ECCS 4141 - Information Theory
Final Project Report
(7,5) Reed-Solomon Code

Joao Lucas Magalini Zago
Rodrigo Azevedo Santos

04/04/2014

CONTENTS

CHAPTER 1

INTRODUCTION

A Reed-Solomon (RS) Code is a cyclic error correction and detection code that uses symbol patterns instead of the standard bit patterns in order to transmit and receive messages. Each symbol is constituted of m different bits and the correlation between bits and symbols is done using the theory of Finite Fields.
The first part of this project will be designing a (7,5) RS code in order to obtain

- A generating polynomial for this specific code.
- The encoder and decoder hardware designs.
- The codeword for a given random sample message using the designed encoder.
- The corrupted codeword syndrome and its comparison to the original error syndrome.
- All the possible syndromes considering a 1 symbol error.

The second part of the project focus on the encoder hardware design. For this part the project will present a possible hardware and software implementation for a (7,5) Reed-Solomon Enconder while explaning in details how the components that are built tend to communicate with each other to make the system as a whole.
The third part of this project have a similar objective but with focus on the (7,5) Reed-Solomon Syndrome Calculator.
The forth part of the project is to design the system as a whole with all its components. A non-perfect channel will be designed to simulate possible errors during transmission and the communinication between the components is again explained in details.
This project also consists of several simulations of the encoder, syndrome calculator and the system with a special focus on message streamming and how does synchronization issues affect the system if certain conditions are not satisfied.

All the simulations and components used in this project are done using the Verilog Hardware Description Language (VHDL) and the simulations are done using the iSim tool of XiliX's ISE. The final VHDL code is presented by the end of this report.

CHAPTER 2

THEORY

2.1 Finite Fields

The Finite Fields or Galois Fields (GF) exists with m elements for any p a prime number. Given that a binary number is represented two possible different ways (either a logical '0' or '1') and also that $p = 2$ is a prime number then it is possible to find a finite field for a set a binary numbers.

If we let a symbol α and its possible multiples represent an infinite field then we can assume the following equation is true for the first elements to be those which are represented in binary

$$F = \{0, \alpha^0, \alpha, \alpha^2, ..., \alpha^j, ...\} \tag{2.1}$$

Since a Galois Field has a finite number of elements we need to assume certain restrictions. In order to generate any number out of a finite number of elements the Modulo technique needs to be used. Therefore the for the binary Galois Field to exist the following equation needs to be satisfied

$$\alpha^{2^m-1} = 1 = \alpha^0 \tag{2.2}$$

That is, after m symbols the symbols will start to repeat. Hence if m is the number of elements in the Galois Field then any number not contained in the field will have an equivalent number which is certainly contained in the Galois Field. The processing of mapping the set of numbers not contained in the Galois Field is done through a primitive polynomial which is a irreducible polynomial that has its smallest positive integer n that divides $X^n + 1$ to be $n = 2^m - 1$.

Therefore, if we consider $p = 2$ then the equivalent Finite Field will be given by

$$GF(m) = \{0, \alpha^0, \alpha^1, \alpha^2, ..., \alpha^{2^m-1}\} \tag{2.3}$$

If a symbol represents three different bits then the number of possible symbols contained in the Finite Field is given by $m = 8$. In order to map the bits in the Galois Field a primitive polynomial is required. It can be shown through long division process that $f(X) = X^3 + X + 1$ is a primitive polynomial. If we let α be a root of $f(X)$ then

$$f(\alpha) = \alpha^3 + \alpha + 1 = 0 \tag{2.4}$$

$$\alpha^3 = \alpha + 1$$

$$\alpha^4 = \alpha^2 + \alpha$$

$$\alpha^5 = \alpha^3 + \alpha^2 = \alpha^3 + \alpha + 1$$

$$\alpha^6 = \alpha^3 + \alpha^2 + \alpha = \alpha + 1 + \alpha^2 + \alpha = \alpha^2 + 1$$

Where the first three non-zero elements are mapped through the Identity Matrix and the addition signs are Exclusive OR (XOR) operations bit per bit.

Finally the mapped bits are shown with their respective symbols in table 2.1.

Galois	Binary		
Symbol	X^0	X^1	X^2
0	0	0	0
α^0	1	0	0
α^1	0	1	0
α^2	0	0	1
α^3	1	1	0
α^4	0	1	1
α^5	1	1	1
α^6	1	0	1

Table 2.1: Galois Field to Binary Mapping.

2.2 Reed-Solomon Codes

A Reed-Solomon Code is a cyclic code that uses a Galois Field to encode and decode symbol messages. In general a (n,k) Reed-Solomon Code is constituted of k message symbols and $n - k$ parity symbols. A (n,k) RS Code can be also be represented as $(2^m - 1, 2^m - 1 - 2t)$ where m is the number of bits per symbol and t is the code symbol correction capability. The probability of a symbol error is given by

$$P_E \approx \frac{1}{n} \sum_{j=t+1}^{n} j \binom{n}{j} p^j (1-p)^{n-j} \tag{2.5}$$

where p is the probability of error of the channel symbol and $n = 2^m - 1$

In order to encode a message a generating polynomial $g(X)$ is required and may be calculated using the power series expansion such that

$$g(X) = \sum_{j=0}^{2t} g_j X^j \tag{2.6}$$

where the coefficients $g_0, g_1, ..., g_{2t-1}$ are related to the roots of the polynomials and therefore to the channel symbols and $g_{2t} = 1$. Those coefficients can be obtained by expanding the following form of the generating polynomial in terms of its roots α^n

$$g(X) = \prod_{j=1}^{2t} (X + \alpha^j) \tag{2.7}$$

To obtain the codeword for the message symbols a encoder schematics for the Reed-Solomon code is shown in figure 2.1

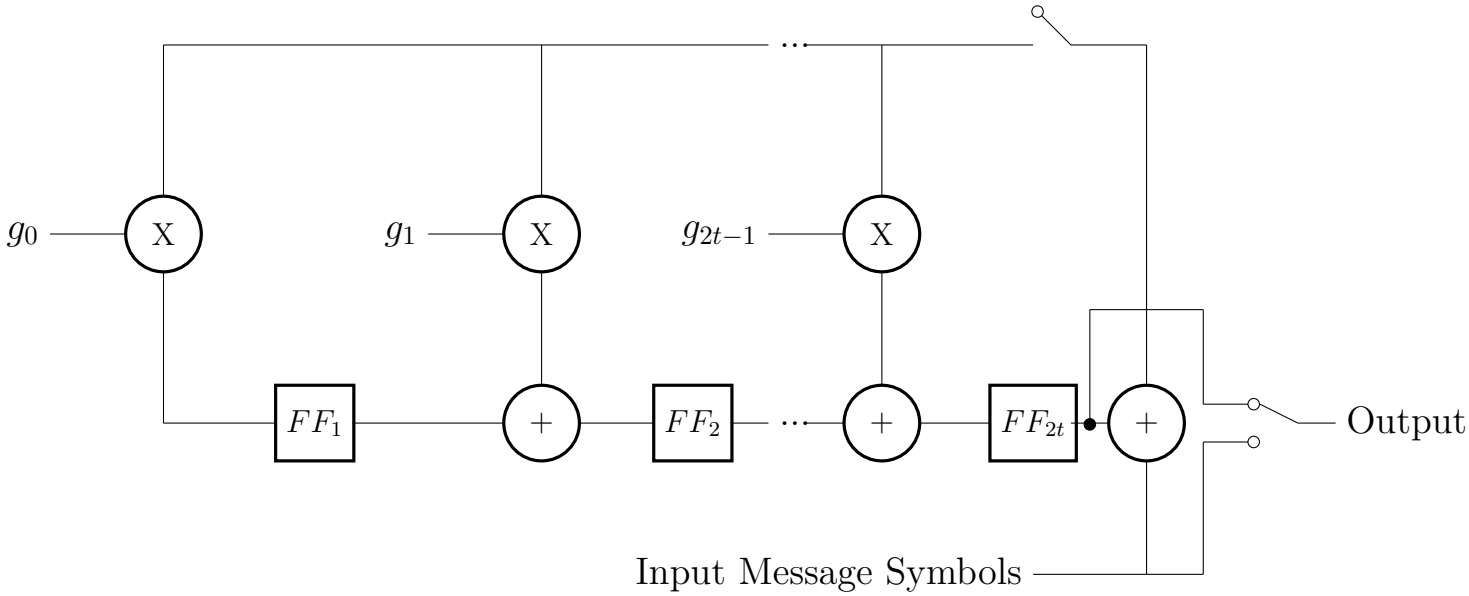

Figure 2.1: RS Code Encoder Schematics

Assuming all Flip-Flops start with a zero value then the message bits will enter the encoder as the upper switch is closed and the second switch is selecting the message bits. The system will then run k times until there are no more message bits remaining and the values of the Flip-Flops will return the parity bits for the message and the final codeword.

The decoder follow a similar strategy but this time the codeword enters from the left side as shows the schematics in figure 2.2

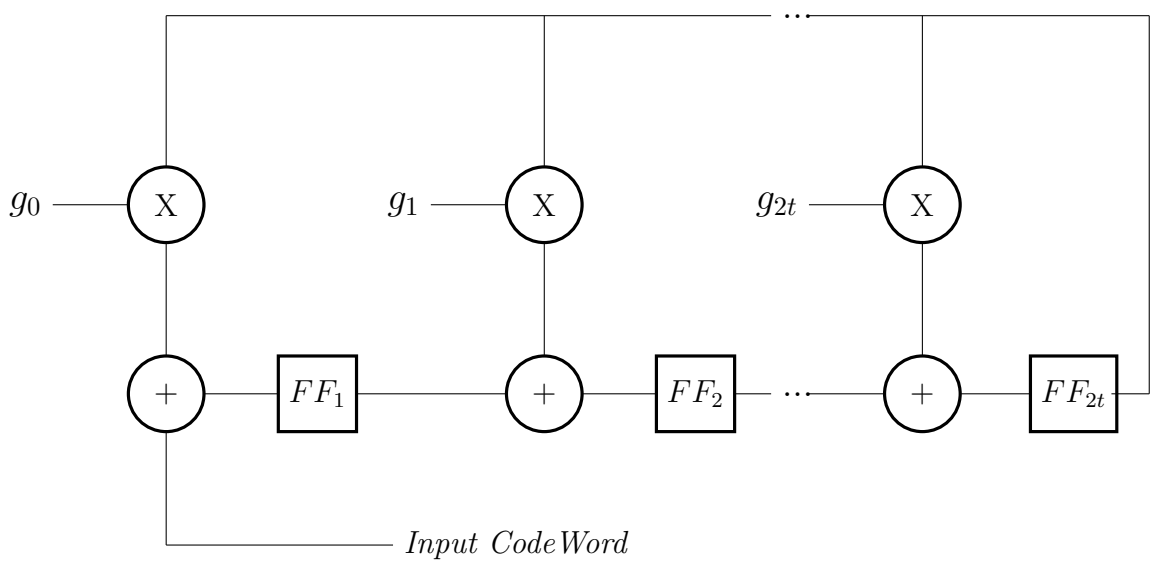

Figure 2.2: RS Code Decoder Schematics

Also assuming all Flip-Flops start with a zero value then after running the system n times the remaining values in the Flip-Flops are called a Syndrome. The Syndrome will point to a possible symbol error pattern depending on the value of t. Therefore the message can be reconstructed even after an error is injected into the codeword through the channel.

CHAPTER 3

THE $(7, 5)$ REED-SOLOMON CODE

3.1 Obtaining a generating polynomial.

In order to obtain a generating polynomial for the $(7, 5)$ RS code it is required to obtain some previous information such as the error correction capability t and the number of bits per symbol m. We can rewrite this code in the form $(2^m - 1, 2^m - 1 - 2t)$. Given the transformation done using table 2.1 if we let $m = 3$ will lead in

$$2^3 - 1 - 2t = 5 \tag{3.1}$$

which has $t = 1$ bit as a solution. Therefore using equation 2.7 it can be easily derived that

$$g(X) = (X + \alpha)(X + \alpha^2) \tag{3.2}$$

$$g(X) = X^2 + X(\alpha^2 + \alpha) + \alpha^3 \tag{3.3}$$

is a generating polynomial for this code. It is possible to simplify it given that $\alpha^2 + \alpha = \alpha^4$ and thus

$$g(X) = \alpha^3 + \alpha^4 X + X^2 \tag{3.4}$$

3.2 Encoder Hardware Design

The encoder hardware design process is straight forward by just plugging the values obtained previously into the design shown by figure 2.1. The final encoder hardware for the $(7, 5)$ Reed-Solomon Code is shown in figure 3.1.

Note that since it is a $(7, 5)$ code it will have five message symbols and two parity symbols in a total of a seven symbols codeword. The parity symbols for a given message can be obtained by running the encoder k times.

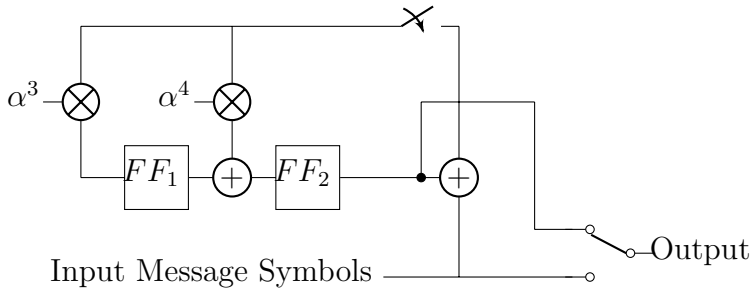

Figure 3.1: (7,5) RS Code 3-bit Symbol Encoder Schematics

3.3 Decoder Hardware Design

The decoder hardware design process is similar to the process shown in figure 3.2. By plugging the values previously obtained it is possible to derive that

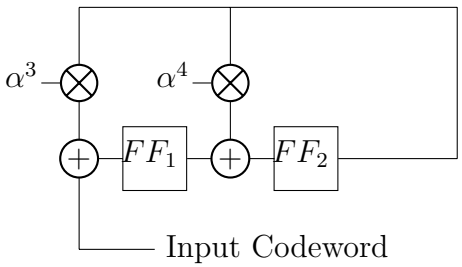

Figure 3.2: (7,5) RS Code 3-bit Symbol Decoder Schematics

3.4 Obtaining a Codeword for a Sample Message

For a given sample message it is desired to obtain its equivalent codeword using the $(7, 5)$ RS code. As a message has a five symbols length and each symbols represent a combination of three different bits then the sample message needs to be fifteen bits long. Let M be a message such that

$$M = 110101001011100 \tag{3.5}$$

if we group each consecutive three bits and map them using the Galois Field then M can be rewritten as

$$M = \alpha^3\alpha^6\alpha^2\alpha^4\alpha^0 \tag{3.6}$$

Hence if M is used at the encoder designed in figure 3.1. The results for each clock cycle assuming all Flip-Flops start with a zero value are shown in table 3.1

Those processes shown in table 3.2 are illustrated in figure 3.3.

Inputs Left	Current Input	Current Clock Cycle	Current Values in the Resgisters
$\alpha^3\alpha^6\alpha^2\alpha^4\alpha^0$	-	0	00
$\alpha^3\alpha^6\alpha^2\alpha^4$	α^0	1	$\alpha^3\alpha^4$
$\alpha^3\alpha^6\alpha^2$	α^4	2	$0\alpha^3$
$\alpha^3\alpha^6$	α^2	3	$\alpha^1\alpha^2$
α^3	α^6	4	$\alpha^3\alpha^2$
-	α^3	5	$\alpha^1\alpha^5$

Table 3.1: Encoding Process Results.

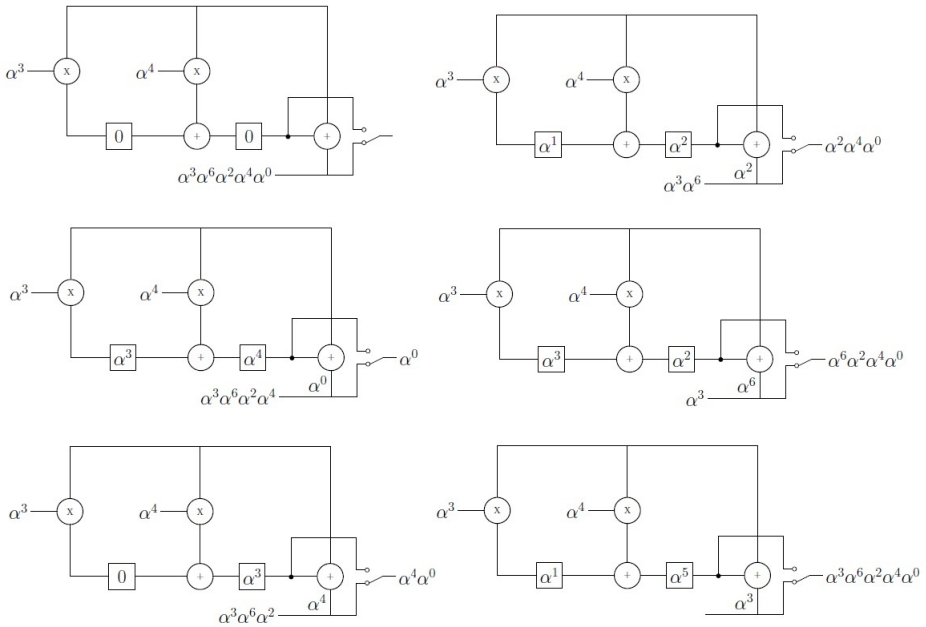

Figure 3.3: Encoding Process

Hence if the values values of the registers are shifted to the output as the switch select them the final codeword for the given message will be given by

$$U_M = \alpha^1\alpha^5\alpha^3\alpha^6\alpha^2\alpha^4\alpha^0 \tag{3.7}$$

where α^1 and α^5 are called parity symbols. In binary the equivalent codeword is given by

$$U_{M_B} = 010111110101001011100 \tag{3.8}$$

In order to prove that the obtained parity bits are in fact correct the following equations need to be satisfied

$$\begin{cases} U(\alpha) = 0 \\ U(\alpha^2) = 0 \end{cases} \tag{3.9}$$

for α be the roots of $U(X)$ where $U(X)$ given by multiplication of the coefficients by the powers of the variable X

$$U(X) = \alpha^1 + \alpha^5 X + \alpha^3 X^2 + \alpha^6 X^3 + \alpha^2 X^4 + \alpha^4 X^5 + \alpha^0 X^6 \tag{3.10}$$

Let us start with $U(\alpha)$

$$\begin{aligned} U(\alpha) &= \alpha^1 + \alpha^5\alpha + \alpha^3\alpha^2 + \alpha^6\alpha^3 + \alpha^2\alpha^4 + \alpha^4\alpha^5 + \alpha^0\alpha^6 \\ &= \alpha^1 + \alpha^6 + \alpha^5 + \alpha^9 + \alpha^6 + \alpha^9 + \alpha^6 \\ &= \alpha^1 + \alpha^6 + \alpha^5 = 0 \end{aligned}$$

Then for $U(\alpha^2)$

$$\begin{aligned} U(\alpha^2) &= \alpha^1 + \alpha^5\alpha^2 + \alpha^3\alpha^4 + \alpha^6\alpha^6 + \alpha^2\alpha^8 + \alpha^4\alpha^{10} + \alpha^0\alpha^{12} \\ &= \alpha^1 + \alpha^7 + \alpha^7 + \alpha^{12} + \alpha^{10} + \alpha^{14} + \alpha^{12} \\ &= \alpha^1 + \alpha^{10} + \alpha^{14} = \alpha^1 + \alpha^3 + \alpha^0 = 0 \end{aligned}$$

Therefore the codeword is correct. In a last verification if we calculate the Syndrome for this specific codeword it should be zero. Using the decoder design presented in figure 3.2 table 3.2 shows the results for the values in the Flip-Flops after each clock cycle while figure 3.4 shows a complete decoding diagram.

Inputs Left	Current Input	Current Clock Cycle	Current Values in the Resgisters
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^2\alpha^4\alpha^0$	-	0	00
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^2\alpha^4$	α^0	1	$\alpha^0 0$
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^2$	α^4	2	$\alpha^4\alpha^0$
$\alpha^1\alpha^5\alpha^3\alpha^6$	α^2	3	$\alpha^5 0$
$\alpha^1\alpha^5\alpha^3$	α^6	4	$\alpha^6\alpha^5$
$\alpha^1\alpha^5$	α^3	5	$\alpha^0\alpha^0$
α^1	α^5	6	$\alpha^2\alpha^5$
-	α^1	7	00

Table 3.2: Decoding Process Results.

As the final syndrome is zero than we can assume that there are no one symbol patterns in error for this codeword and therefore this codeword fits all the requirements.

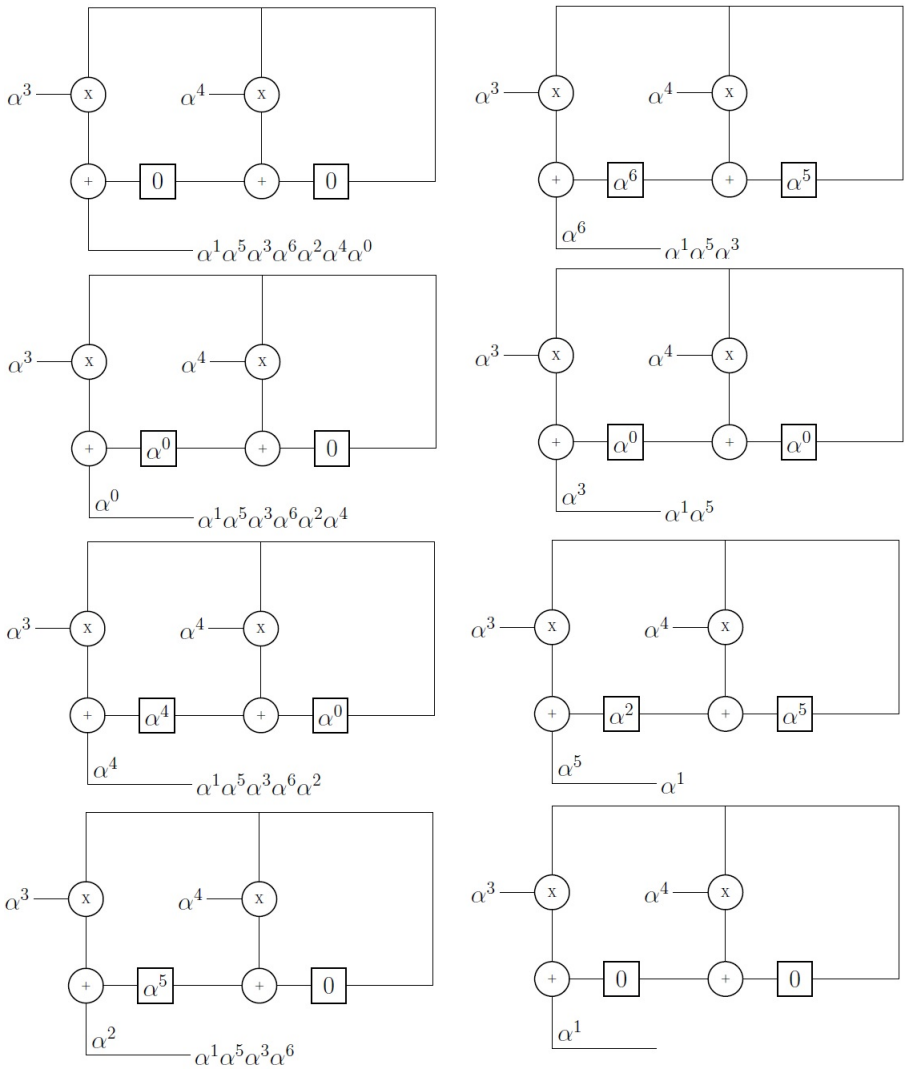

Figure 3.4: Decoding Process

3.5 Single Syndrome Patterns Calculations

Assuming an error can occur between the transmission of the message and that this error can assume any value or then a syndrome table is required in order to correct it. If the error can assume 7 different symbols and replace any of the 7 codeword symbols then there should be 49 different syndrome patterns that will point to the error and correct it. The syndrome calculation for all single symbol errors is shown in table 3.3

Syndromes	$000000X$	$00000X0$	$0000X00$	$000X000$	$00X0000$	$0X00000$	$X000000$
α^0	$\alpha^1\alpha^4$	$\alpha^1\alpha^5$	$\alpha^3\alpha^5$	$\alpha^0\alpha^0$	$\alpha^3\alpha^4$	$0\alpha^0$	α^00
α^1	$\alpha^2\alpha^5$	$\alpha^2\alpha^6$	$\alpha^4\alpha^6$	$\alpha^1\alpha^1$	$\alpha^4\alpha^5$	$0\alpha^1$	α^10
α^2	$\alpha^3\alpha^6$	$\alpha^3\alpha^0$	$\alpha^5\alpha^0$	$\alpha^2\alpha^2$	$\alpha^5\alpha^6$	$0\alpha^2$	α^20
α^3	$\alpha^4\alpha^0$	$\alpha^4\alpha^1$	$\alpha^6\alpha^1$	$\alpha^3\alpha^3$	$\alpha^6\alpha^0$	$0\alpha^3$	α^30
α^4	$\alpha^5\alpha^1$	$\alpha^5\alpha^2$	$\alpha^0\alpha^2$	$\alpha^4\alpha^4$	$\alpha^0\alpha^1$	$0\alpha^4$	α^40
α^5	$\alpha^6\alpha^2$	$\alpha^6\alpha^3$	$\alpha^1\alpha^3$	$\alpha^5\alpha^5$	$\alpha^1\alpha^2$	$0\alpha^5$	α^50
α^6	$\alpha^0\alpha^3$	$\alpha^0\alpha^4$	$\alpha^2\alpha^4$	$\alpha^6\alpha^6$	$\alpha^2\alpha^3$	$0\alpha^6$	α^60

Table 3.3: All Single Symbol Syndromes.

Note from table 3.3 that since the message has only 5 symbols than only the first 5 columns of the syndrome table should be taken into account for the message correcting. That is 35 different syndrome patterns. In fact as only zeros are being inserted after the first error occurs then it is only neccessary to calculate the pattern $000000X$ where X is the first symbol error pattern and can assume any value in the range $[\alpha^0, \alpha^6]$. Therefore tables 3.4 through 3.10 show this syndrome calculations in steps.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^0$	-	0	00
000000	α^0	1	α^00
00000	0	2	$0\alpha^0$
0000	0	3	$\alpha^3\alpha^4$
000	0	4	$\alpha^0\alpha^0$
00	0	5	$\alpha^3\alpha^5$
0	0	6	$\alpha^1\alpha^5$
-	0	7	$\alpha^1\alpha^4$

Table 3.4: $X = \alpha^0$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^1$	-	0	00
000000	α^1	1	$\alpha^1 0$
00000	0	2	$0\alpha^1$
0000	0	3	$\alpha^4\alpha^5$
000	0	4	$\alpha^1\alpha^1$
00	0	5	$\alpha^4\alpha^6$
0	0	6	$\alpha^2\alpha^6$
-	0	7	$\alpha^0\alpha^4$

Table 3.5: $X = \alpha^1$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^2$	-	0	00
000000	α^2	1	$\alpha^2 0$
00000	0	2	$0\alpha^2$
0000	0	3	$\alpha^5\alpha^6$
000	0	4	$\alpha^2\alpha^2$
00	0	5	$\alpha^5\alpha^0$
0	0	6	$\alpha^3\alpha^0$
-	0	7	$\alpha^3\alpha^6$

Table 3.6: $X = \alpha^2$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^3$	-	0	00
000000	α^3	1	$\alpha^3 0$
00000	0	2	$0\alpha^3$
0000	0	3	$\alpha^6\alpha^0$
000	0	4	$\alpha^3\alpha^3$
00	0	5	$\alpha^6\alpha^1$
0	0	6	$\alpha^4\alpha^1$
-	0	7	$\alpha^4\alpha^0$

Table 3.7: $X = \alpha^3$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^4$	-	0	00
000000	α^4	1	$\alpha^4 0$
00000	0	2	$0\alpha^4$
0000	0	3	$\alpha^0\alpha^1$
000	0	4	$\alpha^4\alpha^4$
00	0	5	$\alpha^0\alpha^2$
0	0	6	$\alpha^5\alpha^2$
-	0	7	$\alpha^5\alpha^1$

Table 3.8: $X = \alpha^4$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^5$	-	0	00
000000	α^5	1	$\alpha^5 0$
00000	0	2	$0\alpha^5$
0000	0	3	$\alpha^1\alpha^2$
000	0	4	$\alpha^5\alpha^5$
00	0	5	$\alpha^1\alpha^3$
0	0	6	$\alpha^6\alpha^3$
-	0	7	$\alpha^6\alpha^2$

Table 3.9: $X = \alpha^5$ Syndrome Calculation.

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$000000\alpha^6$	-	0	00
000000	α^6	1	$\alpha^6 0$
00000	0	2	$0\alpha^6$
0000	0	3	$\alpha^2\alpha^3$
000	0	4	$\alpha^6\alpha^6$
00	0	5	$\alpha^2\alpha^4$
0	0	6	$\alpha^0\alpha^4$
-	0	7	$\alpha^0\alpha^3$

Table 3.10: $X = \alpha^6$ Syndrome Calculation.

3.6 Error injection and correction

In order to prove the syndrome tables are correct let us take a random error e. Assume during the transmission of the sample codeword an error occured such that the error pattern is given by

$$e = 0000\alpha^3 00$$

Such that the received vector r by the decoder is the addition bit per bit between the codeword and the error

$$U' = e + U = \alpha^1 + \alpha^5 + \alpha^3 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^0$$

The 5^{th} codeword symbol was altered by the error and changed its value from α^2 to α^5. When trying to decode this codeword the decoder processes are shown in table 3.11

Inputs Left	Current Input	Current Clock Cycle	Current Value in Registers
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^5\alpha^4\alpha^0$	-	0	00
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^5\alpha^4$	α^0	1	$\alpha^0 0$
$\alpha^1\alpha^5\alpha^3\alpha^6\alpha^5$	α^4	2	$\alpha^4\alpha^0$
$\alpha^1\alpha^5\alpha^3\alpha^6$	α^5	3	$\alpha^2 0$
$\alpha^1\alpha^5\alpha^3$	α^6	4	$\alpha^6\alpha^2$
$\alpha^1\alpha^5$	α^3	5	$\alpha^2 0$
α^1	α^5	6	$\alpha^5\alpha^2$
-	α^1	7	$\alpha^6\alpha^1$

Table 3.11: Infected Codeword Syndrome Calculation

Looking at this particular syndrome and referring back to table 3.3 the receiver can easily understand that since the syndrome is not zero an error must have occurred during the transmission process. The receiver then looks for the error pattern for this particular syndrome at table 3.3 and once it finds what is the estimated error it may add it again to the received codeword in order to recover the original message.

$$\hat{e} = 0000\alpha^3 00$$

$$U_{corrected} = U' + \hat{e} = \alpha^1 + \alpha^5 + \alpha^3 + \alpha^6 + \alpha^2 + \alpha^4 + \alpha^0$$

CHAPTER 4

SYSTEM AND COMPONENTS

4.1 System

The Reed-Solomon whole system may be represented as a junction of separate blocks together that will operate synchronously using a universal clock signal. The system starts by receiving a symbol message that is encoded using a (7,5) Reed-Solomon encoder. The encoder have the function of generating parity symbols such that the its output codeword is not affected by single symbol errors while traveling through the channel.
A transmitter is used to transmit the codeword through a channel to a receiver. Assuming a perfect channel with no distortion and noise the codeword and parity bits themselves would be useless, therefore real channel always tend to distort, delay and attenuate the message. The (7,5) RS code will be able to correct any errors caused by the channel if and only if the channel changes at most 1 symbol in the codeword.
A receiver then receives the codeword in error and throws this codeword into a syndrome calculator. Although the system must hold the codeword's symbols in order to correct them in a further action. This temporary holding is done using a FIFO buffer which will delay the codeword until the system is ready to correct it.
The syndrome calculator then calculates the syndrome and a error guessing component compares this syndrome with table 3.3 to get the estimated error.
The system is now ready to output the codeword from the FIFO and add it to the estimated error resulting in the corrected codeword to be recovered. From there the parity symbols may be discarded and the system may proceed using the message.
All these processes are ilustrated in figure 4.1.

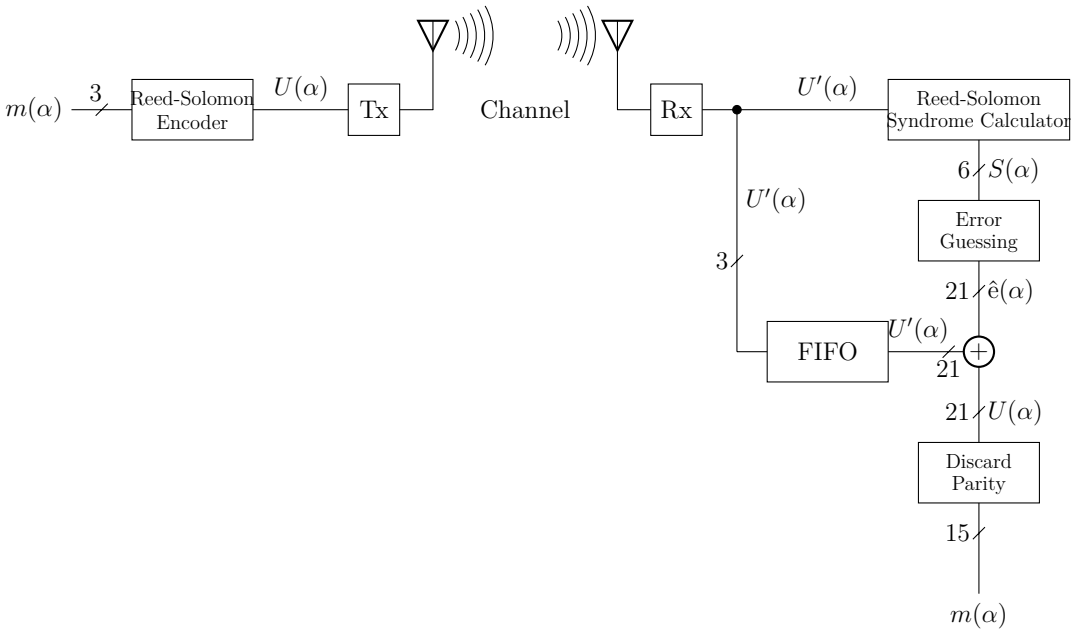

Figure 4.1: Reed-Solomon Channel Coding System.

4.2 Hardware Basic Components

4.2.1 Symbol Hardware Multiplication

The (7,5) Reed-Solomon symbol hardware multiplier schematics is shown in figure 4.2

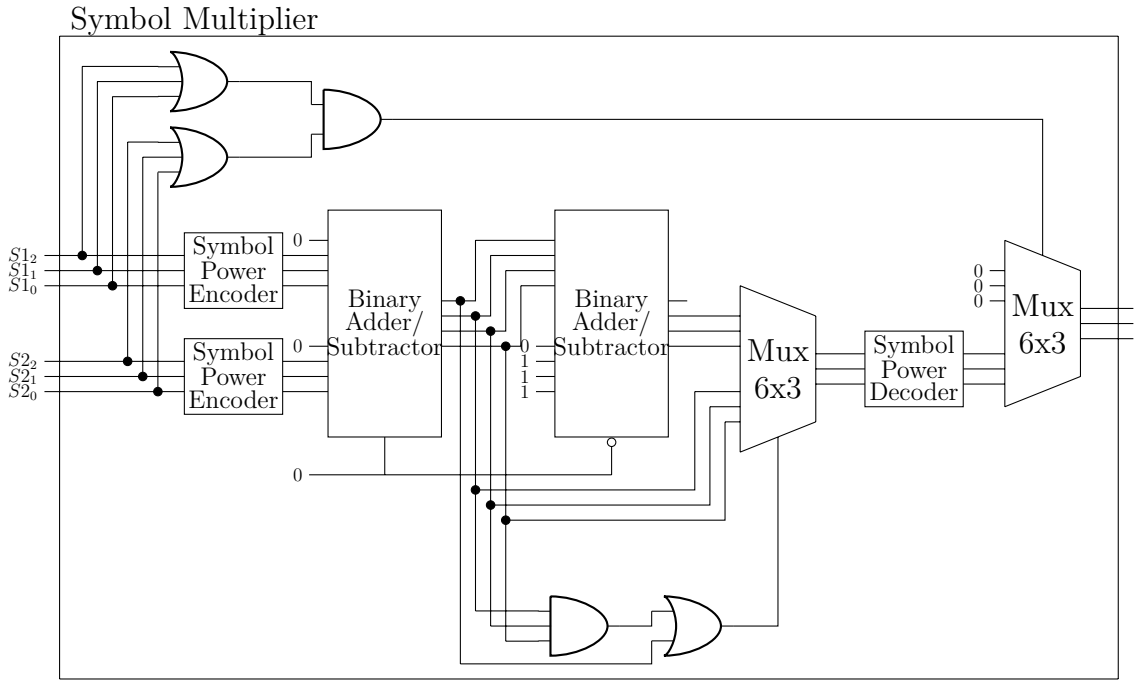

Figure 4.2: Reed-Solomon Symbol Multiplier Hardware

As two symbols are going to be multiplied they enter the system described in figure 4.2. After entering the system each symbol should pass through a Symbol Power Encoder. This encoder will map each symbol power into its equivalent decimal. Note that this mapping process is much smaller then the one required to map all the multiplication possibilities between 3-bit symbols. While this mapping only maps 7 different patterns a full multiplier encoder using a table would have to map 64 different patterns. The Symbol Power Encoder mapping is done through table 4.1.

Input Symbol	Binary Input	Binary Output	Decimal Output
α^0	100	000	0
α^1	010	001	1
α^2	001	010	2
α^3	110	011	3
α^4	011	100	4
α^5	111	101	5
α^6	101	110	6

Table 4.1: Symbol Power Encoder Mapping.

As in algebra in order to multiply both symbols we may add their powers. Therefore the next step is taking the output of each symbol power encoder and add them together using a standard binary adder. As the maximum value of each power is 6 the sum of the symbols would never be greater than 12, which in binary can be represented using 4 bits. Hence the Binary Adder would take 8 inputs and return 4 outputs. As each encoded symbol power has only three bits the most significant bit may be set to zero in both encoded symbols. Now this output may or may not be the output required to be decoded and thrown back into the system. Let α^p and α^q be the symbols that are being multiplied. This output will be decoded if the following condition is true

$$p + q \leq 6$$

If not this means the result of the multiplication will lead into a symbol not originally mapped in the Galois Field. As the multiplication of symbols will never exceed 12 this output is also thrown into a subtractor with 0111 which is 7 in binary. This will always result in a power contained in the Galois Field. The Adder and Subtractor are the exact same block and the operation they should do is controlled and a logic variable.

Therefore a $6x3$ Multiplexer is used to select one of the outputs. Note that it only need to select the three least significant bits since those are the bits that have the information about the symbol power to be decoded.This multiplexer will select the subtractor signal if the result of the adder is either 0111 or if the most significant bit is high, which means the addition was greater than seven. The output of the Multiplexer is then thrown into a Signal Power Decoder, that uses the reverse Encoder logic. The signal power decoder mapping is done using table 4.2.

Input Decimal	Binary Input	Binary Output	Symbol Output
0	000	100	α^0
1	001	010	α^1
2	010	001	α^2
3	011	110	α^3
4	100	011	α^4
5	101	111	α^5
6	110	101	α^6

Table 4.2: Symbol Power Decoder Mapping.

Note that this system up until now has not considered the case for the zero symbol because zero cannot be represented as a power of a number. Although just as in algebra when a symbol is multiplied by zero the output should be zero. Hence a final multiplexer should select either zero or the output of the symbol power decoder for the output of the system. If either of the input symbols is zero then this multiplexer will select zero as the output but if both of them have at least one bit different than zero then the output will be given by the output of the decoder.

4.2.2 Symbol Hardware Addition

The Addition between symbols is a much easier task to be implemented in hardware. It is only described as an XOR operation bit per bit as shows figure 4.3

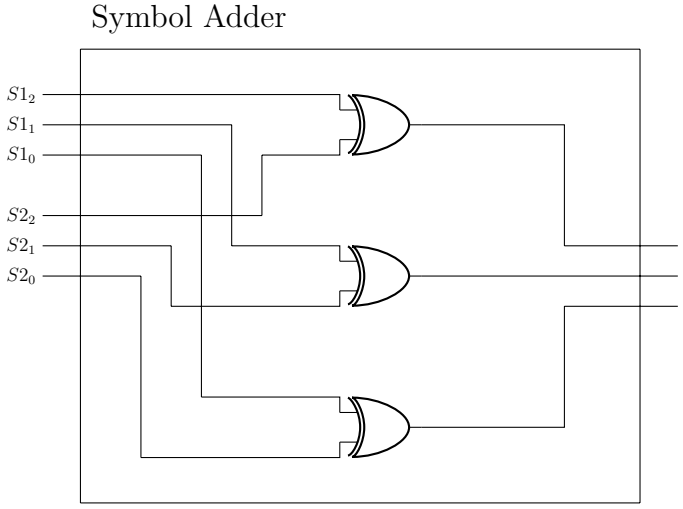

Figure 4.3: Reed-Solomon Symbol Adder Hardware

As the XOR process bit per bit will never result in a carry bit neither will it result in a bit pattern not contained in the Galois Mapping since all the 3-bit patterns are already

mapped this process will always result in a symbol which is surely contained in the Galois Field.

4.2.3 The 3-Bit FlipFlop

In order to store the values of the symbols and insert new symbols into the encoder a 3-bit Flip Flop (FF) is required. This is because a common Flip Flop is a memory component that can only hold 1 bit of information. For all purposes the D Flip Flop was chosen since it provides an easier logic to follow plus a set and reset components. The 3-bit Flip Flop schematics is shown in figure 4.4.

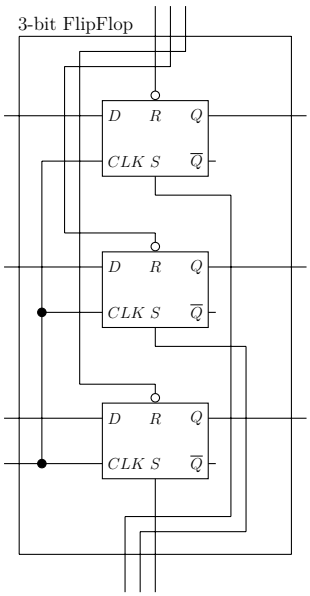

Figure 4.4: 3-Bit FlipFlop Schematics

This FlipFlop is a synchronous device and have all the Clock signals as a common source. As a symbol is stored in the FF its bits are stored in individual FlipFlops connected in parallel. This FlipFlop also have a Set and Reset logic in order to preset its value at the beginning of the encoding process. The output Q of each D FlipFlop constitute the final output of the 3-bit FlipFlop.

4.3 Encoder Hardware

Following the hardware design for the Reed-Solomon Codes it can be shown that for this specific code the Symbolic Hardware Design is given by figure 3.1

Although this encoder is easily described in a symbolic form the actual encoder for the (7,5) RS code is presented in figure 4.5.

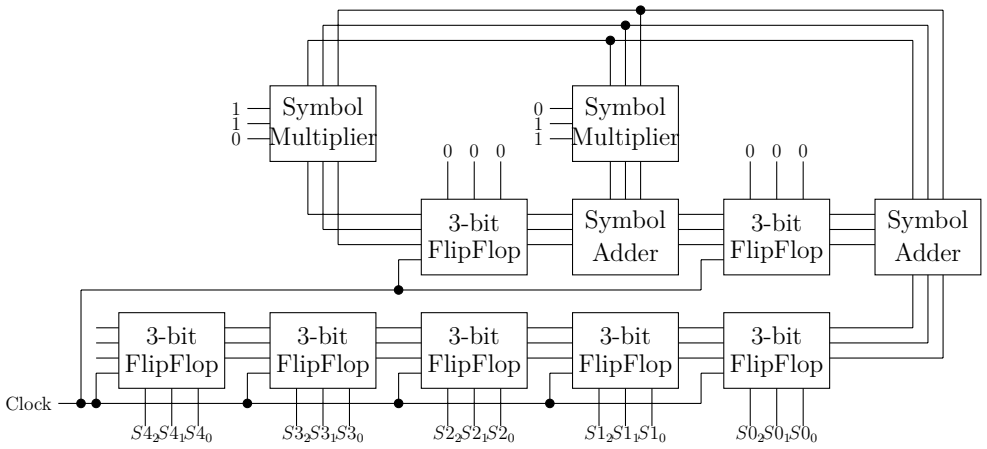

Figure 4.5: Encoder Hardware Schematics

Using the Set and Reset logic it is possible to set the 5 lower FlipFlops with the message symbols and Reset the upper 2 FlipFlops to the zero state as they may have any unsigned start value. Only after setting all the FFs properly the system can start running. If the system is not set properly or the upper 2 FFs are not reset to the zero value errors may occur during the encoding process. As the message is constituted of 5 different symbols there should be 5 clock cycles in order to encode this message. As the clock rises the lower 5 FFs will output its actual value into the next FF until all the message is inserted into the system.

As the symbols have three degrees of freedom each a simple switch is also not trivial to be implemented into the system. In order to send the message symbols followed by the calculated parity symbols the following system design was used for the encoder block.

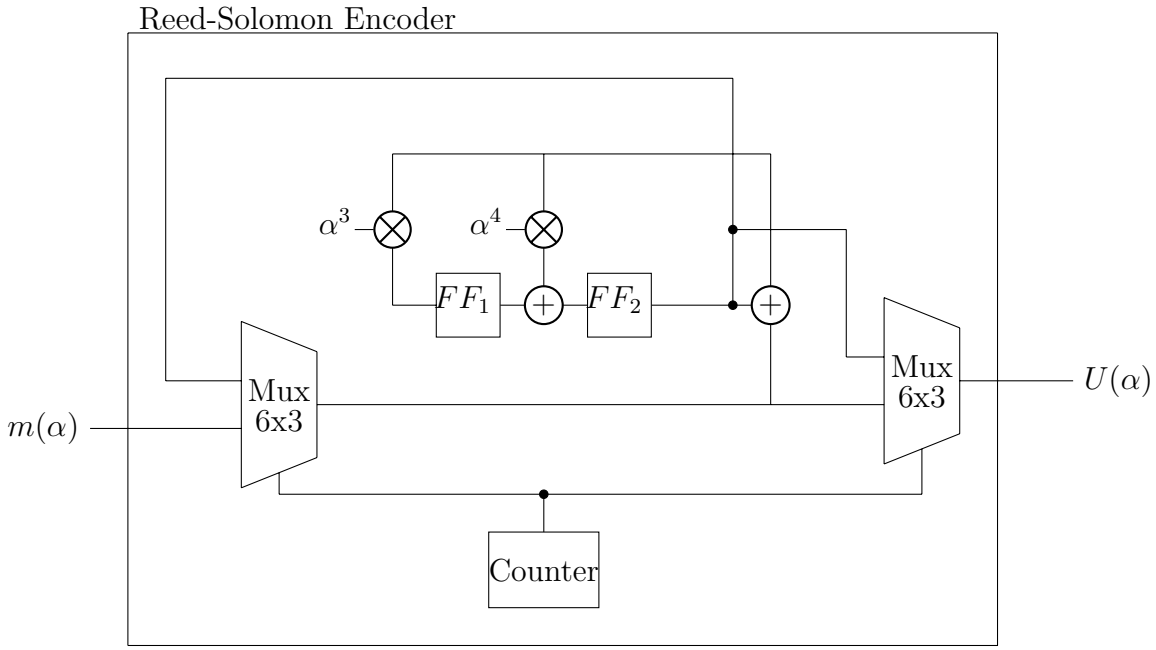

Figure 4.6: Encoder Component.

Note from figure 4.6 that a counter and two multiplexers were used in order to get send the message and the parity symbols in serial. The counter used in the Encoder Component has the following implemented logic table 4.3.

i	Output
0	Low
1	Low
2	Low
3	Low
4	Low
5	High
6	High
7	Low

Table 4.3: Counter Output

As the counter has a low value the message symbols will flow through the system and the output for the first 5 clock cycles, which is the number of clock cycles required to calculate the parity symbols. After that the counter output turns into a high logic value switching the outputs of the multiplexers. Now the parity symbols should start entenring the encoder and leave the system in a serial form at the same time. As a parity symbol reenters the encoder the first operation is an addition between two equal symbols resulting into a 0. This 0 should shift the second parity symbol contained in the first flop foward and eventually output it through the multiplexer as this is applied twice into the system.

4.4 Syndrome Calculator Hardware

Although this decoder is easily described in a symbolic form the actual decoder for the (7,5) RS code is presented in figure 4.7

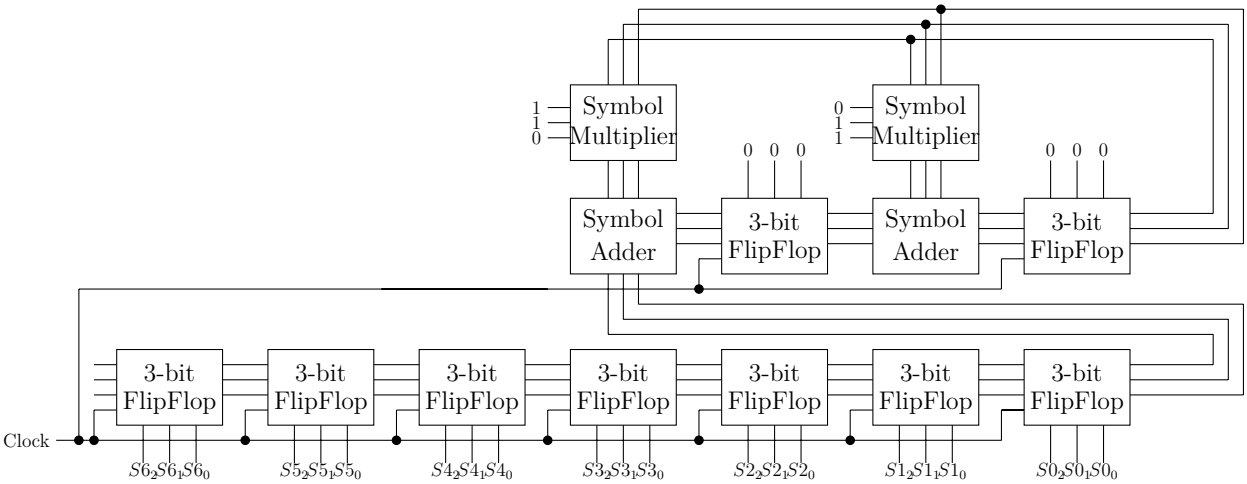

Figure 4.7: Real Syndrome Calculator Hardware Schematics

The same setting and resetting logic used on the Flip Flops by the Encoder is maintained at the Syndrome Calculator. The schematics from the syndrome calculator block are illustrated by figure 4.8.

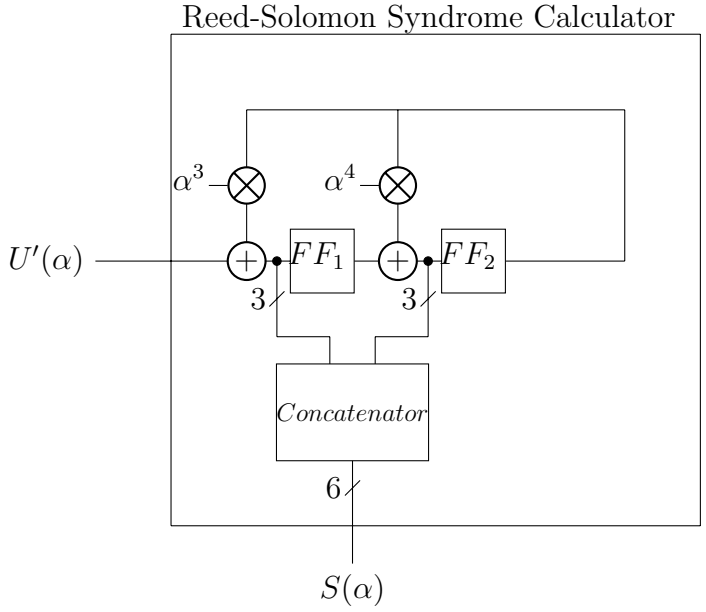

Figure 4.8: Syndrome Calculator Component.

As the codeword enters and the syndrome calculator it takes 7 clock cycles to fully calculate the syndromes. Note that the syndromes are constantly being output by the syndrome calculator since only the symbols calculated after the 7^{th} clock cycle will be taken into account by the system. This is because the system holds the codeword in error in a FIFO that servers as a buffer. The final two syndrome symbols are then output to a concatenator which concatenates both of them in order to check the syndrome table in the error guessing block.

4.5 FIFO Buffer

The FIFO (First In First Out) buffer servers its only purpose to hold the codeword in error until the syndrome is calculated and the estimated error is ready to be added into it. Therefore it is a set of 7 Flip Flops connected in series and this guarantees that once the first symbol enters the FIFO it will only leave it after 7 clock cycles. The FIFO component is shown at figure 4.9. For the purposes of this project the FIFO system also concatenates all the symbols together as the estimated error pattern will have a length of 7 symbols.

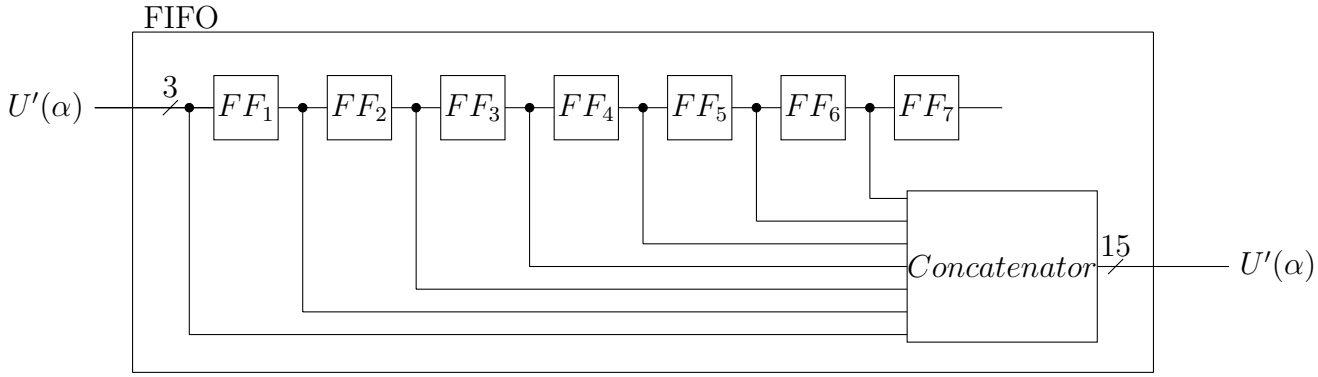

Figure 4.9: FIFO Buffer Component.

4.6 Channel

For the purposes of this project the channel modeling was done using a pseudo random generator that will generate really long sequences of random numbers until it starts repeating. This was done using a Linear Feedback Shift Register (LFSR) which is a cascade of Flip Flops with feedback addition of random outputs that is able to generate a random output in each clock cycle. This block however needs a start value for the Flip Flops and this value was randomly generated using PHP (a high level programming language).
Even though the LFSR is able to generate random symbols it should still be able to place them at random locations in the codeword just as a real channel would. The random placement is done using a multiplexer with its selection being connected to another random linear combination of the LFSR. This linear combination is connected into a Flip Flop

and can be either a high or low logic value. This Flip Flop controls the selection channel and will not add any of the generated error symbols until it changes its logic value. When it does change its value then the symbol is added to its corresponding codeword symbol and this set a signal that does not allow any more additions until the codeword has fully passed through the channel. Once 7 clock cycles have passed this signal is set again to its original value and new errors may be added into the system.
This guarantees that the system will only have 1 symbol in error per codeword and there is also a probability of no errors be added to the system. In this case once the received codeword passes through the syndrome calculator the resulting syndrome should be zero. This channel modeling is presented in figure 4.10.

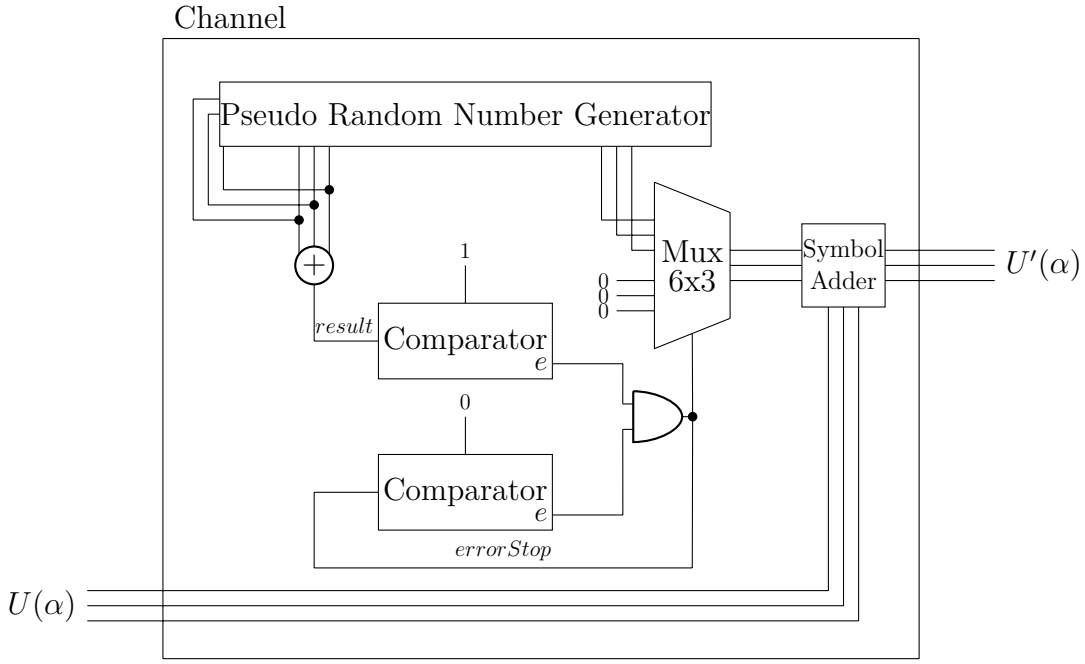

Figure 4.10: Channel Modeling

4.7 Error Guessing

The error guessing block is nothing but a memory block that stores all the possible syndrome combinations with their estimated error patterns. It uses the syndrome as a pointer and returns the estimated error to be added with the received codeword.

CHAPTER 5

SIMULATIONS

5.1 Encoder Simulations

The simulation of the presented Encoder was done using the ISim software, a VHDL simulator which is a part of the XILINX ISE Tools.

The following values were set in the Force Clock.

Leading Edge Value	0
Trailing Edge Value	1
Starting at Time Offset	0.5
Cancel after Time Offset	10
Period	2

Table 5.1: Initial clock settings for encoding simulations.

5.1.1 The first sample message.

The first simulation was for the following symbol and binary message

$$M_{1_{Symbol}} = \alpha^3\alpha^6\alpha^2\alpha^4\alpha^0$$
$$M_{1_{Binary}} = 110101001011100$$

The results for this encoding process is shown in figure 5.1

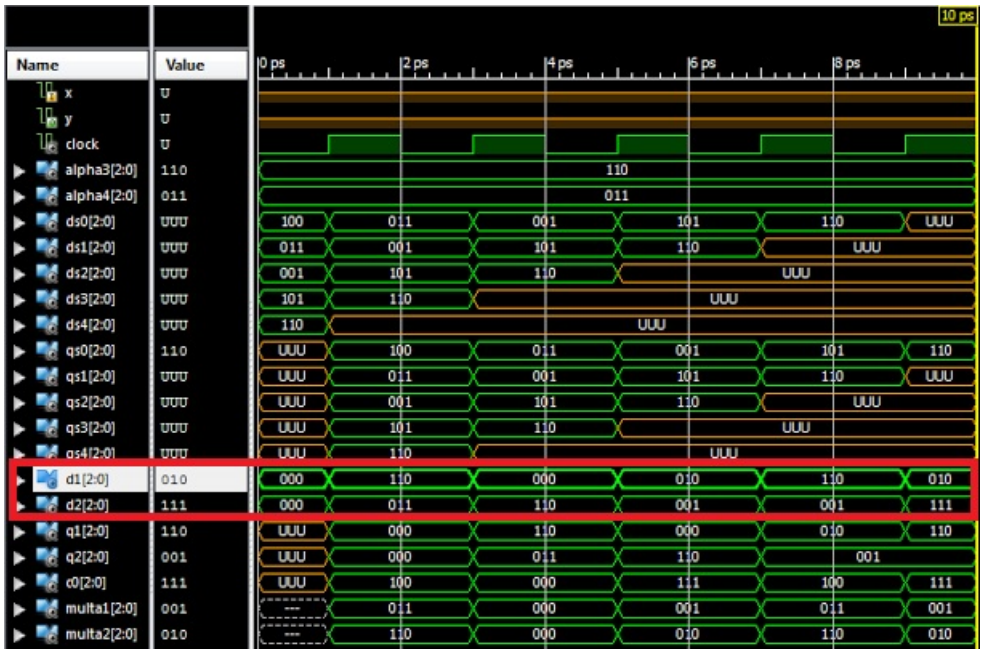

Figure 5.1: Encoding results for the first sample message.

Figure 5.1 shows that for this specific sample message the parity bits and symbols remaining in the FlipFlops are respectively

$$P_{1_{Binary}} = 010111$$
$$P_{1_{Symbol}} = \alpha^1\alpha^5$$

Hence the complete codeword for this sample message will be given by

$$U_{1_{Binary}} = 010111110101001011100$$
$$U_{1_{Symbol}} = \alpha^1\alpha^5\alpha^3\alpha^6\alpha^2\alpha^4\alpha^0$$

5.1.2 The second sample message

The second simulation was for the following symbol and binary message

$$M_{2_{Symbol}} = \alpha^3\alpha^4\alpha^6\alpha^1\alpha^0$$
$$M_{2_{Binary}} = 110011101010100$$

The results for this encoding process is shown in figure 5.2

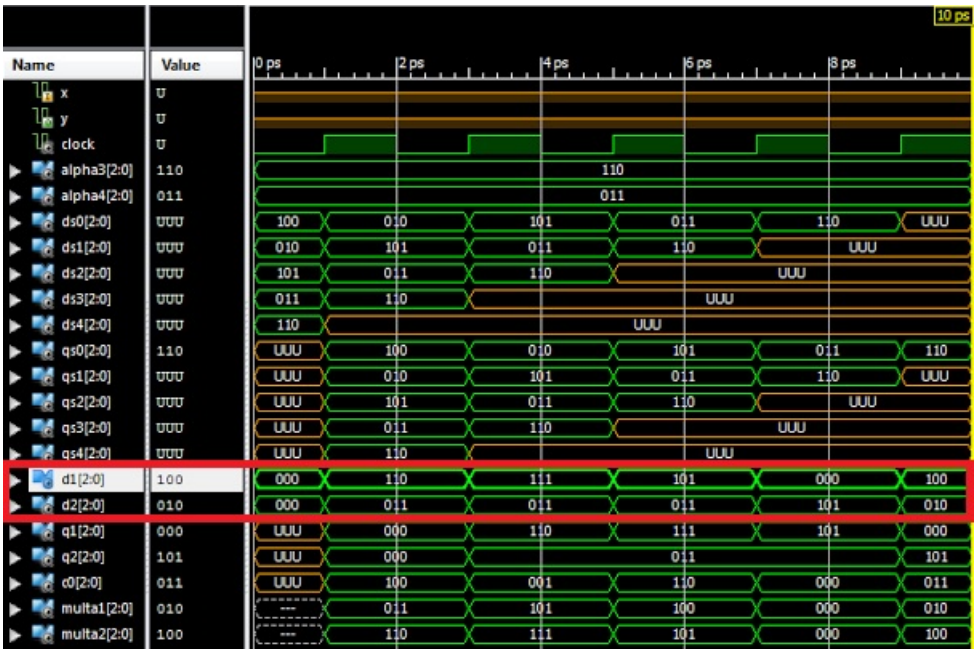

Figure 5.2: Encoding results for the second sample message.

Figure 5.2 shows that for this specific sample message the parity bits and symbols remaining in the FlipFlops are respectively

$$P_{2_{Binary}} = 100010$$
$$P_{2_{Symbol}} = \alpha^0\alpha^1$$

Hence the complete codeword for this sample message will be given by

$$U_{2_{Binary}} = 100010110011101010100$$
$$U_{2_{Symbol}} = \alpha^0\alpha^1\alpha^3\alpha^4\alpha^6\alpha^1\alpha^0$$

5.1.3 The third sample message

The third simulation was for the following symbol and binary message

$$M_{3_{Symbol}} = \alpha^5 0 \alpha^2\alpha^0\alpha^3$$
$$M_{3_{Binary}} = 111000001100110$$

The results for this encoding process is shown in figure 5.3

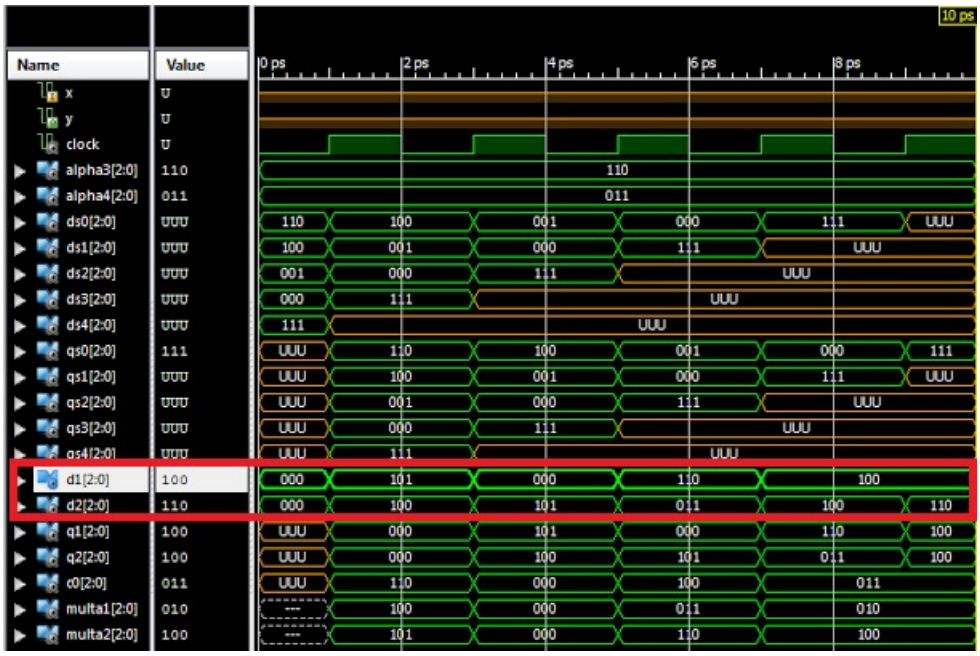

Figure 5.3: Encoding results for the third sample message.

Figure 5.3 shows that for this specific sample message the parity bits and symbols remaining in the FlipFlops are respectively

$$P_{3_{Binary}} = 100110$$
$$P_{3_{Symbol}} = \alpha^0\alpha^3$$

Hence the complete codeword for this sample message will be given by

$$U_{3_{Binary}} = 100110111000001100110$$
$$U_{3_{Symbol}} = \alpha^0\alpha^3\alpha^5 0\alpha^2\alpha^0\alpha^3$$

5.1.4 The fourth sample message

The fourth simulation was for the following symbol and binary message

$$M_{4_{Symbol}} = \alpha^4\alpha^0\alpha^3\alpha^3\alpha^6$$
$$M_{4_{Binary}} = 011100110110101$$

The results for this encoding process is shown in figure 5.4

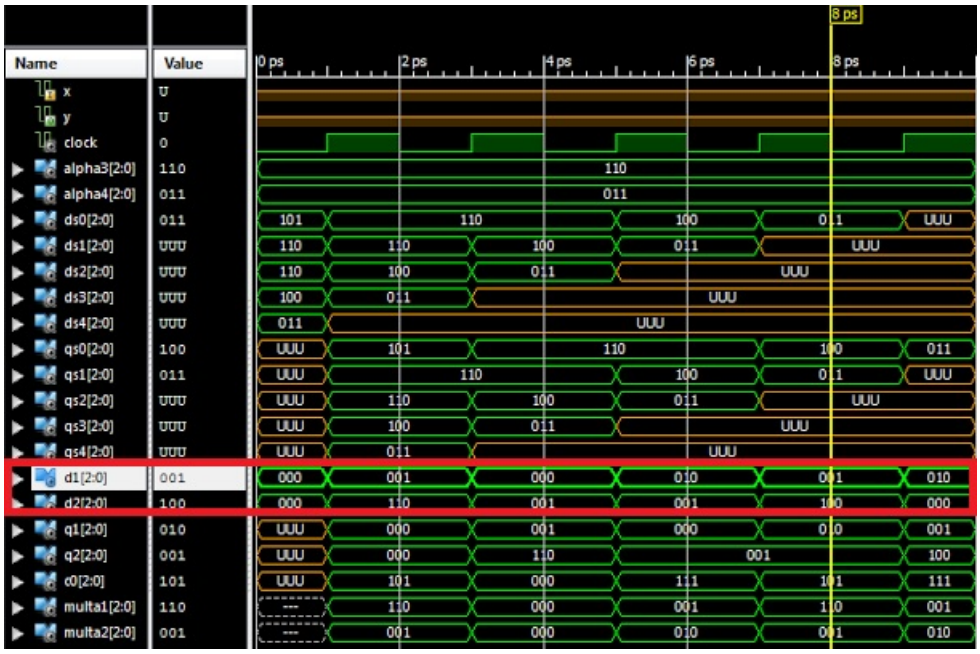

Figure 5.4: Encoding results for the fourth sample message.

Figure 5.4 shows that for this specific sample message the parity bits and symbols remaining in the FlipFlops are respectively

$$P_{4_{Binary}} = 010000$$
$$P_{4_{Symbol}} = \alpha^1 0$$

Hence the complete codeword for this sample message will be given by

$$U_{4_{Binary}} = 010000011100110110101$$
$$U_{4_{Symbol}} = \alpha^1 0 \alpha^4 \alpha^0 \alpha^3 \alpha^3 \alpha^6$$

5.1.5 The fifth sample message

The fifth simulation was for the following symbol and binary message

$$M_{5_{Symbol}} = \alpha^2 \alpha^3 \alpha^6 \alpha^0 \alpha^2$$
$$M_{5_{Binary}} = 001110101100001$$

The results for this encoding process is shown in figure 5.5

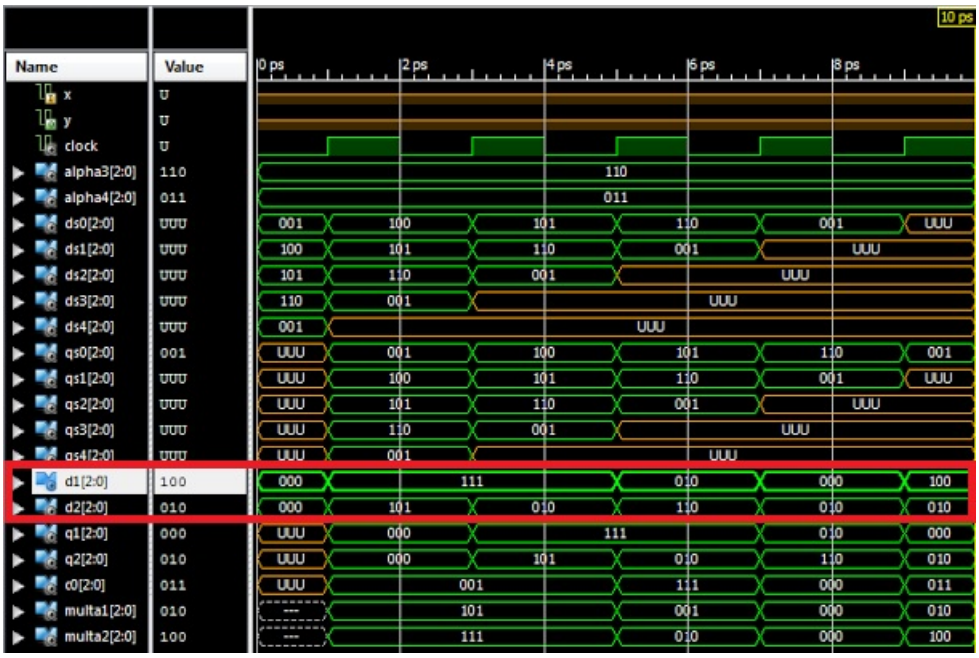

Figure 5.5: Encoding results for the fourth sample message.

Figure 5.5 shows that for this specific sample message the parity bits and symbols remaining in the FlipFlops are respectively

$$P_{5_{Binary}} = 100010$$
$$P_{5_{Symbol}} = \alpha^0\alpha^1$$

Hence the complete codeword for this sample message will be given by

$$U_{5_{Binary}} = 100010001110101100001$$
$$U_{5_{Symbol}} = \alpha^0\alpha^1\alpha^2\alpha^3\alpha^6\alpha^0\alpha^2$$

5.2 Syndrome Calculator Simulations

The simulation of the presented Decoder was done using the ISim software, a VHDL simulator which is a part of the XILINX ISE Tools.

The following values were set in the Force Clock.

Leading Edge Value	0
Trailing Edge Value	1
Starting at Time Offset	0.5
Cancel after Time Offset	14
Period	2

Table 5.2: Initial clock settings for decoding simulations.

5.2.1 The first sample codeword.

The first simulation was for the following symbol and binary codeword

$$U_{1_{Symbol}} = \alpha^1\alpha^5\alpha^3\alpha^6\alpha^2\alpha^4\alpha^0$$
$$U_{1_{Binary}} = 010111110101001011100$$

The results for this decoding process is shown in figure 5.6

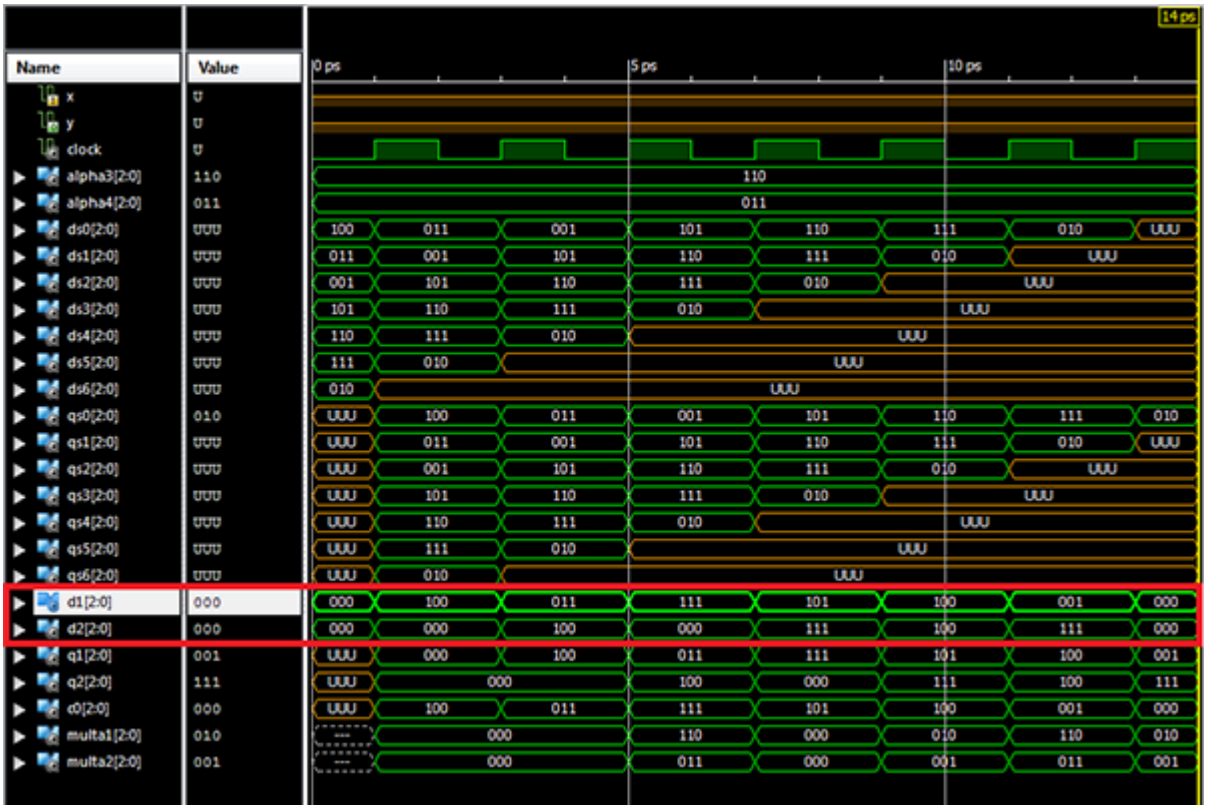

Figure 5.6: Decoding results for the first sample codeword.

Figure 5.6 shows that for this specific sample codeword the syndrome bits and symbols remaining in the FlipFlops are respectively

$$S_{1_{Binary}} = 000000$$
$$S_{1_{Symbol}} = 00$$

5.2.2 The second sample codeword

The second simulation was for the following symbol and binary codeword

$$U_{2_{Symbol}} = \alpha^0\alpha^1\alpha^3\alpha^4\alpha^6\alpha^1\alpha^0$$
$$U_{2_{Binary}} = 100010110011101010100$$

The results for this decoding process is shown in figure 5.7

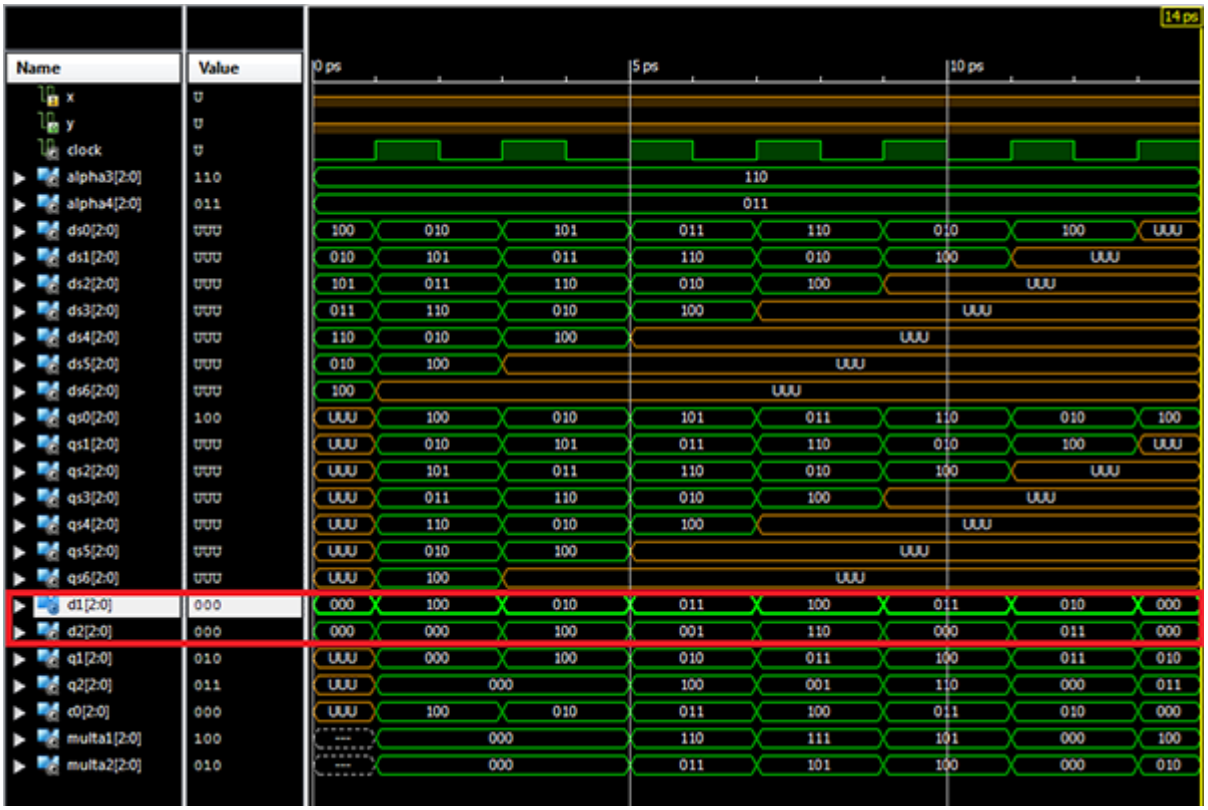

Figure 5.7: Decoding results for the second sample codeword.

Figure 5.7 shows that for this specific sample codeword the syndrome bits and symbols remaining in the FlipFlops are respectively

$$S_{2_{Binary}} = 000000$$
$$S_{2_{Symbol}} = 00$$

5.2.3 The third sample codeword

The third simulation was for the following symbol and binary codeword

$$U_{3_{Symbol}} = \alpha^0\alpha^3\alpha^5 0\alpha^2\alpha^0\alpha^3$$
$$U_{3_{Binary}} = 100110111000001100110$$

The results for this decoding process is shown in figure 5.8

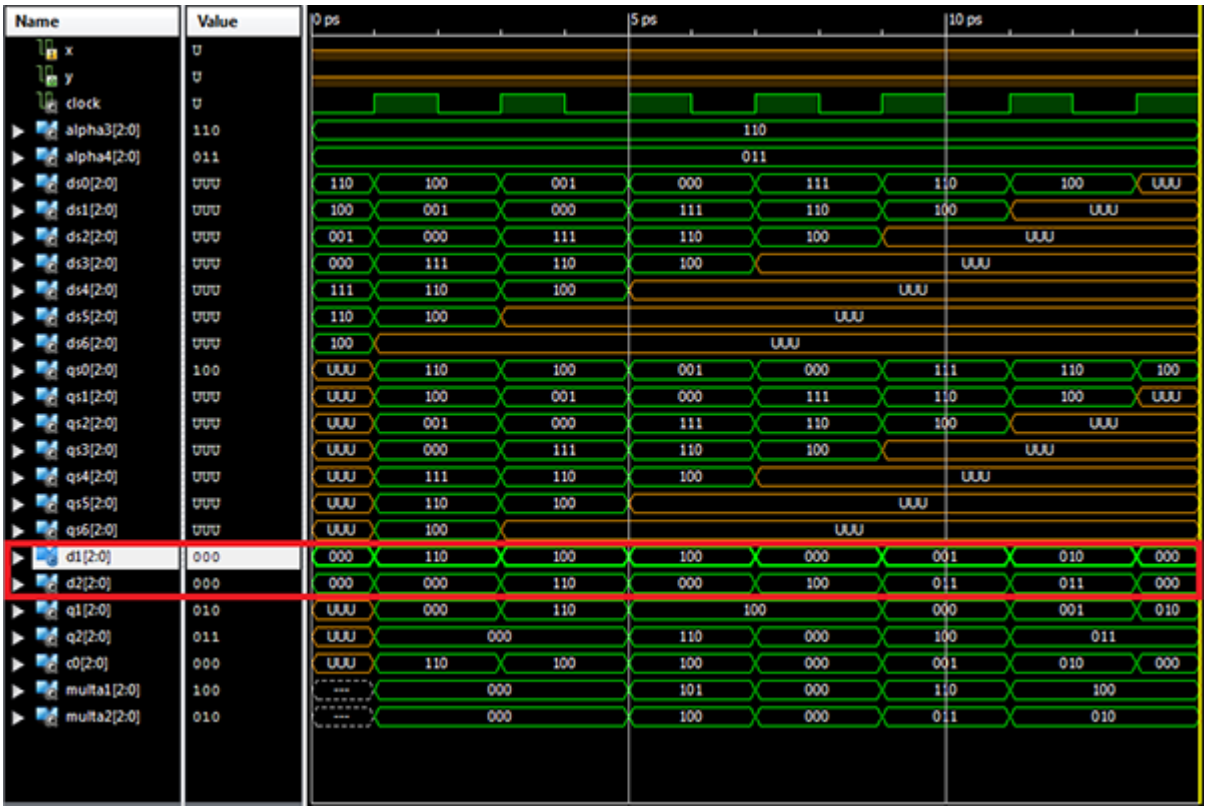

Figure 5.8: Decoding results for the third sample codeword.

Figure 5.8 shows that for this specific sample message the syndrome bits and symbols remaining in the FlipFlops are respectively

$$S_{3_{Binary}} = 000000$$
$$S_{3_{Symbol}} = 00$$

5.2.4 The fourth sample codeword

The fourth simulation was for the following symbol and binary codeword

$$U_{4_{Symbol}} = \alpha^1 0 \alpha^4 \alpha^0 \alpha^3 \alpha^3 \alpha^6$$
$$U_{4_{Binary}} = 010000011100110110101$$

The results for this decoding process is shown in figure 5.9

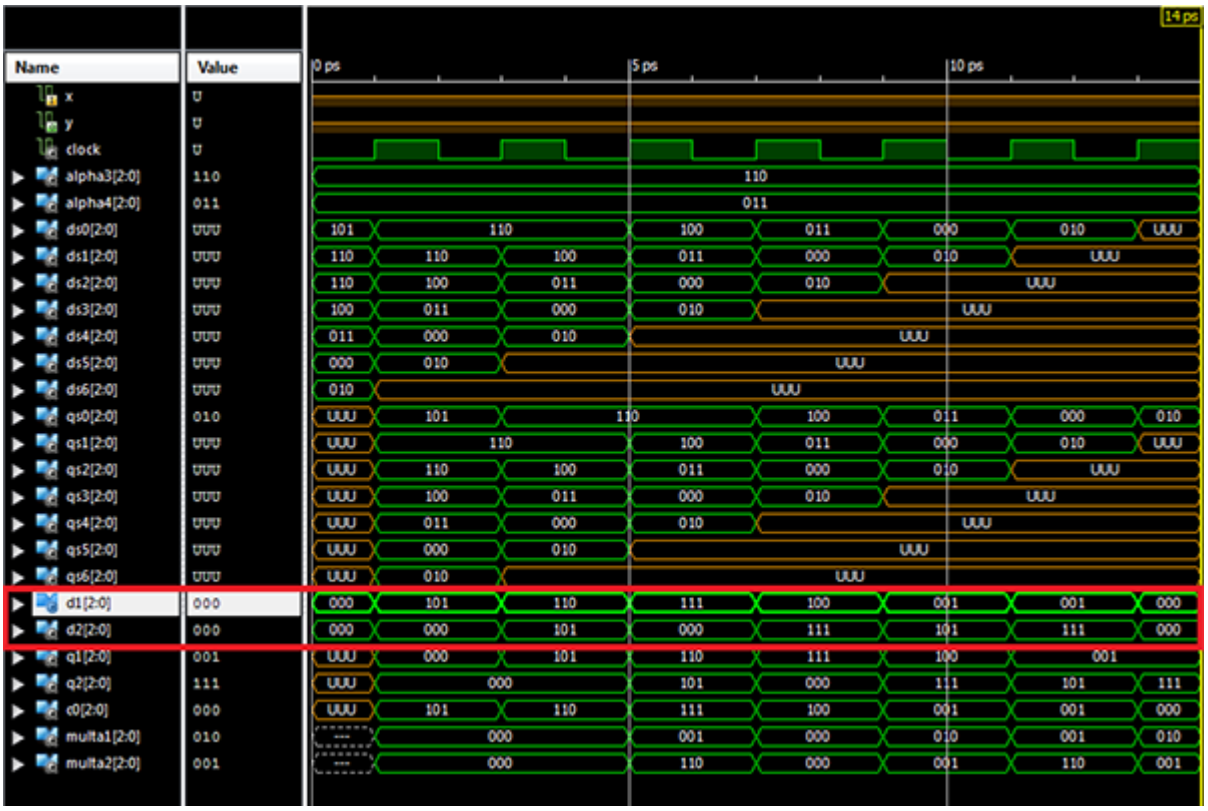

Figure 5.9: Decoding results for the fourth sample codeword.

Figure 5.9 shows that for this specific sample message the syndrome bits and symbols remaining in the FlipFlops are respectively

$$S_{4_{Binary}} = 000000$$
$$S_{4_{Symbol}} = 00$$

5.2.5 The fifth sample codeword

The fifth simulation was for the following symbol and binary codeword

$$U_{5_{Symbol}} = \alpha^0\alpha^1\alpha^2\alpha^3\alpha^6\alpha^0\alpha^2$$
$$U_{5_{Binary}} = 100010001110101100001$$

The results for this decoding process is shown in figure 5.10

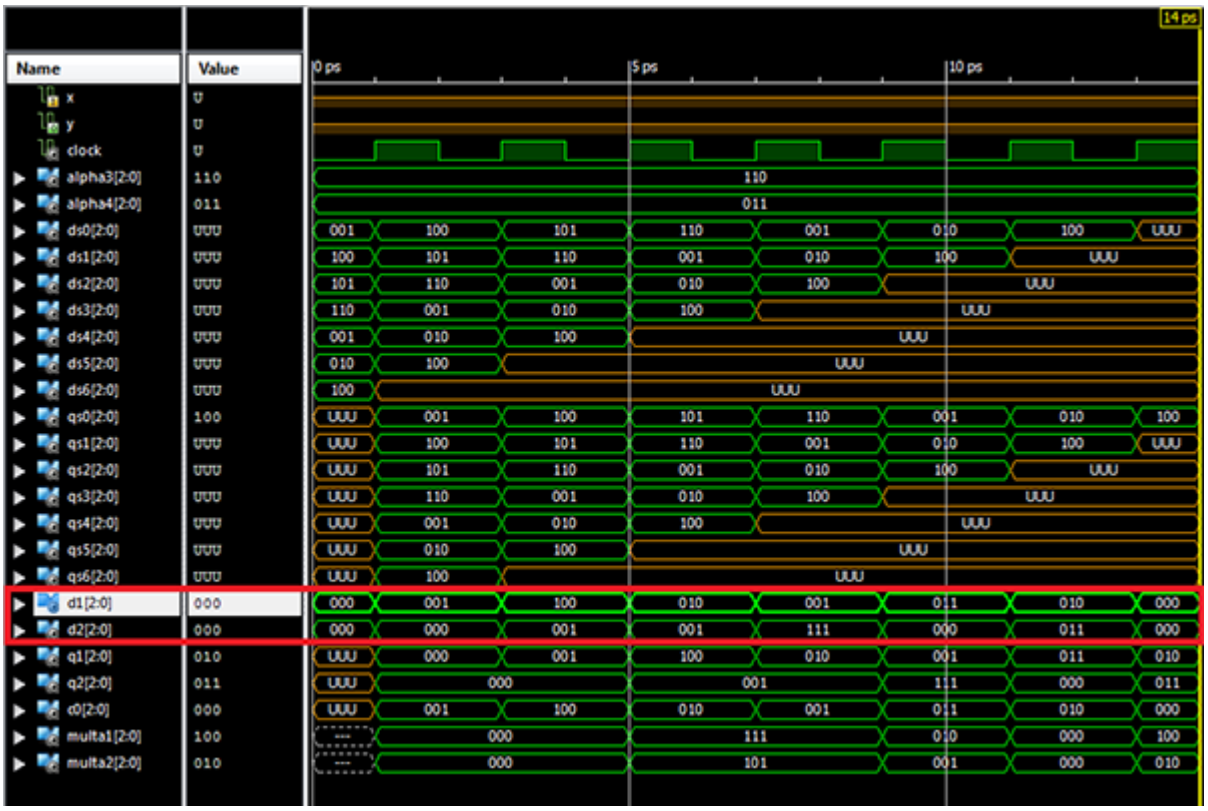

Figure 5.10: Decoding results for the fourth sample coderword.

Figure 5.10 shows that for this specific sample message the syndrome bits and symbols remaining in the FlipFlops are respectively

$$S_{5_{Binary}} = 000000$$
$$S_{5_{Symbol}} = 00$$

5.3 System Simulations

Now that the encoder and syndrome calculator are working in a proper manner the system can be simulated as a whole with an input message beign encoded, passing through the channel and beign decoded afterwards. Two simulations were done using the same sample message but with different constraints for the channel model.
In the first simulation the channel was assumed to generate at most 1 random symbol in error to be added into the system while in the second simulation 2 random symbol errors were generated by the channel. The system's behavorial is as follows.

5.3.1 Assuming 1 random symbol in error.

Let us assume the following sample binary message is to be transmitted by the system

$$M_{Binary} = 100100011000010$$

This message can be mapped in the Galois Field as

$$M_{Symbol} = \alpha^0\alpha^0\alpha^4 0\alpha^1$$

After passing through the Encoder the following codeword was generated

$$U_{Binary} = 1110011001000011000010$$

$$U_{Symbol} = \alpha^5\alpha^2\alpha^0\alpha^0\alpha^4 0\alpha^1$$

This codeword was transmitted through the model channel which has generated a random error of

$$e_{Binary} = 00000000000010000000$$

$$e_{Symbol} = 0000\alpha^1 00$$

The received codeword in the receiver was then

$$U'_{Binary} = 1110011001000001000010$$

$$U'_{Symbol} = \alpha^5\alpha^2\alpha^0\alpha^0\alpha^2 0\alpha^1$$

This codeword holds its value in the FIFO while passing through the syndrome calculator and syndrome for it was estimated to be

$$S_{Binary} = 011101$$

$$S_{Symbol} = \alpha^4\alpha^6$$

This syndrome then checks the estimated error assumed to be

$$\hat{\mathrm{e}}_{Binary} = 00000000000010000000$$

$$\hat{\mathrm{e}}_{Symbol} = 0000\alpha^1 00$$

Finally this is error was added to the codeword and the parity bits were ignored. The final results are

$$M_{Binary} = 100100011000010$$

$$M_{Symbol} = \alpha^0\alpha^0\alpha^4 0\alpha^1$$

Figure 5.11 shows all the processes that are working into the system from the message input until the system output.

Figure 5.11: System Simulation with 1 symbol in error.

The first thing to note from figure 5.11 is that it only takes 7 clock cycles to complete the entire process. This is because while the message is sent to the encoder it is also transmitted at the same time so that in the 5^{th} clock cycle the parity symbols start to be transmitted.
The variable *qp* are the parity symbols inside the encoder beign calculated at each clock cycle. Note how the counters select only the two last parity symbols by implementing the logic described in the encoder component.
The variable *errorgenerator* shows the random error generated by the model channel

using the LFSR logic. Note here that only 1 symbol error was added into the system by the channel. This error is added at each clock cycle to the message and furthermore to the parity symbols and this addition and the received codeword are represented by the variable *out*2.
The variables *db* and *qb* represent the inputs and outputs of the FIFO buffer constantly changing and shifting at each clock cycle. The variable *syndrome* is the syndrome being calculated at the syndrome calculator after each clock cycle. It is actually a concatenation of two individual symbols *syn*1 and *syn*2 that are output from the calculator. The variable *error* is the estimated error which is added to the output of the FIFO at the 7^{th} clock cycle. Even though the output *message* is changing its value before the system completes its cycle all the previous information received may just be ignored.

5.3.2 Assuming 2 random symbols in error.

Let us assume the same previous sample binary message is to be transmitted by the system

$$M_{Binary} = 100100011000010$$

$$M_{Symbol} = \alpha^0\alpha^0\alpha^4 0\alpha^1$$

After passing through the Encoder the following codeword was generated

$$U_{Binary} = 111001100100011000010$$

$$U_{Symbol} = \alpha^5\alpha^2\alpha^0\alpha^0\alpha^4 0\alpha^1$$

This codeword was transmitted through the model channel which has generated a random error of

$$e_{Binary} = 000000000100010000000$$

$$e_{Symbol} = 000\alpha^0\alpha^1 00$$

The received codeword in the receiver was then

$$U'_{Binary} = 111001100000001000010$$

$$U'_{Symbol} = \alpha^5\alpha^2\alpha^0 0\alpha^2 0\alpha^1$$

This codeword holds its value in the FIFO while passing through the syndrome calculator and syndrome for it was estimated to be

$$S_{Binary} = 111001$$

$$S_{Symbol} = \alpha^5\alpha^2$$

This syndrome then checks the estimated error assumed to be

$$\hat{e}_{Binary} = 00000000000000011000$$

$$\hat{e}_{Symbol} = 00000\alpha^4 0$$

Finally this is error was added to the codeword and the parity bits were ignored. The final results are

$$M_{Binary} = 100000001011010$$

$$M_{Symbol} = \alpha^0\alpha^0\alpha^4\alpha^4\alpha^1$$

This process is again illustrated at figure 5.12.

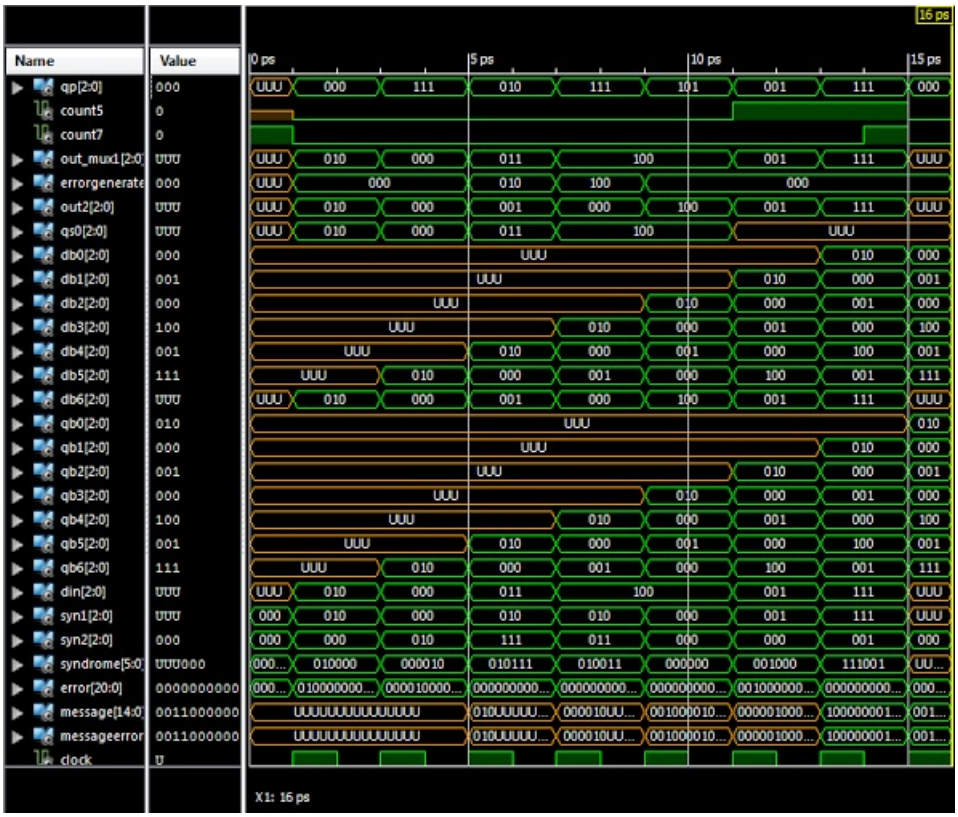

Figure 5.12: System Simulation with 2 symbols in error.

From figure 5.12 it is possible to note that the message was not recovered. Instead there is still one symbol in error that could lead into a possible system failure. One possible solution to this problem is retransmitting this new message and correct the left symbol in error or change the type of the Reed-Solomon Code in order to include more parity symbols and enhance the system performance.

CHAPTER 6

MESSAGE STREAMING

One of the main purposes of channel coding is to prevent errors from being added into the message while it is transmitted. The previous system was tested using a message that fits the exact amount of bits and symbols the system can handle at a time but if that number starts to increase some difficulties may appear. The synchronous system suddenly becomes hard to synchronize as many timing issues related to the Flip Flop resetting and clearing unnecessary and undefined symbols.
One common type of bit streaming is text streaming where the alphabet characters are mapped into binary through the ASCII (American Standard Code for Information Interchange). It maps 128 different characters into 7 bits each.
Let us test now how the system performs in terms of message streaming and really long binary messages. Assume the following text is to be transmitted using a (7,5) Reed-Solomon Code:

```
What we observe is not nature itself, but nature exposed
to our method of questioning. Werner Heisenberg
```

This message can be converted to binary by mapping it into the ASCII system and the corresponding long binary string will be given by:

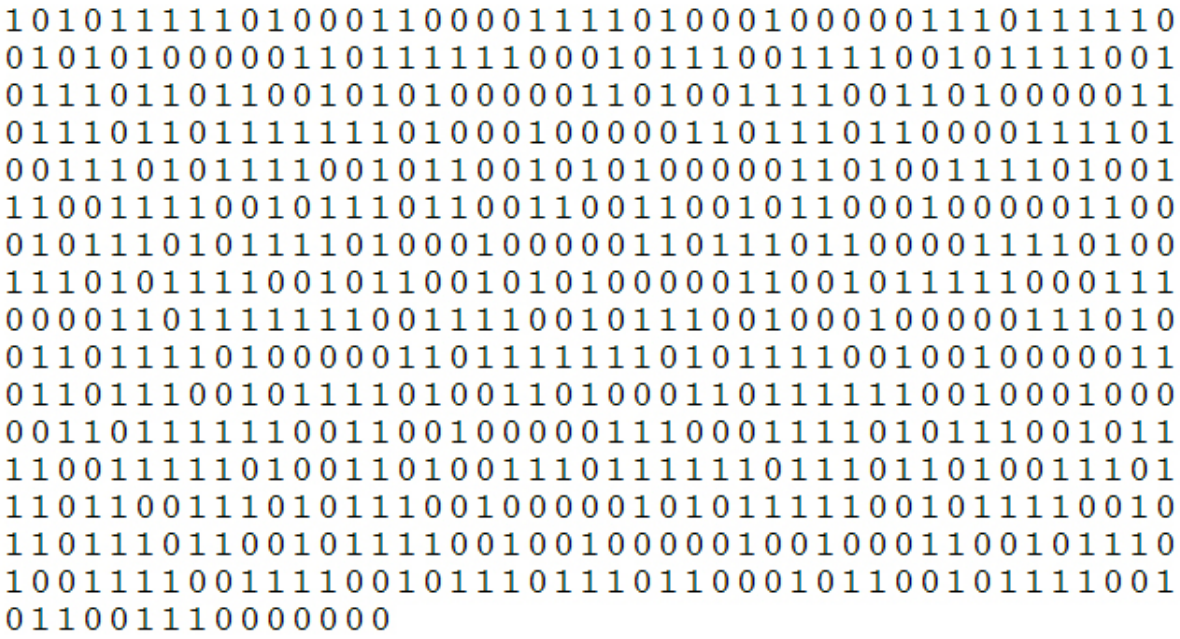

Figure 6.1: Binary text message.

Notice that it is very rare to have a combination of characters that will result into an exact multiple of 5 symbols or 21 bits. Therefore each before going into the system the characters must be grouped together in such a manner that the most significant bit of the following character is grouped with the least significant bit of the previous character. If the character's bits are to be broken apart because they don't fit the message length of 21 bits they will become the most significant bit of the next message to be sent.
The final message will most probably have some initial symbols but not enough bits in order to complete the whole message. In this case a zero padding is done into the least significant symbols as they do not affect the system.
Another consideration is that the Flip Flops must be cleaned right after each parity symbol or syndrome is calculated and output. This will guarantee that no remaining symbols will interfere with the next message. A counter may be used to do so.

The message symbols will enter the system in series and each 5 symbols will have a corresponding 2 parity symbols that will prevent those to be in error when received.

6.1 Message Streaming with 1 Symbol Error per Symbol Message

The (7,5) RS Code will have Symbol Messages containing 5 symbols each. Assume the channel will inject 1 symbol error at most every 7 symbols counting with the parity symbols. From the discussion above it is expected that the system will be able to recover the whole

message by the end of the process. This process was done using VHDL and a function that writes the message in error and the decoded message. The message in error will be the received message right after passing through the channel. A PHP code was used in order to convert the message back from binary to text. The results are as follows in figure 6.1.

Figure 6.2: Message Stream Simulation with 1 symbol in error per symbol message.

As the system requires many clock cycles it is not possible to show to entire functionality in one screen although the results were captured. Notice from figure 6.1 that the error generator is generating only 1 error for each 5 symbols and also that it may generate no error at all. The received message by the receiver after traveling through the channel and ignoring the parity symbols is

Figure 6.3: Received message after passing through the channel.

The bits highlighted in red are the bits in eror that were changed by the channel.After being translated back to text using PHP language this text will be

```
‘Wtap wd@ocse vm\&is"not"nature8ids\%lF, bu4 >aluje e|p
red 4s murmUthof kf,qu\}s~iio    nf.\&Wernuu Hciseo"e"g
```

The received message after the error correction is the same as the original message shown below

Figure 6.4: Corrected message after passing through the decoder.

The bits highlighted in green are the bits the system estimated to correct. After translating the final message received back to text the following results were observed

```
What we observe is not nature itself, but nature exposed
to our method of questioning. Werner Heisenberg
```

Therefore the system is capable to handle any 1 symbol error patterns that may be added by the channel.

6.2 Message Streaming with 2 Symbol Errors per Symbol Message

Now let us test the system using at most 2 symbols in error per symbol message and the same text. Figure 6.2 shows the results of a small part of the process running.

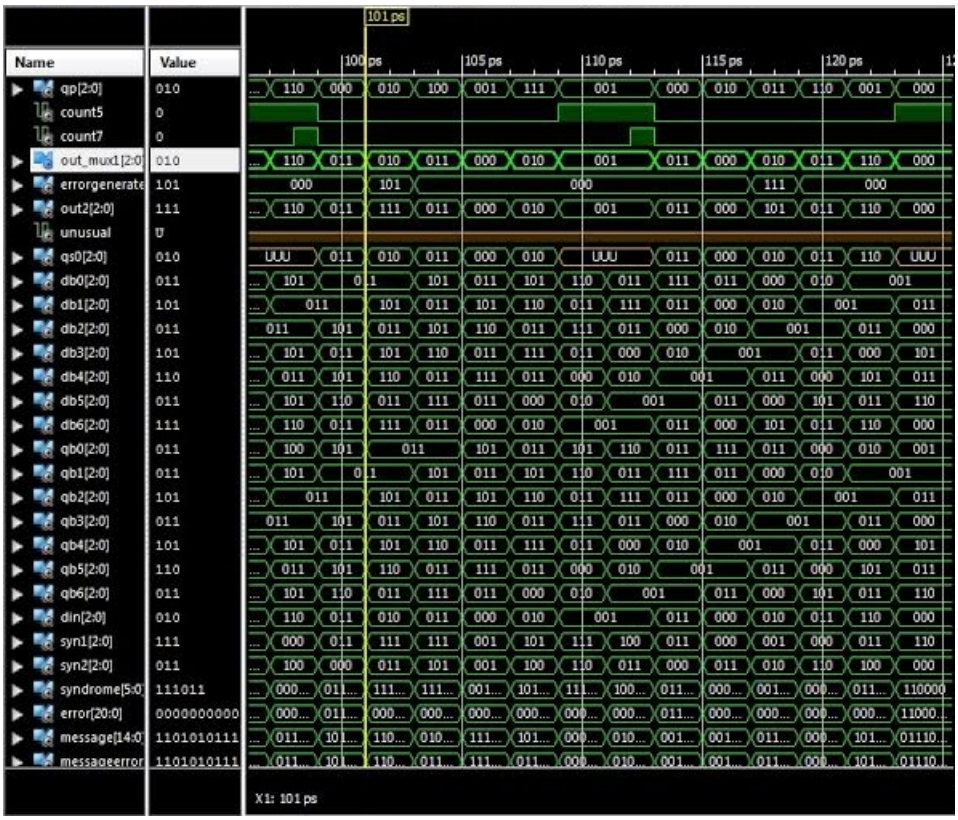

Figure 6.5: Message Stream Simulation with 2 symbols in error per symbol message.

Notice that now the channel is adding up to 2 symbols in error for each 7 symbols that travel through it. The received message with error by the receiver ignoring the parity symbols was

Figure 6.6: Received message after passing through the channel.

Which translated to text is

```
]h‘t od@onseru] \}sPz z!*Av2r‘ i sem ,?" t natune egAOrd\$"tnjn
r,neuJ/d,f Q eSxmhlin\&o Wf2oEV HWif\%DbZr’
```

After trying to correct the symbols in error the following message was decoded by the system

Figure 6.7: Corrected message after passing through the decoder.

The bits highlighted in blue are bits in error added by the system itself in an attempt of correcting previous errors. When translating the final decoded message to text the following results were obtained

```
What wdPobseru] isPz :!*Gv2r‘ itsem ,?"ut nat ne egQWrd\$
tnZour ne J/ ,j‘QueSy h–          n\&o8Wf2oMV(HWif\%DbZ|+
```

CHAPTER 7

CONCLUSION

Reed-Solomon codes are widely used in channel coding as they tend to be efficient. This project presented a small scale (7,5) RS Code capable of correcting 1 symbol in error for each 5 symbols of message.
All different blocks functioned correctly when simulated outside and into the system. The system it self has also proved to work with both static and dynamic messages and the only failures were those expected from 2 or more symbols errors per 5 symbols of message.
The discussions matched the theory and the Reed-Solomon channel coding system is could to be implemented in a larger scale if needed. Also a possible cryptography and compression could be implemented and used together with Reed-Solomon to design a full digital communication system.

CHAPTER 8

VHDL CODE

Listing 8.1: The VHDL code used to encode the symbol power.

```
entity SymbolPowerEncoder is
    Port ( n1 : in  std_logic_vector(2 downto 0);
           n1c : out  std_logic_vector(2 downto 0));
end SymbolPowerEncoder;

architecture Behavioral of SymbolPowerEncoder is

begin
process ( n1 )
        begin
                case n1 is
                        when "100"=> n1c <="000" ;
                        when "010"=> n1c <="001" ;
                        when "001"=> n1c <="010" ;
                        when "110"=> n1c <="011" ;
                        when "011"=> n1c <="100" ;
                        when "111"=> n1c <="101" ;
                        when "101"=> n1c <="110" ;
                        when others=> n1c <="---" ;
                end case ;
        end process ;

end Behavioral;
```

Listing 8.2: The VHDL code used to decode the symbol power.

```
entity SymbolPowerDecoder is
    Port ( n1 : in  std_logic_vector(2 downto 0);
           n1c : out  std_logic_vector(2 downto 0));
end SymbolPowerDecoder;

architecture Behavioral of SymbolPowerDecoder is

begin
process ( n1 )
        begin
                case n1 is
                        when "000"=> n1c <="100" ;
                        when "001"=> n1c <="010" ;
                        when "010"=> n1c <="001" ;
                        when "011"=> n1c <="110" ;
                        when "100"=> n1c <="011" ;
                        when "101"=> n1c <="111" ;
                        when "110"=> n1c <="101" ;
                        when others=> n1c <="---" ;
                end case ;
        end process ;

end Behavioral;
```

Listing 8.3: The FullAdder coded in VHDL.

```
entity fa is
    Port ( a : in  std_logic;
           b : in  std_logic;
           c : in  std_logic;
           sum1 : out std_logic;
           sum0 : out std_logic);
end fa;

architecture Behavioral of fa is

signal s1 : std_logic;
signal s2 : std_logic;
signal s3 : std_logic;
signal s4 : std_logic;
signal s5 : std_logic;
signal s6 : std_logic;
```

```
signal s7 : std_logic;
signal s8 : std_logic;

begin

    s1 <= (not a) and b and c;
    s2 <= a and (not b) and c;
    s3 <= a and b and (not c);
    s4 <= a and b and c;
    sum1 <= s1 or s2 or s3 or s4;
    s5 <= (not a) and b and (not c);
    s6 <= a and (not b) and (not c);
    s7 <= (not a) and (not b) and c;
    s8 <= a and b and c;
    sum0 <= s5 or s6 or s7 or s8;

end Behavioral;
```

Listing 8.4: The VHDL code of the component BinaryAdderSubtractor.

```
entity BinaryAdderSubtractor is
        port( a,b: in std_logic_vector(3 downto 0);
        fnc: in std_logic;
        s_or_d: out std_logic_vector(3 downto 0)
        );
end BinaryAdderSubtractor;

architecture Behavioral of BinaryAdderSubtractor is
component fa is
        Port (  a : in  std_logic;
                b : in  std_logic;
                c : in  std_logic;
                sum1: out std_logic;
                sum0: out std_logic);
end component;

        signal c: std_logic_vector(3 downto 0);
        signal pb: std_logic_vector(3 downto 0);

begin

pb(0) <= b(0) xor fnc;
```

```
pb(1) <= b(1) xor fnc;
pb(2) <= b(2) xor fnc;
pb(3) <= b(3) xor fnc;

fa0: fa port map(a(0), pb(0), fnc, c(0), s_or_d(0));
fa1: fa port map(a(1), pb(1), c(0), c(1), s_or_d(1));
fa2: fa port map(a(2), pb(2), c(1), c(2), s_or_d(2));
fa3: fa port map(a(3), pb(3), c(2), c(3), s_or_d(3));

end Behavioral;
```

Listing 8.5: The VHDL code of Mux6x3.

```
entity Mux6x3 is
        Port (  a : in std_logic_vector(2 downto 0) ;
                b : in std_logic_vector(2 downto 0) ;
                s : in std_logic ;
                f : out std_logic_vector(2 downto 0));
end Mux6x3 ;

architecture Behavior OF Mux6x3 is

begin

        with s select
                f <= b when '0',
                        a when others;

end Behavior ;
```

Listing 8.6: SymbolMultiplier VHDL code.

```
entity SymbolMultiplier is
        port( uncoded_a, uncoded_b:
        in std_logic_vector(2 downto 0);
        uncoded_multab: out std_logic_vector(2 downto 0)
        );
end SymbolMultiplier;

architecture Behavioral of SymbolMultiplier is

component BinaryAdderSubtractor is
```

```
        port( a,b: in std_logic_vector(3 downto 0);
        fnc: in std_logic;
        s_or_d: out std_logic_vector(3 downto 0)
        );
end component;

component Mux6x3 is
        Port (  a : in std_logic_vector(2 downto 0) ;
                b : in std_logic_vector(2 downto 0) ;
                s : in std_logic ;
                f : out std_logic_vector(2 downto 0));
end component ;

component SymbolPowerDecoder is
    Port ( n1 : in  std_logic_vector(2 downto 0);
           n1c : out  std_logic_vector(2 downto 0));
end component;

component SymbolPowerEncoder is
    Port ( n1 : in  std_logic_vector(2 downto 0);
           n1c : out  std_logic_vector(2 downto 0));
end component;

signal iszero: std_logic;
signal zerov: std_logic_vector(2 downto 0);
signal s_or_d: std_logic_vector(3 downto 0);
signal a: std_logic_vector(2 downto 0);
signal b: std_logic_vector(2 downto 0);
signal uncoded_multab_poly: std_logic_vector(2 downto 0);
signal multab: std_logic_vector(2 downto 0);
signal sa: std_logic_vector(3 downto 0);
signal sb: std_logic_vector(3 downto 0);
signal tt: std_logic;
signal t7: std_logic_vector(3 downto 0);
signal tres: std_logic_vector(3 downto 0);
signal sa2: std_logic_vector(2 downto 0);
signal sb2: std_logic_vector(2 downto 0);

begin

iszero <= (uncoded_a(0) or uncoded_a(1) or uncoded_a(2))
 and (uncoded_b(0) or uncoded_b(1) or uncoded_b(2));
```

```
encode1: SymbolPowerEncoder port map(uncoded_a, a);
encode2: SymbolPowerEncoder port map(uncoded_b, b);

sa(0) <= a(0);
sa(1) <= a(1);
sa(2) <= a(2);
sa(3) <= '0';

sb(0) <= b(0);
sb(1) <= b(1);
sb(2) <= b(2);
sb(3) <= '0';

fa0: BinaryAdderSubtractor port map(sa, sb, '0', s_or_d);
tt <= s_or_d(3) or (s_or_d(0) and s_or_d(1)
 and s_or_d(2));

t7(0) <= '1';
t7(1) <= '1';
t7(2) <= '1';
t7(3) <= '0';
fa1: BinaryAdderSubtractor port map(s_or_d, t7,'1',tres);

sa2(0) <= tres(0);
sa2(1) <= tres(1);
sa2(2) <= tres(2);

sb2(0) <= s_or_d(0);
sb2(1) <= s_or_d(1);
sb2(2) <= s_or_d(2);

mux1: Mux6x3 port map(sa2, sb2, tt, multab);

decode1: SymbolPowerDecoder port map(multab,
 uncoded_multab_poly);

zerov(0) <= '0';
zerov(1) <= '0';
zerov(2) <= '0';

mux2: Mux6x3 port map(uncoded_multab_poly, zerov, iszero,
 uncoded_multab);
```

```
end Behavioral;
```

Listing 8.7: SymbolAdder VHDL code.

```
entity SymbolAdder is
        Port ( a : in std_logic_vector(2 downto 0);
               b : in std_logic_vector(2 downto 0);
               c : out std_logic_vector(2 downto 0));
end SymbolAdder;

architecture Behavioral of SymbolAdder is

begin
        c(0) <= a(0) xor b(0);
        c(1) <= a(1) xor b(1);
        c(2) <= a(2) xor b(2);

end Behavioral;
```

Listing 8.8: ThreeBitFlipFlop VHDL code.

```
entity ThreeBitFlipFlop is
        Port ( D : in std_logic_vector(2 downto 0);
               Clock : in std_logic;
               Q : out std_logic_vector(2 downto 0));
end ThreeBitFlipFlop ;

architecture Behavior OF ThreeBitFlipFlop is
begin
        process
        begin
               wait until Clock'event and Clock = '1' ;
               Q <= D ;
        end process ;
end Behavior ;
```

Listing 8.9: Code for the Reed Solomon Encoder.

```
entity ReedSolomonEncoder is
    Port ( Clock : in  std_logic;
        Count7 : in  std_logic;
        Qs0 : in std_logic_vector(2 downto 0);
```

```
        Qp : out std_logic_vector(2 downto 0));
end ReedSolomonEncoder;

architecture Behavioral of ReedSolomonEncoder is
component flipflop is
         Port ( D: in std_logic_vector(2 downto 0) ;
                Clock : in std_logic;
                Reset : in std_logic;
                Q : out std_logic_vector(2 downto 0)) ;
end component;

component AdderXor is
         Port ( a: in   std_logic_vector(2 downto 0) ;
                b: in   std_logic_vector(2 downto 0);
                c: out  std_logic_vector(2 downto 0)) ;
end component;

component Mult is
        port( uncoded_a, uncoded_b:
        in std_logic_vector(2 downto 0);
        uncoded_multab: out std_logic_vector(2 downto 0)
        );
end component;

component mux6 IS
        Port (y1: in    std_logic_vector(2 downto 0 ) ;
                y0: in  std_logic_vector(2 downto 0 ) ;
                s: in   std_logic ;
                f: out  std_logic_vector(2 downto 0 ));
end component;

signal alpha3 : std_logic_vector(2 downto 0);
signal alpha4 : std_logic_vector(2 downto 0);
signal D1 : std_logic_vector(2 downto 0);
signal D2 : std_logic_vector(2 downto 0);
signal Q1 : std_logic_vector(2 downto 0);
signal Q2 : std_logic_vector(2 downto 0);
signal C0 : std_logic_vector(2 downto 0);
signal multa1 : std_logic_vector(2 downto 0);
signal multa2 : std_logic_vector(2 downto 0);

begin
```

```
alpha3(0) <= '0'; alpha3(1) <= '1'; alpha3(2) <= '1';
alpha4(0) <= '1'; alpha4(1) <= '1'; alpha4(2) <= '0';

ff1 : flipflop port map (D1,Clock,Count7,Q1);
ff2 : flipflop port map (D2,Clock,Count7,Q2);

add1 : AdderXor port map (Q2, Qs0, C0);

mult1 : Mult port map (C0, alpha4, multa1);
mult2 : Mult port map (C0, alpha3, multa2);

add2 : AdderXor port map(Q1, multa1, D2);

D1 <= multa2;

Qp <= Q2;

end Behavioral;
```

Listing 8.10: Code for the Reed Solomon Syndrome Calculator.

```
entity ReedSolomonDecoder is
    Port ( Clock : in std_logic;
        Count7 : in std_logic;
        Qs0: in std_logic_vector(2 downto 0);
        Dsyn1: out std_logic_vector(2 downto 0);
        Dsyn2: out std_logic_vector(2 downto 0));
end ReedSolomonDecoder;

architecture Behavioral of ReedSolomonDecoder is
component flipflop is
         Port ( D: in std_logic_vector(2 downto 0);
                Clock : in std_logic;
                Reset : in std_logic;
                Q : out std_logic_vector(2 downto 0));
end component;

component AdderXor is
         Port ( a: in std_logic_vector(2 downto 0);
                b: in std_logic_vector(2 downto 0);
                c: out std_logic_vector(2 downto 0)) ;
end component;
```

```
component Mult is
        port(uncoded_a, uncoded_b:
        in std_logic_vector(2 downto 0);
        uncoded_multab: out std_logic_vector(2 downto 0));
end component;


signal alpha3 : std_logic_vector(2 downto 0);
signal alpha4 : std_logic_vector(2 downto 0);
signal D1 : std_logic_vector(2 downto 0);
signal D2 : std_logic_vector(2 downto 0);
signal Q1 : std_logic_vector(2 downto 0);
signal Q2 : std_logic_vector(2 downto 0);
signal C0 : std_logic_vector(2 downto 0);
signal multa1 : std_logic_vector(2 downto 0);
signal multa2 : std_logic_vector(2 downto 0);

begin

alpha3(0) <= '0'; alpha3(1) <= '1'; alpha3(2) <= '1';
alpha4(0) <= '1'; alpha4(1) <= '1'; alpha4(2) <= '0';

add1 : AdderXor port map (multa1, Qs0, C0);

D1 <= C0;
ff1 : flipflop port map (D1,Clock,Count7,Q1);

add2 : AdderXor port map(Q1, multa2, D2);

ff2 : flipflop port map (D2,Clock,Count7,Q2);

mult1 : Mult port map (Q2, alpha3, multa1);
mult2 : Mult port map (Q2, alpha4, multa2);

Dsyn1 <= D1;
Dsyn2 <= D2;

end Behavioral;
```

Listing 8.11: Code for the complete Reed Solomon.

```
entity ReedSolomon is

end ReedSolomon;

architecture Behavioral of ReedSolomon is
component ReedSolomonEncoder is
    Port ( Clock : in std_logic;
        Count7 : in std_logic;
        Qs0 : in std_logic_vector(2 downto 0);
        Qp : out std_logic_vector(2 downto 0));
end component;

component ReedSolomonDecoder is
    Port ( Clock : in  std_logic;
        Count7 : in  std_logic;
        Qs0: in std_logic_vector(2 downto 0);
        Dsyn1: out std_logic_vector(2 downto 0);
        Dsyn2: out std_logic_vector(2 downto 0));
end component;

component counter is

   Port( Clock: in std_logic;
         Count5: out std_logic;
         Count7: out std_logic);

end component;

component mux6 is
        Port (y1: in std_logic_vector(2 downto 0);
                y0: in std_logic_vector(2 downto 0);
                s: in std_logic;
                f: out std_logic_vector(2 downto 0));
end component ;

component flipflop is
         Port ( D: in std_logic_vector(2 downto 0);
                Clock: in std_logic;
                Reset: in std_logic;
                Q: out std_logic_vector(2 downto 0));
end component ;

component ErrorGenerator is
```

```
    Port ( Clock : in  std_logic;
        Qout: out std_logic_vector(2 downto 0));
end component;

component ErrorGuessing is
    Port ( Syndrome: in std_logic_vector(5 downto 0);
           Error : out  std_logic_vector(20 downto 0));
end component;

component read_file is
        Port (Clock: in std_logic;
                Qout: out std_logic_vector(2 downto 0));

end component;

component write_file is
        Port (Clock: in std_logic;
        Message: in std_logic_vector(14 downto 0));

end component;

component write_error is
        Port (Clock: in std_logic;
        Message: in std_logic_vector(14 downto 0));

end component;

signal Qp: std_logic_vector(2 downto 0);
signal Count5: std_logic;
signal Count7: std_logic;
signal out_mux1: std_logic_vector(2 downto 0);
signal ErrorGenerated: std_logic_vector(2 downto 0);
signal out2: std_logic_vector(2 downto 0);
signal unusual: std_logic;

signal Qs0 : std_logic_vector(2 downto 0);
signal Db0 : std_logic_vector(2 downto 0);
signal Db1 : std_logic_vector(2 downto 0);
signal Db2 : std_logic_vector(2 downto 0);
signal Db3 : std_logic_vector(2 downto 0);
signal Db4 : std_logic_vector(2 downto 0);
signal Db5 : std_logic_vector(2 downto 0);
signal Db6 : std_logic_vector(2 downto 0);
```

```
signal Qb0 : std_logic_vector(2 downto 0);
signal Qb1 : std_logic_vector(2 downto 0);
signal Qb2 : std_logic_vector(2 downto 0);
signal Qb3 : std_logic_vector(2 downto 0);
signal Qb4 : std_logic_vector(2 downto 0);
signal Qb5 : std_logic_vector(2 downto 0);
signal Qb6 : std_logic_vector(2 downto 0);
signal Din : std_logic_vector(2 downto 0);
signal Syn1 : std_logic_vector(2 downto 0);
signal Syn2 : std_logic_vector(2 downto 0);
signal Syndrome : std_logic_vector(5 downto 0);
signal Error : std_logic_vector(20 downto 0);
signal Message: std_logic_vector(14 downto 0);
signal MessageError: std_logic_vector(14 downto 0);

signal Clock: std_logic;

begin

readdata: read_file Port map (Clock, Qs0);

muxCount: mux6 Port map (Qp, Qs0, Count5, Din);

RDE: ReedSolomonEncoder Port map (Clock, Count7, Din, Qp);

muxa: mux6 Port map (Qp, Din, Count5, out_mux1);

RndGen: ErrorGenerator Port map (Clock, ErrorGenerated);

out2(0) <= out_mux1(0) xor ErrorGenerated(0);
out2(1) <= out_mux1(1) xor ErrorGenerated(1);
out2(2) <= out_mux1(2) xor ErrorGenerated(2);

Db6 <= out2;

ffbuffer0 : flipflop Port map (Db0,Clock,'0',Qb0);
ffbuffer1 : flipflop Port map (Db1,Clock,'0',Qb1);
ffbuffer2 : flipflop Port map (Db2,Clock,'0',Qb2);
ffbuffer3 : flipflop Port map (Db3,Clock,'0',Qb3);
ffbuffer4 : flipflop Port map (Db4,Clock,'0',Qb4);
ffbuffer5 : flipflop Port map (Db5,Clock,'0',Qb5);
ffbuffer6 : flipflop Port map (Db6,Clock,'0',Qb6);
```

```
RDD: ReedSolomonDecoder Port map (Clock, Count7, out2,
 Syn1, Syn2);

Syndrome(0) <= Syn2(0);
Syndrome(1) <= Syn2(1);
Syndrome(2) <= Syn2(2);
Syndrome(3) <= Syn1(0);
Syndrome(4) <= Syn1(1);
Syndrome(5) <= Syn1(2);

ErrorGuess: ErrorGuessing Port map (Syndrome, Error);

MessageError(0) <= Db0(0);
MessageError(1) <= Db0(1);
MessageError(2) <= Db0(2);
MessageError(3) <= Db1(0);
MessageError(4) <= Db1(1);
MessageError(5) <= Db1(2);
MessageError(6) <= Db2(0);
MessageError(7) <= Db2(1);
MessageError(8) <= Db2(2);
MessageError(9) <= Db3(0);
MessageError(10) <= Db3(1);
MessageError(11) <= Db3(2);
MessageError(12) <= Db4(0);
MessageError(13) <= Db4(1);
MessageError(14) <= Db4(2);

Message(0) <= Db0(0) xor Error(0);
Message(1) <= Db0(1) xor Error(1);
Message(2) <= Db0(2) xor Error(2);
Message(3) <= Db1(0) xor Error(3);
Message(4) <= Db1(1) xor Error(4);
Message(5) <= Db1(2) xor Error(5);
Message(6) <= Db2(0) xor Error(6);
Message(7) <= Db2(1) xor Error(7);
Message(8) <= Db2(2) xor Error(8);
Message(9) <= Db3(0) xor Error(9);
Message(10) <= Db3(1) xor Error(10);
Message(11) <= Db3(2) xor Error(11);
Message(12) <= Db4(0) xor Error(12);
Message(13) <= Db4(1) xor Error(13);
Message(14) <= Db4(2) xor Error(14);
```

```
Db0 <= Qb1;
Db1 <= Qb2;
Db2 <= Qb3;
Db3 <= Qb4;
Db4 <= Qb5;
Db5 <= Qb6;

count1: counter Port map (Clock, Count5, Count7);

writedata: write_file Port map (Clock, Message);
writeerror: write_error Port map (Clock, MessageError);

end Behavioral;
```

Listing 8.12: Code uses to generate an random symbol error.

```
entity ErrorGenerator is
    Port ( Clock : in  std_logic;
        Qout: out std_logic_vector(2 downto 0));
end ErrorGenerator;

architecture Behavioral of ErrorGenerator is

component flipflop is
         Port ( D: in std_logic_vector(2 downto 0);
                Clock: in std_logic;
                Reset: in std_logic;
                Q: out std_logic_vector(2 downto 0));
end component;

signal SymbolRandom: std_logic_vector(2 downto 0);
signal ErrorStop: std_logic;
signal Result: std_logic_vector(2 downto 0);

signal Dsa0 : std_logic_vector(2 downto 0);
signal Dsa1 : std_logic_vector(2 downto 0);
signal Dsa2 : std_logic_vector(2 downto 0);
signal Dsa3 : std_logic_vector(2 downto 0);
signal Dsa4 : std_logic_vector(2 downto 0);
signal Dsa5 : std_logic_vector(2 downto 0);
signal Dsa6 : std_logic_vector(2 downto 0);
```

```
signal Qsa0 : std_logic_vector(2 downto 0);
signal Qsa1 : std_logic_vector(2 downto 0);
signal Qsa2 : std_logic_vector(2 downto 0);
signal Qsa3 : std_logic_vector(2 downto 0);
signal Qsa4 : std_logic_vector(2 downto 0);
signal Qsa5 : std_logic_vector(2 downto 0);
signal Qsa6 : std_logic_vector(2 downto 0);

signal counter_rnd: std_logic_vector(2 downto 0) := "000";

begin

ffsymbola0 : flipflop Port map (Dsa0,Clock,'0',Qsa0);
ffsymbola1 : flipflop Port map (Dsa1,Clock,'0',Qsa1);
ffsymbola2 : flipflop Port map (Dsa2,Clock,'0',Qsa2);
ffsymbola3 : flipflop Port map (Dsa3,Clock,'0',Qsa3);
ffsymbola4 : flipflop Port map (Dsa4,Clock,'0',Qsa4);
ffsymbola5 : flipflop Port map (Dsa5,Clock,'0',Qsa5);
ffsymbola6 : flipflop Port map (Dsa6,Clock,'0',Qsa6);

Result(0) <= Qsa2(1) xor Qsa4(1) xor Qsa3(1) xor Qsa2(2)
 xor Qsa0(0) xor Qsa2(0) xor Qsa1(2) xor Qsa5(0);
Result(1) <= Qsa4(0) xor Qsa1(0) xor Qsa5(0) xor Qsa2(0)
 xor Qsa2(2) xor Qsa1(2) xor Qsa0(2);
Result(2) <= Qsa2(2) xor Qsa0(2) xor Qsa2(2) xor Qsa4(1)
 xor Qsa1(1) xor Qsa5(1);
SymbolRandom(0) <= Qsa3(0) xor Qsa5(2) xor Qsa1(1)
 xor Qsa6(1) xor Qsa3(2) xor Qsa1(2) xor Qsa5(2);
SymbolRandom(1) <= Qsa5(1) xor Qsa1(1) xor Qsa2(1)
 xor Qsa4(0) xor Qsa0(2) xor Qsa6(0);
SymbolRandom(2) <= Qsa1(2) xor Qsa0(2) xor Qsa0(0)
 xor Qsa3(2) xor Qsa3(1);

Dsa0 <= Qsa1;
Dsa1 <= Qsa2;
Dsa2 <= Qsa3;
Dsa3 <= Qsa4;
Dsa4 <= Qsa5;
Dsa5 <= Qsa6;
Dsa6 <= Result;

process(Clock, Result)
```

```
begin
        if (Clock'event and Clock='1') then
                counter_rnd <= counter_rnd + 1;
                if ((Result(0) xor Result(2)
                xor Result(1))='1' and ErrorStop/='1')
                then
                        Qout(0) <= SymbolRandom(0);
                        Qout(1) <= SymbolRandom(1);
                        Qout(2) <= SymbolRandom(2);
                        ErrorStop <= '1';
                else
                        Qout <= "000";
                end if;
                if(counter_rnd = "111") then
                        counter_rnd <= "001";
                        ErrorStop <= '0';
                end if;
        end if;
end process;

end Behavioral;
```

Listing 8.13: Code used to return the syndrome's respective error.

```
entity ErrorGuessing is
    Port ( Syndrome: in std_logic_vector(5 downto 0);
           Error: out std_logic_vector(20 downto 0));
end ErrorGuessing;

architecture Behavioral of ErrorGuessing is

begin
process ( Syndrome )
        begin
                case Syndrome is
when "010011"=> Error <="00000000000000000100";
when "010111"=> Error <="00000000000000100000";
when "110111"=> Error <="00000000000100000000";
when "100100"=> Error <="00000000100000000000";
when "110011"=> Error <="00000100000000000000";
when "000100"=> Error <="00010000000000000000";
when "100000"=> Error <="10000000000000000000";
```

```
when "001111"=> Error <="00000000000000000010";
when "001101"=> Error <="00000000000000010000";
when "011101"=> Error <="00000000000010000000";
when "010010"=> Error <="00000000010000000000";
when "011111"=> Error <="00000010000000000000";
when "000010"=> Error <="00001000000000000000";
when "010000"=> Error <="01000000000000000000";
when "110101"=> Error <="00000000000000000001";
when "110100"=> Error <="00000000000000001000";
when "111100"=> Error <="00000000000001000000";
when "001001"=> Error <="00000000001000000000";
when "111101"=> Error <="00000001000000000000";
when "000001"=> Error <="00001000000000000000";
when "001000"=> Error <="00100000000000000000";
when "011100"=> Error <="00000000000000000110";
when "011010"=> Error <="00000000000000110000";
when "101010"=> Error <="00000000000110000000";
when "110110"=> Error <="00000000110000000000";
when "101100"=> Error <="00000110000000000000";
when "000110"=> Error <="00011000000000000000";
when "110000"=> Error <="11000000000000000000";
when "111010"=> Error <="00000000000000000011";
when "111001"=> Error <="00000000000000011000";
when "100001"=> Error <="00000000000011000000";
when "011011"=> Error <="00000000011000000000";
when "100010"=> Error <="00000011000000000000";
when "000011"=> Error <="00001100000000000000";
when "011000"=> Error <="01100000000000000000";
when "101001"=> Error <="00000000000000000111";
when "101110"=> Error <="00000000000000000111000";
when "010110"=> Error <="00000000000000111000000";
when "111111"=> Error <="00000000000111000000000";
when "010001"=> Error <="00000011100000000000000";
when "000111"=> Error <="00011100000000000000000";
when "111000"=> Error <="11100000000000000000000";
when "100110"=> Error <="00000000000000000101";
when "100011"=> Error <="00000000000000000101000";
when "001011"=> Error <="00000000000000101000000";
when "101101"=> Error <="00000000000101000000000";
when "001110"=> Error <="00000000101000000000000";
when "000101"=> Error <="00010100000000000000000";
when "101000"=> Error <="10100000000000000000000";
when OTHERS   => Error <="00000000000000000000000";
```

```
                end case;
        end process;

end Behavioral;
```

Listing 8.14: Counter used to counter the quantity of cycles ran.

```
entity counter is
   port(Clock: in std_logic;
        Count5: out std_logic;
        Count7: out std_logic);
end counter;

architecture Behavioral of counter is
   signal temp: std_logic_vector(0 to 2) := "000";

begin process(Clock)
   begin
               if(Clock'event and Clock='0') then
                       if temp = "000" or temp = "111"
                       then
                               Count7 <= '1';
                       else
                               Count7 <= '0';
                       end if;
               end if;
      if(Clock'event and Clock='1') then
                       Count7 <= '0';
                if temp="101" or temp="110" then
                   Count5 <= '1';
                else
                   Count5 <= '0';
            end if;
            if temp="111" then
               temp<="001";
            else
               temp <= temp + 1;
            end if;
      end if;
   end process;
end Behavioral;
```

Listing 8.15: This component was used in order to get the message symbols from a file.

```
entity read_file is
        Port (Clock: in std_logic;
                Qout: out std_logic_vector(2 downto 0)
        );
end read_file;

architecture Behavioral of read_file is

signal bin_value : std_logic_vector(2 downto 0):="000";

begin

    process
      file file_pointer : text;
        variable line_content : string(1 to 15);
      variable line_num : line;
        variable j, i : integer := 0;
        variable char : character:='0';
   begin
      file_open(file_pointer,"read.txt",READ_MODE);
      bin_value <= "UUU";
                wait for 1 ps;
                while not endfile(file_pointer) loop
      readline (file_pointer,line_num);
      READ (line_num,line_content);
        for j in 1 to 15 loop
            i := i + 1;
            char := line_content(16-j);
            if(char = '0') then
                bin_value(i-1) <= '0';
            else
                bin_value(i-1) <= '1';
            end if;
                                if (i = 3) then
                                        i := 0;
                                   wait for 2 ps;
                                end if;
        end loop;
                    bin_value <= "UUU";
                    wait for 4 ps;
      end loop;
```

```
      file_close(file_pointer);
        wait;
    end process;

         Qout <= bin_value;

end Behavioral;
```

Listing 8.16: Code used to write the final message inside a file.

```
entity write_file is
        port (clock: in std_logic;
                message: in std_logic_vector(14 downto 0)
        );
end write_file;

architecture behavioral of write_file is

signal counter_write: std_logic_vector(2 downto 0)
 := "000";
begin

process(clock)
  file file_pointer : text;
    variable line_content : string(1 to 15);
    variable line_num : line;
    variable i,j : integer := 0;
    variable char : character:='0';
begin
if (clock'event and clock='1') then
  counter_write <= counter_write + 1;
  if(i = 0 and counter_write = "111") then
        file_open(file_pointer,"write.txt",write_mode);
  end if;
  if (counter_write = "111") then
          for j in 0 to 14 loop
                if(message(j) = '0') then
                        line_content(15-j) := '0';
                else
                        line_content(15-j) := '1';
                end if;
          end loop;
```

```
            write(line_num,line_content);
            writeline (file_pointer,line_num);
            i := 1;
            counter_write <= "001";
  end if;
end if;
end process;

end behavioral;
```

BIBLIOGRAPHY

[1] LATHI B. P., DING Z. *Modern Digital and Analog Communication Systems*. Oxford University Press, New York, 4th Edition, 2009.

[2] SKLAR B. *Digital Communications: Fundamentals and Applications*. Prentice Hall, New Jersey, 2nd Edition, 2001.

LIST OF FIGURES

LIST OF TABLES