DESIGN OF ERROR DETECTOR HARDWARE USING REED SOLOMON CODE

A project submitted by

Name	Roll No.	Reg. No.
TABREJ ALAM	**12000315135**	**151200120021**
SONU SHRIKANT	**12000314096**	**141200110278**
AMRESH KUMAR	**12000314010**	**141200110192**
OM PRAKASH	**12000314060**	**141200110242**
CHANDAN Kr. SINGH	**12000314026**	**141200110208**

In partial fulfillment of the requirements for the degree of

Electronics & Communication Engineering

BACHELOR OF TECHNOLOGY

Under the guidance of

Prof. Pradipta Sarkar

Department of Electronics & Communication Engineering
(Affiliated to Maulana Abul Kalam Azad University of Technology)
Dr. B. C. Roy Engineering College
Durgapur-713 206, West Bengal, India

2018

Department of Electronics & Communication Engineering

Dr. B. C. Roy Engineering College

Durgapur-713 206, India

Date- 16.05.2018

Prof. Pradipta Sarkar
Asst. Professor
Department of Electronics & Communication Engineering,
E-mail – pradipta.sarkar@bcrec.ac.in

CERTIFICATE

This is to certify that this project entitled "**Design of Error Detector Hardware Using Reed Solomon Code"** is a bonafide record of the project work carried out by **Mr. Sonu Shrikant,** under my supervision in the Department of Electronics & Communication Engineering, Dr. B. C. Roy Engineering College. The results embodied in this work or parts of it have not been presented / communicated elsewhere.

Prof. Pradipta Sarkar

DECLARATION

I hereby declare that the work presented in this thesis entitled **" Design of Error Detector Hardware Using Reed Solomon Code** " is a bonafide record of the project work done by us under the supervision of **Prof. Pradipta Sarkar,** Department of Electronics & Communication Engineering, Dr. B. C. Roy Engineering College, Durgapur-713206, India and that no part thereof has been presented/ communicated elsewhere.

Durgapur

Date- 16.05.2018

E-mail – Sonu Shrikant …….

shrikantsonu92@gmail.com

Department of Electronics & Communication Engineering,

Dr. B. C. Roy Engineering College, Durgapur-713 206, India

ACKNOWLEDGEMENT

My greatest appreciation, sincere gratitude and thanks to Prof. Pradipta Sarkar, Asst. Professor in the Department of Electronics and Communication Engineering, Dr. B. C. Roy Engineering College for his proper guidance and constant encouragement and friendship behavior throughout our project work. He has been an outstanding teacher, guidance and mentor during our project work and I have learned a lot from him. I have been extremely fortunate to get a chance to work under his guidance in the laboratory.

I would like to express Our sincere thanks to Prof(Dr). N. N. Pathak, Head, Department of Electronics & Communication Engineering, Dr. B. C. Roy Engineering College for his support in this work.

I would like to express our thanks to all the teaching and non teaching staff of the Department of Electronics & Communication Engineering, Dr. B. C. Roy Engineering College for their helpful and friendship attitude during Our project work.

Finally, I would like to express our thanks to all my colleagues, Department of Electronics & Communication Engineering, Dr. B.C. Roy Engineering College, Durgapur, India, for providing moral support in this work.

Sonu Shrikant

ABSTRACT

In the present world, communication has got many applications such as telephonic conversations etc. in which the messages are encoded into the communication channel and then decoding it at the receiver end. During the transfer of message, the data might get corrupted due to lots of disturbances in the communication channel. So it is necessary for the decoder tool to also have a function of correcting the error that might occur. Reed Solomon codes are type of burst error detecting codes which has got many applications due to its burst error detection and correction nature. Our aim of the project is to implement this reed Solomon codes in a Verilog test bench, waveform and also to analyze the error probability that is occurring during transmission.

To perform this check one can start with simulating reed Solomon codes in MATLAB and then going for simulation in XILINX writing the Verilog code. The encoder and decoder design of reed Solomon codes have got different algorithms. Based on the requirements I can use those algorithms. There are difference between the algorithms is that of the computational calculations between them. The complexity of the code depends on the algorithm used**.** I will be using Linear Feedback Shift Register circuit for designing the encoder.

LIST OF CONTENTS

LIST OF CHAPTERS

LIST OF SYMBOLS

n =Codeword length

k=Message bits

t=No. of error symbols

α = Primitive element

P(x) = Field generator polynomial

g(x) =Code generator polynomial

M(x) = Message polynomial

q(x) =quotient

r(x) =remainder

T(x) =Transmitted codeword

Λ and Ω =Output produced

ϒ =Constant value

LIST OF ABBREVIATOINS

DRAM =Dynamic Random-access Memory

ARQ = Automatic Repeat Request

FEC = Forward Error Correction

DVD = Digital Versatile Disc

ADSL = Asymmetric Digital Subscriber Line

HDSL = High-Bit Digital Subscriber Line

RS = Reed Solomon

GF = Galois Field

HDL = Hardware Description Language

BCH = Bose-Chaudhuri-Hocquenghem

DVB-T = Digital Video Broadcasting-"Terrestrial"

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Communication Engineering is been the vital field of engineering in last few decades Evolution of digital communication has made this field more interesting as well as challenging. All the advancements in the field of communication are to achieve two important goals namely reliability and efficiency. In most cases reliability is given the priority over efficiency though at certain cases one is compromised for the other. Reliability of communication has an impact even in our day to day life. For example message received on our mobile phone may become unreadable if some error occurs during the transmission, or a scratch in our DVD may make it unreadable. There are wide ranges of concern in the field of digital communication. Error control issues have been addressed in this thesis.

1.2 Errors and methods

Generally communication is understood as transmission and reception of data from one place to other at some distance. If we change the reference it can also include transmission and reception of data at the same place but at a different point of time, which means storage and retrieval of data. Hence storage is also a part of communication. In any system application we come across errors either in communication or in storage. Errors in transmission are mainly because of noise, electromagnetic interferences, cross talk, bandwidth limitation, etc. In case of storage, errors may occur because of increase in magnetic flux as in case of magnetic disc or it can be spurious change of bits because of electromagnetic interferences as in case of DRAM. Hence dealing with these errors when they occur is the matter of concern. The first step is to detect the error. And after the error gets detected there are two alternate approaches to proceed.

- Automatic repeat request (ARQ) : In this approach, the receiver first detects the error and then sends a signal to the transmitter to retransmit the signal. This can be done in two ways:
- (i) continuous transmission mode (ii) wait for acknowledgement. In continuous transmission mode, the data is being sent by the transmitter continuously. Whenever receiver finds any error it sends a request for retransmission. However, the retransmission can either be selective repeat or go back N step type. As the name suggests, in selective repeat those data units containing error are only retransmitted. While in go back N type, retransmission of last N data unit occurs. Next, in wait for acknowledgement mode, acknowledgement is sent by the receiver after it correctly receives each message. Hence, when not sent, the retransmission is initiated by the transmitter. Forward error correction (FEC): In this approach, error is both detected and corrected at the receiver. end. To

enable the receiver to detect and correct the data, some redundant information is sent with the actual information by the transmitter.

After being introduced to both the approaches, one should choose whether which approach is to be used. Automatic repeat request is easier but if the error occurs much frequently, then retransmission at that frequency will particularly reduce the effective rate of data transmission. However, in some cases retransmission may not be feasible to us. In those cases,Forward Error correction would be more suitable. As Forward Error Correction involves additional information during transmission along with the actual data. It also reduces the effective data rate which is independent of rate of error. Hence, if error occurs less frequently then Automatic request approach is followed keeping in mind that retransmission is feasible.

Out of the various FEC's, Reed Solomon code is one. These are block error correcting codes with wide range of applications in the field of digital communications. These codes are used to correct errors in devices such as CD's, DVD,s etc.., wireless communications, many digital subscriber lines such as ADSL,HDSL etc…

They describe a systematic way of building codes that can detect and correct multiple errors. In a block code we have k individual information bits, r individual parity bits and a total of n (=k+r) bits. However, reed Solomon codes are organized in group of bits. This group of bits are referred to as symbols. So we can say, this code has n number of symbols. Each symbol comprises of m number of bits, where

$$\mathbf{n(max)=2^m-1}$$

1.3 Scope of work

As stated in the last paragraph, RS codes are used for many applications. With the objective of developing high speed RS codes, the scope includes: Study of different error detection and correction methods, implementation of RS encoder and decoder in Hardware Description Language (HDL), building fully synchronous synthesizable logic core of RS encoder and decoder to meet the requirement of almost all the standards that employ RS codes, such as CCSDS, DVB, ETIS-BRAN, IEEE802.6, G.709, IESS-308, etc.

CHAPTER 2

LITERATURE SURVEY

2.1 Summary of Previous Papers

This chapter represents an overview of background research to the project. The literature also gives the brief idea about the technical, operational and economical feasibility study.

a. **Kenny Chung Chung Wai, Dr. Shanchieh Jay Yang" Field Programmable Gate Array Implementation of Reed- Solomon Code, RS(255,239)"In proceeding of 9th Annual Military and Aerospace programmable logic Device International Conference,2 september 2011.** Authors Kenny Chung Chung Wai, Dr. Shanchieh Jay Yang introduced a FPGA implementation of Reed Solomon. It is synthesized to Altera's Stratix II and benchmarks are run against Altera's Reed Solomon code. Author was designed Xelic's encoder which is measured to be about half the size of Altera's encoder.[1]

b. **Joaquin Garcia, Rene Cumplido Department of Computer Science , "On the design of an FPGA-Based OFDM modulator for IEEE 802.16-2004" INAOE, Puebla, Mexico 2005** Author Joaquin Garcia introduced OFDM using System generator and Matlab & Simulink.The results on hand show that it is possible to implement an OFDM modulator for IEEE Std.802.16 using Virtex II. In this paper the author implement configurable system that can be implemented for different modulation technique. The future work of this paper is Implementation of the FEC modulator, demodulator. [3]

c. **K. Harikrishna, T. Rama Rao, and Vladimir A. Labay, "FPGA Implementation of FFT Algorithm for OFDM Based IEEE 802.16d (Fixed WiMAX) Communications" Journal Of Electronic Science And Technology, Vol. 8, No. 3,September 2010.**
Authors K. Harikrishna, T. Rama Rao, and Vladimir A. Labay, introduced a high level implementation of a high performance FFT for OFDM modulator and demodulator of 802.16d. The design has been coded in Verilog and besieged into Xilinx Spartan3 field programmable gate arrays. The design of the FFT is implemented and applied to fix WiMAX—IEEE 802.16d communication standard. [4]10

d. **Miljko Bobrek, Kenyon H. Clark, Austin P. Albright "FPGA Implementation of Reed-Solomon Decoder for IEEE 802.16 WiMAX Systems using Simulink-Sysgen Design Environment" Oak Ridge National Laboratory Cognitive Radio Program 1 Bethel Valley Rd Oak Ridge ,TN 37831.**
Authors Miljko Bobrek, Kenyon H. Clark, Austin P. Albright introduced FPGA implementation of the Reed- Solomon decoder for used in IEEE 802.16 WiMAX systems.The decoder is based on RS (255,239) code. It is additionally shortened and punctured according to the WiMAX specifications. A Simulink model was used for simulation and hardware implementation. It is based on the System Generator library of low-level Xilinx blocks [6]

e. **M.A. Mohamed, A.S. Samarah, M.I. Fath Allah , "Implementation of the OFDM Physical Layer Using FPGA" IJCSI International Journal of Computer Science Issues,Vol. 9, Issue**

2, Nov2, March 2012
Author M.A. Mohamed, A.S. Samarah, M.I. Fath Allah introduced the design and implementation of OFDM system. Author has tested system performance by considering various design parameters using MATLAB 2011. All modules are coded using VHDL programming language.[7]

f. **Yuval Cassuto, Jehoshua Bruck,Robert J.McEliece "On the Average Complexity of Reed–Solomon List Decoders" IEEE Vol. 59, No. 4, April 2013.**
Authors Yuval Cassuto, Jehoshua Bruck,Robert J.McEliece**,** introduced the number of monomials required to interrupt a received word in an algebraic list decoder for Reed–Solomon codes depends on the instantaneous channel error. On the basis of logical side, this paper studies the dependence of interpolation costs on instantaneous errors, in both hard- and soft-decision decoders.[8]

g. **Li Chen et.al. "Progressive Algebraic Soft-Decision Decoding of Reed-Solomon Codes" IEEE Vol. 61, No. 2, February 2013**
Author Li Chen, introduced algebraic soft-decision decoding (ASD) algorithm. They are a polynomial-time soft decoding algorithm for Reed- Solomon (RS) codes. It breaks both the 11 algebraic hard decision decoding (AHD) and the conventional unique decoding algorithms;
this paper proposes a progressive ASD (PASD) algorithm that enables the conventional ASD algorithm to perform decoding with an adjustable designed factorization output list size (OLS).[9]

h. **F. Abdelkefi, J. Ayadi "Reed-Solomon code-based sparse channel estimation for OFDM systems" IEEE Vol. 48 27th September 2012.**
Authors F. Abdelkefi, J. Ayadi, introduced a novel efficient algorithm for the opinion of sparse channel impulse response (CIR) is addressed for OFDM systems. The innovation of this algorithm comes from the fact that it equivalently see the CIR estimation problem as a decoding one. To do so, it uses first the channel sparsely through the modeling of the sparse CIR as a Bernoulli-Gaussian process. Then, using the relationship between the Reed-Solomon codes and the OFDM modulator it efficiently estimates the sparse CIR using
directly the decoding of the OFDM received signal. The obtain simulation results highlight that using the planned algorithm gives good estimation performance in terms of mean squares error on the sparse CIR estimates. Which concludes that using a multicarrier OFDM
transmission system, pilot tones can be seen as virtual RS code words. [10]

i. **Jong Kang Park And Jong Tae Kim, " A Soft IP Compiler For Reed-Solomon Decoder", http://www.sipac.org/ap-soc/index_k/index_a/source/A04_lec.pdf.**
Author Christian scholar introduced VHDL code generator for Reed-Solomon decoders using Euclid's algorithm. The VHDL code is synthesizable, and area and speed metrics for several decoder designs is presented, targeted to Xilinx Field Programmable Gate Arrays. The core generator also generates the vectors for the test bench. The code generator is capable of generating VHDL code for most binary FEC decoder, such as Hamming, Cyclical Redundancy Check (CRC), and BCH codes. In the design of the Reed-Solomon decoder, a bank of Galois multipliers is used, as opposed to a dedicated multiplier where it is needed.This result is less area, but requires tristate buses and reliable scheduling of the steps to avoid collisions.[30]

CHAPTER 3

OBJECTIVE OF THE WORK

Reed-Solomon Codes are error-correcting linear cyclic systematic non-binary block 8 code.This codes range from data retrieval from bar codes and QR codes in our daily lives for sending transmissions to and from spacecrafts launched in deep-space missions. The Reed-Solomon (RS) Code was discovered by Irving Reed and Gus Solomon. Since then, RS Codes have been an huge contributor to the telecommunications revolution in the last half of the twentieth century. Reed- Solomon codes are the most frequently used digital error control codes they are used in computer memory and non- volatile memory applications. Some of the applications include the Digital Audio Disk, Deep Space Telecommunication Systems, and Error Control for Systems with Feedback, Spread-Spectrum Systems, and Computer Memory's codes are generated and corrected using Reed-Solomon encoder and decoder In the encoder Redundant symbols are generated using a generator polynomial and appended to the message symbols. In decoder error location and magnitude are calculated using the same generator polynomial. Then the correction is applied on the received code. Reed Solomon code has less coding gain as compared to LDPC and turbo codes. But it has very high coding rate and low complexity. Hence it is suitable for many applications including storage and transmission.
In digital communication, Reed-Solomon (RS) codes refer to as a part of channel coding that had becoming very significant to better withstand the effects of various channel impairments such as noise, interference and fading. This signal processing technique is designed to improve communication performance and can be deliberate as medium for accomplishing desirable system trade-offs.Galois field arithmetic is used for encoding and decoding of Reed– Solomon codes.

Galois field arithmetic is used for encoding and decoding of Reed – Solomon codes. Galois field multipliers are used for encoding the information block. The encoder attaches parity symbols to the data using a predetermined algorithm before transmission. At the decoder, the syndrome of the received codeword is calculated. VHDL implementation creates as flexible, fast method and high degree of parallelism for implementing the Reed –Solomon codes. The purpose of this thesis is to evaluate the performance of RS coding system using M-ary modulation over Additive White Gaussian Noise AWGN channel and implementation of RS encoder in VHDL.

Computer simulation tool and MATLAB will be used to create and run extensively the entire simulation model for performance evaluation and VHDL is used to implement the design of RS encoder.

The purpose of the class project is to expose to you to more advanced practical and theoretical issues in channel coding. The project should be completed in groups of two. The report is due on the day after finals and should be about 5 pages of typeset text including the project description, results, figures, and references. I suggest typesetting the report using LATEX or LyX (a simplified front-end for LATEX), but this is not required. Students are allowed to choose their topic of the project as long as it has a close connection to the material presented in this class. For those interested in a well-defined assignment, there is also a standard project that involves the concatenation of an outer Reed-Solomon code with an inner convolutional code.

CHAPTER 4

BASICS OF COMMUNICATION CODING THEORY

4.1 General communication systems

Communication is the phenomenon of transmitting information. This transmission can either be made between two distinct places (for ex, a phone call) or between two points in time, for example the writing of this thesis so that it can be read later on. We shall restrict ourselves to the study of digital communication. That is, the transmission of messages that are sequences of symbols taken from a set called alphabet.

Digital communication has become predominant in today's world. It ranges from internet, storage disks, satellite communication to d i g i t a l television and so on. Moreover, any analog system can be transformed into digital data by various sampling and signal transformation methods. Typical examples include encoding music in an mp3, numerical cameras, voice recognition and many others. A digital communication system, in all its generality is represented in figure 2.1.

- The information source outputs the data to be communicated. It produces messages to be transmitted to the receiving destination. When it is a digital source, these messages are sequences of symbols taken from a finite alphabet.
- The transmitter takes the source data as input and produces an associated signal suited for the channel. Transmitter aims to achieve one or more of the followings

➢ A maximum of information transmission per unit of time. This is directly connected to data compression techniques taking advantage of the statistical structure of the data.

➢ To ensure a reliable transmission across the noisy channel. In other words, to make it fault tolerant to errors introduced by the channel. This is typically done by adding structured redundancy in the message.

➢ To provide message confidentiality. This typically involves encryption which hides or scrambles the message so that unintended listeners cannot discern the real information content from the message.

- The physical channel is the medium used to transmit the signal from the source to the destination. Examples of channels conveying information is conveyed over space like telephone lines, fiber-optic lines, microwave radio channels. . . Information can also be conveyed between two distinct times like for example by writing data on a computer disk or a DVD and retrieving it later. As the signal propagates through the channel, or on its storage place, it may be corrupted. For example, the telephone lines suffer from parasitic currents, waves are subject to interference issues, a DVD can be scratched... But these perturbations are regrouped under the term of noise. The more noise, the more the signal is altered and the more it is difficult to retrieve the information originally sent. Of course,

there are many other reasons for errors like timing jitter, attenuation due to propagation, carrier offset... But all these perturbations lie beyond the scope of this thesis.

Figure 4.1: General Communication System

- The receiver ordinarily performs the inverse operation done by the transmitter. It reconstructs the original message from the received signal.

The destination is the system or person for whom the message is intended.

4.2 Need For Communication

As was said in the previous section, the transmitter can have several roles together. To compress data, to secure data, to make it more reliable and lastly to transmit it as signals suited for the physical channel. Compressing data is also called source coding; it consists of mapping sequences of symbols in the original data stream to shorter ones. This is done based on the statistical distribution of the original data: the most frequent sequences are mapped to shorter ones while rare sequences are mapped to longer ones. By doing this, the resulting sequences are on average shorter, i.e. Sequences with fewer symbols. On the opposite, in order to make the sequence of symbols robust to errors, redundancy is added to it. This is called channel encoding and consists of mapping shorter sequences to longer ones so that if a few symbols are corrupted the original data can nevertheless be found back. More often we need both of these. This seems contradictory since one reduces the number of sent symbols while the other increases it. However, it is not really. The source coding reduces the redundancy of unstructured data which would not provide protection if symbols were corrupted. For example, despite knowing that a message contains on average 99% of zeros, you cannot know which bits were corrupted when sending the message as it is. On the opposite, channel coding adds structured data to improve protection against such errors during the transmission. By taking the compressed message and repeating three times each bit, you can decode correctly up to one error per three bits introduced. One could wonder if a technique performing both in a single step could be more efficient than doing it sequentially. It turns out that performing source and channel coding sequentially tends to be optimal when the treated sequence length tends to infinity. This is known as the source-channel coding separation theorem and is one of the results of Shannon's ground breaking work. For finite sequence length, such joint encoding techniques are still a subject of research. Until now, we spoke only about symbols and about mapping sequences of symbols to other sequences of symbols with better properties. However, the physical channel does not, technically speaking, transmit symbols but signals (waves, voltage, etc...). We however assume a one-to- one mapping between symbols and signals which is done by a modulator to map a symbol to the corresponding signal and a demodulator mapping back a received signal to a received symbol, or information about

the likelihood of each potential symbol. Notice that by separating the source and channel coding, an encryption module can also be conveniently inserted between both. Putting all together, the obtained refined communication model is illustrated in figure 2.2 in the bottom of this page.

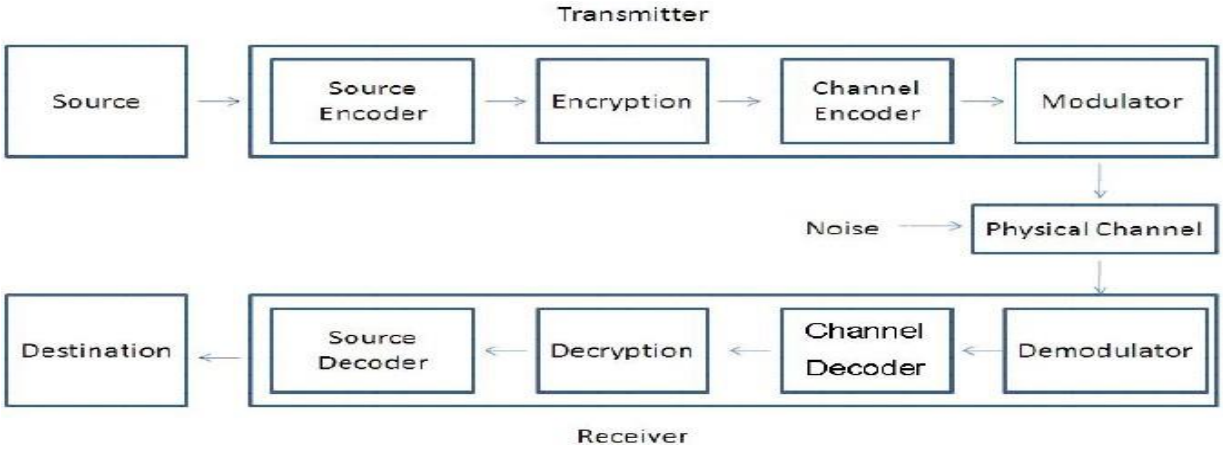

Fig 4.2 Needs of Communication Systems

All three modules: compression, encryption, channel coding are of course optional and not necessarily included in a communication system. The part of interest for us is channel encoding and decoding. As a side note, but important, one consequence of source coding or encryption is that any of them tends to produce equal probability sequences of symbols. This argument will support an important assumption for the input of the channel encoder later on. Moreover, even if these modules are not present and nothing is known about the source's output, this is still the best assumption that can be made. The main part of interest for us is the channel encoder and decoder and it can now be isolated from the remaining system. Lastly, the modulator and demodulator constitute the glue between the transmitter, the physical channel and the receiver. The channel can be considered as the modulator, the physical channel and the demodulator together, providing an abstract model having as input and output symbols. However, the received signals, altered by noise, may not match any of the sent ones. So either it is mapped to other symbols in a bigger alphabet or some threshold decision must be used to decide which symbol of the original alphabet it should be. This is seen in more details in the section about channels.

4.3 Channel Encoder

The role of the encoder is to add structured redundancy to sequences of symbols. There are different ways of doing this but the biggest classes of encoders are by far block encoders. They work by cutting the data stream in blocks (of symbols) of fixed size and encoding these blocks one after another. Such a block, a finite sequence of symbols, is also known as a word.

Definition 2.1: A word $W = (w_1, \ldots\ldots, w_m)$ of length m over an alphabet A is an ordered sequence of m symbols taken from A.

The data block is called the message word. By hypothesis, it has a fixed length, say k. The encoder maps each such word to a longer one, called code word, also of fixed size, say n. A reasonable

assumption is that both words, the message and the encoded code word, are defined over the same alphabet Aq. The encoder is thus a mapping from words of length k to words of length n where n > k. Words can be seen as points in a space over the alphabet. In this light, the encoder is a mapping of points in A^k_q to points in A^n_q . If the alphabet has q elements, then **qk** message words are mapped onto **qk** from **qn** possible words. The set of all code words is thus only a subset of An q and this set forms the code An q which is the image of the encoder function. On a closer look, it turns out that what is of interest is not the mapping, but the code C itself.

4.4 Basics of coding

As we have discussed in previous chapter a code is the set of all the encoded words, the code words that an encoder can produce. That means when actual set of data is encoded it becomes a code.

Definition 2.2: A q_ary code C of length n is a set of code words in A_n, where A is an alphabet of q symbols. The size of the code, noted |C|, is the number of code words in the code.

Example 2.1: A ternary code of length 5 is the following set:
01201

00210

12012

20021

12120

10021
The ternary alphabet used is A = {0, 1, 2} and the code size is |C| = 6 (It contains 6 code words). Code words can be seen as vectors in the space A_n where the i^{th} symbol is the i^{th} coordinate. To compare words, the space A_n can be equipped with a convenient metric called the Hamming distance.

Definition 2.3: The Hamming distance between two words x; yϵA_n is the number of co-ordinates in which symbols differ.

$$d_H(x; y) = |i \mid x_i \neq y_i \mid$$

Example 2.2: d_H(ASHUTOSH; PARITOSH) = 4
d_H(001010; 010010) = 2
Minimal distance is a basic and important property of a code as it is directly related to the error correcting capability of the code. For nearest neighbor decoding, in order for the code to provide unambiguous decoding of the received code words up to e errors, it is necessary and sufficient that the minimal distance be at least d = 2e+1 . This follows directly from the triangle inequality since no received word w can lie at distance less than or equal to e to two code words if they are all at a distance d = 2e + 1.

CHAPTER 5

BACKGROUND THEORY

5.1 Classification of Reed-Solomon Code

There are many categories of error correcting codes, the main ones being block codes and convolution codes. A Reed-Solomon code is a block code, meaning that the message to be transmitted is divided up into separate blocks of data. Each block then has parity protection information added to it to form a self-contained code word. It is also, as generally used, a systematic code, which means that the encoding process does not alter the message symbols and the protection symbols are added as a separate part of the block. This is shown diagrammatically in Figure 3.1.

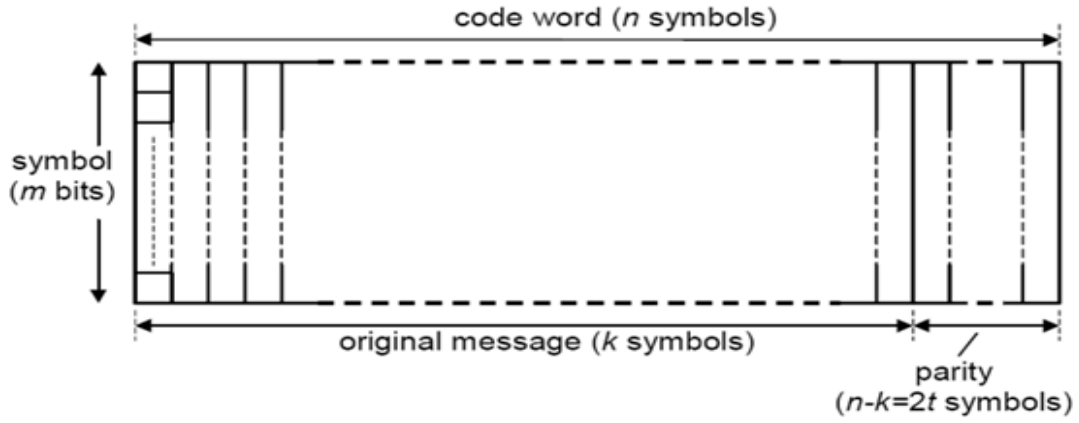

Figure 5.1 - Reed-Solomon code definitions

Also, a Reed-Solomon code is a linear code (adding two code words produces another code word) and it is cyclic (cyclically shifting the symbols of a code word produces another code word). It belongs to the family of Bose-Chaudhuri-Hocquenghem (BCH) codes [3, 4], but is distinguished by having multi-bit symbols. This makes the code particularly good at dealing with bursts of errors because, although a symbol may have all its bits in error, this counts as only one symbol error in terms of the correction capacity of the code.

Choosing different parameters for a code provides different levels of protection and affects the complexity of implementation. Thus a Reed-Solomon code can be described as an (n, k) code, where n is the block length in symbols and k is the number of information symbols in the message. Also

$$n \leq 2^m - 1 \quad \text{.....} \quad (1)$$

where m is the number of bits in a symbol. When (1) is not an equality, this is referred to as a shortened form of the code. There are n-k parity symbols and t symbol errors can be corrected in a block, where

$t = (n-k)/2$ for $n-k$ even

or

$t = (n-k-1)/2$ for $n-k$ odd.

Unfortunately, the notation used in the literature on error correcting codes, although generally similar, is not universally consistent. Although the notation used here is representative of that used elsewhere, the reader should be aware that confusing differences in terminology can arise. It is therefore particularly important to be clear about how the parameters of a code are specified.

5.2 Galois field

To proceed further requires some understanding of the theory of finite fields, otherwise known as Galois fields after the French mathematician.

5.2.1 Galois field elements:

A Galois field consists of a set of elements (numbers). The elements are based on a primitive element, usually denoted α, and take the values:

$0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{N-1}$ .....(2)

to form a set of 2^m elements, where $N=2^m - 1$. The field is then known as GF(2^m). The value of α is usually chosen to be 2, although other values can be used. Having chosen α, higher powers can then be obtained by multiplying by α at each step. However, it should be noted that the rules of multiplication in a Galois field are not those that we might normally expect. This is explained in Section 3.2.4.

$$a_{m-1} x^{m-1} + .. + a_1 x + a_0$$

where the coefficients a_{m-1} to a_0 take the values 0 or 1. Thus we can describe a field element using the binary number $a_{m-1}\, a_1 a_0$ and the 2^m field elements correspond to the 2^m combinations of the m-bit number.

For example, in the Galois field with 16 elements (known as GF(16), so that m=4), the polynomial representation is:

$$a_3 x^3 + a_2 x^2 + a_1 x^1 + a_0 x^0$$

with $a_3 a_2 a_1 a_0$ corresponding to the binary numbers 0000 to 1111. Alternatively, we can refer to the field elements by the decimal equivalents 0 to 15 as a short-hand version of the binary numbers.

Arithmetic in a finite field has processes of addition, subtraction, multiplication and division, but these differ from those we are used to with normal integers. The effect of these differences is that any arithmetic combination of field elements always produces another field element.

5.2.2 Galois field addition and subtraction:

When we add two field elements, we add the two polynomials:

$(a_{m-1}\, x^{m-1} + \ldots\ldots\ldots\ldots a_1\, x^1 + a_0 x^0) + (b_{m-1}\, x^{m-1} + \ldots\ldots\ldots\ldots a_1\, x^1 + b_0 x^0) = c_{m-1}\, x^{m-1} + \ldots\ldots\ldots c_1\, x^1 + c_0 x^0$

Where,

$c_{1=}\, a_1 b_1$ for $0 =< i =< m-1$

However, the coefficients can only take the values 0 and 1, so

$c_i = 0$ for $a_i = b_i$

and

$c_i = 1$ for $a_i \neq b_i$ .....*(3)*

Thus two Galois field elements are added by modulo-two addition of the coefficients, or in binary form, producing the bit-by-bit exclusive-OR function of the two binary numbers.

For example, in GF(16) we can add field elements $x^3 + x$ and $x^3 + x^2 + 1$ to produce $x^2 + x + 1$. As binary numbers, this is:

1010 + 1101 = 0111

or as decimals

10 + 13 = 7

which can be seen from:

1010 (10)
1101 (13)
0111 (7)

Because of the exclusive-OR function, the addition of any element to itself produces zero. So we should not be surprised that in a Galois field:

2 + 2 = 0.

Table 7 in Appendix 8.3 provides a look-up table for additions in GF(16).

Subtraction of two Galois field elements turns out to be exactly the same as addition because although the coefficients produced by subtracting the polynomials take the form:

$c_i = a_i - b_i$ for $0 \leq i \leq m-1$

the resulting values for c_i are the same as in (3). So, in this case, we get the more familiar result:

2 - 2 = 0

and for our other example:

10 - 13 = 7.

It is useful to realise that a field element can be added or subtracted with exactly the same effect, so minus signs can be replaced by plus signs in field element arithmetic.

5.2.3 The field generator polynomial:

An important part of the definition of a finite field, and therefore of a Reed-Solomon code, is the field generator polynomial or primitive polynomial, $p(x)$. This is a polynomial of degree m which is irreducible, that is, a polynomial with no factors. It forms part of the process of multiplying two field elements together. For a Galois field of a particular size, there is sometimes a choice of suitable polynomials. Using a different field generator polynomial from that specified will produce incorrect results.

For GF(16), the polynomial

$$p(x) = x^4 + x + 1 \qquad \text{..} \ (4)$$

is irreducible and therefore will be used in the following sections. An alternative which could have been used for GF(16) is

$$p(x) = x^4 + x^3 + 1.$$

5.2.4 Constructing the Galois field:

All the non-zero elements of the Galois field can be constructed by using the fact that the primitive element α is a root of the field generator polynomial, so that

$$p(\alpha) = 0.$$

Thus, for GF(16) with the field generator polynomial shown in (4), we can write:

$$\alpha^4 + \alpha + 1 = 0$$

or

$\alpha^4 = \alpha + 1$ (remembering that + and – are the same in a Galois field).

Multiplying by α at each stage, using $\alpha + 1$ to substitute for α^4 and adding the resulting terms can be used to obtain the complete field as shown in Table 1. This shows the field element values in both index and polynomial forms along with the binary and decimal short-hand versions of the polynomial representation.

If the process shown in Table 1 is continued beyond α^{14}, it is found that $\alpha^{15} = \alpha^0$, $\alpha^{16} = \alpha^1$, so that the sequence repeats with all the values remaining valid field elements.

index form	polynomial form	binary form	decimal form
0	0	0000	0
α^0	1	0001	1
α^1	α	0010	2
α^2	α^2	0100	4
α^3	α^3	1000	8
α^4	$\alpha + 1$	0011	3
α^5	$\alpha^2 + \alpha$	0110	6
α^6	$\alpha^3 + \alpha^2$	1100	12
α^7	$\alpha^3 + \alpha + 1$	1011	11
α^8	$\alpha^2 + 1$	0101	5
α^9	$\alpha^3 + \alpha$	1010	10
α^{10}	$\alpha^2 + \alpha + 1$	0111	7
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1110	14
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1111	15
α^{13}	$\alpha^3 + \alpha^2 + 1$	1101	13
α^{14}	$\alpha^3 + 1$	1001	9

Table 5.1- The field elements for GF(16) with $p(x) = x^4 + x + 1$

5.2.5 Galois field multiplication and division:

Straightforward multiplication of two polynomials of degree m-1 results in a polynomial of degree 2m-2, which is therefore not a valid element of GF(2^m). Thus multiplication in a Galois field is defined as the product modulo the field generator polynomial, $p(x)$. The product modulo $p(x)$ is obtained by dividing the product polynomial by $p(x)$ and taking the remainder, which ensures that the result is always of degree m-1 or less and therefore a valid field element.

For example, if we multiply the values 10 and 13 from GF(16) represented by their polynomial expressions, we get:

$$(x^3 + x)(x^3 + x^2 + 1) = x^6 + x^5 + x^3 + x^4 + x^3 + x$$

$$= x^6 + x^5 + x^4 + x \qquad \text{..... (5)}$$

To complete the multiplication, the result of (5) has to be divided by $x^4 + x + 1$.

Division of one polynomial by another is similar to conventional long division. Thus it consists of multiplying the divisor by a value to make it the same degree as the dividend and then subtracting (which for field elements is the same as adding). This is repeated using the remainder at each stage until the terms of the dividend are exhausted. The quotient is then the series of values used to multiply the divisor at each stage plus any remainder left at the final stage.

This can be shown more easily by arranging the terms of the polynomials in columns according to their significance and then the calculation can be made on the basis of the coefficient values (0 or 1) as shown below.

	x^6 x^5 x^4 x^3 x^2 x^1 x^0
dividend:	1 1 1 0 0 1 0
divisor × x^2:	1 0 0 1 1
	1 1 1 1 1
divisor × x:	1 0 0 1 1
	1 1 0 0 0
divisor × 1:	1 0 0 1 1
	1 0 1 1

So the quotient is $x^2 + x + 1$ and the remainder, which is the product of 10 and 13 that we were originally seeking, is $x^3 + x + 1$ (binary 1011 or decimal 11). So we can write:

$$10 \times 13 = 11.$$

Table 8 in Appendix 8.3 provides a look-up table for multiplications in GF(16) with the field generator polynomial of equation (4).

Alternatively the process of equation (5) can be performed using the coefficients of the polynomials in columns. First shifted versions of $x^3 + x$ are added according to the non-zero coefficients of $x^3 + x^2 + 1$. Then, instead of the division process, we can use substitutions taken from Table 1 for any non-zero terms that exceed the degree of the field elements and add these as shown:

	x^6 x^5 x^4 x^3 x^2 x^1 x^0
× x^3:	1 0 1 0
× x^2:	1 0 1 0
× 1:	1 0 1 0
	1 1 1 0 0 1 0
$x^4 = x + 1$	0 0 1 1
$x^5 = x^2 + x$	0 1 1 0
$x^6 = x^3 + x^2$	1 1 0 0
	1 0 1 1

A further alternative technique for multiplication in a Galois field, which is also convenient for division, is based on logarithms. If the two numbers to be multiplied are represented in index form, then the product can be obtained by adding the indices modulo 2^m-1. For example, by inspecting the values in Table 1 we find:

$$10 = \alpha^9 \quad \text{and} \quad 13 = \alpha^{13}$$

so ,

$$10 \times 13 = \alpha^9 \times \alpha^{13} = \alpha^{(9+13) \bmod 15} = \alpha^{(22) \bmod 15} = \alpha^7.$$

Again by inspection using Table 1 we find that:

$$\alpha^7 = 11$$

giving the result obtained by multiplying the polynomials.

A slight disadvantage of the logarithmic method is that field element 0 cannot be represented in index form. The method therefore has to sense the presence of zero values and force the result accordingly.

The logarithmic technique can also be used for division:

$$11 \div 10 = \alpha^7 \div \alpha^9 = \alpha^{(7-9) \bmod 15} = \alpha^{(-2) \bmod 15} = \alpha^{13} = 13.$$

However, division of two field elements is often accomplished by multiplying by the inverse of the divisor. The inverse of a field element is defined as the element value that when multiplied by the field element produces a value of 1 (= α^0). It is therefore possible to tabulate the inverse values of the field elements using Table 1.

5.3 Constructing a Reed-Solomon code

The values of the message and parity symbols of a Reed-Solomon code are elements of a Galois field. Thus for a code based on *m*-bit symbols, the Galois field has 2^m elements.

5.3.1 The code generator polynomial:

An (*n*, *k*) Reed-Solomon code is constructed by forming the code generator polynomial $g(x)$, consisting of n-k=2t factors, the roots of which are consecutive elements of the Galois field. Choosing consecutive elements ensures that the distance properties of the code are maximised. Thus the code generator polynomial takes the form:

$$g(x) = (x+\alpha^b)(x+\alpha^{b+1}).....(x+\alpha^{b+2t-1}) \qquad .....(6)$$

It should be noted that this expression is often quoted in the literature with subtle variations to catch the unwary. For example, the factors are often written as $(x - \alpha^i)$, which emphasises that $g(x) = 0$ when $x = \alpha^i$ and those familiar with Galois fields realise that $-\alpha^i$ is exactly the same as α^i. Also, some choose roots starting with α^0 (b=0 in equation 6), while many others start with α, the primitive element (b=1 in equation 6). While each is valid, it results in a completely different code requiring changes in both the coder and decoder operation. If the chosen value of b is near 2^m-1, then some of the roots may reach or exceed α^{2^m-1}. In this case the index values modulo 2^m-1 can be substituted. Small reductions in the complexity of hardware implementations can result by choosing b=0, but this is not significant.

5.3.2 Worked example based on a (15,11) code

Specific examples are very helpful for obtaining a full understanding of the processes involved in Reed-Solomon coding and decoding. However, the (255, 239) code used in DVB-T is too unwieldy to be used for an example. In particular, the message length of several hundred symbols leads to polynomials with several hundred terms! In view of this, the much simpler (15, 11) code will be used to go through the full process of coding and decoding using methods that can be extended generally to other Reed-Solomon codes.

For a (15, 11) code, the block length is 15 symbols, 11 of which are information symbols and the remaining 4 are parity words. Because t=2, the code can correct errors in up to 2 symbols in a block. Substituting for n in:

$$n = 2^m - 1$$

gives the value of m as 4, so each symbol consists of a 4-bit word and the code is based on the Galois field with 16 elements. The example will use the field generator polynomial of

equation(4), so that the arithmetic for the code will be based on the Galois field shown in Table 1.

The code generator polynomial for correcting up to 2 error words requires 4 consecutive elements of the field as roots, so we can choose:

$$g(x) = (x + \alpha^0)(x + \alpha^1)(x + \alpha^2)(x + \alpha^3)$$

$$= (x + 1)(x + 2)(x + 4)(x + 8)$$

using the index and decimal short-hand forms, respectively.

This expression has to be multiplied out to produce a polynomial in powers of *x*, which is done by multiplying the factors and adding together terms of the same order. As multiplication is easier in index form and addition is easier in polynomial form, the calculation involves repeatedly converting from one format to the other using Table 1. This is best done with a computer program for all but the simplest codes. Alternatively, we can take it factor by factor and use the look-up tables of Appendix 8.3 to give:

$$g(x) = (x + 1)(x + 2)(x + 4)(x + 8)$$

$$= (x^2 + 3x + 2)(x + 4)(x + 8)$$

$$= (x^3 + 7x^2 + 14x + 8)(x + 8)$$

$$= x^4 + 15x^3 + 3x^2 + x + 12 \quad \text{..... (7).}$$

This can also be expressed as:

$$g(x) = \alpha^0 x^4 + \alpha^{12} x^3 + \alpha^4 x^2 + \alpha^0 x + \alpha^6$$

with the polynomial coefficients in index form.

CHAPTER 6

REED-SOLOMON ENCODING

The Galois field theory of Section 2 provides the grounding to the processes of Reed-Solomon encoding and decoding described in this and the following sections. In particular, the arithmetic processes on which hardware implementations are based rely heavily on the preceding theory.

6.1The Encoding Process

6.1.1 The message polynomial

The k information symbols that form the message to be encoded as one block can be represented by a polynomial $M(x)$ of order k-1, so that:

$$M(x) = M_{k-1}x^{k-1} + ... + M_1x + M_0$$

Where each of the coefficients M_{k-1}, ..., M_1, M_0 is an m-bit message symbol, that is, an element of GF(2^m). M_{k-1} is the first symbol of the message.

6.1.2 Forming the code word

To encode the message, the message polynomial is first multiplied by x^{n-k} and the result divided by the generator polynomial, $g(x)$. Division by $g(x)$ produces a quotient $q(x)$ and a remainder $r(x)$, where $r(x)$ is of degree up to n-k-1. Thus:

$$\frac{M(X)\times x^{n-k}}{g(x)} = \mathrm{q(x)} + \frac{r(x)}{g(x)} \qquad \text{.....(8)}$$

Having produced $r(x)$ by division, the transmitted code word $T(x)$ can then be formed by combining
$M(x)$ and $r(x)$ as follows:

$$T(x) = M(x) \times x^{n-k} + r(x)$$

$$= M_{k-1}x^{n-1} + ... + M_0x^{n-k} + r_{n-k-1}x^{n-k-1} + ... + r_0$$

which shows that the code word is produced in the required systematic form.

6.2 Encoding example

We can now choose a message consisting of eleven 4-bit symbols for our (15, 11) code, for example, the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 which we wish to encode. These values are

represented by a message polynomial:

$x^{10} + 2x^9 + 3x^8 + 4x^7 + 5x^6 + 6x^5 + 7x^4 + 8x^3 + 9x^2 + 10x + 11$ ..... (10).

The message polynomial is then multiplied by x^4 to give:

$x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 6x^9 + 7x^8 + 8x^7 + 9x^6 + 10x^5 + 11x^4$

to allow space for the four parity symbols. This polynomial is then divided by the code generator polynomial, equation (7), to produce the four parity symbols as a remainder. This can be accomplished in columns as a long division process as shown before, except that in this case, the coefficients of the polynomials are field elements of GF(16) instead of binary values, so the process is more complicated.

6.2.1 Polynomial Division

At each step the generator polynomial is multiplied by a factor, shown at the left-hand column, to make the most significant term the same as that of the remainder from the previous step. When subtracted (added), the most significant term disappears and a new remainder is formed. The 11 steps of the division process are as follows:

	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
	1	2	3	4	5	6	7	8	9	10	11	0	0	0	0
$\times x^{10}$	1	15	3	1	12										
		13	0	5	9	6									
$\times 13x^9$		13	7	4	13	3									
			7	1	4	5	7								
$\times 7x^8$			7	11	9	7	2								
				10	13	2	5	8							
$\times 10x^7$				10	12	13	10	1							
					1	15	15	9	9						
$\times 1x^6$					1	15	3	1	12						
						0	12	8	5	10					
$\times 0x^5$						0	0	0	0	0					
							12	8	5	10	11				
$\times 12x^4$							12	8	7	12	15				
								0	2	6	4	0			
$\times 0x^3$								0	0	0	0	0			
									2	6	4	0	0		
$\times 2x^2$									2	13	6	2	11		
										11	2	2	11	0	
$\times 11x$										11	3	14	11	13	
											1	12	0	13	0
$\times 1$											1	15	3	1	12
												3	3	12	12

$r(x) = 3x^3 + 3x^2 + 12x + 12$.

The quotient, $q(x)$, produced as the left-hand column of multiplying values is not required and is discarded.

The encoded message polynomial $T(x)$ is then:
$x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 6x^9 + 7x^8 + 8x^7$

$+ 9x^6 + 10x^5 + 11x^4 + 3x^3 + 3x^2 + 12x + 12$ ..... (11)

or, written more simply:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 3, 3, 12, 12.

6.3 Encoder Hardware:

6.3.1 General arrangement

The pipelined calculation shown in section 3.2.2 is performed using the conventional encoder circuit shown in Figure 2. All the data paths shown provide for 4-bit values.

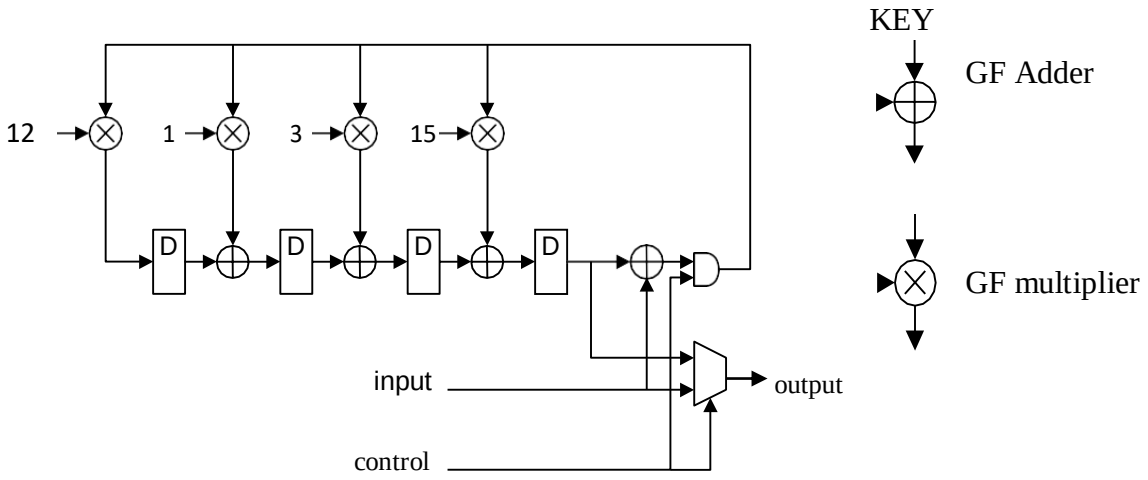

Figure 6.1- A (15, 11) Reed-Solomon encoder

During the message input period, the selector passes the input values directly to the output and the AND gate is enabled. After the eleven calculation steps shown above have been completed (in eleven consecutive clock periods) the remainder is contained in the D-type registers. The control waveform then changes so that the AND gate prevents further feedback to the multipliers and the four remainder symbol values are clocked out of the registers and routed to the output by the selector.

6.3.2 Galois field adder

The adders of Figure 2 perform bit-by-bit addition modulo-2 of 4-bit numbers and each consists of four 2-input exclusive-OR gates. The multipliers, however, can be implemented in a number of different ways.

6.3.3 Galois field constant multipliers

Since each of these units is multiplying by a constant value, one approach would be to use a full multiplier and to fix one input. Although a full multiplier is significantly more complicated, with an FPGA design, the logic synthesis process would strip out at least some of the unused circuitry.

More will be said of full multipliers in Section 5.3.4.1. The other two approaches that come to mind are either to work out the equivalent logic circuit or to specify it as a look-up table, using a read-only memory.

Dedicated logic constant multipliers

For the logic circuit approach, we can work out the required functionality by using a general polynomial representation of the input signal $a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$. This is then multiplied by the polynomials represented by the values 15, 3, 1 and 12 from Table 1. This involves producing a shifted version of the input for each non-zero coefficient of the multiplying polynomial. Where the shifted versions produce values in the α^6, α^5 or α^4 columns, the 4-bit equivalents (from Table 1) are substituted. The bit values in each of the α^3, α^2, α^1 and α^0 columns are then added to give the required input bit contributions for each output bit.

For example, for multiplication by 15 (= $\alpha^3 + \alpha^2 + \alpha + 1$):

	α^6	α^5	α^4	α^3	α^2	α^1	α^0
$\times\alpha^3$	a_3	a_2	a_1	a_0	0	0	0
$\times\alpha^2$		a_3	a_2	a_1	a_0	0	0
$\times\alpha$			a_3	a_2	a_1	a_0	0
$\times 1$				a_3	a_2	a_1	a_0
	a_3	a_2+a_3	$a_1+a_2+a_3$	0	0	$a_1+a_2+a_3$	$a_1+a_2+a_3$
				0	a_2+a_3	a_2+a_3	0
				a_3	a_3	0	0
				$a_0+a_1+a_2$	a_0+a_1	a_0	$a_0+a_1+a_2+a_3$

The input bits contributing to a particular output bit are identified by the summation at the foot of each column. Similar calculations can be performed for multiplication by 3 (= $\alpha + 1$), 1 (=1) and 12 (= $\alpha^3 + \alpha^2$) and give the results:

	α^3	α^2	α^1	α^0
$\times 3$	a_2+a_3	a_1+a_2	$a_0+a_1+a_3$	a_0+a_3
$\times 1$	a_3	a_2	a_1	a_0
$\times 12$	$a_0+a_1+a_3$	a_0+a_2	a_1+a_3	a_1+a_2

As the additions are modulo 2, these are implemented with exclusive-OR gates as shown in Figure 6.2.

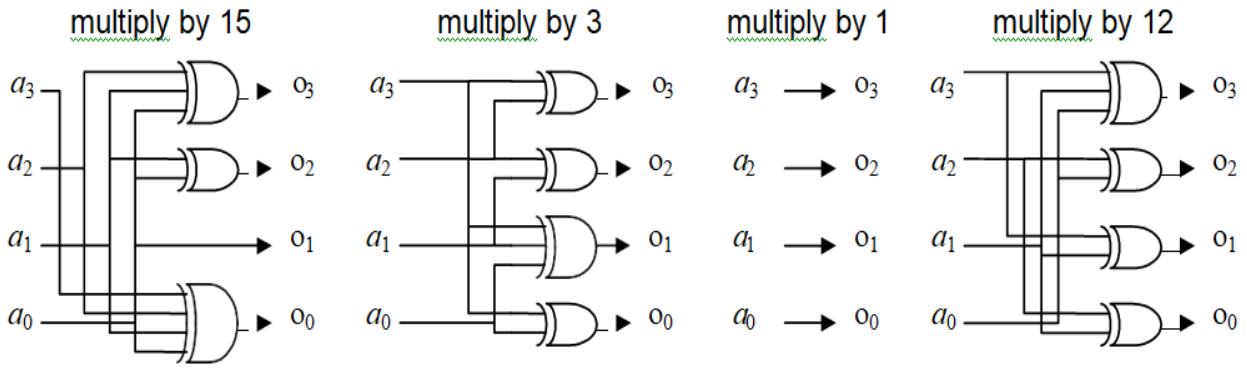

Figure 6.2 - Multipliers for the circuit of Figure 4.1

Look-up table constant multipliers:

Alternatively, each multiplier can be implemented as a look-up table with $2^m = 16$ entries. The entry values can be obtained by cyclically shifting the non-zero elements from Table 1 according to the index of the multiplication factor. This is because the multiplying index is added to the index of the input, modulo 15, thus shifting the results according to the multiplying value. However, the binary values of the polynomial coefficients of the input need to be arranged in ascending order to match the binary addressing of the look-up table memory. When this is done the values shown in Table 3 are produced.

input		$\times 15 = \alpha^{12}$	$\times 3 = \alpha^{4}$	$\times 1 = \alpha^{0}$	$\times 12 = \alpha^{6}$
index form	decimal form	decimal form	decimal form	decimal form	decimal form
0	0	0	0	0	0
α^0	1	15	3	1	12
α^1	2	13	6	2	11
α^4	3	2	5	3	7
α^2	4	9	12	4	5
α^8	5	6	15	5	9
α^5	6	4	10	6	14
α^{10}	7	11	9	7	2
α^3	8	1	11	8	10
α^{14}	9	14	8	9	6
α^9	10	12	13	10	1
α^7	11	3	14	11	13
α^6	12	8	7	12	15
α^{13}	13	7	4	13	3
α^{11}	14	5	1	14	4
α^{12}	15	10	2	15	8

Table 6.1 - Look-up tables for the fixed multipliers of Figure 6.1

CHAPTER 7

REED-SOLOMON DECODING

Whereas the previous section has dealt with the underlying theory and, in some cases, identified several alternative approaches to some processes, this section will describe a specific approach to decoding hardware based around the Euclidean algorithm.

7.1 Main units of a Reed-Solomon decoder:

The arrangement of the main units of a Reed-Solomon decoder reflects, for the most part, the processes of the previous Section.

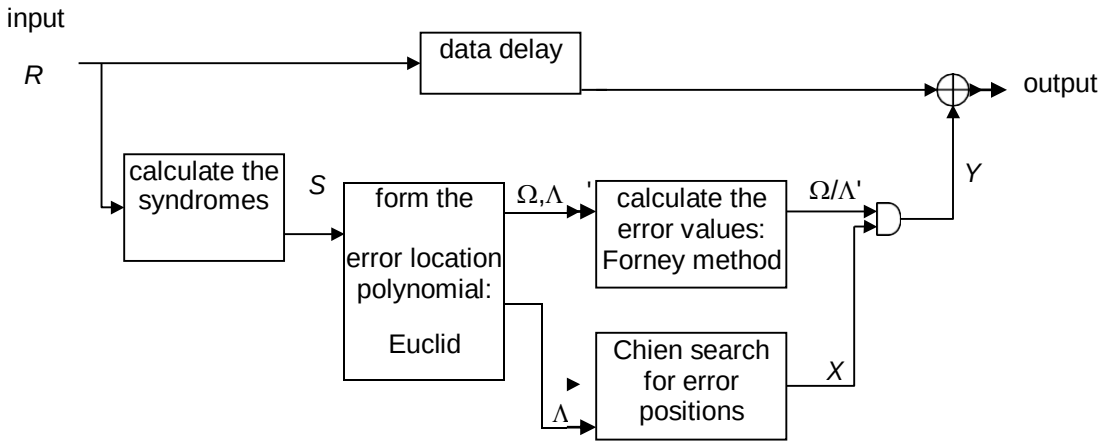

Figure 7.1 - Main processes of a Reed-Solomon decoder

Thus, in Figure 4, the first process is to calculate the syndrome values from the incoming code word. These are then used to find the coefficients of the error locator polynomial Λ_1 Λ_v and the error value polynomial Ω_0 Ω_{v-1} using the Euclidean algorithm. The error locations are identified by the Chien search and the error values are calculated using Forney's method. As these calculations involve all the symbols of the received code word, it is necessary to store the message until the results of the calculation are available. Then, to correct the errors, each error value is added (modulo 2) to the symbol at the appropriate location in the received code word.

7.2 Hardware of Syndrome calculation:

The hardware arrangement used for syndrome calculation, shown in Figure 5, can be interpreted either as a pipelined polynomial division or as an implementation of Horner's method.

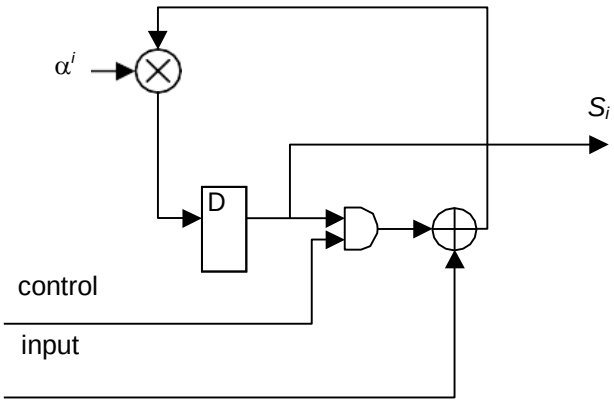

Figure 7.2 - Forming a syndrome

In the case of polynomial division, the process is basically the same as that described for encoding in section 3.2.2 (and shown in Figure 2), except much simpler because the degree of the divisor polynomial is one. Thus there is only one feedback term with one multiplier, multiplying by α^i, and only one register. As before, the input values are added to the output of the register and all the data paths are m-bit signals. The only difference in this case is that the AND-gate is used to prevent the contents of the register contributing at the start of the code word, which was achieved in Figure 2 by clearing the registers at the start of each block. Alternatively, this circuit can be seen as a direct implementation of Horner's method, in which the incoming symbol value is added to the contents of the register before being multiplied by α^i and the result returned to the register.

Clearly, all n symbols of the code word have to be accumulated before the syndrome value is produced. Also, $2t$ circuits of the form of Figure 5 are required, one for each value of α^i, each corresponding to a root of the generator polynomial. The Galois field adders and fixed multipliers can be implemented using the techniques described in Sections 6.3.3.

7.4 Full Multipliers:

The Galois field multipliers described up to this point have involved multiplication by a constant, whereas those of Figure 6 are full multipliers. Full multipliers can be implemented by similar techniques to those described in Section 3.3.3, either as dedicated logic multipliers or as look-up tables, although with 2^{2m} locations, the latter technique rapidly becomes inefficient as the value of m increases. It is also possible to use look-up tables with 2^m locations to convert to logarithms, which can then be added modulo 2^m - 1 and the result converted back with an inverse look-up table. The need to sense zero inputs and produce a modulo 2^m - 1 result generally makes this technique more complicated than the shift-and-add approach.

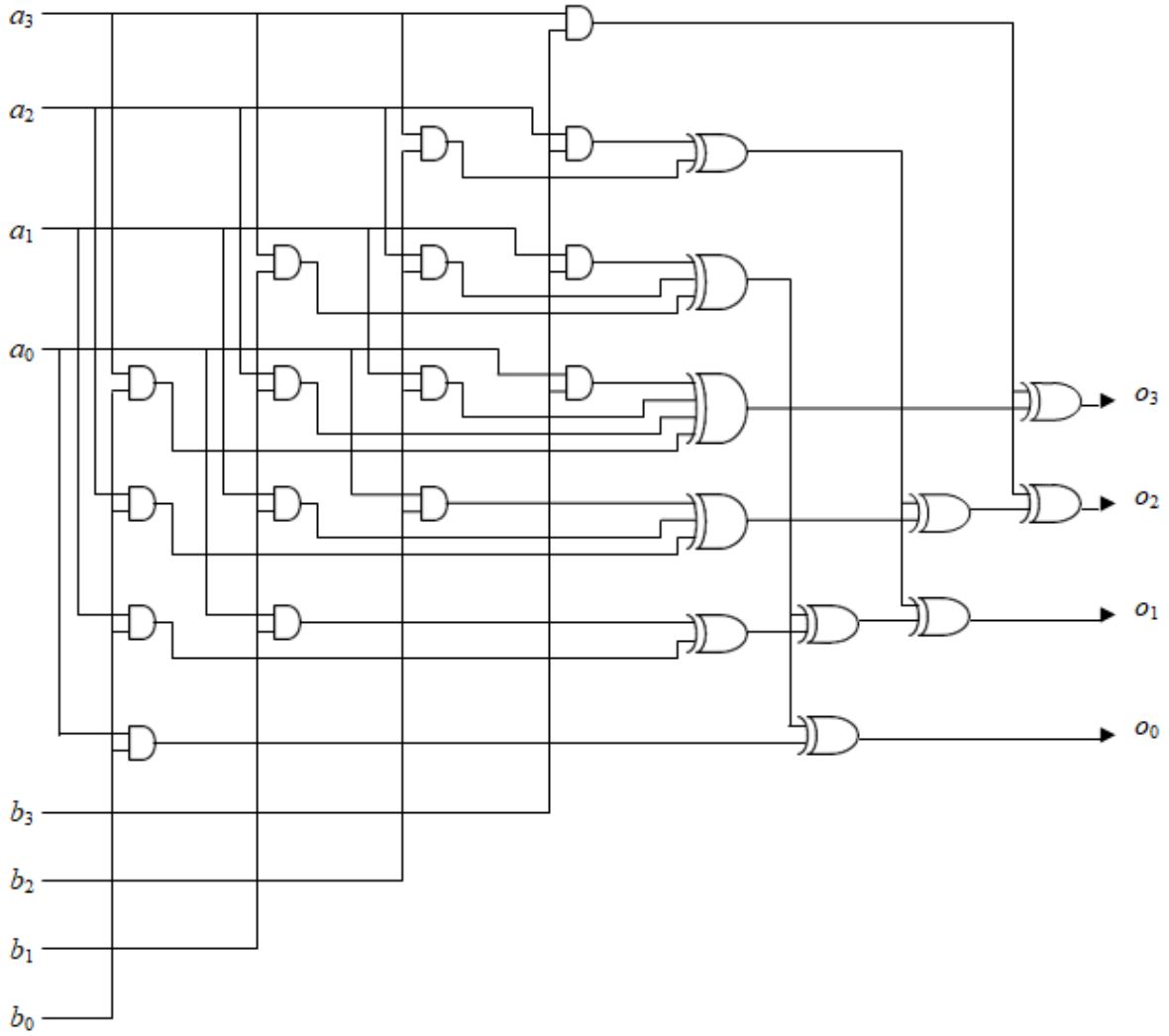

Figure 7.4 - A full multiplier for GF(16)

Figure 7 shows the arrangement of a 4-bit by 4-bit shift-and-add multiplier, drawn to emphasise the three underlying processes. First the array of AND gates generates the set of shifted product terms, producing seven levels of significance. Next the first column of exclusive-OR gates sums (modulo2) the products at each level. Finally, the three upper levels beyond the range of field values are converted to fall within the field, using the relationships of Table 1, and the contributions added by the three pairs of exclusive-OR gates.

7.5 Division or Inversion

Having designed a multiplier, then it is probably most straightforward to implement Galois field division using a look-up table with 2^m locations to generate the inverse and then to multiply. The inverses are easily calculated as shown in Table 5 using field elements in index form. This shows the element values in ascending order to correspond with the addressing of the look-up table.

input (decimal)	input (index)	inverse (index)	inverse (decimal)
0	0	0	0
1	α^0	α^0	1
2	α^1	$\alpha^{-1} = \alpha^{14}$	9
3	α^4	$\alpha^{-4} = \alpha^{11}$	14
4	α^2	$\alpha^{-2} = \alpha^{13}$	13
5	α^8	$\alpha^{-8} = \alpha^7$	11
6	α^5	$\alpha^{-5} = \alpha^{10}$	7
7	α^{10}	$\alpha^{-10} = \alpha^5$	6
8	α^3	$\alpha^{-3} = \alpha^{12}$	15
9	α^{14}	$\alpha^{-14} = \alpha^1$	2
10	α^9	$\alpha^{-9} = \alpha^6$	12
11	α^7	$\alpha^{-7} = \alpha^8$	5
12	α^6	$\alpha^{-6} = \alpha^9$	10
13	α^{13}	$\alpha^{-13} = \alpha^2$	4
14	α^{11}	$\alpha^{-11} = \alpha^4$	3
15	α^{12}	$\alpha^{-12} = \alpha^3$	8

Table 7.1 - Look-up table for inverse values in GF(16)

The table includes 0 as the Galois field inverse of 0.

****

CHAPTER 8

SIMULATIONS CODES

8.1 MATLAB Program

8.1.1 Galois field elements table generation for GF(16):

Logic Explanation:

Step1 – Initialize the finite field of GF(2^m) = 2^4.

Step2 – Taken primitive element a^0 = 0 0 0 1

a^0 left shift by1 bit = a^1= 0 0 1 0

a^1 left shift by1 bit = a^2= 0 1 0 0

a^2 left shift by1 bit = a^3 =1 0 0 0

Step3 – Again left shift a^3 by 1 bit, overflow of finite field at a^4 , then

a^4 = a+1 taken as generating polynomial.

a^4 = 0 0 1 1

Step4– Taken primitive element a^4 = 0 0 11

a^4 left shift by1 bit = a^5,

a^5 left shift by1 bit = a^6,

a^6left shift by1 bit = a^7

Step5 – At a^7 ,overflow occurred , thus to reduce it a^7= XOR (a^6, a^4)

Step6- By using this methodology we have generated GF(16) element table.

MATLAB Command:

```
clc;
x0=[0 0 0 1];
x1=[0 0 1 0];
x2=[0 1 0 0];
x3=[1 0 0 0];
x4=[0 0 1 1];
x5=xor(x2,x1);
x6=xor(x3,x2);
x7a=xor(x3,x2);
```

```
x7=xor(x7a,x0);
x8=xor(x2,x0);
x9=xor(x3,x1);
a=xor(x2,x1);
x10=xor(a,x0);
xb=xor(x3,x2);
x11=xor(xb,x1);
xc=xor(x3,x2);
xd=xor(xc,x1);
x12=xor(xd,x0);
xe=xor(x3,x2);
x13=xor(xe,x0);
x14=xor(x3,x0);
A=[x0;x1;x2;x3;x4;x5;x6;x7;x8;x9;x10;x11;x12;x13;x14]
```

MATLAB Output:

```
A =

    0    0    0    1
    0    0    1    0
    0    1    0    0
    1    0    0    0
    0    0    1    1
    0    1    1    0
    1    1    0    0
    1    1    0    1
    0    1    0    1
    1    0    1    0
    0    1    1    1
    1    1    1    0
    1    1    1    1
    1    1    0    1
    1    0    0    1

>>
```

8.1.2 Multipliers:

Multiplication in a finite field is multiplication modulo an irreducible reducing polynomial used to define the finite field. (I.e., it is multiplication followed by division using the reducing polynomial as the divisor—the remainder is the product.) The symbol "•" may be used to denote multiplication in a finite field.

Logic Explanation:

Step-1 Initially the generative polynomial has taken $g(x)= x^4+x+1=0$ which has to be multiply with coefficient 1,3,12 and 15 consecutively activation needed each of them once of at a time while running the code.
Step-2 Evaluation of the row and column for limiting in the finite field.
Step-3 Reductions of each rows while existing from finite field rearrange with the help of generative polynomial.
Step-4 The gfconv function multiplies polynomials over a Galois field. (To multiply elements of a Galois field, use gfmul instead.) Algebraically, multiplying polynomials over a Galois field is equivalent to convolving vectors containing the polynomials' coefficients, where the convolution operation uses arithmetic over the same Galois field.

Step-4 c = gfconv(a,b) multiplies two GF(2) polynomials, a and b, which can be either polynomial character vectors or numeric vectors. The polynomial degree of the resulting GF(2) polynomial c equals the degree of a plus the degree of b.

Step- 5 c = gfconv(a,b,p) multiplies two GF(p) polynomials, where p is a prime number. a, b, and c are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and p-1.

Step-6 c = gfconv(a,b,field) multiplies two GF(p^m) polynomials, where p is a prime number and m is a positive integer. a, b, and c are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). field is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element.

Step-7 Search for all symbols in row. And check the occurrence of a symbol in a row.

Step-8 Finally out_row will contain all the symbols in a row and note the simulation output.

MATLAB Command:

```
clc;
clear all;
disp('Program Start');
% Field Gen Poly: x^4+x+1=0
% Select the multliplier's Coefficient Here:
% mult= [0 0 0 1]; % coeff =1
% mult= [0 0 1 1]; % coeff =3
% mult= [1 0 1 0]; % coeff =12
  mult= [1 1 1 1]; % coeff =15
coef= bin2dec(num2str(mult))
```

```
syms d3 d2 d1 d0;
dt_org = [d3 d2 d1 d0];
dt_org_ln7 = [ 0 0 0 d3 d2 d1 d0];
% Eval Multiplication Rows
for kk=1:1:4
   if mult(5-kk) == 1
      row_ln7(kk,:)= dt_org_ln7;
   else
      row_ln7(kk,:)= zeros(1,7,'sym');
   end
   ['row_ln7 no: ' num2str(kk)];
   row_ln7(kk,:);
   dt_org_ln7= [dt_org_ln7 0];
   dt_org_ln7= dt_org_ln7(2:8);
end
row_ln7(:,:);

kk=1;
% 1st Row Reduction
if row_ln7(1,:) == zeros(1,7)
else
row_ln4(kk,:) = row_ln7(1,4:7);
end

% 2nd Row Reduction
if row_ln7(2,:)== zeros(1,7)
else
   kk=kk+1;
   row_ln4(kk,:)= row_ln7(2,4:7);
   m=row_ln7(2,3);
   kk=kk+1;
   row_ln4(kk,:)= [0 0 m m];
end

% 3rd Row Reduction
if row_ln7(3,:)== zeros(1,7,'sym');
else
   kk=kk+1;
   row_ln4(kk,:)= row_ln7(3,4:7);
   m=row_ln7(3,3);
   kk=kk+1;
   row_ln4(kk,:)= [0 0 m m];
   m=row_ln7(3,2);
   kk=kk+1;
   row_ln4(kk,:)= [0 m m 0];
end

% 4th Row Reduction
if row_ln7(4,:)== zeros(1,7,'sym');
else
   kk=kk+1;
```

```
    row_ln4(kk,:)= row_ln7(4,4:7);
    m=row_ln7(4,3);
    kk=kk+1;
    row_ln4(kk,:)= [0 0 m m];
    m=row_ln7(4,2);
    kk=kk+1;
    row_ln4(kk,:)= [0 m m 0];
    m=row_ln7(4,1);
    kk=kk+1;
    row_ln4(kk,:)= [m m 0 0];
end

% Converting coloums into rows
row_ln4= row_ln4.';
% Search for all symbols in all rows
sym_set= [d3 d2 d1 d0];
out=zeros(4,4,'sym');

for ii=1:1:4
y=row_ln4(5-ii,:);
% Search for all symbols in a rows
out_rw=[];
for jj= 1:1:4
    cnt=0;
   % Search for a symbol in a row
    for kk= 1:1:length(y)
        if y(kk)== sym_set(jj);
            cnt=cnt+1;
        end
    end
    % Check the occurance of a symbol in a row
    z= rem(cnt,2);
    if z==1
    % Finally out_row will contain all the symbols in a row
        out_rw= [out_rw sym_set(jj)];
    end
end
out(ii,1:length(out_rw))= out_rw;
end
for kk=1:1:4
   w= ['out(', num2str(kk-1), ')= xor of:'];
   disp(w);
   out(kk,:)
end
disp('Program End');
```

MATLAB Output:

```
%for multiplier coefficient = 1
Program Start

coef =

    1

out(0)= xor of:

ans =

[ d0, 0, 0, 0]

out(1)= xor of:

ans =

[ d1, 0, 0, 0]

out(2)= xor of:

ans =

[ d2, 0, 0, 0]

out(3)= xor of:
ans =

[ d3, 0, 0, 0]

Program End
>>
```

```
%for multiplier coefficient = 3

program Start

coef =

     3

out(0)= xor of:

ans =

[ d3, d0, 0, 0]

out(1)= xor of:

ans =

[ d3, d1, d0, 0]

out(2)= xor of:

ans =

[ d2, d1, 0, 0]

out(3)= xor of:

ans =

[ d3, d2, 0, 0]

Program End
>>
```

```
%for multiplier coefficient = 12

program Start

coef =

    12

out(0)= xor of:

ans =

[ d2, d1, 0, 0]

out(1)= xor of:

ans =

[ d3, d1, 0, 0]

out(2)= xor of:

ans =

[ d2, d0, 0, 0]

out(3)= xor of:

ans =

[ d3, d1, d0, 0]

Program End
>>
```

```
%for multiplier coefficient = 15

    Program Start

    coef =

       15

    out(0)= xor of:

    ans =

    [ d3, d2, d1, d0]

    out(1)= xor of:

    ans =

    [ d0, 0, 0, 0]

    out(2)= xor of:

    ans =

    [ d1, d0, 0, 0]

    out(3)= xor of:

    ans =

    [ d2, d1, d0, 0]

    Program End
    >>
```

8.2 Verilog Module & Test fixture

Verilog Module of RS Encoder:

Logic Explanation:

```
// Specifications of this Reed Solomon Encoder:
// (n,k)= (15,11) : No. of Incoming Message symbols= 11,
// No. of Parity symbols generated by this encoder= (15-11)= 4,
// No. of Total Message symbol + Parity symbol= 15;
// Code Generator Polynomial Specification:
// g(x)= (x+a^0)(x+a^1)(x+a^2)(x+a^3)=(x+1)(x+2)(x+4)(x+8)= x^4+15x^3+3x^2+x+12 ;

`timescale 100ns/1ns
module Reed_Solomon_Encoder
(input  clk,  // clock
 input  rst,  // reset synchronus
 input  in_dt_vld, // incoming data valid
 input  rx_start_sym_set, // Start of a new message set of 11 Received
 input [3:0] msg_in, // incoming message symbol
 output reg [3:0] msg_par_out_D,  // outgoing message symbol + parity symbols
 output reg out_dt_vld_D, // outgoing data valid
 output reg tx_start_sym_set, // Start of a new message set of 11 Transmistted
 output        busy );   // parity symbol generation stage

wire [3:0] gf_mult_12_out, gf_mult_1_out, gf_mult_3_out, gf_mult_15_out;
reg  [3:0] d_out_1, d_out_2, d_out_3, d_out_4;
wire [3:0] xor_4_out;
wire [3:0] gf_mult_in;
wire               cntrl_in ;
reg  [3:0] counter;
wire                    par_sym_gen;
wire [3:0]    msg_par_out;
// reg                  out_dt_vld;
wire                    out_dt_vld;
wire divider_enb;

// gf multiplier circuits

// gf multiplier with coeff 12
assign gf_mult_12_out[0]= gf_mult_in[1] ^ gf_mult_in[2];
assign gf_mult_12_out[1]= gf_mult_in[1] ^ gf_mult_in[3];
assign gf_mult_12_out[2]= gf_mult_in[0] ^ gf_mult_in[2];
assign gf_mult_12_out[3]= gf_mult_in[0] ^ gf_mult_in[1]^ gf_mult_in[2];


// gf multiplier with coeff 1
assign gf_mult_1_out[0]= gf_mult_in[0];
```

```
assign gf_mult_1_out[1]= gf_mult_in[1];
assign gf_mult_1_out[2]= gf_mult_in[2];
assign gf_mult_1_out[3]= gf_mult_in[3];

// gf multiplier with coeff 3
assign gf_mult_3_out[0]= gf_mult_in[0] ^ gf_mult_in[3] ;
assign gf_mult_3_out[1]= gf_mult_in[0] ^ gf_mult_in[1]^ gf_mult_in[3];
assign gf_mult_3_out[2]= gf_mult_in[1] ^ gf_mult_in[2];
assign gf_mult_3_out[3]= gf_mult_in[2] ^ gf_mult_in[3];


// gf multiplier with coeff 15
assign gf_mult_15_out[0]= gf_mult_in[0] ^ gf_mult_in[1]^ gf_mult_in[2] ^ gf_mult_in[3];
assign gf_mult_15_out[1]= gf_mult_in[0];
assign gf_mult_15_out[2]= gf_mult_in[0] ^ gf_mult_in[1];
assign gf_mult_15_out[3]= gf_mult_in[0] ^ gf_mult_in[1]^ gf_mult_in[2];

// inputs: gf_mult_12_out, gf_mult_1_out, gf_mult_3_out, gf_mult_15_out
always@(posedge clk)
begin
if(rst == 1'b1)
begin
     d_out_1 <= 4'b0;
     d_out_2 <= 4'b0;
     d_out_3 <= 4'b0;
     d_out_4 <= 4'b0;
end
else if(rx_start_sym_set == 1'b1)
begin
     d_out_1 <= 4'b0;
     d_out_2 <= 4'b0;
     d_out_3 <= 4'b0;
     d_out_4 <= 4'b0;
end
else if(divider_enb == 1'b1)
begin
     d_out_1 <= gf_mult_12_out;
     d_out_2 <= d_out_1 ^ gf_mult_1_out;
     d_out_3 <= d_out_2 ^ gf_mult_3_out;
     d_out_4 <= d_out_3 ^ gf_mult_15_out;
end
end

assign divider_enb = in_dt_vld | par_sym_gen;
assign xor_4_out = d_out_4 ^ msg_in;
assign gf_mult_in = xor_4_out & {4{cntrl_in}};

assign msg_par_out= (cntrl_in == 1'b1)? msg_in : d_out_4;


// counter to count number of message symbols received
```

```
// and to generate states for parity symbol generation

always@(posedge clk)
begin
if (rst == 1'b1)
      counter <= 4'b0;
else if (rx_start_sym_set == 1'b1)
      counter <= 4'b0;
// rollback to reset state
else if (counter == 4'd14)
      counter <= 4'd0;
// state 11 to 14 required to generate parity symbols
else if (counter >= 4'd11)
      counter <= counter + 1'b1;
// counting the number of received message symbols
else if(in_dt_vld == 1'b1)
      counter <= counter + 1'b1;
end

assign par_sym_gen = (counter >= 4'd11) ? 1'b1: 1'b0;
assign busy = par_sym_gen;
assign cntrl_in = ~par_sym_gen;

assign         out_dt_vld = in_dt_vld | par_sym_gen;

always@(posedge clk)
begin
if (rst == 1'b1)
begin
      msg_par_out_D <= 4'd0;
      out_dt_vld_D  <= 1'b0;
      tx_start_sym_set <= 1'b0;
end
else
begin
      msg_par_out_D <= msg_par_out;
      out_dt_vld_D  <= out_dt_vld;
      tx_start_sym_set <=  rx_start_sym_set;
end
end
endmodule
```

Verlilog Test fixture of RS Encoder:

```
`timescale 100ns/1ns
module TB_Reed_Solomon_Encoder;
// TB Design Interfacing Signals
reg tb_clk;
reg tb_rst;
```

```
reg tb_in_dt_vld;
reg tb_rx_start_sym_set;
reg [3:0] tb_msg_in;

wire [3:0] tb_msg_par_out_D;
wire tb_out_dt_vld_D;
wire tb_tx_start_sym_set;
wire tb_busy;

// TB internal signals
reg [3:0] msg_ary [10:0]; // Each Signal =4bit; Array Size=11;
reg [7:0] ii;

Reed_Solomon_Encoder RSE_1
( .clk (tb_clk ) ,
  .rst (tb_rst ) ,
  .in_dt_vld (tb_in_dt_vld ),
  .rx_start_sym_set (tb_rx_start_sym_set ) ,
  .msg_in (tb_msg_in ) ,
  .msg_par_out_D (tb_msg_par_out_D ) ,
  .out_dt_vld_D (tb_out_dt_vld_D ) ,
  .tx_start_sym_set (tb_tx_start_sym_set ) ,
  .busy (tb_busy)  );

// Clock Signal Genrtn
always
#5 tb_clk= ~tb_clk;

initial
begin

tb_clk= 1'b0;
tb_rst= 1'b1;
tb_in_dt_vld = 1'b0;
tb_rx_start_sym_set = 1'b0;
tb_msg_in = 4'd0;

//Apply & Release Reset
#3 tb_rst= 1'b1;
@(posedge tb_clk);
#3 tb_rst= 1'b0;

// generate 1st set set of 11 message symbols
for (ii=0; ii<11; ii=ii+1)
    msg_ary[ii]  = $random;
@(posedge tb_clk);
```

```
#3;
tb_rx_start_sym_set = 1'b1;
@(posedge tb_clk);
#3;
tb_rx_start_sym_set = 1'b0;
ii= 0;
repeat(11)
begin
@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b1;
tb_msg_in = msg_ary[ii];
ii= ii+1;
end

@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b0;
@(posedge tb_clk);
#1;
wait(tb_busy == 1'b0);

// generate 2nd set set of 11 message symbols
for (ii=0; ii<11; ii=ii+1)
    msg_ary[ii]  = $random;
@(posedge tb_clk);
#3;
tb_rx_start_sym_set = 1'b1;
@(posedge tb_clk);
#3;
tb_rx_start_sym_set = 1'b0;
ii= 0;
repeat(11)
begin
@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b1;
tb_msg_in = msg_ary[ii];
ii= ii+1;
end
@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b0;
@(posedge tb_clk);
#1;
wait(tb_busy == 1'b0);
```

```
// generate 3rd set set of 11 message symbols
for (ii=0; ii<11; ii=ii+1)
    msg_ary[ii]  = $random;
@(posedge tb_clk);
#3;
tb_rx_start_sym_set = 1'b1;
@(posedge tb_clk);
#3;
tb_rx_start_sym_set = 1'b0;
ii= 0;
repeat(11)
begin
@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b1;
tb_msg_in = msg_ary[ii];
ii= ii+1;
end
@(posedge tb_clk);
#3;
tb_in_dt_vld = 1'b0;
//End of simulation
repeat(10)
@(posedge tb_clk);
$finish;
end // end of initial
endmodule
```

Verlilog Test fixture of RS Decoder:

```
`timescale 100ns / 1ns

module TB_Reed_Solomon_Decoder;

    // Inputs of Design
    reg clk;
    reg rst;
    reg in_dt_vld;
    reg tx_start_sym_set;
    reg [3:0] msg_par_in;

    // Outputs of Design
    wire syndrome_valid;
    wire [3:0] syndromes_0;
    wire [3:0] syndromes_1;
    wire [3:0] syndromes_2;
```

```
wire [3:0] syndromes_3;
wire error_detected;

/*
// Design Instantiation
RTL_Reed_Solomon_Decoder RS_Decoder_1
(           .clk (clk),
            .rst (rst),
   .in_dt_vld (in_dt_vld ),
   .tx_start_sym_set (tx_start_sym_set),
   .msg_par_in (msg_par_in),
   .syndrome_valid (syndrome_valid),
   .syndromes_0 (syndromes_0),
   .syndromes_1 (syndromes_1),
   .syndromes_2 (syndromes_2),
   .syndromes_3 (syndromes_3),
   .error_detected (error_detected)
);
*/

// TB Internal Registers
reg [3:0] message_parity_set1 [14:0];
reg [3:0] message_parity_set2 [14:0];
reg [3:0] message_parity_set3 [14:0];
reg [3:0] message_parity_set1_err [14:0];
reg [3:0] message_parity_set2_err [14:0];
reg [3:0] message_parity_set3_err [14:0];
reg [4:0] ii;
reg [2:0] set_num;

// Clock Signal Genrtn
always
#5 clk= ~clk;

initial
begin
clk= 1'b0;
rst= 1'b0;
in_dt_vld= 1'b0;
tx_start_sym_set= 1'b0;
msg_par_in= 4'h0;

$readmemh("./RS_Message_Parity_Set_1.txt", message_parity_set1);
$readmemh("./RS_Message_Parity_Set_2.txt", message_parity_set2);
$readmemh("./RS_Message_Parity_Set_3.txt", message_parity_set3);

@(posedge clk);
#2;
rst=1;
@(posedge clk);
@(posedge clk);
```

```
#2;
rst=0;
@(posedge clk);
@(posedge clk);

// Start Sending 1st Error Free Symbol Set
#2;
set_num= 3'h1;
tx_start_sym_set = 1'b1;
@(posedge clk);
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set1[ii];
        @(posedge clk);
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
@(posedge clk);

// Start Sending 2nd Error Free Symbol Set
#2;
set_num= 3'h2;
tx_start_sym_set = 1'b1;
@(posedge clk);
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set2[ii];
        @(posedge clk);
        #2;
        in_dt_vld=1'b0;
        @(posedge clk);
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
@(posedge clk);

// Start Sending 3rd Error Free Symbol Set
#2;
set_num= 3'h3;
tx_start_sym_set = 1'b1;
@(posedge clk);
```

```
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set3[ii];
        @(posedge clk);
        #2;
        in_dt_vld=1'b0;
        @(posedge clk);
        @(posedge clk);
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
@(posedge clk);
#2;

// Generate errornous symbol set by introducing errors
for(ii=0;ii<15;ii=ii+1)
begin
        message_parity_set1_err[ii]= message_parity_set1[ii];
        message_parity_set2_err[ii]= message_parity_set2[ii];
        message_parity_set3_err[ii]= message_parity_set3[ii];
end

// introducing errors in certain symbols
message_parity_set1_err[2] = 4'h7;
message_parity_set1_err[7] = 4'h5;

message_parity_set2_err[4] = 4'h4;
message_parity_set2_err[14] = 4'h9;

message_parity_set3_err[11] = 4'ha;
message_parity_set3_err[12] = 4'hc;

// Start Sending 1st Error Containing Symbol Set
@(posedge clk);
#2;
set_num= 3'h4;
tx_start_sym_set = 1'b1;
@(posedge clk);
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set1_err[ii];
        @(posedge clk);
```

```
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
@(posedge clk);

// Start Sending 2nd Error Containing Symbol Set
@(posedge clk);
#2;
set_num= 3'h5;
tx_start_sym_set = 1'b1;
@(posedge clk);
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set2_err[ii];
        @(posedge clk);
        #2;
        in_dt_vld=1'b0;
        @(posedge clk);
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
@(posedge clk);

// Start Sending 3rd Error Containing Symbol Set
@(posedge clk);
#2;
set_num= 3'h6;
tx_start_sym_set = 1'b1;
@(posedge clk);
tx_start_sym_set = 1'b0;
@(posedge clk);
#2;
for(ii=0;ii<15;ii=ii+1)
begin
        in_dt_vld=1'b1;
        msg_par_in = message_parity_set3_err[ii];
        @(posedge clk);
        #2;
        in_dt_vld=1'b0;
        @(posedge clk);
        @(posedge clk);
        #2;
end
wait(syndrome_valid == 1'b1);
repeat(3)
```

```
        @(posedge clk);
        #2;
        $finish;
        end // end initial
    endmodule
```

8.2.3 RTL Schematic View

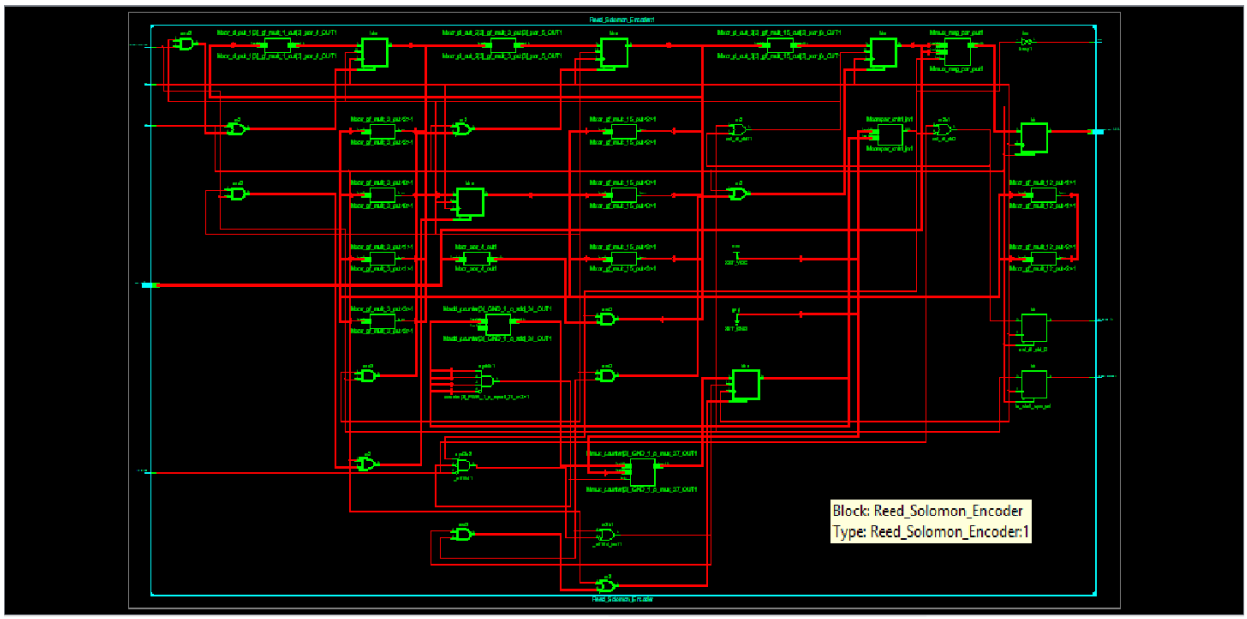

Figure 8.1 RTL Schematic view of RS code

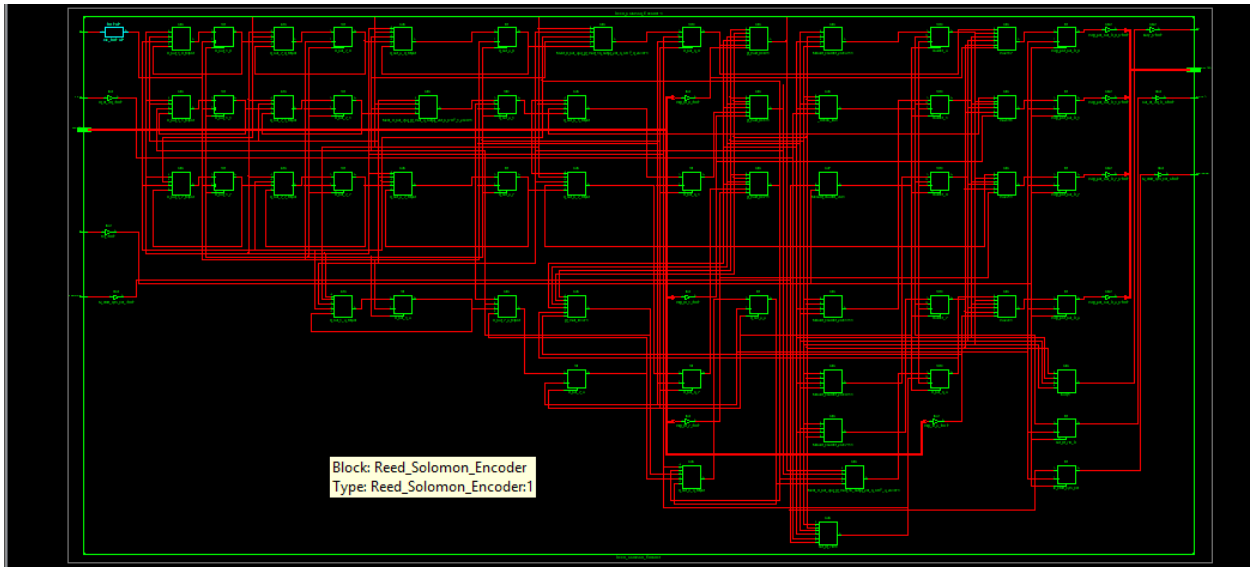

Figure 8.2 RTL Technology Schematic view of RS code

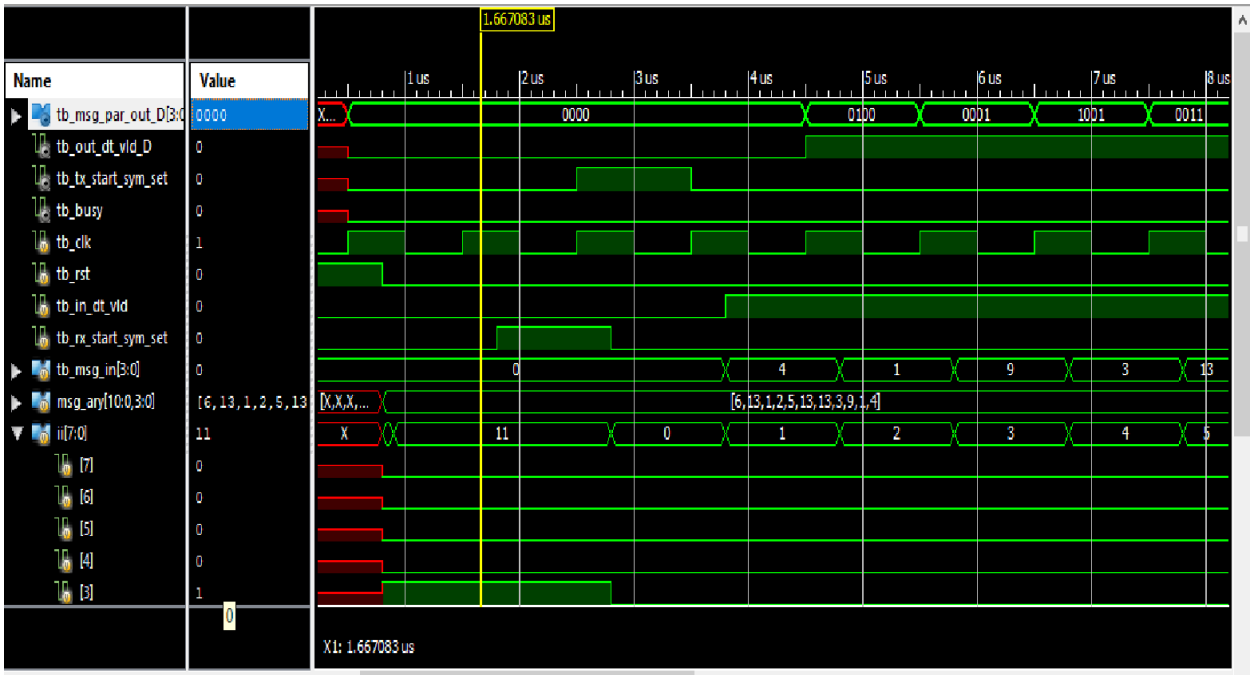

Figure 8.3 RTL Simulation output view of RS code

8.2.4 Synthesis Report:

A synthesis report is a summarized form of text. To make a synthesis you need to find suitable sources, and then to select the relevant parts in those sources. You will then use your paraphrase and summary skills to write the information in your own words. The information from all the sources has to fit together into one continuous text. Please remember, though, that when you synthesise work from different people, you must acknowledge it.

The synthesis report for the above verilog program which indicate all the digital components being use in this hardware model is given below.

Generated Synthesis report:

Release 14.5 - xst P.58f (nt64)

Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.

--> Parameter TMPDIR set to xst/projnav.tmp

Total REAL time to Xst completion: 0.00 secs

Total CPU time to Xst completion: 0.30 secs

--> Parameter xsthdpdir set to xst

Total REAL time to Xst completion: 0.00 secs
Total CPU time to Xst completion: 0.30 secs

--> Reading design: Reed_Solomon_Encoder.prj

TABLE OF CONTENTS

==========

* Synthesis Options Summary *

==========

---- Source Parameters
Input File Name : "Reed_Solomon_Encoder.prj"
Ignore Synthesis Constraint File : NO

---- Target Parameters
Output File Name : "Reed_Solomon_Encoder"
Output Format : NGC
Target Device : xc6slx4-2-tqg144

---- Source Options
Top Module Name : Reed_Solomon_Encoder
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FSM Style : LUT
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Shift Register Extraction : YES
ROM Style : Auto
Resource Sharing : YES
Asynchronous To Synchronous : NO
Shift Register Minimum Size : 2
Use DSP Block : Auto
Automatic Register Balancing : No

---- Target Options
LUT Combining : Auto
Reduce Control Sets : Auto
Add IO Buffers : YES
Global Maximum Fanout : 100000
Add Generic Clock Buffer(BUFG) : 16
Register Duplication : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Auto
Use Synchronous Set : Auto
Use Synchronous Reset : Auto
Pack IO Registers into IOBs : Auto
Equivalent register Removal : YES

---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Power Reduction : NO
Keep Hierarchy : No
Netlist Hierarchy : As_Optimized
RTL Output : Yes
Global Optimization : AllClockNets

```
Read Cores                      : YES
Write Timing Constraints        : NO
Cross Clock Analysis            : NO
Hierarchy Separator             : /
Bus Delimiter                   : <>
Case Specifier                  : Maintain
Slice Utilization Ratio         : 100
BRAM Utilization Ratio          : 100
DSP48 Utilization Ratio         : 100
Auto BRAM Packing               : NO
Slice Utilization Ratio Delta   : 5

=========================================================================

=========================================================================
*                            HDL Parsing                                *
=========================================================================
Analyzing Verilog file "D:\Project Material\verilog program\tabrej_verlog\RS_encoder\rs_encoder.v" into library work
Parsing module <Reed_Solomon_Encoder>.

=========================================================================
*                          HDL Elaboration                              *
=========================================================================

Elaborating module <Reed_Solomon_Encoder>.

=========================================================================
*                            HDL Synthesis                              *
=========================================================================

Synthesizing Unit <Reed_Solomon_Encoder>.
    Related source file is "D:\Project Material\verilog program\tabrej_verlog\RS_encoder\rs_encoder.v".
    Found 4-bit register for signal <d_out_2>.
    Found 4-bit register for signal <d_out_3>.
    Found 4-bit register for signal <d_out_4>.
    Found 4-bit register for signal <counter>.
    Found 4-bit register for signal <msg_par_out_D>.
```

```
    Found 1-bit register for signal <out_dt_vld_D>.
    Found 1-bit register for signal <tx_start_sym_set>.
    Found 4-bit register for signal <d_out_1>.
    Found 4-bit adder for signal <counter[3]_GND_1_o_add_34_OUT> created at line 111.
    Found 4-bit comparator greater for signal <cntrl_in> created at line 114
    Summary:
        inferred   1 Adder/Subtractor(s).
        inferred  26 D-type flip-flop(s).
        inferred   1 Comparator(s).
        inferred   2 Multiplexer(s).
Unit <Reed_Solomon_Encoder> synthesized.

=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                   : 1
 4-bit adder                          : 1
# Registers                            : 8
 1-bit register                       : 2
 4-bit register                       : 6
# Comparators                          : 1
 4-bit comparator greater             : 1
# Multiplexers                         : 2
 4-bit 2-to-1 multiplexer             : 2
# Xors                                 : 13
 1-bit xor2                           : 9
 4-bit xor2                           : 4

=========================================================================

=========================================================================
                        *  Advanced HDL Synthesis  *
=========================================================================


Synthesizing (advanced) Unit <Reed_Solomon_Encoder>.
The following registers are absorbed into counter <counter>: 1 register on signal <counter>.
Unit <Reed_Solomon_Encoder> synthesized (advanced).
```

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Counters                          : 1
 4-bit up counter                   : 1
# Registers                         : 22
 Flip-Flops                         : 22
# Comparators                       : 1
 4-bit comparator greater           : 1
# Multiplexers                      : 4
 1-bit 2-to-1 multiplexer           : 4
# Xors                              : 13
 1-bit xor2                         : 9
 4-bit xor2                         : 4

=========================================================================
                *               Low Level Synthesis             *
=========================================================================

Optimizing unit <Reed_Solomon_Encoder> ...

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block Reed_Solomon_Encoder, actual ratio is 1.

Final Macro Processing ...



=========================================================================
Final Register Report

Macro Statistics
# Registers                         : 26
 Flip-Flops                         : 26

=========================================================================
```

```
*                  Partition Report                  *
=====================================================================

Partition Implementation Status
-------------------------------

  No Partitions were found in this design.

-------------------------------

=====================================================================
*                  Design Summary                    *
=====================================================================

Top Level Output File Name         : Reed_Solomon_Encoder.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                      : 32
#      LUT2                : 1
#      LUT4                : 15
#      LUT5                : 7
#      LUT6                : 9
# FlipFlops/Latches              : 26
#      FDR                 : 19
#      FDRE                : 7
# Clock Buffers               : 1
#      BUFGP               : 1
# IO Buffers                  : 14
#      IBUF                : 7
#      OBUF                : 7

Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-2


Slice Logic Utilization:
```

```
 Number of Slice Registers:             25  out of   4800     0%
 Number of Slice LUTs:                  32  out of   2400     1%
    Number used as Logic:               32  out of   2400     1%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     32
   Number with an unused Flip Flop:       7  out of     32    21%
   Number with an unused LUT:             0  out of     32     0%
   Number of fully used LUT-FF pairs:    25  out of     32    78%
   Number of unique control sets:         4

IO Utilization:
 Number of IOs:                        15
 Number of bonded IOBs:                15  out of    102    14%
    IOB Flip Flops/Latches:            1

Specific Feature Utilization:
 Number of BUFG/BUFGCTRL/BUFHCEs:       1  out of     16     6%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.

---------------------------


=================================================================
Timing Report

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
```

```
-----------------------------------+-----------------------+-------+
clk                                | BUFGP                 | 26    |
-----------------------------------+-----------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
No asynchronous control signals found in this design

Timing Summary:
---------------
Speed Grade: -2

   Minimum period: 3.758ns (Maximum Frequency: 266.099MHz)
   Minimum input arrival time before clock: 4.340ns
   Maximum output required time after clock: 5.803ns
   Maximum combinational path delay: No path found

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 3.758ns (frequency: 266.099MHz)
  Total number of paths / destination ports: 305 / 32
-------------------------------------------------------------------------
Delay:               3.758ns (Levels of Logic = 2)
  Source:            counter_1 (FF)
  Destination:       d_out_1_3 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: counter_1 to d_out_1_3
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDRE:C->Q            15   0.525   1.431  counter_1 (counter_1)
     LUT6:I2->O            8   0.254   1.220  gf_mult_in<2>1 (gf_mult_in<2>)
     LUT5:I1->O            1   0.254   0.000  d_out_1_3_rstpot (d_out_1_3_rstpot)
```

```
    FDR:D                  0.074         d_out_1_3
    ----------------------------------------
    Total                  3.758ns (1.107ns logic, 2.651ns route)
                              (29.5% logic, 70.5% route)

==========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 120 / 55
--------------------------------------------------------------------------
Offset:              4.340ns (Levels of Logic = 2)
  Source:            rst (PAD)
  Destination:       d_out_3_1 (FF)
  Destination Clock: clk rising

  Data Path: rst to d_out_3_1
                            Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             7   1.328   1.018  rst_IBUF (rst_IBUF)
     LUT2:I0->O           20   0.250   1.285  Mcount_counter_val1 (Mcount_counter_val)
     FDRE:R                    0.459          counter_0
    ----------------------------------------
    Total                  4.340ns (2.037ns logic, 2.303ns route)
                              (46.9% logic, 53.1% route)

==========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 10 / 7
--------------------------------------------------------------------------
Offset:              5.803ns (Levels of Logic = 2)
  Source:            counter_3 (FF)
  Destination:       busy (PAD)
  Source Clock:      clk rising

  Data Path: counter_3 to busy
                            Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDRE:C->Q            15   0.525   1.431  counter_3 (counter_3)
```

```
    LUT4:I0->O            1   0.254   0.681  busy1 (busy_OBUF)
    OBUF:I->O                 2.912          busy_OBUF (busy)
    ----------------------------------------
    Total                     5.803ns (3.691ns logic, 2.112ns route)
                              (63.6% logic, 36.4% route)

==================================================================================

Cross Clock Domains Report:
--------------------------

Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    3.758|         |         |         |
---------------+---------+---------+---------+---------+

==================================================================================


Total REAL time to Xst completion: 14.00 secs
Total CPU time to Xst completion: 14.06 secs

-->

Total memory usage is 329524 kilobytes

Number of errors   :    0 (   0 filtered)
Number of warnings :    0 (   0 filtered)
Number of infos    :    0 (   0 filtered)
```

CONCLUSION

Reed Solomon (15,11) code was simulated in MATLAB. It was found that if the error is in parity symbols even then the decoder is able to detect the output and it is of no matter to the decoder that in which symbols the error is present. The decoder first corrects the symbols and then removes the redundant parity symbols from the code word and produces the original input code word. In the error probability graph, it was seen that for a particular range of SNR, the number of error bits present can be found out using the bit error rate probability. So if the number of bits in error is beyond our range of our error correcting capability, then the signal can be ignored in the same place and an acknowledgement can be sent to send the signal again. However, if the number of bits in error is within the range then also we can estimate using those bits about how many symbols will be in error and then decide whether to try decode the transmitted signal or not. For different error correcting capabilities of RS code, the range of SNR goes on increasing as the error correcting capability increases. In Verilog HDL, the RS encoder was designed and the timing diagrams and device utilization is stated in the results and discussions chapter. In future, one can design the RS decoder and then test this benchmark in a real time based example. One can also implement this in an FPGA.

REFERENCES

[1] Reed, I. S. and Solomon, G., 1960. "*Polynomial Codes over Certain Finite Fields"*, J. SIAM., Vol. 8, pp. 300-304, 1960.

[2] ETSI, 1997. "*Digital broadcasting systems for television, sound and data services; Framing structure, channel coding and modulation for digital terrestrial television."* European Telecommunication Standard ETS 300 744.

[3] Bose, R. C. and Chaudhuri, D. K. R., 1960. "*On a class of error-correcting binary group codes."* Inf. Control, Vol. 3, 1960.

[4] C. K. P. Clarke , "*Reed Solomon error detection and correction"* Research & Development BRITISH BROADCASTING CORPORATION, 1995

[5] G. O. Young, *"Synthetic structure of industrial plastics (Book style with paper title and editor),"* in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.

[6] James Clarke, *"Reed Solomon Error Correction Hardware Design"*, R&D White Paper, British Broadcating Corporation, July, 2002.

[7] A.R. Dash and T.R. Lenka, "*VLSI Implementation of Reed-Solomon Encoder Algorithm for Communication Systems*" Radio Electronics and Communications Systems, Springer, Sep 2013.

[8] Welch, L. R. (1997),*" The Original View of Reed–Solomon Codes"* (PDF), Lecture Notes

[9] Gill, John (n.d.), EE387 Notes #7, Handout #28 (PDF), Stanford University, archived from the original (PDF) on June 30, 2014, retrieved April 21, 2010

[10] Reed, Irving S.; Solomon, Gustave (1960), *"Polynomial Codes over Certain Finite Fields*", *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, **8** (2): 300–304,