A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science
in Electrical Engineering
By
Haoyi Zhang

December 2013
The Graduate Project of Haoyi Zhang is approved by:

Dr. Deborah van Alphen

Date

Dr. Sharlene Katz

Date

Dr. Nagi El Naga, Chair

Date

California State University, Northridge
Table of Contents

SIGNATURE PAGE .......................................................................................................................... II

ABSTRACT ...................................................................................................................................... V

CHAPTER 1 INTRODUCTION ........................................................................................................ 1
  1.1 HISTORY OF ERROR CONTROL CODING ................................................................. 1
  1.2 THE DEVELOPMENT OF REED-SOLOMON CODES ............................................. 2
  1.3 OBJECTIVE ..................................................................................................................... 2
  1.4 OUTLINE .......................................................................................................................... 2

CHAPTER 2 THEORIES OF ERROR CONTROL CODING ...................................................... 4
  2.1 BASICS OF CODING THEORY ..................................................................................... 4
  2.1.1 Introduction to the Development of Information Theory and Coding ..... 4
  2.1.2 Digital Communication System ............................................................................. 5
  2.1.3 Fundamental Theories and Categories of Channel Coding ......................... 7
  2.1.4 Introduction to error correcting codes ................................................................. 8
  2.2 THEORY OF FINITE FIELD (GAOLOIS FIELD) ...................................................... 11
  2.2.1 The Concept of Group ......................................................................................... 12
  2.2.2 Concept of Field .................................................................................................. 13
  2.2.3 Concept of Finite Field and Related Theories .................................................... 13
  2.2.4 Elements on GF(2^m) ......................................................................................... 14
  2.2.5 Calculations on Finite Field GF(2) ................................................................. 15
  2.2.6 Calculation on finite field GF(2^m) ................................................................. 16
  2.3 LINEAR BLOCK CODE ............................................................................................... 17
  2.3.1 Linear Block Code .............................................................................................. 17
  2.3.2 Cyclic Code ......................................................................................................... 20
  2.3.3 BCH Code .......................................................................................................... 22
  2.3.4 Reed-Solomon Code ......................................................................................... 24
  2.4 CONCLUSION OF THE CHAPTER ........................................................................... 25

CHAPTER 3 DESIGN OF REED-SOLOMON ENCODER ..................................................... 27
  3.1 RS ENCODING ALGORITHM ...................................................................................... 27
  3.2 ARITHMETIC REALIZATION ON THE FINITE FIELD ........................................... 29
  3.2.1 The Elements on GF(2^8) .................................................................................. 29
  3.2.2 Addition and Subtraction on Finite Field .......................................................... 31
  3.2.3 Multiplication on a Finite Field ......................................................................... 32
  3.2.4 Division on a finite field .................................................................................... 33
3.3 HARDWARE REALIZATION OF REED-SOLOMON ENCODERS ........................................ 33
3.4 ENCODING OF (204, 188) REED-SOLOMON CODE ................................................. 34
  3.4.1 Characteristics of RS(204, 188) Code ................................................................. 34
  3.4.2 Construction of GF(2^8) ..................................................................................... 34
  3.4.3 Constant Multiplier under Natural Basis ............................................................. 35

CHAPTER 4 DESIGN OF REED-SOLOMON DECODER ......................................................... 38
  4.1 PROCEDURES OF RS DECODING ........................................................................... 38
  4.2 CALCULATION OF THE SYNDROME ..................................................................... 39
  4.3 SOLVING THE ERROR LOCATION POLYNOMIAL ................................................... 40
    4.3.1 Key Equation ................................................................................................. 40
    4.3.2 Berlekamp-Massey Algorithm ........................................................................ 42
    4.3.3 An Improved Berlekamp-Massey Algorithm ................................................ 44
  4.4 SEARCH FOR ERROR LOCATION (CHIEN SEARCH) .................................................. 45
  4.5 DETERMINE THE ERROR VALUES ......................................................................... 46
  4.6 HARDWARE REALIZATION OF REED-SOLOMON DECODER ......................... 47
    4.6.1 Structure Diagram and Top Level Module of the Decoder .............................. 47
    4.6.2 Syndrome Calculation Module ........................................................................ 48
    4.6.3 Error Location Polynomial Calculation Module ........................................... 49
    4.6.4 Chien Search Module .................................................................................... 50
    4.6.5 Error Value Evaluator Module ...................................................................... 51
    4.6.6 Storage Module ............................................................................................. 51

CHAPTER 5 SYNTHESIS AND SIMULATION OF (204, 188) REED-SOLOMON ENCORDER/DECODER .......................................................... 53
  5.1 SYNTHESIS AND SIMULATION OF THE ENCODER .............................................. 53
  5.2 SYNTHESIS AND SIMULATION OF THE DECODER ................................................. 56
    5.2.1 Input / Output with 0 Error ............................................................................ 56
    5.2.2 Input / Output with 6 Errors .......................................................................... 59
    5.2.3 Input / Output with 8 Errors .......................................................................... 60
    5.2.4 Input / Output with 9 Errors .......................................................................... 62

CHAPTER 6 CONCLUSION ......................................................................................... 64
REFERENCES ........................................................................................................... 66
APPENDIX A RTL DESCRIPTION ............................................................................ 70
APPENDIX B VERILOG HDL FILES .......................................................................... 72
APPENDIX C MATLAB FILES ................................................................................ 119
ABSTRACT

(204, 188) REED-SOLOMON CODE ENCODER/DECODER

DESIGN AND SIMULATION

WITH MODELSIM

By

Haoyi Zhang

Master of Science in Electrical Engineering

Reed-Solomon (RS) codes are commonly used in the digital communication field due to their strong capabilities to eliminate both random errors and burst errors. In this project, the encoding and decoding systems of a (204, 188) Reed-Solomon code are designed, synthesized, and simulated. In the first part of this paper, basic backgrounds and fundamental theories of error control coding are introduced. Then the focus shifts to the illustration of finite field theory and several important linear block codes. Finally, the main discussion moves onto the topic of RS(204, 188) code encoder/decoder design, modeling, simulation, and verification using Verilog HDL and Altera Quartus ii.
Chapter 1 Introduction

1.1 History of Error Control Coding

Error control coding is a branch of information science developed accompanied by the advancing of modern digital communication theory. The purpose of the study is to detect and correct the errors, which are created by the interference from noise, which occurs in the communication process, and hence, to ensure the reliable delivery of data \cite{1}. In 1948, Claude Shannon who was an employee of AT&T published his famous paper *A Mathematical Theory of Communication* \cite{13}, which subsequently became the cornerstone of modern information science. The formula describing the relationship between channel capacity, information transmission rate, and signal-to-noise ratio appears in every textbook on information theory \cite{13}:

For the case of Digital Signals plus white Gaussian noise, there exists a channel capacity $C$ in bits/second such that if the information rate $R$ (bits/second) is less than $C$, the probability that the received signal is in error would approach zero

$$C=\text{Blog}_2(1+S/N).$$

This formula indicates that, theoretically, error-free transmission can be achieved under certain circumstances as long as the information rate is less than channel capacity. Practically, in order to receive ideal bit error rate, the designer needs to choose appropriate baseband signal and modulation/demodulation system, along with other operations such as frequency domain and time domain equalization, or increasing the signal-to-noise ratio on both the receiving and sending ends. Performing error control coding on the information being transmitted, or in other words, channel coding, is the most efficient method among all techniques. One thing that needs to be pointed out is that error control coding is applied at the cost of other performance factors of the information system, such as signal-to-noise ratio, or the efficiency of the transmission. The actual significance of it lies on the prominent decrease of bit error rate or increase of data transmission rate with only a little signal power added. Shannon stated in his paper that,
through expanding the code length of a block code, an arbitrarily small bit error rate can be achieved \cite{13}. He also pointed out that the signal-to-noise ratio needs to be at least -1.6db, which is thereafter called the Shannon limit, to ensure an arbitrarily small bit error rate in an additive white Gaussian noise channel. Hence, the centerpiece of error control coding became the construction of a code which can approach the Shannon limit and receive a better bit error rate at the same time.

1.2 The development of Reed-Solomon codes

Reed-Solomon codes were first presented by Irvine Reed and Gustave Solomon in 1960 \cite{2}. The algorithm of its encoder is comparatively simple and the most important components are multipliers. Although the error correcting capability of RS codes is beyond satisfaction, they were not widely used in their early years because of the lack of efficient decoding algorithms. W.W. Peterson firstly recognized RS codes as a special class of BCH codes and provided an algebraic decoding method based on syndrome and elementary symmetric function. This work is very important to the development of the decoding algorithms of RS codes in spite of the fact that it is seldom mentioned. Then there came E.R. Berlekamp who modified Peterson’s algorithm into iteration form and made the decoding of RS codes with high error correcting capability possible \cite{7}. J. Massey discovered that Berlekamp’s algorithm equals finding the shortest linear shift register which can generate the given sequence \cite{8}. Their work combined is the BM algorithm which is discussed in chapter 4.

1.3 Objective

A Reed-Solomon code with block size n=204 and error correcting capability t=8 is chosen in this project. The objective of this project is mainly reflected in three aspects: firstly, reinforcing the basic rules and theories of error control coding; secondly, studying and understanding RS codes in detail; lastly, designing, modeling, and simulating RS code encoder/decoder with Verilog HDL.

1.4 Outline
The first chapter is the introduction to the whole project. It presents the background of the project. It discusses the history of error control coding and the development of RS codes.

Chapter two briefly introduces the fundamental knowledge of error control coding at the beginning. Then the emphasis transfers to the theories of finite field and several important linear block codes.

The third chapter discusses the design and implementation of RS code encoder, including the algorithm of RS encoder, the calculation on the finite field, hardware implementation of the encoder, and circuit simulation.

Chapter four presents the design and implementation of RS code decoder, including the decoding procedures, the calculation of the syndromes, solving the error location polynomials, locating errors, error value calculation, hardware implementation of the decoder, and circuit simulation.

Chapter five shows the synthesis, simulation, and verification of the whole project. Altera Quartus ii, Modelsim, and MATLAB are employed to assist to accomplish this goal. In this chapter, two sets of data are studied and discussed to verify the functionalities of the (204, 188) Reed-Solomon code designed in the previous chapters.

The last chapter, Chapter six, concludes the report and probes into future possible research directions.
2.1 Basics of Coding Theory

2.1.1 Introduction to the Development of Information Theory and Coding

After more than half century’s progress, information theory has become a professional and scientific research subject which has already developed a complete set of theories.

Information and coding theory is gradually advanced and perfected over years with numerous scientists’ constant studies and researches. In the year of 1928, Ralph Hartley published his paper “Transmission of Information” in which the concept of “information” was raised and defined for the first time. Then “the father of information theory” Claude Shannon scientifically defined the concept of information in his 1948 paper *A Mathematical Theory of Communication*[^13] based on the substance of information transmission and massive studies and researches. He treated the major problems of communication systems using mathematical methods and proposed distortionless source coding theory and noisy channel coding theory which form the foundation of modern information theory and provide the guidance for the development of channel coding techniques. In the mid-50s of last century Richard Hamming invented a linear block code which was named “Hamming Code” afterwards. Thereafter, algebraic methods were implemented into error control coding and led to the birth of algebraic coding theories and techniques. In year 1957 Eugene Prange put forward the concept of cyclic codes which has already been one of the most important parts of error control coding. In 1959 and 1960 the BCH codes were independently invented by Alexis Hocquenghem, Raj Bose, and D. K. Ray-Chaudhuri and then named by the initials of these inventors’ names. BCH codes possess several advantages among which the possibility to design binary BCH codes that can correct multiple bit errors and the ease with which they can be decoded via syndrome decoding are the most recognized two. In 1955, Peter Elias
introduced for the first time convolutional codes as an alternative to block codes. In his 1965 thesis paper, George David Forney presented the idea of concatenated codes.

2.1.2 Digital Communication System

The concept of communication system is the sum of all the systems and devices that are involved in information transmission. Communication system can be generally divided into two classes based on the different types of signals that are transmitted. One is digital communication system and the other is analog communication system. Enjoying the benefit that it is amenable to both error control and encryption, digital communication system is more often used nowadays.

Examples of communication systems in modern life are television system, telephone system, radio system, radar system, and navigation system, to name a few. These systems share the same basic principles in spite of their different properties and functionalities. Through broad research on the common characteristics of all communication systems, a general and universally applicable model is derived, which is shown below in figure 2.1[3].

![Figure 2.1 Communication System Model](image)

The explanations of each part in figure 2.1 is as follows,

Source: origin of information.

Source can be anything and the information it sends out can be either discrete or continuous. In general, the information sequences source transmitted are stochastic and need to be described with random variables.
Encoder: the part where encoding is performed.

Encoders include source encoder and channel encoder. Source encoder converts the output of the source into a certain form to increase the efficiency of information transmitting. Channel encoding is to perform redundancy appended on the output of source encoder to enhance the distortion resistance of channel and the accountability of transmission.

Modulator: Transform the outputs from the encoder into a format that can be transmitted on the channel and modulate its frequency, amplitude, or phase in order to accommodate long distance transfer. The outputs of a modulator are signals.

Channel: The media through which the communication system transmits signals from the sending end to the receiving end, or simply the transmission path of signals.

Common types of channels contain wire, optical fiber, cable, radio propagation space, microwave channel. Interference or noise exists in channels.

Interference Source: Generates noise.

Interference and noise are both random. Interference happens on transmission channel or storage medium.

Demodulator: Performs demodulation on the signals received from channel. Demodulation is the inverse process of modulation.

Decoder: Decode the signals from demodulator. Decoding is the inverse process of encoding.

There are two types of decoding: channel decoding and source decoding. Channel decoding implements error detection and correction according to redundancies added in the channel coding part. Source decoding reduces the redundancies produced by the channel coding to transform the code word into the original information.

Sink: Receives information. It is the destination of data transmission. Sink is normally a person or a machine.
2.1.3 **Fundamental Theories and Categories of Channel Coding**

In communication systems, information is transmitted through physical media which are channels. Since channels cannot avoid the interference from the outside world, the transmitted information will be distorted which ultimately results in errors in the received information. Therefore, error control is employed to help optimize the accuracy and reliability of the transmission and advance the distortion resistance capability of the channel. Error control is a method that can detect and correct errors.

Channel encoding is to add redundancy bits to the information sequences by certain rules to enhance the dependability of the transmission, thus improve the efficiency of the whole communication system. Channel encoding is applied on the message sequences (denoted by M) sent by source encoder. The main goal of channel encoding is to yield a code which possesses the best noise resistance while appending the least redundancy.

Channel encoding can be divided into different types from different aspects.

According to their abilities, channel encoding can be categorized as error detecting code (can locate errors), error correcting code (can detect and correct errors), and erasure correcting code (can correct erasure errors).

Different coding methods need to be applied for different channels. There are three types of errors: random and independent errors, burst errors, and mixed errors.

Theoretically, there are many rules and methods to increase redundancy. Channel coding can be divided into two categories by different rules: linear code and non-linear code. For a linear code, message bits and redundancy bits are mathematically linear and their relationship can be described with linear polynomials. For a non-linear code, message bits and redundancy bits are not mathematically linear.

From the aspect of coding method, channel coding can be assorted as: convolutional code and block code.

Encoding is to group every k message bits sending from the source to produce n-bit (n=k+r) code words.
In a convolutional code, r redundancy bits are decided by the k message bits in the same group and the message bits in the previous L group where L is called code storage. Convolutional code is denoted as (n, k, L).

In block code, r redundancy bits are only decided by the k message bits in the same group. It is represented by (n, k). Block code can further be divided into two types: cyclic code and non-cyclic code. A code is recognized as a cyclic code, if for every code word from this code, the code word obtained by a cyclic shift of components is again a code word. With regard to a non-cyclic code, it is not necessary that the word acquired from a cyclic shift is still a code word.

Based on the carry system of the components, there are binary codes and non-binary codes.

2.1.4 Introduction to error correcting codes

2.1.4.1 Error-control methods

Generally, there are two patterns of error control: the first one is that once errors are detected, the decoder automatically corrects these errors based on certain rules; another method is that once errors occur, instead of correcting the errors, the receiving end sends a feedback signal to the transmitting end telling it that errors occurred and requests that message to be sent again.

Error-control methods can be categorized into four types.

Automatic Repeat Request (ARQ)

ARQ is an error-control method that uses acknowledgements and timeouts to fulfill reliable data transmission over an unreliable service. When the message received by the receiver is error-free, an acknowledgement will be sent to the sender to inform it that the data contains no error. On the other hand, if the sender does not receive an acknowledgement before the timeout, the message will be retransmitted until an acknowledgement reaches the sender.
The advantages of ARQ are that it does not need as many redundancy bits and it is applicable to every type of channels. The weaknesses of it are that a feedback channel must exit in the transmission channel and the quality of real-time transmission is far from acceptable.

**Forward Error Control (FEC)**

In FEC, the sending end encodes the message by appending redundancies using an error correcting code. The redundancies added give the receiver the ability to detect a limited number of errors that may occur anywhere in the transmitted message and often to locate and correct these errors without retransmission. With the assistance of this mechanism, FEC allows the errors to be corrected at the receiving end without the presence of a feedback channel. However, the disadvantage is that FEC requires a fixed and higher forward channel bandwidth because of the redundancies. This drawback makes FEC a better choice where retransmissions are costly or impossible at all, such as multicast and real-time transmission system.

**Hybrid Error Correction**

This methodology is the combination of the previous two methods, therefore it is called hybrid error correction. It works as follows: after the receiver obtains the code words, if the number of errors is within the error correcting capability, then it performs error correction; if it is out of the range of the error correcting capability, then it requests re-transmission. This method combines the advantages of the previous two, which makes it adaptive to high-speed transmission system.

**Information Repeat Request**

This method works as follows: after the receiver obtained the data, it transmits them back to the sending end. The received information will be compared with the original one at the sender to detect errors. This method embraces the benefits of simpler error detection and correction theory and simpler hardware structure. The disadvantages are that the transmitting speed is too slow, and the process is too complicated.
2.1.4.2 Introduction to the Basic Concepts of Error Control Coding

In Error-Correcting codes, at the sending end the information sequence is divided into several segments with the same length. Each segment containing k information bits is called a message word or a message block and it is denoted by \( M = (m_k, m_{k-1}, \ldots, m_1) \), among which \( m_i(i=1,2,\ldots, k) \) are called information elements or information bits. Encoder generates r redundancy bits according to certain rules and then outputs a sequence of which the length is n. The sequence can be represented as

\[
C = (c_n, c_{n-1}, \ldots, c_2, c_1)
\]  

(2.1)

This sequence is called code word or code block and \( n=k+r \) indicates the length of the code, in another word, the number of code elements. The elements \( c_i(i=1,2,\ldots,n) \) are called code elements and the added r information bits are called redundant or parity-check bits. \( R=k/n \) is called source rate or rate. The number of non-zero bits in a binary code word is called the Hamming weight or weight of this code word and it is denoted by
For two code words with the same length, if they have different code bits at the same position, then the number of such positions is called the Hamming distance or distance for short of the two code words and it is denoted by D. In a code, any two arbitrary code words have different Hamming distance, among which the smallest is called the minimum distance and it is denoted by $d_{\text{min}}$. The error correcting capability is related to the minimum distance.

2.1.4.3 Theory of Error Correction Coding

The purpose of error correcting code is to firstly check if there is error in the received data. If there is, it performs error correction to ensure the dependability of the information transmission.

As stated in the previous section, in channel coding, the information sequence is divided each k bits to form the message word M. Then, r redundancy bits are added to the end of each message word to form each code word C of which the length is n. This process is showed as in figure 2.2.

**Figure 2.2 Process of Error Control Coding**

This type of code is called a block code and it is denoted by $(n, k)$. The parity check bits are denoted by $r = n - k$.

The structure of a block code is given by figure 2.3 [3].

**Figure 2.3 Structure of A Block Code**

2.2 Theory of Finite Field (Galois Field)
Finite field or Galois field is an algebraic theory raised by French mathematics genius Évariste Galois. Galois fields are very important in coding theory. The Reed-Solomon codes studied in this paper are based on finite fields. Therefore, before discussing the algorithm of RS codes, knowledge of Galois field theory is a prerequisite.

2.2.1 The Concept of Group

For a given set $G = \{a, b, \ldots\}$ and a given operation ”*”, if it satisfies the following four conditions [1]:

1) Closure. For any $a \in G$, $b \in G$, the following expression is always true

$$a * b = c \in G; \tag{2.2}$$

2) Associativity. For any $a, b, c \in G$, the following expression is always true

$$(a * b) * c = a * (b * c); \tag{2.3}$$

3) Identity element. For any $a \in G$, there is an element $e \in G$ which makes the following expressions always true

$$a * e = a \tag{2.4}$$

$$e * a = a; \tag{2.5}$$

4) Inverse element. For any $a \in G$, there is an element $a^{-1} \in G$ which makes the following expression always true

$$a * a^{-1} = a^{-1} * a = e; \tag{2.6}$$

then the element set $G$ along with operation ”*” form a group.

If the elements of $G$ also satisfies commutativity with operation ”*”, i.e. $a * b = b * a$, then $G$ is called a commutative group.

If the elements of set $G$ are the powers of one particular element $g$, i.e.,

$$G = \{g, g^2, \ldots, g^n\}, \tag{2.7}$$
then group $G$ is called a cyclic group and the element $g$ is called a generator or a primitive element. A cyclic group is a commutative group.

### 2.2.2 Concept of Field

For a given non-empty element set $F = \{a, b, \ldots\}$ and two operations “+” and “*”, if it satisfies the following three conditions \(^{[1]}\):

1) Elements on $F$ form a commutative group along with operation “+”;

2) Elements on $F$, excluding “0”, form a commutative group along with operation “*”:

3) $F$ satisfies distributive property, i.e., for any element $a, b, c \epsilon F$, the following expression is always true

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

$$= a \ast b + a \ast c$$ \hspace{1cm} (2.8)

then set $F$ forms a field along with operations “+” and “*”.

### 2.2.3 Concept of Finite Field and Related Theories

If the number of elements on a field $F$ is finite, this field is called a finite field, or a Galois field \(^{[1]}\). The number of elements is called the order of the field. Galois field $F = \{0, 1, 2, \ldots, p-1\}$ is a finite field with modulus $p$ and order $p$ and it can be represented as $GF(p)$. Number $p$ is a prime number.

A polynomial over a field is a polynomial of which the coefficients are the elements of a Galois field $GF(p)$. The polynomial over $GF(p)$ is represented as \(^{[1]}\)

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_px^p;$$ \hspace{1cm} (2.9)

in which $a_0 \epsilon F, i = 0, 1, 2,\ldots, p.$
Irreducibility: a polynomial \( p(x) \) with coefficients on a finite field \( F \) is said to be irreducible over \( F \) if and only if \( p(x) \) cannot be represented as the product of two polynomials with lower orders.

\[ a_0 + a_1 \alpha + \ldots + a_{m-1} \alpha^{m-1} \in GF(p^m); \]  

in which \( a_i \in GF(p), \) \( i = 0, 1, 2, \ldots, m-1. \)

\[ a_0 + a_1 \alpha + \ldots + a_{m-1} \alpha^{m-1} \in GF(p^m); \]  

GF(\( p^m \)): Let \( p(x) \) be an irreducible polynomial over \( GF(p) \) with order \( m \). Then all the polynomials with coefficients on \( GF(p) \) form a field \( GF(p^m) \) with order \( p^m \). On \( GF(p^m) \), equation \( p(x) = 0 \) has a root \( \alpha \). Galois field \( GF(p^m) \) contains all the polynomials of \( \alpha \) with order less than or equal to \( m-1 \) and all the coefficients of those polynomials are on \( GF(p) \), i.e.,

```
\[ a_0 + a_1 \alpha + \ldots + a_{m-1} \alpha^{m-1} \in GF(p^m); \]  
```

Minimal polynomial: a polynomial is called the minimal polynomial of \( \alpha \) if its coefficients are on \( GF(p) \) and its power is the lowest.

Generator element: The elements in set \( F \) excluding “0” are represented by \( F\{0\} \). The order of set \( F\{0\} \) is \( r = p^m - 1 \). If this group contains the following infinite elements:

\[ 1, \alpha, \alpha^2, \ldots, \alpha^{r-1}, \alpha^r, \]  

then \( \alpha \) is called the generator element of \( F \).

Primitive element: on a finite field \( GF(p) \), an element \( \alpha \) is called the primitive element if its \( (p-1)^{th} \) power is still an element on this field. For any arbitrary finite field \( F \), there is a primitive element which is the generator element of group \( F\{0\} \). Every element on that field can be generated from the primitive element. Every element on the field can be denoted as the combination of the first \( (p-1) \) powers of \( \alpha \).

Primitve polynomial: a primitive polynomial is a minimal polynomial of which the coefficients are on \( GF(p) \) and has the primitive element as its root.

### 2.2.4 Elements on \( GF(2^m) \)
Finite field GF(2) is a binary field. It contains two elements, 0 and 1. GF(2) is a subfield of GF(2\textsuperscript{m}). On finite field GF(2\textsuperscript{m}), there are more elements, indicated by \( f \), other than 0 and 1. An infinite set \( G \) can be constructed by 0, 1, and \( f \), and it is shown as below,

\[
G = \{0, 1, f, f^2, f^3, ..., f^k, ...\} = \{0, f^0, f^1, f^2, f^3, ..., f^k, ...\} \tag{2.12}
\]

In order to get all the elements on Galois field GF(2\textsuperscript{m}), one condition needs to be added,

\[
f^{2^m-1} + 1 = 0, \tag{2.13}
\]

i.e.,

\[
f^{2^m-1} = 1 = f^0. \tag{2.14}
\]

Based on this condition, the elements of which the order is greater than \( 2^m - 1 \) can be represented as the elements of which the order is less than \( 2^m - 1 \),

\[
f^{2^m+n} = f^{2^m-1}f^{1+n} = f^{1+n}. \tag{2.15}
\]

Therefore, the elements on field \( G \) can be shown as below,

\[
\{0, 1, f, f^2, ..., f^{2^{m-2}}, f^{2^{m-1}}/2, f^{2^m}, ...\} \leftrightarrow \{0, f^0, f^1, f^2, ..., f^{2^{m-2}}, f^0, f^1, ...\}. \tag{2.16}
\]

Then all the elements on GF(2\textsuperscript{m}) are

\[
\{0, f^0, f^1, f^2, ..., f^{2^{m-2}}\}. \tag{2.17}
\]

2.2.5 Calculations on Finite Field GF(2)

Calculations on a finite field possess following features,

1) Both addition and multiplication satisfy commutative, associative, and distributive properties.

2) Both addition and multiplication are closed set arithmetics, in another word, the operands and the result are in the same set.

3) Any element on the field satisfies the relationship below
\[ u + 0 = u \]  \hspace{1cm} (2.18)
\[ u \times 1 = u. \]  \hspace{1cm} (2.19)

4) Any element on the field satisfies
\[ u + (-u) = 0. \]  \hspace{1cm} (2.20)

When \( u \neq 0 \), there exists an exclusive inverse element \( u^{-1} \) which satisfies
\[ u \times u^{-1} = 1. \]  \hspace{1cm} (2.21)

GF(2) is the simplest field in finite field.

An order \( n \) polynomial \( p(x) \) on GF(2) can be shown as below
\[ p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + \cdots + p_nx^n, \]  \hspace{1cm} (2.22)
in which the coefficients \( p_0, p_1, \ldots, p_n \) are the binary values on GF(2).

The polynomials on Galois field GF(2) can be achieved according to weighting coefficients. For example, the polynomial related to binary number 10101001 is \( x^7 + x^5 + x^3 + 1 \). The polynomial arithmetic is binary on GF(2). It is shown as below
\[
(1 + x + x^3 + x^4) \cdot (x^2 + x^3) \\
= x^2 + x^3 + x^3 + x^4 + x^5 + x^6 + x^6 + x^7 \\
= x^2 + x^4 + x^5 + x^7. \]  \hspace{1cm} (2.23)

From the calculation above, if the number of the same product is even, it should be removed, such as \( x^3 \) and \( x^6 \); if the number of the same product is odd, it should be retained, such as \( x^2, x^4, x^5, \) and \( x^7 \).

### 2.2.6 Calculation on finite field GF(2^m)

As is discussed above, for any finite field \( F \), there is a primitive element \( \alpha \), and each element on the field can be represented as a polynomial of \( \alpha \) with coefficients on GF(2^m) and order less than m.
The non-zero elements on field GF(2^m) have three display forms: power form, polynomial form, and vector form. For example, let the primitive polynomial and primitive element of field GF(2^4) respectively be \( p(x) = x^4 + x + 1 \) and \( \alpha \). Then it can be shown that \( \alpha^4 + \alpha + 1 = 0 \), or \( \alpha^4 = \alpha + 1 \). Hence, all the elements on Galois field GF(2^4) can be obtained and are shown in table 2.1 [1].

Table 2.1 Elements of field GF(2^4)

<table>
<thead>
<tr>
<th>Power Form</th>
<th>Polynomial Form</th>
<th>Vector Form (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>( \alpha^0 )</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>( \alpha^1 )</td>
<td>( \alpha )</td>
<td>0010</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>( \alpha^2 )</td>
<td>0100</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>( \alpha^3 )</td>
<td>1000</td>
</tr>
<tr>
<td>( \alpha^4 )</td>
<td>( \alpha + 1 )</td>
<td>0011</td>
</tr>
<tr>
<td>( \alpha^5 )</td>
<td>( \alpha^2 + \alpha )</td>
<td>0110</td>
</tr>
<tr>
<td>( \alpha^6 )</td>
<td>( \alpha^3 + \alpha^2 )</td>
<td>1100</td>
</tr>
<tr>
<td>( \alpha^7 )</td>
<td>( \alpha^3 + \alpha + 1 )</td>
<td>1011</td>
</tr>
<tr>
<td>( \alpha^8 )</td>
<td>( \alpha^2 + \alpha + 1 )</td>
<td>0101</td>
</tr>
<tr>
<td>( \alpha^9 )</td>
<td>( \alpha^3 + \alpha )</td>
<td>1010</td>
</tr>
<tr>
<td>( \alpha^{10} )</td>
<td>( \alpha^2 + \alpha + 1 )</td>
<td>0111</td>
</tr>
<tr>
<td>( \alpha^{11} )</td>
<td>( \alpha^3 + \alpha^2 + \alpha )</td>
<td>1110</td>
</tr>
<tr>
<td>( \alpha^{12} )</td>
<td>( \alpha^3 + \alpha^2 + \alpha + 1 )</td>
<td>1111</td>
</tr>
<tr>
<td>( \alpha^{13} )</td>
<td>( \alpha^3 + \alpha^2 + 1 )</td>
<td>1101</td>
</tr>
<tr>
<td>( \alpha^{14} )</td>
<td>( \alpha^3 + 1 )</td>
<td>1001</td>
</tr>
</tbody>
</table>

2.3 Linear Block Code

2.3.1 Linear Block Code
Let $x_{m1}$, $x_{m2}$, ..., $x_{mk}$ be the $k$ information bits of code word $c_m$, and their vector can be shown as

$$x_m = [x_{m1}, x_{m2}, ..., x_{mk}]. \tag{2.24}$$

The output vector of the encoder can be represented as

$$c_m = [c_{m1}, c_{m2}, ..., c_{mn}]. \tag{2.25}$$

The encoding algorithm of binary linear block code can be represented as

$$c_{mj} = x_{m1}g_{1j} + x_{m2}g_{2j} + \cdots + x_{mk}g_{kj}. \tag{2.26}$$

in which $j = 1, 2, ..., n$ and $g_{kj} = 0$ or $1$.

Hence, it can be shown that \[^1\] $c_m = x_mG.$ \tag{2.27}

The matrix $G$ above is called the generator matrix. It shows that the code words are generated by the information bits. Let

$$G = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k1} & g_{k2} & \cdots & g_{kn} \end{bmatrix}, \tag{2.28}$$

then each code word vector can be represented using the linear combination of the row vectors of $G$, i.e.,

$$c_m = x_{m1}g_1 + x_{m2}g_2 + \cdots + x_{mk}g_k. \tag{2.29}$$

$\{g_i\}$ is called the basis of $(n, k)$ block code. Generally, the selection of the basis is not unique; therefore the generator matrix $G$ is not unique.

Each generator matrix of a linear block code can be transformed into the special format below.
\[ G = [I_k | P] = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & p_{11} & p_{12} & \cdots & p_{1,n-k} \\ 0 & 1 & 0 & \cdots & 0 & p_{11} & p_{11} & \cdots & p_{2,n-k} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & p_{k1} & p_{k2} & \cdots & p_{k,n-k} \end{bmatrix}, \quad (2.30) \]

in which \( I_k \) is an identity matrix of size \( k \) and \( P \) is a \( k \times r \) matrix. Let \( H \) be a \( r \times n \) matrix, then its transpose matrix \( H^T \) satisfies \( GH^T = 0 \). \( H \) is called the parity check matrix, and it can be represented as

\[
H = [-P^T | I_{n-k}]. \quad (2.31)
\]

The relationship between the parity check matrix \( H \) and the code word matrix \( C \) is \( CH^T = 0 \).

The encoding of a linear block code is a process that transforms the information bits into code words based on the generator matrix \( G \) or the parity check matrix \( H \). The code words are the outputs of the encoder. They are transmitted through the channel to the decoder. The purpose of the decoder is to detect and correct errors then get the information bits.

Let the parity check matrix of linear block code \((n, k)\) be \( H \), the error pattern be \( E \), and the sequence received by the decoder be \( R \). Then \( S = RH^T = EH^T \) is called the syndrome of sequence \( R \)\(^{[1]}\). Syndrome has the following three characteristics,

1) Syndrome is decided only by the error pattern. It reflects the interference of the channel and it has nothing to do with the code words.

2) Syndrome is utilized to determine whether an error happened in the transmission. If \( S = 0 \), there is no error in the transmission, and the received sequence is the code word; while if \( S \neq 0 \), then there is error, hence error correction is needed.

3) Different error patterns will generate different syndromes. For a binary code, syndrome is the sum of each column vectors corresponding to the error patterns in the parity check matrix.
The error correcting capability of a linear block code is related to its minimum distance $d_{\text{min}}$, and the relationship is described as below:

1) To correct $t$ errors, it is needed that $d_{\text{min}} \geq 2t+1$;

2) To detect $e$ errors, it is needed that $d_{\text{min}} \geq e+1$;

3) To detect $e$ errors and correct $r$ errors at the same time, it is needed that $d_{\text{min}} \geq e+t+1$.

To make the minimum code distance of code $(n, k)$ equal $d_{\text{min}}$, there must exist $d_{\text{min}}-1$ columns in the parity check matrix $H$ which are linearly independent. This is the basis of constructing a linear block code. Therefore, the minimum distance $d_{\text{min}}$ of every linear block code satisfies $d_{\text{min}} \leq n-k+1$.

### 2.3.2 Cyclic Code

Cyclic codes are the most widely used linear block code.

For an $(n, k)$ linear block code $C$, if, for every code word from $C$, the word obtained by a cyclic shift of components is again a code word, then $C$ is called a cyclic code \[^3\]. In another word, if $C=(c_n c_{n-1} \ldots c_2 c_1)$ is a code word, $C=(c_{n-1} c_{n-2} \ldots c_2 c_1 c_n)$ is also a code word.

A cyclic code $C=(c_n c_{n-1} \ldots c_2 c_1)$ can be described with a polynomial with degree less than $n-1$,

$$c(x) = c_n x^{n-1} + c_{n-1} x^{n-2} + \cdots + c_2 x + c_1. \quad (2.32)$$

After $i$ cyclic shifts of this code word, the new word is also a code word of this cyclic code, and its polynomial is

$$c^{(i)}(x) = c_{n-i} x^{n-1} + c_{n-i-1} x^{n-2} + \cdots + c_1 x^i + c_0 x^{i-1} + c_{n-1} x^{i-2} + \cdots + c_{n-i+1}. \quad (2.33)$$

It can be derived from the previous two equations that
\[
\frac{x^i c(x)}{x^n + 1} = c_n x^{i-1} + c_{n-1} x^{i-2} + \cdots + c_{n-i+1} \\
+ \frac{c_{n-i} x^{n-1} + c_{n-i-1} x^{n-2} + \cdots + c_{1} x^i + c_0 x^{i-1} + c_{n-1} x^{i-2} + \cdots + c_{n-i+1}}{x^n + 1} \\
= c_n x^{i-1} + c_{n-1} x^{i-2} + \cdots + c_{n-i+1} + \frac{c^{(i)}(x)}{x^{n+1}}. 
\] (2.34)

It follows that \(c^{(i)}(x)\) is the remainder of dividing the product of \(c(x)\) and \(x^i\) by \((x^n + 1)\), i.e.,
\[
c^{(i)}(x) = x^i c(x) \mod (x^n + 1). 
\] (2.35)

This equation illustrates the relationship between the code polynomials and the cyclic shifts of the code words.

In \((n, k)\) cyclic code, every monic polynomial of order \((n-k)\) which can evenly divide \((x^n + 1)\) is the generator matrix of this code. It is denoted by \(g(x)\). Perform \((k-1)\) cyclic shifts on \(g(x)\) to obtain \(k\) linearly independent code polynomials,
\[
g(x), xg(x), \ldots, x^{k-1} g(x). 
\] (2.36)

The generator matrix \(G(x)\) is acquired using these \(k\) code polynomials,
\[
G(x) = \begin{bmatrix}
         x^{k-1} g(x) \\
x^{k-2} g(x) \\
\vdots \\
x g(x) \\
g(x)
         \end{bmatrix}.
\] (2.37)

Once the generator matrix of the code is fixed, the code is fixed, which means that \((n, k)\) cyclic code can be generated using \(g(x)\). The \(g(x)\) is called the generator polynomial of the code, and it is expressed as below \(^1\),
\[
g(x) = g_{n-k} x^{n-k} + \cdots + g_1 x + g_0. 
\] (2.38)

The properties of the generator polynomial \(g(x)\) are \(^1\),
1) There exists one code polynomial with the minimum order \((n-k)\) in a cyclic code;

2) In a cyclic code, \(g(x)\) is a polynomial with order \((n-k)\) and it is exclusive;

3) In a cyclic code, the code polynomial \(c(x)\) is the multiple of \(g(x)\).

The encoding of a cyclic code is most commonly achieved by using the generator matrix \(g(x)\). The encoding steps of an \((n, k)\) cyclic code are as follows\(^3\),

1) Multiply the message bits \(m(x)\) by \(x^{n-k}\);

2) Divide \(m(x)x^{n-k}\) by \(g(x)\) to get the remainder \(r(x)\), i.e.,

\[
r(x) = m(x)x^{n-k}\mod g(x);
\]  \(2.39\)

3) Obtain the code word \(c(x)\) from \(m(x)\) and \(r(x)\), i.e.,

\[
c(x) = m(x)x^{n-k} + r(x).
\]  \(2.40\)

The decoding of a cyclic code is also based on the syndromes. Because of the cyclic property of a cyclic code, the decoding is easier to accomplish than that of a general linear block code.

The syndrome \(s(x)\) of a cyclic code is the remainder obtained from dividing either the code polynomial \(r(x)\) or the error pattern polynomial \(e(x)\) by the generator polynomial \(g(x)\). The decoding steps of an \((n, k)\) cyclic code are\(^3\),

1) Find the syndrome \(s(x)\),

\[
s(x) = r(x)\mod g(x),
\]  \(2.41\)

or

\[
s(x) = e(x)\mod g(x);
\]  \(2.42\)

2) Find the corresponding error patterns according to the syndromes;

3) Perform error correction based on the error patterns.

### 2.3.3 BCH Code
One of the most important cyclic codes is BCH code. It is discovered by and named after three scholars Hocquenghem, Bose, and Chaudhuri. BCH code can correct multiple random errors and the code can be either binary or non-binary.

The cyclic code is called a BCH code, if the generator polynomial is of the following form \[^1\],

\[ g(x) = \text{LCM}[m_1(x), m_3(x), \ldots, m_{2t-1}(x)], \quad (2.43) \]

in which \( t \) is the error correcting capability; \( m_i(x) \) is the minimal polynomial; and LCM stands for least common multiple. It can correct \( t \) random errors. The minimum distance is \( d_{\text{min}} \geq 2t+1 \).

A primitive BCH code is a BCH code defined using a primitive element \( \alpha \). The parameters of a binary primitive BCH code are shown below,

\[ n = 2^m - 1 \]
\[ n - k \leq mt \]
\[ d_{\text{min}} = 2t + 1, \quad (2.44) \]

in which \( m \) and \( t \) are positive integers; and \( m \geq 3, \ t < 2^{m-1} \).

The procedures for determining a BCH code with length \( n = p^m - 1 \), primitive element \( \alpha \), power \( m \), and error correcting capability \( t \) are,

1) Find a primitive polynomial of power \( m \) \( p(x) \) to construct field \( \text{GF}(p^m) \);
2) Obtain the minimal polynomial \( m_i(x) \), where \( i = 1, 3, \ldots, 2t-1 \);
3) Obtain the generator polynomial \( g(x) \)

\[ g(x) = \text{LCM}[m_1(x), m_3(x), \ldots, m_{2t-1}(x)]; \quad (2.45) \]

4) Fix \( k \) according to the power of \( g(x) \), \( (n-k) \leq mt \);
5) Find the minimum distance \( d_{\text{min}} \geq 2t+1 \).
Let the generator polynomial of a BCH code be \( g(x) \), then its roots
\( (\beta, \beta^2, ..., \beta^{2t}) \in GF(2^m) \) are certainly the roots of the code word polynomial
\[ C(x)=c_nx^{n-1}+c_{n-1}x^{n-2} + \cdots + c_2x+c_1, \]
i.e.,
\[ c_n(\beta^i)^{n-1}+c_{n-1}(\beta^i)^{n-2} + \cdots + c_2(\beta^i)+c_1 = 0, \quad (2.46) \]
where \( i = 1, 2, \ldots, 2t \). Its matrix form is as below,
\[
\begin{bmatrix}
\beta^{n-1} & \beta^{n-2} & \cdots & 1 \\
(\beta^2)^{n-1} & (\beta^2)^{n-2} & \cdots & 1 \\
\vdots & \vdots & \ddots & \vdots \\
(\beta^{2t})^{n-1} & (\beta^{2t})^{n-2} & \cdots & 1 \\
\end{bmatrix}
\begin{bmatrix}
c_n \\
c_{n-1} \\
\vdots \\
c_1 \\
\end{bmatrix}
= 0. \quad (2.47)
\]
The above equation is the matrix form definition of the BCH code, i.e., \( HC^T = 0 \).

H can be simplified as
\[
H = \begin{bmatrix}
\beta^{n-1} & \beta^{n-2} & \cdots & 1 \\
(\beta^3)^{n-1} & (\beta^3)^{n-2} & \cdots & 1 \\
\vdots & \vdots & \ddots & \vdots \\
(\beta^{2t-1})^{n-1} & (\beta^{2t-1})^{n-2} & \cdots & 1 \\
\end{bmatrix}. \quad (2.48)
\]

2.3.4 **Reed-Solomon Code**

A q-element BCH code with length \( n=q-1 \) is called a Reed-Solomon code \(^{[1]} \). When \( q=2^m \), the symbols are on finite field \( GF(2^m) \). In the code RS(n, k), the input sequence is divided into blocks with \( k\times m \) bits in each block. In each block, there are \( k \) symbols with \( m \) bits in each symbol.

The parameters of a t-error correcting RS code are shown in table 2.3.
Table 2.3 RS Code Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Symbols</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Bits</td>
<td>$k$</td>
<td>$mk$</td>
</tr>
<tr>
<td>Code Length $n$</td>
<td>$q-1=2^m-1$</td>
<td>$(2^m-1)×m$</td>
</tr>
<tr>
<td>Parity Check Bits</td>
<td>$n-k=2t$</td>
<td>$(n-k)×m=2mt$</td>
</tr>
<tr>
<td>Minimum Distance $d_{\text{min}}$</td>
<td>$2t+1$</td>
<td>$M(2t+1)$</td>
</tr>
</tbody>
</table>

The roots of the generator polynomial $g(x)$ are on the same field as the symbols.

The minimum distance $d_{\text{min}}=2t+1$ of a RS code is the largest minimum Hamming distance in all of the linear block codes. Therefore, RS codes have the strongest error correcting capability.

In field $GF(2^m)$, the generator polynomial of a $t$-error correcting RS code is \[^{[1]}\]

$$g(x) = (x - \alpha)(x - \alpha^2) \ldots (x - \alpha^{2t}), \quad (2.49)$$

in which, $\alpha^i \in GF(2^m)$, $i=1, 2, \ldots, 2t$.

RS codes are very adaptive to correct burst errors. The error patterns it can correct include:

One-bit burst errors with the total length of $b_1 = (t - 1)m + 1$.

Two-bit burst errors with the total length of $b_2 = (t - 3)m + 3$.

$i$-bit burst errors with the total length of $b_i = (t - 2i + 1)m + 2i - 1$.

The encoding process of RS code is similar as the one of BCH code. It can be achieved using feedback shift register. The difference is that, in RS code, the width of the data passage is $m$ bits, i.e. shift registers are parallel and of order $m$.

2.4 Conclusion of the chapter
This chapter introduced the basic concepts of coding theory. The discussion mainly focused on the fundamental principles of coding theory, the theory of error correction coding, the theory of finite field, and several important linear block codes.
Chapter 3 Design of Reed-Solomon Encoder

3.1 RS Encoding Algorithm

The encoding procedures of RS codes are similar as the encoding of general cyclic codes.

Let the message bits $M$ be

$$M = (m_k, m_{k-1}, ..., m_1). \quad (3.1)$$

The information polynomial $m(x)$ is

$$m(x) = m_kx^{k-1} + m_{k-1}x^{k-2} + \cdots + m_2x + m_1. \quad (3.2)$$

The code word $C$ is

$$C = (c_n, c_{n-1}, ..., c_1). \quad (3.3)$$

The code word polynomial $c(x)$ of the RS code is

$$c(x) = c_nx^{n-1} + c_{n-1}x^{n-2} + \cdots + c_2x + c_1. \quad (3.4)$$

From the discussion in the previous chapter, the generator matrix $g(x)$ of the RS code is

$$g(x) = (x - \alpha)(x - \alpha^2) \cdots (x - \alpha^{2t}). \quad (3.5)$$

The encoding steps are

1) Multiply the information polynomial $m(x)$ by $x^{n-k}$;

2) Divide $m(x)x^{n-k}$ by $g(x)$ to obtain the remainder $r(x)$, i.e.

$$r(x) = m(x)x^{n-k} \mod g(x); \quad (3.6)$$

3) Derive code word polynomial $c(x)$ through $m(x)$ and $r(x)$, i.e.
Through the above three steps, the code word of a RS code is generated as

\[ c(x) = m(x)x^{n-k} + r(x). \]  

(3.7)

The first k symbols are information symbols and the last r=n-k symbols are parity check symbols. This algorithm is called the time-domain algorithm of RS code and the code obtained is called systematic code.

Take the octal RS(7,3) code as an example. The elements of RS(7, 3) code are on field GF(2^3). Its primitive polynomial is

\[ p(x) = x^3 + x + 1. \]  

(3.9)

The eight elements on Galois field GF(2^3) are shown in the table below.

<table>
<thead>
<tr>
<th>Power ( \alpha^i )</th>
<th>Polynomial of ( \alpha )</th>
<th>Binary Form</th>
<th>Decimal Form</th>
<th>Minimal Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \alpha^0 )</td>
<td>1</td>
<td>001</td>
<td>1</td>
<td>( x+1 )</td>
</tr>
<tr>
<td>( \alpha^1 )</td>
<td>( \alpha )</td>
<td>010</td>
<td>2</td>
<td>( x^3+x+1 )</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>( \alpha^2 )</td>
<td>100</td>
<td>4</td>
<td>( x^3+x+1 )</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>( \alpha+1 )</td>
<td>011</td>
<td>3</td>
<td>( x^3+x^2+1 )</td>
</tr>
<tr>
<td>( \alpha^4 )</td>
<td>( \alpha^2+\alpha )</td>
<td>110</td>
<td>6</td>
<td>( x^3+x+1 )</td>
</tr>
<tr>
<td>( \alpha^5 )</td>
<td>( \alpha^2+\alpha+1 )</td>
<td>111</td>
<td>7</td>
<td>( x^3+x^2+1 )</td>
</tr>
<tr>
<td>( \alpha^6 )</td>
<td>( \alpha^2+1 )</td>
<td>101</td>
<td>5</td>
<td>( x^3+x^2+1 )</td>
</tr>
</tbody>
</table>

Let the input information symbols be \( M=(\alpha^3, \alpha, 1) \), then the information polynomial is
\[ m(x) = \alpha^3 x^2 + \alpha x + 1. \] (3.10)

The code length of this RS code is \( n=7 \); number of information symbols is \( k=3 \); and the error correcting capability is \( t=(7-3)/2=2 \). Therefore, the generator polynomial \( g(x) \) is

\[
g(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^3)(x - \alpha^4) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3. \tag{3.11}
\]

\( m(x)x^{n-k} \) is

\[
x^{n-k}m(x) = x^4(\alpha^3 x^2 + \alpha x + 1) = \alpha^3 x^6 + \alpha x^5 + x^4. \tag{3.12}
\]

Divide \( m(x)x^{n-k} \) by \( g(x) \) to get the remainder \( r(x) \)

\[
r(x) = m(x)x^{n-k}\mod g(x) = x^3 + \alpha x. \tag{3.13}
\]

Then, the code word polynomial \( c(x) \) can be derived

\[
c(x) = \alpha^3 x^6 + \alpha x^5 + x^4 + x^3 + \alpha x = \alpha^3 x^6 + \alpha x^5 + x^4 + x^3 + 0 \cdot x^2 + \alpha x + 0. \tag{3.14}
\]

i.e. code word \( C=(\alpha^3, \alpha, 1, 1, 0, \alpha, 0) \).

### 3.2 Arithmetic Realization on the Finite Field

Both the encoding and decoding arithmetics of a RS code are performed on the finite field. The arithmetics include addition, subtraction, multiplication, and division. These operations are the basic units of the RS encoding and decoding systems.

#### 3.2.1 The Elements on \( \text{GF}(2^8) \)

The elements of RS code discussed in this paper are on the field \( \text{GF}(2^8) \). There are \( 2^8=256 \) elements on \( \text{GF}(2^8) \), among which 255 elements are non-zero. The primitive polynomial on \( \text{GF}(2^8) \) is \( p(x)=x^8+x^4+x^3+x^2+1 \). From the primitive polynomial \( p(a)=a^8+a^4+a^3+a^2+1=0 \), the elements with order greater than “7” can be derived,
\[ \alpha^8 = \alpha^4 + \alpha^3 + \alpha^2 + 1; \]
\[ \alpha^9 = \alpha(\alpha^8) = \alpha(\alpha^4 + \alpha^3 + \alpha^2 + 1) \]
\[ = \alpha^5 + \alpha^4 + \alpha^3 + \alpha; \]
\[ \alpha^{10} = \alpha^6 + \alpha^5 + \alpha^4 + \alpha^2; \]
\[ \alpha^{11} = \alpha^7 + \alpha^6 + \alpha^5 + \alpha^3; \]
\[ \alpha^{12} = \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 \]
\[ = (\alpha^4 + \alpha^3 + \alpha^2 + 1) + \alpha^7 + \alpha^6 + \alpha^4 \]
\[ = \alpha^7 + \alpha^6 + \alpha^3 + \alpha^2 + 1; \]
\[ \ldots \]
\[ \alpha^{253} = \alpha^6 + \alpha^2 + \alpha + 1; \]
\[ \alpha^{254} = \alpha^7 + \alpha^3 + \alpha^2 + \alpha. \]

The 256 elements on field GF(2^8) are shown in Table 3.2 \[3\].

Table 3.2 Elements on Field GF(2^8)

<table>
<thead>
<tr>
<th>Power ( \alpha^i )</th>
<th>Polynomial Form</th>
<th>Binary Form</th>
<th>Decimal Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>( \alpha^0 )</td>
<td>1</td>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>( \alpha^1 )</td>
<td>( \alpha )</td>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>( \alpha^2 )</td>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>( \alpha^3 )</td>
<td>00001000</td>
<td>8</td>
</tr>
<tr>
<td>( \alpha^4 )</td>
<td>( \alpha^4 )</td>
<td>00010000</td>
<td>16</td>
</tr>
<tr>
<td>( \alpha^5 )</td>
<td>( \alpha^5 )</td>
<td>00100000</td>
<td>32</td>
</tr>
<tr>
<td>( \alpha^6 )</td>
<td>( \alpha^6 )</td>
<td>01000000</td>
<td>64</td>
</tr>
</tbody>
</table>
### 3.2.2 Addition and Subtraction on Finite Field

Since there is no carry-bit, the addition on a finite field is very simple. It is easier to realize than a normal addition. For the two-element addition on GF($2^m$), it can be achieved by bitwise XOR using only registers and XOR gates. On finite field GF($2^8$), implementing bitwise XOR on the 8-bit binary numbers will achieve the addition of the two numbers.

The block diagram of an adder is demonstrated as in figure 3.1. In the block diagram, a and b are both 8-bit input signals; c is an 8-bit output signal.

| $\alpha^7$ | $\alpha^7$ | 10000000 | 128 |
| $\alpha^8$ | $\alpha^4+\alpha^3+\alpha^2+1$ | 00011101 | 29 |
| $\alpha^9$ | $\alpha^5+\alpha^4+\alpha$ | 00111010 | 58 |
| $\alpha^{10}$ | $\alpha^6+\alpha^5+\alpha^4+\alpha^2$ | 01110100 | 116 |
| $\alpha^{11}$ | $\alpha^7+\alpha^6+\alpha^5+\alpha^3$ | 11101000 | 232 |
| $\alpha^{12}$ | $\alpha^7+\alpha^6+\alpha^3+\alpha^2+1$ | 11001101 | 205 |
| $\alpha^{13}$ | $\alpha^7+\alpha^2+\alpha+1$ | 10000111 | 135 |
| $\alpha^{14}$ | $\alpha^4+\alpha+1$ | 00010011 | 19 |
| ... | ... | ... | ... |
| $\alpha^{253}$ | $\alpha^6+\alpha^2+\alpha+1$ | 01000111 | 71 |
| $\alpha^{254}$ | $\alpha^7+\alpha^3+\alpha^2+\alpha$ | 10001110 | 142 |
For the subtraction on a finite field, since it is binary operation, subtraction is the same as addition, and the subtracrer circuit is the same as the adder circuit.

### 3.2.3 Multiplication on a Finite Field

Multiplication on a finite field is one of the most important arithmetics in RS codes. Compared with addition and subtraction, multiplication is more complicated and demands a very large time delay.

The resources multipliers occupy and their operational speed have a great impact on the performance of a RS encoder and decoder system, since the processes of RS encoding and decoding involves many multiplications.

The most common multiplications include algorithm based on polynomials, algorithm based on linear feedback shift register, Berlekamp algorithm, look-up table, and Messey-Omura algorithm.

From the basis point of view, multipliers can be divided into three types: natural basis multipliers, dual basis multipliers, and standard basis multipliers. They utilize the special algebraic properties of finite field to simplify the algorithm of multipliers. The RS(204, 188) code discussed in this dissertation adopts natural basis multipliers. Natural basis multipliers include bit-serial multipliers and bit-parallel multipliers. In this paper, bit-parallel multipliers are employed for their small time delay and high throughput rate.
3.2.4 Division on a finite field

Another arithmetic commonly used on finite fields is division. For any two elements on a finite field a and b, the outcome of their division is $c = a/b$. The structure of the circuit will be fairly complicated if division is performed directly. A common approach to simulate division is to take inverse and implement multiplication. More specifically, the inverse $b^{-1}$ of b is calculated first, then multiply it by a to obtain c. This method is much simpler than directly executing division.

3.3 Hardware Realization of Reed-Solomon Encoders

The hardware realization of a RS encoder is not complicated. Its main framework is a circuit of mod $g(x)$ division. The circuit consists of finite field adders, multipliers, feedback linear shift registers, and switches. Its logic circuit is shown in figure 3.2.

![Logical Circuit Diagram of the Encoder](image)

Figure 3.2 Logical Circuit Diagram of the Encoder

The working procedure of the circuit is described below.

1) The initial states of the registers $R_0$~$R_{2t-1}$ are all 0. At the beginning, the switch is on 1. At this moment, the input sequence $m(x)$ is transmitted to two destinations, one to the output, and the other, after multiplied by $x^{n-k}$, to the division circuit.
2) After k clock cycles, the whole message sequence m(x) is sent in the circuit and the division arithmetic is executed. The values saved in the registers are the coefficients of the remainder r(x), i.e. the 2t parity check symbols.

3) At the k+1 clock cycle, the switch is switched to position 2. Then, the data in the registers shift out one by one. After 2t=n-k clock cycles, all of the data are transmitted to the output end and form the code word c(x) along with the k information symbols of the message sequence m(x). This is the encoding of one block of code.

4) Reset all registers R₀~R₂t-1 to the initial states 0 and repeat the above procedures to perform the encoding of the next block of code.

3.4 Encoding of (204, 188) Reed-Solomon Code

(204, 188) Reed-Solomon code is a truncated version of RS(255, 239) code. It deletes the first 51 symbols of RS(255, 239) code since they are all zeros. Therefore, their design and realization methods are similar. RS(204, 188) code is defined on Galois field GF(2⁸), with code length n = 204 and information bits k = 188.

3.4.1 Characteristics of RS(204, 188) Code

The characteristics of RS(204, 188) code discussed in this paper are as below.

Code length: n = 204.

Information symbols: k = 188.

Parity check symbols: r = n – k =16.

Minimum distance: d_{min} = n – k + 1 = 17.

Error correcting capability: t = 8.

3.4.2 Construction of GF(2⁸)

The primitive polynomial of GF(2⁸) is given by
\[ p(X) = 1 + X^2 + X^3 + X^4 + X^8. \]  

(3.17)

Let \( \alpha \) be the primitive element on \( \text{GF}(2^8) \) and a root of \( p(x) \), then it can be derived that

\[ 1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8 = 0, \]  

(3.18)

or

\[ 1 + \alpha^2 + \alpha^3 + \alpha^4 = \alpha^8. \]  

(3.19)

The generator polynomial of RS(204, 188) code is given by

\[
g(X) = (X + \alpha)(X + \alpha^2)(X + \alpha^3)(X + \alpha^4)(X + \alpha^5)(X + \alpha^6)
\]

\[
(X + \alpha^7)(X + \alpha^8)(X + \alpha^9)(X + \alpha^{10})(X + \alpha^{11})(X + \alpha^{12})
\]

\[
(X + \alpha^{13})(X + \alpha^{14})(X + \alpha^{15})(X + \alpha^{16}).
\]  

(3.20)

The MATLAB program in Appendix C is used to calculate the coefficients. It can be derived that

\[
g(X) = 59 + 36X + 50X^2 + 98X^3 + 229X^4 + 41X^5 + 65X^6
\]

\[
+ 163X^7 + 8X^8 + 30X^9 + 209X^{10} + 68X^{11} + 189X^{12}
\]

\[
+ 104X^{13} + 13X^{14} + 59X^{15} + X^{16}.
\]  

(3.21)

To be clearer, the coefficients are listed below,

Table 3.3 Coefficients of \( g(x) \)

| \( g_0 = \alpha^{18} = 59 \) | \( g_1 = \alpha^{251} = 36 \) | \( g_2 = \alpha^{215} = 50 \) | \( g_3 = \alpha^{28} = 98 \) |
| \( g_4 = \alpha^{80} = 229 \) | \( g_5 = \alpha^{107} = 41 \) | \( g_6 = \alpha^{248} = 65 \) | \( g_7 = \alpha^{53} = 163 \) |
| \( g_8 = \alpha^{94} = 8 \) | \( g_9 = \alpha^{194} = 30 \) | \( g_{10} = \alpha^{91} = 209 \) | \( g_{11} = \alpha^{59} = 68 \) |
| \( g_{12} = \alpha^{176} = 189 \) | \( g_{13} = \alpha^{99} = 104 \) | \( g_{14} = \alpha^{203} = 13 \) | \( g_{15} = \alpha^{137} = 59 \) |
| \( g_{16} = \alpha^{43} = 1 \) |

3.4.3 Constant Multiplier under Natural Basis
In Reed-Solomon codes, in addition to the multiplication of two variables, there is another situation where a variable multiplies a constant. Based on the specialties of the elements on a finite field, the multiplication can be converted into a simple combinational circuit.

Since the primitive polynomial of GF($2^8$) is $p(x) = x^8 + x^4 + x^3 + x^2 + 1$ and any element $a$ is

$$a = a_0 + a_1 \alpha + a_2 \alpha^2 + \cdots + a_7 \alpha^7,$$  \hfill (3.22)

from table 3.2, it can be derived that

$$a \cdot \alpha = a_0 \alpha + a_1 \alpha^2 + a_2 \alpha^3 + a_3 \alpha^4 + a_4 \alpha^5 + a_5 \alpha^6 + a_6 \alpha^7 + a_7 \alpha^8$$

$$= a_0 \alpha + a_1 \alpha^2 + a_2 \alpha^3 + a_3 \alpha^4 + a_4 \alpha^5 + a_5 \alpha^6 + a_6 \alpha^7$$

$$+ a_7 (\alpha^4 + \alpha^3 + \alpha^2 + 1)$$

$$= a_7 + a_0 \alpha + (a_1 + a_7) \alpha^2 + (a_2 + a_7) \alpha^3 + (a_3 + a_7) \alpha^4 + a_4 \alpha^5 + a_5 \alpha^6$$

$$+ a_6 \alpha^7$$

$$a \cdot \alpha^2 = a_0 + a_7 \alpha + (a_0 + a_6) \alpha^2 + (a_1 + a_6 + a_7) \alpha^3$$

$$+ (a_2 + a_6 + a_7) \alpha^4 + (a_3 + a_7) \alpha^5 + a_4 \alpha^6 + a_5 \alpha^7$$

$$a \cdot \alpha^3 = a_5 + a_6 \alpha + (a_5 + a_7) \alpha^2 + (a_0 + a_5 + a_6) \alpha^3 + (a_1 + a_5 + a_6 + a_7) \alpha^4$$

$$+ (a_2 + a_6 + a_7) \alpha^5 + (a_4 + a_3) \alpha^6 + a_4 \alpha^7$$

$$\cdots$$

$$a \cdot \alpha^7 = (a_1 + a_5 + a_6 + a_7) + (a_2 + a_6 + a_7) \alpha + (a_1 + a_3 + a_5 + a_6) \alpha^2$$

$$+ (a_1 + a_2 + a_4 + a_5) \alpha^3 + (a_1 + a_2 + a_3 + a_7) \alpha^4 + (a_2 + a_3 + a_4) \alpha^5$$

$$+ (a_3 + a_4 + a_5) \alpha^6 + (a_0 + a_4 + a_5 + a_6) \alpha^7.$$  \hfill (3.23)

The multiplication of any element on GF($2^8$) with constants, $1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7$, can be transformed into the combinational circuit with mod 2 adders as the basic elements. It is more convenient to calculate which results in the reduction of chips and the acceleration of computation. The encoder designed in this paper used constant multipliers.
Below is the block diagram of the constant multiplier $g_{\text{mult}0}$, in which $x$ is 8-bit input data and $y$ is 8-bit output data. Other constant multipliers are of the same form.

![Block Diagram of Multiplier $g_{\text{mult}0}$](image)

**Figure 3.3 Block Diagram of Multiplier $g_{\text{mult}0}$**

Top level block diagram of the encoder is shown in figure 3.4, in which $\text{clk}$ is the clock signal; $x$ is the 8-bit input data; enable is the control signal which enables the encoder on high; $\text{clrn}$ is the reset signal, $y$ is the 8-bit output data; $\text{cnt}$ is the counter.

![Top Level Block Diagram of the Encoder](image)

**Figure 3.4 Top Level Block Diagram of the Encoder**

The RTL description of the encoder is included in Appendix A.
Chapter 4 Design of Reed-Solomon Decoder

4.1 Procedures of RS Decoding

The code word $c(x)$ obtained after encoding will be interfered when transmitted in the channel. Let the noise be $e(x)$, i.e. error pattern. It will be added onto $c(x)$ and the sum will be transmitted to the receiver. Let the polynomial of the received vector be $r(x)$, then $r(x) = c(x) + e(x)$. The expressions of $c(x)$, $e(x)$, and $r(x)$ are given below

\begin{align*}
    c(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} \quad (4.1) \\
    e(x) &= e + e_1 x + e_2 x^2 + \cdots + e_{n-1} x^{n-1} \quad (4.2) \\
    r(x) &= r_0 + r_1 x + r_2 x^2 + \cdots + r_{n-1} x^{n-1}. \quad (4.3)
\end{align*}

The basic decoding theory of a RS code is to derive the error pattern $e(x)$ based on the received vector $r(x)$, and then obtain code word $c(x)$ according to equation $r(x) = c(x) + e(x)$.

Decoding methods can be divided into two categories. One is time domain decoding and the other is frequency domain decoding. Time domain decoding is to locate the errors based on the received vector and it does not need transform calculation. Frequency domain decoding is to obtain the error location by perform Fourier transform. Since it is easier to apply, this dissertation employs time domain decoding.

The steps of time domain decoding are described as below \cite{1}\cite{10}:

1) Find out the syndrome $s(x)$ and the key equation using the received signal $r(x)$.

2) Determine the error location polynomial $\sigma(x)$ and the error-value evaluator polynomial $\omega(x)$ based on the syndrome $s(x)$.

3) Determine the error location number $x_i$ using Chien search.

4) Evaluate error-value polynomial $y_i$ according to $\sigma(x)$ and $x_i$. 

38
5) Obtain the information sequence based on r(x) and y_i.

### 4.2 Calculation of the Syndrome

From Chapter 2, syndrome is defined as:

$$ S = RH^T. \quad (4.4) $$

Let the syndrome polynomial be:

$$ s(x) = s_1 + s_2 x + s_3 x^2 + \cdots + s_{2t} x^{2t-1}. \quad (4.5) $$

Calculate the coefficients $s_i$ ($i = 1, 2, \ldots, 2t$) of $s(x)$ based on the polynomial $r(x)$ of the received vector, i.e. substitute $\alpha^i$ ($I = 1, 2, \ldots, 2t$) into $r(x)$, and it can be derived that:

$$ s_i = r(\alpha^i) = \sum_{j=0}^{n-1} r_j (\alpha^i)^j $$

$$ = r_0 + r_1 \alpha^i + r_2 (\alpha^i)^2 + \cdots + r_{n-1} (\alpha^i)^{n-1} $$

$$ = r_0 + \alpha^i (r_1 + \cdots + \alpha^i (r_{n-3} + \alpha^i (r_{n-2} + \alpha^i r_{n-1}))) \ldots. \quad (4.6) $$

If $s_i = 0$, the transmission is error-free; if $s_i \neq 0$, errors occurred in the transmission, and the error pattern needs to be determined to perform error correction. The block diagram of solving syndrome is shown in figure 4.1 [10].

![Figure 4.1 Syndrome Calculation Block Diagram](image)

Since $c(\alpha^i) = 0$, the following equation can be obtained:
\[ s_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) \]
\[ = e(\alpha^i) = \sum_{j=0}^{n-1} e_j(\alpha^i)^j \]
\[ = e_0 + e_1\alpha^i + e_2(\alpha^i)^2 + \cdots + e_{n-1}(\alpha^i)^{n-1}. \quad (4.7) \]

Let the error pattern \( e(x) \) which contains \( e \) errors be of the form:
\[ e(x) = y_1x_1 + y_2x_2 + \cdots + y_{e-1}x_{e-1} + y_ex_e, \quad (4.8) \]
in which \( y_i \) represents the value of the \( i^{th} \) error and \( x_i \) indicates the location of the \( i^{th} \) error.

Let \( (\alpha^i)^j = (\alpha^j)^i \), then it can be calculated that
\[ s_i = \sum_{j=1}^{t} y_jx_j^i, \quad (4.9) \]
where \( i = 1, 2, \ldots, 2t \).

Hence, the following equation set can be derived:
\[
\begin{align*}
\sum_{i=1}^{t} y_1x_1^i + y_2x_2^i + \cdots + y_{e-1}x_{e-1}^i + y_ex_e^i &= s_1 \\
\sum_{i=1}^{t} y_1(x_1)^i + y_2(x_2)^i + \cdots + y_{e-1}(x_{e-1})^i &= s_2 \\
\vdots \\
\sum_{i=1}^{t} y_1(x_1)^{2t} + y_2(x_2)^{2t} + \cdots + y_{e-1}(x_{e-1})^{2t} &= s_{2t}
\end{align*}
\quad (4.10)
\]

The decoding procedure is to solving equation set (4.10).

4.3 Solving the Error Location Polynomial

4.3.1 Key Equation

Equation set (4.10) is a non-linear equation set and it is too complicated to solve directly. Therefore, an indirect method is adopted to transform the equation set into a linear equation to determine the error location \( x_i \) and the error value \( y_i \).

Let the error location polynomial \( \sigma(x) \) be of the following form:
\[ \sigma(x) = \prod_{i=1}^{t}(1 - xx_i) \]
\[ = (1 - xx_1)(1 - xx_2) \cdots (1 - xx_t) \]  
\[ = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t. \] (4.11)

Assume that the location \( x \) of the \( i^{th} \) error is \( x^{-1}_i \), then

\[ \sigma(x^{-1}_i) = \prod_{i=1}^{t}(1 - x^{-1}_i x_i) = 0 \] (4.12)

Hence, once the roots of equation \( \sigma(x) = 0 \) are solved, the error locations are determined.

Let \( x = x^{-1}_i \), i.e.,

\[ \sigma(x^{-1}_i) = 1 + \sigma_1 x^{-1}_i + \sigma_2 (x^{-1}_i)^2 + \cdots + \sigma_t (x^{-1}_i)^t \]
\[ = 1 + \sigma_1 x^{-1}_i + \sigma_2 x^{-2}_i + \cdots + \sigma_t x^{-t}_i = 0, \] (4.13)

where \( i = 1, 2, \ldots, t \). Multiply the above equation by \( x^t_i \), then it follows that

\[ x^t_i + \sigma_1 x^{t-1}_i + \sigma_2 x^{t-2}_i + \cdots + \sigma_{t-1} x_i + \sigma_t = 0. \] (4.14)

Multiply the above equation by \( y_i x^j_i \) (\( j = 1, 2, \ldots, t \)), then it follows that

\[ y_i x^{j+t}_i + \sigma_1 y_i x^{j+t-1}_i + \cdots + \sigma_{t-1} y_i x^{j+1}_i + \sigma_t y_i x^j_i = 0. \] (4.15)

Take the summation of the above equation, then it follows that

\[ \sum_{i=1}^{t} y_i x^{j+t}_i + \sigma_1 \sum_{i=1}^{t} y_i x^{j+t-1}_i + \cdots + \sigma_t \sum_{i=1}^{t} y_i x^j_i = 0 \] (4.16)

Based on equation (4.9), it can be derived that

\[ \sum_{i=1}^{t} y_i x^{j+t}_i = s_{j+t}. \] (4.17)

Therefore, equation (4.16) can be converted to

\[ s_{j+t} + \sigma_1 s_{j+t-1} + \sigma_2 s_{j+t-2} + \cdots + \sigma_t s_j = 0, \] (4.18)

in which \( j = 1, 2, \ldots, t \).
Expand it, then it follows that

\begin{align*}
\begin{cases}
\sigma_1 s_t + \sigma_2 s_{t-1} + \cdots + \sigma_t s_1 = -s_{t+1} \\
\sigma_1 s_{t+1} + \sigma_2 s_t + \cdots + \sigma_t s_2 = -s_{t+2} \\
\sigma_1 s_{2t-1} + \sigma_2 s_{2t-2} + \cdots + \sigma_t s_t = -s_{2t}
\end{cases}
\end{align*}

(4.19)

Through this equation set, the value of \( \sigma_i \) can be obtained.

Let \( s(x) \), \( \sigma(x) \), and \( \omega(x) \) respectively be

\[ s(x) = s_0 + s_1 x + s_2 x^2 + \cdots = \sum_{i=0}^{\infty} s_i x^i, \]  \hspace{1cm} (4.20)

\[ \sigma(x) = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t, \]  \hspace{1cm} (4.21)

\[ \omega(x) = s(x) \sigma(x) = 1 + \omega_1 x + \omega_2 x^2 + \cdots, \]  \hspace{1cm} (4.22)

in which \( s_0 = 1 \). Then it can be derived that

\[ s(x) \sigma(x) = s_0 + (s_1 + s_0 \sigma_1) x + (s_2 + s_1 \sigma_1 + s_0 \sigma_2) x^2 \\
+ \cdots + (s_{t+1} + s_t \sigma_1 + s_{t-1} \sigma_2 + \cdots + s_1 \sigma_t) x^t \\
+ (s_{t+2} + s_{t+1} \sigma_1 + s_t \sigma_2 + \cdots + s_2 \sigma_t) x^{t+1} \\
+ \cdots \]

\[ = \omega(x) \\
= 1 + \omega_1 x + \omega_2 x^2 + \cdots + \omega_t x^t + \omega_{t+1} x^{t+1} + \cdots. \]  \hspace{1cm} (4.23)

From equation (4.19) it can be seen that among all the coefficients of equation (4.23), the ones with power greater than \( x^t \) are all zeros, and the highest power of \( x \) is \( t \) in \( \sigma(x) \). Therefore, the highest power of \( x \) in \( \omega(x) \) is less than \( t \). Hence, in the expression of \( s(x) \), the highest power of \( x \) is \( x^t \). Based on the fact that the highest power of the multiplication of \( s(x) \) and \( \sigma(x) \) is less than \( 2t + 1 \), it follows that \[ [1][3]

\[ s(x) \sigma(x) = \omega(x) \text{mod} x^{2t+1}. \]  \hspace{1cm} (4.24)

Equation (4.24) is called solving the key equation for \( \sigma(x) \). \( \omega(x) \) is called the error-value evaluator polynomial. Solving the key equation is the emphasis of decoding.

**4.3.2 Berlekamp-Massey Algorithm**
The process of directly solving key equation is very complicated. Therefore, Berlekamp-Massey iteration algorithm is employed to simplify the procedure.

The process of iteration is \[^{[7]}\]: firstly set a group of initial value \(\sigma^0(x)\) and \(\omega^0(x)\), and then represent \(\sigma^1(x)\) and \(\omega^1(x)\) using \(\sigma^0(x)\) and \(\omega^0(x)\) through one iteration; next, perform the second iteration to get \(\sigma^2(x)\) and \(\omega^2(x)\); continue iterating until \(\sigma^{i+1}(x)\) and \(\omega^{i+1}(x)\) are received.

In BM iteration \[^{[7]}[^{[8]}\], firstly define two variable \(D(j)\) and \(d_j\), where \(D(j)\) indicates the lowest power of \(\sigma^j(x)\) obtained from the \((j+1)\)th iteration, and \(d_j\) represents the difference between the \((j+1)\)th and \(j\)th iterations if \(\sigma^j(x)\) and \(\omega^j(x)\) do not satisfy the key equation any more. It satisfies the equation below:

\[
s(x)\sigma^j(x) = \omega^j(x) + d_jx^{j+1}\mod x^{j+2}. \tag{4.25}
\]

The left-hand side of this equation can be expanded as:

\[
(s_0 + s_1x + s_2x^2 + \cdots)(1 + \sigma_1^jx + \sigma_2^jx^2 + \cdots) = s_0 + (s_1 + s_0\sigma_1^j)x + (s_2 + s_1\sigma_1^j + s_0\sigma_2^j)x^2 + \cdots.
\]

\[
+ \cdots + (s_j + s_{j-1}\sigma_1^j + s_{j-2}\sigma_2^j + \cdots + s_0\sigma_j^j)x^j + (s_{j+1} + s_j\sigma_1^j + s_{j-1}\sigma_2^j + \cdots + s_0\sigma_{j+1}^j)x^{j+1}. \tag{4.26}
\]

Hence, the expression for \(d_j\) is:

\[
d_j = s_{j+1} + \sum_{i=1}^{\sigma^j(x)} s_{j+1-i}\sigma_i^j, \tag{4.27}
\]

in which \(\sigma_i^j\) is the coefficients of \(x^i\) in \(\sigma^j(x)\).

From equation (4.25) and (4.27) it follows that:

\[
\sigma^{j+1}(x) = \sigma^j(x) - d_j\sigma_i^{j-1}x^{j-i}\sigma^j(x). \tag{4.28}
\]

\[
\omega^{j+1}(x) = \omega^j(x) - d_j\omega_i^{j-1}x^{j-i}\omega^j(x). \tag{4.29}
\]

\(D(j+1)\) is:
Below is the illustration of Berlekamp-Massey algorithm\textsuperscript{[7][8]}:

1) Set initial values:

\[
\sigma^{-1}(x) = 1, \omega^{-1}(x) = 0, \sigma^0(x) = 1, \omega^0(x) = 1, \\
D(-1) = 0, d_{-1} = 1, D(0) = 0, d_0 = s_1.
\]  
\hspace{2cm} (4.31)

2) Determine \(d_j\) according to equation (4.27). There are two possibilities:

i. \(d_j = 0\), it gives that

\[
\sigma^{j+1}(x) = \sigma^j(x) \\
\omega^{j+1}(x) = \omega^j(x) \\
D(j + 1) = D(j).
\]  
\hspace{2cm} (4.32)

Then calculate \(d_{j+1}\) and perform the next iteration.

ii. \(d_j \neq 0\): firstly find the row \(i\) in front of row \(j\), the \(i-D(i)\) in row \(i\) is the greatest among all the rows in front of \(j\); then calculate \(\sigma^{j+1}(x)\) and \(\omega^{j+1}(x)\) based on equation (4.28) and (4.29) to solve for step \(j+1\).

3) Repeat the above procedures, and after \(2t\) iterations \(\sigma(x)\) and \(\omega(x)\) can be acquired.

4.3.3 An Improved Berlekamp-Massey Algorithm

The original BM algorithm needs to use the inverse operation and the calculation is very complex. In this paper, according to S. Reed and M.T. Shih’s paper \textit{VLSI Design of Inverse-Free Berlekamp-Massey Algorithm} published in 1991\textsuperscript{[14]}, an improved BM algorithm without inverse operations is adopted. The iteration theory of this algorithm is shown as below\textsuperscript{[14]}: 

Initial values:

\[
D(j + 1) = \max(D(j), j - i + D(j)).
\]  
\hspace{2cm} (4.30)
\[\sigma^0(x) = 1, \omega^0(x) = 1,\]
\[\lambda^0(x) = 1, \beta^0(x) = 1,\]  \hspace{1cm} (4.33)
\[l_0 = 0, \gamma_0 = 1\]

Iteration equation:

\[\delta_{k+1} = \sum_{j=0}^{l_k} s_{k+1-j}\sigma_j^k\]  \hspace{1cm} (4.34)

In which:

\[\sigma^{k+1}(x) = \gamma_k \omega^k(x) - \delta_{k+1}\lambda^k(x)x\]
\[\omega^{k+1}(x) = \gamma_k \sigma^k(x) - \delta_{k+1}\beta^k(x)x\]  \hspace{1cm} (4.35)

When \(\delta_{k+1} = 0\) or \(2l_k > k\):

\[\lambda^{k+1}(x) = x\lambda^k(x)\]
\[\beta^{k+1}(x) = x\beta^k(x)\]
\[l_{k+1} = l_k\]
\[\gamma_{k+1} = \gamma_k\]  \hspace{1cm} (4.36)

When \(\delta_{k+1} \neq 0\) and \(2l_k \leq k\):

\[\lambda^{k+1}(x) = \sigma^k(x)\]
\[\beta^{k+1}(x) = \omega^k(x)\]
\[l_{k+1} = k + 1 - l_k\]
\[\gamma_{k+1} = \delta_{k+1}\]  \hspace{1cm} (4.37)

### 4.4 Search for Error Location (Chien Search)

Once the error location polynomial \(\sigma(x)\) is solved, the error locations \(x_i\) can be determined using the roots of \(\sigma(x)\). R. T. Chien proposed a method to search for the error location and it is called Chien search \(^{[15]}\). It is an exhaustive searching algorithm which performs a bit-by-bit search for every location. Chien search has already become the standard algorithm to solve for the roots of \(\sigma(x)\).
The error location polynomial $\sigma(x)$ and the received vector polynomial $r(x)$ are respectively shown as below:

$$\sigma(x) = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t,$$

\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (4.38)

$$r(x) = r_0 + r_1 x + r_2 x^2 + \cdots + r_{n-1} x^{n-1}.$$

\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (4.39)

To confirm whether $r_i$ is corrupted, determining whether $\alpha^i$ is an error location number is needed, which is equivalent to determine whether $\alpha^{-i}$ is a root of $\sigma(x)$. Therefore, it is needed to determine whether the below equation is true:

$$\sigma(\alpha^{-i}) = \sigma_0 + \sigma_1 \alpha^{-i} + \sigma_2 \alpha^{-2i} + \cdots + \sigma_t \alpha^{-ti}. \hspace{1cm} (4.40)$$

If $\sigma(\alpha^{-i}) = 0$, $r_i$ contains error; otherwise $r_i$ is error-free.

The block diagram of Chien search is shown in figure 4.2 \[15\].

![Block Diagram of Chien Search](image)

Figure 4.2 Block Diagram of Chien Search

4.5 Determine the Error Values
After the errors are located, the error locations can be plugged into the syndromes to obtain the error value. Since the calculation scale is too large for matrix inversion, Forney’s algorithm \[16\][17] is normally adopted to determine the error values. The algorithm is illustrated as below:

Let the number of errors be \( k \), \( k \leq t \), then it can be derived that

\[
s_l = \sum_{j=1}^{k} y_j x_j^l
\]  
(4.41)

\[
s(x) = s_0 + s_1 x + s_2 x^2 + \cdots = \sum_{i=1}^{\infty} s_i x^i
\]
\[= 1 + \sum_{i=1}^{\infty} (\sum_{j=1}^{k} y_j x_j^i) x^i
\]
\[= 1 + \left( \sum_{j=1}^{k} y_j \left( \sum_{i=1}^{\infty} x_j x^i \right) \right). \]  
(4.42)

From equation \( \sum_{l=1}^{\infty} a^l = \frac{a}{1-a} \mid a \mid < 1 \), it can be derived that

\[
s(x) = 1 + \left( \sum_{j=1}^{k} y_j \frac{x_j x}{1-x_j x} \right). \]  
(4.43)

After arithmetic operations, it ultimately follows that \[17\] (Forney’s algorithm)

\[
y_i = \frac{-x_i \omega(x_i^{-1})}{\omega'(x_i^{-1})}. \]  
(4.44)

Once the error locations \( x_i \) and the error values \( y_i \) are determined, the correct code word can be found—subtract the error values from the received vectors in the corresponding error locations.

4.6 Hardware Realization of Reed-Solomon Decoder

4.6.1 Structure Diagram and Top Level Module of the Decoder

The structure diagram of a RS decoder is shown in figure 4.3
The block diagram of the top level module is shown in figure 4.4. In the block diagram, “clk” is the clock signal; “rst” is the reset signal; “sync” is the initial signal, i.e. the “synchronization” signal; “Din” is the 8-bit long data input signal and it contains 223 signals; “Dout” is the 8-bit output signal.

4.6.2 Syndrome Calculation Module

The block diagram of the syndrome calculation module is shown in figure 4.5. In the figure, “clock” is the clock signal; “init” is the start flag of the data packet; “rcvd” is the 8-bit input data signal; “synd_out” is the 8-bit output syndrome; “sc_finish” is the signal for the completion of this module.
The block diagram of the multiplier is shown in figure 4.6. In the figure, “A” and “B” are 8-bit input sequences; “P” is 8-bit output data.

4.6.3 Error Location Polynomial Calculation Module

Error location polynomial calculation module is the Berlekamp-Massey algorithm module. Its block diagram is shown in figure 4.7. In the block diagram, “clock” is the clock signal; “s” is the 8-bit input of syndrome; “kes_initial” is the start flag of the module, and its input is the “sc_finish” signal from syndrome calculation module; “Lmd0” — “Lmd8” are the output σ(x); “L” is the output ω(x); “kes_fnsh” is the start flag of the next level calculation; “synd_out” is the output syndrome.
4.6.4 Chien Search Module

The block diagram of Chien search module is shown in figure 4.8. In the block diagram, “clock” is the clock signal; “init” is the start flag of the module, and its input is the “kes_fnsh” signal from error location polynomial calculation module; “Lmd0” — “Lmd8” are the input $\sigma(x)$; “Err_Indicator” is the output error location number.
4.6.5 Error Value Evaluator Module

The block diagram of the error value evaluator module is shown in figure 4.9. In the diagram, “clock” is the clock signal; “init” is the start flag of the module, and its input is the “kes_fnsh” signal from error location polynomial calculation module; “synd” is the 8-bit input syndrome; “Lmd0” — “Lmd8” are the input $\sigma(x)$; “ErrValue” is the error value.

![Figure 4.9 Block Diagram of Error Value Evaluator Module](image)

4.6.6 Storage Module

The storage unit of the decoder is rom. The block diagrams of the inverse calculation and the power calculation is respectively shown in figure 4.10 and figure 4.11. “clock” is the input clock; “address” is the input address; “q” is the output.

![Figure 4.10 Block Diagram of Inverse Calculation](image)
Figure 4.11 Block Diagram of Power Calculation
Chapter 5 Synthesis and Simulation of (204, 188) Reed-Solomon Encoder/Decoder

5.1 Synthesis and Simulation of the Encoder

The software used to perform compilation and synthesis is Altera Quartus ii. A test bench is created to execute the simulation. In the test module, an input of decimal numbers from 1 to 204 is created. Transmit the input into the encoder, after calculation, the output received is the code word. The software used to simulate the encoder is Modelsim.

For convenient comparison, a MATLAB program is adopted to create the parity check bits. The program is written as:

```matlab
msg=ones(1, 239);
for i=1:51;
    msg(i)=0;
end
for j=52:239;
    msg(j)=(j-51);
end
m=8;n=255;k=239;b=0;
genpoly=rsgenpoly(n, k, [], b);
msg_p=gf(msg, m);
cwd=rsenc(msg_p, n, k, genpoly)
```

The sixteen parity check symbols are: 195, 231, 90, 194, 142, 112, 85, 171, 63, 242, 251, 154, 1, 82, 33, 222.

Simulate the circuit in Modelsim to obtain the simulation waveform as below.
In figure 5.1, signals “clrn”, “din_ena”, and “cnt” are tested. From the waveform, when “clrn” is 0, “cnt” is 0; when “din_ena” changed from low to high, “cnt” remained 0. Therefore, the three signals satisfy the requirement of the design.

Figure 5.2 shown the status of the circuit after running several clock cycles. In the waveform, “din_ena” and “clrn” are both 1. The encoder started working. The first 188 information symbols are transmitted into the system. Based on the coding theories, the outputs should be the same as these 188 input symbols. From the figure, it can be seen that the first 188 outputs are the same as the inputs.
The above two figures displayed the encoding of the last 16 input symbols. From the waveform, it can be seen that the 188th data of “dout” is 188, and the following 16 symbols are 195, 231, 90, 194, 142, 112, 85, 171, 63, 242, 251, 154, 1, 82, 33, 222, which are consistent with the outcome of the MATLAB program.

Based on the above analysis, the designed encoder embraces the ability to correctly generate code words.
The hardware design of the encoder is realized using EP2S15F484C3 of the Stratix II. The analysis is shown in figure 5.5.

<table>
<thead>
<tr>
<th>Flow Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Status</td>
</tr>
<tr>
<td>Quartus II Version</td>
</tr>
<tr>
<td>Revision Name</td>
</tr>
<tr>
<td>Top-Level Entity Name</td>
</tr>
<tr>
<td>Family</td>
</tr>
<tr>
<td>Timing requirements</td>
</tr>
<tr>
<td>Logic utilization</td>
</tr>
<tr>
<td>Comb. ALUTs</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
</tr>
<tr>
<td>Total registers</td>
</tr>
<tr>
<td>Total pins</td>
</tr>
<tr>
<td>Total virtual pins</td>
</tr>
<tr>
<td>Total block memory bits</td>
</tr>
<tr>
<td>DSP block slice elements</td>
</tr>
<tr>
<td>Total PLLs</td>
</tr>
<tr>
<td>Total DLLs</td>
</tr>
<tr>
<td>Device</td>
</tr>
<tr>
<td>Timing Models</td>
</tr>
</tbody>
</table>

Figure 5.5 Analysis of the Encoder Components

5.2 Synthesis and Simulation of the Decoder

For the decoder, a vector waveform file and several “.mif” files are created to verify the functionality of the decoder. The software used are Altera Quartus ii and MATLAB. The vector waveform file was created by Quartus ii. The “.mif” files are created using MATLAB. The MATLAB programs are included in Appendix A.

5.2.1 Input / Output with 0 Error

The following three figures show the input signal in error-free situation. In the waveforms, “clk” is the clock; “rst” is the reset button; “sync” is the enable button; “Din” is the input to the decoder; “Dout” is the output.
Figures 5.6 – 5.8 Decoder Error-Free Input Waveform

The following three figures show the output of the decoder in error-free situation.
Figure 5.9 – 5.10 Decoder Error-Free Output Waveform

From the waveforms above, it can be seen that the decoder can decode correctly when there is no error.
### 5.2.2 Input / Output with 6 Errors

Below are figures showing the input and output of the decoder, when there are 6 errors in the code word.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
<th>Value 7</th>
<th>Value 8</th>
<th>Value 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rst</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>En</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tdec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>out</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Below is a detailed view of the input and output signals over time.

![Input and Output Diagram](image-url)

- **Input**: clk, rst, En, tdec
- **Output**: out, decoded signals
Figure 5.11 – 5.14 6 Error Decoder Input / Output Waveforms

From the above figures, it can be seen that the decoder corrected the 6 errors (4 in information symbols and 2 in parity check symbols) located in the code word.

5.2.3 Input / Output with 8 Errors

The next set data contains 8 errors in the input symbols.
Figure 5.15 – 5.18 8 Error Decoder Input / Output Waveforms

From the above waveforms, it can be derived that when there are no more than 8 error symbols in the code word, the decoder can accurately perform error correction.
5.2.4 Input / Output with 9 Errors

The following testing verified that once the number of the presence of errors is greater than 8, the error correction will fail.
According to all the waveforms illustrated above, it can be seen that the error correcting capability of RS (204, 188) code is 8 symbols.

The hardware design of the encoder is realized using EP2C35F672C6 of the Cyclone II. The analysis is shown in figure 5.23.
Chapter 6 Conclusion

As a special type of error control codes, due to its outstanding error correcting capability on both random and burst errors, Reed-Solomon codes are nowadays widely used in various types of digital communication systems, especially in DVB (Digital Video Broadcasting) systems, HDTV systems, satellite communication systems, and data storage systems.

In this project an introduction to the theory of error control coding and the fundamentals of finite field were first presented. Then, the Reed-Solomon encoding/decoding algorithms are discussed and the (204, 188) Reed-Solomon encoder/decoder systems are designed; synthesized; simulated; and verified. The achievements of this project included but are not limited to:

1. Used constant multipliers on finite field to replace common multipliers, which leads to the reduction of design complexity and hardware resources.

2. Adopted the improved Berlekamp-Massey algorithm (inverse-free) to solve the error location polynomial

3. Employed Verilog HDL to model the whole systems. Used Quartus ii, Modelsim, and MATLAB to perform design, synthesis, simulation, and verification of the circuit.

4. Utilized pipeline architecture to design the decoder system.

In spite of all the accomplishments of this project, some of the work which deserve future consideration include:

1. Adopt inverse-free Chien Search.

2. Employ other improved BM algorithms.

3. Implement the system with a real FPGA hardware.
For the design of the RS decoder, future research should mainly concentrate on the simplification of the circuit, the concision of the architecture, the acceleration of the running speed, the expansion of the throughput capacity, and the reduction of the delay.
References


Appendix A RTL Description

(204, 188) Reed-Solomon Encoder / Decoder RTL Description

A-1 Reed-Solomon Encoder RTL Description
A-2 Reed-Solomon Decoder RTL Description
Appendix B Verilog HDL Files

B-1 (204, 188) Reed-Solomon Code Encoder

encode.v

`timescale 1ns / 1ns

//Constant Multipliers
//g(x) = x**16 + 59x**15 + 13x**14 + 104x**13 + 189x**12 + 68x**11 + 209x**10 + 30x**9
//     + 8x**8 + 163x**7 + 65x**6 + 41x**5 + 229x**4 + 98x**3 + 50x**2 + 36x + 59.
//**59
module gfmult0(y,x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
end
endmodule
/**36**

module gfmult1(y, x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
  y[1]=x[7]^x[4];
end
endmodule

//**50**

module gfmult2(y, x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
  y[0]=x[4]^x[3];
end
endmodule

//**98**

module gfmult3(y, x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
end
endmodule
module gfmult4(y,x);
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always @(x)
begin
end
endmodule

module gfmult5(y,x);
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always @(x)
begin
end
endmodule

module gfmult6(y,x);
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always @(x)
begin

end
endmodule
```verilog
module gfmult7(y,x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
end
endmodule

module gfmult8(y,x);
input [7:0]x;
output [7:0]y;
reg [7:0]y;
always@(x)
begin
    y[7] = x[4];
    y[1] = x[6];
    y[0] = x[5];
end
endmodule

module gfmult9(y,x);
```

75
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always@(x)
begin
end
endmodule

// *209
module gfmult10(y,x);
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always@(x)
begin
end
endmodule

// *68
module gfmult11(y,x);
input [7:0] x;
output [7:0] y;
reg [7:0] y;
always@(x)
begin
  y[1] = x[3];
  y[0] = x[7] * x[2];
module gfmult12(y, x);
  input [7:0] x;
  output [7:0] y;
  reg [7:0] y;
  always @(x)
    begin
    end
endmodule

module gfmult13(y, x);
  input [7:0] x;
  output [7:0] y;
  reg [7:0] y;
  always @(x)
    begin
    end
endmodule

module gfmult14(y, x);
  input [7:0] x;
  output [7:0] y;
  reg [7:0] y;
  always @(x)
    begin
module gfmult15(y, x);
    input [7:0] x;
    output [7:0] y;
    reg [7:0] y;
    always@(x)
    begin
    end
endmodule

// in this design gfmult15 will not be used
// since it is the same as gfmult0
module encode(clk, rst_n, din, din_ena, din_syn, bypass, dout, dout_ena, dout_syn);
    input clk, rst_n; // clock and reset button
    input [7:0] din; // information input
    input din_ena, din_syn; // enable button
    output [7:0] dout; // code word output
    output dout_ena, dout_syn;
    reg [7:0] cnt_204; // counter
always@(posedge clk or negedge rst_n) begin

// set the counter
if(!rst_n) begin
    cnt_204 <= 8'd0;
end
else if(din_ena) begin
    if(din_syn)
        cnt_204 <= 8'd1;
    else
        cnt_204 <= (cnt_204 < 8'd204) ? (cnt_204 + 1'b1) : 8'd0;
end
else begin
    cnt_204 <= cnt_204;
end
end

// product of the constant multiplication
wire [7:0]mul_59,mul_13,mul_104,mul_189,mul_68,mul_209,mul_30,mul_8,mul_163,
mul_65,mul_41,mul_229,mul_98,mul_50,mul_36;

// values in the shift registers
reg [7:0]shift0,shift1,shift2,shift3,shift4,shift5,shift6,shift7,shift8,
    shift9,shift10,shift11,shift12,shift13,shift14,shift15;

// the feedback
reg [7:0]xor_feedback;

always @ (din or shift15)
begin
    xor_feedback = (cnt_204 > 8'd187) ? 8'd0 : din^shift15;
end

// gfmult15 u15 (mul_59,xor_feedback);
gfmult14 u14 (mul_13, xor_feedback);
gfmult13 u13 (mul_104, xor_feedback);
gfmult12 u12 (mul_189, xor_feedback);
gfmult11 u11 (mul_68, xor_feedback);
gfmult10 u10 (mul_209, xor_feedback);
gfmult9 u9 (mul_30, xor_feedback);
gfmult8 u8 (mul_8, xor_feedback);
gfmult7 u7 (mul_163, xor_feedback);
gfmult6 u6 (mul_65, xor_feedback);
gfmult5 u5 (mul_41, xor_feedback);
gfmult4 u4 (mul_229, xor_feedback);
gfmult3 u3 (mul_98, xor_feedback);
gfmult2 u2 (mul_50, xor_feedback);
gfmult1 u1 (mul_36, xor_feedback);
gfmult0 u0 (mul_59, xor_feedback);

always@(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    //set the initial states of the shift registers
    shift0 <= 0;
    shift1 <= 0;
    shift2 <= 0;
    shift3 <= 0;
    shift4 <= 0;
    shift5 <= 0;
    shift6 <= 0;
    shift7 <= 0;
    shift8 <= 0;
    shift9 <= 0;
    shift10 <= 0;
    shift11 <= 0;
    shift12 <= 0;
    shift13 <= 0;
    shift14 <= 0;
    shift15 <= 0;
  end
  else if(din_ena) begin
    shift15 <= shift14^mul_59;
    shift14 <= shift13^mul_13;
    shift13 <= shift12^mul_104;
    shift12 <= shift11^mul_189;
    shift11 <= shift10^mul_68;
    shift10 <= shift9^mul_209;
    shift9 <= shift8^mul_30;
    shift8 <= shift7^mul_8;
    shift7 <= shift6^mul_163;
    shift6 <= shift5^mul_65;
    shift5 <= shift4^mul_41;
    shift4 <= shift3^mul_229;
    shift3 <= shift2^mul_98;
    shift2 <= shift1^mul_50;
    shift1 <= shift0^mul_36;
    shift0 <= mul_59;
  end
end

assign dout = (cnt_204 < 8’d188) ? din : shift15;
assign dout_ena = din_ena;
assign dout_syn = din_syn;
endmodule
sigin.v

module sigin(data, clk, ena, rst);
input clk, ena, rst; // clock, enable, and reset buttons
output [7:0] data; // Codeword
reg [7:0] data;

always @ (posedge clk or negedge rst)
begin
  if (!rst)
    data = 8'h0;
  else if (ena)
    data = data + 8'b1;
end
endmodule
module tb_encode;
reg clk, clrn;       // clock and reset buttons
wire [7:0] din;     // information symbols input from sigin
reg din_ena, din_syn; // enable and sync buttons
wire [7:0] dout;     // code word output
reg [7:0] address;  // for sigin
reg enc_trigger;    // main control button
reg [7:0] k;        // stop

//tick the clock
always #50 clk=~clk;

//instantiate sigin
sigin x0 (din, address);

//instantiate encode
encode x1 (clk, clrn, din_ena, din_syn, din, dout);

always@(posedge clk or negedge clrn)
begin
  if(~clrn)
  begin
    din_ena <= 0;
    din_syn <= 0;
    address <= 0;
  end
  else
  begin
    if (enc_trigger && address == 0)
begin
  din_ena <= 1;
end
else if (address == k-1)
  begin
    din_ena <= 0;
  end
if (enc_trigger && address == 0)
  begin
    din_syn <= 1;
  end
else
  begin
    din_syn <= 0;
  end
if (din_ena) address <= address + 1;
else address <= 0;
end

initial
begin
  clk = 0;
  clrn = 1;
  enc_trigger = 0;
k=204;
  #1 clrn = 0;
  #20 clrn = 1;
  #200
  #100 enc_trigger = 1;
  #100 enc_trigger =0;
  #20400 $finish;
end
endmodule
B-2 (204, 188) Reed-Solomon Code Decoder

rs_decoder.v

module rs_decoder(clk, rst, sync, Din, Dout);

parameter t = 8, // t—Error correcting capability
            N = 204, // N—Length of codeword
            m = 8; // m—Power of GF(2)

input clk, rst, sync; //clock, reset, and sync buttons
input [m-1:0] Din; //received vector
output [m-1:0] Dout; //output

wire Err_Indicator;
wire [m-1:0] ErrorVal;
reg [m-1:0] Dout;
wire [m-1:0] shift_out;
always @ (posedge clk)
begin
    if (Err_Indicator) Dout <= shift_out ^ ErrorVal;
    else Dout <= shift_out;
end

wire [m-1:0] sc_s_out;
wire sc_finish;
SCalc inst_sc(.clock(clk),
               .init(sync),
               .sc_finish(sc_finish),
               .rcvd(Din),
               .synd_out(sc_s_out));

module rs_decoder(clk, rst, sync, Din, Dout);

//This Verilog file is a part of the (204, 188) Reed-Solomon Code
//Encoder / Decoder Systems Design project.
//This file is the top module of the decoder system. The error correcting capability is 8. It instantiates the syndrome calculation module, the Berlekamp-Massey algorithm module, the error evaluator module, and the Chien search module. Then it shifts out the output.

//This design is synthesized, simulated, and verified using Quartus ii and MATLAB. Each functionality desired in this code is realized.

//Verilog HDL File rs_decoder.v
*Created by Haoyi Zhang 9/21/13
*CSUN, Northridge, CA

This Verilog file is a part of the (204, 188) Reed-Solomon Code Encoder / Decoder Systems Design project. This file is the top module of the decoder system. The error correcting capability is 8. It instantiates the syndrome calculation module, the Berlekamp-Massey algorithm module, the error evaluator module, and the Chien search module. Then it shifts out the output.

This design is synthesized, simulated, and verified using Quartus ii and MATLAB. Each functionality desired in this code is realized.

module rs_decoder(clk, rst, sync, Din, Dout);
parameter t = 8, // t—Error correcting capability
            N = 204, // N—Length of codeword
            m = 8; // m—Power of GF(2)

input clk, rst, sync; //clock, reset, and sync buttons
input [m-1:0] Din; //received vector
output [m-1:0] Dout; //output

wire Err_Indicator;
wire [m-1:0] ErrorVal;
reg [m-1:0] Dout;
wire [m-1:0] shift_out;
always @(posedge clk)
begin
    if (Err_Indicator) Dout <= shift_out ^ ErrorVal;
    else Dout <= shift_out;
end

wire [m-1:0] sc_s_out;
wire sc_finish;
SCalc inst_sc(.clock(clk),
               .init(sync),
               .sc_finish(sc_finish),
               .rcvd(Din),
               .synd_out(sc_s_out));
wire [m-1:0] kes_s_out, Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8, L;
wire      kes_fnsh;

//Instance of Berlekamp-Massey
BM_KES inst_kes(.Synd(sc_s_out),
                  .clock(clk),
                  .kes_initial(sc_finish),
                  .kes_fnsh(kes_fnsh),
                  .Lmd0(Lmd0), .Lmd1(Lmd1), .Lmd2(Lmd2),
                  .Lmd3(Lmd3), .Lmd4(Lmd4), .Lmd5(Lmd5),
                  .Lmd6(Lmd6), .Lmd7(Lmd7), .Lmd8(Lmd8),
                  .L(L),
                  .synd_out(kes_s_out));

//Instance of error evaluator
error_evaluator inst_eval (.clock(clk),
                         .init(kes_fnsh),
                         .synd(kes_s_out),
                         .Lmd0(Lmd0), .Lmd1(Lmd1), .Lmd2(Lmd2),
                         .Lmd3(Lmd3), .Lmd4(Lmd4), .Lmd5(Lmd5),
                         .Lmd6(Lmd6), .Lmd7(Lmd7), .Lmd8(Lmd8),
                         .ErrValue(ErrorVal));

//Instance of Chien search
ChienSearch inst_ChS(.clock(clk),
                      .init(kes_fnsh),
                      .Lmd0(Lmd0), .Lmd1(Lmd1), .Lmd2(Lmd2),
                      .Lmd3(Lmd3), .Lmd4(Lmd4), .Lmd5(Lmd5),
                      .Lmd6(Lmd6), .Lmd7(Lmd7), .Lmd8(Lmd8),
                      .Err_Indicator(Err_Indicator));

altshift_taps inst_SR(  
                       .clock (clk),  
                       .shiftin (Din),  
                       .taps (),  
                       .shiftout (shift_out)  
                       // synopsys translate_off  
                       ,  
                       .clken ()  
                       // synopsys translate_on  
                       );

defparam  
    inst_SR.width = 8,  
    inst_SR.number_of_taps = 1,  
    inst_SR.tap_distance = 243,  
    inst_SR.lpm_type = "altshift_taps";
endmodule
module SCalc(clock, init, sc_finish, rcvd, synd_out);
parameter t = 8, // t -- Error correcting capability
      N = 204, // N -- Length of codeword
      m = 8; // m -- Power of GF(2)
input clock, init; // clock, init buttons
input [m-1:0] rcvd; // received vector
output [m-1:0] synd_out; // syndrome
output sc_finish; // indicate whether the calculation is done

reg [m-1:0] synd[t*2:1], synd_latched[t*2:1];
wire [m-1:0] rcvd_a[t*2:1];
integer cnt;
always @(posedge clock)begin:SC_BLOCK
    integer j;
    if((init) || (cnt==N))begin
        for(j=1; j<=t*2; j=j+1)
            synd_latched[j] <= synd[j];
        for(j=1; j<=t*2; j=j+1)
            synd[j] <= rcvd;
        cnt <= 1;
    end
    else if(cnt<=N-1)begin
        for(j=1; j<=t*2; j=j+1)
            synd[j] <= rcvd ^. rcvd_a[j];
        cnt <= cnt + 1;
    end
    // a^1
    ff_cnst_mult rcvd_x_a1(.din(synd[1]), .dout(rcvd_a[1]));
// a^2
ff_cnst_mult rcvd_x_a2(.din(synd[2]), .dout(rcvd_a[2]));
// a^3
ff_cnst_mult rcvd_x_a3(.din(synd[3]), .dout(rcvd_a[3]));
// a^4
ff_cnst_mult rcvd_x_a4(.din(synd[4]), .dout(rcvd_a[4]));
// a^5
ff_cnst_mult rcvd_x_a5(.din(synd[5]), .dout(rcvd_a[5]));
// a^6
ff_cnst_mult rcvd_x_a6(.din(synd[6]), .dout(rcvd_a[6]));
// a^7
ff_cnst_mult rcvd_x_a7(.din(synd[7]), .dout(rcvd_a[7]));
// a^8
ff_cnst_mult rcvd_x_a8(.din(synd[8]), .dout(rcvd_a[8]));
// a^9
ff_cnst_mult rcvd_x_a9(.din(synd[9]), .dout(rcvd_a[9]));
// a^10
ff_cnst_mult rcvd_x_a10(.din(synd[10]), .dout(rcvd_a[10]));
// a^11
ff_cnst_mult rcvd_x_a11(.din(synd[11]), .dout(rcvd_a[11]));
// a^12
ff_cnst_mult rcvd_x_a12(.din(synd[12]), .dout(rcvd_a[12]));
// a^13
ff_cnst_mult rcvd_x_a13(.din(synd[13]), .dout(rcvd_a[13]));
// a^14
ff_cnst_mult rcvd_x_a14(.din(synd[14]), .dout(rcvd_a[14]));
// a^15
ff_cnst_mult rcvd_x_a15(.din(synd[15]), .dout(rcvd_a[15]));
// a^16
ff_cnst_mult rcvd_x_a16(.din(synd[16]), .dout(rcvd_a[16]));
defparam rcvd_x_a1.CONST  = 15'h4405;
defparam rcvd_x_a2.CONST  = 15'h6202,  rcvd_x_a3.CONST = 15'h7101,
rcvd_x_a4.CONST  = 15'h3880,  rcvd_x_a5.CONST  = 15'h1C40,
rcvd_x_a6.CONST  = 15'h0E20,  rcvd_x_a7.CONST  = 15'h4710,
rcvd_x_a8.CONST  = 15'h2388,  rcvd_x_a9.CONST  = 15'h11C4,
rcvd_x_a10.CONST = 15'h48E2,  rcvd_x_a11.CONST = 15'h2471,
rcvd_x_a12.CONST = 15'h5238,  rcvd_x_a13.CONST = 15'h691C,
rcvd_x_a14.CONST = 15'h748E,  rcvd_x_a15.CONST = 15'h3A47,
rcvd_x_a16.CONST = 15'h1D23;

reg sc_finish;
integer shift_cnt;
always @(posedge clock)begin

if(cnt == N)begin
    sc_finish <= 1;
    shift_cnt <= 1;
end
else begin
    sc_finish <= 0;
    if((0<shift_cnt) && (shift_cnt <= t*2))begin
        shift_cnt <= shift_cnt + 1;
    end
    else
        shift_cnt <= 0;
end
end

reg [m-1:0] synd_out;
always @(shift_cnt)begin
    if((0<shift_cnt) &&(shift_cnt <= t*2))
        synd_out = synd_latched[shift_cnt];
    else
        synd_out = 0;
end
endmodule
module BM_KES(Synd, clock, kes_initial, kes_fnsh, Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8, L, synd_out);

parameter t = 8, N = 204, m = 8;
input clock, kes_initial;
input [m-1:0] Synd;
output [m-1:0] L, Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8,
synd_out;
output [m-1:0] kes_fnsh;
reg [m-1:0] L;

reg [m-1:0] B[t:0];
reg [m-1:0] T[t:0], gama, delta, k;
reg [m-1:0] SyndSR[t*2:1];
wire [m-1:0] Synd_B[t:0];
integer cnt;
reg [m-1:0] synd_latched[t*2:1];
reg compute, kes_fnsh;
always @(posedge clock)begin:BM_BLOCK
    integer j;
    if(kes_initial)begin
        // Initial variables
        // B(D) = B[2]*D^2 + B[1]*D + B[0]: The coefficients of the error locator polynomial
        // gama--Intermediate variables, delta--Discrepancy
        // Registers of LFSR for storing the syndrome
        // Product of (Synd*B)
        // cnt;
        // synd_latched[t*2:1];
        // compute, kes_fnsh;
        //if(kes_initial)begin
        //Initial variables
    end
end

// This Verilog file is a part of the (204, 188) Reed-Solomon Code
// Encoder / Decoder Systems Design project.
// This file is the Berlekamp-Massey algorithm module. It will be
// instantiated in the decoder top module, rs_decoder.
// The algorithm implemented here is an improved inverse-free BM
// algorithm.
// This design is synthesized, simulated, and verified using
// Quartus ii and MATLAB. Each functionality desired in this code
// is realized.
delta <= 0;
for(j=2; j<=t*2; j=j+1)
    SyndSR[j] <= 0;
SyndSR[1] <= Synd;
cnt <= 0;
end
else if(cnt == t*2-1)begin
    for(j=1; j<=t*2-1; j=j+1)
        SyndSR[j+1] <= SyndSR[j];
    SyndSR[1] <= Synd;
    for(j=1; j<=t*2; j=j+1) // Latch the syndrome sequences input
        synd_latched[j] <= SyndSR[j];
cnt <= cnt + 1;
end
else if(cnt <= t*2)begin
    for(j=1; j<=t*2-1; j=j+1)
        SyndSR[j+1] <= SyndSR[j];
    SyndSR[1] <= Synd;
    cnt <= cnt + 1;
end
end

// Finite field multiplication of delta
ff_mult Synd_x_B0(.B(SyndSR[1]), .A(B[0]), .P(Synd_B[0]));
ff_mult Synd_x_B1(.B(SyndSR[2]), .A(B[1]), .P(Synd_B[1]));
ff_mult Synd_x_B3(.B(SyndSR[4]), .A(B[3]), .P(Synd_B[3]));
ff_mult Synd_x_B4(.B(SyndSR[5]), .A(B[4]), .P(Synd_B[4]));
ff_mult Synd_x_B5(.B(SyndSR[6]), .A(B[5]), .P(Synd_B[5]));
ff_mult Synd_x_B6(.B(SyndSR[7]), .A(B[6]), .P(Synd_B[6]));
ff_mult Synd_x_B7(.B(SyndSR[8]), .A(B[7]), .P(Synd_B[7]));
ff_mult Synd_x_B8(.B(SyndSR[9]), .A(B[8]), .P(Synd_B[8]));

wire [m-1:0] gama_B[t:0];
wire [m-1:0] delta_T[t-1:0];
always @(negedge clock)begin:LABEL
    integer j;
    if(kes_initial)begin
for(j=1; j=t; j=j+1) begin
    B[j] <= 0;  T[j] <= 0;
end
B[0] <= 1;  T[0] <= 1;
L <= 0;  k <= 0;  gama <= 1;
end
else if(compute) begin
    B[0] <= gama_B[0];
    if((delta) && ((L<<1)<= k)) begin
        for(j=0; j=t; j=j+1)
            T[j] <= B[j];
        L <= k + 1 - L;
        gama <= delta;
    end
else begin
    for(j=0; j=t; j=j+1) begin
        T[j+1] <= T[j];
    end
end
T[0] <= 0;
end

k <= k + 1;
end

// Finite field multiplication of gama*B(D)
ff_mult gama_x_B0(.A(gama), .B(B[0]), .P(gama_B[0]));
ff_mult gama_x_B1(.A(gama), .B(B[1]), .P(gama_B[1]));
ff_mult gama_x_B3(.A(gama), .B(B[3]), .P(gama_B[3]));
ff_mult gama_x_B4(.A(gama), .B(B[4]), .P(gama_B[4]));
ff_mult gama_x_B5(.A(gama), .B(B[5]), .P(gama_B[5]));
ff_mult gama_x_B6(.A(gama), .B(B[6]), .P(gama_B[6]));
ff_mult gama_x_B7(.A(gama), .B(B[7]), .P(gama_B[7]));
ff_mult gama_x_B8(.A(gama), .B(B[8]), .P(gama_B[8]));

/* Finite field multiplication of delta*lamda(D) */
ff_mult delta_x_T0(.A(delta), .B(T[0]), .P(delta_T[0]));
ff_mult delta_x_T1(.A(delta), .B(T[1]), .P(delta_T[1]));
ff_mult delta_x_T2(.A(delta), .B(T[2]), .P(delta_T[2]));
ff_mult delta_x_T3(.A(delta), .B(T[3]), .P(delta_T[3]));
ff_mult delta_x_T4(.A(delta), .B(T[4]), .P(delta_T[4]));
ff_mult delta_x_T5(.A(delta), .B(T[5]), .P(delta_T[5]));
ff_mult delta_x_T6(.A(delta), .B(T[6]), .P(delta_T[6]));
ff_mult delta_x_T7(.A(delta), .B(T[7]), .P(delta_T[7]));

always @(cnt)begin
    if((0<cnt)&&(cnt<=t*2))
        compute = 1;
    else
        compute = 0;
end

integer shft_cnt;
always @(posedge clock)begin
    if(cnt == t*2-1)begin
        kes_fnsh <= 1;
        shft_cnt <= 1;
    end
    else begin
        kes_fnsh <= 0;
        if((0<shft_cnt) && (shft_cnt <= t*2))begin
            shft_cnt <= shft_cnt + 1;
        end
        else
            shft_cnt <= 0;
    end
end

reg [m-1:0] synd_out;
always @(shft_cnt)begin
    if((0<shft_cnt) &&(shft_cnt <= t*2))
        synd_out = synd_latched[t*2-shft_cnt+1];
    else
        synd_out = 0;
end

assign Lmd0 = B[0];
assign Lmd1 = B[1];
assign Lmd2 = B[2];
assign Lmd3 = B[3];
assign Lmd4 = B[4];
assign Lmd5 = B[5];
assign Lmd6 = B[6];
assign Lmd7 = B[7];
assign Lmd8 = B[8];
endmodule
module ChienSearch(clock, init,
    Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8,
    Err_Indicator);

parameter t = 8,         // t--Error correcting capability
    N = 204,         // N--Length of codeword
    m = 8,          // m--Power of GF(2)
    Delay_Length = 18;

input clock, init;
input [m-1:0] Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8;
output Err_Indicator;

reg Err_Indicator;
reg [m-1:0] LmdIR[t:0];    // Iterative register
wire [m-1:0] LmdIR_a[t:1], a52Lmd[t:1];
always @(posedge clock)begin:CHIEN_SEARCH
    integer j;
    if(init)begin
        for(j=1; j<=t; j=j+1)
            LmdIR[j] <= a52Lmd[j];
        LmdIR[0] <= Lmd0;
    end
    else begin
        for(j=1; j<=t; j=j+1)
            LmdIR[j] <= LmdIR_a[j];
        LmdIR[0] <= LmdIR[0];
    end
    // LmdIR1*a^-1
    ff_cnst_mult LmdIR_x_a1(.din(LmdIR[1]), .dout(LmdIR_a[1]));
    // LmdIR2*a^-2
ff_cnst_mult LmdIR_x_a2(.din(LmdIR[2]), .dout(LmdIR_a[2]));  
    // LmdIR3*a^3
ff_cnst_mult LmdIR_x_a3(.din(LmdIR[3]), .dout(LmdIR_a[3]));  
    // LmdIR4*a^4
ff_cnst_mult LmdIR_x_a4(.din(LmdIR[4]), .dout(LmdIR_a[4]));  
    // LmdIR5*a^5
ff_cnst_mult LmdIR_x_a5(.din(LmdIR[5]), .dout(LmdIR_a[5]));  
    // LmdIR6*a^6
ff_cnst_mult LmdIR_x_a6(.din(LmdIR[6]), .dout(LmdIR_a[6]));  
    // LmdIR7*a^7
ff_cnst_mult LmdIR_x_a7(.din(LmdIR[7]), .dout(LmdIR_a[7]));  
    // LmdIR8*a^8
ff_cnst_mult LmdIR_x_a8(.din(LmdIR[8]), .dout(LmdIR_a[8]));
    
defparam LmdIR_x_a1.CONST = 15'h6202,  LmdIR_x_a2.CONST = 15'h7101,
       LmdIR_x_a3.CONST = 15'h3880,  LmdIR_x_a4.CONST =
       15'h1C40,
       LmdIR_x_a5.CONST = 15'h0E20,  LmdIR_x_a6.CONST =
       15'h4710,
       LmdIR_x_a7.CONST = 15'h2388,  LmdIR_x_a8.CONST =
       15'h11C4;
    
    // a^52*lambda1
ff_cnst_mult a52_x_Lmd1(.din(Lmd1),  .dout(a52Lmd[1]));  
    // a^52*lambda2
ff_cnst_mult a52_x_Lmd2(.din(Lmd2),  .dout(a52Lmd[2]));  
    // a^52*lambda3
ff_cnst_mult a52_x_Lmd3(.din(Lmd3),  .dout(a52Lmd[3]));  
    // a^52*lambda4
ff_cnst_mult a52_x_Lmd4(.din(Lmd4),  .dout(a52Lmd[4]));  
    // a^52*lambda5
ff_cnst_mult a52_x_Lmd5(.din(Lmd5),  .dout(a52Lmd[5]));  
    // a^52*lambda6
ff_cnst_mult a52_x_Lmd6(.din(Lmd6),  .dout(a52Lmd[6]));  
    // a^52*lambda7
ff_cnst_mult a52_x_Lmd7(.din(Lmd7),  .dout(a52Lmd[7]));  
    // a^52*lambda8
ff_cnst_mult a52_x_Lmd8(.din(Lmd8),  .dout(a52Lmd[8]));
    
defparam a52_x_Lmd1.CONST = 15'h6D41,  a52_x_Lmd2.CONST = 15'h0D84,
       a52_x_Lmd3.CONST = 15'h1BB9,  a52_x_Lmd4.CONST =
       15'h1F55,
       a52_x_Lmd5.CONST = 15'h0E20,  a52_x_Lmd6.CONST =
       15'h6B6A,
       a52_x_Lmd7.CONST = 15'h3C6C,  a52_x_Lmd8.CONST =
       15'h3CDD;

reg  Err_Delay[Delay_Length:0];  
always @(posedge clock)begin:ERROR_DELAY
integer j;
Err_Delay[0] <= ((LmdIR[0] \ LmdIR[1]) \ (LmdIR[2] \ LmdIR[3])

\ (LmdIR[4] \ LmdIR[5]) \ (LmdIR[6] \ LmdIR[7]) \ LmdIR[8]) ? 0:1;

for(j=0; j<Delay_Length; j=j+1)
    Err_Delay[j+1] <= Err_Delay[j];

Err_Indicator <= Err_Delay[Delay_Length];
end

endmodule
module error_evaluator(clock, init, synd, Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8, ErrValue);
  parameter t = 8, // t--Error correcting capability
    N = 204, // N--Length of codeword
    m = 8; // m--Power of GF(2)

  input clock, init;
  input [m-1:0] synd, Lmd0, Lmd1, Lmd2, Lmd3, Lmd4, Lmd5, Lmd6, Lmd7, Lmd8;
  output [m-1:0] ErrValue;

  wire [m-1:0] InvDLmdA;

  reg [m-1:0] OmgC[t*2-1:0], SyndSR[t+1:1], LmdC[t:0];
  wire [m-1:0] syndLmdC[t:0];
  integer cnt1;
  always @(posedge clock)begin:OMEGA_COEFFICIENT
    integer j;
    if(init)begin
      // Load the syndrome (from 1 to t*2)
      for(j=2; j<=t+1; j=j+1)
        SyndSR[j] <= 0;
      SyndSR[1] <= synd;
      // Load error-locator polynomial coefficients omega(x)
      LmdC[0] <= Lmd0; LmdC[1] <= Lmd1; LmdC[2] <= Lmd2;

// Initiate OmgC[j] registers
for(j=0; j<t*2-1; j=j+1)
    OmgC[j] <= 0;

cnt1 <= 1;
end
else if((0<cnt1) && (cnt1 <= t*2))begin // Computation of omega0, omega1, omega2, omega3
for(j=1; j=t; j=j+1)
    SyndSR[j+1] <= SyndSR[j];
    SyndSR[1] <= synd;
for(j=0; j<t*2-1; j=j+1)
    OmgC[j] <= OmgC[j+1];
    OmgC[t*2-1] <= (syndLmdC[0]^syndLmdC[1])^(syndLmdC[2]^syndLmdC[3])

cnt1 <= cnt1 + 1;
end
else
    cnt1 <= 0;
end
ff_mult synd_x_Lmdc0(.A(SyndSR[1]), .B(LmdC[0]), .P(syndLmdC[0]));
ff_mult synd_x_Lmdc1(.A(SyndSR[2]), .B(LmdC[1]), .P(syndLmdC[1]));
ff_mult synd_x_Lmdc2(.A(SyndSR[3]), .B(LmdC[2]), .P(syndLmdC[2]));
ff_mult synd_x_Lmdc3(.A(SyndSR[4]), .B(LmdC[3]), .P(syndLmdC[3]));
ff_mult synd_x_Lmdc4(.A(SyndSR[5]), .B(LmdC[4]), .P(syndLmdC[4]));
ff_mult synd_x_Lmdc5(.A(SyndSR[6]), .B(LmdC[5]), .P(syndLmdC[5]));
ff_mult synd_x_Lmdc6(.A(SyndSR[7]), .B(LmdC[6]), .P(syndLmdC[6]));
ff_mult synd_x_Lmdc7(.A(SyndSR[8]), .B(LmdC[7]), .P(syndLmdC[7]));
ff_mult synd_x_Lmdc8(.A(SyndSR[9]), .B(LmdC[8]), .P(syndLmdC[8]));

reg Compute_Error, Load_Lmd; // signal of the computation of omega(a^i)
always @(cnt1)begin
    if(cnt1 == (t*2+1) ) Compute_Error = 1;
    else Compute_Error = 0;
end

/*************************************************************
* Convolution of coefficients of omega(a^i)*
*************************************************************/
reg [m-1:0] OmgIR[t*2-1:0], OmgA;
wire[m-1:0] aOmgIR[t*2-1:1], a520mgC[t*2-1:1];
always @(posedge clock)begin:OMEGA_ALPHA

97
integer j;
if(Compute_Error)begin
  // Load the OmgIR[j] registers
  for(j=1; j<t*2-1; j=j+1)
    OmgIR[j] <= a52OmgC[j];
  OmgIR[0] <= OmgC[0];
end
else begin
  // Stage--computation of \omega(a^i)
  for(j=1; j<t*2-1; j=j+1)
    OmgIR[j] <= aOmgIR[j];
  OmgIR[0] <= OmgIR[0];
end

// a^1*\omega_1
ff_cnst_mult a1_x_OmgIR1 (.din (OmgIR[1]), .dout (aOmgIR[1]));
// a^1*\omega_2
ff_cnst_mult a2_x_OmgIR2 (.din (OmgIR[2]), .dout (aOmgIR[2]));
// a^1*\omega_3
ff_cnst_mult a3_x_OmgIR3 (.din (OmgIR[3]), .dout (aOmgIR[3]));
// a^1*\omega_4
ff_cnst_mult a4_x_OmgIR4 (.din (OmgIR[4]), .dout (aOmgIR[4]));
// a^1*\omega_5
ff_cnst_mult a5_x_OmgIR5 (.din (OmgIR[5]), .dout (aOmgIR[5]));
// a^1*\omega_6
ff_cnst_mult a6_x_OmgIR6 (.din (OmgIR[6]), .dout (aOmgIR[6]));
// a^1*\omega_7
ff_cnst_mult a7_x_OmgIR7 (.din (OmgIR[7]), .dout (aOmgIR[7]));
// a^1*\omega_8
ff_cnst_mult a8_x_OmgIR8 (.din (OmgIR[8]), .dout (aOmgIR[8]));
// a^1*\omega_9
ff_cnst_mult a9_x_OmgIR9 (.din (OmgIR[9]), .dout (aOmgIR[9]));
// a^1*\omega_10
ff_cnst_mult a10_x_OmgIR10 (.din (OmgIR[10]), .dout (aOmgIR[10]));
// a^1*\omega_11
ff_cnst_mult a11_x_OmgIR11 (.din (OmgIR[11]), .dout (aOmgIR[11]));
// a^1*\omega_12
ff_cnst_mult a12_x_OmgIR12 (.din (OmgIR[12]), .dout (aOmgIR[12]));
// a^1*\omega_13
ff_cnst_mult a13_x_OmgIR13 (.din (OmgIR[13]), .dout (aOmgIR[13]));
// a^1*\omega_14
ff_cnst_mult a14_x_OmgIR14 (.din (OmgIR[14]), .dout (aOmgIR[14]));
// a^1*\omega_15
ff_cnst_mult a15_x_OmgIR15 (.din (OmgIR[15]), .dout (aOmgIR[15]));
defparam a1_x_OmgIR1.CONST  = 'h6202,
a2_x_OmgIR2.CONST  = 'h7101,
a3_x_OmgIR3.CONST  = 'h3880,
a4_x_OmgIR4.CONST  = 'h1C40,
a5_x_OmgIR5.CONST  = 'h0E20,
a6_x_OmgIR6.CONST  = 'h1C40,
15'h4710,
    a7_x_OmgIR7.CONST = 15'h2388,    a8_x_OmgIR8.CONST =
15'h11C4,
    a9_x_OmgIR9.CONST = 15'h48E2,    a10_x_OmgIR10.CONST =
15'h2471,
    a11_x_OmgIR11.CONST = 15'h5238,   a12_x_OmgIR12.CONST =
15'h691C,
    a13_x_OmgIR13.CONST = 15'h748E,   a14_x_OmgIR14.CONST =
15'h3A47,
    a15_x_OmgIR15.CONST = 15'h1D23;

   // a^52*omega1
   ff_cnst_mult a52_x_OmgC1(.din(OmgC[1]),   .dout(a52OmgC[1]));
   // a^52*omega2
   ff_cnst_mult a52_x_OmgC2(.din(OmgC[2]),   .dout(a52OmgC[2]));
   // a^52*omega3
   ff_cnst_mult a52_x_OmgC3(.din(OmgC[3]),   .dout(a52OmgC[3]));
   // a^52*omega4
   ff_cnst_mult a52_x_OmgC4(.din(OmgC[4]),   .dout(a52OmgC[4]));
   // a^52*omega5
   ff_cnst_mult a52_x_OmgC5(.din(OmgC[5]),   .dout(a52OmgC[5]));
   // a^52*omega6
   ff_cnst_mult a52_x_OmgC6(.din(OmgC[6]),   .dout(a52OmgC[6]));
   // a^52*omega7
   ff_cnst_mult a52_x_OmgC7(.din(OmgC[7]),   .dout(a52OmgC[7]));
   // a^52*omega8
   ff_cnst_mult a52_x_OmgC8(.din(OmgC[8]),   .dout(a52OmgC[8]));
   // a^52*omega9
   ff_cnst_mult a52_x_OmgC9(.din(OmgC[9]),   .dout(a52OmgC[9]));
   // a^52*omega10
   ff_cnst_mult a52_x_OmgC10(.din(OmgC[10]), .dout(a52OmgC[10]));
   // a^52*omega11
   ff_cnst_mult a52_x_OmgC11(.din(OmgC[11]), .dout(a52OmgC[11]));
   // a^52*omega12
   ff_cnst_mult a52_x_OmgC12(.din(OmgC[12]), .dout(a52OmgC[12]));
   // a^52*omega13
   ff_cnst_mult a52_x_OmgC13(.din(OmgC[13]), .dout(a52OmgC[13]));
   // a^52*omega14
   ff_cnst_mult a52_x_OmgC14(.din(OmgC[14]), .dout(a52OmgC[14]));
   // a^52*omega15
   ff_cnst_mult a52_x_OmgC15(.din(OmgC[15]), .dout(a52OmgC[15]));

   defparam   a52_x_OmgC1.CONST = 15'h6D41,    a52_x_OmgC2.CONST =
15'h6D84,
    a52_x_OmgC3.CONST = 15'h1BB9,    a52_x_OmgC4.CONST =
15'h1F55,
    a52_x_OmgC5.CONST = 15'h0E20,    a52_x_OmgC6.CONST =
15'h6B6A,
    a52_x_OmgC7.CONST = 15'h3C6C,    a52_x_OmgC8.CONST =
15'h3CDD,
Reg [m-1:0] DLmdIR[3:0], DLmA;
wire[1:0] aDLmdIR[3:1], a52LmdC[3:1];
always @ (posedge clock) begin
  // Computation of the coefficients of delta'(x)
  if(Compute_Error) begin
    DLmdIR[0] <= LmdC[1];
    DLmdIR[1] <= a52LmdC[1];
    DLmdIR[2] <= a52LmdC[2];
    DLmdIR[3] <= a52LmdC[3];
  end
  // Error Compute_Error output of codeword
  else begin
    DLmdIR[1] <= aDLmdIR[1];
    DLmdIR[3] <= aDLmdIR[3];
  end
end

// a^2*omega1
ff_cnst_mult a2_x_DLmdIR1 (.din(DLmdIR[1]), .dout(aDLmdIR[1]));
// a^4*omega1
ff_cnst_mult a4_x_DLmdIR2 (.din(DLmdIR[2]), .dout(aDLmdIR[2]));
// a^6*omega1
ff_cnst_mult a6_x_DLmdIR3 (.din(DLmdIR[3]), .dout(aDLmdIR[3]));

defparam a2_x_DLmdIR1.CONST = 15'h7101,
a4_x_DLmdIR2.CONST = 15'h1C40,
a6_x_DLmdIR3.CONST = 15'h4710;

// a^104*omega1
ff_cnst_mult a52_x_LmdC3 (.din(LmdC[3]), .dout(a52LmdC[1]));
// a^208*omega2
ff_cnst_mult a52_x_LmdC5 (.din(LmdC[5]), .dout(a52LmdC[2]));
// a^312*omega3
ff_cnst_mult a52_x_LmdC7 (.din(LmdC[7]), .dout(a52LmdC[3]));
defparam a52_x_LmdC3.CONST = 15'h0D84,
a52_x_LmdC5.CONST = 15'h1F55,
a52_x_LmdC7.CONST = 15'h6B6A;
// Generate the power of alpha^i
reg [m-1:0] ExpIndex;
always @(posedge clock)begin
    if(Compute_Error)  ExpIndex <= 203;
    else                ExpIndex <= ExpIndex - 1;
end

always @(posedge clock)begin
    OmgA <= (OmgIR[0] ^ OmgIR[1]) ^ (OmgIR[2] ^ OmgIR[3])
        ^ (OmgIR[4] ^ OmgIR[5]) ^ (OmgIR[6] ^ OmgIR[7])
        ^ (OmgIR[8] ^ OmgIR[9]) ^ (OmgIR[10] ^ OmgIR[11])
        ^ (OmgIR[12] ^ OmgIR[13]) ^ (OmgIR[14] ^ OmgIR[15]);
end

// Compute alpha^i * omega(alpha^i)
wire [m-1:0] AlphaPower, OmgAlpha;
ff_mult_R Omega_x_AlphaPower(.clk(clock),
    .A(OmgA),
    .B(AlphaPower),
    .P(OmgAlpha));

// Compute error_pattern = alpha^i*omega(alpha^i)/ delta`(alpha^i)
ff_mult_R OmgAlpha_x_InvDifC(.clk(clock),
    .A(OmgAlpha),
    .B(InvDLmdA),
    .P(ErrValue));

// Inverse delta`(alpha^i)
rom_inv     DifC_Inv(.clock(clock), .address(DLmdA), .q(InvDLmdA));

// Output alpha^i
rom_power    Alpha_Power(.clock(clock), .address(ExpIndex), .q(AlphaPower));
endmodule
Verilog HDL File

* Created by Haoyi Zhang       9/21/13
* CSUN, Northridge, CA

This Verilog file is a part of the (204, 188) Reed-Solomon Code

* Encoder / Decoder Systems Design project.
* This file is the finite field multiplication module. It will be
* instantiated in the Berlekamp-Massey module, BM_KES, and error
* evaluator module, error_evaluator.

This design is synthesized, simulated, and verified using
* Quartus ii and MATLAB. Each functionality desired in this code
* is realized.

This Verilog file is a part of the (204, 188) Reed-Solomon Code

* Encoder / Decoder Systems Design project.
* This file is the finite field multiplication module. It will be
* instantiated in the Berlekamp-Massey module, BM_KES, and error
* evaluator module, error_evaluator.

This design is synthesized, simulated, and verified using
* Quartus ii and MATLAB. Each functionality desired in this code
* is realized.

`define M 8
module ff_mult(A, B, P);

//parameter
CONST = 8'h3b;

input ['M-1:0] A, B; // B needs to be converted to dual basis
output ['M-1:0] P;
reg ['M-1:0] P;

reg [2*`M-2:0] dual_base;
reg ['M-1:0] mul_temp;
always @(A or B)begin: Block1
    integer i;
    // C-D
    dual_base[0] = B[0] ^ B[2];
    dual_base[1] = B[1];
    dual_base[2] = B[0];
    dual_base[3] = B[7];

    // Extension
    for(i=0; i<`M-1; i=i+1) begin
end
end

/*
 */
// Multiplication
for(i=0; i<`M; i=i+1) begin
    mul_temp[i] = (((dual_base[i+0]&A[0]) ^(dual_base[i+1]&A[1]))
        ^((dual_base[i+2]&A[2])
        ^((dual_base[i+3]&A[3]))))
        ^((dual_base[i+4]&A[4])^(dual_base[i+5]&A[5]))
        ^((dual_base[i+6]&A[6])
        ^(dual_base[i+7]&A[7])));
end

// D-C
P[0] = mul_temp[2];
P[1] = mul_temp[1];
P[2] = mul_temp[0] ^ mul_temp[2];
P[4] = mul_temp[6];
P[5] = mul_temp[5];
P[6] = mul_temp[4];
P[7] = mul_temp[3];
end
endmodule

`define M 8
module ff_mult_R(clk, A, B, P);

input clk;
input [`M-1:0] A, B; // B needs to be converted to dual basis
output [`M-1:0] P;
reg [`M-1:0] P;

reg [2*`M-2:0] dual_base;
reg [`M-1:0] mul_temp;
always @(posedge clk)begin: Block1
    integer i;
    // C-D
    dual_base[0] = B[0] ^ B[2];
dual_base[1] = B[1];
dual_base[2] = B[0];
dual_base[3] = B[7];
end
// Extension
for (i=0; i<`M-1; i=i+1) begin
end

// Multiplication
for (i=0; i<`M; i=i+1) begin
    mul_temp[i] = (((dual_base[i+0]&A[0]) ^ (dual_base[i+1]&A[1]))
                   ^ (dual_base[i+2]&A[2])
                   ^ (dual_base[i+3]&A[3]))
                   ^ (dual_base[i+4]&A[4])
                   ^ (dual_base[i+5]&A[5])
                   ^ (dual_base[i+6]&A[6])
                   ^ (dual_base[i+7]&A[7]));
end

// D-C
P[0] = mul_temp[2];
P[1] = mul_temp[1];
P[2] = mul_temp[0]^mul_temp[2];
P[4] = mul_temp[6];
P[5] = mul_temp[5];
P[6] = mul_temp[4];
P[7] = mul_temp[3];
end
endmodule
ff_cnst_mult.v

/******************************************************************************
*Verilog HDL File ff_cnst_mult.v                                        *
*Created by Haoyi Zhang 9/21/13                                           *
*CSUN, Northridge, CA                                                    *
******************************************************************************
*This Verilog file is a part of the (204, 188) Reed-Solomon Code          *
*Encoder / Decoder Systems Design project.                                *
*This file is the finite field constant multiplication module. It*       *
*will be instantiated in the Chien search module, syndrome               *
*calculation module, SCalc, and error evaluator module,                  *
*error_evaluator.                                                         *
******************************************************************************
*Primitive polynomial: p(x) = x^8 + x^4 + x^3 + x^2 + 1                    *
*Polynomial basis: {1, a^1, a^2, a^3, a^4, a^5, a^6, a^7}                  *
*Weak dual basis: {1+a^2, a^1, 1, a^7, a^6, a^5, a^4, a^3+a^7}             *
******************************************************************************
*This design is synthesized, simulated, and verified using               *
*Quartus ii and MATLAB. Each functionality desired in this code*       *
is realized.                                                              *
******************************************************************************

`define M 8
module ff_cnst_mult(din, dout);
    parameter CONST = 15'h0CE7;

    input [`M-1:0] din;
    output [`M-1:0] dout;

    wire [2*`M-2:0] dual_base;
    assign dual_base = CONST;

    reg [`M-1:0] dout, temp;
    always @(din or dual_base)begin: Block1
        integer i;
        // multiplication
        for(i=0; i<`M; i=i+1) begin
            temp[i] = (((dual_base[i+0]&din[0]) ^(dual_base[i+1]&din[1]))
                           ^((dual_base[i+2]&din[2])
                             ^((dual_base[i+3]&din[3])))
                           ^((dual_base[i+4]&din[4])
                             ^((dual_base[i+5]&din[5]))
                           ^((dual_base[i+6]&din[6]))
                           ^((dual_base[i+7]&din[7])));
        end
    // transfer from weak dual basis to polynomial basis

endmodule
dout[0] = temp[2];

dout[1] = temp[1];
dout[2] = temp[0] ^ temp[2];
dout[4] = temp[6];
dout[5] = temp[5];
dout[6] = temp[4];
dout[7] = temp[3];

end
endmodule
rom_inv.v

// megafonction wizard: %LPM_ROM%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// File Name: rom_inv.v
// Megafunction Name(s):
//       altsyncram

// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

// 5.0 Build 148 04/26/2005 SJ Full Version

// Copyright (C) 1991-2005 Altera Corporation
// Your use of Altera Corporation’s design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, Altera MegaCore Function License
// Agreement, or other applicable license agreement, including,
// without limitation, that your use is for the sole purpose of
// programming logic devices manufactured by Altera and sold by
// Altera or its authorized distributors. Please refer to the
// applicable agreement for further details.

// synopsys translate_off
// `timescale 1 ps / 1 ps
// synopsys translate_on
module rom_inv (  
    address,
    clock,
    q);

    input [7:0] address;
    input clock;
    output [7:0] q;

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];
altsyncram altsyncram_component (  
    .clock0 (clock),
    .address_a (address),
    .q_a (sub_wire0)
    // synopsys translate_off
    ,
    .aclr0 (),
    .aclrl (),
    .address_b (),
    .addressstall_a (),
    .addressstall_b (),
    .byteena_a (),
    .byteena_b (),
    .clock1 (),
    .clocken0 (),
    .clocken1 (),
    .data_a (),
    .data_b (),
    .q_b (),
    .rden_b (),
    .wren_a (),
    .wren_b ()
    // synopsys translate_on
);

defparam
    altsyncram_component.intended_device_family = "Cyclone",
    altsyncram_component.width_a = 8,
    altsyncram_component.widthad_a = 8,
    altsyncram_component.numwords_a = 256,
    altsyncram_component.operation_mode = "ROM",
    altsyncram_component.outdata_reg_a = "NONE",
    altsyncram_component.outdata_aclr_a = "NONE",
    altsyncram_component.width_byteena_a = 1,
    altsyncram_component.init_file = "ROM_INV.mif",
    altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
    altsyncram_component.lpm_type = "altsyncram";
endmodule
module rom_power(
  address,
  clock,
  q);

input [7:0] address;
input clock;
output [7:0] q;

wire [7:0] sub_wire0;
wire [7:0] q = sub_wire0[7:0];
altsyncram altsyncram_component (  
    .clock0 (clock),  
    .address_a (address),  
    .q_a (sub_wire0)  
    // synopsys translate_off  
    ,  
    .aclr0 (),  
    .aclr1 (),  
    .address_b (),  
    .addressstall_a (),  
    .addressstall_b (),  
    .byteena_a (),  
    .byteena_b (),  
    .clock1 (),  
    .clocken0 (),  
    .clocken1 (),  
    .data_a (),  
    .data_b (),  
    .q_b (),  
    .rden_b (),  
    .wren_a (),  
    .wren_b ()  
    // synopsys translate_on  
);  

defparam  
altsyncram_component.intended_device_family = "Cyclone",  
altsyncram_component.width_a = 8,  
altsyncram_component.widthad_a = 8,  
altsyncram_component.numwords_a = 256,  
altsyncram_component.operation_mode = "ROM",  
altsyncram_component.outdata_reg_a = "UNREGISTERED",  
altsyncram_component.address_aclr_a = "NONE",  
altsyncram_component.outdata_aclr_a = "NONE",  
altsyncram_component.width_byteena_a = 1,  
altsyncram_component.init_file = "rom_power.mif",  
altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",  
altsyncram_component.lpm_type = "altsyncram";  

d module  
    // CXI file retrieval info  
    // ----------------------------------------------------------------------  
    // Retrieval info: PRIVATE: WidthData NUMERIC "8"  
    // Retrieval info: PRIVATE: WidthAddr NUMERIC "8"  
    // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "256"  
    // Retrieval info: PRIVATE: SingleClock NUMERIC "1"  
    // Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
module testdata (address, clock, q);

input [7:0] address;
input clock;
output [7:0] q;
wire [7:0] sub_wire0;
wire [7:0] q = sub_wire0[7:0];
altsyncram altsyncram_component (  
  .clock0 (clock),  
  .address_a (address),  
  .q_a (sub_wire0),  
  .aclr0 (1'b0),  
  .aclrl (1'b0),  
  .address_b (1'b1),  
  .addressstall_a (1'b0),  
  .addressstall_b (1'b0),  
  .byteena_a (1'b1),  
  .byteena_b (1'b1),  
  .clock1 (1'b1),  
  .clocken0 (1'b1),  
  .clocken1 (1'b1),  
  .clocken2 (1'b1),  
  .clocken3 (1'b1),  
  .data_a ({8{1'b1}}),  
  .data_b (1'b1),  
  .eccstatus (),  
  .q_b (),  
  .rden_a (1'b1),  
  .rden_b (1'b1),  
  .wren_a (1'b0),  
  .wren_b (1'b0));
defparam altsyncram_component.clock_enable_input_a = "BYPASS",  
  altsyncram_component.clock_enable_output_a = "BYPASS",  
  altsyncram_component.init_file = "testdata.mif",  
  altsyncram_component.intended_device_family = "Cyclone II",  
  altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",  
  altsyncram_component.lpm_type = "altsyncram",  
  altsyncram_component.numwords_a = 204,  
  altsyncram_component.operation_mode = "ROM",  
  altsyncram_component.outdata_aclr_a = "NONE",  
  altsyncram_component.outdata_reg_a = "CLOCK0",  
  altsyncram_component.widthad_a = 8,  
  altsyncram_component.width_a = 8,  
  altsyncram_component.width_byteena_a = 1;
endmodule
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata_bb.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata_waveforms.html TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL testdata_wave*.jpg FALSE
// Retrieval info: LIB_FILE: altera_mf
module testdatabench(clk, dataout, first);
input clk;
output [7:0] dataout;
output first;

reg [7:0] count;
always @(posedge clk)begin
  if (count==8'd203)
    count<=8'd0;
  else count<=count+8'd1;
end

reg first1, first;
always @(posedge clk)begin
  if (count==8'd0) begin
    first1<=8'd1;
    first<=first1;
  end
  else begin
    first1<=8'd0;
    first<=first1;
  end
end

testdata test(
  .address(count),
  .clock(clk),
  .q(dataout))
endmodule
module rs_decoder204_188(clk, rst, sync, Din, Dout);
    input clk, rst; // clock, reset buttons
    output [7:0] Din; // received vector
    output [7:0] Dout; // output
    output sync; // sync button

    wire [7:0] dataout;
    wire first;
    testdatatabench test(clk, dataout, first); // instance of testbench
    rs_decoder rsdec(clk, rst, first, dataout, Dout); // instance of decoder

    reg [7:0] Din;
    always @(dataout) begin
        Din<=dataout;
    end

    reg sync;
    always @(first) begin
        sync<=first;
    end
endmodule
Appendix C MATLAB Files

C-1 (204, 188) Reed-Solomon Code Encoder

**Generator_polynomial.m**

```matlab
n = 255;
k = 239;
b = 0;
genpoly = rsgenpoly(n, k, [], b)
```
Parity_Check_Bits.m

msg=ones(1, 239);
for i=1:51;
    msg(i)=0;
end
for j=52:239;
    msg(j)=(j-51);
end
m=8; n=255; k=239; b=0;
genpoly=rsgenpoly(n, k, [], b);
msg_p=gf(msg, m);
cwd=rsenc(msg_p, n, k, genpoly)
function miffile_create(filename,var,width,depth)

if(nargin~=4)
    error('Need 4 parameters! Use help miffile for help!');
end,

fh=fopen(filename,'w+');
fprintf(fh,'--Created by xxxx.
');
fprintf(fh,'--xxxxx@126.com.
');
fprintf(fh,'--%s.
',datestr(now));
fprintf(fh,'WIDTH=%d;
',width);
fprintf(fh,'DEPTH=%d;
',depth);
fprintf(fh,'ADDRESS_RADIX=UNS;
');
fprintf(fh,'DATA_RADIX=HEX;
');
fprintf(fh,'CONTENT BEGIN
');

var=rem(var,2^width);
[sx, sy, sz]=size(var);

value=var(1,1,1);
sametotal=1;
idepth=0;
addrlen=1;
temp=16;
while(temp<depth)
    temp=temp*16;
    addrlen=addrlen+1;
end,

datalen=1;
while(temp<width)
    temp=temp*16;
    datalen=datalen+1;
end,

for k=1:sz,
    for j=1:sy,
        for i=1:sx,
            if(~((i==1) && (j==1) && (k==1)))
                if(idepth<depth),
                    idepth=idepth+1;
                    if(value==var(i,j,k))
                        sametotal=sametotal+1;
                        continue;
                    else
                        if(sametotal==1)
                            fprintf(fh,['\t%d:%' num2str(datalen) 'X;\n'],idepth-1,value);
else
    fprintf(fh, ['	\t%d..%d:
num2str(datalen) 'X;\n'],idepth-sametotal,idepth-1,value);
end,
sametotal=1;
    value=var(i,j,k);
end,
else
    break;
end,
end,
end,
end,
end,
end,

if(idepth<depth)
    if(sametotal==1)
        fprintf(fh, ['	\t%d:
num2str(datalen) 'X;\n'],idepth,value);
    else
        fprintf(fh, ['	\t%d..%d:
num2str(datalen) 'X;\n'],idepth-sametotal+1,idepth,value);
    end,
end,
if(idepth<depth-1)
    if(idepth==(depth-2))
        fprintf(fh, ['	\t%d:
num2str(datalen) 'X;\n'],idepth+1,0);
    else
        fprintf(fh, ['	\t%d..%d:
num2str(datalen) 'X;\n'],idepth+1,depth-1,0);
    end,
end,

fprintf(fh,'END;\n');
fclose(fh);
```matlab
testdata_error_free.m

a = ones(1, 188);
for i = 1:188
    a(i) = i;
end

a(189:204) = [195 231 90 194 142 112 85 171 63 242 251 154 1 82 33 222];
a
miffile_create('J:\MATLAB\zhanghaoyi\paper\decoder\testdata.mif', a, 8, 204);
```
testdata_6_errors.m

a=ones(1,188);
a(1:9)=[1 0 3 0 5 0 7 0 9];
for i=10:188
a(i)=i;
end

a(189:204)=[195 0 90 194 142 0 85 171 63 242 251 154 1 82 33 222];
a=miffile_create('J:\MATLAB\zhanghaoyi\paper\decoder\testdata.mif',a,8,204);
testdata_8_errors.m

a = ones(1, 188);
a(1:9) = [1 0 3 0 5 0 7 0 9];
for i = 10:188
    a(i) = i;
end

a(189:204) = [195 0 90 194 0 0 0 171 63 242 251 154 1 82 33 222];
a
miffile_create('J:\MATLAB\zhanghaoyi\paper\decoder\testdata.mif', a, 8, 204);
testdata_9_errors.m

a=ones(1,188);
a(1:14)=[1 0 3 0 5 0 7 0 9 10 11 0 0 14];
for i=15:188
a(i)=i;
end

a(189:204)=[195 231 90 194 0 0 0 171 63 242 251 154 1 82 33 222];
a
miffle_create('J:\MATLAB\zhanghaoyi\paper\decoder\testdata.mif',a,
8,204);