

# Visvesvaraya Technological University, Belagavi.



PROJECT REPORT

on

## **“NAND FLASH BASED IN-FLIGHT ACQUISITION AND RECORDING UNIT FOR ACCELERATION SENSOR ASSEMBLY OF FLIGHT CONTROL SYSTEM”**

Project Report submitted in partial fulfillment of the requirement for the award of  
the degree of

**Bachelor of Engineering**

in

**Electronics and Communication Engineering**

For the academic year 2019-20

Submitted by

USN

Name

1CR16EC027

Bhavana R Reddy

1CR16EC052

Hrusna Chakri Shadakshri.V

Under the guidance of

Internal Guide

Dr. Sharmila K. P

Professor

Department of ECE

CMRIT,

Bengaluru-560 037.

External Guide

Mr.E. Krishna Kishore,

Mr.Rajesh Kumar Garg

Scientist 'D'

ADA

Bengaluru -560 017.



Department of Electronics and Communication Engineering  
**CMR Institute of Technology, Bengaluru – 560 037**

Fax: +91-80-25238493

Tel. : +91-80-25087002 / 25087566  
25087777 / 25087555

**ADA**

**AERONAUTICAL DEVELOPMENT AGENCY**

(Ministry of Defence, Govt. of India)

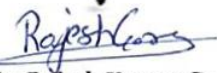
P B No. 1718, Vimanapura Post, Bangalore - 560 017, India

30<sup>th</sup> June 2020

**CERTIFICATE**

This is to Certify that **Ms. Bhavana R Reddy** (ICR16EC027) and **Ms. Hrusna Chakri Shadakshri V** (ICR16EC052) pursuing **B.E. in Electronics and Communication Engineering** has completed the project "**NAND Flash Based In-Flight Acquisition And Recording Unit For Acceleration Sensor Assembly of Flight Control System**" successfully at Aeronautical Development Agency (ADA), Autonomous Body —Ministry of Defence, Govt. of India P.B. No. 1718, Vimanapura Post, Bengaluru - 560 017 from 16<sup>th</sup> Sept. 2019 to 5<sup>th</sup> June 2020.

External Guide



**Mr. Rajesh Kumar Garg,**  
Scientist/Engineer 'D',  
Aeronautical Development Agency (ADA)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING



**CERTIFICATE**

This is to Certify that the dissertation work “**NAND Flash Based In-Flight Acquisition And Recording Unit For Acceleration Sensor Assembly Of Flight Control System**” carried out by Bhavana R Reddy, 1CR16EC027, Hrusna Chakri Shadakshri V, 1CR16EC052, bonafide students of **CMRIT** in partial fulfillment for the award of **Bachelor of Engineering in Electronics and Communication Engineering** of the **Visvesvaraya Technological University, Belagavi**, during the academic year **2019-20**. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said degree.

Signature of Guide

Signature of HOD

Signature of Principal

\_\_\_\_\_  
Dr. Sharmila K. P,  
Professor,  
Dept. of ECE.,  
CMRIT, Bengaluru.

\_\_\_\_\_  
Dr. R. Elumalai  
Head of the Department,  
Dept. of ECE.,  
CMRIT, Bengaluru.

\_\_\_\_\_  
Dr. Sanjay Jain  
Principal,  
CMRIT,  
Bengaluru.

**External Viva**

Name of Examiners

- 1.
- 2

Signature & date

## ACKNOWLEDGEMENT

The satisfaction and euphoria that accompanies the successful completion of any task would be incomplete without mentioning the people, whose consistent guidance and encouragement has served as a beacon and crowned my efforts with success.

We take an opportunity to thank all the distinguished personalities for their enormous and precious support and encouragement throughout the duration of this seminar.

We take this opportunity to express our sincere gratitude and respect to CMR Institute of Technology, Bengaluru for providing us an opportunity to carry out our project work.

We express our gratitude to Principal **Dr. Sanjay Jain**, Principal, CMRIT, Bengaluru, for having provided me the golden opportunity to undertake this project work in their esteemed organization.

We sincerely thank **Dr. R. Elumalai**, HOD, Department of Electronics and Communication Engineering, CMR Institute of Technology for the immense support given to me. We express our warm thanks to our guide **Mr. E. Krishna Kishore & Mr. Rajesh Kumar Garg**, Scientist 'D', Aeronautical Development Agency, Bengaluru-17, for their skillful guidance, constant supervision, timely suggestions and constructive criticism in the successful completion of project work in time.

We express our gratitude to our project guide **Dr. Sharmila K. P**, Professor, Department of Electronics and Communication Engineering, CMRIT, Bengaluru. for their support, guidance, and suggestions throughout the project work. Their guidance gave us the environment to enhance our knowledge, skills, and to reach the pinnacle with sheer determination, dedication, and hard work.

We also extend our thanks to the faculties of Electronics and Communication Engineering Department who directly or indirectly encouraged us throughout the course of project work.

Last but not the least, heartfelt thanks to our parents and friends for all their moral support they have given us during the completion of this work.

# TABLE OF CONTENTS

CHAPTER 1	1
INTRODUCTION	1
1.1 Requirement of ECC in NAND Flash	2
1.2 Overview of BCH	3
1.3 Motivation	3
1.4 Objectives	4
CHAPTER 2	5
LITERATURE SURVEY	5
2.1 LCA-FCS, DFCC	5
2.2 Accelerometer Sensor Assembly (ASA)	6
2.3 Data Integrity:	7
2.3.1 Data integrity vs. Data security:	7
2.4 Error Control Coding	8
2.4.1 Linear Block Codes	9
2.5 Galois Field (GF):	10
2.5.1 Properties of Galois Field	11
2.5.2 Galois field GF(2) "Binary Field"	11
2.5.3 Extension Fields	12
2.6 BCH Codes	13
2.6.1 BCH Merits:	13
2.6.2 BCH Demerits:	14
2.6.3 Some examples of BCH Applications:	15
2.7 BCH Encoder	15
2.7.1 Code generation:	15
2.7.2 Primitive Polynomials:	16
2.7.3 Minimal Polynomials:	17

2.8 BCH Decoder	17
2.8.1 Algebraic Decoding	17
A. Peterson-Gorenstein-Zierler Decoding	17
B. The Berlekamp-Massey Decoding Algorithm	19
C. Sugiyama's Euclidean Decoding Algorithm	21
2.8.2 Chien Search Algorithms	23
CHAPTER 3	28
HARDWARE	28
3.1 Acquisition and Recording Unit (ARU)	28
3.2 System Initialization	29
3.3 NAND Flash Controller	30
3.1.1 Data integrity for ASA-ARU:	32
3.2 FPGA-ARTIX 7:	32
3.3 NAND Flash Memory	35
3.3.1 Types of NAND:	36
CHAPTER 4	38
SOFTWARE	38
4.1 ECC for NAND Flash	38
4.1.1 Data Recording – NAND Flash	39
4.2 BCH Codes in ASA-ARU Application	40
4.3 BCH Encoder	41
4.3.1 Design and Implementation of Systolic- Array type Binary BCH Encoder	42
A. Generator Polynomial	42
B. Construction of BCH (8191,8139,4) Encoder	43
4.4 BCH Decoder	45
4.4.1 Design and Implementation Binary BCH Decoder	45
A. Galois Field roots generation	45
B. Syndrome calculation	45

C. Coefficients of error locator polynomial	46
D. Roots of $\Lambda(x)$	47
CHAPTER 5	49
RESULTS	49
5.1 Simulation results of BCH(8191,8139,4) Encoder	49
5.2 Performance Comparison of Conventional and Parallel BCH(63,39,4) Encoder	49
5.3 Simulation Results of BCH Decoder	51
CHAPTER 6	53
APPLICATIONS AND ADVANTAGES	53
6.1 Applications of BCH codes	53
6.1.1 Digital Communications and Storage	53
6.1.2 BCH Codes as Industry Standards	53
6.1.3 BCH Code in image encryption	54
6.1.4 Error-free Communication in NB-IoT	55
6.2 Advantages	56
CHAPTER 7	57
CONCLUSIONS AND SCOPE FOR FUTURE WORK	57
REFERENCES	58
APPENDIX A	59
BCH Encoder: Module definition	59
BCH Encoder: Test Bench	63
APPENDIX B	68
BCH Decoder: Module Definition	68
BCH Decoder: Test Bench	87
APPENDIX C	94
Verification using built-in matlab function:	94

## LIST OF FIGURES

<b>Figure 1.</b> ASA interface with DFCC	1
<b>Figure 2.1:</b> Aircraft Flight Control System	5
<b>Figure 2.2:</b> Cantilever Capacitor Output	7
<b>Figure 2.3:</b> Classification of ECC	8
<b>Figure 2.4</b> Systematic form of codeword of a linear block code	10
<b>Figure 2.5:</b> The Peterson-Gorenstein-Zierler Algorithm Flowchart	19
<b>Figure 2.6:</b> BMA Algorithm with flowchart	20
<b>Figure 2.7(a) :</b> Conventional Chien search circuit	23
<b>Figure 2.7(b) :</b> p-parallel Chien search architecture: direct unfolded version	24
<b>Figure 2.7(c) :</b> p-parallel Chien search architecture: equivalent architecture with shorter critical path	24
<b>Figure 2.8</b> Basic components in Chien search architecture(a) MPCNj(b) BTj (c) GBT	25
<b>Figure 2.9</b> MPCN-based parallel-p Chien search architecture.	26
<b>Figure 2.10</b> Parallel-p joint syndrome calculator and Chien search with MPCN based architecture.	27
<b>Figure 3.1</b> ASA-ARU system interface with ASA-LRU	28
<b>Figure 3.2</b> ASA-ARU system-Main Control Board	29
<b>Figure 3.3</b> Controller modules of FPGA	30
<b>Figure 3.4</b> NAND Flash Controller module	30
<b>Figure 4.1A:</b> NAND FLASH Array Organisation	38
<b>Figure 4.1B:</b> Main area and its division	39
<b>Figure 4.2</b> Hardware Systolic Array Type BCH Encoder	41
<b>Figure 4.3</b> Block Flow Diagram of BCH Encoder-Decoder	42
<b>Figure 4.5</b> Conventional Chien search	47
<b>Figure 4.6</b> Flow diagram for proposed BCH Decoder	48
<b>Figure 5.1</b> Simulated waveform for BCH(8191,8139,4) Encoder (message = 2.9230e+47 (in decimal))	49
<b>Figure 5.2</b> Serial BCH(63,39,4) Encoder	50
<b>Figure 5.3</b> Parallel BCH(63,39,4) Encoder	50
<b>Figure 5.4A :</b> Syndrome when no errors in rx	51



<b>Figure 5.4B</b> : Syndrome when rx has errors	51
<b>Figure 5.5A</b> : Coefficients when no errors in rx	51
<b>Figure 5.5B</b> : Coefficients when rx has errors	52
<b>Figure 5.6A</b> : Roots when no errors in rx	52
<b>Figure 5.6B</b> : Roots when rx has errors	52
<b>Figure 6.1</b> (a) Original Lena image;(b) Image by AES;(c) Image by AES-C	54
<b>Figure 6.2</b> NB-IoT architecture with BCH arrangement	55

## LIST OF TABLES

<b>Table 2.1</b> : (a)Modulo-2 addition(XOR) ;(b) Modulo-2 Multiplication (AND)	11
<b>Table 2.2</b> : (a) Addition for $GF(4) = \{0,1,2,3\}$ ;(b)Multiplication for $GF(4) = \{0,1,2,3\}$	12
<b>Table 2.3</b> :(a)Addition for $GF(4)=\{0,1,a,b\}$ ;(b)Multiplication for $GF(4)=\{0,1,a,b\}$	12
<b>Table 2.4</b> : BCH codes parameters of lengths less than $2^{10}-1$	14
<b>Table 3.1</b> Operating Environment of Hardware Description Language	33
<b>Table 3.2</b> I/O Pin/Device/Package Combinations for Artix-7 FPGAs	34
<b>Table 3.3</b> Characteristic Comparison of NAND and NOR	36
<b>Table 3.4</b> Parameteric Comparison of NAND and NOR	37
<b>Table 4.1</b> Recommended BCH Code	39
<b>Table 4.2</b> Root Table for $GF(2^{13})$	45
<b>Table 5.1</b> : Performance comparison of parallel and serial BCH encoder observed in simulation	50

## Chapter 1

### INTRODUCTION

There is need to develop an on-board acquisition unit for interfacing with ASA and assess the in-flight performance of the unit. The in-flight acquisition and recording unit provide excitation voltages to four channels of the ASA unit. ASA operated in two modes namely bit mode and normal mode. During bit mode, ARU provides bit excitation signals to four channels of ASA and need to acquire bit outputs from the ASA unit. In the normal mode of operation, all four channels accelerometer sensor outputs are required to acquire continuously and record them on board. The On-board recorded data need to be downloaded through a high-speed serial interface after the flight on ground for post flight performance analysis of the ASA unit. The unit should provide isolated power to all four channels of ASA. ASA-ARU needs to be compact, lightweight, rugged, low power and airworthy unit to use along with indigenously developed ASA. The ASA-ARU needs to provide a similar interface of DFCC to ASA. The existing interface of ASA with DFCC is given Fig. 1.

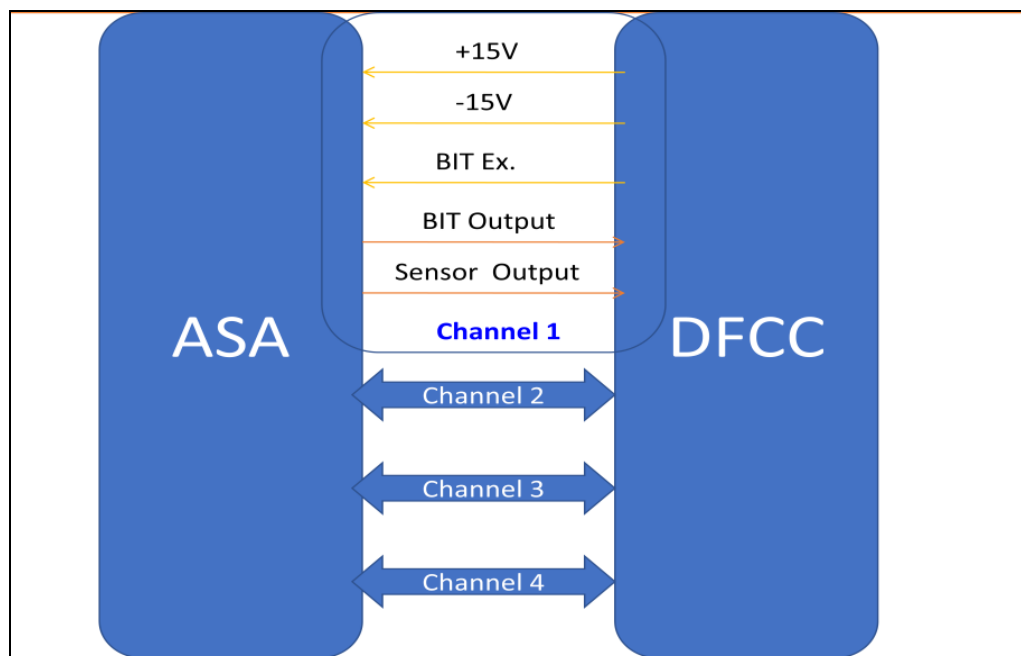


Figure 1. ASA interface with DFCC

The ASA-ARU will have FPGA based main control module for BIT excitation generation, acquisition of BIT output, normal mode sensor output and recording to on-board NAND flash memory. The Ethernet interface is used for downloading of in-flight recorded data.

The RS 232 interface is planned to monitor system health on ground. There will be separate power supply module for the generation of four independent voltages to four channels of ASA.

### 1.1 Requirement of ECC in NAND Flash

Practically, there exists no channel that is noise free and even a single bit error might lead to a major setback for a safety critical system of an aircraft, like flight control systems of a fighter aircraft. Hence, a need for encoding the data along with error correcting and controlling mechanisms for error-free retrieval at the receiver end arises. The interest for a completely dependable computerized framework has been quickened by the accessibility and fast advancement of VLSI innovation, rapid information systems, and capacity of advanced data.[1] Error control coding schemes are linear codes, categorised into Convolution codes and Block codes whose examples are Reed-Solomon, BCH (Bose-Chaudhuri-Hocquenghem) codes, Golay, Cyclic codes, Repetition codes, Polynomial codes, Hamming codes and non-linear codes. The most straightforward block code being Hamming codes, is just appropriate for basic error control circuit while BCH, the generalization of Hamming codes, forms a wide range of effective arbitrary error correcting cyclic codes that is capable of rectifying multiple errors [2].

The ECC mechanism is implemented in two opposing functions. The first is the encoding operation and second is the decoding operation where the former adds spare bits and the latter removes these added bits iteratively [3].

The widely used NAND Flash memory systems are vulnerable to multiple types of errors, such as, retention errors due to charge leakage, physical errors due to coupling noises, errors generated due to shift of threshold voltages as the memory density increases and many more. Thus, the data reliability is of utmost importance for any communication and storage systems in terms of high operational speed and other aspects. Therefore, to improve data reliability we use ECC. With the occurrence of random errors, the first preference would be BCH codes as these are adaptable to wide ranges of code length and possess a versatile error correcting capability over Reed Solomon as the latter is mostly suitable for handling burst errors [4].

## 1.2 Overview of BCH

A conventional BCH design includes an elementary shift register, the LFSR (linear feedback shift register), which in correspondence with explicit XOR, computes single message bit per cycle. Considering the need of circuits with high operational speed, the BCH serial encoder is replaced with the parallel BCH encoders which process p-bit data at an instance. Matrix multiplication, CRT based encoding, unfolding method are some of the parallel processing methods used [5]. In this paper, we employ a BCH encoder with tree-type systolic array architecture. This architecture does without modifying the generator polynomial and extra hardware requirement unlike other three methods mentioned above.

Galois Field (GF) is named after Evariste Galois. The existence of a finite count of elements characterises the GF. Data in vector form in a GF allows mathematical operations to scramble data easily and effectively. Some of the significant properties of GF are:

- All elements of GF are defined on addition and multiplication and the resultant must also be an element of GF.
- Addition (a) and subtraction (b) are inversely related (i.e.  $a+b=0$ ) and similarly, multiplication(c) and division(d) are inverse to one another (i.e.  $c*d=1$ ) [6].

A BCH decoding system is designed for correction of errors in the codeword that might have occurred in the intermediate channel. In the proposed BCH decoder, there are four sub modules, the  $GF(2^{13})$  root table generation, syndrome calculation, computation of coefficients of error locator polynomial (PGZ (Peterson-Gorenstein-Zierler) Algorithm) [7] and determining roots of the error locator polynomial(Chien search) [8].

## 1.3 Motivation

The aerospace systems demand very high level of reliability and safety. Stringent development process is followed for development for aerospace systems to meets these requirements. The design and development exposure in aerospace domain will help in developing systems for all other domains. This has motivated to take up project related fighter aircraft application. In this aerospace domain, the need for compact, rugged, high speed data acquisition and on-board storage is becoming crucial for flight test applications. This in-turn has motivated to take up a Project on **“NAND Flash Based In-Flight Acquisition and Recording Unit for Accelerometer Sensor Assembly of Flight Control System”**

## 1.4 Objectives

The main objectives of this Project are given bellow:

1. General understanding of LCA-FCS, DFCC, ASA and its sensors.
2. Understanding of ASA in-flight test requirements and its interface.
3. Study of ARU design and development process.
4. Study of NAND Flash interface with FPGA.
5. Implementation of ECC module using VHDL.
6. Functional simulation
7. Verification using in-built MATLAB functions

## Chapter 2

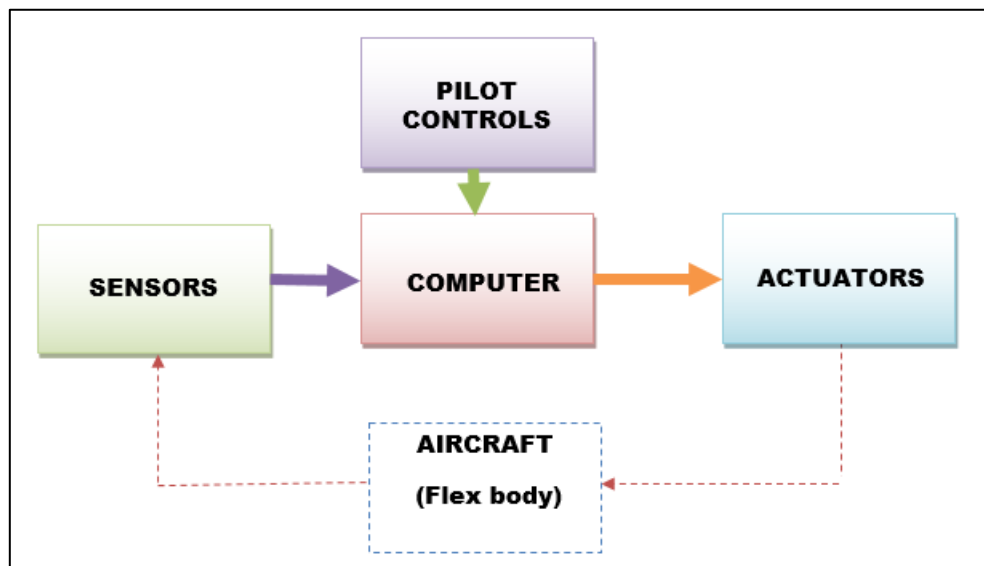
# LITERATURE SURVEY

Extensive literature survey helps in understanding the work already carried in the field of investigation and also provides the technology trend in the domain to help to home on to the challenging problem to take up and further investigate as a part of the project.

As a part of literature survey, multiple papers were examined and out of these, few important papers which are in the area of interest were investigated in detail. The outcome of the literature survey is given below:

### 2.1 LCA-FCS, DFCC

Flight control system of an aircraft consists of Flight control computer, Sensors, Cockpit sensors and actuators as shown in figure 2.1.



**Figure. 2.1:** Aircraft Flight Control System

LCA flight Control System employees Digital Fly-by-wire flight control system. The heart of the fly-by-wire flight control system is Digital Flight Control Computer (DFCC).

DFCC interfaces with Accelerometer Sensor Assembly, Rate Sensor assembly, Cockpit controls like Pilot Stick, Rudder pedals etc., and Direct Drive Valve (DDV) based Actuators. It contains Flight program with Control Laws.

Fly-by-wire control system, developed in the early 1970's, is purely electrically signaled control system, which use computer to process flight control input by pilot/autopilot and

send corresponding electrical signal to flight control surface actuators. This replaces mechanical linkage, that is, pilot inputs do not directly move to control surface.

Features:

- Provides safety and reliability
- Reduces pilot work load
- Higher Fuel efficiency
- Overall cost reduction

## 2.2 Accelerometer Sensor Assembly (ASA)

An accelerometer is an electromechanical device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic which is caused by moving or vibrating the accelerometer.

FCS of TEJAS uses body acceleration for stabilization and command augmentation. The body acceleration is obtained by axes accelerometer sensor assembly.

### **BAE's ASA VS Indigenous ASA:**

- A single LRU, Line Replacement Unit consist of two accelerometer sensors, that is,
  - o Lateral axis
  - o Normal axis
- These sensors are imported from M/s BAE systems, USA.
- Since few components of BAE –ASA are obsolete (production from vendors has stopped), ASA is out of production as the alternatives are expensive.
- Thereby, ADA initiated indigenous development of MEMS based ASA.

**MEMS (Micro-electromechanical system)** devices that have characteristics of very small size ranges from few micrometres to millimetres combine both mechanical and electrical components fabricated using IC batch processing technologies.

MEMS-ASA:

- MEMS-ASA consists of three Axes:
  - o Longitudinal axis
  - o Lateral axis
  - o Normal axis
- And it is a quadruplex redundant channel. Therefore 12 sensor output data is available.

- Types:
  - o Cantilever capacitor output
  - o Proof mass pendulum

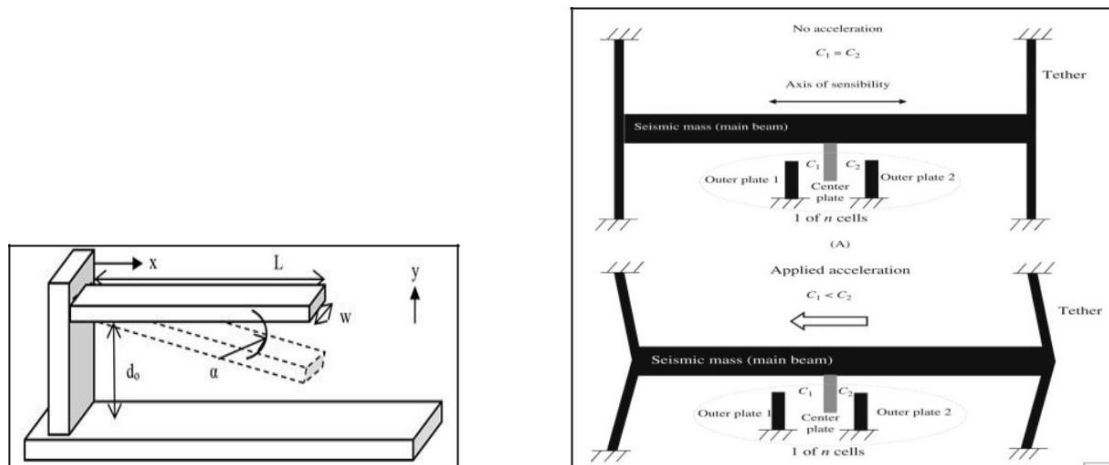


Figure.2.2: Cantilever Capacitor Output

### 2.3 Data Integrity:

- Maintaining the data consistent throughout its lifecycle is a matter of protecting it so that it's reliable. Uncorrupted data is considered to be whole and then stay unchanged.
- Data integrity refers to the fact that data must be reliable and accurate over its entire lifecycle. Data integrity(uncorrupted) and data security(protection) go hand in hand.
- Data is expected to be attributable, legible, contemporaneous, original and accurate (ALCOA principle).

#### 2.3.1 Data integrity vs. Data security:

Data security refers to the protection of data against unauthorized access or corruption and is necessary to ensure data integrity. Data integrity[9] is a desired result of data security. Data security, in other words, is one of several measures which can be employed to maintain data integrity. Whether it's a case of malicious intent or accidental compromise, data security plays an important role in maintaining data integrity.

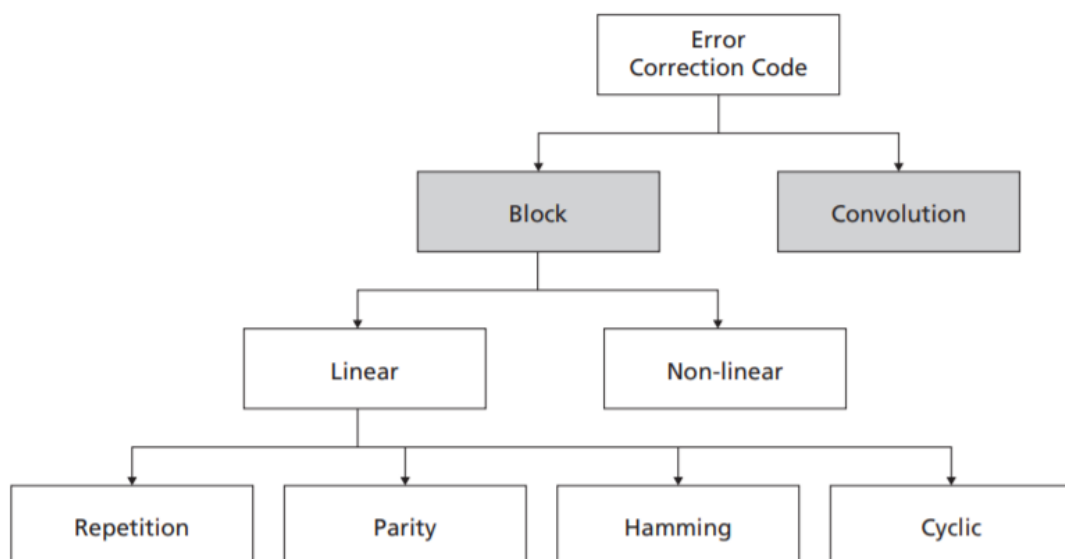


## 2.4 Error Control Coding

In recent years there has been an increasing demand for digital transmission and storage systems. This demand has been accelerated by the rapid development and availability of VLSI technology and digital processing. It is frequently the case that a digital system must be fully reliable, as a single error may shutdown the whole system, or cause unacceptable corruption of data, e.g. in a bank account. In situations such as this error control must be employed so that an error may be detected and afterwards corrected. The simplest way of detecting a single error is a parity checksum [10], which can be implemented using only exclusive-or gates. But in some applications this method is insufficient and a more sophisticated error control strategy must be implemented.

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

Classification of error control coding schemes[2]:



**Figure.2.3:** Classification of ECC

The two types of linear codes are,

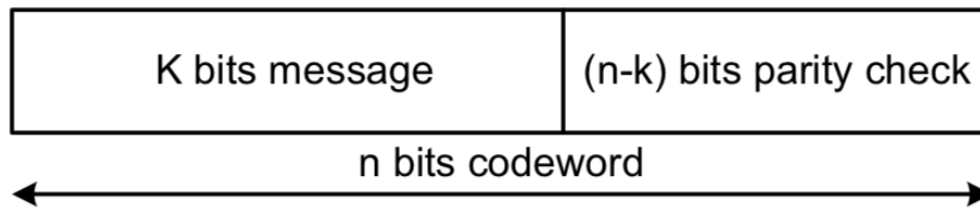
1. **Block codes:** These codes are referred to as “n” and “k” codes. A block of k data bits is encoded to become a block of n bits called a code word. In block codes, code words do not have any dependency on previously encoded messages. NAND Flash memory devices typically use block codes. Example: RS, Golay, Cyclic codes, Repetition codes, Polynomial codes.
2. **Convolution codes:** These codes produce code words that depend on both the data message and a given number of previously encoded messages. The encoder changes state with every message processed. Typically, the length of the code word constant. Example: Systematic codes, Nonsystematic codes.

#### 2.4.1 Linear Block Codes

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

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

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



**Figure. 2.4** Systematic form of codeword of a linear block code

Linear block codes are summarized by their symbol alphabets (e.g., binary or ternary) and parameters  $(n, m, d_{min})$  where

- $n$  is the length of the codeword, in symbols,
- $m$  is the number of source symbols that will be used for encoding at once,
- $d_{min}$  is the minimum hamming distance for the code.

There are many types of linear block codes, such as

1. Cyclic codes (e.g., Hamming codes)
2. Repetition codes
3. Parity codes
4. Polynomial codes (e.g., BCH codes)
5. Reed–Solomon codes
6. Algebraic geometric codes
7. Reed–Muller codes
8. Perfect codes

## 2.5 Galois Field (GF):

In this chapter finite fields[6] and finite field arithmetic operators are introduced. The definitions and main results underlying finite field theory are presented and it is shown how to derive extension fields. The various finite field arithmetic operators are reviewed. In addition, new circuits are presented carrying out frequently used arithmetic operations in decoders. These operators are shown to have faster operating speeds and lower hardware requirements than their equivalents and consequently have been used extensively throughout this project.

### 2.5.1 Properties of Galois Field

The main properties of a Galois field are:

1. A finite field always contains a finite number of elements and it must be a prime power, say  $q = p^r$ , where  $p$  is prime and  $r$  is unique. In Galois field  $GF(q)$ , the elements can take  $q$  different values. Field is another algebraic system.
2. All elements of  $GF$  are defined on two operations, called addition and multiplication.
3. The result of adding or multiplying two elements from the Galois field must be an element in the Galois field.
4. Identity of addition “zero” must exist, such that  $a + 0 = a$  for any element  $a$  in the field.
5. Identity of multiplication “one” must exist, such that  $a * 1 = a$  for any element  $a$  in the field.
6. For every element  $a$  in the Galois field, there is an inverse of addition element  $b$  such that  $a + b = 0$ . This allows the operation of subtraction to be defined as addition of the inverse.
7. For every non-zero element  $b$  in the Galois field, there is an inverse of multiplication element  $b^{-1}$  such that  $bb^{-1} = 1$ . This allows the operation of division to be defined as multiplication by the inverse.
8. Both addition and multiplication operations should satisfy the commutative, associative, and distributive laws.

### 2.5.2 Galois field $GF(2)$ “Binary Field”

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

**Table 2.1:** (a) Modulo-2 addition (XOR); (b) Modulo-2 Multiplication (AND)

+	0	1
0	0	1
1	1	0

(a)

×	0	1
0	0	0
1	0	1

(b)

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

$$f(x) = f_0 + f_1X + f_2X^2 + \dots + f_KX^K$$

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

### 2.5.3 Extension Fields

Finite fields exist for all prime numbers q and for all  $p^m$  where p is prime and m is a positive integer. GF(q) is a sub-field of GF( $p^m$ ) and as such the elements of GF(q) are a sub-set of the elements of GF( $p^m$ ), therefore GF( $p^m$ ) is an extension field of GF(q).

**Table 2.2:** (a) Addition for GF(4)={0,1,2,3};(b)Multiplication for GF(4)={0,1,2,3}

<b>+</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>*</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<b>0</b>	0	1	2	3	<b>0</b>	0	0	0	0
<b>1</b>	1	2	3	0	<b>1</b>	0	1	2	3
<b>2</b>	2	3	0	1	<b>2</b>	0	2	0	2
<b>3</b>	3	0	1	2	<b>3</b>	0	3	2	1

(a)

(b)

Consider GF(4)={0,1,2,3} in Table 2.2(a) and 2.2(b), which is not a Galois field because it is of order 4, which is not a prime. The element 2 has no multiplicative inverse and therefore we cannot divide by 2. Instead, we could define GF(4)={0,1,a,b} with addition and multiplication as shown in Table 2.3(a) and 2.3(b).

Now all elements do have additive and multiplicative inverses.

**Table 2.3:** (a)Addition for GF(4)={0,1,a,b};(b)Multiplication for GF(4)={0,1,a,b}

<b>+</b>	<b>0</b>	<b>1</b>	<b>a</b>	<b>b</b>	<b>*</b>	<b>0</b>	<b>1</b>	<b>a</b>	<b>b</b>
<b>0</b>	0	1	a	b	<b>0</b>	0	0	0	0
<b>1</b>	1	0	b	a	<b>1</b>	0	1	a	b
<b>a</b>	a	b	0	1	<b>a</b>	0	a	b	1
<b>b</b>	b	a	1	0	<b>b</b>	0	b	1	a

(a)

(b)

These extension fields are used to handle non-binary codes where code symbols are expressed as  $m$ -bit binary code symbols, For example, GF(4) consists of four different two-bit symbols and GF(16) of 16 hexadecimal symbols. To obtain multiplication for binary, numbers are expressed as polynomials, they are multiplied and divided by the prime polynomial while the remainders taken as result.

## 2.6 BCH Codes

BCH codes forms a class of random multiple error-correcting cyclic codes. defined over a Galois Field (GF) of  $q$  elements GF( $q$ ), with  $q=2^m$ . The parameter  $m$  corresponds to the degree of the GF,  $q$  is the number of states that takes each component of the GF elements, and they are related with the codeword length as  $n=2^m-1$ .

Binary BCH codes are identified by their codeword length  $n$ , their message length  $k$ , the maximum error capability of the code is  $t$ , and are represented as BCH ( $n, k, t$ )

For any positive integer  $m \geq 3$  (where  $3 \leq m \leq 16$ ) and  $t < 2^{m-1}$ , there exists a binary BCH code with the following parameters:

Block length:  $n = 2^m - 1$

Number of parity-check digits:  $n - k = mt$

Minimum distance:  $d_{\min} = 2t + 1$ .

BCH codes are subset of the Block codes. In block codes, the redundancy bits are added to the original message bits and the resultant longer information bits called “codeword” for error correction is transmitted. The block codes are implemented as  $(n, k)$  codes where  $n$  indicates the codeword and  $k$  the original information bits.

### 2.6.1 BCH Merits:

- It can be decoded using syndrome decoding method
- Highly flexible allowing control over block length and acceptable error thresholds
- Reed Solomon codes are nonlinear BCH codes used in applications such as satellite communication, compact disk players, DVD’s, disk drives, 2-dimensional bar code
- Low amount of redundancy
- Easy to implement in hardware
- Widely used

The parameters of some useful BCH codes of lengths less than  $2^{10}-1$  are given below:

**Table 2.4:** BCH codes parameters of lengths less than  $2^{10}-1$

m	n	k	t	m	n	k	t	m	n	k	t	n	k	t	n	k	t
3	7	4	1	63	24	7		127	50	13		255	187	9	255	71	29
4	15	11	1		18	10			43	14			179	10		63	30
		7	2		16	11			36	15			171	11		55	31
		5	3		<del>10</del>	<del>13</del>			29	21			163	12		47	42
5	31	26	1		7	15			22	23			155	13		45	43
		21	2	7	127	120	1		15	27			147	14		37	45
		16	3			113	2		8	31			139	15		29	47
		11	5			106	3	8	255	247	1		131	18		21	55
		6	7			99	4		239	2			123	19		13	59
6	63	57	1			92	5		231	3			115	21		9	63
	<del>51</del>	<del>2</del>				85	6		223	4			107	22	511	502	1
	45	3				78	7		215	5			99	23		493	2
	39	4				71	9		207	6			91	25		484	3
	36	5				64	10		199	7			87	26		475	4
	30	6				57	11		191	8			79	27		466	5

For t small  
n - k = mt

n	k	t	n	k	t	n	k	t	n	k	t	n	k	t
511	457	6	511	322	22	511	193	43	511	58	91	1023	933	9
	448	7		313	23		184	45		49	93		923	10
	439	8		304	25		175	46		40	95		913	11
	430	9		295	26		166	47		31	109		903	12
	421	10		286	27		157	51		28	111		893	13
	412	11		277	28		148	53		19	119		883	14
	403	12		268	29		139	54		10	121		873	15
	394	13		259	30		130	55		1013	1		863	16
	385	14		250	31		121	58	1023	1003	2		858	17
	376	15		241	36		112	59		993	3			
	367	16		238	37		103	61		983	4			
	358	18		229	38		94	62		973	5			
	349	19		220	39		85	63		963	6			
	340	20		211	41		76	85		953	7			
	331	21		202	42		67	87		943	8			

### 2.6.2 BCH Demerits:

- Complexity.
- Iterative complex decoding algorithm.
- Decoder cannot decide whether decoded package is false or not.

### 2.6.3 Some examples of BCH Applications:

- (511,493) BCH code is used in ITU-T Rec.H.261 video codec for video conferencing and video phone
- (40,32) BCH is used in ATM (Asynchronous Transfer Mode) it is a shorten cyclic code that can correct 1-bit or 2-bit error
- ECC in NAND Flash memory for reliable data storage

## 2.7 BCH Encoder

An encoder is a device, circuit, transducer, software program, or algorithm that converts the information from one format or code to another, for the purposes of standardisation, speed or compressions. A simple encoder assigns binary code to an active input line. The BCH encoder block creates a BCH code with message length  $k$  and codeword length  $n$ . The input must contain exactly  $k$  elements.  $n$  must have the form  $2^m - 1$  where  $m$  is an integer greater than or equal to 3.

### 2.7.1 Code generation:

To generate all the field elements a primitive polynomial in Galois Field :

In order to obtain the generator polynomial [6] of the BCH code we need an auxiliary polynomial called primitive polynomial. The generator polynomial is the polynomial of lowest degree over  $GF(2)$  with  $\alpha, \alpha^2, \alpha^3, \dots, \alpha^{2^t}$  as roots. Let  $m_i(x)$  be the minimal polynomial of  $\alpha^i$ . Then, must be the *least common multiple (LCM)* of  $m_1(x), m_2(x), \dots, m_{2^t}(x)$ . That is

$$g(x) = \text{LCM}\{m_1(x), m_2(x), \dots, m_{2^t}(x)\}$$

A simplification is possible because every even power of a primitive element has the same minimal polynomial as some odd power of the element, halving the number of factors in the polynomial. Then

$$g(x) = \text{LCM}\{m_1(x), m_3(x), \dots, m_{2^{t-1}}(x)\}$$

Hence, every even power of  $\alpha$  in the sequence has the same minimal polynomial as some preceding odd power of  $\alpha$  in the sequence. As a result, the generator polynomial  $g(x)$  of the binary  $t$ -error-correcting BCH code of length  $2^m - 1$  can be reduced to

$$g(x) = \text{LCM}\{\phi_1(x), \phi_3(x), \dots, \phi_{2^{t-1}}(x)\}$$



The generator polynomial is  $g(x) = 1 + g_1x + g_2x^2 + g_3x^3 + \dots + g_{n-k-1}x^{n-k-1}$

Code word  $c(x) = c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \dots + c_1x + c_0$

Data polynomial,  $d(x) = d_{k-1}x^{k-1} + d_{k-2}x^{k-2} + \dots + d_1x + d_0$

$$C(x) = d(x).g(x)$$

### 2.7.2 Primitive Polynomials:

A primitive polynomial is a polynomial that generates all elements of an extension field from a base field. Primitive polynomials are also irreducible polynomials. For any prime or prime power  $q$  and any positive integer  $n$ , there exists a primitive polynomial of degree  $n$  over  $GF(q)$ .

The primitive polynomial for various value of  $m$  is shown in table:

**Table 2.5:** Primitive polynomials for  $3 \leq m \leq 20$

M	Primitive Polynomial
3	$1 + x + x^3$
4	$1 + x + x^4$
5	$1 + x^2 + x^5$
6	$1 + x + x^6$
7	$1 + x^3 + x^7$
8	$1 + x^2 + x^3 + x^4 + x^8$
9	$1 + x^4 + x^9$
10	$1 + x^3 + x^{10}$
11	$1 + x^2 + x^{11}$
12	$1 + x + x^4 + x^6 + x^{12}$
13	$1 + x + x^3 + x^4 + x^{13}$
14	$1 + x + x^6 + x^{10} + x^{14}$
15	$1 + x + x^{15}$
16	$1 + x + x^3 + x^{12} + x^{16}$
17	$1 + x^3 + x^{17}$
18	$1 + x^7 + x^{18}$
19	$1 + x + x^2 + x^5 + x^{19}$
20	$1 + x^3 + x^{20}$

There are

$$a_q(n) = \frac{\phi(q^n - 1)}{n}$$

Primitive polynomials over GF(q), where  $\Phi(n)$  is the totient function. A polynomial of degree n over the finite field GF(2) is primitive if it has polynomial order  $2^n - 1$ .

### 2.7.3 Minimal Polynomials:

The even powers minimal polynomials are duplicates of odd powers minimal polynomials, so we only use the first two minimal polynomials corresponding to odd powers of the primitive element.

## 2.8 BCH Decoder

A BCH decoding system is designed for the correction of errors in the codeword. Some of the popular methods used for decoding are PGZ (Peterson-Gorenstein-Zierler) Algorithm, Berlekamp-Massey (BMA) algorithm and Euclidean (EA) algorithm. There are different Chien search algorithms for fast encoding like the Conventional p-parallel - Chien architecture, MPCN-based parallel architecture, Joint Chien Search & Syndrome-Calculator Architecture.

### 2.8.1 Algebraic Decoding

#### A. Peterson-Gorenstein-Zierler Decoding

If there are  $\nu$  errors, then the syndrome relationships (3) for  $j = 1, 2, \dots, n - k$  provide  $n - k$  equations involving the  $2\nu$  unknowns  $\sigma(i)$  and  $e_{\sigma(i)}$  for  $i = 1, 2, \dots, \nu$ . Since these equations are nonlinear, solving for these unknowns requires a clever trick. For  $\nu \leq t \leq \lfloor \frac{n-k}{2} \rfloor$ , let  $\Lambda(x) \triangleq \sum_{j=0}^t \Lambda_j x^j$  be any degree- $t$  polynomial that satisfies  $\Lambda(0) = 1$  and  $\Lambda(\beta^{-\sigma(i)}) = 0$  for  $i = 1, \dots, \nu$ . Then, the coefficients of  $\Lambda(x)$  have a linear relationship with the syndromes. This can be seen by summing the equation

$$0 = \Lambda(\beta^{-\sigma(i)}) = \Lambda_t (\beta^{-\sigma(i)})^t + \Lambda_{t-1} (\beta^{-\sigma(i)})^{t-1} + \dots + \Lambda_1 (\beta^{-\sigma(i)}) + \Lambda_0$$

for  $i = 1, \dots, \nu$  with the coefficients  $e_{\sigma(i)} \alpha^a (\beta^{\sigma(i)})^k$ . For  $k = t + 1, t + 2, \dots, 2t$ , this gives

$$\begin{aligned}
 0 &= \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \left( \beta^{\sigma(i)} \right)^k \Lambda \left( \beta^{-\sigma(i)} \right) \\
 &= \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \left( \beta^{\sigma(i)} \right)^k \sum_{j=0}^t \Lambda_j \left( \beta^{-\sigma(i)} \right)^j \\
 &= \sum_{j=0}^t \Lambda_j \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \left( \beta^{\sigma(i)} \right)^{k-j} \\
 &= \sum_{j=0}^t \Lambda_j S_{k-j}.
 \end{aligned} \tag{1}$$

The derivation implies that any polynomial  $\Lambda(x)$  with constant term 1 and roots at  $\beta^{-\sigma(i)}$  (i.e., the inverse of  $\alpha$  to the error location) for  $i = 1, \dots, \nu$  must satisfy this equation. The minimal-degree polynomial  $\Lambda(x)$  that satisfies these conditions is called the *error-locator polynomial*. It is easy to see that it must have one root at each location and is, therefore, the degree- $\nu$  polynomial defined by

$$\Lambda(x) \triangleq \prod_{i=1}^{\nu} \left( 1 - x \beta^{\sigma(i)} \right).$$

This polynomial allows the error position to be revealed by factoring  $\Lambda(x)$ .

Since  $\Lambda_0 = 1$ , (1) defines the linear system

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

where the  $i$ -th row is given by the equation for  $k = t + i$ . If  $t = \nu$ , then this matrix will be invertible because there is a unique solution. If it is not invertible, one can sequentially reduce  $t$  by 1 until the matrix becomes invertible. After solving for the error-locator polynomial, one can evaluate it at all points in  $\mathbb{F}_q^*$  to determine the error locations. An efficient method of doing this is called a *Chien search*.

For binary BCH codes, the error magnitudes must be 1. After correcting the “errors”, one must also check that the resulting vector is a codeword by reducing it modulo  $g(x)$ . If it is not a codeword, then the decoder should declare a detected error. For non-binary codes, one can solve for the error magnitudes using the error locations and the implied frequency-domain parity-check matrix[7]. For fixed error locations, one can write the first  $\nu$  syndromes as linear functions of the error magnitudes using

$$\begin{bmatrix} \alpha^a (\beta^{\sigma(1)})^1 & \alpha^a (\beta^{\sigma(2)})^1 & \dots & \alpha^a (\beta^{\sigma(\nu)})^1 \\ \alpha^a (\beta^{\sigma(1)})^2 & \alpha^a (\beta^{\sigma(2)})^2 & \dots & \alpha^a (\beta^{\sigma(\nu)})^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^a (\beta^{\sigma(1)})^\nu & \alpha^a (\beta^{\sigma(2)})^\nu & \dots & \alpha^a (\beta^{\sigma(\nu)})^\nu \end{bmatrix} \begin{bmatrix} e_{\sigma(1)} \\ e_{\sigma(2)} \\ \vdots \\ e_{\sigma(\nu)} \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_\nu \end{bmatrix}$$

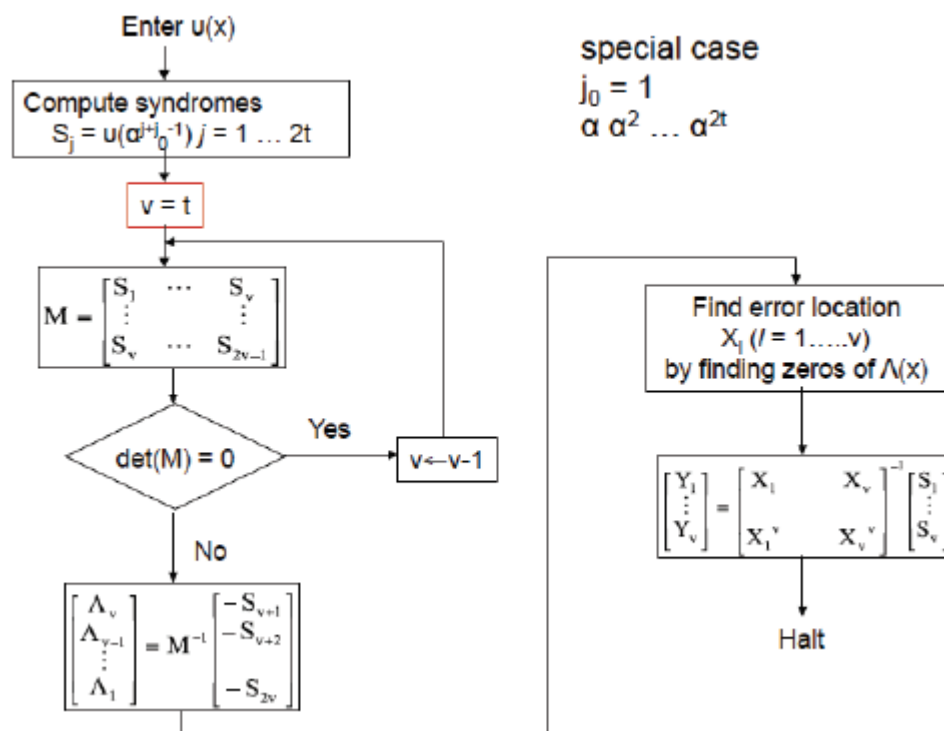


Figure 2.5: The Peterson-Gorenstein-Zierler Algorithm Flowchart

## B. The Berlekamp-Massey Decoding Algorithm

While the PGZ algorithm is conceptually simple, it can require the inversion of an  $i \times i$  matrix for  $i = 1, 2, \dots, t$  in the worst case. Since each inversion has a complexity of roughly  $i^3/2$  operations, this approach leads to a worst case complexity of roughly  $t^4/6$  operations. The Berlekamp-Massey algorithm [11] starts with the observation that (1) can be rewritten, for  $j = t + 1, t + 2, \dots, n - k$ , as

$$S_j = - \sum_{i=1}^t \Lambda_i S_{j-i}.$$

This implies that the syndrome sequence  $S_1, S_2, \dots$  can be generated by a linear feedback shift register (LFSR) with coefficients  $\Lambda_1, \Lambda_2, \dots, \Lambda_t$ . The shortest LFSR that generates the syndrome sequence is unique and corresponds to  $t = \nu$  and gives the error-locator polynomial.

The trick is to solve recursively for a sequence of LFSRs that generate the initial part of the syndrome sequence. Let the connection polynomial of a length- $L_k$  minimal length LFSR that generates the first  $k$  elements of the syndrome be

$$\Lambda^{[k]}(x) = \sum_{i=0}^{L_k} \Lambda_i^{[k]} x^i.$$

To be precise, we say that  $\Lambda^{[k]}(x)$  generates the first  $k$  elements if

$$S_j = - \sum_{i=1}^{L_k} \Lambda_i^{[k]} S_{j-i},$$

for  $j = L_k + 1, \dots, k$ . In particular, the shift register is initialized to contain the first  $L_k$  elements,  $S_1, S_2, \dots, S_{L_k}$ . Then, the  $j$ -th clock outputs  $S_j$  and computes  $S_{j+L_k}$  from  $S_1, S_2, \dots, S_{L_k}$ . This also introduces a subtle distinction between  $L_k$  and the degree of  $\Lambda^{[k]}(x)$ .

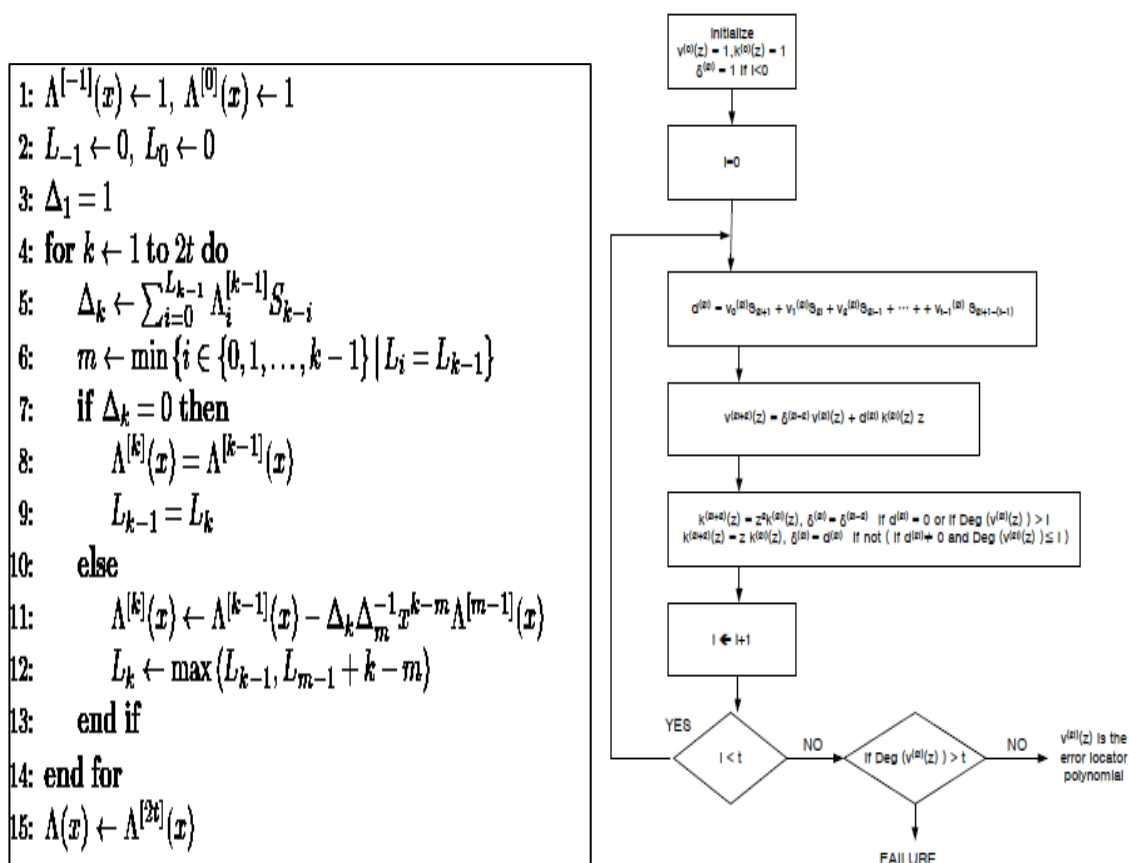


Figure 2.6: BMA Algorithm with flowchart

### C. Sugiyama's Euclidean Decoding Algorithm

An alternative approach is to use the Euclidean algorithm[12] to find the error-locator polynomial. To describe this, the syndrome is first extended to a semi-infinite sequence  $(S_1, S_2, \dots)$  by defining

$$S_j = \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \left( \beta^{\sigma(i)} \right)^j, \quad (3)$$

for  $j \in \{1, 2, \dots\}$  and noting that the two definitions coincide for  $j = 1, 2, \dots, n - k$ . The *extended syndrome function*  $\tilde{S}(x)$  is defined to be

$$\begin{aligned} \tilde{S}(x) &\triangleq \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \frac{\beta^{\sigma(i)}}{1 - x\beta^{\sigma(i)}} \\ &= \sum_{j=1}^{\infty} \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a x^{j-1} \left( \beta^{\sigma(i)} \right)^j \\ &= \sum_{j=1}^{\infty} S_j x^{j-1}. \end{aligned}$$

It is important here to comment on the meaning of infinite sums over finite fields. Unlike the continuous case, no two distinct points can be considered close to one another. Therefore, convergence in the limit is the same as eventual equality. Thus, the second and third equalities do not hold for evaluations but instead imply that the (infinite) power series expansions of the two expressions match term by term.

The *error-evaluator polynomial*  $\Omega(x) \triangleq \sum_{j=0}^{\nu} \Omega_j x^j$  is given by

$$\begin{aligned} \Omega(x) &\triangleq \Lambda(x) \tilde{S}(x) \\ &= \prod_{j=1}^{\nu} \left( 1 - x\beta^{\sigma(j)} \right) \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \frac{\beta^{\sigma(i)}}{1 - x\beta^{\sigma(i)}} \\ &= \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \beta^{\sigma(i)} \prod_{j \neq i} \left( 1 - x\beta^{\sigma(j)} \right). \end{aligned}$$

It is easy to see that  $\Omega(x)$  has degree at most  $\nu - 1$ . The polynomial  $\Omega(x)$  is called the error-evaluator polynomial because

$$\begin{aligned} \Omega\left(\beta^{-\sigma(k)}\right) &= \sum_{i=1}^{\nu} e_{\sigma(i)} \alpha^a \beta^{\sigma(i)} \prod_{j \neq i} \left( 1 - \beta^{-\sigma(k)} \beta^{\sigma(j)} \right) \\ &= e_{\sigma(k)} \alpha^a \left[ \beta^{\sigma(k)} \prod_{j \neq k} \left( 1 - x\beta^{\sigma(j)} \right) \right]_{x=\beta^{-\sigma(k)}} \\ &= -e_{\sigma(k)} \alpha^a \Lambda' \left( \beta^{-\sigma(k)} \right), \end{aligned}$$

where, using the product rule, the formal derivative of  $\Lambda(x)$  is given by

$$\Lambda'(x) = - \sum_{i=1}^{\nu} \beta^{\sigma(i)} \prod_{j \neq i}^{\nu} (1 - x\beta^{\sigma(j)})$$

Using  $\Omega(x)$ , one can compute the error magnitudes using

$$e_{\sigma(k)} = - \frac{\Omega(\beta^{-\sigma(k)})}{\alpha^a \Lambda'(\beta^{-\sigma(k)})}. \quad (4)$$

This approach is known as Forney's method.

The decoder is required to compute both the error-locator and error-evaluator polynomials from the finite syndrome polynomial  $S(x) \triangleq \sum_{j=1}^{2t} S_j x^{j-1}$ . Since  $S_e(x) = S(x) + x^{2t}w(x)$ , for some  $w(x)$ , we find that

$$\Omega(x) = \Lambda(x)S_e(x) = \Lambda(x)S(x) + \Lambda(x)x^{2t}w(x)$$

and  $\deg(\Omega(x)) < \nu$ . Thus, we arrive at the *key equation* for RS decoding which is given by

$$\Omega(x) \equiv \Lambda(x)S(x) \pmod{x^{2t}}.$$

In fact, if  $\nu \leq t$ , then any degree- $\nu$  polynomial  $\Theta(x)$  that satisfies  $\deg(\Theta(x)S(x) \pmod{x^{2t}}) < \nu$  must also satisfy  $\Theta(x) = c\Lambda(x)$  [4, Prop. 6.1]. Therefore, this equation can also be used to find error-locator and error-evaluator polynomials.

The extended Euclidean algorithm (EEA) computes the greatest common divisor of two elements  $a_1, a_2$  from a Euclidean domain  $E$  (e.g., a ring of polynomials over a field) and coefficients  $u, v \in E$  such that  $a_0u + a_1v = \gcd(a_1, a_2)$ . The algorithm proceeds by dividing  $a_j$  by  $a_{j+1}$  so that  $a_j = a_{j+1}q_{j+1} + a_{j+2}$  with quotient  $q_{j+1}$  and remainder  $a_{j+2}$ . Each step of the Euclidean algorithm works because the division implies that  $\gcd(a_j, a_{j+1}) = \gcd(a_{j+1}, a_{j+2})$ . For polynomials, the Euclidean algorithm terminates when  $a_j = 0$ . This always occurs because  $\deg(a_2) < \deg(a_1)$  holds by assumption and  $\deg(a_{j+2}) < \deg(a_{j+1})$  holds by induction.

The extended algorithm also computes  $u_j, v_j$  recursively so that  $a_j = u_j a_1 + v_j a_2$ . Starting from  $a_3 = a_1 - q_2 a_2$  (i.e.,  $u_3 = 1$  and  $v_3 = -q_2$ ), we have the recursion

$$a_{j+2} = a_j - q_{j+1} a_{j+1} = (u_j a_1 + v_j a_2) - q_{j+1} (u_{j+1} a_1 + v_{j+1} a_2).$$

This gives the recursions  $u_{j+2} = u_j - q_{j+1} u_{j+1}$  and  $v_{j+2} = v_j - q_{j+1} v_{j+1}$  starting from  $u_3 = 1$  and  $v_3 = -q_2$ .

The decoding the RS codes is accomplished using a partial application of the EEA algorithm to compute  $\gcd(x^{2t}, S(x))$ . The extended part of the algorithm generates a sequence of relationships of the form  $u_j(x)x^{2t} + v_j(x)S(x) = a_j(x)$ ,

where the degree of  $a_j(x)$  is decreasing with  $j$ . Let  $j^*$  be the first step where  $\deg(a_j(x)) < t$  and stop the algorithm at this point. Viewing the above relationship as a congruence modulo  $x^{2t}$  gives  $v_j(x)S(x) \equiv a_j(x) \pmod{x^{2t}}$ , and we see that  $v_{j^*}(x)$  and  $a_{j^*}(x)$  satisfy the key equation with  $\deg(a_{j^*}(x)) < t$ . In this case, the polynomials  $v_{j^*}(x), a_{j^*}(x)$  must also satisfy

$$\begin{aligned} v_{j^*}(x) &= c\Lambda(x) \\ a_{j^*}(x) &= c\Omega(x), \end{aligned}$$

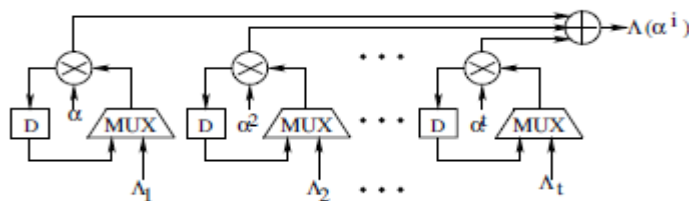
for some constant  $c$ . This means that we can run the EEA until the remainder term has degree less than  $t$ . After that, we can solve for  $c$  using  $c = v_{j^*}(0)$  and compute  $\Lambda(x), \Omega(x)$ . After the error-locator and error-evaluator polynomials are known, decoding proceeds by factoring  $\Lambda(x)$  to find the error locations and then using (4) to compute the error magnitudes.

### 2.8.2 Chien Search Algorithms

Once  $\Lambda(x)$  is found, the decoder searches for error locations by checking whether  $\Lambda(\alpha^i) = 0$  for  $0 \leq i \leq (n - 1)$ , which is normally achieved by Chien search. A conventional serial Chien search architecture is shown in Fig. 2.7(a), and

$$\Lambda(\alpha^i) = \sum_{j=0}^t \Lambda_j \alpha^{ij} = \sum_{j=1}^t \Lambda_j \alpha^{ij} + 1$$

where  $0 \leq i \leq (n - 1)$ . All the multiplexers select  $\Lambda(x)$  in the first clock cycle, then select the registered data afterwards.



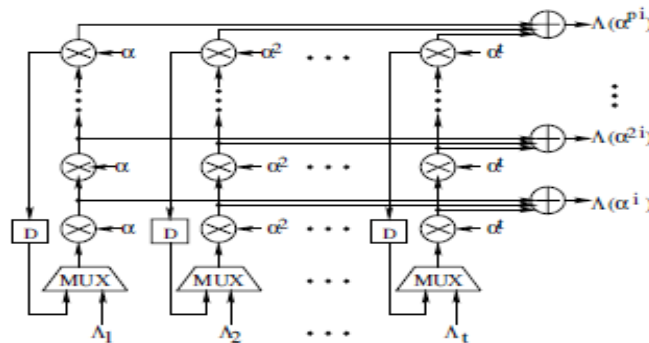
**Figure 2.7(a)** : Conventional Chien search circuit

Since all the  $n$  possible locations have to be evaluated for the  $\Lambda(x)$ , it takes  $n$  clock cycles to complete the Chien search process. To speed up this process, parallel Chien search architecture that evaluates several locations per clock cycle is essential. Two different

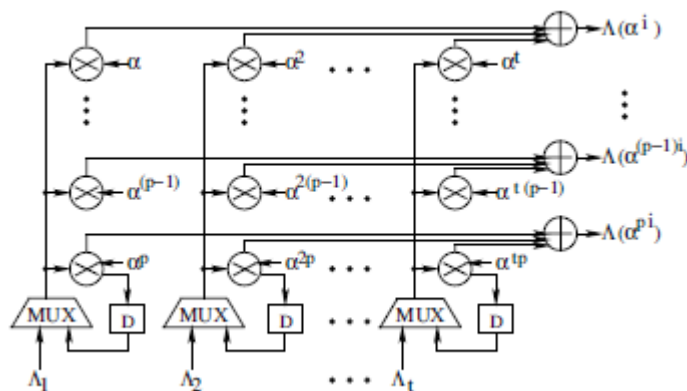


possible architectures [13] with parallel factor  $p$  are depicted in Fig. 2.7(b) and Fig. 2.7(c), where Fig. 2.7(b) actually is just a direct unfolded version of Fig. 2.7(a) with an unfolding factor of  $p$ .

As both designs can reduce the number of clock cycles searching for error locations from  $n$  down to  $\lceil n/p \rceil$ , they also share similar hardware complexity. Denoting the parallel factor as  $p$ , both designs have the exactly same  $(p \times t)$  constant finite field multipliers (FFM),  $p$   $t$ -input  $m$ -bit finite field adders (FFA),  $p$   $m$ -bit registers and  $p$   $m$ -bit multiplexers. However, the critical path of Fig. 2.6(b) is  $(T_{mux} + p \times T_m + T_a)$  while it is only  $(T_{mux} + T_m + T_a)$  for Fig. 2.7(c), where  $T_{mux}$ ,  $T_m$  and  $T_a$  stand for the critical path of multiplexer, FFM and  $t$ -input  $m$ -bit FFA, respectively. Obviously, once the parallel factor  $p$  is greater than 1, much faster clock speed could be achieved for the design in Fig. 2.7(c) than that in Fig. 2.7(b). For example, assuming  $T_m$  is dominant, critical path of Fig. 2.7(c) is  $p$  times shorter.



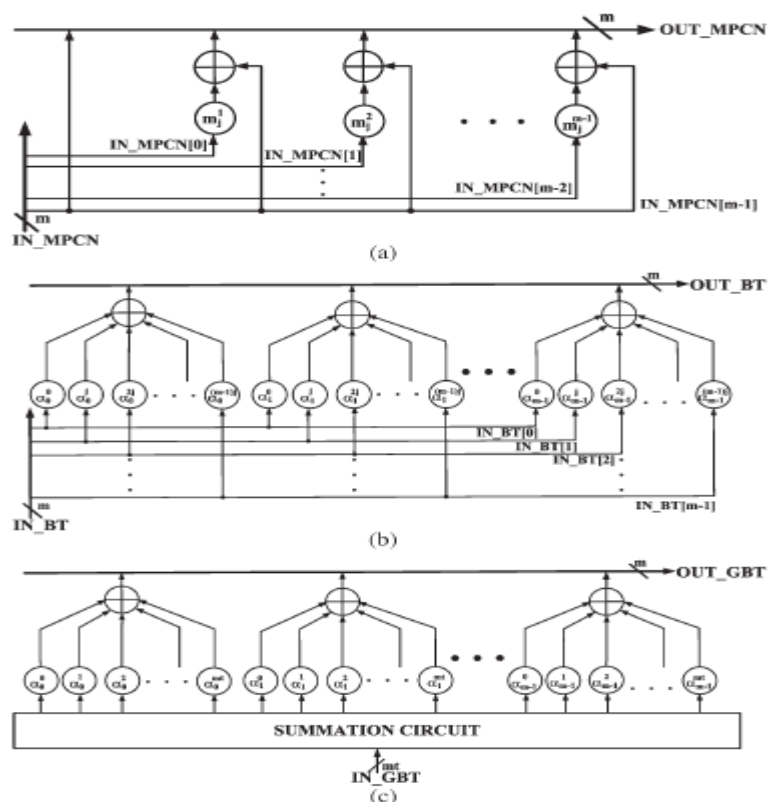
**Figure 2.7(b)** :  $p$ -parallel Chien search architecture: direct unfolded version



**Figure 2.7(c)** :  $p$ -parallel Chien search architecture: equivalent architecture with shorter critical path

Fig. 2.8 shows the architectures of three basis components, including the  $j$ th MPCN, the  $j$ th BT, and the GBT. The  $j$ th MPCN  $MPCN_j$  shown in Fig. 2.8(a) executes modulo operation with divisor  $M_j(x)$ . It is constructed by the combinational circuit of the linear feedback shift

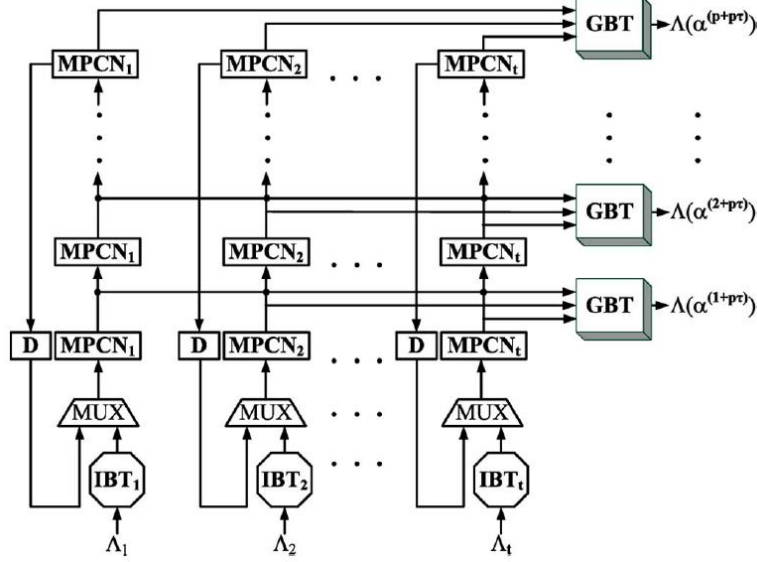
register with the connection polynomial  $M_j(x)$ . Each binary element  $mkj$  in Fig. 2.8(a) is the  $k$ th coefficient of  $M_j(x)$ , indicating the wire connection. In the  $j$ th BT shown in Fig. 2.8(b), each  $akl$  is a binary element and can be represented whether the wire is connected or not. Fig. 2.8(c) illustrates the block diagram of the GBT. The additions are first executed with all the coefficients of  $D_j(x)$  for  $j = 1 \sim t$  (total  $mt$  bits), and the similar operations as a BT are applied with basis  $\alpha_0 \sim \alpha_{mt}$ .



**Figure 2.8** Basic components in Chien search architecture. (a) MPCN<sub>j</sub>. (b) BT<sub>j</sub>. (c) GBT.

In the MPCN-based parallel- $p$  Chien search architecture shown in Fig. 2.9, the coefficients of  $\Lambda(x)$  are applied to the IBTs for transforming the operating basis. The transformed values are evaluated with minimal polynomials for obtaining the Chien search results. All the multiplexers select the outputs of IBTs in the first cycle and then select the register data afterward. Searching from the  $(N - 1)$ th to zeroth location, the proposed design checks  $p$  locations at each cycle. In each row,  $mt$ -bit data are fed into a GBT to examine the error locations. An error is found at the  $(N + r - p(\tau + 1) - 1)$ th location if the output of the  $r$ th-row GBT is equal to zero at the  $\tau$ th cycle. MPCN Chien search architecture utilizes  $p \times t$  MPCNs to replace  $p \times t$  CFFMs. Notice that the XOR gate count requirement of one MPCN

is at most  $m - 1$ , which is much smaller than that of one CFFM. Therefore, it is area efficient to apply the MPCNs, particularly in the large parallelism conditions.

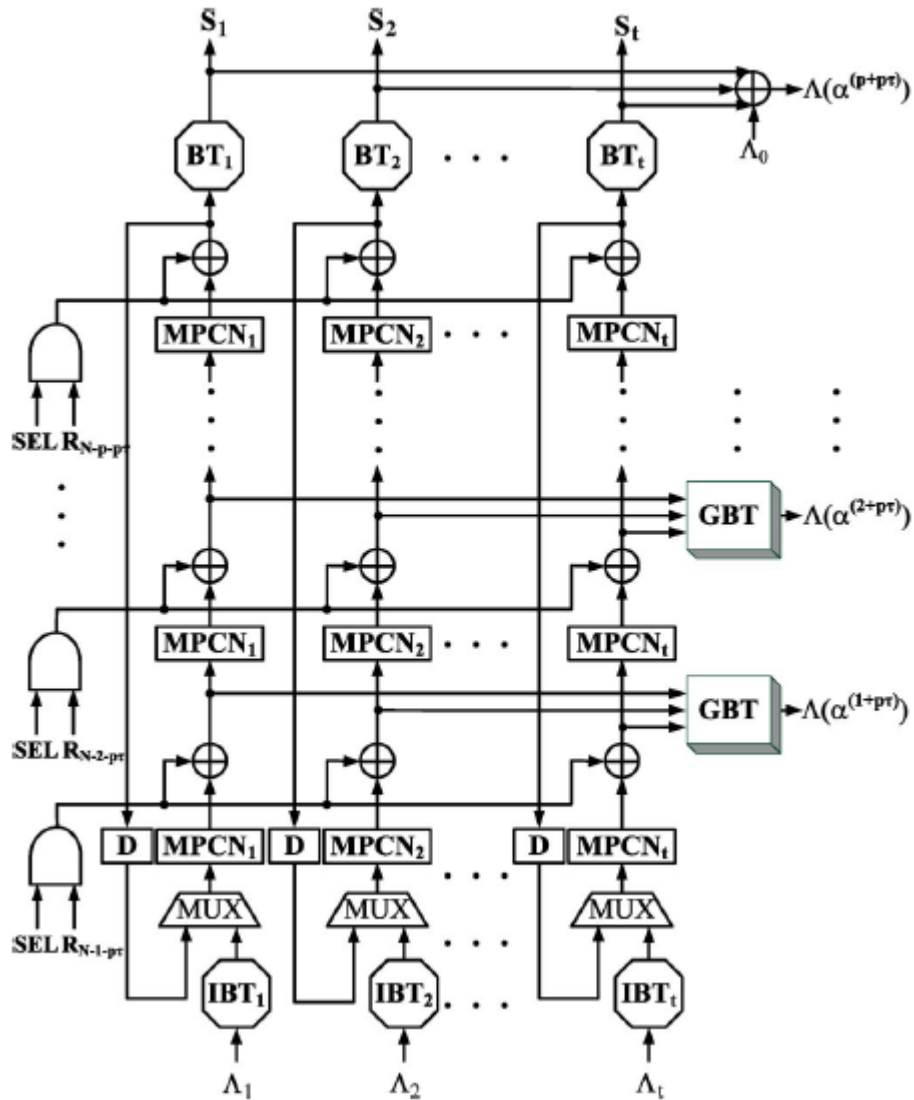


**Figure 2.9** MPCN-based parallel-p Chien search architecture.

The MPCN-based architecture can merge the syndrome calculator and the Chien search in the same hardware with small overhead. Fig. 2.10 illustrates parallel-p joint syndrome calculator and Chien search with the MPCN-based architecture. The syndrome calculator and Chien search phases are determined by the SEL signal. When the SEL signal is high, the  $j$ th syndrome value is formulated as

$$S_j = (((R_{N-1}x^{p-1} + \dots + R_{N-p-1}) \bmod M_j(x)) x^p + (R_{N-p-2}x^{p-1} + \dots + R_{N-2p-1}) \bmod M_j(x)) x^p + \dots + R_{p-1}x^{p-1} + \dots + R_0) \bmod M_j(x) |_{x=\alpha^j}$$

The partial remainder stored in the register is multiplied by  $x^p$  and accumulated with the received symbols. After all the received symbols are processed, BT $_j$  transforms the accumulated result to the  $j$ th syndrome value. In contrast with Fig. 2.9,  $t$  BTs are applied instead of one GBT in the first row to evaluate individual syndrome value. Note that the FFA in Fig. 2.10 is only a 1-bit operation because each coefficient of  $R(x)$  is a binary value. Therefore, except for the difference between the BT and the GBT, the overhead of the supporting syndrome calculation is only  $p$  NAND and  $p \times t$  XOR gates.

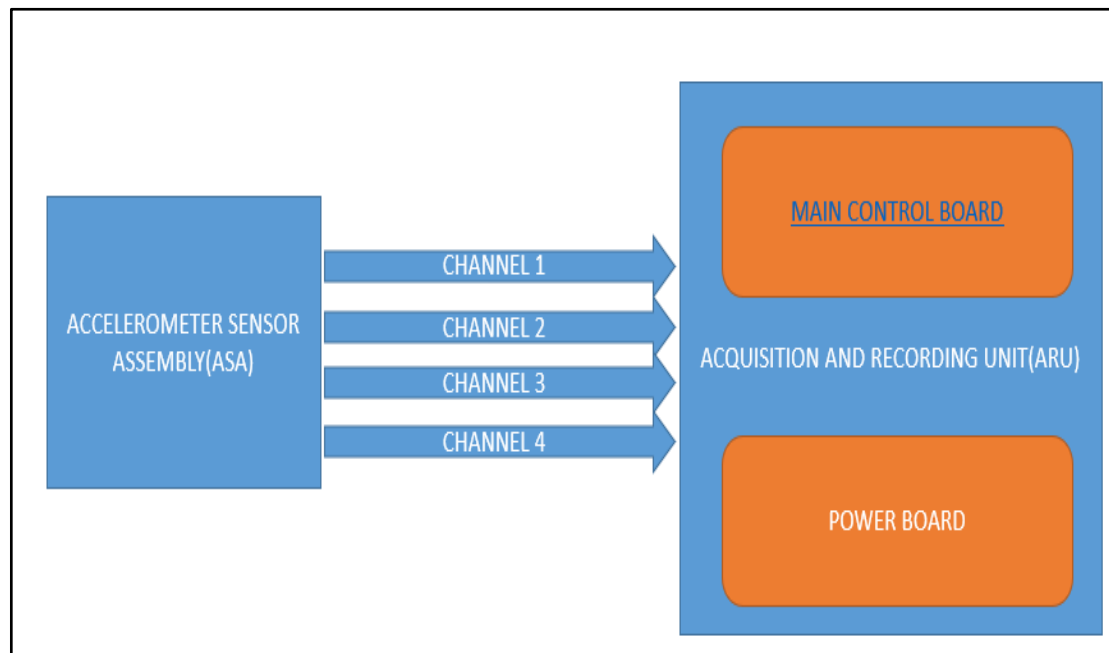


**Figure 2.10** Parallel- $p$  joint syndrome calculator and Chien search with MPCN based architecture.

## Chapter 3

# HARDWARE

### 3.1 Acquisition and Recording Unit (ARU)



**Figure 3.1** ASA-ARU system interface with ASA-LRU

The main objective of ASA-ARU is used to store the 12 Accelerometer Sensors data. The data is stored in the digital but the sensors outputs are analog. So, to convert analog to digital ADC's are required. ASA has 12 sensor outputs. Hence, two ADC's are used. NAND FLASH is used to store the acquired data. RTC with on-board memory is used for time-stamp and storing data. To retrieve the data from the NAND FLASH Ethernet is used. RS232 is used to retrieve the data from the RTC. ASA-ARU uses ADC, NAND FLASH, RTC, Ethernet physical layer and RS232 transceiver.

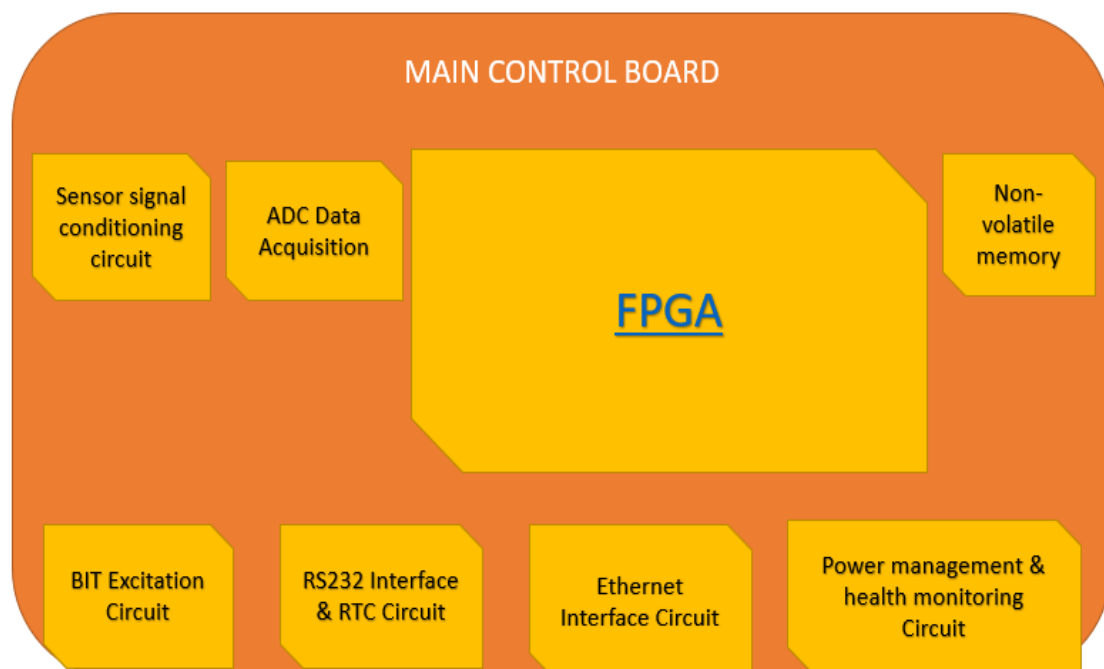
The ASA-ARU will acquire and record the 12-accelerometer sensors data of ASA continuously. The recorded data can be later retrieved for analysis purpose at ground. The ASA-ARU contains RS-232 interface, which shall be used to debug the system and Ethernet interface for data retrieval.

The objective of the ASA-ARU is used to store the 12 Accelerometer Sensor Data, store in NAND FLASH Memory during Flight, retrieve the data from NAND FLASH Memory and

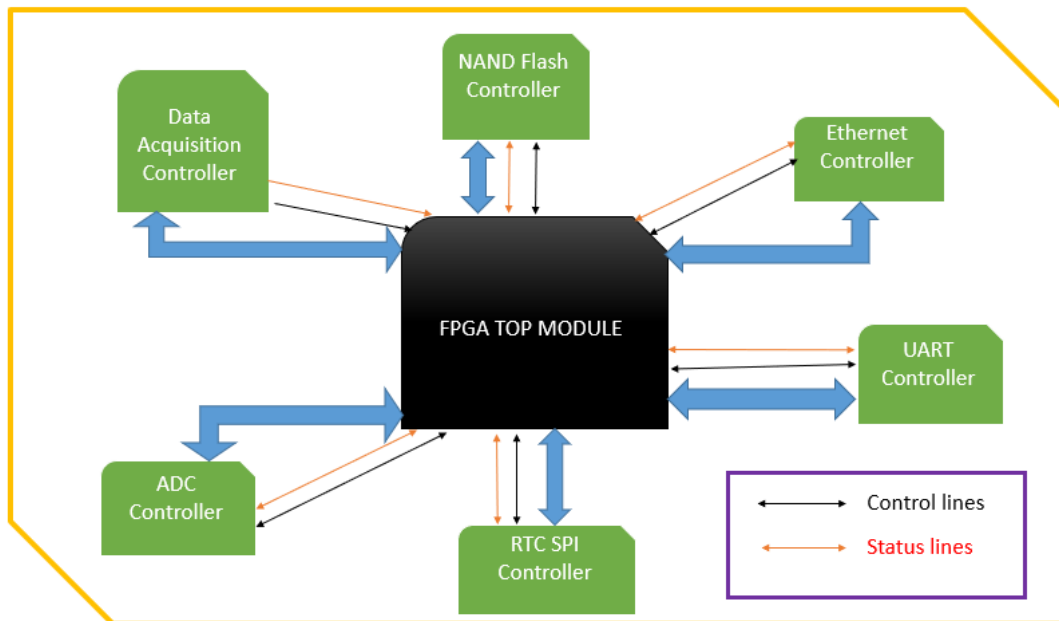
send to Rugged Laptop through Ethernet and RS232 during POST flight. In ASA-ARU, FPGA is communicating with ADC, RTC Memory, NAND FLASH Memory, Ethernet Physical Layer and RS232 Driver. The below sections gives the related information regarding System Initialization, Power on Self Test, Mode Identification, Acquisition Mode, Retrieval Mode, Inside FPGA modules, FPGA Constraints and Safety Considerations.

### 3.2 System Initialization

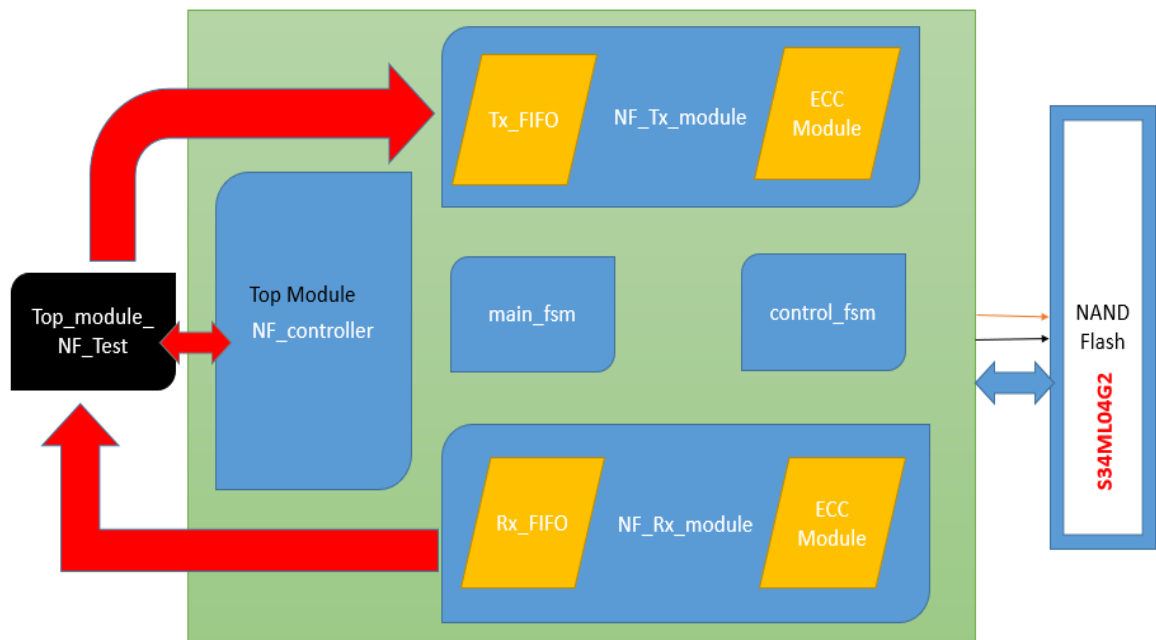
ASA-ARU has ADC, NAND FLASH Memory, RTC with on-board Memory, Ethernet Physical Layer and RS232 transceiver. Each one is initialized using FPGA sub Modules such as Data Acquisition Controller, NAND FLASH Controller and RTC SPI Controller, Ethernet Controller, UART Controller and all these modules are controlled by the Top Module.



**Figure 3.2** ASA-ARU system-Main Control Board



**Figure 3.3** Controller modules of FPGA



**Figure 3.4** NAND Flash Controller module

### 3.3 NAND Flash Controller

Top module will enables operations like page read, page write, reset operation, readID (manufacture ID) operation and block erase operation.

1. Once FPGA\_RESET\_n is deasserted, POST operation will starts automatically.  
Check flash health status .
2. Check flash health is good and ASA-ARU health is good, go for the mode identification operation else exit the function.
3. If Acquisition mode (mode=0), It enables the bad block checking operation by making signal BB\_FSM\_en = 1, else Retrieval Mode (mode=1). In retrieval mode it enables the bad blocks read operation from the RTC controller by making BB\_retrieval\_FSM\_en = 1.
4. During Acquisition mode:
  - i. If BB\_FSM\_en = 1 it enables the bad block checking operation by making page\_BB\_read\_en =1. This signal enables the bad block page read operation. Read 1<sup>st</sup> location of NF spare memory data. check for BB if block is bad block update the BB's address into BB\_FIFO memory. Check for all blocks (4096); once completion of this operation enables the BB\_opr\_comp BB\_opr\_comp it. Enable the program operation by making progms\_fsm\_en = 1 .
  - ii. Once Progm\_fsm\_en = 1, Read NF write completed page address from the RTC then read acquisition data from the DAQ controller. Assign page completion address to NF page\_address and block completion address to NF block\_address. Check if daq\_stop = 0 send DAQ\_FIFO\_rd\_data along with DAQ\_FIFO\_data\_valid signal and page\_write\_en=1. Wait for write\_complete=1 once write\_complete=1, check NF page and block address with completed page and block address if it not equal compare current address with bad block address if it not equal go for DAQ\_stop check and repeat the process up to daq\_stop = 1 else updated the NF address in the BB\_FIFO memory and compare next BB address with NF address.
  - iii. Once daq\_stop=1, read session data from RTC controller and send to NF on last two pages.
5. During Retrieval mode:
  - i. BB management operation :- If BB\_retrival\_fsm\_en = 1 read BB's address from RTC and store into BB\_FIFO memory. Once stored the all BB's into BB\_FIFO it generate BB\_FIFO\_filled = 1. BB\_FIFO\_filled it Enable the read operation by making read\_fsm\_en = 1.



- ii. Once read\_FSM\_en = 1, read BB address from DAQ controller and store into BB\_FIFO. Compare BB address with NF block address, if address are equal increment NF address by 1 and compare with BB address else address are different read data from NF by send Page\_read\_en = 1. Wait for Read\_complete=1 repeat the process .
- iii. Once page complete address and NF block and page address are equal read last two blocks first page data by enabling page\_read\_en =1.
- iv. After reading last two blocks 1<sup>st</sup> page data generate erase\_FSM\_en = 1 .
- v. Once erase\_FSM\_en = 1, it will wait for the NF\_block\_erase\_en = 1. Once NF\_block\_erase\_en = 1, compares NF block address with BB block address if both are equal increment NF block address by 1 and compare with next BB address else both address are not equal erase NF block repeat the process.

### 3.1.1 Data integrity for ASA-ARU:

The data recorded in-flight is fed to the actuators which then leads to the smooth functioning of the flight without any turbulence. The slightest change in this data, would lead to a large deviation from the original position, causing the flight to topple in the worst case. Thus, to avoid such disasters, the data from the accelerometer has to be accurate and reliable throughout the whole process of giving an input, processing and providing the feedback for the normal functioning of the whole system.

### 3.2 FPGA-ARTIX 7:

- FPGA hardware: FPGA[14] is the main data processing hardware of the system. It is used for controlling various functions of the ASA-ARU such as POST analysis, Mode Identification, Data Acquisition Controlling, recording data into NAND Flash memory, retrieving data from NAND Flash memory, Transmitting retrieved data to GSE via Ethernet Interface and UART Controller.
- FPGA Clock: A 25 MHz clock is used for FPGA operations and peripherals controlling.

- Artix-7 FPGAs are available in -3, -2, -1, -1LI, and -2L speed grades, with -3 having the highest performance. The Artix-7 FPGAs predominantly operate at a 1.0V core voltage.
- FPGA is used as a controller in ASA-ARU system. FPGA hardware used in ASA-ARU system is XC7A100T-2FTG256I - ARTIX 7 FPGA. Artix 7 FPGA Hardware Description Language (HDL) developed in the 'VHDL' language. The operating environment details of the HDL are given in the below Table.
- VHDL coding standard followed for Artix 7 FPGA is “VHDL coding standards for programmable hardware used in the development of software system of LCA”.

**Table 3.1** Operating Environment of Hardware Description Language

Sl. No	Platform	Details
1	Firmware type	Hardware Description Language (HDL)
2	FPGA (Simulation, Synthesis, Implementation, Download)	Artix7- XC7A100T-2FTG256I (ISE Design Suit 14.6, ISE Design Suit 14.6(XST), ISE Design Suit 14.6, iMPACT (or) Vivado Tools 2014.2
3	IDE for Firmware	ISE Design Suit 14.6 (or) Vivado Tool 2014.2
4	Programming Language	“VHDL” Language
5	GSE Software	LabWindow\CVI
6	GSE Software Language	‘C’ Language
7	Windows OS	7 or high

**OVERVIEW:**

- Artix-7 address the complete range of system requirements, ranging from low cost, small form factor, cost-sensitive, high-volume applications to ultra high-end connectivity bandwidth, logic capacity and signal processing capability for the most demanding high-performance applications. Artix-7 Optimized for lowest cost and power with small form-factor packaging for the highest volume applications.

**Table 3.2** I/O Pin/Device/Package Combinations for Artix-7 FPGAs

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48 E1 Slices	Block RAM Blocks			CMT	PCIe	GTP	XADC Blocks	Total I/O Bank	Max User I/O
		Slices	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7A35T	33280	5200	400	90	100	50	1800	5	1	4	1	5	250
XC7A50T	52160	8150	600	120	150	75	2700	5	1	4	1	5	250
XC7A75T	75520	11800	892	180	210	105	3780	6	1	4	1	6	300
XC7A100T	101440	15850	1188	240	270	135	4860	6	1	4	1	6	300
XC7A200T	215360	33650	2888	740	730	365	13140	10	1	4	1	10	500

**a. CLBs, Slices, and LUTs:**

Some key features of the CLB architecture include:

1. Real 6-input look-up tables (LUTs)
2. Memory capability within the LUT
3. Register and shift register functionality

The LUTs in Artix-7 can be configured as either one 6-input LUT (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs. Each LUT output can optionally be registered in a flip-flop. Four such LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and

two slices form a configurable logic block (CLB). Four of the eight flip-flops per slice (one per LUT) can optionally be configured as latches.

Between 25–50% of all slices can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s. Modern synthesis tools take advantage of these highly efficient logic, arithmetic, and memory features.

#### b. Block RAM:

Some of the key features of the block RAM include:

1. Dual-port 36 Kb block RAM with port widths of up to 72
2. Programmable FIFO logic
3. Built-in optional error correction circuitry

Artix-7 has between 135 and 4860 dual-port block RAMs, each storing 36 Kb. Each block RAM has two completely independent ports that share nothing but the stored data.

### 3.3 NAND Flash Memory

- NAND is the most popular type of flash storage memory for USB flash drives, memory cards, and SSDs. It is used in some of the best SSDs in the market today.
- This flash memory technology is non-volatile chip-based storage, and unlike DRAM does not require a persistent power source. NAND cell arrays store 1, 2, 3, or 4 bits of data. When the NAND SSD or card is detached from a power source, metal-oxide semiconductors called floating-gate transistors (FGT) provide electrical charges to the memory cells, and data remains intact.
- It stores data in memory cell arrays that are defined by transistors. Each transistor has two gates instead of one, like an electrical switch where the current flows between two points. A floating gate and a control gate control the energy flow in a flash memory cell. The control gate captures electrons and moves them as needed into the floating gate.
- NAND flash development concentrates on reducing the size of the chips while maintaining or increasing their capacity. This reduces bit costs and increases

density. Another feature is connecting cells in series of FTGs, which takes less space than parallel connections and further reduces NAND flash costs.

### 3.3.1 Types of NAND:

The most common types of NAND are between cells containing 1, 2, or 3 bits a cell. We call these SLC, MLC, and TLC. 3D NAND is also gaining ground and high-performance, high density environments[15].

- ❖ **SLC:** Single-Level Cell stores 1 bit in each cell.
- ❖ **MLC:** Multi-Level Cell stores 2 bits per cell.
- ❖ **eMLC:** Enterprise Multi-Level Cell increases MLC endurance
- ❖ **TLC:** Triple-Level Cell stores 3 bits per cell. However, advances in 3D NAND and sophisticated controllers are positioning TLC to perform in read-heavy enterprise applications.
- ❖ **QLC:** Quad-Level Cells store 4 bits per cell. However, increasing density by storing more bits per cell has serious disadvantages. The more bits per cell, the more often writes and erasures occur in the cell, which decreases endurance. Voltage is also an issue in QLCs, since voltage changes cause instability in surrounding cells.
- ❖ **3D NAND:** Flash manufacturers are on a mission to decrease cell sizes in order to pack more chips and thus more capacity on a NAND device. However, shrinking cells using the above cell level technologies resulted in cell to cell interference, which reduced data integrity in NAND flash.

**Table 3.3** Characteristic Comparison of NAND and NOR

Characteristic	NAND Flash	NOR (Q-Flash)
Random access read	25 $\mu$ s (first byte) .03 $\mu$ s each for remaining 2111 bytes	.12 $\mu$ s
Sustained read speed (sector basis)	23 MB/s (x8) or 37 MB/s (x16)	20.5 MB/s (x8) or 41MB/s (x16)
Random write speed	~300 $\mu$ s/2112 bytes	180 $\mu$ s/32 bytes
Sustained write speed (sector basis)	5 MB/s	.178 MB/s

PARAMETER	NAND	NOR
Erase block size	128KB	128K8
Erase time per block (typ)	2ms	750ms
<b>Deployment</b>	More widely used	Moderately used
<b>Memory cell connections</b>	The cells are connected in series, do not allow direct writes to individual memory cells.	The cells are connected in parallel, the system can write and read to individual memory cells
<b>Read performance</b>	Reads are slower since it supports page and block access, not random access.	Allows random access to any memory address. This allows the system to read bytes independently of pages and blocks.
<b>Write and erasure performance</b>	Writes and erasures are faster in NAND with its smaller cell sizes.	Writes and erasures are slower on NOR's larger cells.
<b>Endurance</b>	NAND cells typically have 98% good bits when shipped, and end-users know to expect additional bit failures over the cell's lifetime.  NAND manufacturers usually add error correcting code (ECC)	NOR cells have 100% known good blocks over the life of the cell. NOR does not need any error correcting code (ECC).
<b>Density</b>	Ranges between 1Gb to 16Gb.	Ranges from 64MB to 2Gb

**Table 3.4** Parametric Comparison of NAND and NOR

## Chapter 4

# SOFTWARE

### 4.1 ECC for NAND Flash

Due to the Manufacturing issues, usage and environmental factors, data stored in NAND FLASH may not return its data value as written. However, the probability for this happening is very small. ECC is the good way to recover the wrong value from the remaining good data bits. NAND FLASH manufactures recommended using ECC for NAND FLASH would give better performance with reducing bit errors. ECC is implemented either hardware or software. Implementation of ECC through software reduces the hardware components and power. For software, various Algorithms are used to find ECC such as Hamming Algorithm, Reed Solomon Algorithm, BCH Algorithm, for two bit error detection and one bit error correction done by Hamming Algorithm, for multiple error detection and correction by Reed Solomon Algorithm and BCH Algorithm.

Each page in NAND FLASH is divided into Main area and Spare area as shown in Fig.4.1A. Main area consists of 2048 bytes and spare area consists of 128 bytes. Calculated ECC values are stored in Spare area. The main area is divided into four chunks. Each chunk is 512 bytes as shown in Figure 4.1B . Calculate ECC for each chunk and store in Spare area. Spansion recommended[16] 4-bit ECC for each 512 bytes data.

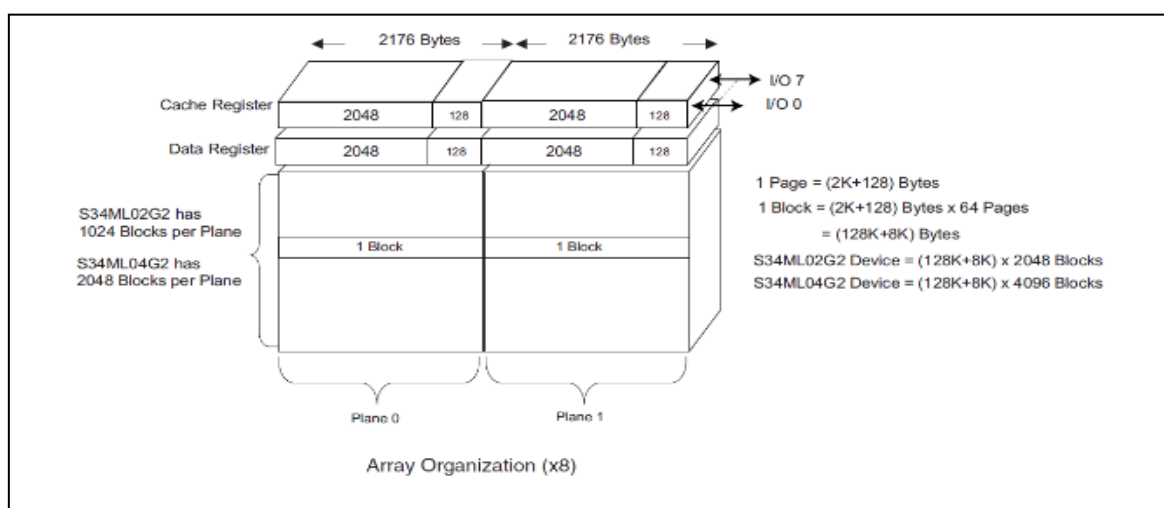
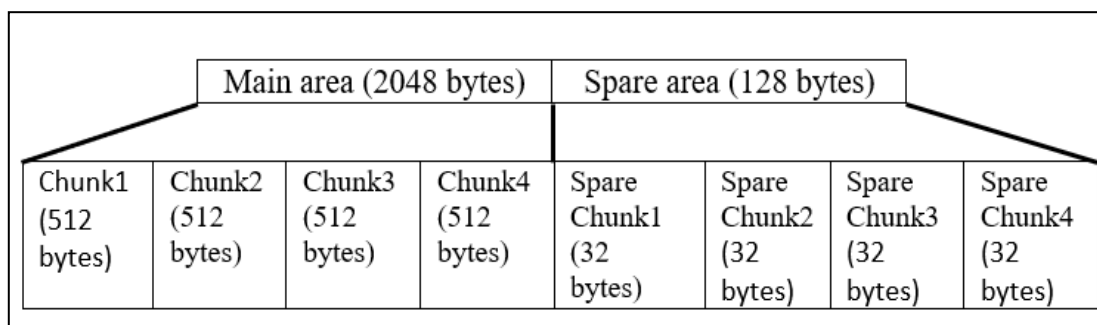


Figure 4.1A: NAND FLASH Array Organisation



**Figure 4.1B:** Main area and its division

**Table 4.1** Recommended BCH Code

ECC Selection Factor	NAND Flash Error Characteristics	Hamming	Reed-Solomon	Binary BCH
<b>Error Correction</b>	Some error detection beyond the correction power of the code improves system performance.	Error detection is not inherent in the code, but can be added by increasing overhead.	An arbitrary level of additional error detection is possible.	An arbitrary level of additional error detection is possible.
<b>Length of error patterns</b>	Bit errors are un-correlated. When given a bit error, the probability of any other bit being in error is not increased.	Not applicable since it only corrects 1-bit errors.	Efficient code for correlated error patterns (such as burst errors).	Efficient code for un-correlated error patterns (1-bit errors).
<b>Distribution of error patterns</b>	Errors are randomly distributed within a page.	Good for single-bit random errors.	Good for randomly distributed error patterns.	Good for randomly distributed error patterns.
<b>Frequency of errors</b>	Raw error rates vary by technology, but errors are relatively rare.	Unacceptable when the raw error rate is greater than $10^{-9}$ .	Applies when error rates are less than approximately $10^{-4}$ .	Applies when error rates are less than approximately $10^{-4}$ .

#### 4.1.1 Data Recording – NAND Flash

Functional Requirements:

It has to store the 12 channels data into NAND Flash.

240 minutes of the ADC sampled data has to store in NAND Flash.

Design approach:

Memory size required for storing data with 640 samples/sec:

$$\begin{aligned}
 &1 \text{ samples/sec for 1 channel} &&= 2 \text{ bytes} &&= 16 \text{ bits/sec} \\
 &640 \text{ samples/sec for 1 channel} &&= 640 * 16 &&= 10,240 \text{ bits/sec} \\
 &640 \text{ samples/sec for 12 channels} &&= 12 * 10240 &&= 1,22,880 \text{ bits/sec} \\
 &640 \text{ samples/minutes for 12 channels} &&= 60 * 122880 &&= 73,72,800 \text{ bits/sec} \\
 &640 \text{ samples/240minutes for 12 channels} &&= 240 * 7372800 &&= 1,76,94,72,000 \text{ bits/sec} \\
 &&&&&&\approx 1770 \text{ Mbits}
 \end{aligned}$$



So, Finally 4 Gbits NAND Flash selected based on above calculations - S34ML04G200TFI000.

## 4.2 BCH Codes in ASA-ARU Application

S34ML04G200TFI000 NAND FLASH requires 4-bit ECC per (512 bytes + 52 bits) of data, which is 4148 bits. BCH code requires 52 parity bits.

We have,

$$n = 4096 \text{ (data area)} + 52 \text{ (parity bits)} \leq 2^m - 1 \rightarrow m = 13$$

GF of 'w' elements  $GF(w)$ ,  $w=2^m$  are used to characterise BCH codes. The GF degree is given by m, the quantity of states taking each integrant over GF elements is given by w, and codeword length is given by  $n=2^m-1$ .

BCH (n, k, t) is the binary BCH code representation. The terms n, t and k give the codeword-length, maximum error capability of the code and message-length respectively [17].

For any non-negative numeral,  $m \geq 3$  (where  $3 \leq m \leq 16$ ) and also  $t < 2^m - 1$ , BCH code of given specifications exists [17]:

Codeword-length:  $n = 2^m - 1$

Message-length:  $k \geq n - mt$

Minimum distance:  $d_{min} \geq 2t + 1$ .

The MLC flash memories organisation for which the BCH encoder is implemented is as shown in Figure 4.1A. There are two planes each having 4096 blocks and each block consists of 64 pages. Each page has a main area of 2048 bytes and 128 spare bytes. The main area is further broken down into chunks each of size 512 bytes and calculated ECC for each chunk is stored in Spare area as shown in Figure 4.1B.

Based on the above specifications of memory, below are the parameters for binary BCH code:

As we saw, 1 chunk=512B=4096 bits ( $2^{12}$ ), block length,

$n \geq 2^m-1$ . Thus, considering  $m=13$ , we get  $n=8191$ . For a 4-bit error correcting BCH code,  $t=4$ . Using the above values in the following equation,  $k > n-mt$ , we get  $k= 8139$ . Therefore, the parity check (n-k) bits = 52bits.

### 4.3 BCH Encoder

Inputs of BCH Encoder are clock, reset, input data valid, 8-bit data. Outputs from BCH Encoder are 8-bit parity data with parity data valid.

#### SYSTOLIC ARRAY STRUCTURE

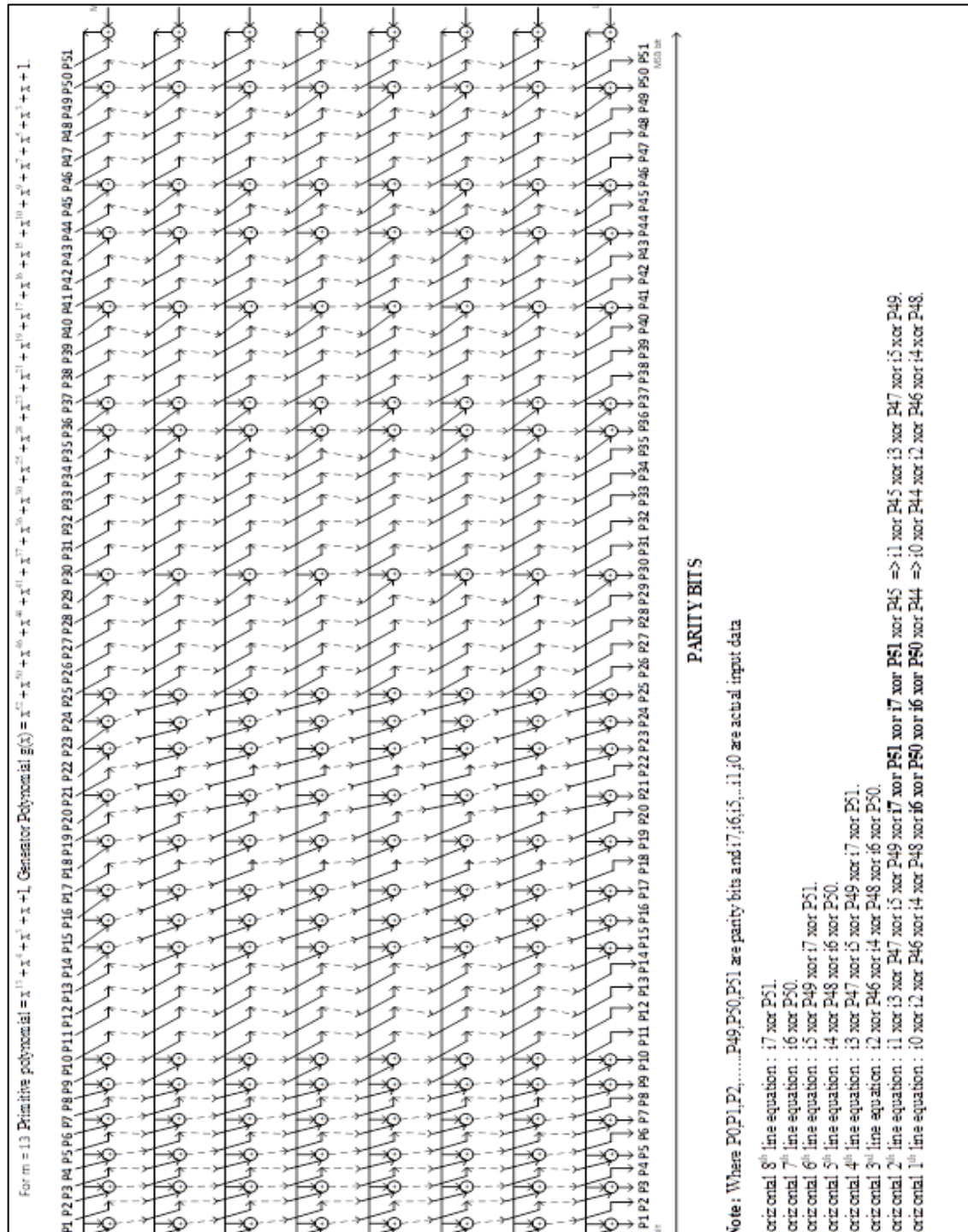


Figure 4.2 Hardware Systolic Array Type BCH Encoder

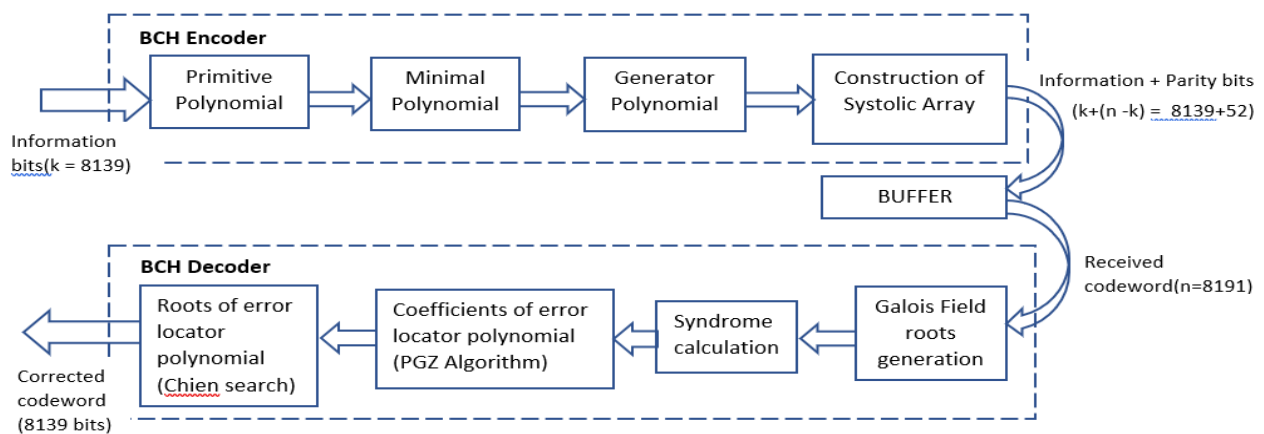


Figure 4.3 Block Flow Diagram of BCH Encoder-Decoder

The BFD of BCH encoder is as shown in Figure 4.3. The design of proposed encoder follows these stages.

### 4.3.1 Design and Implementation of Systolic- Array type Binary BCH Encoder

#### A. Generator Polynomial

The codeword( $n$ -bit) in  $BCH(n,k,t)$  is  $(C_{x_{n-1}}, C_{x_{n-2}}, \dots, C_{x_0})$ ,

$C_{x_i} \in GF(2)$ ,  $(0 \leq i \leq n-1)$  and message( $k$ -bit) is  $(m_{k-1}, m_{k-2}, \dots, m_0)$ ,  $m_i \in GF(2)$ ,  $(0 \leq i \leq k-1)$ .

Generator polynomial,  $g(x)$  is of  $(n-k)$  degree. The expression,  $cx(x) = m(x)g(x)$  gives the encoding of BCH codes in terms of  $g(x)$ .

$$g(x) = g_{n-k-1} x^{n-k-1} + \dots + g_3 x^3 + g_2 x^2 + g_1 x + 1$$

$$cx(x) = c_{x_{n-1}} x^{n-1} + c_{x_{n-2}} x^{n-2} + \dots + c_{x_1} x + c_{x_0}$$

$$m(x) = m_{k-1} x^{k-1} + m_{k-2} x^{k-2} + \dots + m_1 x + m_0$$

lowest degree polynomial of over  $GF(2)$  with roots as  $\beta, \beta^2, \beta^3, \dots, \beta^{2t}$  (also known as primitive elements) is known as generator polynomial [6]. Let  $f_i(x)$  be the minimal polynomial of  $\alpha_i$ . Then,  $g(x)$  is give as:

$$g(x) = LCM\{f_1(x), f_2(x), \dots, f_{2t}(x)\} \quad (1)$$

The conjugate roots of  $g(x)$  have same minimal polynomials .i.e.  $\beta^i = (\beta^i)^{2^w}$ ,  $f_i(x) = f_{i'}(x)$ , where  $i = i' * 2^w$  for  $w \geq 1$ , thus for a BCH code with  $t$ -error rectification,  $g(x)$  in eqn.(1) can be turndown to:

$$g(\kappa) = LCM\{f_1(\kappa), f_3(\kappa), \dots, f_{2^t-1}(\kappa)\} \quad (2)$$

For BCH (8191,8139,4), the primitive polynomial for  $GF(2^{13})$  is given as

$$p(\kappa) = \kappa^{13} + \kappa^4 + \kappa^3 + \kappa + 1. \quad (3)$$

The minimal polynomials for  $GF(2^{13})$  in binary BCH codes are:

$$\left. \begin{aligned} f_1(\kappa) &= \kappa^{13} + \kappa^4 + \kappa^3 + \kappa + 1 \\ f_3(\kappa) &= \kappa^{13} + \kappa^{10} + \kappa^9 + \kappa^7 + \kappa^5 + \kappa^4 + 1 \\ f_5(\kappa) &= \kappa^{13} + \kappa^{11} + \kappa^8 + \kappa^7 + \kappa^4 + \kappa + 1 \\ f_7(\kappa) &= \kappa^{13} + \kappa^{10} + \kappa^9 + \kappa^8 + \kappa^6 + \kappa^3 + \kappa^2 + \kappa + 1 \end{aligned} \right\} \quad (4)$$

Therefore from eqn. (2) and (4),  $g(\kappa)$  is computed as:

$g(\kappa) = LCM(f_1(\kappa), f_3(\kappa), f_5(\kappa), f_7(\kappa))$ , where

$$\begin{aligned} g(\kappa) &= \kappa^{52} + \kappa^{50} + \kappa^{46} + \kappa^{44} + \kappa^{41} + \kappa^{37} + \kappa^{36} + \kappa^{30} + \kappa^{25} + \kappa^{24} + \kappa^{23} + \kappa^{21} + \kappa^{19} + \kappa^{17} + \kappa^{16} \\ &+ \kappa^{15} + \kappa^{10} + \kappa^9 + \kappa^7 + \kappa^5 + \kappa^3 + \kappa + 1 \end{aligned} \quad (5)$$

## B. Construction of BCH (8191,8139,4) Encoder

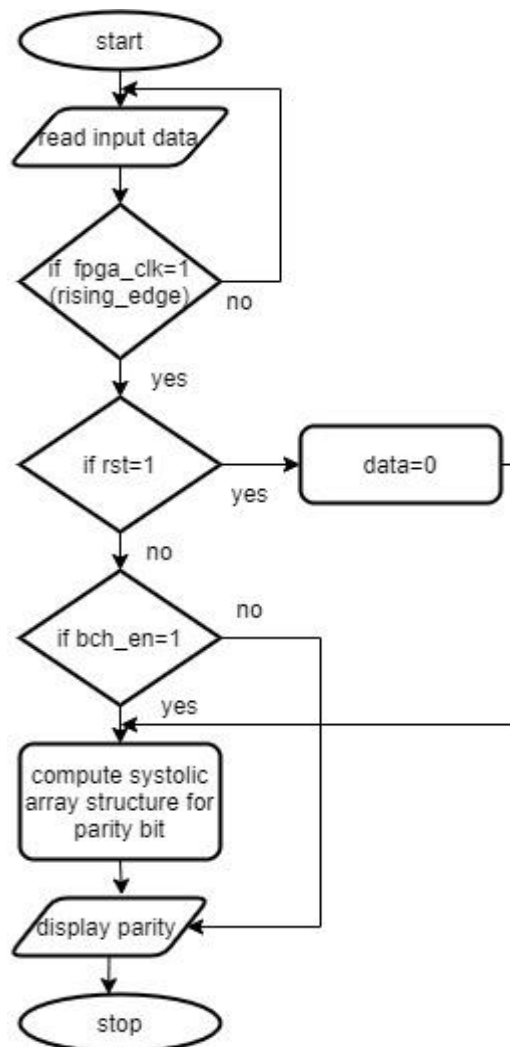
The proposed BCH encoder is nearly an equivalent of the traditional serial BCH encoder in which the yield of XOR situated furthest to the right acts an input to remaining XORs in addition to first register but the difference being, the further stages input is taken from the previous stage output. Each of the horizontal lines in Figure. 3 are considered as parallel factor  $j$ , where 0th stage is the first and  $(j-1)$  is the last stage. In each stage, the orientation of the XOR gates is an imitation of the preliminary stage but the contribution to each stage is from the preceding stage [6]. This procedure majorly operates on shifting mechanism. The  $(j-1)$ th stage output is fed as input to the (0th) stage, and is reshaped sequentially once the XOR operation is completed.

The procedure to calculate the parity bits for 512 bytes is as follows. Let 'P' is the parity bits consists of 52 bits, named as P0, P1, ..., P5, P6, ..., P49, P50, P51 and 'i' is the input bits consisting of 8 bits named as i0, i1, i2, ..., i6, i7. Initially parity bits 'P' are set to zeros. The first input byte data is loaded in i register which is 8-bit register connected to hardware systolic array type BCH Encoder when enable is high. Now the parity bits 'P' value is changed according to the 'i' register and Hardware digital circuit. The parity bits output for the input byte are input to the next byte in the field of parity bits 'P'. The next input byte is loaded in i register.

The parity bits 'P' corresponding data 'i' are calculated. Continue to load the input bytes and the preceding parity bits till 512 bytes are completed. After completion, parity bits for the 512 bytes are available in the parity bit register.

One major advantage of systolic array type architecture is that the stages can be changed to any number causing no effect on the complexity of the circuit as the stages are just replicas of the first. Hence, in the proposed BCH encoder implemented for the reliability of storage of data in NAND Flash memory, this architecture provides the ease of encoding the 2048 bytes of main area without increasing hardware design and thus preferred for high speed applications with maximum reutilisation of modules.

Once the message is encoded successfully, the information bits, along with the parity bits are now ready to enter the storage area. Once stored, the data can be retrieved into the buffer byte-wise and the decoding process is initiated.



**Figure 4.4** Flow diagram for proposed BCH Encoder

## 4.4 BCH Decoder

Inputs of BCH Decoder are clock, reset, data (stored data + parity data), data valid.

Outputs from BCH Decoder are No of errors, valid error enables, Error byte location, Error bit location

### 4.4.1 Design and Implementation Binary BCH Decoder

#### A. Galois Field roots generation

The root table generation is the primary stage for decoding. The values of roots initially are 0,  $\beta^0$  and  $\beta$ . Raising  $\beta$  to increasing powers, we get  $\beta^2, \beta^3, \dots, \beta^{11}$  and  $\beta^{12}$  as the following roots to initial vales. When  $\beta^{13}$  is encountered, knowing  $p(x)$  from eqn. (3), we get,

$$\beta^{13} = \beta^4 + \beta^3 + \beta + 1. \quad (6)$$

The primitive polynomial,  $p(x)$  aids in generating elements for the extension field ( $GF(2^m)$ ) from base field ( $GF(2)$ ). Any power beyond 13 (i.e  $\beta^{14}$  to  $\beta^{8190}$ ) is reduced using eqn. (6).

#### B. Syndrome calculation

Once the  $\beta^i$  is generated, the phase two is calculation of the syndromes( $S_i$ ). The indication of the received code being valid or not is given by this syndrome computation, that is, if syndrome is zero( $S_i = 0$ ), the received codeword is error-free.

**Table 4.2** Root Table for GF ( $2^{13}$ )

Coefficients	Root values
$\beta^0$	0000000000001
$\beta^1$	0000000000010
.	.
$\beta^{12}$	1000000000000
$\beta^{13}$	0000000011011
.	.
$\beta^{8190}$	1000000001101

Received codeword,  $rx(\kappa)$  is the syndrome module's input. The  $rx(\kappa)$  may be erroneous with a pattern  $er(\kappa)$ .

$$rx(\kappa) = cx(\kappa) + er(\kappa) \quad (7)$$

The received codeword is:

$$rx(\kappa) = rx_0 + rx_1\kappa + \dots + rx_{n-1}\kappa^{n-1} \quad (8)$$

Transmitted codeword is given by:

$$cx(\kappa) = cx_0 + cx_1\kappa + \dots + cx_{n-1}\kappa^{n-1} \quad (9)$$

The error pattern is:

$$er(\kappa) = er_0 + er_1\kappa + \dots + er_{n-1}\kappa^{n-1} \quad (10)$$

Syndrome  $S_i$  can be computed by:

$$S_i = rx(\beta^i)$$

$$rx(\beta^i) = rx_0 + rx_1 \beta^i + rx_2 \beta^{2i} + rx_3 \beta^{3i} + \dots + rx_{n-1} \beta^{(n-1)i} \quad (11)$$

where  $1 \leq i \leq 2t-1$  and  $\beta$  is the primitive element of  $GF(2^{13})$ .

### C. Coefficients of error locator polynomial

Some of the popular methods used for decoding are PGZ (Peterson-Gorenstein-Zierler) Algorithm [7], Berlekamp- Massey ( BMA ) algorithm [11] and Euclidean ( EA ) algorithm[12].

Among these, the most effortless way to comprehend a BCH decoder is by using the PGZ algorithm for any error correction capacity(t), that is, any decoder can be realised without the requirement of algebraic computation with high level of difficulty.

The error locator polynomial is defined as:

$$\lambda(\beta^i) = \lambda_0 + \lambda_1\beta^i + \lambda_2\beta^{2i} + \lambda_3\beta^{3i} + \lambda_4\beta^{4i} + \dots + \lambda_t\beta^{ti} \quad (12)$$

Here, we use the PGZ Algorithm to determine the  $\lambda(\beta^i)$  coefficients, i.e. the Eigen values from the determinant of the  $\Lambda_1, \Lambda_2, \dots, \Lambda_v$  of polynomial,

$$\Lambda(\kappa) = 1 + \Lambda_1\kappa + \Lambda_2\kappa^2 + \dots + \Lambda_v\kappa^v \quad (13)$$

For 4-bit( $v=4$ ) error correction, the following equations are used to calculate the eigen values:

$$\Lambda_1 = S_1 \tag{14}$$

$$\Lambda_2 = \frac{s_1(s_7+s_1^7)+s_3(s_5+s_1^5)}{s_3(s_3+s_1^3)+s_1(s_5+s_1^5)} \tag{15}$$

$$\Lambda_3 = (S_3 + S_1^3) + S_1\Lambda_2 \tag{16}$$

$$\Lambda_4 = \frac{(s_5+s_3s_1^2)+\Lambda_2(s_3+s_1^3)}{s_1} \tag{17}$$

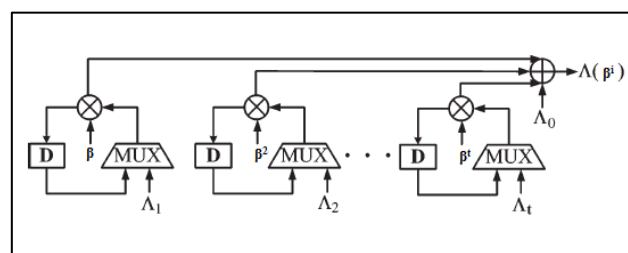
Newton's Identities [11] are used to verify the generated Eigen values ( $\Lambda_v$ ) with the below set of equations:

$$\begin{aligned} S_1 + \Lambda_1 &= 0 \\ S_2 + \Lambda_1 S_1 + 2\Lambda_2 &= 0 \\ \cdot & \\ \cdot & \\ \cdot & \\ S_v + \Lambda_1 S_{v-1} + \dots + \Lambda_{v-1} S_1 + v\Lambda_v &= 0 \\ S_{v+1} + \Lambda_1 S_v + \dots + \Lambda_{v-1} S_2 + \Lambda_v S_1 &= 0 \\ \cdot & \\ \cdot & \\ \cdot & \\ S_{2t} + \Lambda_1 S_{2t-1} + \dots + \Lambda_{v-1} S_{2t-v+1} + \Lambda_v S_{2t-v} &= 0 \end{aligned} \tag{18}$$

#### D. Roots of $\Lambda(x)$

Once the coefficients,  $\Lambda_v$  are computed, Polynomial,  $\Lambda(x)$  roots are to be determined which indicates the reciprocal of error location.

There are different Chien search algorithms for fast encoding like the Conventional p-parallel - Chien architecture , MPCN-based parallel architecture [13], Joint Chien Search & Syndrome-Calculator Architecture .



**Figure 4.5** Conventional Chien search



The hardware implementation of Chien search block [8] used proposed decoder for determining the roots of  $\Lambda(x)$  is shown in Figure. 5.

$$\Lambda(x) = \Lambda_t x^t + \Lambda_{t-1} x^{t-1} + \dots + \Lambda_1 x + \Lambda_0 \quad (19)$$

Element  $\beta^i$  is root of  $\Lambda(x)$  if the below condition is satisfied:

$$\Lambda(\beta^i) = \Lambda_t \beta^{it} + \Lambda_{t-1} \beta^{i(t-1)} + \dots + \Lambda_1 \beta^i + \Lambda_0 = 0 \quad (20)$$

Knowing,  $\Lambda_0 = 1$

$$\Lambda(\beta^i) - 1 = \Lambda_t \beta^{it} + \Lambda_{t-1} \beta^{i(t-1)} + \dots + \Lambda_1 \beta^i = -1 \quad (21)$$

For element  $\beta^{(i+1)}$ :

$$\Lambda(\beta^{(i+1)}) - 1 = \Lambda_t \beta^{(i+1)t} + \Lambda_{t-1} \beta^{(i+1)(t-1)} + \dots + \Lambda_1 \beta^{i+1} \quad (22)$$

$$\Lambda(\beta^{(i+1)}) - 1 = \Lambda_t \beta^{it} \beta^t + \Lambda_{t-1} \beta^{i(t-1)} \beta^{t-1} + \dots + \Lambda_1 \beta^i \beta \quad (23)$$

Value of  $\Lambda(\beta^{(i+1)})$  is computed from previous  $\Lambda(\beta^i)$  values. The eqn. (23), when iteratively executed, yields the roots of  $\Lambda(x)$ .

As mentioned earlier, the inverse roots now obtained are reciprocated to determine the bit locations at which the error occurred while transmission. Based on the error correcting capacity(t), those number of bits will be corrected.

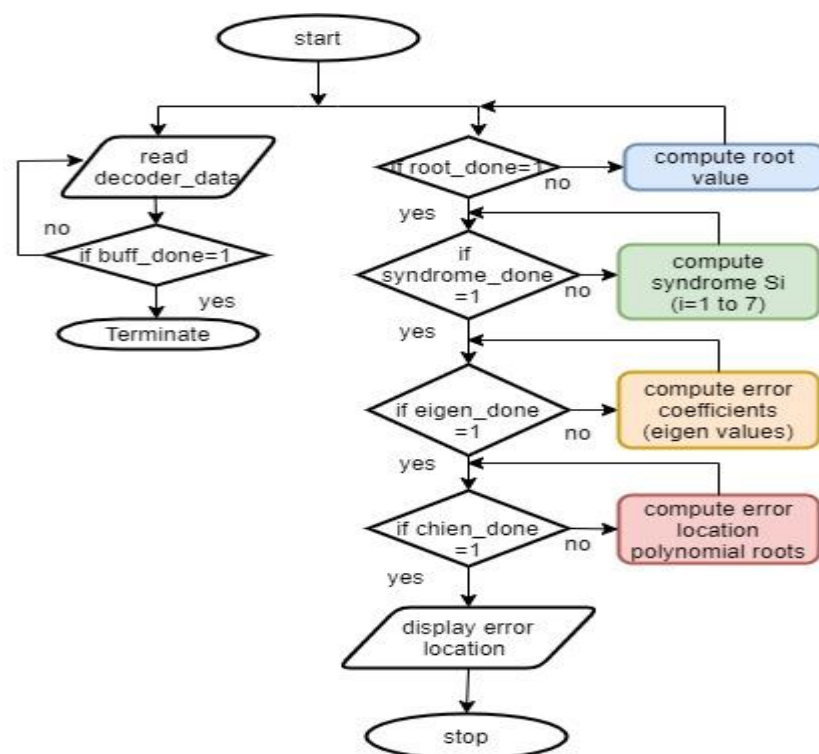


Figure 4.6 Flow diagram for proposed BCH Decoder

## Chapter 5

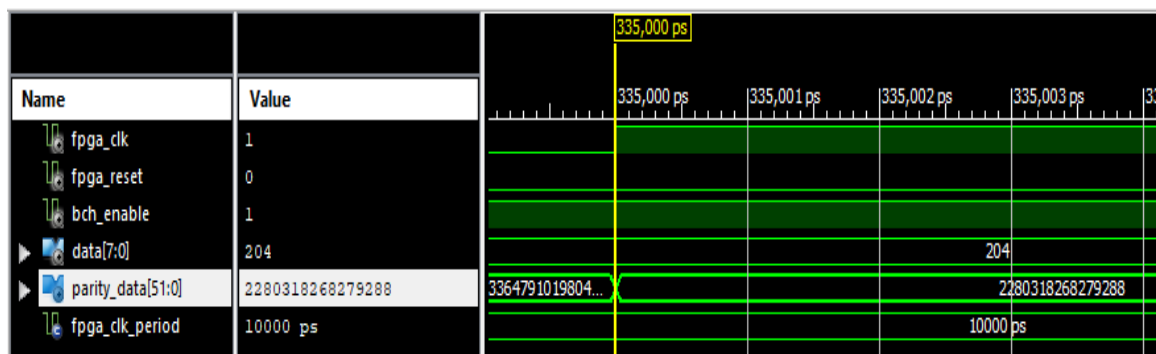
# RESULTS

### 5.1 Simulation results of BCH(8191,8139,4) Encoder

The proposed Encoder is simulated in Xilinx ISE version 14.7 using VHDL. The simulation waveform result for parallel BCH(8191,8139,4) encoder is shown in Figure 5.1.

The clock period for the waveform simulation is 10ns.

The number of clock cycles utilised to generate 52 bit parity for 8139 input bits is 1018 cycles.



**Figure 5.1** Simulated waveform for BCH(8191,8139,4) Encoder (message = 2.9230e+47 (in decimal))

Considering the conventional BCH encoder (Serial BCH encoder), it is observed that clock cycles required for computing the 52 parity bits for 8139 input bits would be (8139+52)cycles. When compared to the proposed parallel BCH encoder, the former takes nearly 8 times longer thus making the proposed design more suitable for high speed operations.

### 5.2 Performance Comparison of Conventional and Parallel BCH(63,39,4) Encoder

To further show the advantage of parallel BCH encoders over the conventional encoders, let us consider BCH(63,39,4) encoder.

Figure 5.2 shows serial BCH(63,39,4) encoder. The clock period is taken as 100ns. Thus, to generate 24-bit parity for 39 input bits, 39 clock cycles will be utilised along with an extra of 24 cycles to output the 24 parity bits from the register in hardware.

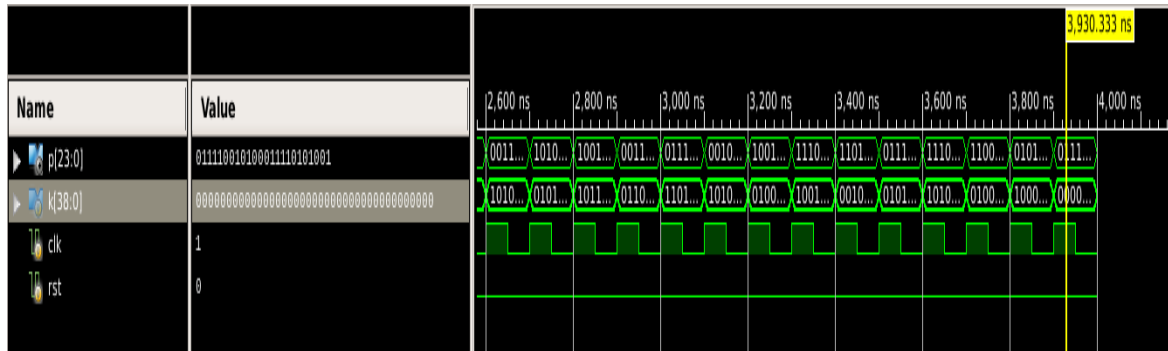


Figure 5.2 Serial BCH(63,39,4) Encoder

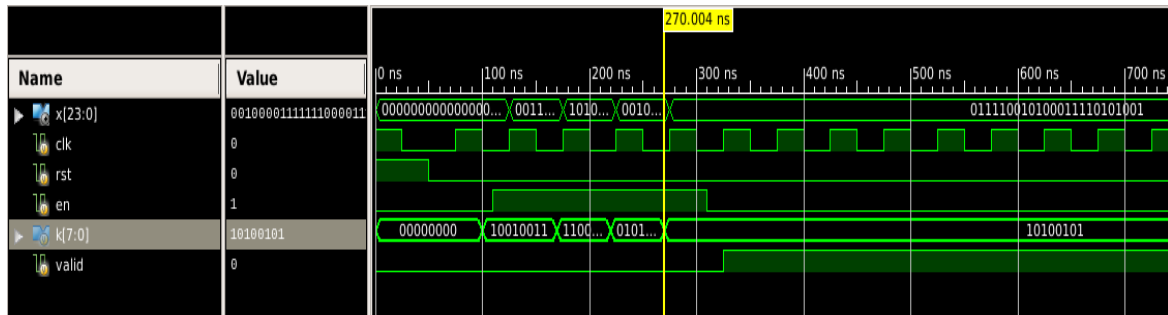


Figure 5.3 Parallel BCH(63,39,4) Encoder

Figure 5.3 shows parallel BCH(63,39,4) encoder. The clock period is taken as 50ns. Thus, to generate 24-bit parity for 39 input bits, 4 clock cycles will be utilised with no extra cycles required to read the parity.

The comparison summary is shown in Table 5.1.

**Table 5.1:** Performance comparison of parallel and serial BCH encoder observed in simulation

	Serial BCH (63,39,4) Encoder	Parallel BCH (63,39,4) Encoder
Clock period	100ns	50ns
Message Bits	39	39
Parity bits	24	24
Clock Cycle utilization-parity computation	39	4
Clock Cycle utilization – to output parity bits	24	0

### 5.3 Simulation Results of BCH Decoder

Once the process of encoding is completed, the received codeword (message + parity bits) are fed to the decoding system designed. The received codeword (rx) is now fed byte wise and the root table is obtained. Later, the codeword is used to calculate the syndrome. As seen in Figure 5.4A, an error-free codeword generates  $S_i = 0$ . When errors are introduced (considering 4 bits of error), it is observed that the syndrome is computed accordingly ( $S_i \neq 0$ ) as in Figure 5.4B.

s1[12:0]	000000000000
s2[12:0]	000000000000
s3[12:0]	000000000000
s4[12:0]	000000000000
s5[12:0]	000000000000
s6[12:0]	000000000000
s7[12:0]	000000000000

**Figure 5.4A :** Syndrome when no errors in rx





s1[12:0]	1110010101111
s2[12:0]	0011110100010
s3[12:0]	0100101010101
s4[12:0]	0101000001010
s5[12:0]	1001010100001
s6[12:0]	0011111010010
s7[12:0]	0111010010110

**Figure 5.4B :** Syndrome when rx has errors

The coefficients,  $\Lambda_v$  are calculated in correspondence with the syndrome generated above. Therefore, when  $S_i = 0$ ,  $\Lambda_v = 0$  as well when no errors in rx as shown in Figure 5.5A and when ( $S_i \neq 0$ ), the  $\Lambda_v$  generated are as shown in Figure 5.5B.






a1[12:0]	000000000000
a2[12:0]	000000000000
a3[12:0]	000000000000
a4[12:0]	000000000000

**Figure 5.5A:** Coefficients when no errors in rx






 a1[12:0]	1110010101111
 a2[12:0]	1100100000111
 a3[12:0]	1110000100101
 a4[12:0]	1010000000001

**Figure 5.5B:** Coefficients when rx has errors

The roots are computed from  $\lambda$  ( $\beta^i$ ) once the coefficients are generated. In Figure 5.6A, roots don't exist as  $S_i$  and  $\Lambda_v$  are zero thus indicating no errors in the received codeword. The generated roots in Figure 5.6B represent the reciprocal of the actual error location, that is if root is 8127, the error location is 64(63<sup>rd</sup> bit).

 er_loc_root[0:3]	UUUUUUUUUUUUU, U...
 [0]	x
 [1]	x
 [2]	x
 [3]	x

**Figure 5.6A:** Roots when no errors in rx

 er_loc_root[0:3]	1111100111101, 1...
 [0]	7997
 [1]	7998
 [2]	8058
 [3]	8127

**Figure 5.6B:** Roots when rx has errors

## Chapter 6

### APPLICATIONS AND ADVANTAGES

The BCH encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage due to a number of reasons (for example, noise or interference, scratches on a CD, etc.). The BCH decoder processes each block of data and attempts to correct the errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the BCH code. BCH encoding and decoding can be carried out either in software or in special-purpose hardware.

#### 6.1 Applications of BCH codes

Bose–Chaudhuri–Hocquenghem (BCH) codes are of great practical importance for error correction, particularly if the expected number of errors is small compared with the length. BCH codes were constructed as a generalization of Hamming codes. BCH codes are best considered as cyclic codes. The original applications of BCH codes were restricted to binary codes of length  $2^m - 1$  for some integer  $m$ . These were extended later by Gorenstein and Zierler to the nonbinary codes with symbols from the Galois field  $GF(q)$ .

##### 6.1.1 Digital Communications and Storage

BCH error correction codes are block-based error correcting codes with a wide range of applications in digital communications and storage. BCH codes are used to correct errors in many systems including:

- Storage devices (including tape, Compact Disk, DVD, barcodes etc.)
- Wireless or mobile communications (including cellular telephones, microwave links, pager etc.)
- Satellite communications
- Digital television / DVB
- High-speed modems such as ADSL, xDSL, etc.

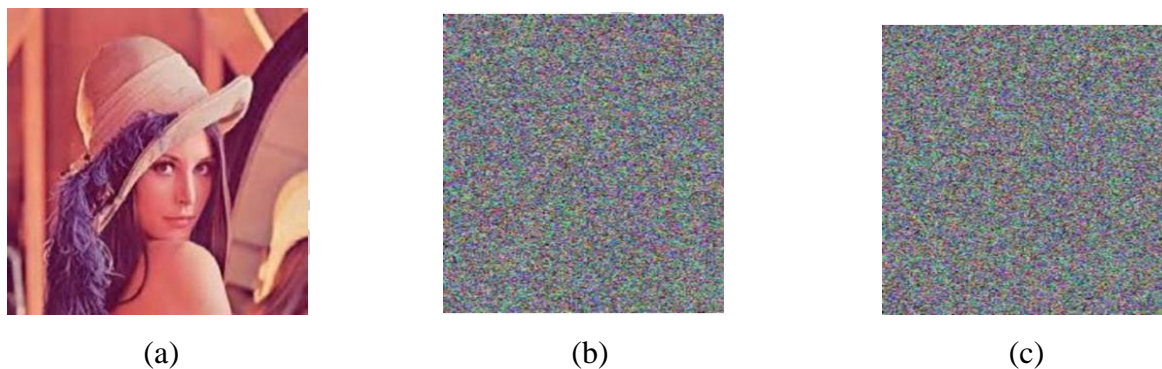
##### 6.1.2 BCH Codes as Industry Standards

- (511, 493) BCH code in ITU-T. Rec. H.261 “video codec for audiovisual service at kb/s” a video coding standard used for video conferencing and video phone.  $n=511$   
 $m=9$   $k=493$   $n-k=18$   $t=2$
- (40, 32) BCH code in ATM (Asynchronous Transfer Mode) is a shortened cyclic code that can correct 1-bit error and detect 2-bit errors

### 6.1.3 BCH Code in image encryption

For the data security perspective we utilize a BCH code in round key addition and mixed column matrix steps in AES algorithm and then put on this modified AES(Advanced Encryption Standard) [18] algorithm to image encryption. The image encryption quality permits to incorporate this alteration to AES.

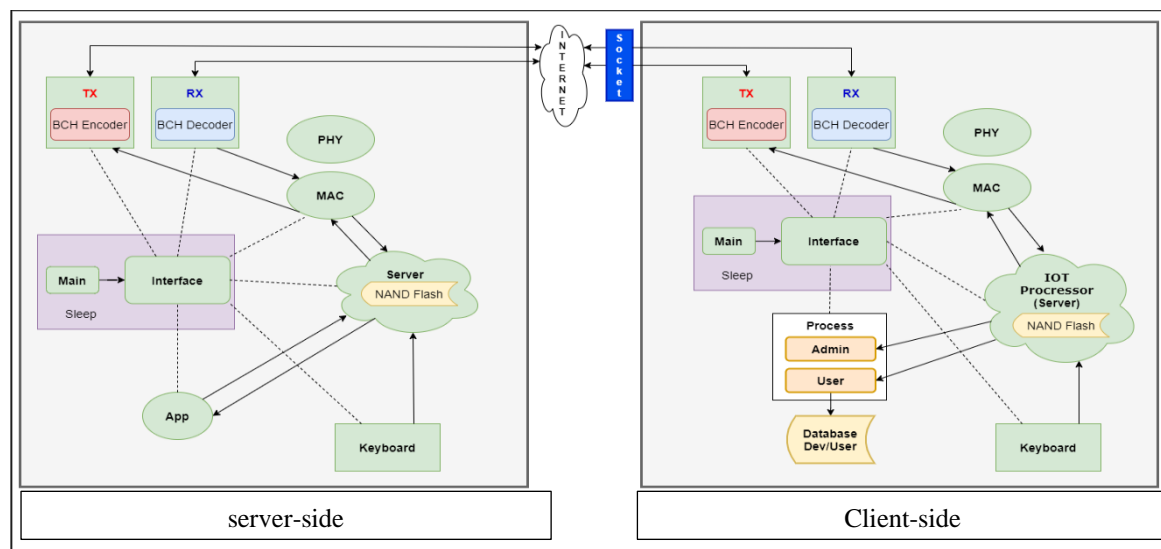
AES algorithm consists of four steps: Substitute byte , Shift Rows, Mix Columns and Add Round Keys .The AES algorithm is modified with the help of BCH codes have the following steps: (1) Convert 128 bits of data into 16 data bytes and write these 16 bytes in a  $4 * 4$  state matrix;(2) The generator polynomial of BCH code is used as a round key. This key is served as the secret key and the current state matrix is XOR with this key in each add round key step;(3) Now the entries of the current state matrix are substituted with the S-box entries; (4) Then perform the circular shift on each row of the current state matrix. Row 0 is shifted 0 byte left, row 1 is shifted 1 byte left, row 2 is shifted 2 bytes left and row 3 shifted 3 bytes left and so on;(5) Now the current state matrix is multiplied with the mix column matrix which is constructed by using the BCH code. Repeat these steps ten times for AES-128 encryption results for data. This scheme is called AES-C. The modified AES-C for the image encryption scheme. Figure 6.1(a) shows that original Lena image, Figure 6.1(b) and 6.1(c) shows encrypted image by original AES and modified AES-C respectively. From Figure 6.1(b) and 6.1(c), it is analyzed that, encrypted image by AES-C is better than encrypted image by AES. The results of AES-C are better in each channel because correlation and energy are close to zero. If Contrast is greater, then it means that image encryption quality is strong and the most important part of the quality of image encryption technique is an entropy AES-C gives entropy close to 8 which give high security for the confidential image.



**Figure 6.1** (a) Original Lena image;(b) Image by AES;(c) Image by AES-C

### 6.1.4 Error-free Communication in NB-IoT

Internet of Things (IoT), a parasitic innovation and one of a kind revolutionizing the cyber networks far and wide over the globe have infiltrated several sectors of human lives. Ensuring and elevating the quality of human living, it continues to evolve with time. The much-evolved version of IoT, Narrow Band Internet of things (NB-IoT) has left the world astonished owing to its unique ability to digitally transform lives with much lesser frequency of operation. The ability to coexist with technologies such as Artificial Intelligence (AI) and Machine Learning (ML) having a contained asset wastage, provides NB-IoT a high ground over existing innovations. With the introduction of error-correcting codes such as BCH in the client and server-side of the network, NB-IoT prospers over Industrial sectors ensuring nearly absolute error-free data transfer. BCH codes incorporated alongside NB-IoT structures a favored arrangement in ensuring the quality of data and assuring the well-defined future of several industrial sectors.



**Figure 6.2** NB-IoT architecture with BCH arrangement

NB-IoT is surely evinced in terms of a cost-effective, reliable, and low power solution. But still, persist with attenuation of the received signal due to external factors. For a critical system data, integrity is must, therefore, the BCH is inculcated in NBIoT at the transceiver with error correction capability close to achieving absolute error-free data and successful data transfer.



## 6.2 Advantages

A class of powerful multiple-error-correcting cyclic codes was discovered by Bose and Ray-Chaudhuri in 1960 and independently by Hocquenghem in 1959. These codes are known as the BCH codes.

The BCH codes provide a wide variety of block lengths and corresponding code rates. They are important not only because of their flexibility in the choice of their code parameters, but also because, at block lengths of a few hundred or less, many of these codes are among the most used codes of the same lengths and code rates.

Another advantage is that there exist very elegant and powerful algebraic decoding algorithms for the BCH codes. The importance of the BCH codes also stems from the fact that they are capable of correcting all random patterns of  $t$  errors by a decoding algorithm that is both simple and easily realized in a reasonable amount of equipment. BCH codes occupy a prominent place in the theory and practice of multiple-error correction.

## **Chapter 7**

### **CONCLUSIONS AND SCOPE FOR FUTURE WORK**

The proposed Systolic Array type BCH encoder can be molded to whichever parallelization factor without any complications. BCH codes are illustrated to be eminent error correcting codes for codes of any length and irrespective of how random the errors are. The exact capabilities of these codes have driven large attention as the encoding-decoding system is simple. Thus, adopting methods involving parallel approach, the speed of operation and device utilization are improved to a great extent as illustrated with the comparisons made considering BCH(63,39,4) encoder.

The error correction is successfully done by implementing a BCH decoder. It is seen that when errors occur in the received codeword, the corresponding syndrome, coefficients and roots are computed to locate position of the errors precisely which can be corrected by simply flipping only those bits. In this paper, we have considered only 4-bit error correction. One prime highlight of this method is that the error correction is done in the parity as well in addition to the message as there are chances of the parity also being affected while transmission. This provides a more reliable retrieval of data at receiver end.

The same concept can be extended to multiple error correction by generating the syndrome and based on  $i(2t-1)$  value in  $S_i$ , the  $\Lambda_v$  equations can be obtained from PGZ algorithm and the corresponding Chien search can be applied for determining the error locations and correcting them.

The further study will be on the concept of error detection when there are more than  $t$ -bits of error, that is, here, when we encounter more than 4-bits of error, the roots generated may not yield the exact error locations. A system having adaptability to correct any number of error bits is to be designed.

## REFERENCES

- [1] Rajesh Mehra & Garima Saini , Sukhbir Singh- “FPGA Based High Speed BCH Encoder for Wireless Communication Applications” 2011 International Conference on Communication Systems and Network Technologies
- [2] Po-Ning Chen, Hsuan-Yin Lin, Stefan M. Moser – “Nonlinear codes outperform the best linear codes on the binary erasure channel” 2015 IEEE International Symposium on Information Theory (ISIT)
- [3] FJ MacWilliams, NJA Sloane – “The theory of error-correcting codes”- 1977 - books.google.com
- [4] Faisal Rasheed Lone, Arjun Puri, Sudesh Kumar , “Performance Comparison of Reed Solomon Code and BCH Code over Rayleigh Fading Channel” International Journal of Computer Applications 71(20):23-26, June 2013
- [5] 1961Je-Hoon Lee and Sharad Shakya- “Implementation of Parallel BCH Encoder Employing Tree-Type Systolic Array Architecture” International Journal of Sensor and Its Applications for Control Systems Vol.1, No.1 (2013), pp.1-12
- [6] Nabil Abu-Khader, Pepe Siy -“Inversion/Division in Galois Field Using Multiple-Valued Logic” 37th International Symposium on Multiple-Valued Logic (ISMVL'07)
- [7] D. Gorenstein and N. Zierler, “A Class of Cyclic Linear Error-Correcting Codes in pm Symbols” Journal of the Society of Industrial and Applied Mathematics, vol. 9, pp. 207-214, June
- [8] Yanni Chen and Keshab K. Parhi – “Area Efficient Parallel Decoder Architecture for long Bch Codes” 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing
- [9] Data Integrity Checks <https://aeronavdata.com/capabilities/airport-mapping-data/integrity-checks/>
- [10] Technical note “NAND Error Correction Codes Introduction” by Macronix International Co., Ltd.- AN0271 (Rev. 1), 17-02-2014
- [11] E. R. Berlekamp,- ”On Decoding Binary Bose-Chaudhuri-Hocquenghem codes”, IEEE Transactions on Information Theory, IT-11: 577-80, October 1965.
- [12] Henry D. Pfister “Algebraic Decoding of Reed-Solomon and BCH Codes” November 15th, 2013 (rev. 2)
- [13] Yi-Min Lin, Chi-Heng Yang, Chih-Hsiang Hsu, Hsie-Chia Chang, and Chen-Yi Lee – “A MPCN-Based Parallel Architecture in BCH Decoders for NAND Flash Memory Devices” IEEE Transactions on Circuits And Systems—ii: Express Briefs, Vol. 58, No. 10, October 2011.
- [14] 7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide
- [15] 002-00499\_S34ML01G2\_S34ML02G2\_S34ML04G2\_1\_Gb\_2\_Gb\_4\_Gb\_3\_V\_4-bit\_ECC\_SLC\_NAND\_Flash\_Memory\_for\_Embedded.pdf
- [16] Micron TN-29-71: Enabling Software BCH ECC on a Linux Platform Introduction
- [17] Amit Kumar Panda, Shahbaz sarik, Abhishek Awasthi- “FPGA Implementation of Encoder for (15, k) Binary BCH Code Using VHDL and Performance Comparison for Multiple Error Correction Control” 2012 International Conference on Communication Systems and Network Technologies
- [18] Muhammad Asif and Tariq Shah “BCH Codes with computational approach and its applications in image encryption”, Journal of Intelligent & Fuzzy Systems

## APPENDIX A

### **BCH Encoder: Module definition**

-----

-- Company:  
-- Engineer:  
--  
-- Create Date: 17:34:03 03/16/2020  
-- Design Name:  
-- Module Name: bch\_encoder - Behavioral  
-- Project Name:  
-- Target Devices:  
-- Tool versions:  
-- Description:  
--  
-- Dependencies:  
--  
-- Revision:  
-- Revision 0.01 - File Created  
-- Additional Comments:  
--

-----

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

-- Uncomment the following library declaration if using  
-- arithmetic functions with Signed or Unsigned values  
--use IEEE.NUMERIC\_STD.ALL;

-- Uncomment the following library declaration if instantiating  
-- any Xilinx primitives in this code.

--library UNISIM;

--use UNISIM.VComponents.all;

entity bch\_encoder is

```
Port ( fpga_clk : in STD_LOGIC;  
      fpga_reset : in STD_LOGIC;  
      bch_enable : in STD_LOGIC;  
      data : in STD_LOGIC_VECTOR (7 downto 0);  
      parity_data_valid : out STD_LOGIC;  
      parity_data : out STD_LOGIC_VECTOR (51 downto 0));
```

end bch\_encoder;

architecture Behavioral of bch\_encoder is

signal p: std\_logic\_vector(51 downto 0);

begin

process(fpga\_clk)

begin

if(rising\_edge( )) then

if(fpga\_reset='1') then

p <= x"00000000000000";

elsif(bch\_enable='1') then

p(51) <= p(43) xor data(1) xor data(3) xor p(47) xor p(45);

p(50) <= p(42) xor data(0) xor p(44) xor data(2) xor p(46);

p(49) <= p(41) xor data(3) xor p(47);

p(48) <= p(40) xor data(7) xor p(51) xor data(2) xor p(46);

p(47) <= p(39) xor data(6) xor p(50) xor data(7) xor p(51) xor data(1) xor p(45);

p(46) <= p(38) xor data(0) xor p(44) xor data(5) xor p(49) xor data(7) xor p(51) xor  
data(6) xor p(50);

p(45) <= p(37) xor data(1) xor p(45) xor data(3) xor p(47) xor data(5) xor p(49) xor  
data(4) xor p(48) xor data(6) xor p(50);

p(44) <= p(36) xor data(0) xor p(44) xor data(2) xor p(46) xor data(4) xor p(48) xor  
data(3) xor p(47) xor data(5) xor p(49);

p(43) <= p(35) xor data(2) xor p(46) xor data(4) xor p(48) xor data(7) xor p(51);

p(42) <= p(34) xor data(1) xor p(45) xor data(3) xor p(47) xor data(7) xor p(51) xor  
data(6) xor p(50);

p(41) <= p(33) xor data(0) xor p(44) xor data(2) xor p(46) xor data(5) xor p(49) xor  
data(7) xor p(51) xor data(6) xor p(50);

$p(40) \leq p(32) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50);$

$p(39) \leq p(31) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(38) \leq p(30) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49);$

$p(37) \leq p(29) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(36) \leq p(28) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50);$

$p(35) \leq p(27) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(34) \leq p(26) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50);$

$p(33) \leq p(25) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(32) \leq p(24) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(31) \leq p(23) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(30) \leq p(22) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(5) \text{ xor } p(49);$

$p(29) \leq p(21) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(28) \leq p(20) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(27) \leq p(19) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51) \text{ xor data}(6) \text{ xor } p(50);$

$p(26) \leq p(18) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51) \text{ xor data}(5) \text{ xor } p(49);$

$p(25) \leq p(17) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50);$

$p(24) \leq p(16) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(7) \text{ xor } p(51);$

$p(23) \leq p(15) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(22) \leq p(14) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(21) \leq p(13) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(20) \leq p(12) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(1) \text{ xor } p(45);$

$p(19) \leq p(11) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(18) \leq p(10) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(17) \leq p(9) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(16) \leq p(8) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(15) \leq p(7) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(14) \leq p(6) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50);$

$p(13) \leq p(5) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(12) \leq p(4) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(11) \leq p(3) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(3) \text{ xor } p(47) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50);$

$p(10) \leq p(2) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(2) \text{ xor } p(46) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(9) \leq p(1) \text{ xor data}(0) \text{ xor } p(44) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50);$

$p(8) \leq p(0) \text{ xor data}(1) \text{ xor } p(45) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(7) \leq \text{data}(0) \text{ xor } p(44) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

$p(6) \leq \text{data}(1) \text{ xor } p(45) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(6) \text{ xor } p(50);$

$p(5) \leq \text{data}(0) \text{ xor } p(44) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(5) \text{ xor } p(49) \text{ xor data}(7) \text{ xor } p(51);$

$p(4) \leq \text{data}(1) \text{ xor } p(45) \text{ xor data}(4) \text{ xor } p(48) \text{ xor data}(6) \text{ xor } p(50) \text{ xor data}(7) \text{ xor } p(51);$

```
p(3) <= data(0) xor p(44) xor data(3) xor p(47) xor data(5) xor p(49) xor data(6)
xor p(50) xor data(7) xor p(51);
p(2) <= data(1) xor p(45) xor data(2) xor p(46) xor data(3) xor p(47) xor data(4)
xor p(48) xor data(5) xor p(49) xor data(6) xor p(50);
p(1) <= data(0) xor p(44) xor data(1) xor p(45) xor data(2) xor p(46) xor data(3)
xor p(47) xor data(4) xor p(48) xor data(5) xor p(49);
p(0) <= data(0) xor p(44) xor data(2) xor p(46) xor data(4) xor p(48);
end if;
end if;
end process;
parity_data<= p;
end Behavioral;
```

## **BCH Encoder: Test Bench**

---

```
-- Company:
-- Engineer:
--
-- Create Date: 15:18:37 04/23/2020
-- Design Name:
-- Module Name: /home/ise/mp_bch/bchenc/bch_encoder_tb.vhd
-- Project Name: bchenc
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: bch_encoder
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
```



-- Notes:

-- This testbench has been automatically generated using types std\_logic and  
-- std\_logic\_vector for the ports of the unit under test. Xilinx recommends  
-- that these types always be used for the top-level I/O of a design in order  
-- to guarantee that the testbench will bind correctly to the post-implementation  
-- simulation model.

-----  
LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

-- Uncomment the following library declaration if using  
-- arithmetic functions with Signed or Unsigned values  
--USE ieee.numeric\_std.ALL;

ENTITY bch\_encoder\_tb IS

END bch\_encoder\_tb;

ARCHITECTURE behavior OF bch\_encoder\_tb IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT bch\_encoder

PORT(

    fpga\_clk : IN std\_logic;

    fpga\_reset : IN std\_logic;

    bch\_enable : IN std\_logic;

    data : IN std\_logic\_vector(7 downto 0);

    -- parity\_data\_valid : OUT std\_logic;

    parity\_data : OUT std\_logic\_vector(51 downto 0)

);

END COMPONENT;

```
--Inputs
signal fpga_clk : std_logic := '0';
signal fpga_reset : std_logic := '0';
signal bch_enable : std_logic := '0';
signal data : std_logic_vector(7 downto 0) := (others => '0');
```

```
--Outputs
--signal parity_data_valid : std_logic;
signal parity_data : std_logic_vector(51 downto 0);
```

```
-- Clock period definitions
constant fpga_clk_period : time := 10 ns;
```

BEGIN

```
-- Instantiate the Unit Under Test (UUT)
 uut: bch_encoder PORT MAP (
   fpga_clk => fpga_clk,
   fpga_reset => fpga_reset,
   bch_enable => bch_enable,
   data => data,
   -- parity_data_valid => parity_data_valid,
   parity_data => parity_data
 );
```

```
-- Clock process definitions
fpga_clk_process :process
begin
   fpga_clk <= '0';
   wait for fpga_clk_period/2;
   fpga_clk <= '1';
   wait for fpga_clk_period/2;
end process;
```

```
-- Stimulus process
stim_proc: process
begin
  -- hold reset state for 100 ns.
  wait for 100 ns;
  -- insert stimulus here
    data <= "11111111";fpga_reset <= '1';bch_enable <= '1';wait for fpga_clk_period;

    data <= "11111111";fpga_reset <= '1';bch_enable <= '0';wait for fpga_clk_period;

    data <= "11111111";fpga_reset <= '0';bch_enable <= '0';wait for fpga_clk_period;

    --data <= "00011111";fpga_reset <= '0';bch_enable <= '1';wait for
fpga_clk_period;

    fpga_reset <= '1'; bch_enable <= '1';data <= "10100101";wait for
fpga_clk_period;
    for i in 1 to 10 loop
      fpga_reset <= '0'; bch_enable <= '1';data <= "10101010";wait for fpga_clk_period;
    end loop;
    for i in 1 to 10 loop
      fpga_reset <= '0'; bch_enable <= '1';data <= "11001100";wait for fpga_clk_period;

    end loop;

  --
  for i in 1 to 127 loop
  --
    data <= "11111111";wait for fpga_clk_period;
  --
  END LOOP;
  --
  for i in 1 to 127 loop
  --
    data <= "00000000";wait for fpga_clk_period;
  --
  END LOOP;
```

```
--      for i in 1 to 127 loop
--          data <= "11111111";wait for fpga_clk_period;
--      END LOOP;
--      for i in 1 to 127 loop
--          data <= "00000000";wait for fpga_clk_period;
--      END LOOP;
--      for i in 1 to 127 loop
--          data <= "11111111";wait for fpga_clk_period;
--      END LOOP;
--      for i in 1 to 127 loop
--          data <= "00000000";wait for fpga_clk_period;
--      END LOOP;
--      for i in 1 to 127 loop
--          data <= "11111111";wait for fpga_clk_period;
--      END LOOP;
--      for i in 1 to 127 loop
--          data <= "00000000";wait for fpga_clk_period;
--      END LOOP;
--      data <= "-----";wait for fpga_clk_period;
--      bch_enable <= '0';
--      $$stop;
--
--      wait;
--      end process;

END;
```

## APPENDIX B

### BCH Decoder: Module Definition

-----

-- Company:  
-- Engineer:  
--  
-- Create Date: 23:02:12 04/29/2020  
-- Design Name:  
-- Module Name: bch - Behavioral  
-- Project Name:  
-- Target Devices:  
-- Tool versions:  
-- Description:  
--  
-- Dependencies:  
--  
-- Revision:  
-- Revision 0.01 - File Created  
-- Additional Comments:  
--

-----

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.Numeric_Std.all;  
--use IEEE.STD_LOGIC_arith.all;  
use ieee.std_logic_signed.all;  
use std.textio.all;
```

```
entity bch_decoder is  
Port ( fpga_clk : in STD_LOGIC;  
BCH_RESET : in STD_LOGIC;  
bch_decoder_en : in STD_LOGIC;
```

```
bch_decoder_data :in std_logic_vector (7 downto 0);
error_byte_location_OP_1 : inout STD_LOGIC_VECTOR (9 downto 0);
error_bit_location_OP_1 : out STD_LOGIC_VECTOR (7 downto 0);
error_byte_location_OP_2 : inout STD_LOGIC_VECTOR (9 downto 0);
error_bit_location_OP_2 : out STD_LOGIC_VECTOR (7 downto 0);
error_byte_location_OP_3 : inout STD_LOGIC_VECTOR (9 downto 0);
error_bit_location_OP_3 : out STD_LOGIC_VECTOR (7 downto 0);
error_byte_location_OP_4 : inout STD_LOGIC_VECTOR (9 downto 0);
error_bit_location_OP_4 : out STD_LOGIC_VECTOR (7 downto 0);
-- mem_OP : out STD_LOGIC_VECTOR (5 downto 0);
num_of_error : out STD_LOGIC_VECTOR (2 downto 0)
--error_location_valid : out STD_LOGIC
);
end bch_decoder;
```

architecture Behavioral of bch\_decoder is

-----RX\_buffer-----

```
type buff is array (0 to 1023) of std_logic_vector(7 downto 0);
```

```
signal rx : buff;
```

```
signal rx_done : std_logic := '0';
```

-----roots mem declaration-----

```
type memory is array (0 to 8190) of std_logic_vector(12 downto 0);
```

```
signal mem : memory;
```

```
signal root_done : std_logic := '0';
```

-----syndrome declaration-----

```
signal s1 : STD_LOGIC_VECTOR (12 downto 0);
```

```
signal s2 : std_logic_vector (12 downto 0);
```

```
signal s3 : std_logic_vector (12 downto 0);
```

```
signal s4 : std_logic_vector (12 downto 0);
```

```
signal s5 : std_logic_vector (12 downto 0);
```

```
signal s6 : std_logic_vector (12 downto 0);
```

```
signal s7 : std_logic_vector (12 downto 0);
```

```
signal i1 : integer range 0 to 8191 := 0;
```

```
-----eigen value declaration-----
signal A1 : std_logic_vector (12 downto 0);
signal A2 : std_logic_vector (12 downto 0);
signal A3 : std_logic_vector (12 downto 0);
signal A4 : std_logic_vector (12 downto 0);
signal eigen_en :std_logic := '0';
signal eigen_done :std_logic := '0';
-----chien search declaration-----
signal bch_decoder_chien_en : std_logic := '0';
signal ch_ini : std_logic := '0';
signal chien_done : std_logic := '0';
signal jc: integer range 0 to 4 := 0;
type error_loc_roots is array (0 to 3) of std_logic_vector(12 downto 0);
signal er_loc_root : error_loc_roots;
-----GF multiplication function-----
function mul (v1, v2 : in std_logic_vector) return std_logic_vector is
constant m      : integer := 13;
variable dummy   : std_logic;
variable v_temp  : std_logic_vector(m-1 downto 0);
variable ret     : std_logic_vector(m-1 downto 0);

begin
v_temp := (others=>'0');  --- p(x)=1+ x+x3+ x4 + x13
for i in 0 to m-1 loop
dummy   := v_temp(12);
v_temp(12 ) := v_temp(11 );
v_temp(11 ) := v_temp(10 );
v_temp(10 ) := v_temp(9 );
v_temp(9 ) := v_temp(8 );
v_temp(8 ) := v_temp(7 );
v_temp(7 ) := v_temp(6 );
v_temp(6 ) := v_temp(5 );
v_temp(5 ) := v_temp(4 );
v_temp(4 ) := v_temp(3 ) xor dummy;
```

v\_temp(3) := v\_temp(2) xor dummy;

v\_temp(2) := v\_temp(1);

v\_temp(1) := v\_temp(0) xor dummy;

v\_temp(0) := dummy;

for j in 0 to m-1 loop

v\_temp(j) := v\_temp(j) xor (v1(j) and v2(m-i-1));

end loop;

end loop;

ret := v\_temp;

return ret;

end mul;

-----GF division function-----

function div (a, b : in std\_logic\_vector) return std\_logic\_vector is

constant m : integer := 13;

variable ret : std\_logic\_vector(m-1 downto 0);

variable temp : std\_logic\_vector(m-1 downto 0);

begin

temp := b; ---  $b^{(-1)} = b^{((2m)-2)}$  (since,  $b^{((2m)-1)} = 1$ )

for i in 0 to 10 loop

temp := mul (temp,temp);

temp := mul (b,temp);

end loop;

temp := mul (temp,temp);

ret := mul (a,temp);

return ret;

end div;

-----

-----

begin

-----RX\_buffer-----

---



```
process(fpga_clk)
variable a : integer range 0 to 1023:= 1023;
variable f : integer range 0 to 1023 := 1023;
begin
if(fifo_done = '0') then ----receive padded zeros first then (msb)data and parity(lsb) bits
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(BCH_RESET = '0' ) then
if(f >=0 and f < 1024) then
rx(f) <= bch_decoder_data; a:= f; f:=f-1;
end if;
end if;
end if;
end if;
if(a = 0) then rx_done <= '1';
end if;
end if;
end process;
```

-----mem -----

```
process(fpga_clk)
variable temp_mem : memory;-----calculating roots for extension field (0 to 8190
alphas)
variable j : integer := 0;
variable root: std_logic_vector(12 downto 0);
variable var1: std_logic;
variable var3: std_logic;
variable var4: std_logic;
```

```
-----
begin
if(root_done = '0') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
```

```
if(BCH_RESET = '1') then
root := "00000000000000";
root := "00000000000001";
temp_mem(0) := root;
else
if(j < 8190) then
if(root(12) /= '1') then
root := root(11 downto 0) & root(12);
else
var1 := root(0);
var3 := root(2);
var4 := root(3);
root := (root(11 downto 0)&root(12));
root(1) := '1' xor var1;
root(3) := '1' xor var3;
root(4) := '1' xor var4;
end if ;
j := j+1;
temp_mem(j) := root;
end if;
end if;
end if;
end if;
if(j = 8190) then mem <= temp_mem; root_done <= '1';end if;
end if;
end process;
-----Syndrome calculation-----
process(fpga_clk) ---s1----
variable mul : std_logic_vector(12 downto 0);
variable n : integer range 0 to 1023 := 0;
variable j: integer range 0 to 7 := 0;
variable si : integer range 0 to 8191:= 0;
variable sum: Std_logic_vector(12 downto 0);
variable s11: Std_logic_vector(12 downto 0) := (others => '0');
```

---

---

```
variable i_done1: Std_logic := '0';
begin
if(i_done1 /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
0
if(si < 8190) then
for n in 0 to 1023 loop
for j in 0 to 7 loop

if( si /= 8191) then
mul := mem(si) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum := mul xor S11;
S11 := sum;
S1 <= (mem(i1) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) )) xor S1;
si := si+1;
end if;
end loop;
end loop;
end if;
end if;
end if;
end if;
if(si > 8190) then i_done1 := '1' ; i1 <= si; s1 <= s11; end if;
end if;
end process;

process(fpga_clk) ---s2----
variable mul2 : std_logic_vector(12 downto 0);
variable si2 : integer range 0 to 8200:= 0;
variable sum2: Std_logic_vector(12 downto 0);
variable s22: Std_logic_vector(12 downto 0) := (others => '0');
```

---

---

```
variable i_done2: Std_logic := '0';
begin
if(i_done2 /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(root_done = '1') then
s_2:for n in 0 to 1023 loop
for j in 0 to 7 loop
if(si2 < 8191) then
mul2 := mem(si2) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum2 := mul2 xor S22;
S22 := sum2;
elsif(si2 > 8190) then si2 := si2 - 8191;
end if;
exit s_2 when si2 = 8189;
si2 := si2+2;
end loop;
end loop;
end if;
end if;
end if;
if(si2 = 8189) then s2 <= s22; i_done2 := '1';end if;
end if;
end process;
--
process(fpga_clk) ---s3----
variable mul3 : std_logic_vector(12 downto 0);
variable si3 : integer range 0 to 8200:= 0;
variable sum3: Std_logic_vector(12 downto 0);
variable s33: Std_logic_vector(12 downto 0) := (others => '0');
variable i_done3: Std_logic := '0';
begin
if(i_done3 /= '1') then
```

```
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(root_done = '1') then
s_3:for n in 0 to 1023 loop
for j in 0 to 7 loop
if(si3 < 8191) then
mul3 := mem(si3) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum3 := mul3 xor S33;
S33 := sum3;
elsif(si3 > 8190) then si3 := si3 - 8191;
end if;

exit s_3 when si3 = 8188;
si3 := si3+3;
end loop;
end loop;
end if;
end if;
end if;
if(si3 = 8188) then s3 <= s33; i_done3 := '1';      end if;
end if;
end process;
```

```
process(fpga_clk) ---s4---
variable mul4: std_logic_vector(12 downto 0);
variable si4 : integer range 0 to 8200:= 0;
variable sum4: Std_logic_vector(12 downto 0);
variable s44: Std_logic_vector(12 downto 0) := (others => '0');
variable i_done4: Std_logic := '0';
begin
if(i_done4 /= '1') then
if(rising_edge(fpga_clk)) then
```

---

---

```
if(bch_decoder_en = '1') then
if(root_done = '1') then
s_4:for n in 0 to 1023 loop
for j in 0 to 7 loop
if(si4 < 8191) then
mul4 := mem(si4) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum4 := mul4 xor S44;
S44 := sum4;
elsif(si4 > 8190) then si4 := si4 - 8191;
end if;

exit s_4 when si4 = 8187;
si4 := si4+4;
end loop;
end loop;
end if;
end if;
end if;
if(si4 = 8187) then s4 <= s44; i_done4 := '1';end if;
end if;
end process;
----
process(fpga_clk) ---s5----
variable mul5 : std_logic_vector(12 downto 0);
variable si5 : integer range 0 to 8200:= 0;
variable sum5: Std_logic_vector(12 downto 0);
variable s55: Std_logic_vector(12 downto 0) := (others => '0');
variable i_done5: Std_logic := '0';
begin
if(i_done5 /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(root_done = '1') then
```

```
s_5:for n in 0 to 1023 loop
for j in 0 to 7 loop
if(si5 < 8191) then
mul5:= mem(si5) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum5 := mul5 xor S55;
S55 := sum5;
elsif(si5 > 8190) then si5 := si5 - 8191;
end if;

exit s_5 when si5 = 8186;
si5 := si5+5;
end loop;
end loop;
end if;
end if;
end if;
if(si5 = 8186) then s5 <= s55; i_done5 := '1';      end if;
end if;
end process;
```

```
process(fpga_clk) ---s6----
variable mul6 : std_logic_vector(12 downto 0);
variable si6 : integer range 0 to 8200:= 0;
variable sum6: Std_logic_vector(12 downto 0);
variable s66: Std_logic_vector(12 downto 0) := (others => '0');
variable i_done6: Std_logic := '0';
begin
if(i_done6 /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(root_done = '1') then
s_6:for n in 0 to 1023 loop
for j in 0 to 7 loop
```

```
if(si6 < 8191) then
mul6:= mem(si6) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum6 := mul6 xor S66;
S66 := sum6;
elsif(si6 > 8190) then si6 := si6 - 8191;
end if;
exit s_6 when si6 = 8185;
si6 := si6+6;
end loop;
end loop;
end if;
end if;
end if;
if(si6 = 8185) then s6 <= s66; i_done6 := '1';end if;
end if;
end process;
```

```
process(fpga_clk) ---s7----
variable mul7 : std_logic_vector(12 downto 0);
variable si7 : integer range 0 to 8200:= 0;
variable sum7: Std_logic_vector(12 downto 0);
variable s77: Std_logic_vector(12 downto 0) := (others => '0');
variable i_done7: Std_logic := '0';
begin
if(i_done7 /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_en = '1') then
if(root_done = '1') then
s_7:for n in 0 to 1023 loop
for j in 0 to 7 loop
if(si7 < 8191) then
```



```
mul7:= mem(si7) and (rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) &
rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) & rx(n)(j) );
sum7 := mul7 xor S77;
S77 := sum7;
elsif(si7 > 8190) then si7 := si7 - 8191;
end if;
exit s_7 when si7 = 8184;
si7 := si7+7;
end loop;
end loop;
end if;
end if;
end if;
if(si7 = 8184) then s7 <= s77; i_done7 := '1';end if;
end if;
if((s1 or s2 or s3 or s4 or s5 or s6 or s7) /= "-----") then eigen_en <= '1';end if;
end process;
-----eigen calculations-----
```

```
process(fpga_clk)
variable k1: std_logic_vector(12 downto 0);
variable k2: std_logic_vector(12 downto 0);
variable k3: std_logic_vector(12 downto 0);
variable k4: std_logic_vector(12 downto 0);
variable k5: std_logic_vector(12 downto 0);
variable k6: std_logic_vector(12 downto 0);
variable k7: std_logic_vector(12 downto 0);
variable k8: std_logic_vector(12 downto 0);
variable k9: std_logic_vector(12 downto 0);
variable k10: std_logic_vector(12 downto 0);
variable k11: std_logic_vector(12 downto 0);
variable k12: std_logic_vector(12 downto 0);
variable k13: std_logic_vector(12 downto 0);
variable k14: std_logic_vector(12 downto 0);
```

```
variable k15: std_logic_vector(12 downto 0);
variable k16: std_logic_vector(12 downto 0);
variable k17: std_logic_vector(12 downto 0);
variable k18: std_logic_vector(12 downto 0);
```

```
variable nr1: std_logic_vector(12 downto 0);
variable nr2: std_logic_vector(12 downto 0);
variable dr1: std_logic_vector(12 downto 0);
```

```
variable A11: std_logic_vector(12 downto 0);
variable A22: std_logic_vector(12 downto 0);
variable A33: std_logic_vector(12 downto 0);
variable A44: std_logic_vector(12 downto 0);
```

```
begin
```

```
if(eigen_done /= '1') then
```

```
if( eigen_en = '1') then
```

```
if(rising_edge(fpga_clk)) then
```

```
if(bch_decoder_en = '1') then
```

```
k1 := mul (S1,S7); --op s1s7
```

```
k2 := mul (S1,S1); --op s1^2
```

```
k3 := mul (k2,k2); --op s1^4
```

```
k4 := mul (k3,k3); --op s1^8
```

```
k5 := mul (S1,k3); --op s1^5
```

```
k6 := mul (S3,k5); --op s3s1^5
```

```
k7 := mul (S3,S5); --op s3s5
```

```
k8 := mul (S1,k2); --op s1^3
```

```
k9 := mul (S3,k8); --op s3s1^3
```

```
k10 := mul (S3,S3); --op s3^2
```

```
k11 := mul (k5,S1); --op s1^6
```

```
k12 := mul (S1,S5); --op s1s5
```

```
nr1 := (k1 xor k4 xor k6 xor k7);
```

dr1 := (k9 xor k10 xor k11 xor k12);

k13 := div (nr1,dr1);

k14 := mul (S1,A22);

k15 := mul (S3,k2);

k16 := mul (A22,k8);

k17 := mul (A22,S3);

nr2 := (k15 xor S5 xor k16 xor k17);

k18 := div (nr2,S1);

A11 := s1;

A22 := k13;

A33 := (k8 xor S3 xor k14);

A44 := k18;

end if;

end if;

end if;

A1 <= A11;

A2 <= A22;

A3 <= A33;

A4 <= A44;

if(( A1 or A2 or A3 or A4 ) /= ( "-----" )) then bch\_decoder\_chien\_en <= '1' ;

eigen\_done <= '1'; end if;

end if;

end process;

-----chien search-----

process(fpga\_clk)

variable ch1 : std\_logic\_vector (12 downto 0);

variable ch2 : std\_logic\_vector (12 downto 0);

variable ch3 : std\_logic\_vector (12 downto 0);

variable ch4 : std\_logic\_vector (12 downto 0);

```
variable Ach1: std_logic_vector(12 downto 0) := (others => '0');
variable Ach2: std_logic_vector(12 downto 0) := (others => '0');
variable Ach3: std_logic_vector(12 downto 0) := (others => '0');
variable Ach4: std_logic_vector(12 downto 0) := (others => '0');
variable g : std_logic_vector(12 downto 0) := (others => '0');
variable r : std_logic_vector(12 downto 0) := (others => '1');
begin
if(chien_done /= '1') then
if(rising_edge(fpga_clk)) then
if(bch_decoder_chien_en = '1' and ch_ini = '0') then
Ach1 := A1;
Ach2 := A2;
Ach3 := A3;
Ach4 := A4;
r := "00000000000000";
ch_ini <= '1';

elsif(ch_ini = '1') then
count <= count+1;
g := "00000000000001" xor Ach1 xor Ach2 xor Ach3 xor Ach4;
if(("00000000000001" xor Ach1 xor Ach2 xor Ach3 xor Ach4) = "00000000000000") then
er_loc_root(jc) <= r+"1";
jc <= jc + 1;
end if;
r := r+1;

ch1 := mul (Ach1,mem(1));
ch2 := mul (Ach2,mem(2));
ch3 := mul (Ach3,mem(3));
ch4 := mul (Ach4,mem(4));

Ach1 := ch1;
Ach2 := ch2;
Ach3 := ch3;
```

Ach4 := ch4;

if(r = "111111111111") then chien\_done <= '1';end if;

end if;

end if;

end if;

end process;

-----error location op -----

process(fpga\_clk)

variable er1 : std\_logic\_vector(12 downto 0) := (others => '0');

variable er2 : std\_logic\_vector(12 downto 0) := (others => '0');

variable er3 : std\_logic\_vector(12 downto 0) := (others => '0');

variable er4 : std\_logic\_vector(12 downto 0) := (others => '0');

variable temp\_rx : std\_logic\_vector(7 downto 0) := (others => '0');

variable empty: std\_logic := '0';

begin

if(rising\_edge(fpga\_clk)) then

if(bch\_decoder\_en = '1') then

case jc is

when 4 =>

er1 := er\_loc\_root(0)(12 downto 0) ;

er2 := er\_loc\_root(1)(12 downto 0) ;

er3 := er\_loc\_root(2)(12 downto 0) ;

er4 := er\_loc\_root(3)(12 downto 0) ;

er1 := 8191 - (er1);

er2 := 8191 - (er2);

er3 := 8191 - (er3);

er4 := 8191 - (er4);

```
error_byte_location_OP_1 <= er1/8 ;
error_bit_location_OP_1 <= er1 mod 8;
error_byte_location_OP_2 <= er2 /8 ;
error_bit_location_OP_2 <= er2 mod 8;
error_byte_location_OP_3 <= er3/8 ;
error_bit_location_OP_3 <= er3 mod 8;
error_byte_location_OP_4 <= er4/8 ;
error_bit_location_OP_4 <= er4 mod 8;
num_of_error <= jc;

--temp_rx := rx;
--temp_rx(to_integer(unsigned(er1))) := not(temp_rx(to_integer(unsigned(er1))));
--temp_rx(to_integer(unsigned(er2))) := not(temp_rx(to_integer(unsigned(er2))));
--temp_rx(to_integer(unsigned(er3))) := not(temp_rx(to_integer(unsigned(er3))));
--temp_rx(to_integer(unsigned(er4))) := not(temp_rx(to_integer(unsigned(er4))));
--corr_msg_out=temp_rx; //output corrected msg declare at port
when 3 =>

er1 := er_loc_root(0)(12 downto 0) ;
er2 := er_loc_root(1)(12 downto 0) ;
er3 := er_loc_root(2)(12 downto 0) ;

er1 := 8191 - (er1);
er2 := 8191 - (er2);
er3 := 8191 - (er3);

error_byte_location_OP_1 <= er1/8 ;
error_bit_location_OP_1 <= er1 mod 8;
error_byte_location_OP_2 <= er2 /8 ;
error_bit_location_OP_2 <= er2 mod 8;
error_byte_location_OP_3 <= er3/8 ;
error_bit_location_OP_3 <= er3 mod 8;
```

```
num_of_error <= jc;
--temp_rx := rx;
--temp_rx(to_integer(unsigned(er1))) := not(temp_rx(to_integer(unsigned(er1))));
--temp_rx(to_integer(unsigned(er2))) := not(temp_rx(to_integer(unsigned(er2))));
--temp_rx(to_integer(unsigned(er3))) := not(temp_rx(to_integer(unsigned(er3))));
--corr_msg_out=temp_rx; //output corrected msg declare at port
when 2 =>
```

```
er1 := er_loc_root(0)(12 downto 0) ;--(er_loc_root(0) / 2);
er2 := er_loc_root(1)(12 downto 0) ;--(er_loc_root(1) / 2);
```

```
er1 := 8191 - (er1);
er2 := 8191 - (er2);
```

```
error_byte_location_OP_1 <= er1/8 ;
error_bit_location_OP_1 <= er1 mod 8;
error_byte_location_OP_2 <= er2 /8 ;
error_bit_location_OP_2 <= er2 mod 8;
```

```
num_of_error <= jc;
```

```
--temp_rx := rx;
--temp_rx(to_integer(unsigned(er1))) := not(temp_rx(to_integer(unsigned(er1))));
--temp_rx(to_integer(unsigned(er2))) := not(temp_rx(to_integer(unsigned(er2))));
--corr_msg_out=temp_rx; //output corrected msg declare at port
when 1 =>
```

```
er1 := er_loc_root(0)(12 downto 0) ;
```

```
er1 := 8191 - (er1);
```

```
error_byte_location_OP_1 <= er1/8 ;
error_bit_location_OP_1 <= er1 mod 8;
```

```
num_of_error <= jc;

--                               temp_rx := rx;
--                               temp_rx(to_integer(unsigned(er1)))
:= not(temp_rx(to_integer(unsigned(er1))));

when others => empty := '1';

end case;
end if;
end if;
end process;
end Behavioral;
```

## **BCH Decoder: Test Bench**

```
-----
-- Company:
-- Engineer:
--
-- Create Date: 14:16:32 06/04/2020
-- Design Name:
-- Module Name: F:/hdl codes/syndrome/syn_tb.vhd
-- Project Name: syndrome
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: syn_fifo_try
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
```

---

---



```
--  
-- Notes:  
-- This testbench has been automatically generated using types std_logic and  
-- std_logic_vector for the ports of the unit under test. Xilinx recommends  
-- that these types always be used for the top-level I/O of a design in order  
-- to guarantee that the testbench will bind correctly to the post-implementation  
-- simulation model.
```

```
-----  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;
```

```
-- Uncomment the following library declaration if using  
-- arithmetic functions with Signed or Unsigned values  
--USE ieee.numeric_std.ALL;
```

```
ENTITY syn_tb IS  
END syn_tb;
```

```
ARCHITECTURE behavior OF syn_tb IS
```

```
-- Component Declaration for the Unit Under Test (UUT)
```

```
COMPONENT syn_fifo_try  
PORT(  
    fpga_clk : IN std_logic;  
    BCH_RESET : IN std_logic;  
    bch_decoder_en : IN std_logic;  
    bch_decoder_data : IN std_logic_vector(7 downto 0)  
);  
END COMPONENT;
```

```
--Inputs  
signal fpga_clk : std_logic := '0';
```

```
signal BCH_RESET : std_logic := '0';
signal bch_decoder_en : std_logic := '0';
signal bch_decoder_data : std_logic_vector(7 downto 0) := (others => '0');

-- Clock period definitions
constant fpga_clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: syn_fifo_try PORT MAP (
        fpga_clk => fpga_clk,
        BCH_RESET => BCH_RESET,
        bch_decoder_en => bch_decoder_en,
        bch_decoder_data => bch_decoder_data
    );

    -- Clock process definitions
    fpga_clk_process :process
    begin
        fpga_clk <= '0';
        wait for fpga_clk_period/2;
        fpga_clk <= '1';
        wait for fpga_clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- BCH_RESET <= '1'; bch_decoder_en <= '1'; wait for fpga_clk_period;
```

```
-- *****TESTCASE 20 Bytes *****
--
--ip:
--   fpga_reset <= '1'; bch_enable <= '1';data <= "10100101";wait for fpga_clk_period;
--       for i in 1 to 10 loop
--   fpga_reset <= '0'; bch_enable <= '1';data <= "10101010";wait for fpga_clk_period;
--   end loop;
--   for i in 1 to 10 loop
--   fpga_reset <= '0'; bch_enable <= '1';data <= "11001100";wait for fpga_clk_period;
--
--       end loop;
--
--op:1000 00011001 11101111 11111000 01011011 00001001 11111000
--
--*****
BCH_RESET <= '1';bch_decoder_en <= '1'; wait for fpga_clk_period;

bch_decoder_data <= "-0000000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period; --rx(1023) msb
for i in 1 to 996 loop
bch_decoder_data <= "00000000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
end loop;
bch_decoder_data <= "00001010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;

bch_decoder_data <= "10101010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "10100010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;--10101010 err
bch_decoder_data <= "10101010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
```

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101010"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "10101101"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;--10101100 err

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;-- 11001100

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;

bch\_decoder\_data <= "11001100"; BCH\_RESET <= '0'; bch\_decoder\_en <= '1'; wait for fpga\_clk\_period;--11001100

```
bch_decoder_data <= "11001000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;--parity + 4b of data
bch_decoder_data <= "00011001"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "11101111"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "11111000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "01011011"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "00001001"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
bch_decoder_data <= "11111000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
```

```
---- =====enc working test case
1=====
-----ip=10101010,11000011,00001111,,
--
-----10101010=01010101,11000011=11000011,00001111=11110000
-----
-----op=51,0010 11010001 11011011 10100101 10010011 10100011 10111000 ,0
--
--bch_decoder_data <= "-0000000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period; --rx(1023) msb
--for i in 1 to 1013 loop
--bch_decoder_data <= "00000000"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;
--end loop;
--bch_decoder_data <= "00001010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for
fpga_clk_period;--data --00001010
```

```
--bch_decoder_data <= "10100100"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--original 10101100 err68  
--bch_decoder_data <= "00111001"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--00110000 error 57 59  
--  
--bch_decoder_data <= "11110010"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--parity- 11110010  
--bch_decoder_data <= "11010001"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--11010001  
--bch_decoder_data <= "11011011"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--11011011  
--bch_decoder_data <= "10100101"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;  
--bch_decoder_data <= "10010011"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;--10010011  
--bch_decoder_data <= "10100011"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period;  
--bch_decoder_data <= "10111001"; BCH_RESET <= '0';bch_decoder_en <= '1'; wait for  
fpga_clk_period; --rx(0)10111000 err 1
```

end process;

END;









# 4<sup>th</sup> World Conference on SMART TRENDS IN SYSTEMS, SECURITY AND SUSTAINABILITY

London, United Kingdom

## Certificate

This is to certify that

**BHAVANA R. REDDY, HRUSNA CHAKRI SHADAKSHRI V., SHARATH ANAND, SHARMILA K. P.**

has contributed a paper titled

**Error-free Communication in NB-IoT using ECC Approach**

in 4<sup>th</sup> World Conference on Smart Trends in Systems, Security & Sustainability (WorldS4 2020)  
held during July 27 - 28, 2020. The conference was held through digital platform ZOOM.

The paper has also been selected for publication in the (WorldS4) conference as per fulfilment of guidelines issued by IEEE.

We wish the authors all the very best for future endeavors.

**MIKE HINCHEY**

Chair - IFIP  
Past Chair - IEEE UK & Ireland Section

**NILANJAN DEY**

Publication Chair  
WorldS4 2020

**XIN-SHE YANG**

Conference Chair  
WorldS4 2020

**AMIT JOSHI**

Organising Secretary, WorldS4 2020  
Chair- Inter YIT, IFIP

