Design and Implementation of Reed Solomon Encoder/Decoder Using FPGA

Thesis report submitted towards the partial fulfilment of requirements for the award of the degree of

Master of Technology (VLSI Design & CAD)

Submitted by

RAGHAVENDRA SINGH SOLANKY
60761014

Under the Guidance of

Mr. H.K.S. RANDHAWA
Lecturer, ECED

Department of Electronics and Communication Engineering
THAPAR UNIVERSITY
PATIALA-147004
INDIA

JULY 2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Design and Implementation of Reed Solomon Encoder/Decoder Using FPGA"** in partial fulfillment of the requirement for the award of degree of M.Tech (VLSI Design & CAD) at Electronics and communication Department of Thapar University Patiala, is an authentic record of my own carried out under the supervision of **Mr. H.K.S. Randhawa**, Lecturer, ECED.

The matter embodied in this thesis has not been submitted in any other University/Institute for the award of any degree.

Date: 15.07.09

Raghavendra Singh Solanky
60761014

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Mr.H.K.S. Randhawa
Lecturer
ECED, Thapar University

Counter signed by:

(Dr. A. K. Chatterjee)
Professor & Head
ECE Department,
Thapar University, Patiala

(Mr. R. K Sharma)
Dean(Academic Affairs)
Thapar University,
Patiala.

ACKNOWLEDGEMENT

Words are often too less to reveal one's deep regards. An understanding of the work like this is never the outcome of the efforts of a single person. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis.

First, I would like to thank the Supreme Power, the GOD, who guided me to work on the right path of life. This work would not have been possible without the encouragement and able guidance of '**Mr. H.K.S. RANDHAWA**', Lecturer and **'Dr. SANJAY SHARMA'**, Associate Professor, TIET, PATIALA. Their enthusiasm and optimism made this experience both rewarding and enjoyable. Most of the novel ideas and solutions found in this thesis are the result of our numerous stimulating discussions. His feedback and editorial comments were also invaluable for the writing of this thesis. I am grateful to Head of the Department '**Dr. A.K. CHATTERJEE'** for providing the facilities for the completion of thesis.

I take pride of myself being son of ideal great parents whose everlasting desire, sacrifice, affectionate blessing and help without which it would have not been possible for me to complete my studies.

At last, I would like to thank all the members and employees of Electronics and Communication Department, TU Patiala whose love and affection made this possible.

RAGHAVENDRA S. SOLANKY

Abstract

Channel coding is used for providing reliable information through the transmission channel to the user. It is an important operation for the digital communication system transmitting digital information over a noisy channel. Forward error correction technique depending on the properties of the system or on the application in which the error correcting is to be introduced. Reed solomon codes are an important sub class of nonbinary BCH codes. These are cyclic codes and are very effectively used for the detection and correction of burst errors. Galois field arithmetic is used for encoding and decoding of reed solomon codes.

Galois field multipliers are used for encoding the information block. At the decoder, the syndrome of the received codeword is calculated using the generator polynomial to detect errors. Then to correct these errors, an error locator polynomial is calculated. From the error locator polynomial, the location of the error and its magnitude is obtained. Consequently a correct codeword is obtained. Block lengths and symbol sizes can be readily adjusted to accommodate a wide range of message sizes. Reed solomon codes provides a wide range of code values that can be chosen to optimize performance.

The results constitute simulation of verilog codes of different modules of the reed solomon codes in Xilinx. The results demonstrate that the reed solomon codes are very efficient for the detection and correction of burst errors.

Contents

List of Figures

Figures **Page No.**

Chapter-1

Introduction

1.1 Overview

Digital communication system is used to transport an information bearing signal from the source to a user destination via a communication channel. The information signal is processed in a digital communication system to form discrete messages which makes the information more reliable for transmission. Channel coding is an important signal processing operation for the efficient transmission of digital information over the channel. It was introduced by Claude E. Shannon in 1948 by using the channel capacity as an important parameter for error free transmission. In channel coding the number of symbols in the source encoded message is increased in a controlled manner in order to facilitate two basic objectives at the receiver one is Error detection and other is Error correction. Error detection and Error correction to achieve good communication is also employed in devices. It is used to reduce the level of noise and interferences in electronic medium. The amount of error detection and correction required and its effectiveness depends on the signal to noise ratio (SNR) [2].

In digital communication, a channel code is a broadly used term mostly referring to the forward error correction code. Forward error correction (FEC) is a system of error control for data transmission, whereby the sender adds redundant data to its messages, also known as an error correction code. This allows the receiver to detect and correct errors (within some bound) without the need to ask the sender for additional data. The advantage of forward error correction is that a back-channel is not required, or that retransmission of data can often be avoided, at the cost of higher bandwidth requirements on average. FEC is therefore applied in situations where retransmissions are relatively costly or impossible. In particular, FEC information is usually added to most mass storage devices to protect against damage to the stored data [3].

There are many types of block codes, but the most notable is reed solomon coding, Golay, BCH, Multidimensional parity, and Hamming codes are other examples of block codes. In real world communication, errors are introduced in messages sent from one point to another (Fig-1.1). Reed solomon is an error-correcting coding system that was devised to

address the issue of correcting multiple errors - especially burst-type errors in mass storage devices (hard disk drives, DVD, barcode tags), wireless and mobile communications units, satellite links, digital TV, digital video broadcasting (DVB), and modem technologies like xDSL ("x" referring to all the existing DSL solutions, whether ADSL, VDSL, SDSL, or HDSL)[8].

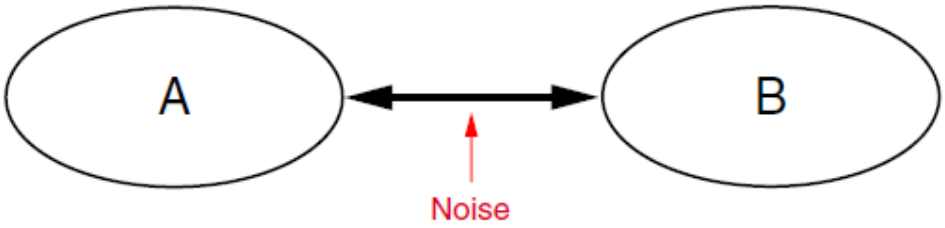


Fig 1.1: Two Points Exchanging Information

In order for the transmitted data to be corrected in the event that it acquires errors, it has to be encoded. The receiver uses the appended encoded bits to determine and correct the errors upon reception of the transmitted signal. The number and type of errors that are correctable depend on the specific reed solomon coding scheme used.

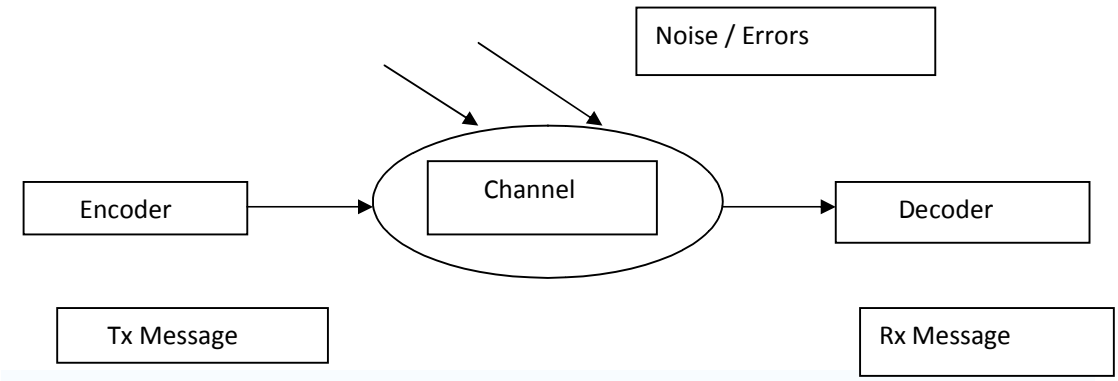


Fig 1.2: Application of RS Codes

Reed solomon codes are a subset of BCH codes and are linear block codes. A reed solomon code is specified as RS (n, k) with s-bit symbols. This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol

codeword. There are n-k parity symbols of s bits each. A Reed solomon decoder can correct up to t symbols that contain errors in a codeword, where 2t = n-k. The following diagram shows a typical reed solomon codeword (this is known as a systematic code because the data is left unchanged and the parity symbols are appended)

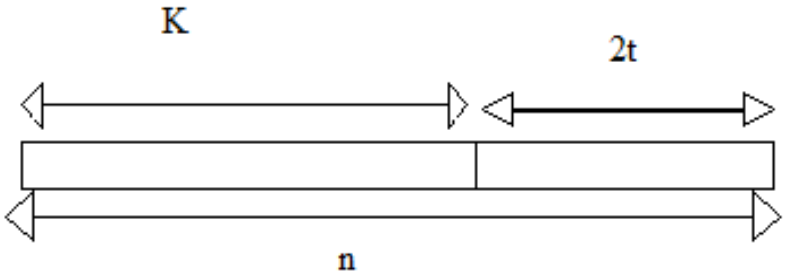


Fig 1.3: Reed Solomon Codeword

Example: A popular reed solomon code is RS (255,223) with 8-bit symbols. Each codeword contains 255 code word bytes, of which 223 bytes are data and 32 bytes are parity for this code

n = 255, k = 223, s = 8

2t = 32, t = 16

The decoder can correct any 16 symbol errors in the code word: i.e. errors in up to 16 bytes anywhere in the codeword can be automatically corrected. Given a symbol size s, the maximum codeword length (n) for a reed solomon code is $\mathbf{n = 2^s - 1.}$ For example, the maximum length of a code with 8-bit symbols (s=8) is 255 bytes. Reed solomon codes may be shortened by (conceptually) making a number of data symbols zero at the encoder, not transmitting them, and then re-inserting them at the decoder.

1.2 Encoder

The reed solomon encoder reads in k data symbols computes the n – k symbols, append the parity symbols to the k data symbols for a total of n symbols. The encoder is essentially a 2t tap shift register where each register is m bits wide. The multiplier coefficients are the coefficients of the RS generator polynomial. The general idea is the construction of a polynomial, the coefficient produced will be symbols such that the gene-

-rator polynomial will exactly divide the data/parity polynomial.

1.3 Decoder

The Reed solomon decoder tries to correct errors and/or erasures by calculating the syndromes for each codeword. Based upon the syndromes the decoder is able to determine the number of errors in the received block. If there are errors present, the decoder tries to find the locations of the errors using the berlekamp-massey algorithm by creating an error locator polynomial. The roots of this polynomial are found using the chien search algorithm. Using forney's algorithm, the symbol error values are found and corrected. For an RS (n, k) code where n - k = 2T, the decoder can correct up to T symbol errors in the code word. Given that errors may only be corrected in units of single symbols (typically 8 data bits), reed solomon coders work best for correcting burst errors [5].

Chapter -2

Reed Solomon Encoder

2.1 Introduction

The reed solomon code is an algebraic code belonging to the class of BCH (Bose-Chaudry-Hocquehen) multiple burst correcting cyclic codes. The reed solomon code operates on bytes of fixed length. Given m parity bytes, a reed solomon code can correct up to m byte errors in known positions (erasures), or detect and correct up to m/2 byte errors in unknown positions. This is an implementation of a reed solomon code with 8 bit bytes, and a configurable number of parity bytes. The maximum sequence length (codeword) that can be generated is 255 bytes, including parity bytes. In practice, shorter sequences are used.

2.2 Reed Solomon Encoder Hardware

Reed solomon codes are based on a specialist area of mathematics known as galois fields or finite fields. A finite field has the property that arithmetic operations (+, -, x, / etc.) on field elements always have a result in the field.

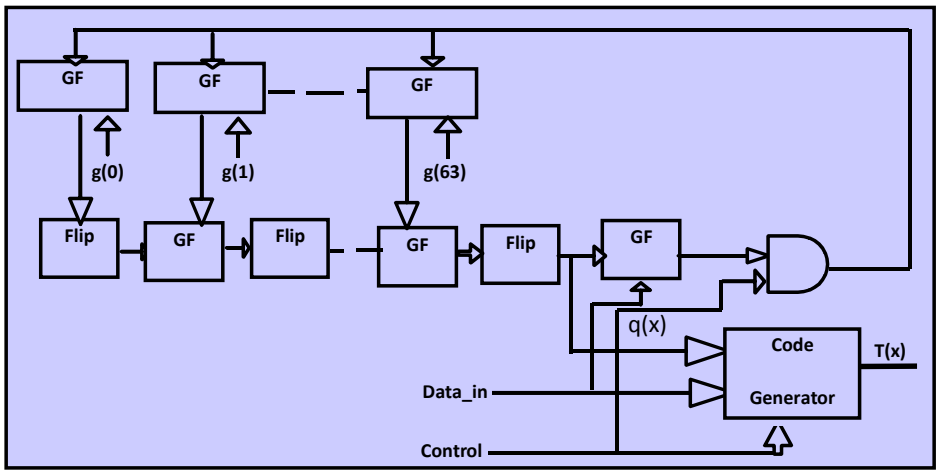


Fig 2.1: RS Encoder Block

The basic principle of encoding is to find the remainder of the message divided by a generator polynomial G(x). The encoder works by simulating a linear feedback shift register with degree equal to G(x), and feedback taps with the coefficients of the generating polynomial of the code [7].

2.2.1Micro Architecture of RS Encoder Block

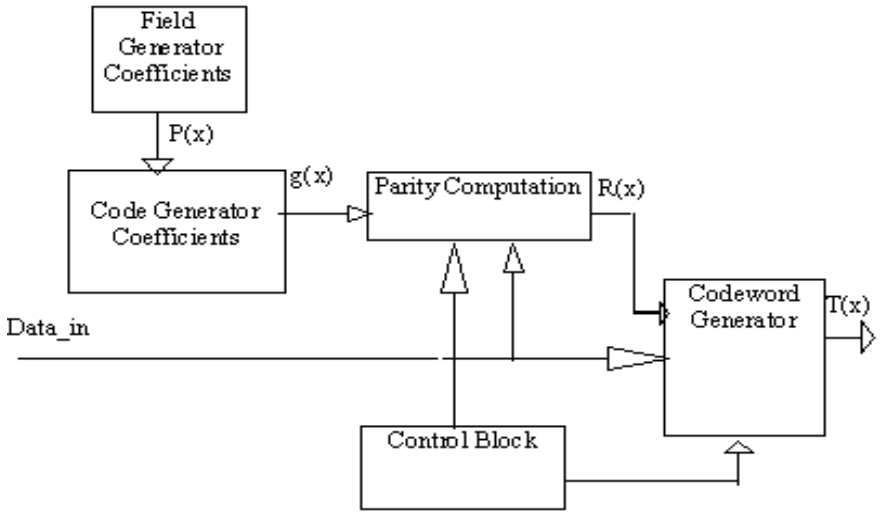


Fig 2.2: Micro Architecture of RS Encoder Block

Where

Data_in = Input symbols

P(x) = Field generator coefficients

T(x) = Codeword

R(x) = Parity symbols

g(x) = Code generator coefficients

2.3 Block Description

2.3.1 Field Generator Coefficients: - This block stores the constant values of field generator coefficients required for the generation of code generator polynomial. For the design we used the following field generator polynomial

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \quad (2.1)$$

Fig 2.3: Field Generator Block

2.3.2 Code Generator Coefficients

This block stores the constant values of the generator polynomial coefficients required for the encoding process.

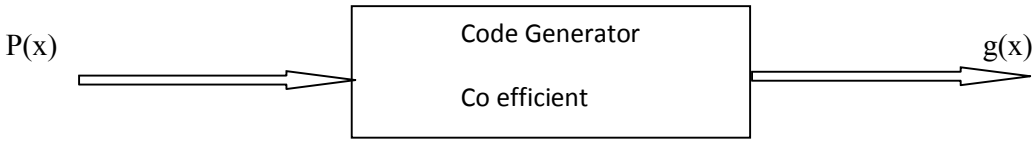


Fig 2.4: Code Generator Block

2.3.3 Generator Start

This is the galois field logarithm of the first root of the generator polynomial. i.e.

$$g(x) = \prod_{i=0}^{n-k-1} x - \alpha^{h*(Generator\_start+i)} \quad (2.2)$$

Normally generator start is 0 or 1, however it can be any whole number.

n - Number of symbols in an entire code block. If this is a shortened code, then n should be the shortened number.

k - Number of information or data symbols in a code block.

h - Scaling factor for the generator polynomial root index. Normally h is 1, however it can be any positive integer. The code generator polynomial obtained from p(x) is

2.3.4 Parity Computation

This block performs the necessary galois field multiplication and additions between input symbols and the constant coefficients g(x) in order to evaluate the parity symbols.

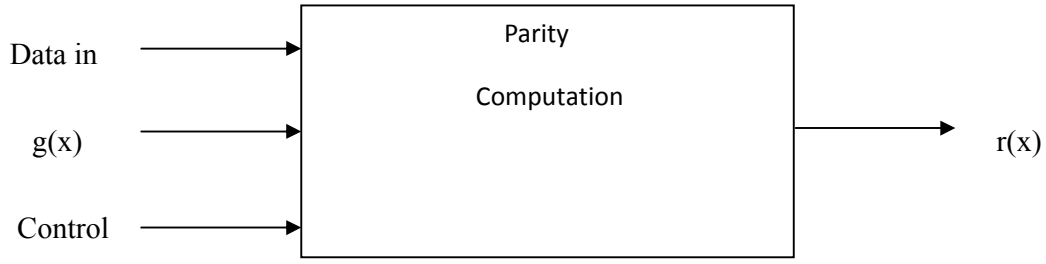


Fig 2.5: Parity Computation Block

2.3.5 Control block

This block controls the information flow through the encoder.

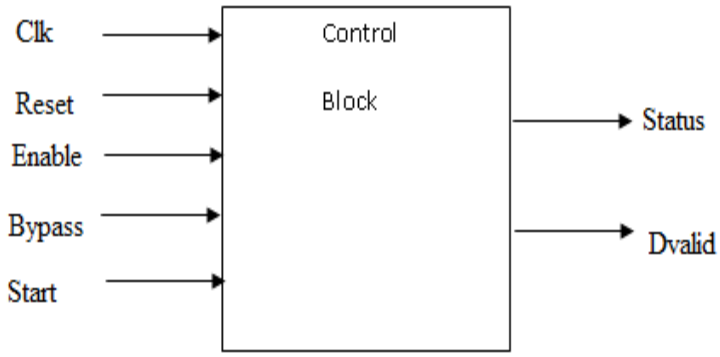


Fig 2.6: Control Block

2.3.6 Codeword Generator

This block generates the codeword by combining the message symbols and parity symbols. Data input, control signal and parity computation output are inputs of it. The generator polynomial g(x) for an RS code takes the following form.

$$g(x) = g_o + g_1(x) + g_2(x^2) + g_{2t-1}(x^{2t-1}) + x^{2t} \quad (2.3)$$

The degree of the generator polynomial is equal to the number of parity symbols 2t. Since the generator polynomial is of degree 2t there must be precisely 2t successive powers of α that are roots of the polynomial. In terms of root of polynomial alpha

$$g(x) = (x- \alpha)(x- \alpha^2)\ldots\ldots\ldots\ldots\ldots\ldots(x- \alpha^n) \quad (2.4)$$

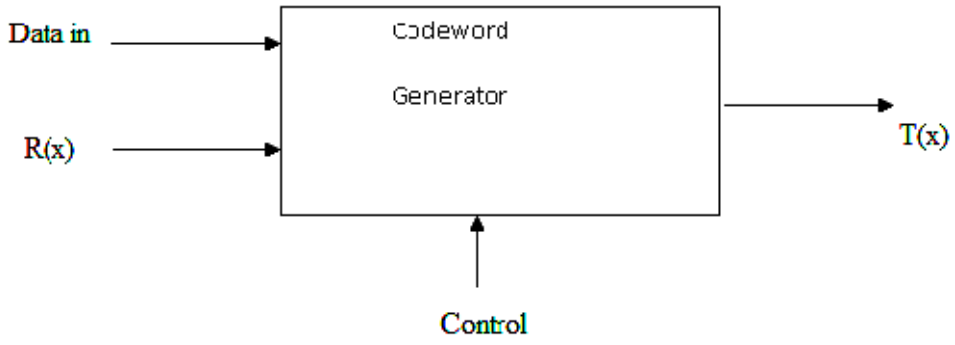


Fig 2.7: Codeword Generator

2.4 Galois Field

The theory of error control codes uses a mathematical construct known as finite fields or galois fields (GFs). A GF is a set that contains a finite number of elements. The operations of addition and multiplication on this set are defined and the operations behave as would be expected from normal arithmetic. For example, the additive identity element is 0 and the multiplicative identity element is 1. RS codes operate on GFs of order $q = p^m$ where p is a prime positive integer and m is a positive integer. A GF of order q is denoted by GF (q) and it contains q distinct elements The elements of a GF are typically denoted using the variable α. The elements of GF(8) have different notations. Elements are typically represented using either power or polynomial notation when performing calculations by hand, but binary notation is used when the codes are actually implemented in hardware. All three notations are simply three different ways to represent a given GF element. Multiplication is easier in power notation because the exponents are added. Similarly, addition is easier in polynomial notation. Numerous books and computer programs exist that list or generate the elements for GFs of various sizes. To implement an RS encoder and decoder, two special hardware blocks will be needed: a GF adder and a GF multiplier [6].

2.5 GF Multiplier

A simple, but inefficient, way to implement a GF multiplier is to take the inputs in binary notation and use a lookup table to find their corresponding power notations. The powers can be added (e.g., $\alpha^2 \times \alpha^5 = \alpha^7$) and the result can be sent to an inverse lookup table to find the corresponding binary notation for the output. These lookup tables can be stored in read only memory (ROM), but that is not practical because several multiplier outputs will typically need to be computed during a single clock cycle. A more efficient solution is to compute the equations by hand and simplify the terms. An RS encoder needs fixed multipliers that multiply an arbitrary input by a constant value such as α^2. The RS decoder needs both fixed multipliers as well as generic multipliers that are capable of multiplying any two arbitrary numbers together. The following example will show how to derive the equations for a fixed multiplier that multiplies its input by α^2.

Denote an arbitrary 3-bit multiplier input by $b_2\alpha^2 + b\alpha + b_0$. The multiplier will multiply this input by the constant α^2. Recall from Table 1 that $\alpha^4 = x^2 + x$ and $\alpha^3 = x + 1$. By replacing x with and substituting the result back into the above equation. The above equation represents the simplified equation for multiplying an arbitrary input by α^2. For example, the most significant bit (MSB) of the multiplier output is computed by XOR the input bits b_2 and b_0 together. The derivation for the generic multiplier can be found by simplifying the equations after using two arbitrary values for the inputs [10].

This multiplier is also called as "Finite Field Multiplier". As the name indicates that the values remain in a finite field. For one case the value range is from 0 to 255. Let

$$b = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0 \quad (2.6)$$

$$a = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 \quad (2.7)$$

Then the codeword coefficients are given below

$$C1 = b_1a^7 + b_2a^6 + b_3a^5 + b_4a^4 + b_5a^3 + b_6a^2 + b_7a^1 \quad (2.8)$$

$$C2 = b_2a^7 + b_3a^6 + b_4a^3 + b_5a^4 + b_6a^3 + b_7a^2 \quad (2.9)$$

$$C3 = b_3a^7 + b_4a^6 + b_5a^5 + b_6a^4 + b_7a^3 \quad (2.10)$$

$$C4 = b_4a^7 + b_3a^6 + b_2a^5 + b_1a^4 \quad (2.11)$$

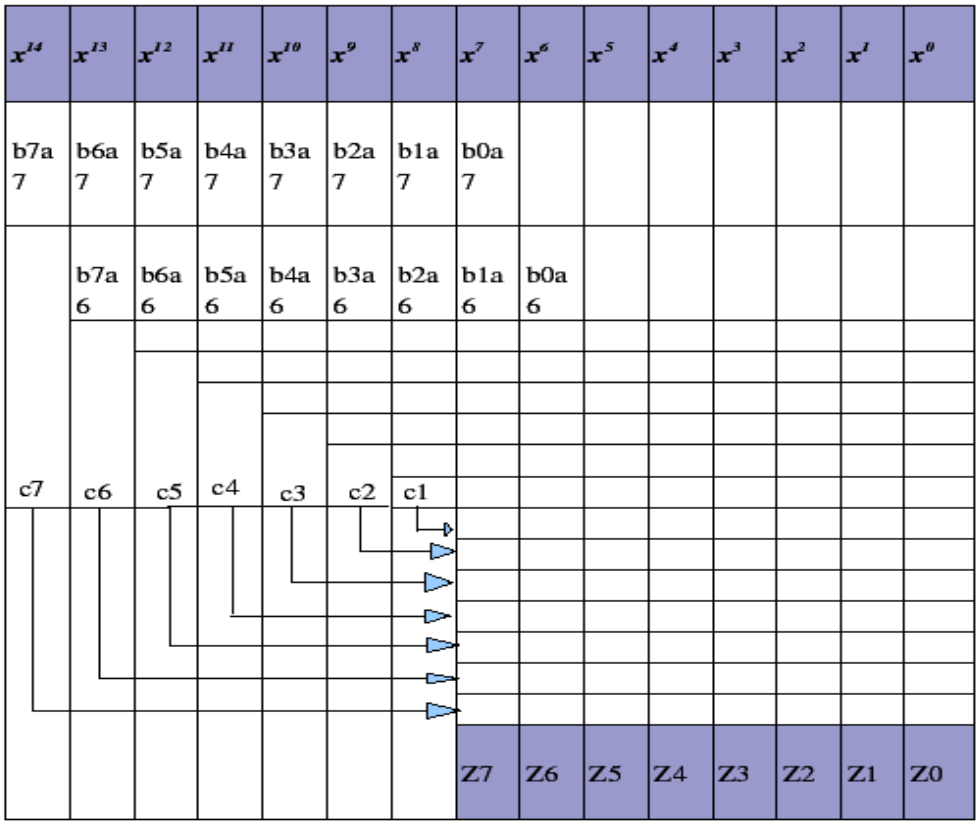
$$C5 = b_5a^7 + b_6a^6 + b_7a^5 \quad (2.12)$$

$$C6 = b_6a^7 + b_7a^6 \quad (2.13)$$

$$C7 = b_7a^7 \quad (2.14)$$

We can see the output from Table 2.1 as shown below

Table 2.1 Multiplication of GF multiplier



z0	b0a0 + c1 + c5 + c6 + c7
z1	b0a1 + b1a0 + c2 + c6 + c7
z2	b0a2 + b1a1 + b2a0 + c1 + c3 + c5 + c6
z3	b0a3 + b1a2 + b2a1 + b3a0 + c1 + c2 + c4 + c5
z4	b0a4 + b1a3 + b2a2 + b3a1 + b4a0 + c1 + c2 + c3 + c7
z5	b0a5 + b1a4 + b2a3 + b3a2 + b4a1 + b5a0 + c2 + c3 + c4

z6	b0a6 + b1a5 + b2a4 + b3a3 + b4a2 + b5a1 + b6a0 + c3 + c4 + c5
z7	b0a7 + b1a6 + b2a5 + b3a4 + b4a3 + b5a2 + b6a1 + b7a0 + c4 + c5 + c6

2.6 GF Adder

The adder computes the sum of two GF elements by XOR the corresponding bits of each symbol together. For example, the sum of α^3 and α^5 can be found, by hand, using polynomial notation. α^3 is equivalent to x + 1 and α^5 to $x^2 + \alpha x + 1$. Because of the modulo properties of GFs, addition is the same as subtraction. Two galois field elements are added by modulo-two addition of the coefficients or in binary form, producing the bit by bit exclusive-OR function of two binary numbers. Thus $\alpha^3 + \alpha^5 = \alpha^2$. This operation would be computed in hardware by XOR the bits of the two symbols together as follows.

If $b = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0$ (2.15)

$$a = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0 \quad (2.16)$$

Then, Output (Z_0 - Z_7) of GF adder will be as below

$$Z_0 = b_0 \wedge a_0 \quad (2.17)$$

$$Z_1 = b_1 \wedge a_1$$

$$Z_2 = b_2 \wedge a_2 \quad (2.18)$$

$$Z_3 = b_3 \wedge a_3 \quad (2.19)$$

$$Z_4 = b_4 \wedge a_4 \quad (2.20)$$

$$Z_5 = b_5 \wedge a_5 \quad (2.21)$$

$$Z_6 = b_6 \wedge a_6 \quad (2.22)$$

$$Z_7 = b_7 \wedge a_7 \quad (2.23)$$

Chapter -3

Reed Solomon Decoder

3.1 Introduction

The reed solomon decoder tries to correct errors and/or erasures by calculating the syndromes for each codeword. Based upon the syndromes the decoder is able to determine the number of errors in the received block. If there are errors present, the decoder tries to find the locations of the errors using the berlekamp massey algorithm by creating an error locator polynomial. The roots of this polynomial are found using the chien search algorithm. Using forney's algorithm, the symbol error values are found and corrected. For an RS (n, k) code where n - k = 2T, the decoder can correct up to T symbol errors in the code word. Given that errors may only be corrected in units of single symbols (typically 8 data bits).

3.2 Reed Solomon Decoder Hardware

3.2.1 Block Description

The purpose of the decoder is to process the received code word to compute an estimate of the original message symbols.

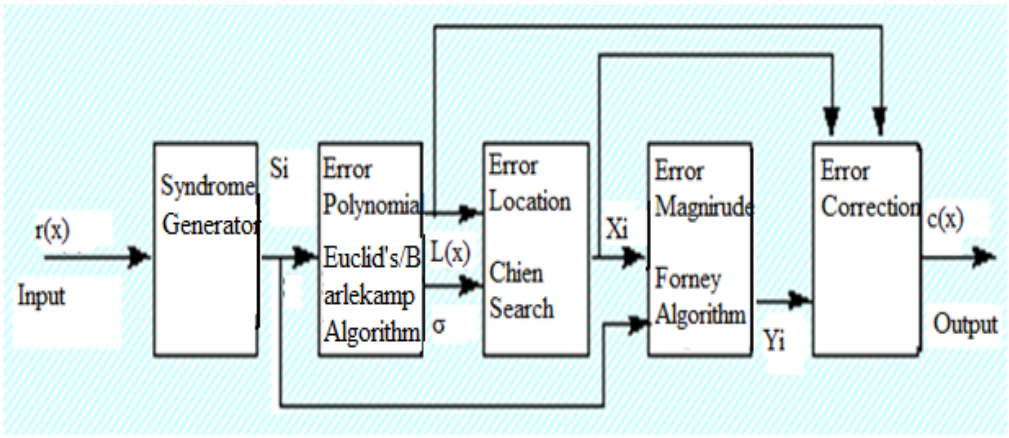


Fig 3.1 RS Decoder Block

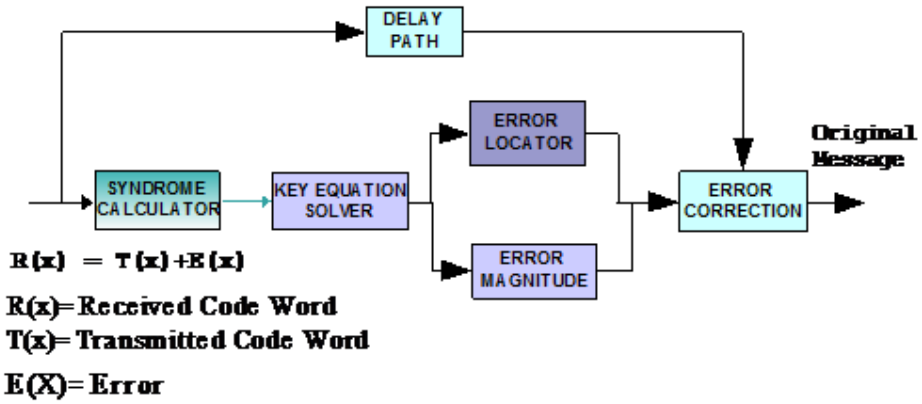


Fig 3.2: Reed Solomon Decoder Block Diagram in Another Form

There are three main blocks to the decoder first is syndrome generator, then euclid's/barlekamp massey algorithm and the chien/forney block. The output of the chien/forney block is an estimate of the error vector. This error vector is then added to the received codeword to form the final codeword estimate. A top-level block diagram of the decoder is shown in Fig3.2. Note that the error value vector Y comes out of the chien/forney block in reverse order, and it must pass through a LIFO/FIFO block before it is added to the received codeword R(x) [13].

Where R (x) is received code word, S_i is syndromes, L(x) is error locatorp polynomial x_i is error locations, Y_i is error magnitudes, C(x) is recovered code word and V is no of errors,

3.2.2 Syndrome Generator

The first step in the decoder is to compute the syndrome. The syndrome consists of n - k symbols and the values are computed from the received code word. The syndrome depends only on the error vector, and is independent of the transmitted code word. That is, each error vector has a unique syndrome vector. However, many different received code words will have the same syndrome if their error pattern is the same. The purpose of computing the syndrome first is that it narrows the search for the actual error vector. Originally, a total of 2^n possible error vectors would have to be searched. However, by finding the syndrome first, this search is narrowed down to looking at just 2^{n-k} possibilities. The syndrome can be computed mathematically by dividing the received code word by the generator polynomial using GF algebra. The remainder of this division

is called the syndrome polynomial s(x). The actual syndrome vector S(x) is computed by evaluating s(x) at a through α^{n-k}. However, this method is not efficient from a hardware standpoint. The alternative method typically used in hardware is to directly evaluate the received code word R(x) at a through α^{n-k}.

The syndrome generator module computes the syndrome S by evaluating the received code word R(x) at through α^{n-k}. That is, R(α) through R(α^{n-k}). In the RS code n - k = 2t, and thus there are 2t syndrome values to compute [S1 S2 S3…S(2t)]. These values are computed in parallel as shown in fig.(3.2). The first syndrome generator evaluates the received code word at to form S1, the next generator evaluates the received code word at a^2 to form S2, and so on. The syndrome generator module will also contain hardware that checks for an all-zero syndrome. If all of the syndrome values are zero, then there are no errors and the euclid's algorithm block and the chien/forney block are disabled. The received code word then becomes the codeword estimate.

A reed solomon codeword has 2t syndromes that depend only on errors (not on the transmitted code word). If there are any errors in the received code word then it will detect them. The syndromes can be calculated by substituting the 2t roots of the generator polynomial g(x) into r(x). In other words as the transmitted code word is always completely divisible by the generator polynomial, so this property extends to the individual factors of the generator polynomial therefore the transmitted code word be completely divisible by each factors without remainder. So the first step in Decoding is to divide the received polynomial by each of the factors of the generator polynomial.

$$R(x) / (x + a^i) = Q_i (x) + S_i / (x + a^i) \qquad \text{for } 0 <= I <= 2t - 1 \qquad (3.1)$$

Where Qi = Quotient

$(x + a^i)$ = Factor of the generate polynomial

α^i = Root of the primitive polynomial

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \qquad (3.2)$$

These remainder S_i resulting from these divisions are known as syndromes. So the syndromes are $S_0, S_1, S_2, \ldots\ldots\ldots\ldots, S_{2t-1}$.

From the equation (1), after rearranging it

$$S_i = Q_i(x) * (x+ \alpha^i) + R(x) \qquad (3.3)$$

So, when $x = \alpha^i$ the above equation reduces to

$$S_i = R(\alpha^i)$$

$$= R_{n-1}(\alpha^i)^{n-1} + R_{n-2}(\alpha^i)^{n-2} + R_{n-3}(\alpha^i)^{n-3} + \ldots\ldots\ldots\ldots\ldots + R_1(\alpha^i) + R_0 \qquad (3.4)$$

Where R_{n-1}, R_{n-2} ....., R_0 are the symbols of received code word. So the syndrome values can be obtained by substituting $x = \alpha^i$ in the received polynomial.

3.2.3 Syndrome Calculator

The hardware arrangement used for syndrome calculation is shown in figure (3.2) .Here the circuit is a direct implementation of the horner's method. The incoming symbol value is added and then multiplied by α^i . All the n symbols of the code word have to be accumulated before the syndrome value is produced. In order to achieve maximum speed, 2t circuits are required as the operation of calculating the 2t syndrome values is performed parallel.

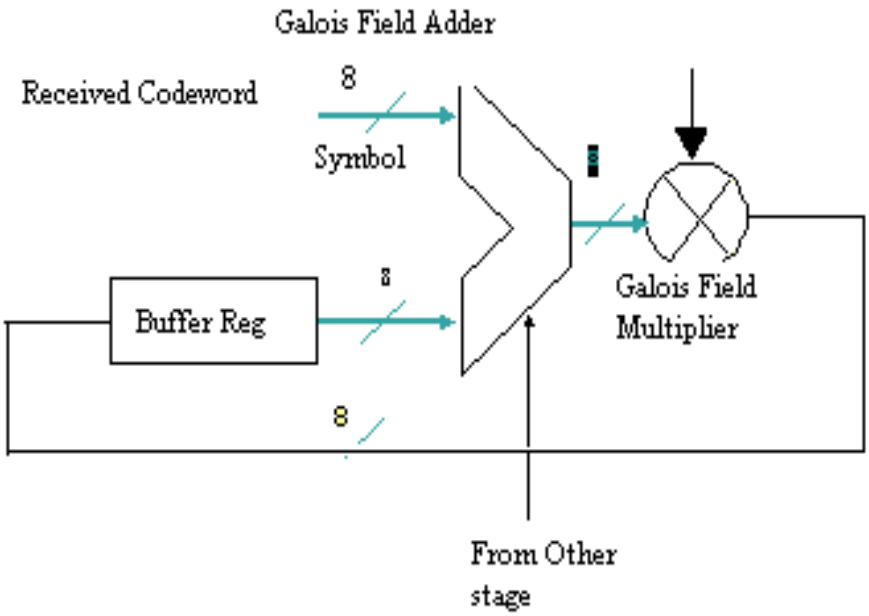


Fig 3.3: Syndrome Calculator

The clk is responsible for driving flip-flop that had the codeword symbols as inputs. For instance, the clk, clocked the received symbols into the decoder and the decoded symbols out of the decoder. The length of time required for each pipeline stage was 255 clk cycles. This was the amount of time required by the syndrome block to give it output syndrome

3.2.4 Horner's Method

The equation (3.2) can be re-written as

$$S_i = (((..(.....(R_{n-1}\,\alpha^i + R_{n-2})\,\alpha^i + ...+ R_1)\,\alpha^i + R_0 \qquad (3.5)$$

As shown in the above equation (3.4), the process starts by multiplying the first coefficient R_{n-1} by α^i . Then each subsequent coefficient is added to the previous product and the resulting sum multiplied by α^i untill finally R_0 is added. The advantage of this process is that the multiplication is always by the same value α^i at each stage.

3.2.5 Reed Solomon Decoding Algorithms

The most challenging aspect of RS codes is finding efficient techniques for decoding the received symbols. There are several problematic issues that arise with decoding RS codes. For instance how to minimize the occurrence of undetectable errors and how to reduce the number of decoder failures. Research in this area has led to the development of a few relatively reliable decoding algorithms. This section succinctly discusses these algorithms. Hence, the goal here is to provide a basic understanding of the mathematical steps each algorithm requires, rather than a theoretical in depth explanation.

Before the ensuing discussion, some concepts are required to be defined for some of the algorithms. First the error vector be $E = (E_0, E_1 \ldots E_{n-1})$ which has a polynomial representation[14].

$$E(x) = E_0 + E_1\,x + E_2\,x^2 + ...+ E_{n-1}x^{n-1} \qquad (3.6)$$

If the original encoded codeword is C and it has a corresponding polynomial C(x), then the received polynomial at the decoder is

$$R(x) = C(x) + E(x) = R_0 + R_1\,x + R_2\,x^2 + ...+ R_{n-1}x^{n-1} \qquad (3.7)$$

This polynomial can be evaluated at $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ which are the roots of the generator

polynomial g(x). Therefore, a set of 2t equations must be solved in order to obtain what are known as the syndromes S_i, where i = 1,2,3,.....,2t. Where

$$S_i = R_{n-1}(\alpha^i)^{n-1} + R_{n-2}(\alpha^i)^{n-2} + R_{n-3}(\alpha^i)^{n-3} + ...+ R_1(\alpha^i) + R_0 \quad (3.8)$$

The syndrome polynomial can then be defined as:

$$S(x) = S_{2t-1} x^{2t-1} + S_{2t-2} x^{2t-2} + ...+ S_1 x + S_0 \quad (3.9)$$

When all the syndrome values are zero then the received codeword is a correct codeword. If any of the syndrome value is nonzero then there is error in the received codeword.

These syndrome values are then input of the berlekamp massey algorithm. This Key equation solver (KES) block gives two polynomial. First is error evaluator polynomial and second one is error position polynomials. These polynomials are used to calculate the error locations and error values and correct the errors as the received word is being read out of the decoder.

$$\Lambda(x)\, S(x) = \Omega(x) \bmod x^{2t} \quad (3.10)$$

Where $\Lambda(x) = 1+\lambda_1 x+ \lambda_2 x^2+ \lambda_3 x^3+ \ldots\ldots\ldots\ldots.+ \lambda_e x^e$ (3.11)

$$\Omega(x) = \omega_0 + \omega_1 x + \omega_2 x^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + \omega_{e-1} x^{e-1} \quad (3.12)$$

Where if e <= t, then error can be corrected.

if e > t, then error can't be corrected.

The BM (Berlekamp-Massey) algorithm is an iterative procedure for solving key equation (2.9). The error locator polynomial computation and error evaluator polynomial computation takes place in this BM algorithm and performs the evaluation of the error locator polynomial coefficients Λ(x) and error evaluator polynomial coefficients Ù(x).

3.2.6 Euclid's Algorithm

Euclid's algorithm processes the syndrome S(x) to generate the error locator polynomial Λ(x) and the error magnitude polynomial Ω(x). That is, it solves the following equation that is referred to as the key equation.

$$\Lambda(x)\,[1 + S(x)] = \Omega(x) \bmod x^{2t+1} \qquad (3.13)$$

The algorithm used in RS decoding is based on euclid's algorithm for finding the greatest common devisor (GCD) of two polynomials. Euclid's algorithm is an iterative polynomial division algorithm (for the actual steps, contact the author).

3.2.7 Chien Search Algorithm

Once the error locator polynomial $\Lambda(x)$ has been computed, it needs to be evaluated to find its roots. The chien search (CS) algorithm is used to find these roots (fig 3.2) The CS is a brute force algorithm that evaluates the polynomial for all possible input values, and then checks to see which outputs are equal to zero. If an error occurs in position i, then the following equation equals zero. Where i = 0 .. (n - 1).

The CS evaluates the above equation for all the values of i and j and counts the number of times that the equation is equal to zero. The locations of the zeros are the error locations, and the number of zeros is the number of symbols in error. There are (t + 1) stages of the CS that are implemented in hardware. Each of these stages (where a stage consists of a multiplier, mux and register) represents a different value for j in the above CS equation. The search is run for n clock cycles (each clock cycle represents a different value of i in the above equation) and the output of the adder is examined to see if it is equal to zero. If it is equal to zero, the zero detect block will output a 1, otherwise, it will output a zero. The output of the chien search block is thus a string of n bits that have values of either 0 or 1. Each 1 represents the location of a symbol in error. For the first clock cycle, the mux will route the error locator polynomial coefficient into the register. For the remaining (n - 1) clock cycles, the output of the multiplier will be routed via the mux into the register. The exponents of the multipliers have negative values. However, these values can be precomputed using the modulo operator. The exponent of i is equal to (- i modulo n) = (-i modulo 255). For example, α^{-1} equals α^{254},α^{-2} equals α^{253} and so on.

3.2.8 Error Magnitude Algorithm (Forney Algorithm)

The forney algorithm is used to compute the error values Y_i. To compute these values, forney algorithm needs the error locator polynomial $\Lambda(x)$ and error magnitude polynomial $\Omega(x)$. The equation for the error values is for $x=\alpha^{-1}$ where α^{-1} is a root of $\Lambda(x)$. The computation of the formal derivative $\Lambda'(x)$ is actually quite simple.

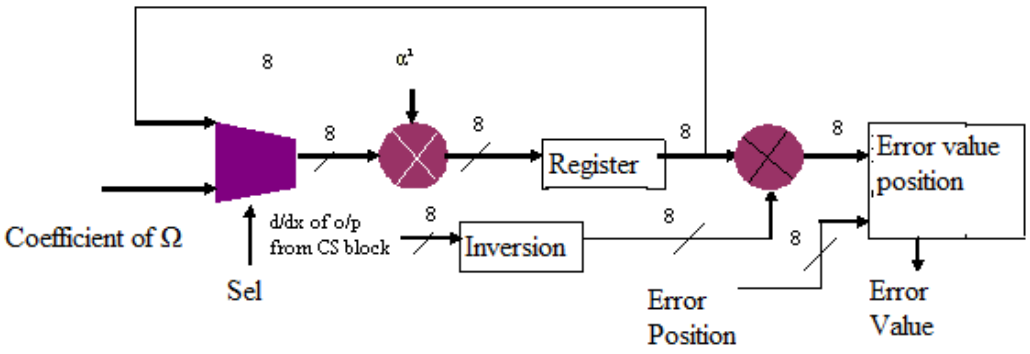


Fig 3.4 Error Magnitude Block

The derivative is formed by taking the coefficients of the odd powers of x, and assigning them to the next lower power of x (which will be even). However, in hardware, it is actually easier to find xΛ'(x). This x has the effect of shifting the derivative coefficients to the next higher power of x (i.e., the power of x that the coefficient originally belonged to). Thus, the denominator of the forney algorithm is found as follows using the same Λ(x) as the above example. In hardware, this x Λ'(x) term is found by zeroing out every other term of the original Λ(x) polynomial. If the denominator of the forney equation is modified by multiplying by the x term, then the numerator must also be multiplied by x in order for the equation to still work. Thus, the actual forney equation implemented in hardware is that the x Ω(x) polynomial is then evaluated along with the x Λ'(x) polynomial using the same type of hardware as used for the CS. However, in order to form x Ω(x), the coefficients of Ω(x) are shifted to the left by one location. To evaluate Ω(x), the Ω_0 coefficient would be added with the Ω_1 coefficient times α^{-1}, the Ω_2 coefficient times α^{-2} all the way up to the Ω_t coefficient times α^{-t}. To evaluate x Ω(x), the Ω_0 coefficient is multiplied by α^{-1}, the Ω_1 coefficient by α^{-2} all the way up to multiplying the Ω_t coefficient times $\alpha^{-(t+1)}$. The output of these multipliers is then summed. The numerator is then multiplied by the denominator using an inverse multiply. The inverse multiply contains a lookup table that finds the inverse of the denominator. For example, if the denominator was α^3, the inverse is α^{-3}. This can then be expressed as:

$$\alpha^{-i} = \alpha^{(-i \bmod n)} = \alpha^{(-3 \bmod 255)} = \alpha^{252} \qquad (3.14)$$

Since the same type of hardware is needed for both the chien search and the forney algorithm, the two functions can be combined in the same block. The chien search is

shown in the top of fig (3.2). The output of the adder for the odd stages is also used in the forney algorithm, shown in the lower part of the fig (3.2). The sum of the odd stages represents the denominator of the forney equation. This value is inversed in the inverse multiply block and then multiplied by the numerator value that is formed from evaluating the error magnitude polynomial. The output is AND with the zero detect output since the error values are only valid for the actual error locations (and they should be set to zero otherwise).

A decoder failure is detected by comparing the degree of the error locator polynomial $\Lambda(x)$ to the number of errors found by the chien search module. If the two values are not equal, then a decoder failure has occurred, the decoder failure flag is asserted and the codeword estimate is the received codeword. The degree of $\Lambda(x)$ is the power of the highest non-zero coefficient of the polynomial. Note that the decoder failure detection is not guaranteed to work if there are more than t errors. Most of the time, it will detect when more than t errors have occurred, but there will still be some cases that are not detected.

3.2.9 Error correction

The output of the chien/forney block is the error vector.

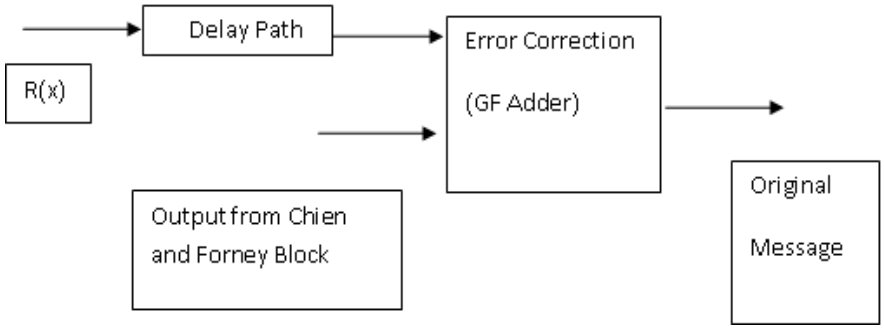


Fig 3.5: Error Correction Block

This vector is the same size as the codeword. The vector contains non zero values in locations that correspond to errors. Because the error vector is generated in the reverse order of the received codeword, a FIFO must be applied to either the received codeword or the error vector to match the order of the bytes in both vectors. The output of the adder is the decoder's estimate of the original codeword[]

3.3 FIFO

FIFO is an acronym for **First In, First Out**. This expression describes the principle of a queue or first-come, first-served (FCFS) behavior, what comes in first is handled first, what comes in next waits until the first is finished, etc. Thus it is analogous to the behavior of persons queuing, where the persons leave the queue in the order they arrive. In hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be SRAM, flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size a dual-port SRAM is usually used where one port is used for writing and the other is used for reading. A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing.

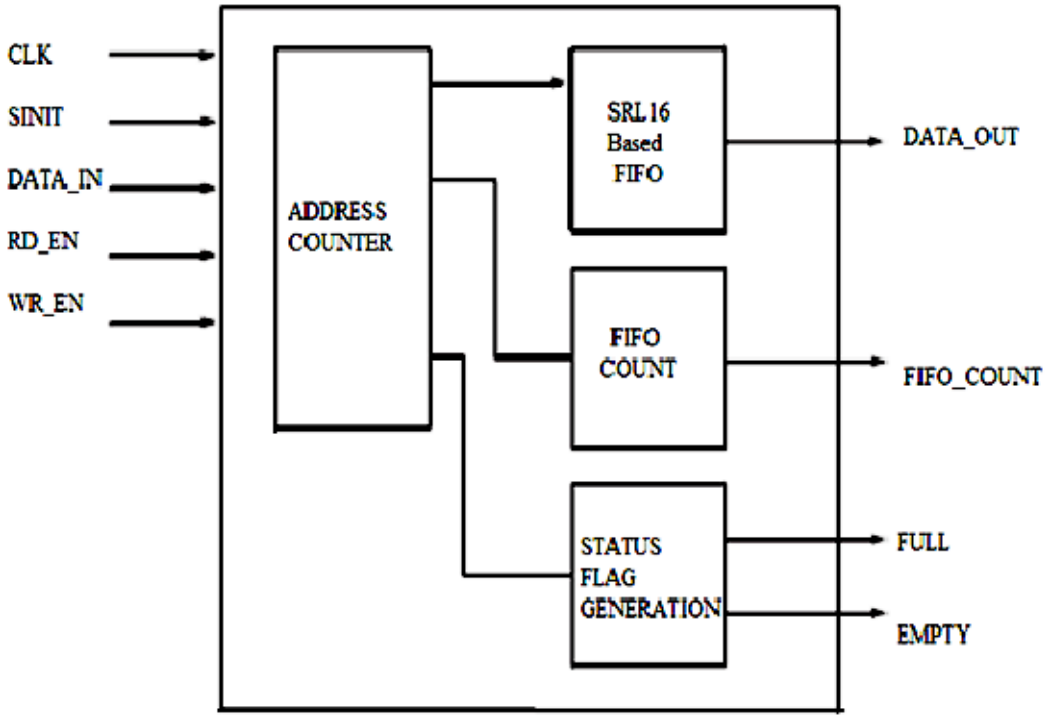


Fig 3.6: FIFO Using SRL16 Shift Register

Asynchronous FIFOs introduce met stability issues. A common implementation of an asynchronous FIFO uses a gray code (or any unit distance code) for the read and writes pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO

implementations. Conversely, one may use either a “leaky bucket” approach or pointer arithmetic to generate flags in synchronous FIFO implementations.

3.3.1 Block diagram of FIFO

There are five inputs and four outputs in FIFO.

CLK

CLK is for read and write operation.

SNIT

This signal is used to synchronization the FIFO.

DATA_IN

There are the input lines through which input is given.

WRITE_EN

This signal is driven high prior to a rising clock. Before performing a write check the FIFO to make sure it is not full. The address for the write is provided by the address counter after the data has been written the address counter increments the address by one.

READ_EN

This signal is driven high prior to a rising clock. Before performing a read check the FIFO to make sure it is not empty. During each READ, the address from the address counter is decremented by one.

FIFO_COUNT

To show how full the FIFO is. The address counter is used to represent the final address into which FIFO has been filled up.

DATA_OUT

There are the output lines through which outputs can be taken.

FULL FLAG

The full flag is generalized when the FIFO is full and no more data can be written into the FIFO. When write address register reaches to read address register, the FIFO triggers the FULL signal.

EMPTY FLAG

The empty flag become logic ‘1’ when there is no data to read. When read address register reaches to write address register, the FIFO triggers the Empty signal.

Chapter 4

Verification of RS Codes

4.1 Introduction

Reed solomon coding is a type of forward error correction that is used in data transmission (vulnerable to channel noise) plus data-storage and retrieval systems. Reed solomon codes (encoders/decoders) can detect and correct errors within blocks of data. Reed solomon codes operate on blocks of data, these codes are generally designated as (n, k) block codes, k is the number of information symbols input per block, and n is the number of symbols per block that the encoder outputs. Reed solomon decoder is useful in correcting burst noise; it replaces the whole word if it is not a valid code word.

4.2 Pin Diagram of RS Encoder

The following diagram illustrates RS encoder that incorporates 20 input and 16 output pins and their brief description is given below.

4.2.1 Input Pins

data_in

This is input signal and accepts 8 bits symbols.

rst

This is input signal. This is master reset, initialize all registers and port to zero.

clkin

This is input signal. This is master clock for the encoder.

g0 to g15

These are GF multiplier input which is given to RS encoder, each input is taken 8 bit.

valid

It will be high when valid data is present in the output port. If the data is not valid it will be de-asserted.

4.2.2 Output pins

q0 to q15

The encoded output is given out through these pins. Each output is taken 8 bit.

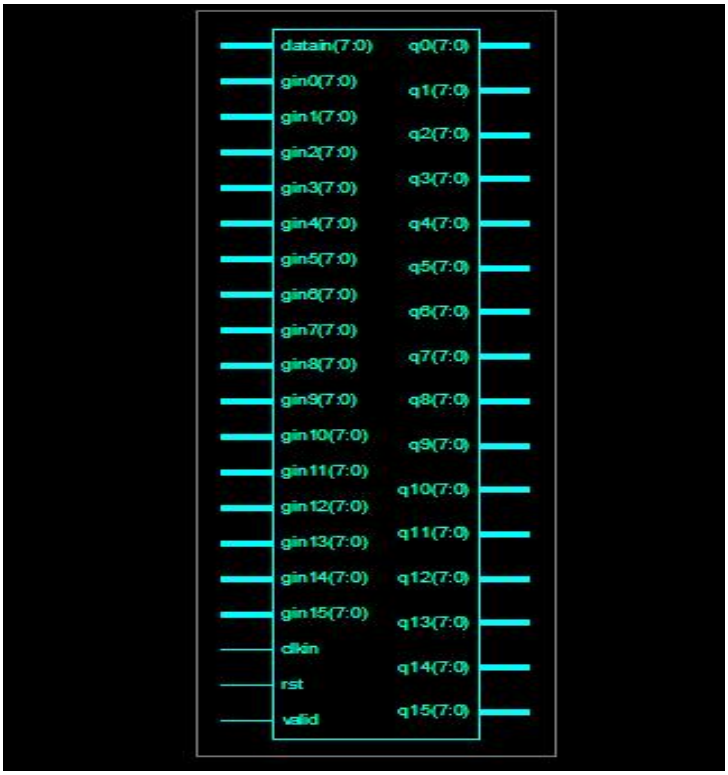


Fig 4.1: Pin Diagram of RS Encoder

4.3 RS Encoder Waveforms

Fig. (4.1-7) shows the simulation results for RS encoder including input signal valid. If valid signal is low that means data input is not valid, data is encoded only when this signal goes high. Moreover, all the inputs and outputs go low when reset is de-asserted. Inputs g_0 to g_{15} are outputs from code generator.

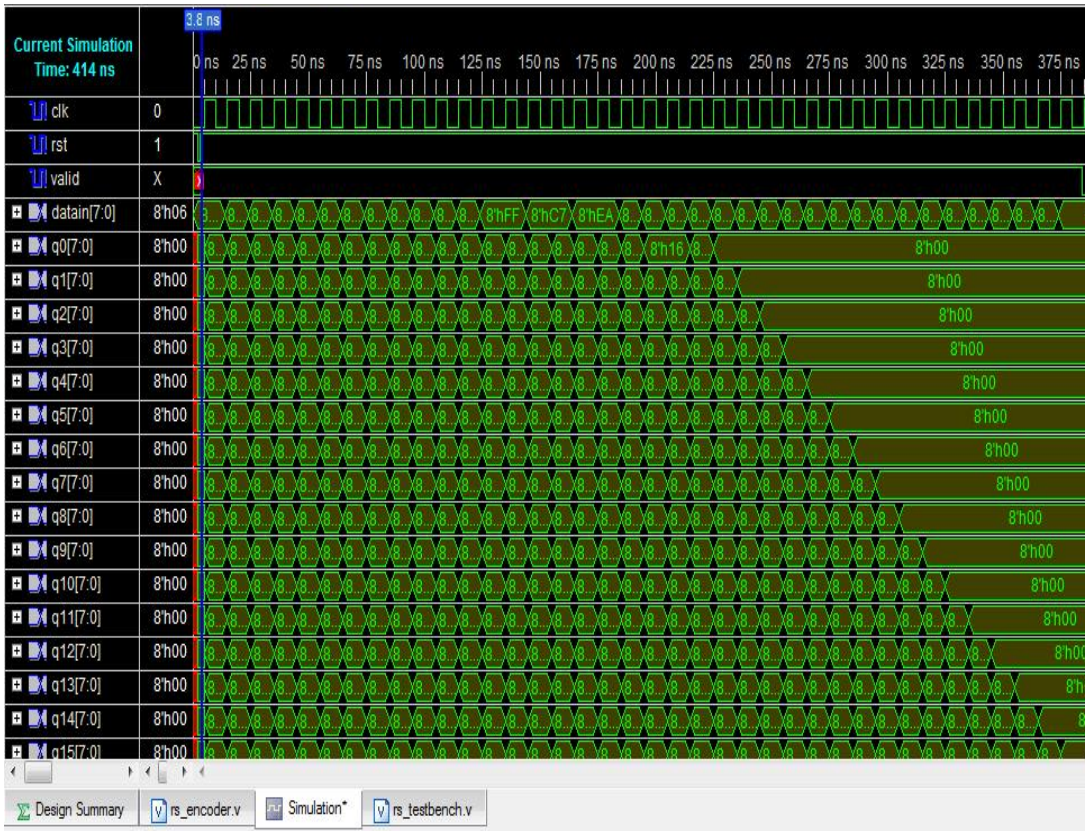


Fig 4.2: Encoded Outputs (1)

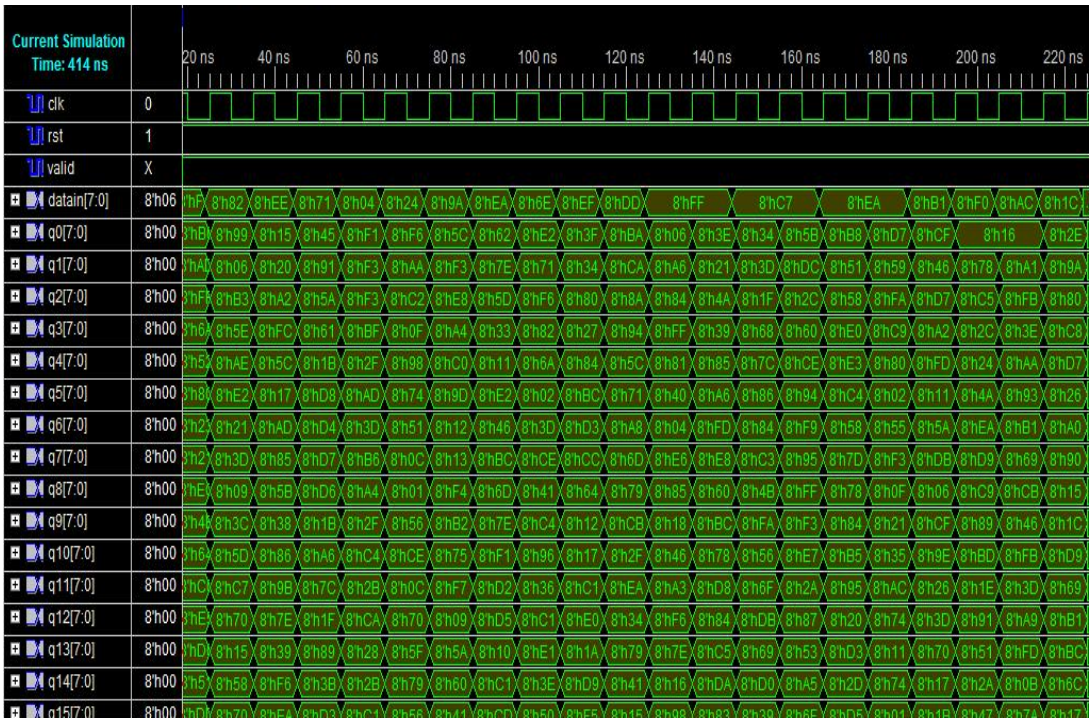


Fig 4.3: Zoomed Encoded Outputs (2)

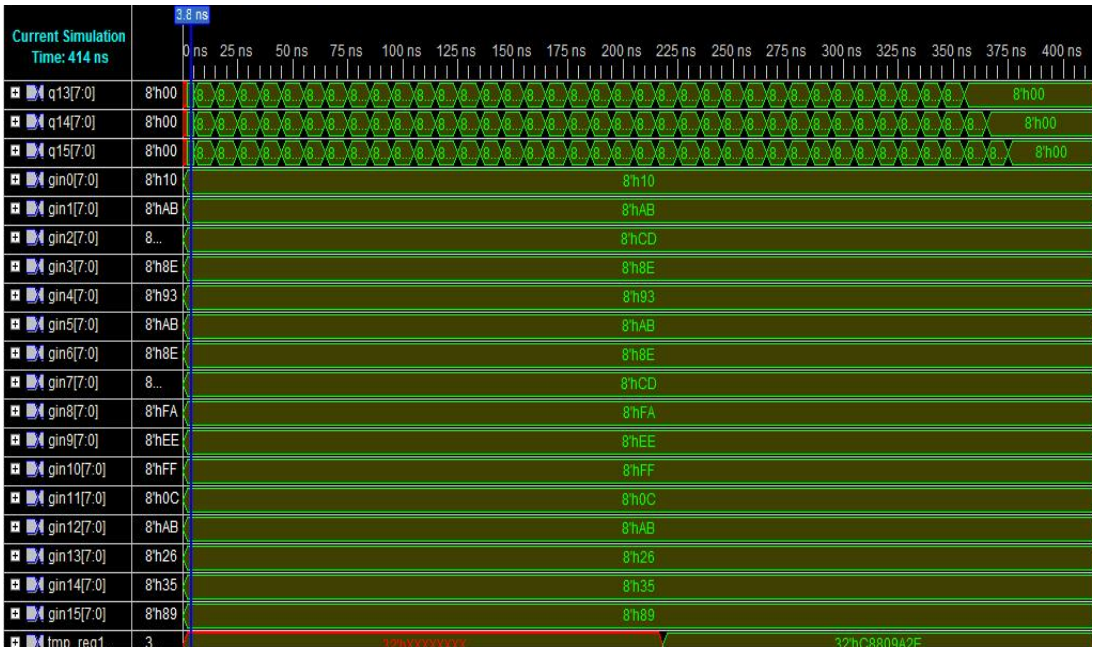


Fig 4.4: Encoded Outputs (3)

32-bit temporary storage registers are utilized to store the encoded output which is further assigned as an input to the data pins to facilitate the verification by avoiding requirement for alternate data inputs as shown in fig. 4.5.

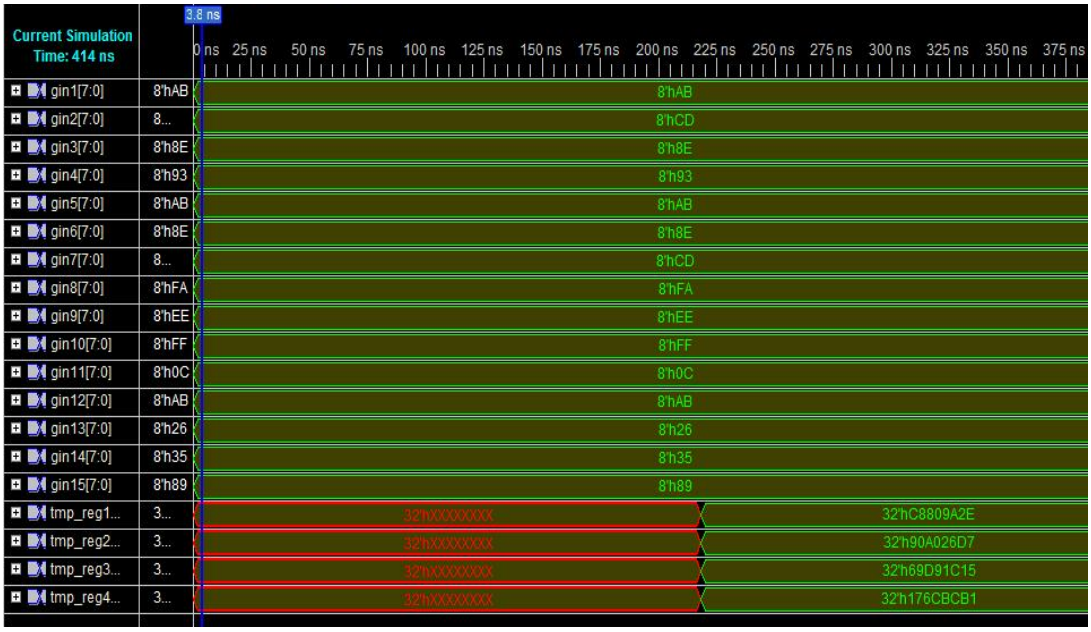


Fig 4.5: Shows Inputs Signals to GF multiplier and Temporary Register Contents.

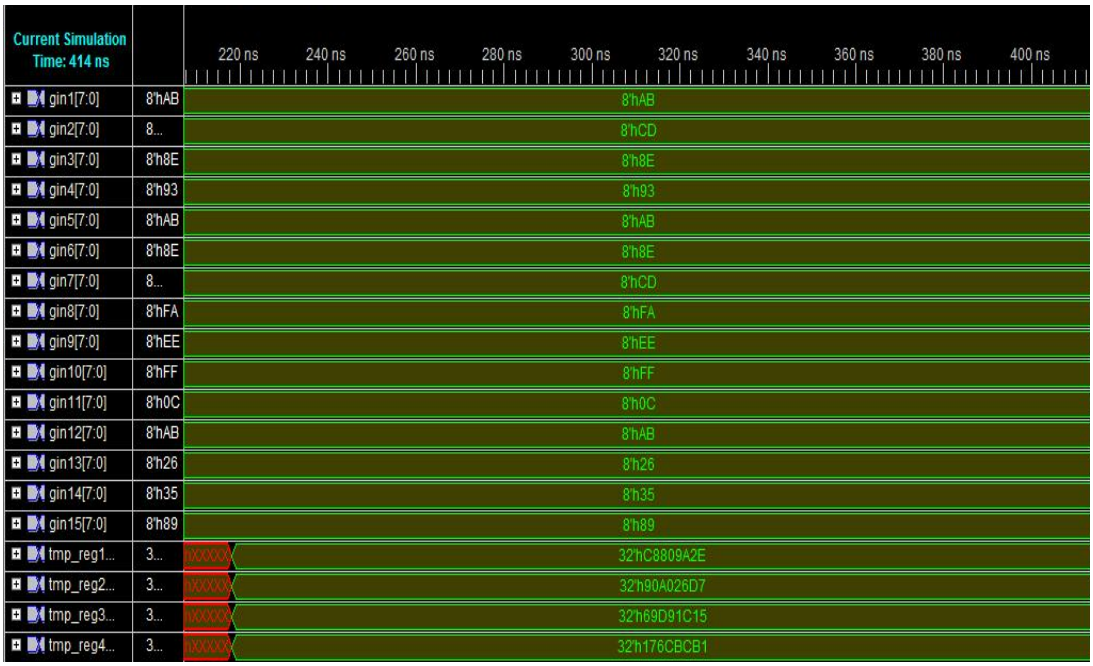


Fig 4.6: Encoded Outputs (4)

4.4 Pin Diagram of RS Decoder

The following diagram illustrates RS decoder that incorporates 4 input and 6 output pins and their brief description is given below.

4.4.1 Input Pins

recword

This is an input signal and accepts 7 bits symbols.

reset

This is a master reset that initializes all registers and ports to zero.

clock

This is an input signal that is a master clock for the encoder.

start

Decoder will start decoding the message just after the start is asserted.

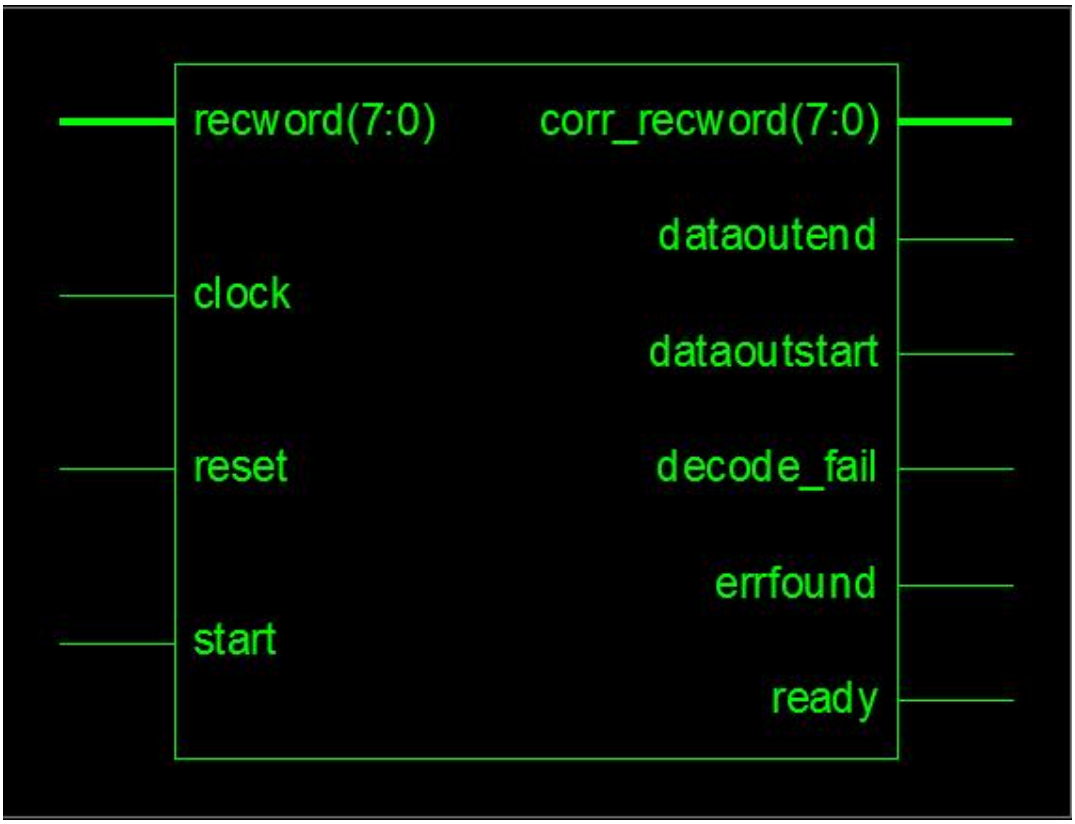


Fig 4.7: Pin Diagram of RS Decoder

4.4.2 Output pins

corr_recword

This is an 8 bits data output pin from a decoder circuit.

dataoutend

This flag indicates that the last symbol of data has been received at the output data bus.

dataoutstart

This flag indicates that the first symbol of data has been received at the output data bus.

ready

If set to 1, decoder is ready to take new data input.
If set to 0, decoder is not ready to accept any data input.

errfound

Set to 1, if any data byte gets corrupted.

decode_fail

Set to 1, if decode fails to correct the received word.

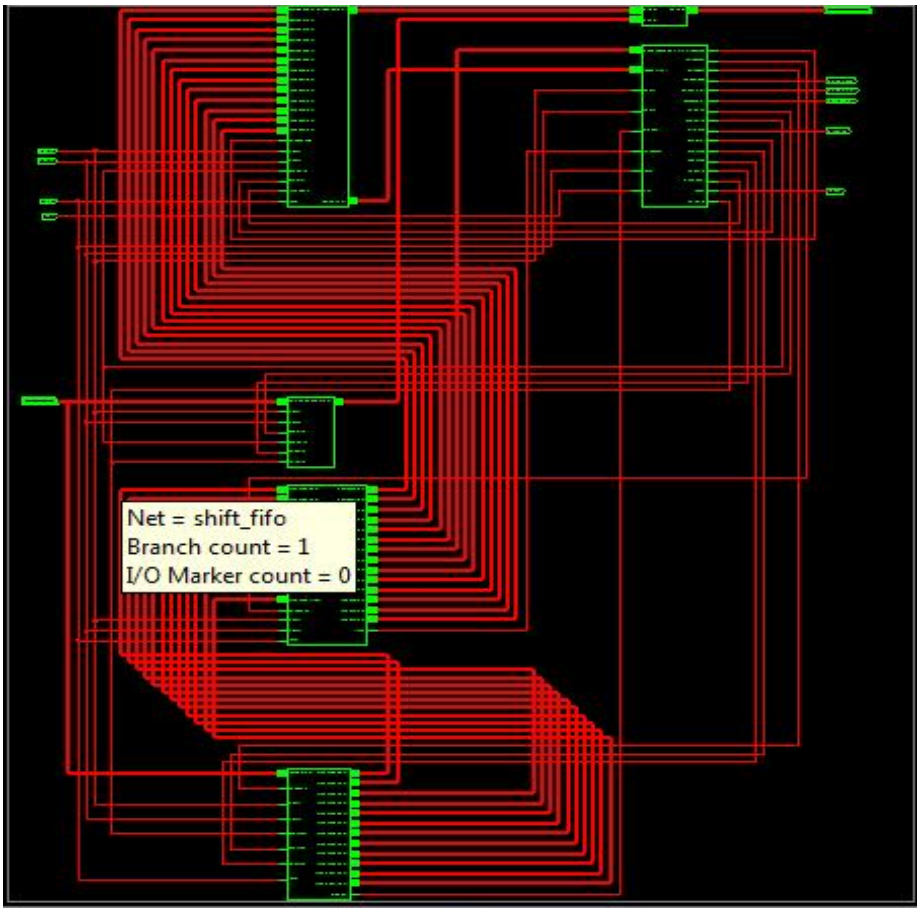


Fig 4.8: Internal Component of RS Decoder

4.5 RS Decoder Waveforms

Fig. (4.9-13) shows the simulation results for RS decoder including input signal start. Decoding will be started when start goes from high to low availing ready assertion. Decoding will be stopped immediately after the ready is de-asserted as shown in a fig (4.9-10).

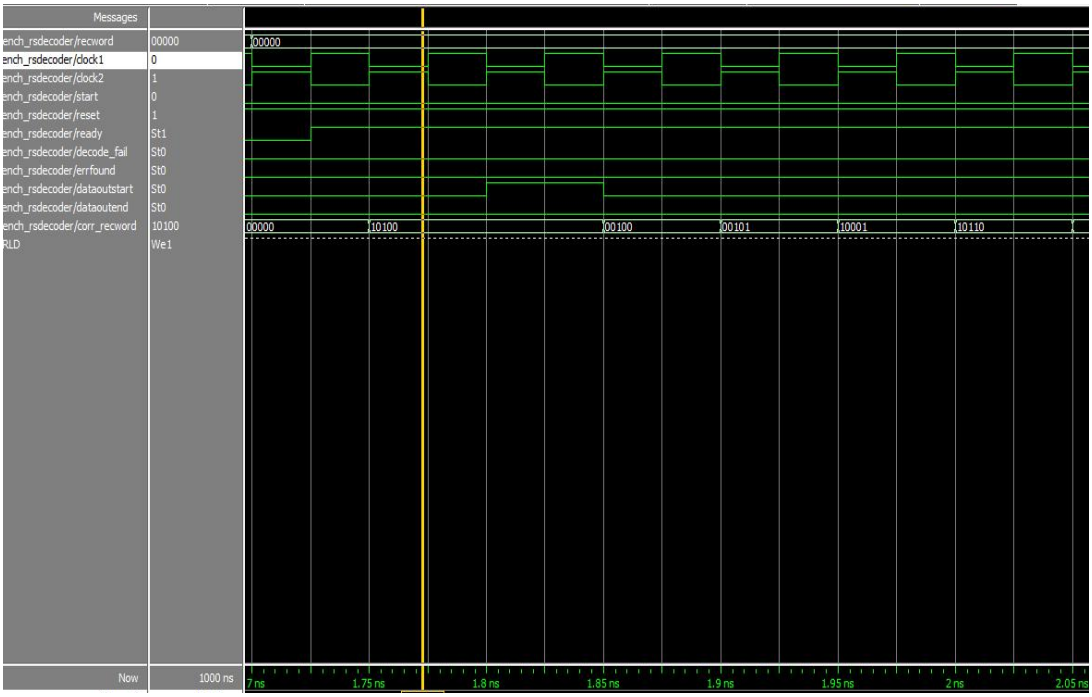


Fig 4.9: Output of RS Decoder (1)

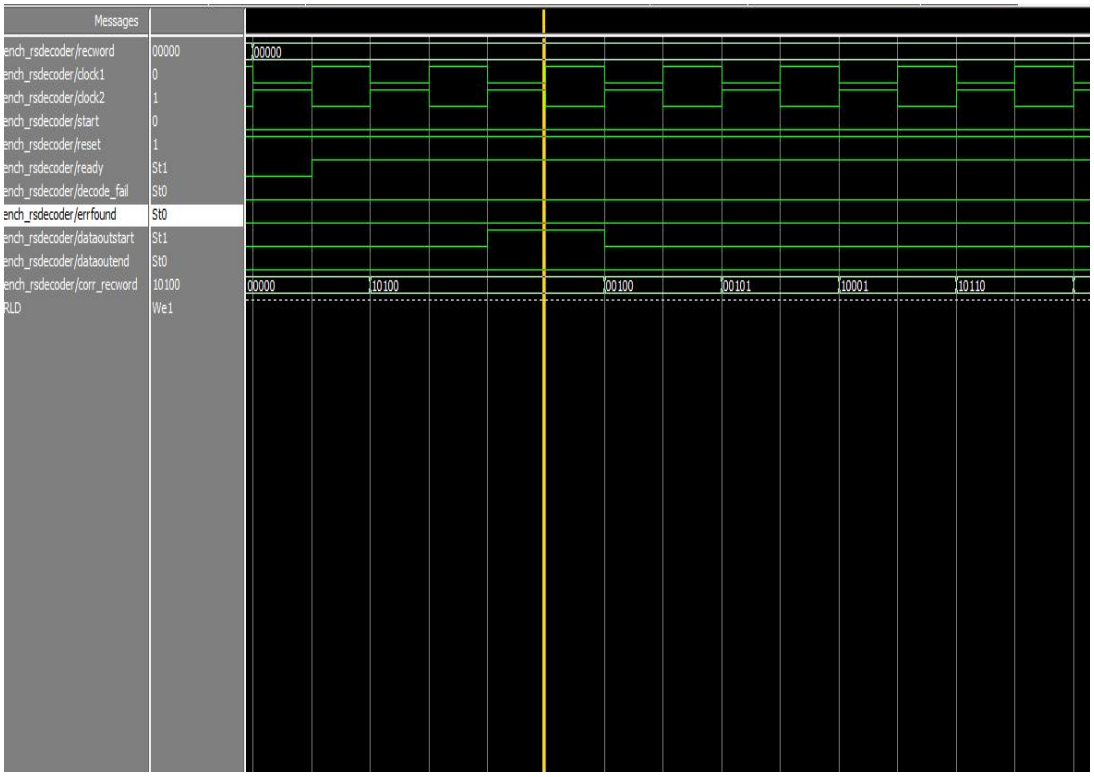


Fig 4.10: Output of RS Decoder (2)

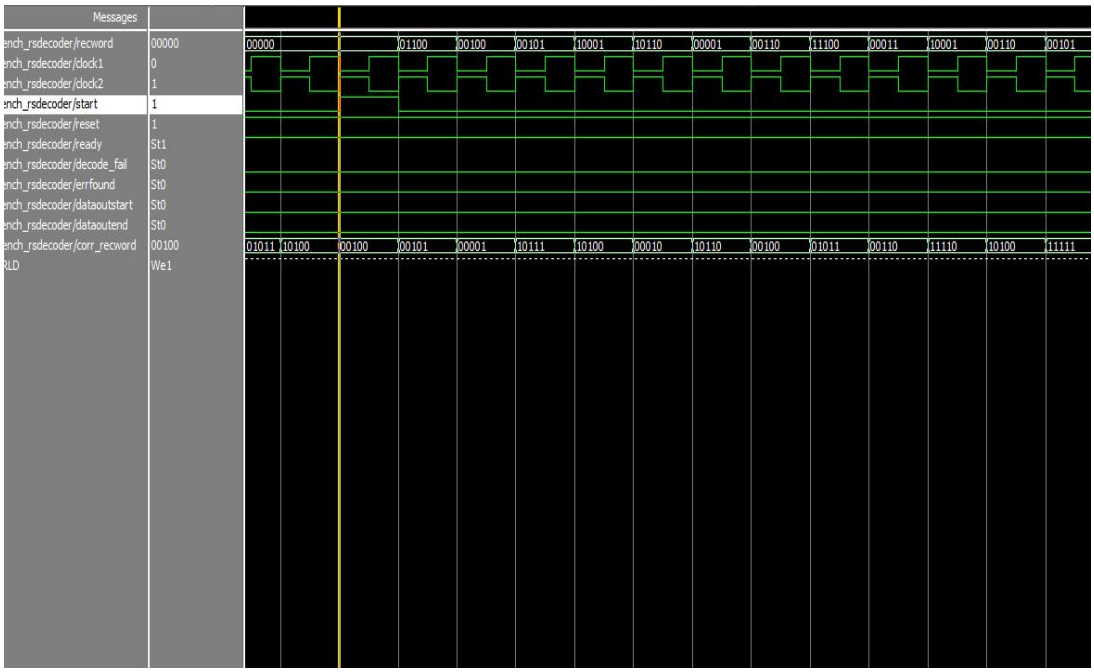


Fig 4.11: Output of RS Decoder (3)

All input and output signals are synchronized with clock 2. First received word contains no error symbol. Thus, all syndrome values will be zero and decoder will pass the received word. Second received word contains 6 error symbols. So decoder will make the required amendments and through it out.

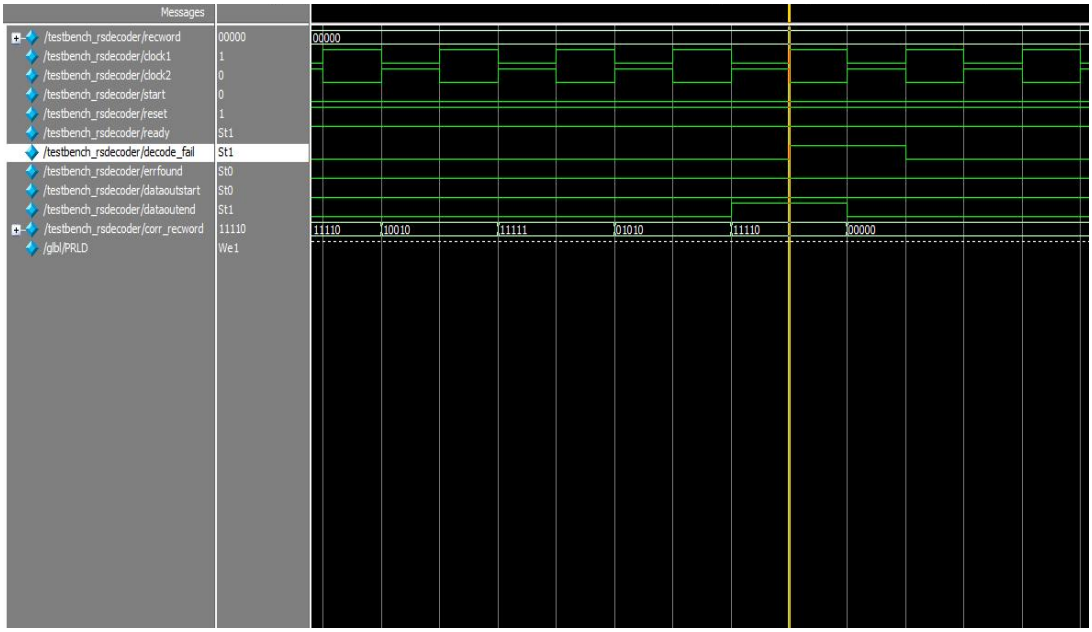


Fig 4.12: Output of RS Decoder (4)

Third received word contains 8 error symbols as shown in a fig. (4.12). Decoder can't make the errors correct because number of errors go beyond the limitation of RS decoder i.e. 6. It will signal decode_fail to 1.

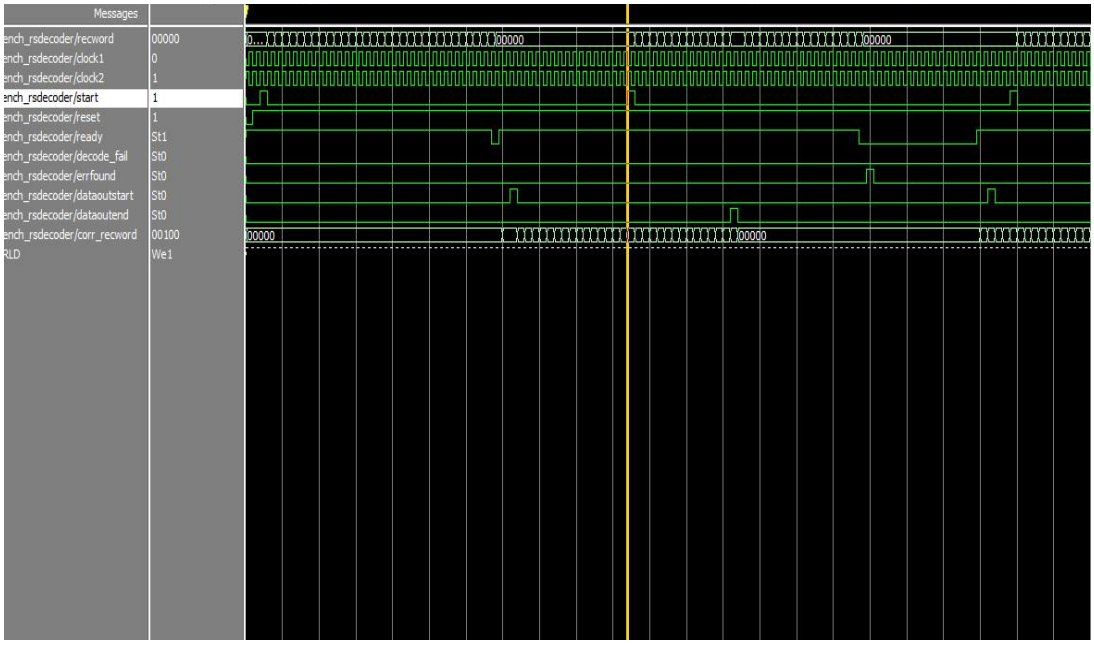


Fig 4.13: Output of RS Decoder (Zoomed out)

Above simulations signifies 3 different received word vectors. First received word contains no error symbol. Second word contains 6 error symbols and the last word contains 8 error symbols. The decoder has some limitations such as its flag decoding failure at the end of output. So, other block outside the decoder cannot differentiate between uncorrected word and corrected word until it receives decoding failure flag at the end of the word. Decoding failure is detected when degree of error location polynomial and number of its roots is not equal. It means the error location polynomial doesn't have roots in the underlying GF (2^5). To determine the roots, decoder must activate CSEE block first. Hence, decoding failure is detected after all elements in GF (2^5) have been evaluated. Uncorrectable word still have to be summed with wrong error values because decoding failure is detected at the end of word, there is no other mechanism to solve the problem, unless decoder start to output the word after all GF (2^5) elements has been evaluated.

Chapter -5

Implementation of RS codes using FPGA

5.1 FPGA Implementation

FPGA stands for field programmable gate arrays that can be configured by the customer or designer after manufacturing. Field programmable gate arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC. This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used. For example an FPGA may be used in a design that needs to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost. FPGAs are programmed using a logic circuit diagram or a source code in a hardware description language (HDL) to specify how the chip will work. FPGAs contain programmable logic components called "logic blocks" and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". The programmable logic blocks are called configurable logic blocks and reconfigurable interconnects are called switch boxes. Logic blocks (CLBs) can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

5.2 FPGA Architecture

Each FPGA vendor has its own FPGA architecture. The architecture consists of configurable logic blocks, configurable I/O blocks, and programmable interconnect. Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of programmable elements for an FPGA are Static RAM and anti-fuses. An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs and I/Os required is easily determined from the design, the number of

routing tracks needed may vary considerably even among designs with the same amount of logic.

5.3 The Design Flow

This section examines the flow for design using FPGA. This is the entire process for designing a device that guarantees that you will not overlook any steps and that you will have the best chance of getting backs a working prototype that functions correctly in your system. The design flow consists of the steps in

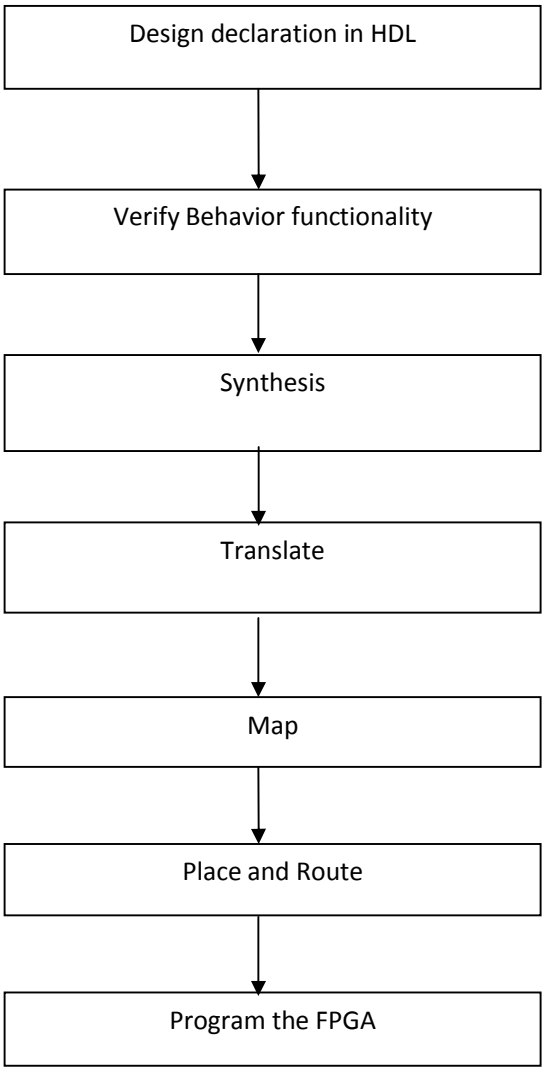


Fig 5.1 FPGA Design Flow

5.3.1 Design Entity

The basic architecture of the Reed Solomon Encoder/Decoder system is designed in this step which is coded in Verilog Hardware Description Language. VHDL, System verilog can also be used.

5.3.2 Behavioral Simulation

After the design phase, the RS code is verified using simulation software i.e. Xilinx ISE Simulator for different inputs to generate outputs and if it verifies then proceed further otherwise modification and necessary correction will be done in the HDL code. This is called as the behavioral simulation. Simulation is an ongoing process while the design is being done. Small sections of the design should be simulated separately before hooking them up to larger sections. There will be much iteration of design and simulation in order to get the correct functionality. Once design and simulation are finished, another design review must take place so that the design can be checked. It is important to get others to look over the simulations and make sure that nothing was missed and that no improper assumption was made. This is one of the most important reviews because it is only with correct and complete simulations that verify whether this chip will work correctly in the required system.

5.3.3 Design Synthesis

After the correct simulation results the design is then synthesized. During synthesis the Xilinx ISE tool does the following operation.

Design Information

Command Line : map -ise
C:/Users/raghav/Desktop/RS_Encoder/RS_Codes/rs_encoder/rs_encoder.ise -intstyle
ise -p xc3s500e-fg320-5 -cm area -pr off -k 4 -c 100 -o rs_encode_map.ncd
rs_encode.ngd rs_encode.pcf
Target Device : xc3s500e
Target Package : fg320
Target Speed : -5

Mapper Version : spartan3e -- $Revision: 1.46.12.2 $
Mapped Date : Sun Jul 12 14:44:25 2009

5.3.4 HDL Compilation

The tool compiles all the sub-modules in the main module (RS Codes) if having any problem regarding module, check the syntax of the code written for the design.

5.3.5 Design Hierarchy Analysis

Analysis the hierarchy of the RS Encoder/Decoder design.

5.3.6 HDL synthesis

Synthesizes the HDL code for RS Encoder/Decoder in terms of the components such as Multiplexer, Adder/Sub tractors, Counter, Register, Latches, Comparators, XORs tri state buffers, decoders etc.

5.3.7 Advanced HDL Synthesis

In low level synthesis, the blocks synthesized in the HDL synthesis and the Advanced HDL synthesis is further defined in terms of the low level blocks such as buffers, look up tables. It also optimizes the logic entities in the design by eliminating the redundant logic, if any. The tool then generates a 'netlist' file (NGC file) and then optimizes it. The final net list output file has an extension of .ngc. This NGC file contains both the design data and constraints. The optimization goal can be pre defined to be the faster speed of operation or the minimum area of implementation before running this process. The level optimization effort requires larger CPU times (i.e. the design time) because multiple optimization algorithms are tried to get the best result for the target architecture.

RS Encoder HDL Synthesis Report

HDL Synthesis Report

Macro Statistics
Registers : 16
8-bit register : 16

# Xors	: 273
1-bit xor10	: 16
1-bit xor11	: 16
1-bit xor2	: 48
1-bit xor3	: 32
1-bit xor4	: 32
1-bit xor5	: 16
1-bit xor6	: 32
1-bit xor7	: 16
1-bit xor8	: 16
1-bit xor9	: 32
8-bit xor2	: 17

5.4 Design Implementation

The design implementation process consists of the following sub process:

5.4.1 Translation

The translate process merges all of the input net list and design constraint outputs a Xilinx NGD (Native information and Generic Database) file. The .ngd file describes the logical design reduced to the Xilinx device primitive cells. The output in the floor planner tool supplied with Xilinx ISE software. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named .ucf (User Constraints File). Tools use to create or modify the UCF are PACE, Constraint Editor etc.

5.4.2 Mapping

The map process is run after the translate process is complete. Mapping maps the logical design described in the NGD file to the components/primitives (Slices/CLBs) present on the NCD file created by the Map process to place and route the design on the target FPGA design. In this process the whole circuit is divided into sub blocks so that they can fit into FPGA logic blocks. The logic defined in the input NGD file is mapped into

targeted FPGA elements i.e. CLBs and IOBs and an output NCD (native circuit description) file is generated which depicts the design mapped into the FPGA.

Map Report of RS Encoder

Release 10.1.03 Map K.39 (nt)

Xilinx Mapping Report File for Design 'rs_encode'

Design Summary

Number of errors: 0

Number of warnings: 0

Logic Utilization:

Number of Slice Flip Flops: 139 out of 9,312 1%

Number of 4 input LUTs: 1,276 out of 9,312 13%

Logic Distribution:

Number of occupied Slices: 669 out of 4,656 14%

Number of Slices containing only related logic: 669 out of 669 100%

Number of Slices containing unrelated logic: 0 out of 669 0%

*See NOTES below for an explanation of the effects of unrelated logic.

Total Number of 4 input LUTs: 1,276 out of 9,312 13%

Number of bonded IOBs: 267 out of 232 115% (OVERMAPPED)

Number of BUFGMUXs: 1 out of 24 4%

Peak Memory Usage: 180 MB

Total REAL time to MAP completion: 11 secs

Total CPU time to MAP completion: 8 secs

Map Report of RS Decoder

Release 10.1.03 Map K.39 (nt)

Xilinx Mapping Report File for Design 'RSDecoder'

Design Information

Command Line : map -ise C:/Users/raghav/Desktop/RS_Encoder/RS_Codes/rs_dec/rs_dec.ise -intstyle ise -p xc3s500e-fg320-5 -cm area -pr off -k 4 -c 100 -o RSDecoder_map.ncd RSDecoder.ngd RSDecoder.pcf

Target Device : xc3s500e

Target Package : fg320

Target Speed : -5

Mapper Version : spartan3e -- $Revision: 1.46.12.2 $

Mapped Date : Sun Jul 12 15:07:39 2009

Design Summary

Number of errors: 0

Number of warnings: 0

Logic Utilization:

Number of Slice Flip Flops: 517 out of 9,312 5%

Number of 4 input LUTs: 1,480 out of 9,312 15%

Logic Distribution:

Number of occupied Slices: 773 out of 4,656 16%

Number of Slices containing only related logic: 773 out of 773 100%

Number of Slices containing unrelated logic: 0 out of 773 0%

*See NOTES below for an explanation of the effects of unrelated logic.

Total Number of 4 input LUTs: 1,480 out of 9,312 15%

Number of bonded IOBs: 18 out of 232 7%

Number of BUFGMUXs: 1 out of 24 4%

Peak Memory Usage: 183 MB

Total REAL time to MAP completion: 9 secs

Total CPU time to MAP completion: 9 secs

5.4.3 Place and Route

After map the design is placed and routed. Place and Route **(**PAR) tool is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Example if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may affect some other constraint. So tradeoff between all the constraints is taken account by the place and route process .The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consist the routing information.

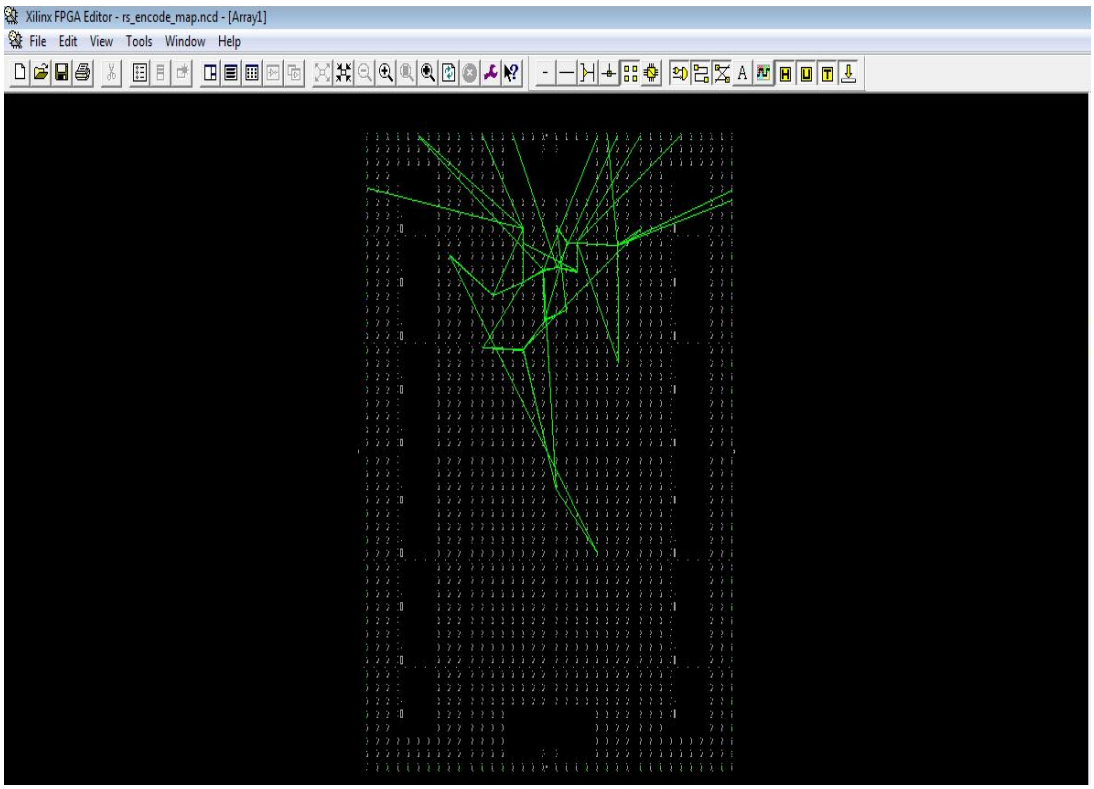


Fig 5.1: Manual Place and Route for RS Codes

During the place process the sub blocks are placed according to the logic but these blocks do not have physical routing among them and with I/O pads also but there is only a logical connection between them which can be clearly seen using the FPGA Editor just after the place process ends. These logical connections are shown by “rats nets” in FPGA Editor. Then the Route process is run which makes physical connections between the sub blocks placed on FPGA. The connections are made using the switch matrices.

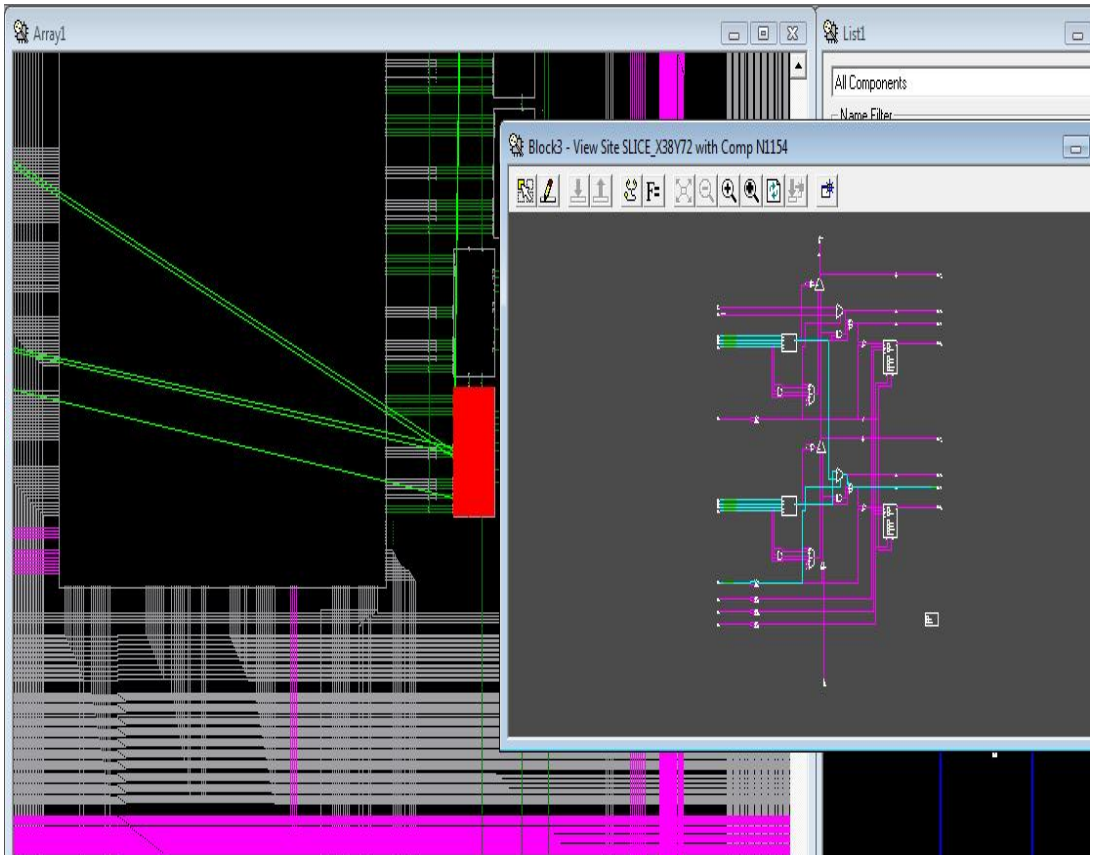


Fig 5.2: Internal Structure of Switch Matrix

5.4.4 Bitstream Generation

The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream," although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming." While in an instruction set processor the configuration data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase. A programming file is generated by running the Generate Programming File process. This process can be run after the FPGA design has been completely routed. The Generate Programming File process runs BitGen, the Xilinx bitstream generation program, to produce a bitstream (.BIT) or (.ISC) file for Xilinx device configuration. The FPGA device is then configured with the .bit file using the JTAG boundary scan method. After the Spartan device is

configured for the RS Encoder/Decoder design, then its working is verified by applying different inputs.

5.4.5 Functional Simulation

Post Translate (functional) simulation can be performed prior to mapping of the design. This simulation process allows the user to verify that design has been synthesized correctly and any differences due to the lower level of abstraction can be identified.

5.4.6 Static timing Analysis: Three types of static timing analysis can be performed that are given below.

(a) Post fit Static timing analysis: The timing results of the Post fit process can be analyzed. The analyze Post fit Static Timing process opens the Timing Analyzer window, which lets you interactively select timing paths in the design for tracing.

(b) Post map Static Timing Analyze: Post Map timing reports can be very useful in evaluating timing performance. Although route delays are not accounted for the logic delay can provide valuable information about the design. If logic delay account for a significant portion (> 50 %) of the total allowable delay of a path, the path may not be able to meet timing requirements when routing delays are added. Routing delay typically account for 45% to 65% of the total path delays. By identifying problem path, potential can be mitigated before investing time in place and route.

Logic path can be redesigned the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path. If logic only delays account for much less (>35%) than the total allowable delay for a path or timing constraint, then the place-route software can use very low placement effort levels. In these cases, reducing effort levels allow to decrease runtimes while still meeting performance requirements. Timing constraint can be altered. Clock period wizard is as shown in fig.5.3.

Timing Summary of RS Encoder:

Speed Grade: -5

Minimum period: 6.109ns (Maximum Frequency: 163.686MHz)

Minimum input arrival time before clock: 7.502ns

Maximum output required time after clock: 4.782ns

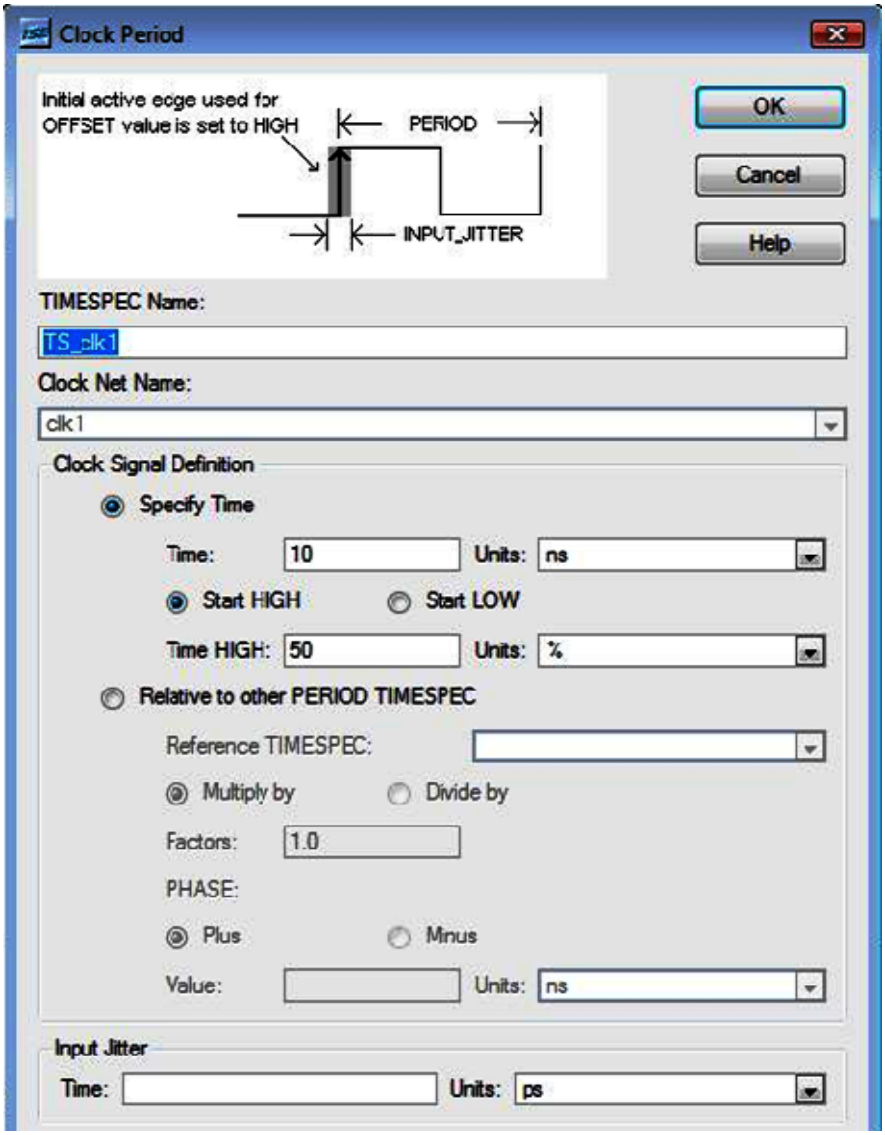


Fig 5.3 Clock Period Wizard

Maximum combinational path delay: No path found

Timing Summary of RS Decoder:

Speed Grade: -5

Minimum period: 6.936ns (Maximum Frequency: 144.181MHz)

Minimum input arrival time before clock: 6.041ns

Maximum output required time after clock: 5.792ns

Maximum combinational path delay: No path found

5.4.7 Testing

For a programmable device, It can be easily programmed and immediately have prototypes. After that to place these prototypes in the system and determine that the entire system actually works correctly. These steps make the chance that your system will perform correctly with only minor problems. These problems can often be worked around by modifying the system or changing the system software. These problems need to be tested and documented so that they can be fixed on the next revision of the chip. System integration and system testing is necessary at this point to insure that all parts of the system work correctly together. When the chips are put into production, it is necessary to have some sort of burn-in test of your system that continually tests your system over some long amount of time. If a chip has been designed correctly, it will only fail because of electrical or mechanical problems that will usually show up with this kind of stress testing.

5.5 Design Summary of RS Encoder

rs_encoder Project Status			
Project File:	rs_encoder.ise	Current State:	Synthesized
Module Name:	rs_encode	• Errors:	No Errors
Target Device:	xc3s500e-5fg320	• Warnings:	1 Warning
Product Version:	ISE 10.1.03 - Foundation Simulator	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	139	9,312	1%	
Number of 4 input LUTs	1,275	9,312	13%	
Logic Distribution				
Number of occupied Slices	669	4,656	14%	
Number of Slices containing only related logic	669	669	100%	
Number of Slices containing unrelated logic	0	669	0%	
Total Number of 4 input LUTs	1,275	9,312	13%	
Number of bonded IOBs				
Number of bonded	267	232	115%	OVERMAPPED
Number of BUFGMUXs	1	24	4%	

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Jul 12 03:34:50 2009	0	1 Warning	0
Translation Report	Out of Date	Sun Jul 12 02:37:32 2009	0	160 Warnings	0

Fig 5.4 Design Summary Report of RS Encoder

5.6 Design Summary of RS Decoder

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	517	9,312	5%	
Number of 4 input LUTs	1,480	9,312	15%	
Logic Distribution				
Number of occupied Slices	773	4,656	16%	
Number of Slices containing only related logic	773	773	100%	
Number of Slices containing unrelated logic	0	773	0%	
Total Number of 4 input LUTs	1,480	9,312	15%	
Number of bonded IOBs				
Number of bonded	18	232	7%	
Number of BUFGMUXs	1	24	4%	

rs_dec Project Status			
Project File:	rs_dec.ise	Current State:	Placed and Routed
Module Name:	RSDecoder	• Errors:	No Errors
Target Device:	xc3s500e-5fg320	• Warnings:	4 Warnings
Product Version:	SE 10.1.03 - Foundation Simulator	• Routing Results:	All Signals Completely Routed
Design Goal:	Balanced	• Timing Constraints:	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	0 (Timing Report)

Performance Summary			
Final Timing Score:	0	Pinout Data:	Pinout Report
Routing Results:	All Signals Completely Routed	Clock Data:	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					
Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Sun Jul 12 15:03:12 2009	0	4 Warnings	0
Translation Report	Current	Sun Jul 12 15:07:34 2009	0	0	0
Map Report	Current	Sun Jul 12 15:07:49 2009	0	0	2 Infos
Place and Route Report	Current	Sun Jul 12 15:08:49 2009	0	0	2 Infos
Static Timing Report	Current	Sun Jul 12 15:09:00 2009			
Bitgen Report					

Fig 5.5 Design Summary Report of RS Decoder

CHAPTER-6
CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

In this thesis, error detection and correction techniques have been used which are essential for reliable communication over a noisy channel. The effect of errors occurring during transmission is reduced by adding redundancy to the data prior to transmission. The redundancy is used to enable a decoder in the receiver to detect and correct errors. Cyclic linear block codes are used efficiently for error detection and correction. The encoder splits the incoming data stream into blocks and processes each block individually by adding redundancy in accordance with a prescribed algorithm. Likewise, the decoder processes each block individually and it corrects errors by exploiting the redundancy present in the received data. An important advantage of cyclic codes is that they are easy to encode. Also they possess a well defined mathematical structure which has lead to very efficient decoding schemes for them.

Galois finite fields are used for the processing of linear block codes. In each galois field there exists a finite element α, and all other nonzero elements of the field are represented as powers of this primitive element α. The symbols used in the block codes are the elements of a finite galois field. Due to modulo arithmetic followed in the finite fields, the resultant of any algebraic operation is mapped into the field and hence rounding issues are conveniently solved. So the addition, subtraction, multiplication or division of two code words is again a valid codeword inside the field.

Reed solomon codes are one of the most powerful and efficient non binary error correcting codes for detecting and correcting burst errors. An irreducible generator polynomial is used for generating the encoded data called codeword. All encoded data symbols are elements of the galois field defined by the parameters of the application and the properties of the system. The encoder is implemented using linear shift registers with feedback. The decoder checks the received data for any errors by calculating the syndrome of the codeword. If an error is detected, the process of correction begins by locating the errors first. Generally euclid's algorithm is used to calculate the error – locator polynomial it is very easy to implement, while its counterpart berlekamp massey algorithm is more hardware efficient. The precise location of the errors is calculated by using chien search algorithm. Then magnitude of the errors is calculated using forney's

algorithm. The magnitude of the error is added to the received codeword to obtain a correct codeword. Hence the using the reed solomon codes, burst errors can be effectively corrected. Reed solomon codes are efficiently used for compact discs to correct the bursts which might occur due to scratches or fingerprints on the discs.

6.2 Future Scope

The CD player is just one of the many commercial, mass applications of the reed solomon codes. The commercial world is becoming increasingly mobile, while simultaneously demanding reliable, rapid access to sales, marketing, and accounting information. Unfortunately the mobile channel is a problematic environment with deep fades an ever present phenomenon. Reed solomon codes are the best solution to this problem. There is no other error control system that can match their reliability performance in the mobile environment.

The optical channel provides another set of problems altogether. Shot noise and a dispersive, noisy medium plague line-of-sight optical system, creating noise bursts that are best handled by reed solomon codes. As optical fibers see increased use in high-speed multiprocessors, reed solomon codes can be used there as well.

There features make reed solomon codes an apparent choice for deep space probes. They have been efficiently used in the image communication system of NASA's Voyager mission. Reed solomon codes will continue to be used to force communication system performance ever closer to the line drawn by Shannon.

References

1. IEEE Standard 802.16-2004, "Part 16: Air interface for fixed broadband wireless access systems," October 2004. http://ieee802.org/16/published.html.
2. Carl Eklund, et.al., "WirelessMAN: Inside the IEEE 802.16 Standard for Wireless Metropolitan Area Networks," IEEE Press, 2006.
3. Hagenauer, J., and Lutz, E., “Forward Eror Correction Coding for Fading Compensation in Mobile Satellite Channels,” IEEE JSAC, vol. SAC -5, no. 2, Feb 1987, pp. 215 -225.
4. Ling-Pei Kung, “Introduction To Error Correcting Codes”, NSDL Scout Report for Math, Engineering, and Technology, 2003.
5. Wicker, S. B. and Bhargava, V. K., ed., Reed-Solomon Codes and Their Applications (Piscataway, NJ: IEEE Press, 1983).
6. Raghav Bhaskar, Pradeep K. Dubey, Vijay Kumar, Atri Rudra, “Efficient Galois Field Arithmetic On SIMD Architectures”, ACM Symp. On ParallelAlgorithms and Architectures, pp 256-257, 2003.
7. H.M. Shao, T.K. Truong, L.J. Deutsch, J. Yuen and LS. Reed, "A VLSI Design of a Pipeline Reed-Solomon Decoder," IEEE Trans. Comput., vol. C-34, no. 5, pp. 393-403, May 1985.
8. High-speed VLSI Architecture for Parallel Reed-Solomon Decoder", IEEE Trans. on VLSI, April 2003
9. Favalli M., Metra C., “Optimization Of Error Detecting Codes For The Detection Of Crosstalk Originated Errors”, Design Automation and Test in Europe, pp 290-296, March 2001.
10. E. D. Mastrovito. “VLSI Architectures for Computations in Galois Fields”. Link¨oping University, Dept. Electr. Eng., Link¨oping, Sweden, 1991.
11. J. G. Andrews, A. Ghosh and R. Muhamed, "Fundamentals of WiMAX: Understanding Broadband Wireless Networking," Prentice Hall, 2007.
12. Syed Shahzad Shah, Saqib Yaqub, and Faisal Suleman, Self-correcting codes conquer noise Part 2: Reed-Solomon codecs, Chameleon Logics, (Part 1: Viterbi Codecs), 2001.

13. Berlekamp, E. R., Peile, R. E., and Pope, S. P., “The Application of Error Control to Communications,” IEEE Communications Magazine, vol. 25, no. 4, April 1987, pp. 44-57.
14. S. B. Wicker and V. K. Bhargava, “Reed-Solomon Codes And Their Applications”, New Tork, IEEE Press, 1994.
15. Sklar, B., Digital Communications: Fundamentals and Applications, Second Edition (Upper Saddle River, NJ: Prentice-Hall, 2001).
16. Reed, I. S. and Solomon, G., “Polynomial Codes Over Certain Finite Fields,” SIAM Journal of Applied Math., vol. 8, 1960, pp. 300-304.
17. Gallager, R. G., Information Theory and Reliable Communication (New York: John Wiley and Sons, 1968).
18. Odenwalder, J. P., Error Control Coding Handbook, Linkabit Corporation, San Diego, CA, July 15, 1976.
19. Blahut, R. E., Theory and Practice of Error Control Codes (Reading, MA: Addison-Wesley, 1983).
20. C.E. Shannon, “A Mathematical Theory Of Communication ”, Bell System Technology Journal, volume 27, pp. 379-423, 623-656, 1948.S.
21. E. Prange, "Cyclic Error-Correcting Codes in Two Symbols," Air Force Cambridge Research Center-TN-57-103, Cambridge, Mass., September 1957.
22. E. Prange, "Some Cyclic Error-Correcting Codes with Simple Decoding Algorithms," Air Force Cambridge Research Center-TN-58-156, Cambridge, Mass., April 1958.
23. E. Prange, "The Use of Coset Equivalence in the Analysis and Decoding of Group Codes," Air Force Cambridge Research Center-TR-59-164, Cambridge, Mass., 1959.
24. R. C. Bose and D. K. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," Information and Control, Volume 3, pp. 68-79, March 1960.
25. R. C . Bose and D. K. Ray-Chaudhuri, "Further Results on Error Correcting Binary Group Codes," Information and Control, Volume 3, pp. 279-290, September 1960.
26. A. Hocquenghem, "Codes correcteurs d'erreurs," Chiffres, Volume 2, pp. 147-156, 1959
27. I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," SI AM Journal of Applied Mathematics, Volume 8, pp. 300-304, 1960.

28. R. J. McEliece, Finite Fields for Computer Scientists and Engineers, Boston: Kluwer Academic, 1987
29. S. B. Wicker, Error Control Systems for Digital Communication and Storage, N.J.: Prentice-Hall, 1994.
30. R. C. Singleton, "Maximum Distance Q-nary Codes," IEEE Transactions on Information Theory, Volume IT-10, pp. 116-118,1964.
31. D. Gorenstein and N. Zierler, "A Class of Error Correcting Codes in pm Symbols," Journal of the Society of Industrial and Applied Mathematics, Volume 9, pp. 207-214, June 1961.
32. W. W. Peterson, "Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes," IRE Transactions on Information Theory, Volume IT-6, pp. 459-470, September 1960.
33. R. T. Chien, "Cyclic Decoding Procedure for the Bose- Chaudhuri- Hocquenghem Codes," IEEE Transactions on Information Theory, Volume IT-10, pp. 357-363, October 1964.
34. G. D. Forney, "On Decoding BCH Codes " IEEE Transactions on Information Theory, Volume IT-11, pp. 549-557, October 1965.
35. E. R. Berlekamp, "Nonbinary BCH Decoding," paper presented at the 1967 International Symposium on Information Theory, San Remo, Italy.
36. E. R. Berlekamp, Algebraic Coding Theory, New York: McGraw-Hill, 1968.
37. J. L. Massey, "Shift Register Synthesis and BCH Decoding," IEEE Transactions on Information Theory, Volume IT-15, Number 1, pp. 122- 127, January 1969.
38. Y. Sugiyama, Y. Kasahara, S. Hirasawa, and T. Namekawa, "A Method for Solving Key Equation for Goppa Codes," Information and Control, Volume 27, pp. 87-99, 1975.
39. Tommy Oberg, “Modulation, Detection and Coding”, John Wiley and Sons, 2001.
40. N K Jha, “Separable Codes For Detecting Unidirectional Errors”, IEEE Trans. On Computer-Aided Design, v8, 571 - 574, 1989.
41. J E Smith, “On Separable Unordered Codes”, IEEE Trans. Computers, C33, 741 - 743, 1984.
42. H Dong, “Modified Berger Codes For Detection of Unidirectional Errors”, IEEE Trans. Comput, C33, 572 - 575, 1984.
43. Nicolaidis, “Design for Soft-Error Robustness to Rescue Deep Submicron Scaling”, White Paper, iRoC Technologies, 2000.

44. Lin Shu, “An Introduction To Error-Correcting Codes”, Prentice Hall, Inc., 1970.
45. B Bose, “Burst Unidirectional Error Detecting Codes”, IEEE Trans. Comput, C35, 350 -353, 1989.
46. T.K. Truong, L.J. Deutsch, I.S. Reed, I.S. Hsu, K. Wang, and CS. Yeh, 'sThe VLSI Design of a Reed-SoIomon Encoder Using Berlekamp's Bit-Senal Multiplier Algorithm," Third Caltech Conf. on VLSI, pp. 303-329, 1983.
47. I.S. Hsu, I.S. Reed, T.K. Truong, K. Wang, CS. Yeh, and L.J. Deutsch, "The VLSI Implementation of a Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm," IEEE Trans. Comput., vol. (3-33, no. 10, pp. 906-911, Oct. 1984.
48. C.C. Wang, T.K Truong, H.M. Shao, L.J. Deutsch, J.K. Omura, and I.S. Reed,"VLSI Architectures for Computing Multiplications and Inverses in GF(2m)' IEEE Trans. Comput., vol. G34, no. 8, pp. 709-716, Aug. 1985.
49. James S.Plank, A Tutorial on Reed-Solomon Coding for Fault- Tolerance in RAID- like Systems, Technical Report CS-96-332.
50. M .A. Hasan and V. K. Bhargava, "Division and Bit-Serial Multiplication over GF(2m)," IEE Proc., part E, vol. 139, no. 3, pp. 230-236, May 1992.
51. G. Orlando and C. Paar, “A Super-Serial Galois Fields Multiplier For FPGAs And Its Application To Public-Key Algorithms”, Proc. of the 7th Annual IEEE Symposium on Field Programmable Computing Machines, FCCM’99, Napa Valley, California, pp. 232-239, April. 1999.
52. Joel Sylvester, “Reed Solomon Codes”, Elektrobit , January 2001.
53. R.E Blahut, “Theory and Practice of Error Control Codes”, Reading, Mass: Addison Wesley, 1984.
54. Lin, S. and Costello, D. J., “Error Control Coding: Fundamentals and Applications,” Prentice-Hall, NJ, 1983.
55. Wicker, S. B., “Error Control Coding for Digital Communication and Storage,” Prentice Hall, NJ, 1995.
56. Wicker, S. B. and Bhargava, V. K., “Reed-Solomon Codes and Their Applications,” IEEE Press, NY, 1994.
57. S. M. Alamouti, "A simple transmit diversity technique for wireless communications," IEEE Journal on Selected Areas in Communications, Vol. 16, No. 8, Oct. 1998, pp. 1451-1458.