register as Latch	0	0	269200	0.00
F7 Muxes	1193	0	67300	1.77
F8 Muxes	535	0	33650	1.59

Figure 3.22: Product code decoder utilization report.

4. SUMMARY AND CONCLUSIONS

The goal of the thesis was to implement a full product code decoder on FPGA. The efficiency of the full-product code decoder relies heavily on the efficiency of the component code decoder. For this work, a (1023,993) BCH code was selected as a component code. BCH decoder has a computationally expensive step for finding the roots of the error locator polynomial. In this thesis, we have implemented an efficient method to solve the error locator polynomial which is simpler to implement in hardware. A light-weight design for the BCH decoder allows us to instantiate multiple such decoder blocks without overloading the FPGA. These multiple instantiations enable us to decode multiple rows and columns in parallel. The BCH decoder presented was synthesized and run on a Virtex-7 FPGA. The results shown for the product code decoder are simulation results.

4.1 Challenges

A codeword from the product code with a BCH(1023,993) code as a component code has nearly a million bits. It is very convenient and relatively simpler to have a design where a 2 dimensional array with instant access to rows and columns is available. Here instant access implies an access which is not synchronized with the clock. However such a "custom" design for a memory construct forces the FPGA to use distributed RAMs and the logic utilization blows up. In reality, all forms of dedicated memories on FPGA allow only synchronous accesses which adds an extra layer of latency to the design. The final implementation for the product code decoder uses the dedicated Block RAM resource available on the FPGA to store the data.

Also, since each codeword comprises of a million bits, it becomes considerably difficult to analyze the data manually. Several scripts were written in Matlab to validate the design and functionality of the decoder.

4.2 Future work

The next step for this work involves implementing the complete design on an Application Specific Integrated Circuit(ASIC). ASIC fabricated using the same process can run at a much

higher frequency than FPGA and hence, yields better throughput. An ASIC design is much more power efficient than implementing the same design on FPGA. Power consumption of ASICs can be very minutely controlled and optimized.

The work presented in this thesis involves a single instance of BCH decoder which is responsible for decoding the rows and columns. The design of BCH decoder presented in this thesis is very light-weight in terms of logic utilization. Infact, it only uses 2.41 percent of the actual logic units on the FPGA. We can include several instances of the BCH decoder which can operate in a parallel fashion. Along with instantiating multiple decoders, design for each decoder module can be pipelined more to add register stages.

REFERENCES

[1] P. Elias, "Error free coding," *IRE Trans. on Information Theory*, vol. PGIT:4, pp. 29–37, 2006.

[2] C. L. Chen, "Formulas for solutions of quadratic equations over $\mathbb{GF}(2^m)$," *IEEE Trans. Information Theory*, vol. IT-28, pp. 792–794, 1982.

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

[4] J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, and J. Pelzl, "Efficient hardware implementation of finite fields with applications to cryptography," *Acta Appl. Math*, vol. 93, pp. 75–118, 2006.

[5] H. Pfister, S. Emmadi, and K. Narayanan, "Symmetric product codes," *2015 Information Theory and Applications Workshop(ITA)*, pp. 282–290, 2015.

[6] F. Chang, K. Onohara, and T. Mizuochi, "Forward error correction for 100g transport networks," *IEEE Communications Magazine*, vol. 48, pp. S48–S55, 2010.

[7] G. Tzimpragos, C. Kachris, I. B. Djordjevic, M. Cvijetic, D. Soudris, and I. Tomkos, "A survey on fec codes for 100 g and beyond optical networks," *IEEE Communications Surveys Tutorials*, vol. 18, pp. 209–221, Firstquarter 2016.

[8] J.-Y. Qin, Z.-P. Huang, J.-Y. Wei, Z. Dong, and Z. Zuo, "Efficient fec decoding algorithm for 100gbps optical transport networks in fpga," pp. 19–27, 05 2017.

[9] X. Zhang and Z. Wang, "A low-complexity three-error-correcting bch decoder for optical transport network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, pp. 663–667, Oct 2012.

[10] D. A. Morero, M. A. Castrillon, F. A. Ramos, T. A. Goette, O. E. Agazzi, and M. R. Hueda, "Non-concatenated fec codes for ultra-high speed optical transport networks," in *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, pp. 1–5, Dec 2011.

[11] D. Kim, I. Yoo, and I. Park, "Fast low-complexity triple-error-correcting bch decoding architecture," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, pp. 764–768, June 2018.

[12] Y. Krainyk, V. Perov, M. Musiyenko, and Y. Davydenko, "Hardware-oriented turbo-product codes decoder architecture," in *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, pp. 151–154, Sep. 2017.

[13] R. Le Bidan, C. Leroux, C. Jego, P. Adde, and R. Pyndiah, "Reed-solomon turbo product codes for optical communications: From code optimization to decoder design," *EURASIP Journal on Wireless Communications and Networking*, vol. 2008, p. 658042, May 2008.

[14] N. Abramson, "Cascade decoding of cyclic product codes," *IEEE Transactions on Communication Technology*, vol. 16, pp. 398–402, June 1968.

[15] C. Yang, Y. Emre, and C. Chakrabarti, "Product code schemes for error correction in mlc nand flash memories," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, pp. 2302–2314, Dec 2012.