NTNU – Trondheim
Norwegian University of
Science and Technology

Energy Efficient Reed-Solomon Error Correction

Sindre Drolsum Flaten

Master of Science in Electronics
Submission date: June 2013
Supervisor: Bjørn B. Larsen, IET
Co-supervisor: Erling Furunes, Energy Micro AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Energy Efficient Reed-Solomon Error Correction

Sindre Drolsum Flaten

Master's Thesis

Department of Electronics and Telecommunications
Norwegian University of Science and Technology

June 27, 2013

Abstract

Energy efficient implementations are very important in order to increase the operating time for battery-powered devices. In this thesis a Reed-Solomon encoder and decoder have been implemented. The implementations have been synthesized using a 45nm technology library and power estimations have been performed. To find the most energy efficient implementation, several implementation techniques were evaluated. The implemented system is a 5-bit, RS(31, 27) code. For a Reed-Solomon encoder with low activity, the energy consumption can be reduced by over 40% with the use of clock gating. Several different Reed-Solomon decoder configurations were implemented and synthesized. When comparing the energy consumption of the different configurations, a configuration with two-parallel syndrome cells and pipelined Chien search, Forney and error correction module were found to be the most energy efficient. This configuration had a 36% lower energy consumption compared to a configuration with the same parallel syndrome cells, and no pipelined modules. It also had a 7% lower energy consumption compared to a configuration with the same pipelined modules and the standard syndrome cells.

Problem Description

This problem description has been formulated based on the problem description given by Energy Micro for the 2012 autumn project

In radio communication, Forward Error Correction is used to detect and correct errors that may occur during transmission of data. Forward Error Correction can be achieved using Reed-Solomon codes.

In battery-powered radio applications, a low-power Reed-Solomon implementation will improve the reliability of the system and increase the battery lifetime. Energy efficient hardware is crucial when designing a battery-powered application.

The student should implement a suggested topology for both the Reed-Solomon encoder and decoder, where the main focus is on low energy consumption. The implementation should be simulated and the energy consumption of the chosen design is to be evaluated. Based on these results, are there room for further improvements when it comes to energy consumption, area and speed? If so, try to use other techniques to reduce the energy consumption. What is the trade-off of different implementation techniques (measured in energy consumption, area and speed)?

The chosen implementation can be compared to the energy consumption of other hardware implementations.

Preface

This Master's thesis has been written at NTNU spring/- summer 2013 as a continuation of my project work autumn 2012. The assignment was given by Energy Micro in Oslo and involves implementation of a Reed-Solomon system where the main focus is on low energy consumption.

When starting the project work I did not have any experience with error correction or signal encoding. I have spent much time studying the Reed-Solomon concepts and how these could be implemented in HDL. Learning to use new tools and debugging the HDL implementations have also been vary time consuming. These are the things that are not directly described in a thesis, but they are a big part of the process. The most rewarding thing is often getting an implementation to work after hours of trail and error.

I would like to thank my project supervisor, Associate Professor Bjørn B. Larsen, and Erling Furunes at Energy Micro for guidance during this whole last year. I would also like to thank Dr. George Petrides for helping me with the Reed-Solomon theory.

- Sindre Drolsum Flaten
Trondheim, 27.06.2013

Contents

List of Figures

List of Tables

List of Abbreviations

ARQ	Automatic Repeat Request
ASIC	Application-Specific Integrated Circuit
BCH	Bose-Chaudhuri-Hocquenhem
CG	Clock Gating
DVB	Digital Video Broadcasting
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Array
GCD	Greatest Common Divider
GF	Galois Field
GTECH	Generic Technology
HDL	Hardware Description Language
LFSR	Linear Feedback Shift Register
PDK	Process Design Kit
ROM	Read Only Memory
RS	Reed-Solomon
RTL	Register-Transfer Level
SAIF	Switching Activity Interchange Format
VCD	Value Change Dump

Chapter 1

Introduction

1.1 Motivation

Many portable electronic devices have a digital communication system enabling them to receive and transmit data. These systems often use error detection and error correction techniques to achieve good communication. Error detection and correction enables digital data to be restored after being corrupted, because of noise or other types of error. There are different error detection schemes and error correction schemes. Error detection is often accomplished by a parity-, polarity- or a checksum scheme. In these techniques extra data is added before transmission, which then can be checked on the receiving end for possible errors. Error correction is often done using automatic repeat request (ARQ), or forward error correction (FEC) [1]. When using ARQ, the data is retransmitted if an error is detected. FEC is achieved by using an error correction code to encode the data. The encoder adds redundant data to the message, and the decoder uses this redundancy to correct the error. Reed-Solomon codes are one type of error correction code used to perform FEC.

When implementing FEC on a battery powered electronic device, a fast and energy efficient implementation is highly desirable. This will increase the battery lifetime of the device, and enable the system to be more reliable. Therefore a FEC implementation using Reed-Solomon codes should be designed with regards to minimize energy consumption.

1.2 Problem Description

A Reed-Solomon encoder and decoder are to be designed and implemented using Verilog. The main goal in this thesis is to find an energy efficient Reed-Solomon implementation. The Reed-Solomon encoder and decoder shall be synthesized

using a cell library. The power consumption is also to be estimated, and based on these estimations the energy consumption should been evaluated.

The Reed-Solomon decoding process is the most complex part of the Reed-Solomon system. The main focus should therefore been on implementing different techniques that could reduce the energy consumption of the decoder. The different decoder configurations are to be be compared and evaluated with regards to energy consumption.

1.3 Report Structure

A brief overview of this thesis can be given as:

- In chapter 2 general low power design techniques, the concept of Reed-Solomon codes and Galois field are described.
- In chapter 3 the Reed-Solomon encoding process is described.
- Chapter 4 explains the basics of Reed-Solomon decoding.
- In chapter 5 the different techniques used to implement the Reed-Solomon encoder and decoder are described.
- Chapter 6 presents methods for verification and test.
- Chapter 7 explains the synthesis and power estimation procedure.
- In chapter 8 the synthesis results and power estimation of the Reed-Solomon encoder and decoder are presented.
- In chapter 9 the results and power estimations are evaluated.
- Chapter 10 contains a discussion of the results in this thesis.
- Chapter 11 concludes this thesis.
- Chapter 12 contains suggestions for further work.

Chapter 2

Theory

2.1 Low Power Design Techniques

The need for lower power consumption is increasing, as we develop more and more portable electronic devices. Reducing the power consumption will increase the battery lifetime, which is important for portable devices. Low power design techniques can be used to make a more energy efficient device. These techniques are based on understanding the concepts of power consumption, which will be explained in the next sections.

2.1.1 Power Consumption in CMOS Technology

The power consumption in CMOS technology can be put into two main categories, *static* and *dynamic* power consumption. The total power consumption can then be written as in equation 2.1.

$$P_{total} = P_{static} + P_{dyn} \tag{2.1}$$

Another important factor when talking about power consumption, is the CMOS delay. The CMOS delay is shown in equation 2.2 [2].

$$T_d = \frac{C_L \times V_{dd}}{\mu C_{OX} \left(\frac{W}{L}\right) (V_{dd} - V_t)^2} \tag{2.2}$$

Where $\mu C_{OX} \left(\frac{W}{L}\right)$ is a technology dependent factor.

If, for instance, the supply voltage (V_{dd}) is lowered to reduce power consumption, the delay of the transistor will increase. Energy is power over time. The increased delay can therefore make the total amount of energy spend on one operation the

same, as if we had a circuit with higher supply voltage, thus lower delay. The CMOS delay must therefore be taken into account when designing for low power.

Static Power Consumption
Static power consumption is due to leakage currents in the CMOS transistor. There are three main types of leakages, sub-threshold leakage, gate leakage and diode leakage. As CMOS technology and threshold voltage are scaled down, the leakage currents increase [2]. Static power consumption can be written as equation 2.3, where $I_{leakage}$ consists of the three leakage currents mentioned in this section.

$$P_{static} = I_{leakage} V_{dd} \tag{2.3}$$

There are several methods that can be used to reduce the static power consumption. These include multiple V_T, multiple V_{dd}, power supply scaling and back biasing [3].

Dynamic Power Consumption
Dynamic power consumption occurs when CMOS gates are active and switching. The dynamic power consumption is composed of two factors: the power consumption when switching and the short-circuit power consumption. The dynamic power consumption can be written as shown in equation 2.4 [2]. The dynamic power consumption when the gate is switching is dependent on the switching activity α, the capacitance factor C_L, the power supply V_{dd} and the operating frequency f. The second part of dynamic power consumption is the short-circuit power consumption ($I_{sc}V_{dd}$). Short-circuit power occurs because CMOS transistors do not switch instantaneously. This means that there is a small period of time, when both the pull-up and pull-down paths in the gate are on simultaneously [3].

$$P_D = \alpha C_L V_{dd}^2 f + I_{sc} V_{dd} \tag{2.4}$$

The switching power is the dominant part of the dynamic power consumption. By reducing the factors in equation 2.4, we can reduce the power consumption. When, for instance, working at a Register-transfer level (RTL), the power supply and operating frequency are parameters that are not directly handled. To reduce the dynamic power when working at RTL, we focus on the switching activity, α. In the next subsections some RTL techniques for low power will be presented.

2.1.2 Glitch Reduction

Glitches occur when converting combinatorial paths with different propagation delays [3]. This conversion can cause oscillation in the system, making values on a register switch many times. The oscillation will consume dynamic power, which is unwanted. The oscillation can also start propagating to other parts of the design, which again will lead to more power being consumed. Glitches can also cause

timing delays, which are undesirable.

There are several techniques that can be used to handle the propagation of glitches. Gate-level control can be used to reduce glitches by pipelining the design. This can be very efficient, but it has some disadvantages. When the design is pipelined extra logic, such as registers and control logic, are added. This can cause a higher latency in the design. When using this technique the designer have to consider the trade-off.

Another approach is to add extra logic and rearrange the logic structure, so components that can cause oscillation are moved further back in the signal chain [3]. If, for instance, a multiplexer selects different signals that are going to be added together, the control signal for the multiplexer can oscillate giving the adders a oscillating input signal. To remove oscillation on the input to the adders, we can double the amount of adders and add the signals before we multiplex them. The downside is that extra logic must be implemented.

2.1.3 Clock Gating

Clock gating is an efficient technique for reducing the dynamic power in a system. The idea behind clock gating is to disable transitions from propagating to parts of the clock path, by using clock gating circuits [3]. This means, that logic elements in a circuit are not clocked when we do not need them. This is, for instance, when the system is in idle or redundant information is computed. By not loading unnecessary transitions when the clock is not active, we can save power due to decreased switching activity of the logic elements [3]. Clock gating can be implemented using a latch and an AND gate as shown in figure 2.1. Testability of a design can become worse when introducing clock gating, because this leads to multiple clock domains.

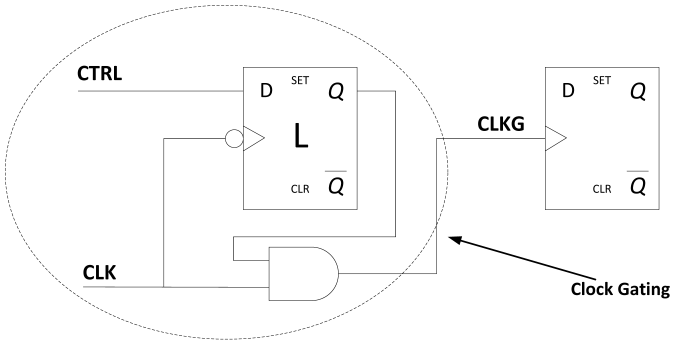

Figure 2.1: Clock gating principle

2.1.4 Precomputation and Parallelism

When using precomputation logic the idea is to precompute circuit output values one clock cycle before they are required [4]. This makes it possible turn off logic, and thus save power because of decreased switching activity. If there is no change of output values, we simply keep the previous computed values.

Parallel architecture can also be utilized to reduce power consumption [2]. By using parallel logic the system can work on half the speed and still be able to maintain the same data throughput. This comes at the expense of an increased circuit area.

2.2 Reed-Solomon

Reed-Solomon (RS) error correction is an error correction technique, which is used in communication systems and data storage applications for correcting errors that may occur during transmission or from disc reading errors [5]. Reed-Solomon codes were first described in the paper *Polynomial Codes Over Certain Finite Fields* [6]. Reed-Solomon codes are often used to perform Forward Error Correction (FEC). When using FEC, redundant (parity) information is added to the data before it gets transmitted or stored [7]. When the data is received/read, the receiver can detect and correct errors. The number of errors that can be corrected depends on the implementation of Reed-Solomon code. One of the advantages of the Reed-Solomon code is the ability to correct both random and burst errors [5]. The principal concept for Reed-Solomon codes are shown in figure 2.2.

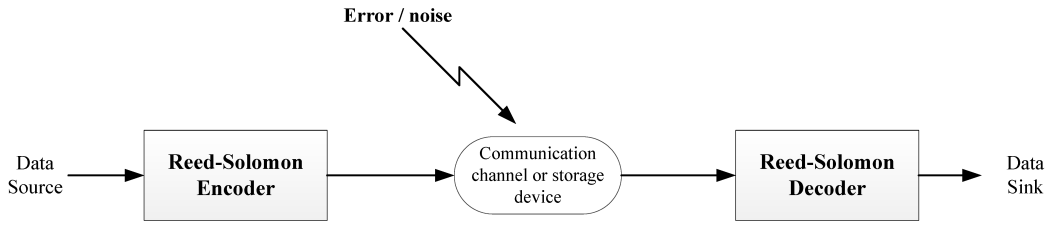

Figure 2.2: The concept of FEC using Reed-Solomon codes

Reed-Solomon codes are a sub class of Bose-Chaudhuri-Hocquenhem (BCH) codes [5] [8]. The Reed-Solomon code has several defined characteristics. Reed-Solomon codes are block codes, which means that the data message is divided into several blocks of data, often called symbols [7]. It is also a cyclic and linear code, meaning

that a codeword can be produced by adding two codewords together, or shifting the symbols of a codeword. The block symbols of a Reed-Solomon codeword are elements of a finite field [5]. Finite fields and finite field arithmetic will be further described in section 2.3 and 2.4.

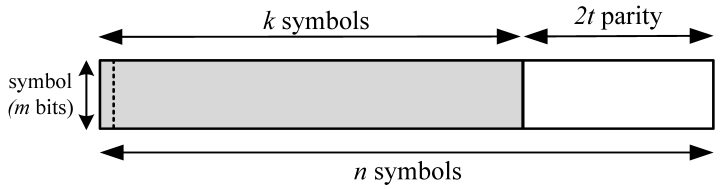

Figure 2.3: Reed-Solomon codeword structure

A Reed-Solomon code can be described as RS(n,k), where n is the total length of the codeword, k is the number of data symbols and $n - k + 1$ is the minimum distance. Each of the symbols in a Reed-Solomon code contain m number of bits, where the relationship between n and m is shown in equation 2.5. The structure of the Reed-Solomon code can be seen in figure 2.3.

$$n = 2^m - 1 \tag{2.5}$$

The Reed-Solomon code can correct up to t errors by adding $2t$ parity symbols to the codeword, as shown in equation 2.6. If the locations of the errors are known, the Reed-Solomon code can correct up to twice as many errors. Errors with known locations are called erasures. A Reed-Solomon code can also correct a combination of errors and erasures as long as equation 2.7 is satisfied.

$$2t = n - k \tag{2.6}$$

$$2v_{errors} + v_{erasures} \leq n - k \tag{2.7}$$

Where v_{errors} in equation 2.7 is the number of errors and $v_{erasures}$ is the number of erasures in the code. In this thesis only error correction are dealt with.

Since Reed-Solomon codes are able to correct burst errors, they are used in storage devices (CDs, DVDs, Blu-Ray Discs etc.), mobile communication, modems, Digital Video Broadcasting (DVB) and barcodes, to name a few.

2.3 Galois Field

The symbols in a Reed-Solomon code are elements of a finite field, also called Galois field (GF) . Galois field consists of a finite set of elements, meaning that it can be represented by a fixed length word. A Galois field can be written as $GF(p^m)$, where p is a prime number and m is an integer. The elements in a Galois field are based on a primitive element, denoted α [7]. By using α, the elements of a Galois field can be represented in index form as:

$$0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, ..., \alpha^{N-1} \tag{2.8}$$

A binary field can be constructed by choosing the the primitive element α to be 2. The binary field will form a set of 2^m elements, and the field can be written as $GF(2^m)$. If the primitive element equals 2, the power N in (2.8) will then be $N = 2^m - 1$.

A Galois field element can also be represented by a polynomial expression with the base x^m, as shown in equation 2.9 [7].

$$a_{m-1}x^{m-1} + ... + a_1x^1 + a_0x^0 \tag{2.9}$$

If coefficients a_{m-1} to a_0 of the polynomial expression take the values 0 or 1, we can represent a field element with a binary number. 2^m elements in Galois field, can then represent 2^m combinations of a m-bit number. If $m = 4$ the Galois field is GF(2^4), which is GF(16). This field has 16 elements, and can be represented by a 4-bit number (binary: 0000 to 1111 or decimal: 0 to 15).

2.3.1 Field Generator Polynomial

A Galois field can be constructed using a field generator polynomial or primitive polynomial. This polynomial is the minimal polynomial of a primitive element of the finite extension field GF(p^m). The primitive polynomial $p(x)$ is of degree m and is irreducible, meaning it has no factors [7]. The primitive element α is a root of the primitive polynomial $p(x)$. By using this, all non-zero elements of GF(p^m) can be constructed using a successive power of α. A large field have several primitive polynomials, and each primitive polynomial give a unique representation of the elements. For instance, the field GF(16) have two primitive polynomials, $p(x) = x^4 + x + 1$ and $p(x) = x^4 + x^3 + 1$ [9]. To construct the field for GF(16) the primitive polynomial $p(x) = x^4 + x + 1$ will be used.

When constructing the field, the primitive polynomial is set equal to zero, $p(\alpha) = 0$. This can be done because the primitive element α is a root of the primitive polynomial. For the given primitive polynomial this can be written as:

$$p(\alpha) = \alpha^4 + \alpha + 1 = 0 \tag{2.10}$$

which is:

$$\alpha^4 = \alpha + 1 \tag{2.11}$$

To construct the whole field in polynomial form, α is multiplied in at each stage. When the polynomial form reaches α^4, $\alpha + 1$ is substitute for α^4. The resulting terms is finally added together using Galois field addition. The first five elements of GF(16) in polynomial form are $\{0, 1, \alpha, \alpha^2, \alpha^3\}$, and rest of the non-zero elements in $GF(16)$ are found in the following way:

$$\begin{array}{lcll}
\alpha^4 & = & \alpha + 1 & \\
\alpha^5 & = & \alpha(\alpha^4) & = \alpha(\alpha + 1) = \alpha^2 + \alpha \\
\alpha^6 & = & \alpha(\alpha^5) & = \alpha(\alpha^2 + \alpha) = \alpha^3 + \alpha^2 \\
\alpha^7 & = & \alpha(\alpha^6) & = \alpha(\alpha^3 + \alpha^2) = \alpha^4 + \alpha^3 = \alpha^3 + \alpha + 1 \\
\alpha^8 & = & \alpha(\alpha^7) & = \alpha(\alpha^3 + \alpha + 1) = \alpha^4 + \alpha^2 + \alpha = \alpha^2 + \alpha + \alpha + 1 = \alpha^2 + 1 \\
\alpha^9 & = & \alpha(\alpha^8) & = \alpha(\alpha^2 + 1) = \alpha^3 + \alpha \\
\alpha^{10} & = & \alpha(\alpha^9) & = \alpha(\alpha^3 + \alpha) = \alpha^4 + \alpha^2 = \alpha^2 + \alpha + 1 \\
\alpha^{11} & = & \alpha(\alpha^{10}) & = \alpha(\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha \\
\alpha^{12} & = & \alpha(\alpha^{11}) & = \alpha(\alpha^3 + \alpha^2 + \alpha) = \alpha^4 + \alpha^3 + \alpha^2 = \alpha^3 + \alpha^2 + \alpha + 1 \\
\alpha^{13} & = & \alpha(\alpha^{12}) & = \alpha(\alpha^3 + \alpha^2 + \alpha + 1) = \alpha^4 + \alpha^3 + \alpha^2 + \alpha = \alpha^3 + \alpha^2 + \alpha + \alpha + 1 = \alpha^3 + \alpha^2 + 1 \\
\alpha^{14} & = & \alpha(\alpha^{13}) & = \alpha(\alpha^3 + \alpha^2 + 1) = \alpha^4 + \alpha^3 + \alpha = \alpha^3 + \alpha + \alpha + 1 = \alpha^3 + 1
\end{array}$$

Table A.1 in appendix A.1.1 shows all the elements of $GF(16)$ in index form, polynomial form, binary form and decimal form. It should be noted that the elements after α^{14} will only repeat the same sequence as shown in table A.1.

2.4 Galois Field Arithmetic

An arithmetic operation performed on a Galois field element will result in another element of the same field, since only a finite set of elements exist. In this section arithmetic operations such as addition, subtraction, multiplication and division will be explained. Implementation of these arithmetic operations using hardware are described in section 5.2.

2.4.1 Addition and Subtraction in Galois Field

Addition and subtraction in Galois field are done in exactly the same way using a exclusive-OR function (XOR), or by modulo 2 addition/subtraction of the coefficients [7]. Since addition and subtraction have exactly the same effect, addition is used when performing a subtraction operation. In polynomial form this is written as shown in equation 2.12.

$$\sum_{i=m-1}^{0} a_i x^i + \sum_{i=m-1}^{0} b_i x^i = \sum_{i=m-1}^{0} c_i x^i \tag{2.12}$$

Since addition is a XOR operation and the coefficients can only take the value 0 or 1, $c_i = 0$ when $a_i = b_i$ and $c_i = 1$ when $a_i \neq b_i$ for $0 \leq i \neq m - 1$.

If we want to add the numbers 12 and 15 in GF(16) this will result in 3. Using a polynomial expression this gives:

$$(x^3 + x^2) + (x^3 + x^2 + x + 1) = x + 1$$

Shown with binary numbers:

$$\begin{array}{ll} 1100 & (12) \\ \underline{1111} & (\underline{15}) \\ 0011 & (3) \end{array}$$

An example with subtraction is not shown, since subtraction in done in exactly the same way as addition.

2.4.2 Multiplication and Division in Galois Field

When multiplying two polynomials with degree $m-1$, the resulting product polynomial would have a degree of $2m - 2$. In Galois field multiplication the product can not be larger than the largest element of the field $GF(2^m)$, thus multiplication in Galois field is defined as the product modulo the field generator polynomial $p(x)$ [7]. The product modulo can be found by dividing the product polynomial by the field generator polynomial $p(x)$, and then take the remainder. This will always give a result that is inside the Galois field.

There are several different ways in which the remainder can be found. One possible way is to first multiply the values using the polynomial expression, and then divide the result by the field generator polynomial. This division is done by multiplying the divisor by a value to make it the same degree as the dividend, and then subtracting the divisor from the dividend [7]. An example on how this is done is shown below:

We want to multiply the two values 12 and 15 in Galois field GF(16). First, we multiply the two values using the polynomial expression. Second, we use Galois addition on the values with the same exponents, as shown below.

$$\begin{aligned} (x^3 + x^2)(x^3 + x^2 + x + 1) &= x^6 + x^5 + x^4 + x^3 + x^5 + x^4 + x^3 + x^2 \\ &= x^6 + x^2 \end{aligned}$$

Then the result is divided by the field generator polynomial.

	x^6	x^5	x^4	x^3	x^2	x^1	x^0
dividend:	1	0	0	0	1	0	0
divisor $\times x^2$:	1	0	0	1	1		
				1	0	0	0

The resulting remainder is then the product of the two values, which was binary 1000 or 8 in decimal. The value that was used to get the divisor in the same degree as the dividend, is called the quotient. In this example the quotient was x^2.

Dividing two elements in a Galois field can be done by multiplying by the inverse of the divisor. The inverse of a field element is defined as the element value, that when multiplied by the field element produces a value of 1 [7]. Below is an example on how this can be done.

We want to divide 15 by 12. First the inverse of 12 is found, which is:
$12 = \alpha^6$
$\alpha^{(-6)mod15} = \alpha^9 = 10$
$15 \div 12 = 15 \times 10$

Then 15 is multiplied by 10 to get the result. This can be done using the multiplication technique previously described in this section.

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

	x^6	x^5	x^4	x^3	x^2	x^1	x^0
dividend:	1	1	0	0	1	1	0
divisor $\times x^2$:	1	0	0	1	1		
		1	0	1	0	1	
divisor $\times x$:		1	0	0	1	1	
				1	1	0	0

Dividing 15 by 12 gives us binary 1100, which is 12 in decimal.

Chapter 3

Reed-Solomon Encoder

3.1 Generator Polynomial

The Reed-Solomon generator polynomial $g(x)$ is used to construct the Reed-Solomon code. It consists of $n - k = 2t$ factors and is described in equation 3.1 [7].

$$\begin{aligned} g(x) &= (x + \alpha^b)(x + \alpha^{b+1}).....(x + \alpha^{b+2t-1}) \\ &= x^{2t} + g_{2t-1}x^{2t-1} + g_2x^2 + g_1x + \cdots + g_0 \end{aligned} \tag{3.1}$$

Where α in equation 3.1 is a primitive element of the Galois field, and b can be chosen to be $b = 0$. It is possible to choose another value for b. This will however give a generator polynomial constructing a different Reed-Solomon code. g_0 to g_{2t-1} are the coefficients of the generator polynomial. The Reed-Solomon generator polynomial must not be mistaken for Galois field generator polynomial which was presented in section 2.3.1.

3.2 Reed-Solomon Encoding

The Reed-Solomon encoder is used to encode the message block by adding the parity symbols to the original message. The message polynomial which is to be encoded can be written as shown in equation 3.2 [7].

$$M(x) = M_{k-1}x^{k-1} + ... + M_2x^2 + M_1x^1 + M_0x^0 \tag{3.2}$$

k in equation 3.2 is total number of information symbols that form the message polynomial $M(x)$. The size of a symbol varies depending on the amount of information the user wants to send.

Before the message is transmitted, the encoder shifts the message polynomial $n-k$ times. This is done so the parity symbols can be added to the end of the message. Then the encoder divide by the generator polynomial $g(x)$. This will give a quotient $q(x)$ and a remainder $r(x)$ as shown in equation 3.3 [7]. Multiplying by $g(x)$ on both sides in equation 3.3 and using the fact that addition and subtraction are the same in finite field, equation 3.3 can be written as 3.4.

$$\frac{M(x) \times x^{n-k}}{g(x)} = q(x) + \frac{r(x)}{g(x)} \tag{3.3}$$

As seen from equation 3.4, the remainder is added to the end of the message $M(x)$ as the parity symbols.

$$M(x) \times x^{n-k} + r(x) = q(x) \times g(x) \tag{3.4}$$

The message $M(x)$ and the remainder $r(x)$ form the transmitted codeword $T(x)$ as shown in equation 3.5.

$$T(x) = M(x) \times x^{n-k} + r(x) \tag{3.5}$$

As seen from equation 3.5, the transmitted codeword is equal to the code generator polynomial multiplied by a quotient. The code generator polynomial consist of a number of factors and these are also factors of the encoded message [7]. The received message will therefore be divisible by the generator polynomial without remainder. However, if this is not true the received message will contain errors. This is further described in section 4.3.

Chapter 4

Reed-Solomon Decoder

4.1 The Received Codeword

The transmitted codeword $T(x)$ can get corrupted when being transmitted. The errors that occur in the transmission phase, can be described by the error polynomial $E(x)$. These errors can be on any of the symbols in the transmitted codeword, and can therefore be described as shown in equation 4.1.

$$E(x) = E_{n-1}x^{n-1} + E_{n-2}x^{n-2} + ... + E_2x^2 + E_1x^1 + E_0x^0 \tag{4.1}$$

When the transmitted codeword is received by the decoder, the codeword will consist of the messaged sent from the encoder and an error part. This is shown in equation 4.2 [7].

$$R(x) = T(x) + E(x) \tag{4.2}$$

The received message is represented by the polynomial $R(x)$ and take the form:

$$R(x) = R_{n-1}x^{n-1} + R_{n-2}x^{n-2} + ... + R_2x^2 + R_1x^1 + R_0x^0 \tag{4.3}$$

4.2 Decoding Techniques

The Reed-Solomon decoding process can be split into two parts. The first part of the decoding is to detect if an error has occurred during transmission of the codeword. The decoder does this by checking if the codeword is a valid codeword or not. If an error has occurred, the decoder will try to correct this error by finding the error position and error value. In this chapter methods for detecting and correcting errors in a Reed-Solomon code will be presented.

Figure 4.1, taken from [10] gives an overview of some of the Reed-Solomon decoding techniques. The figure shows that Reed-Solomon decoding techniques can be divided into two main classes. The left side of the figure consists of algorithms that use syndrome calculation to find the error locations and the error values in the received codeword. On the right side of the figure, no syndrome calculation is needed to find the error locations and error values. These techniques are sometimes called transform decoding without transforms [5].

An algebraic decoding process using syndrome calculation consists of five main steps. First the syndromes are calculated based on the received codeword. Then the error locator and error evaluator polynomials are found using the syndromes. The error locations are found by evaluating the error locator polynomial. When the locations of the errors are found, the two polynomials are used to find the error values. The last step consists of correcting the errors in the received codeword. When using transform decoding, the received codeword is first transformed to the frequency domain [5]. Then the frequency error vector is obtained by recursive extension. At the end, an inverse Fourier transform is done to find the error values in the time domain.

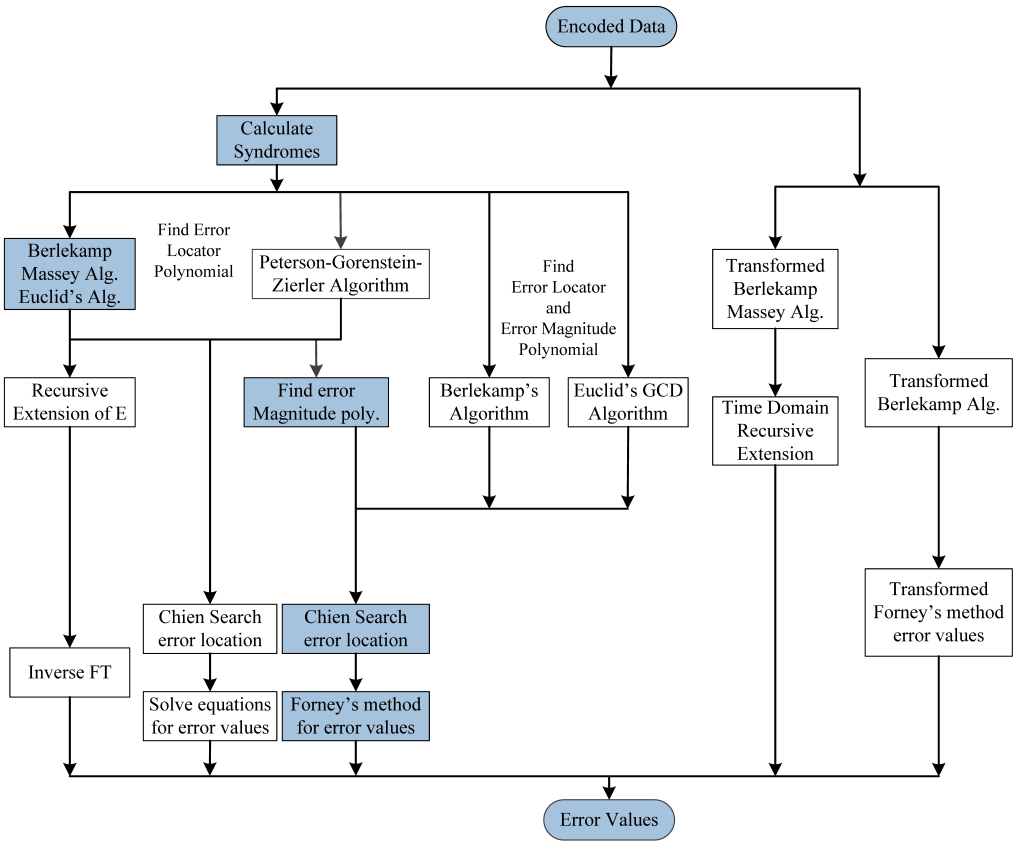

Figure 4.1: Reed-Solomon decoding techniques [10]

As seen from figure 4.1, the error locator polynomial and error evaluator polynomial can be found using several different algorithms. The Berlekamp-Massey and the Euclidean algorithm are widely used in decoding schemes for finding these polynomials [10] [11] [12] [13]. In this thesis a decoding process using syndrome calculation has been chosen. The methods highlighted in figure 4.1 are the methods presented in the next subsections.

4.3 Syndrome Decoding

To be able to find the errors and correct them, the decoder first has to calculate the syndromes. There are $n - k$ syndrome values that have to be calculated. The syndromes are only dependent on the error pattern, so if all the syndromes are equal to zero there are no errors on the received codeword [7].

The syndrome calculations can be done in different ways. The received codeword can be divided by the generator polynomial $g(x)$, which essentially is dividing it by all it's factors, to get the syndromes. The division will produce a quotient and a remainder. It is also possible to use substitution of the roots on the received codeword. This will also give us the syndrome values.

The first approach can be written as shown in equation 4.4, where each syndrome is represented by S_i and i is in the range $0 \leq i \leq (n-k)-1$ [7]:

$$\frac{R(x)}{g_i(x)} = q_i(x) + \frac{S_i}{g_i(x)} \qquad where \quad g_i(x) = (x + \alpha^i) \tag{4.4}$$

If all the syndromes are zero there are no errors, as previously explained. This is an important property of a Reed-Solomon code, and will have an impact on how the syndrome equation is described.

When using substitution to find the syndromes, equation 4.4 is rewritten to equation 4.5 [7]. When α^i is substituted in for x this will give $q_i \times (x + \alpha^i) = 0$ because adding the same values in a Galois field produces zero, as shown in section 2.4.1.

$$\begin{aligned} S_i &= q_i(x) \times (x + \alpha^i) + R(x) \\ &= q_i(\alpha^i) \times (\alpha^i + \alpha^i) + R(\alpha^i) \\ &= q_i(\alpha^i) \times 0 + R(\alpha^i) \\ &= R(\alpha^i) \end{aligned}$$

$$S_i = R_{n-1}(\alpha^i)^{n-1} + R_{n-2}(\alpha^i)^{n-2} + ... + R_2(\alpha^i)^2 + R_1(\alpha^i)^1 + R_0(\alpha^i)^0 \tag{4.5}$$

The syndromes can also be expressed as an syndrome polynomial as shown in equation 4.6.

$$S(x) = \sum_{i=0}^{2t-1} S_i x^i \tag{4.6}$$

As was mentioned earlier the, syndromes are only dependent on the errors introduced. This is used when producing an equation used to represent the error locations and the error values. This can be written as shown in equation 4.7 [7]. Y_v represent the error values and X_v represent the error location of a specific error. v is total number of errors where $v \leq t$.

$$\begin{aligned} S_i = E(\alpha^i) &= \sum_{l=1}^{v} Y_l \alpha^{ie_l} \\ &= \sum_{l=1}^{v} Y_l X_l^i \end{aligned} \tag{4.7}$$

$$\begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ \vdots \\ S_{2t-1} \end{bmatrix} = \begin{bmatrix} X_1^0 & X_2^0 & \cdots & X_v^0 \\ X_1^1 & X_2^1 & \cdots & X_v^1 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ X_1^{2t-1} & X_2^{2t-1} & \cdots & X_v^{2t-1} \end{bmatrix} \times \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_v \end{bmatrix}$$

The matrices above shows how the $2t$ syndrome equations can be written with regards to the location and value of the error.

4.4 Decoding Algorithm

The usual approach when wanting to find and correct errors using Reed-Solomon codes is to first calculate the syndromes, then find the error locator polynomial based on the syndromes. After this is done, the roots of the error locator polynomial can be evaluated to find the location of the error. When the location is known, the error value can be found. The last step in the decoder process is to correct the error.

A equation know as the *key equation*, is often used to find the error locator polynomial and error evaluator polynomial. These two polynomials are used in the decoding process to find the error locations and error values. The process of finding the error locator polynomial and the error evaluator polynomial, thus solving equation 4.8, is sometimes refereed to as the *key equation solver* [5].

$$\Omega(x) = S(x)\sigma(x) \, mod \, x^{2t} \tag{4.8}$$

The key equation shown in equation 4.8 describes the relationship between the error locator polynomial, the syndrome polynomial and the error evaluator polynomial.

The error locator polynomial, which is used to find the error locations, can be defined as shown in equation 4.9 [7].

$$\begin{aligned} \sigma(x) &= (1 + X_1 x)(1 + X_2 x)....(1 + X_v x) \\ &= 1 + \sigma_1 x + ... + \sigma_{v-1} x^{v-1} + \sigma_v x^v \end{aligned} \tag{4.9}$$

In equation 4.9 the inverse X_v^{-1} of the error locators are the roots and v is the errors.

To solve equation 4.8, the error evaluator polynomial also has to be found. The error evaluator polynomial, also referred to as the error magnitude polynomial, is a polynomial defined as shown in equation 4.10. This polynomial is later used to find the error values.

$$\Omega(x) = \sum_{i=v-1}^{0} \Omega_i x^i \tag{4.10}$$

Based on the key equation we can write the error evaluator polynomial as shown in equation 4.11. Where v is the errors.

$$\Omega(x) = \sum_{l=1}^{v} Y_l X_l \prod_{j=1, j\neq l}^{v} (1 - X_j x) \tag{4.11}$$

The syndrome polynomial used in the key equation was defined in equation 4.6 section 4.3.

Several different techniques can be used to solve the key equation, and find the coefficients of the error locator polynomial $\sigma(x)$. One method is to use the fact that equation 4.9 has errors located at X_l, if the root X_l^{-1} makes $\sigma(x) = 0$ for $l = 1, 2, ..., v$. By using this, we can multiply by $Y_l X_l^{i+v}$ on both sides of equation 4.9, as shown in equation 4.12 [5].

$$Y_l(X_l^{i+v} + \sigma_1 X_l^{i+v-1} + ... + \sigma_v X_l^i) = 0 \tag{4.12}$$

For each value of l and i we can write:

$$\sum_{l=1}^{v} Y_l X_l^{i+v} + \sigma_1 \sum_{l=1}^{v} Y_l X_l^{i+v-1} + ... + \sigma_v \sum_{l=1}^{v} Y_l X_l^i = 0 \tag{4.13}$$

If equation 4.7 is combined with 4.13 we can write [7]:

$$S_{i+v} + \sigma_1 S_{i+v-1} + ... + \sigma_v S_i = 0 \qquad i = 0, 1, ..., 2t - v - 1 \tag{4.14}$$

$$S_i \sigma_v + S_{i+1} \sigma_{v-1} + ... + S_{i+v-1} \sigma_1 = -S_{i+v} \tag{4.15}$$

Equation 4.15 can be used to find the coefficients of the error locator polynomial. For v errors v equations can be written to find the coefficients σ. This can be written as a matrix as shown in 4.16 [5].

$$\begin{bmatrix} S_0 & S_1 & \cdots & S_{v-1} \\ S_1 & S_2 & \cdots & S_v \\ S_2 & S_3 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ S_{v-1} & S_v & \cdots & S_{2v-2} \end{bmatrix} \times \begin{bmatrix} \sigma_v \\ \sigma_{v-1} \\ \sigma_{v-2} \\ \vdots \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_v \\ -S_{v+1} \\ -S_{v+2} \\ \vdots \\ -S_{2v-1} \end{bmatrix} \tag{4.16}$$

The equations in matrix 4.16 are called Newton identities. To solve the matrix and find the values of σ, the inverse of the matrix in 4.16 has to be found [5]. This would require that we know how many errors there are (must know v). One way of finding v is by choosing an appropriate value for v, for instance $v = t$, and then calculate the determinant of the matrix in 4.16. If the determinant is non-zero, the correct value of v was chosen (correct amount of errors) [5]. If the determinant is zero, another value for v must be chosen. This is the *direct method* for finding the error locator polynomial. When the error locator polynomial has been found using the direct method, the error evaluator polynomial can be found by substituting the error locator polynomial into the key equation.

There are also other approaches that can be used to find the coefficients of the error locator polynomial and the error evaluator polynomial. The Berlekamp-Massey and the Euclidean algorithm are two algorithms, that are commonly used to solve the key equation. These techniques are more efficient than the one described here.

4.4.1 Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm is a algorithm used to find the coefficients of the error locator polynomial shown in equation 4.9. These coefficients can then be substituted into equation 4.8 to find the coefficients of the error evaluator polynomial, equation 4.10, thus solving the *key equation*. The coefficients can be found by evaluating the syndromes shown in equation 4.6. The Berlekamp-Massey algorithm is an iterative method, and it uses the equations shown in matrix 4.16 to find the coefficients of the error locator polynomial [1].

The Berlekamp-Massey algorithm finds the error locator polynomial using two main steps:

First the minimum-degree polynomial $\sigma_{BM}^{\mu}(x)$, that satisfies the μth Newton identity in equation 4.16, is calculated.

The second step checks if the first polynomial $\sigma_{BM}^{\mu}(x)$ satisfies the $(\mu+1)$ Newton identity. If it does, then $\sigma_{BM}^{\mu+1}(x) = \sigma_{BM}^{\mu}(x)$. However, if it does not satisfy the $(\mu+1)$ Newton identity, a correction term is added to $\sigma_{BM}^{\mu}(x)$ to get the polynomial $\sigma_{BM}^{\mu+1}(x)$. This correction term is called discrepancy, and is denoted d_μ. These steps are continued until μ equals $2t$ ($2t$ iteration steps). The error locator polynomial is then formed by the $\sigma_{BM}^{2t}(x)$ polynomial. The minimum-degree polynomial obtained in the μth iteration can be written as shown in equation 4.17 [1].

$$\sigma_{BM}^{\mu}(x) = 1 + \sigma_1^{\mu}x + \sigma_2^{\mu}x^2 + \cdots + \sigma_{l_\mu}^{\mu}x^{l_\mu} \tag{4.17}$$

Where l_μ is the degree of the polynomial $\sigma_{BM}^{\mu}(x)$. The discrepancy d_μ can be found using equation 4.18 [1].

$$d_\mu = s_{\mu+1} + \sigma_1^{\mu}s_\mu + \sigma_2^{\mu}s_{\mu-1} + \cdots + \sigma_{l_\mu}^{\mu}s_{\mu+1-l_\mu} \tag{4.18}$$

Calculation of the minimum-degree polynomial $\sigma_{BM}^{\mu+1}(x)$ in iteration $\mu+1$ can be done in the following way:

Algorithm 1 Calculating the $\mu+1$ iteration using the Berlekamp-Massey algorithm

1: $Input : d_\mu$
2: **if** $d_\mu = 0$ **then**
3: $\quad \sigma_{BM}^{\mu+1}(x) = \sigma_{BM}^{\mu}(x)$
4: $\quad l_{\mu+1} = l_\mu$
5: **else if** $d_\mu \neq 0$ **then**
(Goes back to a previous row ρ, such that $d_\rho \neq 0$ and $\rho - l_\rho$ is maximum)
6: $\quad \sigma_{BM}^{\mu+1}(x) = \sigma_{BM}^{\mu}(x) + d_\mu d_\rho^{-1} x^{(\mu-\rho)}\sigma^{(\rho)}(x)$
7: $\quad l_{(\mu+1)} = max(l_\mu, l_\rho + \mu - \rho)$
8: $\quad d_{\mu+1} = s_{\mu+2} + \sigma_1^{\mu+1}s_{\mu+1} + \sigma_2^{\mu+1}s_\mu + \cdots + \sigma_{l_{\mu+1}}^{\mu+1}s_{\mu+2-l_\mu}$
9: **end if**

The Berlekamp-Massey algorithm is initialized using the following parameters:

$$\begin{aligned} \mu &= -1 \\ \sigma_{BM}^{\mu} &= 1 \\ d_\mu &= 1 \\ l_\mu &= 0 \\ \mu - l_\mu &= -1 \end{aligned}$$

When the $2t$ iterations are complete, the minimum degree error locator polynomial $\sigma_{BM}^{(2t)}$ has been found. The Berlekamp-Massey algorithm is a serial algorithm where the error locator polynomial is first found. Then the error locator polynomial is substituted into the key equation to find the error evaluator polynomial. Once these two polynomials have been found, the Chien search and the Forney method can then be used to find the error locations and error values.

As seen in line six of algorithm 1, the Berlekamp-Massey algorithm requires inversion. To remove the use of inversion in the Berlekamp-Massey algorithm, an inversionless Berlekamp-Massey algorithm can be used.

4.4.2 Inversionless Berlekamp-Massey Algorithm

In [14] an inversionless Berlekamp-Massey algorithm was presented, which removes the use of inversion when using the Berlekamp-Massey algorithm. The algorithm presented in [14] is a $2t$-step iterative algorithm and is shown in algorithm 2.

Algorithm 2 Inversionless Berlekamp-Massey algorithm

Initial Condition:

$D^{-1} = 0, \quad \delta = 1$

$\sigma^{-1}(x) = \tau^{-1}(x) = 1$

$\Delta^0 = S_1$

1: **for** $i = 0 \rightarrow 2t - 1$ **do**
2: $\quad \sigma^i(x) = \delta \times \sigma^{i-1}(x) + \Delta^i x \tau^{i-1}(x)$
3: $\quad \Delta^{i+1} = S_{i+2}\sigma_0^i + S_{i+1}\sigma_1^i + \cdots + S_{i-v_i+2}\sigma_{v_i}^i$
4: $\quad$ **if** $\Delta^i = 0$ **or** $2D^{i-1} \geq i + 1$ **then**
5: $\quad\quad D^i = D^{i-1}, \quad \tau^i = x\tau^{i-1}(x)$
6: $\quad$ **else**
7: $\quad\quad D^i = i + 1 - D^{i-1}, \quad \delta = \Delta^i$
8: $\quad\quad \tau(x) = \sigma^{i-1}(x)$
9: $\quad$ **end if**
10: **end for**

In algorithm 2, $\sigma^i(x)$ is the ith step error locator polynomial with degree v_i, and the coefficients of $\sigma^i(x)$ are σ_j^i. Δ^i is the ith step discrepancy and δ is the previous discrepancy. $\tau^i(x)$ is an auxiliary polynomial and D^i is used to track the degree of the polynomial in the ith step.

From algorithm 2 we can see that there is no division, therefore no inversion is needed to calculate the error locator polynomial.

Having calculated the error locator polynomial, the error evaluator polynomial can be calculated using the key equation. Calculation of the error evaluator polynomial $\Omega(x)$ can be written as shown in equation 4.19 and 4.20 [14].

$$\begin{aligned}\Omega(x) &= S(x)\sigma(x)\, mod\, x^{2t} \\ &= (S_1 + S_2x + \cdots + S_{2t}^{2t-1}) \\ &\quad \times(\sigma_0 + \sigma_1 x + \cdots + \sigma_v x^v)\, mod\, x^{2t} \\ &= \Omega_0 + \Omega_1 x + \cdots + \Omega_{t-1}x^{t-1} \end{aligned} \tag{4.19}$$

$$\Omega_i = S_{i+1}\sigma_0 + \cdots + S_1\sigma_i\,, \qquad i = 0, 1, \cdots, t-1 \tag{4.20}$$

When using the inversionless Berlekamp-Massey algorithm, calculation of the error evaluator polynomial is performed after the error locator polynomial has been found. Since the two polynomials are calculated in sequence the latency of the inverionless Berlekamp-Massey algorithm is higher then an algorithm where the two polynomials are calculated in parallel.

4.4.3 Euclidean Algorithm

The Euclidean algorithm is a well known algorithm, which is used to find the greatest common divisor of two numbers [1]. For two numbers a and b the greatest common divider (GCD) is $r = GCD(a, b)$. This algorithm also calculates two coefficients, s and t, which is used to solve the *key equation*.

$$r = sa + tb \tag{4.21}$$

To easier see that the Euclidean algorithm can be used so solve the key equation, the key equation can be rewritten as shown in equation 4.22 [1].

$$\Lambda(x)S(x) - \mu(x)x^{2t} = -\Omega(x) \tag{4.22}$$

Where $\mu(x)$ is a polynomial that satisfy the key equation [1]. By applying the Euclidean algorithm to $S(x)$ and x^{2t}, which are the two known, we get GCD r, s and t. This can be written for the ith recursion as shown in equation 4.23 [1].

$$r_i(x) = s_i(x)x^{2t} + t_i(x)S(x) \tag{4.23}$$

By solving the key equation we want to obtain Ω and σ. When using the Euclidean Algorithm we get the two following solutions, shown in equation 4.24 and 4.25, after applying the algorithm.

$$\Omega(x) = -\sigma r_i(x) \tag{4.24}$$

$$\sigma(x) = \sigma t_i(x) \tag{4.25}$$

As the equations show we need to obtain $r_i(x)$ and $t_i(x)$. This is done by using the two following recursions [1]:

$$r_i(x) = r_{i-2}(x) + q(x)r_{i-1}(x) \tag{4.26}$$

$$t_i(x) = t_{i-2}(x) + q(x)t_{i-1}(x) \tag{4.27}$$

The value of r_i is the remainder when dividing r_{i-2} by r_{i-1}, and t_i is the remainder when dividing t_{i-2} by t_{i-1}. $q(x)$ is the quotient polynomial at the i division. The algorithm is initialized with the following parameters:

$$\begin{aligned} i &= -1 \\ r_{-1} &= x^{2t} \\ r_0 &= S(x) \\ t_{-1} &= 0 \\ t_0 &= 1 \end{aligned}$$

The recursion continuous as long as $deg(r_i(x)) \geq t$. When $deg(r_i(x)) < t$, the recursion stops and a constant λ is multiplied by equation 4.23 to get the resulting equations, previously shown in equation 4.24 and 4.25 [1]. The Chien search and Forney method can then be used to find the error locations, and the error values.

4.5 The Chien Search

After the error locator polynomial and error evaluator polynomial have been calculated, a method called the *Chien search* can be used to find the roots of the error locator polynomial [15]. The Chien search is a trial and error method, where each of the elements of the field $GF(2^m)$ are substituted into the error locator polynomial.

$$\sigma(x) = \sigma(\alpha^i), \quad for\ i = 0, 1, ..., n-1 \tag{4.28}$$

$$\sigma(\alpha^i) = 0 \tag{4.29}$$

If the condition in equation 4.29 is satisfied, the value is a root and the error location is then the inverse position of i, which is $(n-1)$. If $\sigma(\alpha^i) \neq 0$ there is no error at that position.

4.6 Forney Algorithm

After having found the error locations with the Chien search, Forney's algorithm can be used to find the error values [16]. The Forney algorithm uses the error evaluator polynomial and the error locator polynomial to find the error values. When using the Forney algorithm the error value Y_l at location X_l is given by equation 4.30.

$$Y_l = X_l^{1-b} \frac{\Omega(X_l^{-1})}{\sigma'(X_l^{-1})} \qquad l = 1, 2, ..., v \tag{4.30}$$

Where $\Omega(x)$ is the error evaluator polynomial, $\sigma'(X_l^{-1})$ is the derivative of $\sigma(X_l^{-1})$ and v is the number of errors. Equation 4.30, which is used to find the error value, only gives a valid result for symbol positions containing an error.

Chapter 5

Implementation

5.1 Symbol and Correction Size

Before implementing the Reed-Solomon encoder, the symbol bit size and correction capability of the Reed-Solomon code have to be chosen. The symbol bit size determines the amount of symbols in the Reed-Solomon code. Larger symbol bit size gives a larger Galois field size, which again increases the number of information bits the encoder can encode. The symbol size is given by m and the field size is given by $GF(2^m)$, where m is the number of bits in each symbol, as explained in section 2.3.

The correction capability of the Reed-Solomon code will also affect how much information it is possible to encode. By increasing the correction capability i.e. the parity symbols added, the number of information bits we can send will decrease. A trade-off between symbol size and correction capability should therefore be found.

It is desirable that the data packages that are going to be encoded are between 16 – 32 bytes (128 – 256 bits). In this thesis, the size of the data packages were chosen to be 16 bytes per package. Choosing to use 16-byte data package, give the possibility of exploring smaller Reed-Solomon codes.

Several different symbol size options and correction sizes were explored. Table 5.1 shows the different symbols and parity sizes that were evaluated. Choosing a symbol size of either 4 or 8-bit is probably the easiest solution. Both can divide the 16-byte package and give an integer. This makes it easier to divide the data package into symbols. Especially, when using 8-bit shift registers for shifting data in to the encoder and data out from the encoder.

Bit size	n	k	Parity	Correction capability	Correction (%)	Information bit	Transmissions needed
4	15	11	4	2	13.33	44	3
5	31	27	4	2	6.45	135	1
5	31	25	6	3	9.67	125	2
5	31	23	8	4	12.90	115	2
6	63	55	8	4	6.35	330	1
8	255	239	16	8	3.14	1912	1

Table 5.1: Different symbol and correction sizes

A 4-bit encoder would need to divide the 16-byte data package into 32 symbols. A Reed-Solomon code using a symbol size of 4-bit only encodes and sends a total of 15 symbols each time. Multiple transfers would therefore be needed to send the 16 bytes. How many transmissions that are needed, depends on the chosen correction capability. When using a symbol size of 8-bit, only 16 symbols would be needed for encoding the 16 bytes. A Reed-Solomon code using 8-bit have a total of 255 symbols. Choosing a 8-bit symbol size would give an inefficient system, as only 16 of the 255 symbols would contain useful information.

Both a 5-bit system and a 6-bit system were evaluated, as shown in table 5.1. The 6-bit system was quickly found unsuitable, as it transfers over twice as much data as needed. The 6-bit system would be using time to encode symbols that is considered useless. To save time as smaller system should be chosen.

Three different 5-bit systems were considered, where the symbol correction size was two, three and four. The first 5-bit system can correct up to two symbol errors, and has the capability of sending up to 135-bit of information in one transmission. The 16-byte data package could be transmitted after only one encoding cycle, which is desirable. The down side is that it only has a correction capability of 6.45%. The two other 5-bit system have a larger correction capability. The 5-bit systems with the possibility of correction up to three and four symbol errors, need to transmit two times to send the 16-byte data package. Since both systems need to transfer two times, the 5-bit system that can correct four errors would be the better option between the two. The reason for this is the increased error correction capability.

It is not known what kind of environment the system is going to be used in. Choosing the right correction capability is therefore hard. For Reed-Solomon systems used to transfer data where there is a lot of noise, it is obvious that a larger correction capability is important if not wanting to retransmit. But lager correction capability equals less “information” in each transmission. Sending the whole data packet in one transmission could be very desirable. In this thesis the smallest 5-bit system with the possibility of correcting two symbol errors have been chosen. This makes it possible to send 135 bits of information in one transmission, thus the whole 16 bytes of data.

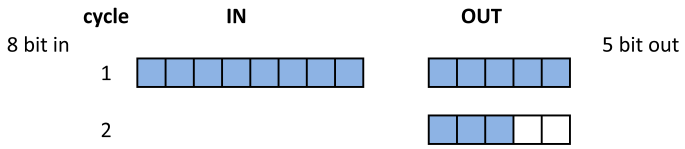

Figure 5.1: 8-bit to 5-bit conversion

Microcontrollers often use 8-bit shift registers to shift the data. It is therefore desirable to look at how these 8-bit can be converted to a data package of 5-bit. The shifting of a 8-bit data package to 5-bit data package can be done as shown in figure 5.1. As shown in the figure, the first five bits are put in the first output. The remaining three bits are put in the second output. The last two places in the second output is filled with bit values from the second input. This bit converting process becomes complicated when 16 bytes is to be shifted into the encoder, and a fair amount of hardware is needed. In this thesis a module converting a 8-bit package to a 5-bit package has been implemented. This module was not a main focus of the thesis, and it has therefore not been optimized. Because no optimizations were performed, the in shifting module consumed a large area. The module was therefore not used when synthesis and power estimations were performed on the Reed-Solomon encoder and decoder.

In this thesis $p(x) = x^5 + x^2 + 1$ is chosen as the primitive polynomial for $GF(2^5)$. The field elements for $GF(2^5)$ with $p(x) = x^5 + x^2 + 1$ can be seen in appendix A.1.2.

5.2 Implementing Arithmetic Operations

When designing the Reed-Solomon system, different arithmetic operations need to be implemented. The Galois field arithmetic operations were described in section 2.4.1 and 2.4.2. Techniques for implementing these operations will be described in the next subsections.

5.2.1 Addition

Galois field addition, which is an addition modulo-2 operation, can be implemented using a two input XOR gate. To implement a m bit addition module, we need m such XOR gates.

5.2.2 Galois Field Multipliers

Multiplication in Galois field is a more complicated process than the addition process, as mentioned in section 2.4.2. The multipliers can be implemented using

constant multipliers, memory multipliers and full multipliers.

Constant Multipliers
Multipliers with one variable input and one fixed input is called a constant multiplier. A constant multiplier can be implemented using a bit-parallel full multiplier, where one of the inputs are fixed. This is not necessarily the optimal solution, as a full multiplier used as a constant multiplier will have redundant circuitry. The representation with the least Hamming weight, i.e. the representation with the fewest non-zero values, is the minimal representation the constant multiplier can have [17]. By finding the minimal representation, a dedicated logic constant multiplier can be implemented. This reduces the implementation complexity of the multiplier.

A constant multiplier can be constructed using the general polynomial representation of $a = a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}$, and then multiplying by the constant value. When multiplying with the constant value, a shifted version for each non-zero coefficient is produced in the columns α^m to α^{2m-2}. These shifted values are substituted with a m-bit equivalent from the field $GF(2^m)$. The values in the columns α^0 to α^{m-1} are then added to give the minimal representation from the input to the output. An example on how a 5-bit constant multiplier is constructed, is shown in appendix B.1.1.

Memory Multipliers
Multiplication in Galois field can also be implemented using a look-up table. This look-up table is implemented using read only memory (ROM) . When multiplying a value by a constant in a small Reed-Solomon design, this type of implementation method can be used. For a bigger design with 8 bit, the number of entries for each multiplier would be $2^8 = 256$. If a full multiplier is implemented as a look-up table, the number of entries will be $2^{2\times 8}$. From this we can draw the conclusion, that implementing multipliers as look-up tables are very inefficient when the bit size increases.

Full Multipliers
Full multipliers can be implemented by using polynomial multiplication. In polynomial multiplication the finite field elements are represented using the polynomial basis. This multiplication approach can be written as a multiplication modulo $p(x)$, as shown in equation 5.1.

$$c(x) = a(x)b(x)\ mod\, p(x) \tag{5.1}$$

Where $a(x)$ and $b(x)$ are two field elements, $p(x)$ is a degree m irreducible polynomial over $GF(2^m)$ and $c(x)$ is the product.

The polynomial basis multiplication, shown in equation 5.1, is performed in two steps [18]. First polynomial multiplication is performed, then a reduction modulo an irreducible polynomial (field generator polynomial) is performed. The polynomial multiplication is written as shown in equation 5.2, where $d(x)$ is the product

of the multiplication.

$$d(x) = a(x)b(x) \tag{5.2}$$

The polynomial $d(x)$ has a maximum degree of $2m-2$, and the coefficients of $d(x)$ are determined by the expression in equation 5.3 [18]. Equation 5.3 is a bit-parallel computation, and have a gate complexity of m^2 AND gates and $(m-1)^2$ XOR gates.

$$d_k = \begin{cases} \sum_{i=0}^{k} a_i b_{k-i}, & k = 0, \ldots, m-1 \\ \sum_{i=k}^{2m-2} a_{k-i+(m-1)} b_{i-(m-1)}, & k = m, \ldots, 2m-2 \end{cases} \tag{5.3}$$

When $d(x) = a(x)b(x)$ has been computed, a reduction modulo an irreducible polynomial $p(x)$ is performed. Reducing the polynomial product $d(x)$ of degree $2m-2$ by a degree m polynomial $p(x)$, gives a resulting polynomial with degree $deg(c(x) \leq m-1)$. The reduction involves mapping the coefficients $d_m - d_{2m-2}$ of $d(x)$ into the coefficients $c_0 - c_{m-1}$ of $c(x)$ by using $p(x)$. An example on how a 5-bit full multiplier is constructed can be seen in appendix B.1.2.

5.2.3 Division

The easiest way to perform division in finite field is to multiply with the inverse of the divisor, as described in section 2.4.2. The inverse field values can be stored in a look-up table with 2^m entries to represent all the inverses field elements. To complete the division process, the inverse value is multiplied with the dividend using a full multiplier.

5.3 Implementation Techniques

To find a suitable implementation with the lowest energy consumption for both the Reed-Solomon encoder and decoder, different implementation techniques have been tested.

The Reed-Solomon encoder implementation chosen in this thesis has very low complexity. Clock gating was therefore the only implementation technique directly used to reduce the power consumption, and thereby reducing the energy consumption.

The different implementation techniques used to possibly reduce the energy consumption in the decoder are:

- Reduce decoding time by using pipelined modules

- Compute only half of the syndrome values
- Two-parallel syndrome cells to reduce decoding time
- Reduce number of calculations in Forney module
- Clock gating implementation of the modules

How these techniques have been implemented are described in the next subsections. The different decoder configurations used for synthesis are described in section 7.3 and 7.4.

5.4 Implementation of Encoder

The Reed-Solomon encoder can be implemented using different techniques. One of the most common ways of implementing the encoder, is by using a pipelined bit-serial architecture. This implementation can be achieved by using a linear feedback shift register (LFSR) circuit, [5], [7], [19]. The Reed-Solomon encoder using LFSR is illustrated in figure 5.2. The architecture consist of 2t Galois multipliers, 2t Galois adders, 2t registers, a multiplexor and an AND-gate to prevent further feedback into the encoder, when all the m-bit symbols have been shifted in. The registers are used to store the remainder polynomial at each clock cycle. In this thesis the Reed-Solomon encoder is implemented using a LFSR circuit.

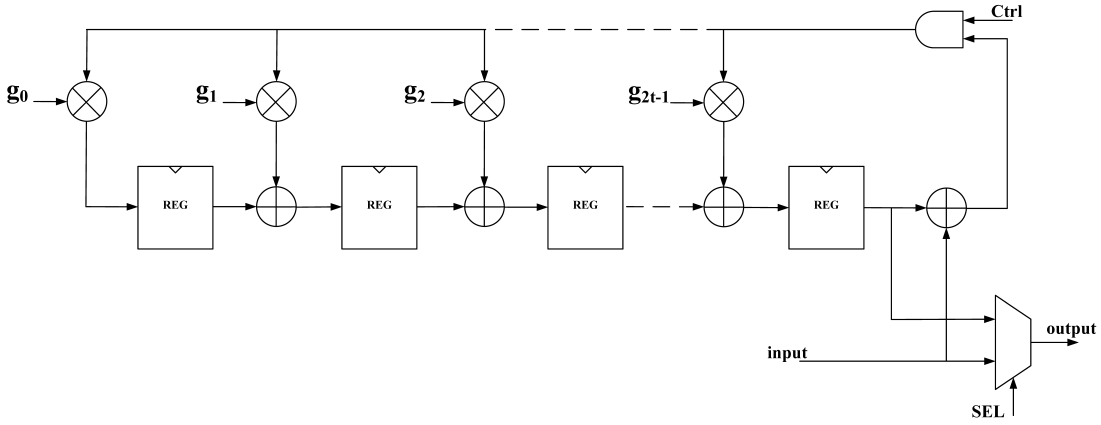

Figure 5.2: Reed-Solomon encoder architecture

The encoding process using the LFSR based design can be described in the following way:

First the message symbols are shifted from position $n-k$ down to 0, to position $n-1$ down to $n-k$. The $n-k-1$ positions are then filled with zeroes. This

allows the parity symbols to be placed in the $n-k-1$ positions. Then the message symbols are shifted symbol by symbol into the encoder. Each message symbol is multiplied with the generator polynomial using Galois field multiplication. The results of these multiplication are then added to the previous results, which are stored in the registers in figure 5.2. During this time, the output control enable only the input data to be shifted through. When all the symbols have been shifted through the encoder, the registers shown in figure 5.2 will hold the remainders. These remainders are the parity symbols, which will be added to the original message polynomial before being transmitted. To shift out the remainders stored in the registers, the control signal connected to the AND gate is set to zero. This allows input values of only zeroes to be passed into the encoder. In this way the parity symbols will be shifted to the output without altering their values. It takes $k+1$ clock cycles to shift through all the message symbols, and $n+1$ clock cycles to output the whole codeword. After the $n+1$th clock cycle the encoder can start encoding new data for transmission [20].

To reduce the hardware needed for the implementation, a fixed-rate encoder can be used. A fixed-rate encoder has a predetermined generator polynomial, which allows the encoder architecture to be implemented using constant multipliers. This reduces the hardware complexity of the multipliers, thus reducing the overall area of the encoder. In this thesis the code generator polynomial shown in equation 5.4 has been used. Since the Reed-Solomon system implemented in this thesis can correct up to two symbol errors, four consecutive elements of the field as roots are required. To reduce a small part of the hardware implementation the roots starting from α^b where $b = 0$ have been chosen. By choosing the roots starting from α^0 the syndrome module and Chien search module in the decoder will have a reduction in complexity since $\alpha^0 = 1$.

$$\begin{aligned} g(x) &= \alpha^0 x^4 + \alpha^{23} x^3 + \alpha^{17} x^2 + \alpha^{26} x + \alpha^6 \\ &= x^4 + 15x^3 + 19x^2 + 23x + 10 \end{aligned} \tag{5.4}$$

5.5 Implementation of Decoder

The Reed-Solomon decoder implemented in this thesis uses a syndrome based architecture. The syndrome based architecture calculates $2t$ syndrome values to detect if errors have occurred in the codeword. Reed-Solomon with syndrome based decoding was described in chapter 4. The syndrome calculation, key equation solver, Chien search, Forney method and error correction unit can be implemented in different ways. An overview of the different implementations done in this thesis can be seen in figure 5.3. In the next subsection the methods for implementing the Reed-Solomon modules are described.

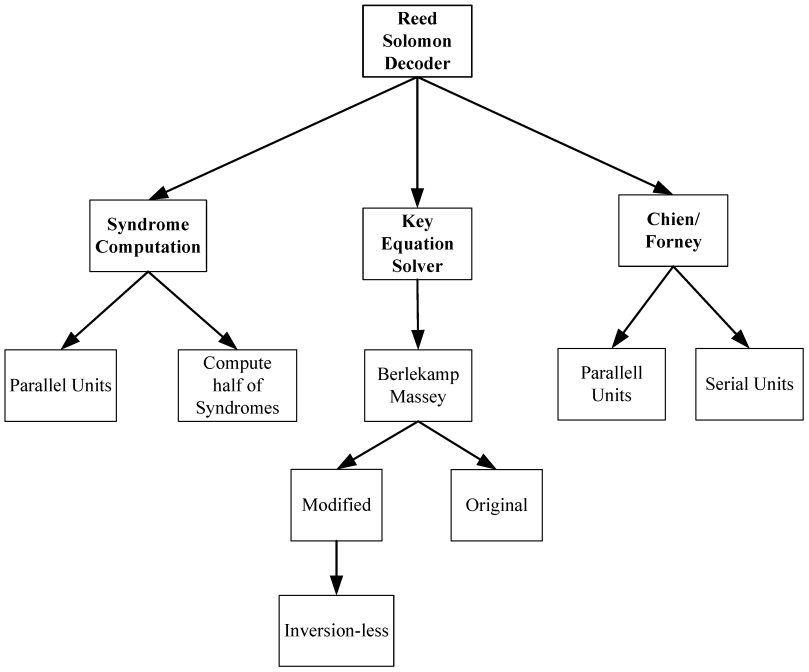

Figure 5.3: Implemented Reed-Solomon decoding techniques

5.5.1 Implementation of Syndrome Calculation

Implementation of the syndrome calculation module is done using equation 4.7 as a reference. The received symbols are multiplied by the roots of the generator polynomial, α^i, and then the terms are added together. The implementation is illustrated in figure 5.4.

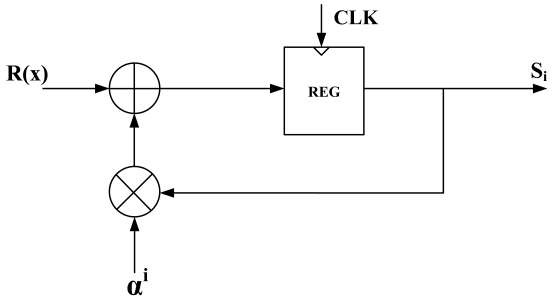

Figure 5.4: Syndrome cell

The syndrome cell, shown in figure 5.4, consist of a adder, a constant multiplier,

which multiply by α^i, and a register which holds the previous result. Each root of the generator polynomial, α^i, has to be substituted into the received polynomial. This requires the syndrome module to consist of $2t$ syndrome cells to be able to calculate all the syndrome values.

The each symbol of the codeword is serially input into the syndrome module, and it takes n clock cycles to calculate all the syndrome values.

By implementing a parallel syndrome architecture, the number of clock cycles needed to calculate the syndrome values can be reduced to $\frac{n}{2}$. The parallel architecture can be achieved by using two-parallel syndromes cells [21]. A two-parallel syndrome cell is shown in figure 5.5. When using the two-parallel syndrome cell, two symbols are used as input at the time. One input is for the odd symbols, and one is for the even symbols. As seen from the input sequence in figure 5.5, the input containing the odd symbols are delayed one clock cycle. This enables us to calculate $r_{n-1}(\alpha^i)^2 + r_{n-2}\alpha^i + r_{n-3}$ in the same clock cycle.

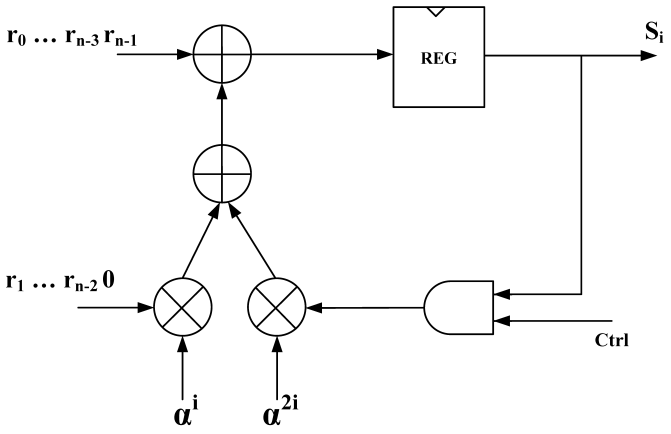

Figure 5.5: Two-parallel syndrome cell

In [22] a technique for reducing the power consumption of the syndrome computations is suggested. The idea is to reduce the number of syndrome calculations needed to determine if the received codeword has errors or not. The standard approach is that a codeword is error free if all the $2t$ syndromes are equal to zero. By using the first t syndrome values as error detectors, we can reduce the number of calculations needed to determine if a codeword is error free. If the first continuous t symbols equal zero, all $2t$ syndromes will equal to zero. This implies that if t syndromes are zero, all error values are zero or the decoder is out of correction capability ($v_{errors} > t$). Either way the decoding procedure can be stopped. If not t continuous syndromes are zero, the decoder will continue to calculate the rest of

the $2t$ syndromes. For applications with low probability of errors occurring, this technique could reduce the power consumption.

5.5.2 Implementation of Key Equation Solver

In this thesis, three different techniques for finding the error locator polynomial and error evaluator polynomial have been described. These algorithms essentially solve the key equation, which was presented in section 4.4. The direct method described in section 4.4, can be very efficient way of solving the matrix shown in 4.16 for a small number of errors. As the number of errors increase this algorithm becomes more and more complicated, thus inefficient because we have to do an exhaustive search to find the number of errors.

The Euclidean algorithm described in section 4.4.3, can also be used to solve the key equation. The algorithm uses Euclid's GCD method to solve key equation, thus obtaining the error-location polynomial and error evaluation polynomial. The Euclidean algorithm is a parallel algorithm, which means that it computes the error locator and error evaluator polynomial in parallel.

In this thesis the Berlekamp-Massey algorithm is used to solve the key equation. The Berlekamp-Massey algorithm was described in section 4.4.1. The algorithm has been chosen because of its low hardware complexity, and the possibility of reusing some of the the logic elements. The hardware used to calculate the discrepancy can for instance be reused to calculate the error evaluator polynomial. An inversionless Berlekamp-Massey algorithm [14] is described in section 4.4.2. This modified Berlekamp-Massey algorithm allows for the ith step error locator polynomial to be calculated in parallel with the *i+1* discrepancy. The algorithm also removes the need for inversion when computing the error locator polynomial.

By decomposing the inversionless Berlekamp-Massey algorithm described in section 4.4.2 we can easier see how the algorithm can be implemented [14].

$$\sigma_j^{(i)} = \begin{cases} \delta \times \sigma_0^{(i-1)}, & \text{for } j = 0 \\ \delta \times \sigma_j^{(i-1)} + \Delta^{(i)} \tau_{j-1}^{(i-1)}, & \text{for } 1 \leq j \leq v_i \end{cases} \tag{5.5}$$

$$\Delta_j^{(i+1)} = \begin{cases} 0, & \text{for } j = 0 \\ \Delta_{j-1}^{(i+1)} S_{i-j+3} \times \sigma_{j-1}^{(i)}, & \text{for } 1 \leq j \leq v_i \end{cases} \tag{5.6}$$

In equation 5.5 and 5.6, $\sigma_j^{(i)}$ is the jth coefficient for the ith step error locator polynomial, $\tau_j^{(i)}$'s are the coefficients of of $\tau^{(i)}(x)$ and $\Delta_j^{(i+1)}$'s are the partial results when computing $\Delta^{(i+1)}$. As seen from the equation, computation of $\sigma_j^{(i)}$ requires the discrepancy computed at cycle zero and the jth error locator coefficient computed at the previous step. Computation of $\Delta_j^{(i+1)}$ requires the error

locator coefficient $\sigma_{j-1}^{(i)}$ and the partial discrepancy result $\Delta_{j-1}^{(i+1)}$, which is both computed at previous cycle.

Similar, the computation of the error evaluator polynomial coefficients Ω_i in equation 4.20 can be computed as shown in equation 5.7.

$$\Omega_i^{(j)} = \begin{cases} S_{i+1}\sigma_0, & \text{for } j = 0 \\ \Omega_i^{(j-1)} + S_{i-j-1}\sigma_j, & \text{for } 1 \leq j \leq i \end{cases} \tag{5.7}$$

By looking at the decompositions in equation 5.5 and 5.6, we can see that two finite field multipliers and one finite field adder are needed to calculate the ith step error locator polynomial. One finite field multiplier and one finite field adder are also needed to calculate each partial result discrepancy. The architecture for implementing the inversionless Berlekamp-Massey algorithm [14] can be seen in figure 5.6. As seen from the architecture three finite field multipliers and two finite field adders are required to calculate the error locator polynomial and the discrepancy.

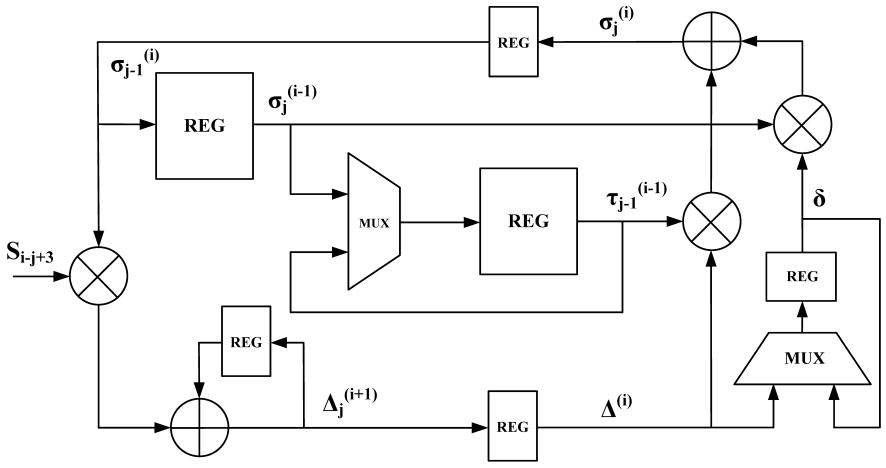

Figure 5.6: Inversionless Berlekamp-Massey algorithm architecture

The decomposed error evaluator polynomial in equation 5.7 can also be implemented with a similar architecture as the one shown in figure 5.6. Figure 5.7 shows the architecture for computing the error evaluator polynomial. The architecture needs one finite field multiplier and one finite field adder if implemented using equation 5.7.

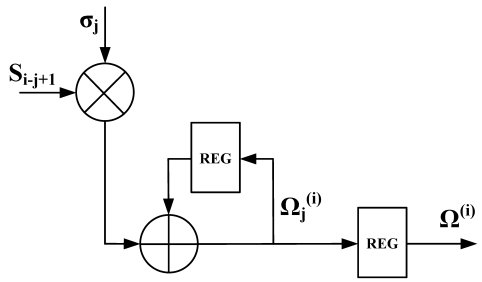

Figure 5.7: Architecture for computing the error evaluator polynomial

The finite field multipliers in figure 5.6 and 5.7 are full multipliers, which was described in section 5.2.2. The adders in figure 5.6 and 5.7 are implemented using five 2-input XOR gates. The inversionless Berlekamp-Massey algorithm was implemented as a finite state machine.

5.5.3 Implementation of Chien Search

Chien search is used to find the location of the errors using the error locator polynomial $\sigma(x)$. The error locations are found by substituting the roots α into the error locator polynomial, and then adding all the terms together. If the sum is equal to zero, an error position has been found. If the sum do not equal zero, there is no error on that position. To check all the positions for a possible error, the Chien search module has to compute all the values from $\sigma(\alpha^0)$ to $\sigma(\alpha^{n-1})$. This takes n clock cycles when the field size is $n = GF(2^m)$. The Chien search cell for substituting α^i into σ_i is shown in figure 5.8. Since σ_0 only is a constant, we need t such Chien search cells to compute $\sigma(\alpha^i)$.

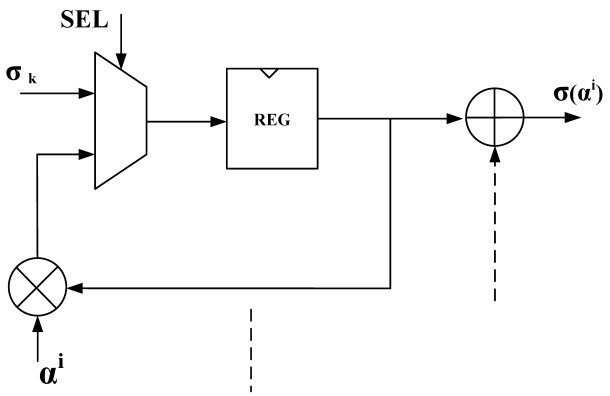

Figure 5.8: Chien search cell

The Forney algorithm needs both the derivative of the error locator polynomial $\sigma'(\alpha^i)$ and the $\Omega(\alpha^i)$ values. Both these values can be computed in the Chien search block. By adding together all the odd terms $\sigma_{odd}(\alpha^i)$ we get the derivative $\sigma'(\alpha^i)$. $\Omega(\alpha^i)$ can be calculated using the same Chien search cell used to calculate $\sigma(\alpha^i)$. The only difference is that the coefficient of the error evaluator polynomial are used as input. $t-1$ Chien search cells are needed to calculate $\Omega(\alpha^i)$.

5.5.4 Implementation of Forney Algorithm

The Forney algorithm calculates the error values by using the error positions found with Chien search. The error values are found using equation 4.30 and then substituting the location X_j into the equation. For a given location a^i this gives:

$$e_l = \alpha^{-i} \frac{\Omega(\alpha^{-i})}{\sigma'(\alpha^{-i})} \tag{5.8}$$

From equation 5.8 we can see that finite field division is needed to find the error value. This can be achieved by multiplying with the inverse value of the divisor, $\sigma'(\alpha^{-i})$. The inverse values are found by implementing a ROM, which stores all the inverse field values. Figure 5.9 shows how the Forney algorithm can be implemented [19]. $\Omega(\alpha^i)$, $\sigma'(\alpha^{-i})$ and $\sigma(\alpha^i)$ are all calculated in the Chien search module.

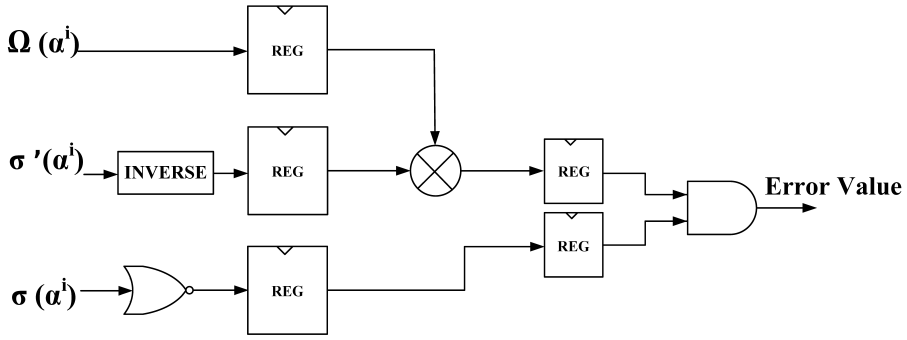

Figure 5.9: Forney module

The multiplier shown in figure 5.9 is a full multiplier, and is implemented as described section 5.2.2. The first set of registers are used to store the values while the inverse of $\sigma'(\alpha^{-i})$ is found.

5.5.5 Error Correction

When the error positions and error values have been found using the Chien search and Forney algorithm, the error correction module corrects the errors. This is done by adding the error values to the received codeword. The error correction module reads the received codeword, which is temporarily stored in the decoder, and then adds the error values to the codeword using XOR gates.

5.5.6 Decoder Architecture

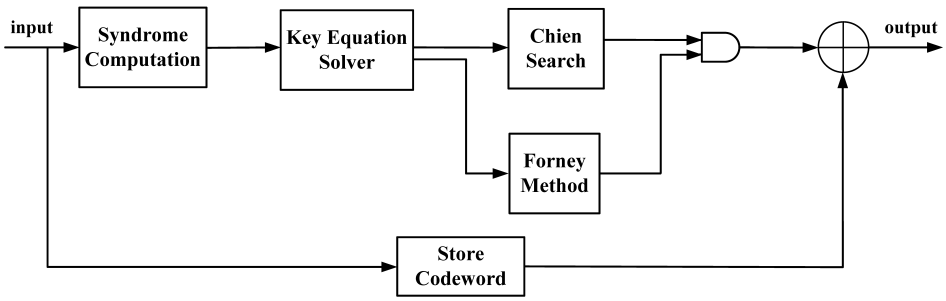

Figure 5.10: Reed-Solomon pipelined decoding scheme

In section 5.5.1 through 5.5.5 implementation techniques for the Reed-Solomon decoder have been presented. In this thesis two decoder architectures are implemented using different modules. The first architecture is the pipelined architecture,

which is shown in figure 5.10. In this architecture the three last stages of the decoding process are connected in a pipelining stage. This means that the Forney method calculates the error value right after the first symbol has been checked for errors. In this architecture all the positions in the codeword are checked for possible errors. This could lead to many unnecessary calculation in the Foreny module, since not all symbols will have a valid error value. On the other hand, this architecture has a low latency, because the Chien search, Forney and error correction module are pipelined. A decoding diagram of the pipelined architecture can be seen in figure 5.11.

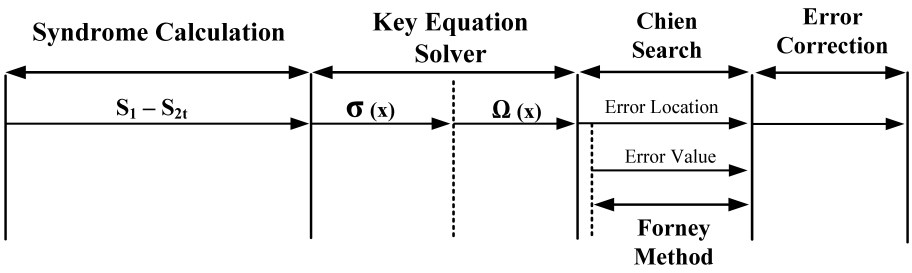

Figure 5.11: Reed-Solomon pipelined decoding diagram

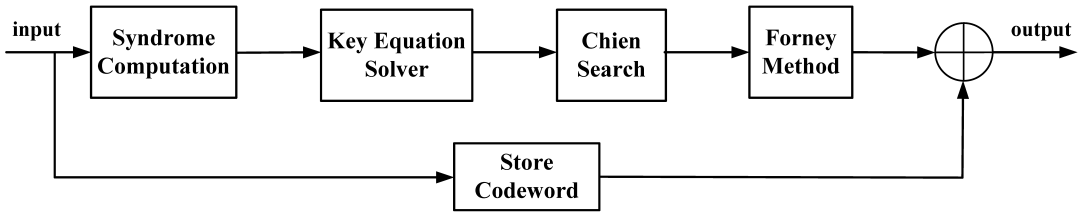

Figure 5.12: Reed-Solomon full serial decoding scheme

The second decoder architecture shown in figure 5.12, is a full serial architecture. This means that the Forney module waits for the Chien search module to finish all its calculations. When the Chien search is finished, the Forney module starts performing its calculations, i.e. no pipelined modules. If all the error positions are first found, only the error values to the know positions have to be calculated. This reduces the number of calculations performed in the Forney module. The latency of this architecture, however, is larger then the pipelined architecture, because no errors are corrected until all the positions in the codeword have been checked for errors. The decoding diagram for the full serial architecture can be seen in figure 5.13.

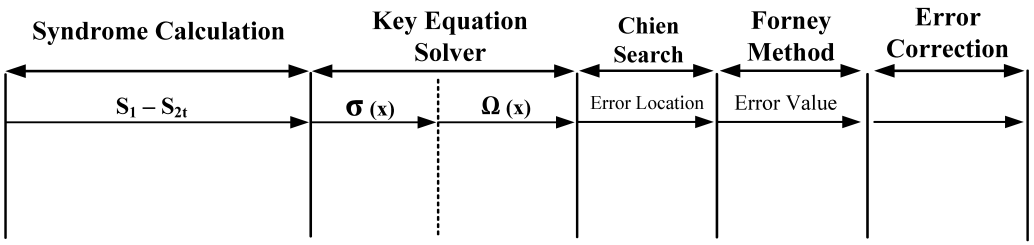

Figure 5.13: Reed-Solomon full serial decoding diagram

A more detailed block diagram of the general decoder architecture can be seen in figure 5.14. The top level decoder consist of several modules. The first module in the decoder is the input shift module. This module shifts in the codeword five bits at the time. Each symbol is sent to the memory module and the syndrome calculation module, using the IN_STAGE_OUT bus shown in figure 5.14. The symbols are stored in separate addresses in the codeword memory module while the decoder is decoding. Once the first symbol of the codeword has been shifted into the decoder, the I_START signal is set high and the syndrome module starts to calculate the syndrome values. When all the symbols have been shifted in and all the syndrome values are calculated, the syndrome module check if a error has occurred. This is done by checking if all the syndrome values are zero. If all the syndromes equal zero the NO_ERROR flag goes high and the decoding process stops. If not, the S_READY flag goes high, indicating that the error locator polynomial can be calculated using the Berlekamp-Massey algorithm.

The syndrome values are sent to the Berlekamp-Massey module using the bus signals *S1* - *S4*. In the Berlekamp-Massey module the error locator polynomial is calculated. When the error locator polynomial has been calculated, the L_READY flag goes high. The error evaluator polynomial is then calculated using the syndrome values and the error locator polynomial. The Omega module reads the coefficients of the error locator polynomial using the bus signals *L0* - *L2*. Once the error evaluator polynomial is calculated, the LW_READY flag is set high. The Chien search will then use the two polynomials found by solving the key equation to find the error positions. The Omega module transfers the coefficients of the error evaluator polynomial to the Chien search module with the buses *W0* and *W1*.

Depending on the architecture, the EP_READY flag will go high either after the first position in the codeword has been check for error (pipelined architecture), or after all the positions have been checked (full serial architecture). When EP_READY is set high, the Forney module calculates the error values. The Forney module reads the error positions using the $ERROR\_POS$ bus, the derivative of the error locator polynomial using the L_DER bus and the $\Omega(\alpha)$ value using the W_TOT bus. The EV_READY signal indicates that the error values are

ready. When both the EP_READY and EV_READY signals are high, the error correction module reads the codeword symbols from the memory module, and corrects the symbols that have an error. The decoder starts shifting out the corrected symbols using the $DATA\_OUT$ bus when the RS_DONE signal is set high.

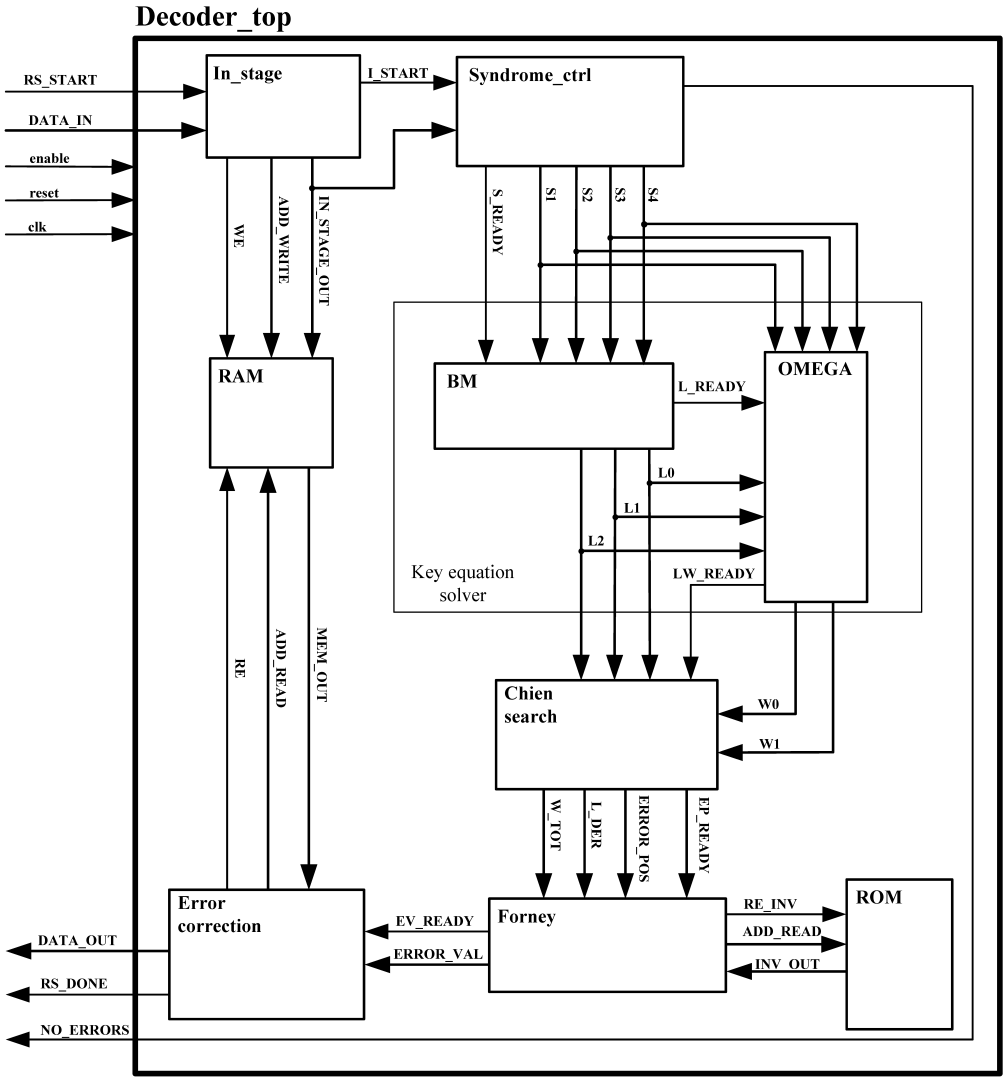

Figure 5.14: Block diagram decoder

Chapter 6

Verification and Test

The design and simulation of the Reed-Solomon encoder and decoder were both done using *Aldec Active-HDL 9.2 Education Edition*. The HDL implementation was written using Verilog.

To verify and test the Reed-Solomon encoder and decoder, each sub-module of the designs were first tested separately, and then connected together and tested as a system. The test vectors used for verification were generated using the *Matlab R2012b* functions RSEncoder and RSDecoder in the communication toolbox.

6.1 Verification and Test of Encoder

The encoder system using LFSRs has four constant multipliers. These multipliers were first tested separately, before being connected into the encoder system. This was done using different values as input to the multipliers and then observing the result at the output. Since the Reed-Solomon design is a 5-bit system, only 32 possible values ($2^5 = 32$), ranging from 0 to 31, can be used as input. The results from the multiplication were verified using the look-up table provided by [23]. This type of testing is not the most efficient way of doing verification, but for such a small number of values it can easily be done without writing a large test bench.

To verify the encoder system with all sub-modules, a test bench was written using Active-HDL. This test bench reads data from a text file, and then inputs it into the Reed-Solomon encoder for encoding. The encoded values, i.e. the original input data and parity symbols, are then compared to encoded values made using Matlab. If the encoded data is equal to the data encoded using the RSEncoder function in Matlab, the encoding is presumed to be correct.

To know that the encoder works for all possible input messages, the ideal solution would be to test all possible values, which is $27^{32} = 6,36 \times 10^{45}$ (27 symbols of information, each symbol can have a value between 0 and 31). This would take a lot of time, and is not a very efficient way of testing the system.

To verify the system, the corner cases and some other special cases were tested as input to the encoder. The preliminary test cases were used to check how the encoder handles different input values. The test cases used are described below.

1. All symbol values are 31
2. All symbol values are zero
3. All symbol values are one
4. Increasing input values
5. Decreasing input values
6. Rapidly changing bit values
7. Different bit patterns

The test vectors used in these tests can be seen in table 6.1. 2000 randomly generated vectors were also tested, to verify the functionality of the encoder. These test vectors were randomly generated using a Matlab script.

All test vectors used for synthesis purposes were generated using the Matlab script shown in appendix D.1.1. The script first randomly generates 16 bytes of data, and then pad the remaining 7 bits with zeros, since only 16 bytes of information is to be encoded each time. This gives a total of 135 bits, which is the total number of bits that can be encoded with the 5-bit Reed-Solomon system. The randomly generated data are then encoded using the RSEncoder function. Both the generated data message and the codeword are written to separate text files. These text files are uses as input to the test bench.

An example of some of the test vectors used is shown in appendix C.1.1.

Case	Input symbols	Parity symbols
1	31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31	6 21 18 30
2	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0
3	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	16 20 31 26
4	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27	24 2 8 18
5	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	3 22 1 16
6	21 10 21 10 21 10 21 10 21 10 21 10 21 10 21 10 21 10 21 10 21 10 21 10 21	2 19 10 17
7	31 0 31 0 31 0 31 0 31 0 31 0 31 0 31 0 31 0 31 0 31 0 31 0 31	19 15 18 14
	27 4 27 4 27 4 27 4 27 4 27 4 27 4 27 4 27 4 27 4 27 4 27 4 27	25 21 1 9
	31 31 31 0 0 0 31 31 31 0 0 0 31 31 31 0 0 0 31 31 31 0 0 0 31 31 31	5 11 20 5

Table 6.1: Test vectors used to verify encoder functionality

6.2 Verification and Test of Decoder

The decoder consists of the five main modules, i.e. syndrome calculation, key equation solver, Chien search, Forney algorithm and error correction unit. It also has a input stage, a memory module to store the symbols while decoding and a ROM module, which holds the inverse field values. The functionality of the

decoder was verified in several steps before the design was synthesized. Five test steps were used during the verification process. These five steps are listed below:

1. Individual module test
2. Decoding of codewords with no errors
3. Decoding of codewords with one error
4. Decoding of codewords with two errors
5. Decoding of several codewords in sequence

In the first test a hand calculated example was used to verify each module individually. When each module gave the expected result, all the modules were connected and the same test case again applied. This was done to correct any timing issues that occurred when connecting all the modules in the decoder. The hand calculated example is shown in appendix B.2.

In the second step of the decoder verification, codewords containing zero errors were tested. This was done to verify that the syndrome module can detect a codeword with no errors, and notify this. The codewords used in this test were the nine encoded messages shown in table 6.1.

The third step in the decoder verification was performed using encoded messages with added noise. First codewords with one error were tested. All possible one error combinations, which are $n \times n$ combinations, were added to an encoded message and then decoded. To test all possible codeword combinations with one error would not be very efficient, therefore the nine test vectors from table 6.1 were used. These test cases were presented in section 6.1. The total number of test vectors for the nine test cases were 9×961.

To test the decoder on codewords with two errors, the same encoded messages used to test codewords with one error were used. To reduce the number of test vectors, the two errors had the same value. For a codeword with the same two error values there are 465 combinations, and for all error values the number of combinations are $465 \times n$ for each codeword. With the nine codewords used in this test, the total number of test vectors are $9 \times 465 \times n$. The Matlab script shown in appendix D.1.2, with some modifications, was used to generate the test vectors. Testing all possible two value combinations would give a total of $465 \times n^2$ test vectors for each codeword. This would give a large amount of test vectors, if all codeword combinations should have been tested. However, by choosing good corner cases, one can assume that the system works correct if all the selected test vectors pass.

The last step in the decoder verification was performed using randomly generated codewords as input to the decoder. This test was done to verify that the decoder could handle decoding of several codewords in sequence. The codewords were generated using a Matlab script and the RSDecoder function. The script can be seen in appendix D.1.2.

The Matlab script first generates 27 random values between 0 and 31, and then encodes these values. The encoded values are then added together with random values (noise). The noise is randomly added to zero symbols, one symbol or two symbols in the encoded codeword. The noisy codeword is then decoded using the decoder function. The noisy codeword was used as input to the decoder, and the Matlab decoded codeword was used as a comparison to the values from the decoder system.

2000 random test vectors were generated and tested for the last verification step. Each of these test vectors either had zero errors, one error or two errors. After testing all these vectors, the functionality of decoder was assumed to be correct. To verify the functionality of the all the decoder designs, the same procedure as described above was used.

Chapter 7

Synthesis

The Reed-Solomon encoder and decoder were both synthesized using Synopsys synthesize tools and a 45nm CMOS cell-library.

7.1 FreePDK 45nm CMOS Technology Library

The technology library used for ASIC synthesis is the 45nm CMOS technology library found in the FreePDK package [24]. This library is developed by Oklahoma State University, and is freely distributed for use in research projects. Information about FreePDK can be found in [25]. Ideally for this thesis 90nm or 180nm CMOS-technology should have been used, as this is the technology used by Energy Micro in their microcontroller development. Cell-libraries are often very expensive and license restricted. The FreePDK 45nm technology library was chosen for synthesis in this thesis, because it was the only one available.

The cell-library uses a power supply voltage of 1,1 V.

7.2 Synopsys Synthesis Tools

In this thesis three Synopsys synthesis tools were used: *Synplify Pro*, *Design Compiler* and *Power Compiler*. Design Compiler and Power Compiler are accessed by prompting the *dc_shell* command in a terminal window. The *Design Vision* tool by Synopsys has also been used in some extent in this thesis. Design Vision is the graphical user interface for the Design Compiler synthesis tools. Design Vision makes it possible to graphically view the synthesized design at both generic technology (GTECH) level and gate level. It also lets the user perform Design Compiler commands using a graphical user interface.

7.2.1 Synplify Pro

Synplify Pro is an FPGA synthesis software used for synthesizing RTL code into FPGA logic. Synplify Pro was used at an early stage in the design process to check for design violations, and to detect unwanted latches. The FPGA synthesis also determines the speed and area usage of the design. These results are however not relevant for comparison to an ASIC implementation, and are therefore not presented in this thesis. Xilinx Virtex-IV was used as target FPGA when synthesizing with Synplify Pro.

7.2.2 Design Compiler and Power Compiler

Design Compiler is a synthesis tool, which compiles and converts a design written in a hardware description language (HDL), into a gate-level netlist. This netlist is then mapped to a technology library specified by the user.

The Design Compiler tool can be set up using the *.synopsys_dc.setup* file. In this file the target library, symbol library, synthetic library and file source path are defined. The setup file used in this thesis can be seen in appendix D.2.1.

When performing synthesis with Design Compiler, the area of the design is measured in absolute area. In this thesis the area results from synthesis are also presented using NAND2 gate count. In the FreePDK 45nm technology library file the NAND2 area was specified to be 1,887200.

The Power Compiler tool is accessed through Design Compiler. It is used for analyzing and minimizing power consumption at RTL and gate-level. Power Compiler makes it possible to perform clock gating to reduce dynamic power consumption and leakage optimization, to reduce standby power.

To be able to report the power consumption, Power Compiler analyzes and propagates switching activity through the synthesized design. The switching activity is obtained by simulating the design and generating a value change dump (.vcd). The VCD files can be made using the following commands when simulating in Active-HDL.

1. vcd file <*file_name*>.vcd
2. vcd add –r <*tb_design*>/<*module_instance*>/*
3. run

The VCD files are then converted to a SAIF file using the vcd2saif command in Design Compiler. Switching Activity Interchange Format (SAIF) is a format used by Power Compiler to calculate the power consumption of the design. The work flow of how this is done can be seen in figure 7.1.

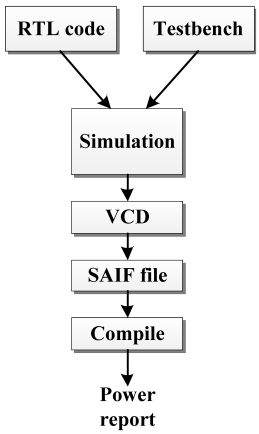

Figure 7.1: Generating SAIF files from RTL simulation

In this thesis only RTL-level power estimation has been preformed. The reason for this is to speed up the estimation process. This comes to the expense of some accuracy of the power estimations.

Three basic scripts were used when synthesizing the Reed-Solomon designs. The first script was the constraints.tcl script. This script sets all the design constraints, such as clock speed, input and output delay, input and output load and maximum area when synthesizing. The clock speed was set to 40MHz, which is in between the operation frequencies of Energy Micro's microcontrollers (they range from 32 – 48MHz [26]). The constraints.tcl script can be seen in appendix D.2.2.

The other two scripts shown in appendix D.2.3 and D.2.4 are very similar to each other. The main difference is the *insert_clock_gating* command, which is used in the second script to automatically insert clock gating in the design. The flow of the synthesis script is described below:

1. Read Verilog design files using the analyze command.
2. Elaborate the top-level module
3. Apply constrains using the constraints.tcl file
4. Insert clock gating using the insert_clock_gating command
5. Compile the design
6. Read .saif file containing switching activity using read_saif command
7. Write report files

7.3 Synthesis of Reed-Solomon Encoder

The Reed-Solomon encoder design in this thesis was synthesized using the Design Compiler scripts described in section 7.2.2. The synthesis of the encoder was performed in two parts. First the encoder design was synthesized and power estimations were performed using the switching activity generated when one codeword was being encoded.

In the second part, the encoder was synthesized with clock gate insertion. To see the effect the clock gating had on the power consumption of the encoder, different activity levels when encoding was simulated. Activity level indicates how much of the simulation time the encoder is encoding data. The simulations were performed using 20 test vectors and the activity levels were set to 100% 75%, 50%, 25% and 5%. The test vectors used can be seen in appendix C.1.1.

7.4 Synthesis of Reed-Solomon Decoder

Seven different Reed-Solomon decoder configurations were designed and synthesized. Each design was simulated in Active-HDL to generate switching activity. The switching activity was used during power estimation of the system.

7.4.1 Decoder Configuration 1

The first decoder configuration is implemented using the standard syndrome module, key equation solver, Forney algorithm, Chien search and correction module. These modules are described in section 5.5. Configuration 1 uses the standard Berlekamp-Massey algorithm to calculate the error locator polynomial, and to solve the key equation. The standard Berlekamp-Massey algorithm was described in section 4.4.1. The Forney, Chien search and correction module are implemented and connected together using pipelining. This is done to reduce latency in the decoder. The pipelined architecture was described in section 5.5.6.

Decoder configuration 1 was simulated with test vectors containing zero errors, one error and two errors. Switching activity generated during the simulation was used to estimate the power consumption during the synthesis process. The test vectors used can be seen in appendix C.2.1.

7.4.2 Decoder Configuration 2

Decoder configuration 2 is implemented with the same standard modules as used in configuration 1, except for the standard Berlekamp-Massey algorithm. Configuration 2 uses an inversionless Berlekamp-Massey algorithm instead of the standard

algorithm. The inversionless Berlekamp-Massey algorithm removes the need for division, and therefore no inversion is needed to calculate the error locator polynomial. The algorithm was further described in section 4.4.2.

Configuration 2 was simulated using the same test vectors as in configuration 1. Information from this simulation was used to estimate the power consumption during synthesis.

7.4.3 Decoder Configuration 3

The third decoder configuration is implemented with the standard syndrome module, the inversionless Berlekamp-Massey algorithm, the standard Chien search, Forney module and error correction module. This module uses the same pipelined implementation on the Chien, Forney and correction module as used in configuration 1 and 2. To reduce the power consumption in the design, the modules in this configuration are implemented in such away that all modules not performing any calculation, are not clocked.

Switching activity used for estimation of power consumption was generated during simulation with test vectors that contained zero errors, one error and two errors. The test vectors were the same as the one used for simulation of configuration 1 and 2.

Decoder configuration 3 was synthesized both with full clock gate insertion and no clock gate insertion. The clock gate insertion was performed using the synthesis process in Design Compiler. This process was descried in section 7.2.2.

7.4.4 Decoder Configuration 4

Decoder configuration 4 is implemented with a modified syndrome module. This modified syndrome module calculates the first t syndrome values to check for errors. If the first t syndromes are equal to zero, it is assumed that there are no errors. If the syndrome values are not equal to zero, rest of the $2t$ syndromes are calculated. The modified syndrome module is described in section 5.5.1. All other modules in configuration 4 are the same one as used for configuration 3.

Configuration 4 was simulated using the test vectors in appendix C.2.1. The switching activity from this simulation was used for estimation of the power consumption. Synthesis was performed with no clock gate insertion.

7.4.5 Decoder Configuration 5

Configuration 5 is designed with a syndrome module that has a parallel architecture. The module uses two-parallel syndrome cells to calculate the syndrome

values, instead of the normal syndrome cells described in section 5.5.1. The two-parallel syndrome cells enables the decoder to input two symbols into the syndrome module at the time. This reduces the number of clock cycles needed to compute the syndromes. The two-parallel syndrome cells architecture is described in section 5.5.1. To reduce the power consumption, modules that are not performing any calculations are not clocked.

The test vectors used as input during simulation were the same as used for the other configurations. During synthesis of configuration 5, full clock gate insertion was implemented. The synthesis results can be seen in section 8.2.5.

7.4.6 Decoder Configuration 6

Configuration 6 uses the two-parallel syndrome cell configuration together with a modified Chien and Forney module. The Chien search module is implemented so that it finds all the error positions, before the Forney module finds the error values. The maximum number of calculations done in the Forney module are therefore two. There is no pipelining in configuration 6, since the Chien search finds all the error positions before any error values are found. This is the full serial architecture described in section 5.5.6. The decoding time in configuration 6 is longer then in configuration 5. This is because the error correction module waits for all the error positions and then for all the error values, before it starts to correct errors.

Configuration 6 was synthesized with and without clock gate insertion. The test vectors with zero errors, one error and two errors were used to generate switching activity for the power estimations. Results from the synthesis and the estimated power consumption are presented in section 8.2.6.

7.4.7 Decoder Configuration 7

Decoder configuration 7 is based on decoder configuration 5. This configuration uses two-parallel syndrome cells in the syndrome module, the inversionless Berlekamp-Massey algorithm and pipelined implementation of the Chien search, Forney and error correction module. Configuration 7 has some improvements compared to configuration 5. Configuration 7 has two data inputs into the decoder. This allows two symbols to be shifted into the decoder in each clock cycle. This improvements removes the need for a wait time before the syndrome values can be calculated, and it should lead to a shorter decoding time.

Configuration 7 was synthesized with clock gate insertion, and switching activity was generated using the test vectors in appendix C.2.1. The results from the synthesis and the power estimations performed on configuration 7 are presented in section 8.2.7.

Chapter 8

Synthesis and Simulation Results

The encoder and the decoder configurations were simulated and synthesized using Active-HDL and Design Compiler from Synopsys. Switching activity from the simulations was used to estimate the power consumption of the designs using Synopsys Power Compiler.

8.1 Synthesis Results Encoder

Figure 8.1 shows a view of Design Vision and the synthesized encoder design. The area and gate count results for the encoder can be seen in table 8.1. A NAND2 gate has been used as gate equivalent in this thesis. The NAND2 gate has an area of 1,887200, as described in section 7.2.2.

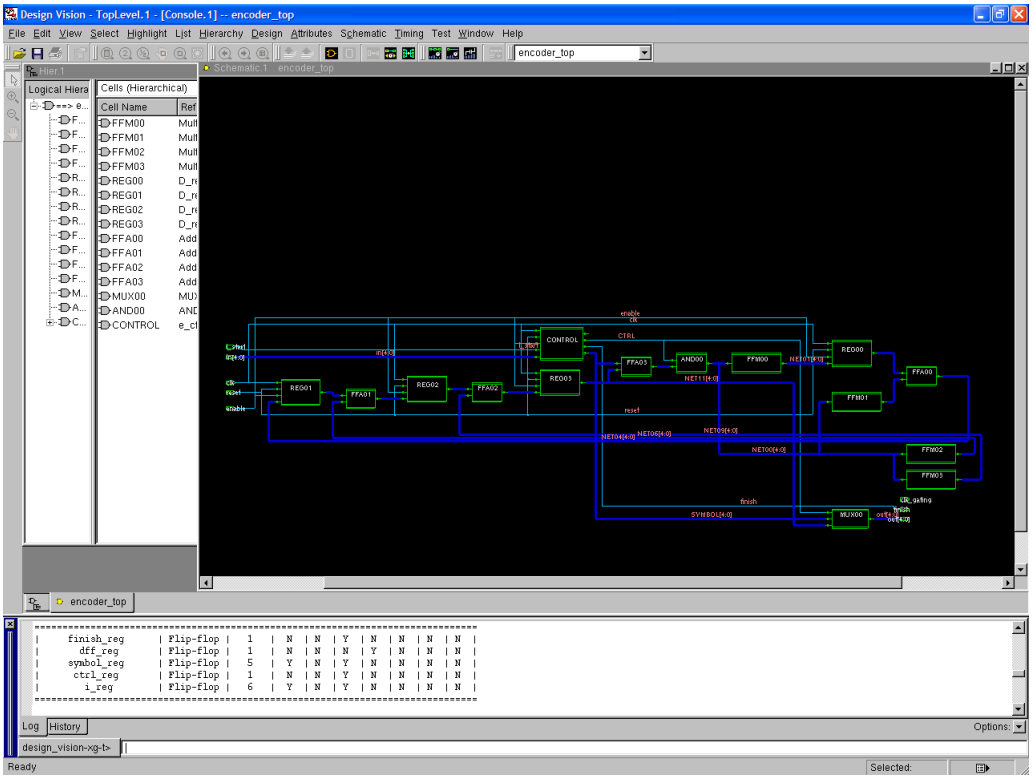

Figure 8.1: Synthesized encoder view

Area		**NAND2 gate equivalent**
Combinational	749	397
Sequential	372	197
Total	1121	594

Table 8.1: Area usage of encoder

The estimated total power consumption for encoding one codeword can be seen in table 8.2. The table also shows the dynamic and leakage power for the same test case. Table 8.3 shows the calculated energy consumption for the encoder that encodes one codeword.

Internal Power (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
70.29	17.27	8.30	95.86

Table 8.2: Power consumption encoder

Total Power (μW)	**Runtime** (μs)	**Total Energy** (nJ)
95.86	0.863	0.0827

Table 8.3: Energy consumption encoder

The dynamic power consumption for different activity levels is shown in figure 8.2. The blue bars show the dynamic power for encoder with no clock gating and the red bars show the dynamic power consumption for the encoder design with clock gating.

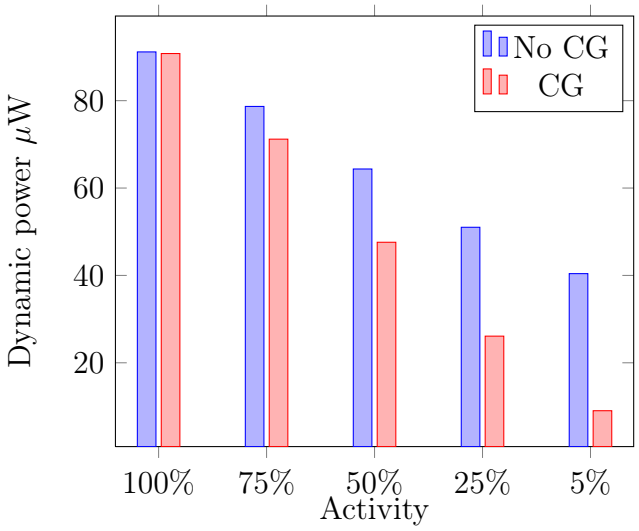

Figure 8.2: Dynamic power consumption with and without clock gating

Table 8.4 shows the power consumption when the encoder design is clock gated.

Internal Power (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
65.85	18.66	7.81	92.32

Table 8.4: Power consumption encoder with clock gating

8.2 Synthesis Results Decoder

8.2.1 Configuration 1 Synthesis and Simulation Results

The Reed-Solomon decoder configuration 1 is implemented with the standard Berlekamp-Massey algorithm, which is used to solve the key equation. This configuration is described in section 7.4.1. Figure 8.3 shows a screenshot of the synthesized design. The synthesis and simulation results for decoder configuration 1 are shown in table 8.5, and table 8.6.

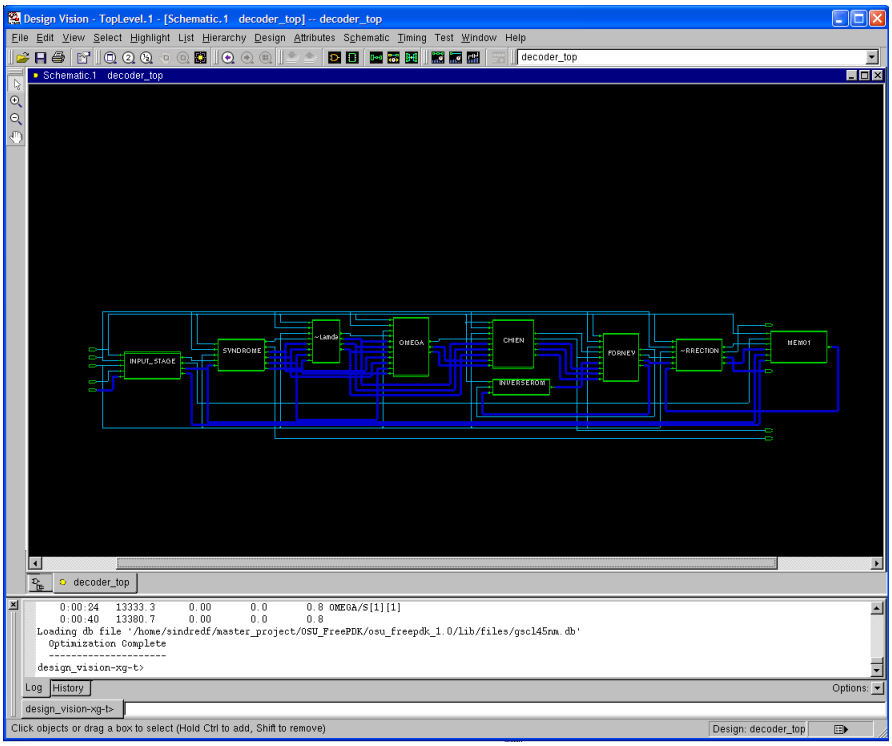

Figure 8.3: Block diagram view of configuration 1

Area		NAND2 gate equivalent
Combinational	10593	5613
Sequential	5332	2825
Total	15925	8438

Table 8.5: Area usage of decoder configuration 1

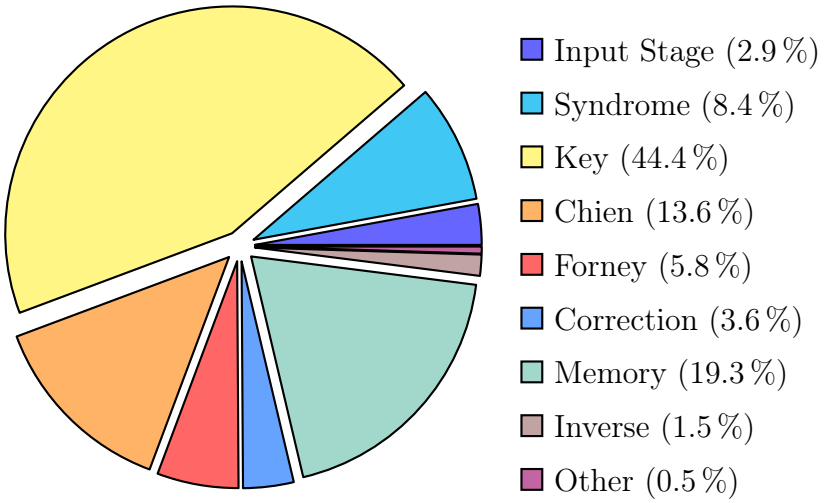

Figure 8.4: Area distribution decoder configuration 1

In figure 8.4 the area distribution for configuration 1 is shown. The estimated power consumption for the standard Berlekamp-Massey module is shown in table 8.7. The energy consumption of decoder configuration 1 is shown table 8.8

Errors	**Internal Power** (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
0	565.11	22.68	101.01	689
1	569.50	28.73	101.01	699
2	575.15	31.42	101.01	708

Table 8.6: Power consumption decoder configuration 1

Errors	**Total Power** (μW)
0	283.1
1	303.2
2	301.6

Table 8.7: Power consumption standard Berlekamp-Massey algorithm

Errors	**Total Power** (μW)	**Runtime** (μs)	**Total Energy** (nJ)
0	689	0.913	0.629
1	699	2.713	1.896
2	708	2.738	1.939

Table 8.8: Energy consumption configuration 1

8.2.2 Configuration 2 Synthesis and Simulation Results

Decoder configuration 2 was simulated and synthesized with the inversionless Berlekamp-Massey algorithm. Configuration 2 is further described in section 7.4.2. Synthesis results and estimated power consumption for decoder configuration 2 are shown in table 8.9 and 8.10. A more detailed view of the power consumption for the inversionless Berlekamp-Massey module is shown in table 8.11. The energy consumption of configuration 2 can be seen in table 8.12.

Area		**NAND2 gate equivalent**
Combinational	10204	5407
Sequential	5188	2749
Total	15392	8156

Table 8.9: Area usage of decoder configuration 2

Errors	**Internal Power** (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
0	550.55	22.48	97.69	671
1	554.48	27.92	97.69	680
2	560.63	31.19	97.69	690

Table 8.10: Power consumption decoder configuration 2

Errors	**Total Power** (μW)
0	266.1
1	290.1
2	287.4

Table 8.11: Power consumption inversionless Berlekamp-Massey algorithm

Errors	**Total Power** (μW)	**Runtime** (μs)	**Total Energy** (nJ)
0	671	0.913	0.613
1	680	2.813	1.913
2	690	2.813	1.941

Table 8.12: Energy consumption configuration 2

8.2.3 Configuration 3 Synthesis and Simulation Results

The third decoder configuration was simulated and synthesized using the configurations described in section 7.4.3. In this configuration inactive modules were not clocked. This was done to reduce the power consumption of the system. Synthesis

results and the estimated power consumption for this configuration can be found in table 8.13 and 8.15. The power consumption was estimated using test vectors with zero errors, one error and two errors as shown in the table.

Area		**NAND2 gate equivalent**
Combinational	11889	6299
Sequential	5668	3003
Total	17557	9302

Table 8.13: Area usage of decoder configuration 3

Area		**NAND2 gate equivalent**
Combinational	7938	4206
Sequential	6065	3214
Total	14003	7420

Table 8.14: Area usage of decoder configuration 3 with clock gate insertion

Table 8.14 shows the area of configuration 3 with full clock gate insertion. Estimated power consumption with full clock gate insertion can be seen in table 8.16.

Errors	**Internal Power** (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
0	276.91	31.35	108.42	417
1	307	35.36	108.42	451
2	317.25	39.27	108.42	465

Table 8.15: Power consumption decoder configuration 3

Errors	**Internal Power** (μW)	**Switching Power** (μW)	**Leakage Power** (μW)	**Total Power** (μW)
0	175.40	34.56	100.16	310
1	126.42	38.72	100.20	265
2	133.39	41.53	100.20	275

Table 8.16: Power consumption decoder configuration 3 with clock gate insertion

Table 8.17 shows the energy consumption of configuration 3 with no clock gate insertion. The energy consumption of configuration 3 with full clock gating can be seen in table 8.18.

Errors	**Total Power** (μW)	**Runtime** (μs)	**Total Energy** (nJ)
0	417	0.938	0.391
1	451	2.963	1.336
2	465	2.963	1.378

Table 8.17: Energy consumption configuration 3

Errors	**Total Power** (μW)	**Runtime** (μs)	**Total Energy** (nJ)
0	310	0.938	0.291
1	265	2.963	0.778
2	275	2.963	0.815

Table 8.18: Energy consumption configuration 3 with clock gating

Figure 8.5 shows a detailed view of the power consumption for each module when there are zero errors, one error and two errors. The power consumed by each module measured in percentage is shown in figure 8.6 and 8.7.

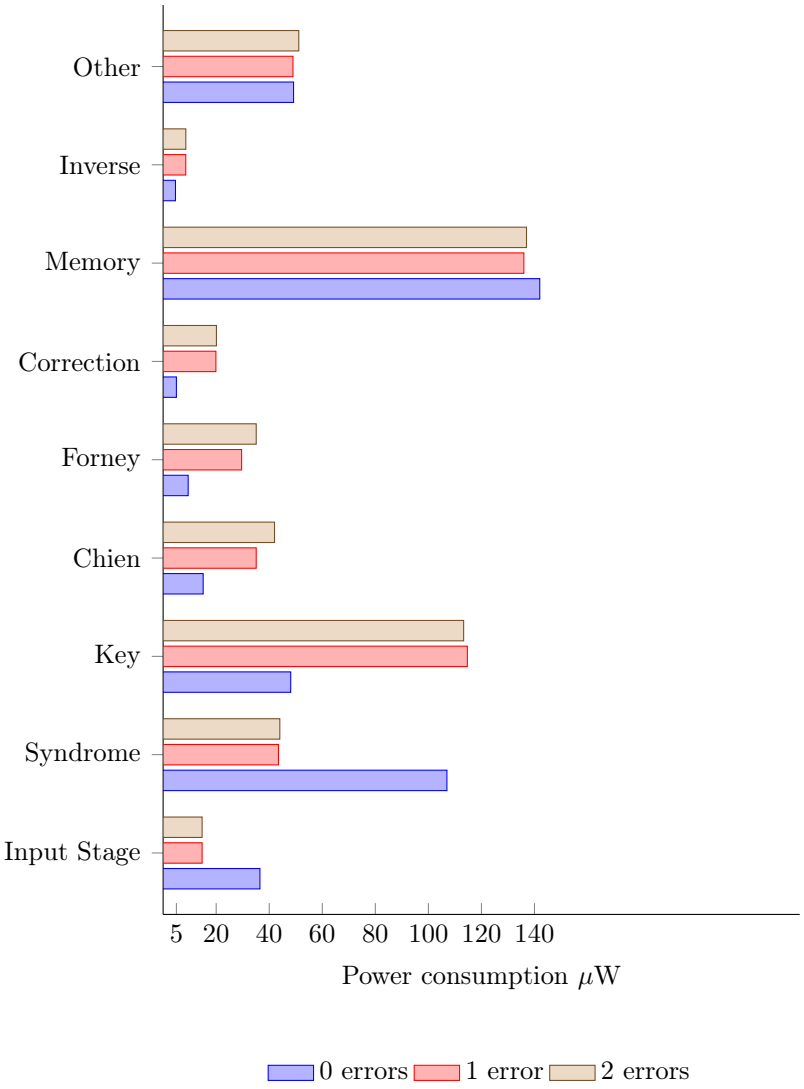

Figure 8.5: Power consumption of different modules in configuration 3

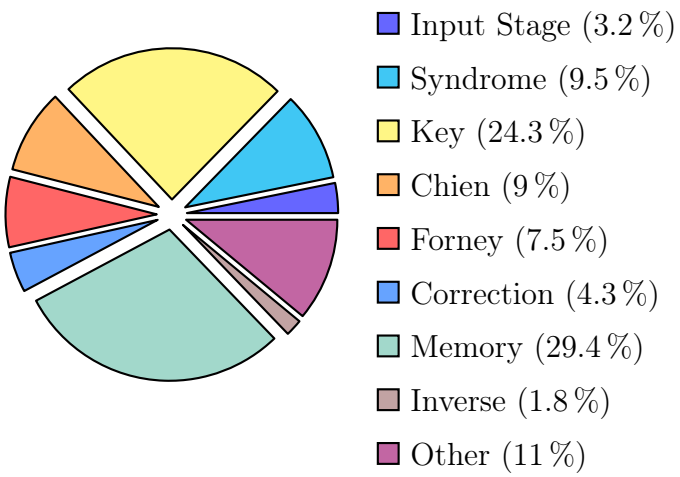

Figure 8.6: Power consumption distribution with two errors

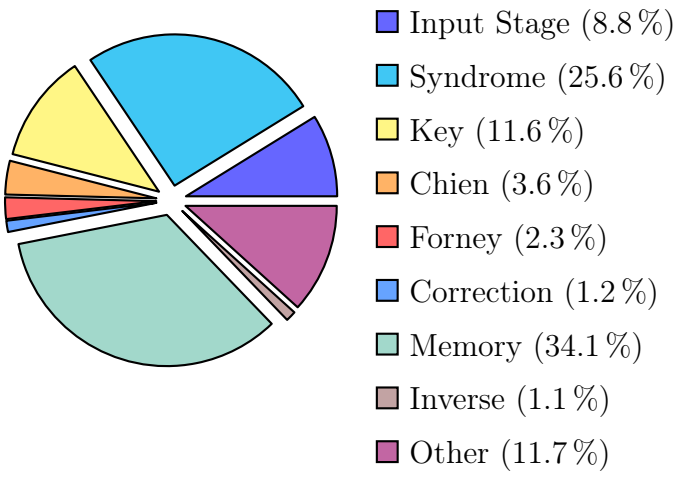

Figure 8.7: Power consumption distribution with zero errors

8.2.4 Configuration 4 Synthesis and Simulation Results

Configuration 4 was simulated and synthesized with a modified syndrome module, which calculates t syndromes first, to check if the received codeword is error free. This configuration is described in section 7.4.4. The estimated power consumption was done with regards to received codewords that had zero errors, one error and two errors. The synthesis and power consumption results for decoder configuration 4 are shown in table 8.19 and 8.20. Table 8.21 shows the energy consumption of configuration 4.

Area		NAND2 gate equivalent
Combinational	11914	6313
Sequential	5781	3063
Total	17695	9376

Table 8.19: Area usage of decoder configuration 4

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	272.55	28.12	110.06	411
1	297.11	32.95	110.06	440
2	304.15	36.15	110.06	450

Table 8.20: Power consumption decoder configuration 4

Errors	Total Power (μW)	Runtime (μs)	Total Energy (nJ)
0	411	0.938	0.385
1	440	3.763	1.655
2	450	3.763	1.693

Table 8.21: Energy consumption configuration 4

8.2.5 Configuration 5 Synthesis and Simulation Results

Decoder configuration 5 was implemented using two-parallel syndrome cells, as described in section 7.4.5. Synthesis results for configuration 5 can be seen in table 8.22. The estimated power consumption is shown in table 8.24. Synthesis and simulation results with full clock gate insertion are shown in table 8.23 and 8.25. The energy consumption of configuration 5 with full clock gating can be seen in table 8.26.

Area		NAND2 gate equivalent
Combinational	13222	7006
Sequential	5953	3154
Total	19175	10160

Table 8.22: Area usage of decoder configuration 5

Area		NAND2 gate equivalent
Combinational	9089	4816
Sequential	6372	3376
Total	15461	8192

Table 8.23: Area usage of decoder configuration 5 with clock gate insertion

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	243.20	25.18	118.46	387
1	290.46	31.79	118.46	441
2	297.99	35.15	118.46	452

Table 8.24: Power consumption decoder configuration 5

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	115.82	31.46	109.54	257
1	111.71	36.80	109.54	258
2	118.44	39.16	109.54	267

Table 8.25: Power consumption decoder configuration 5 with clock gate insertion

Errors	Total Power (μW)	Runtime (μs)	Total Energy (nJ)
0	257	1.438	0.369
1	258	3.463	0.893
2	267	3.463	0.925

Table 8.26: Energy consumption configuration 5 with clock gating

8.2.6 Configuration 6 Synthesis and Simulation Results

Configuration 6 was implemented using a modified Chien search module, as described in section 7.4.6. The synthesis results and the estimated power consumption for this configuration are shown in table 8.27 and table 8.29. Synthesis and simulation results with full clock gate insertion are shown in table 8.28 and 8.30. Table 8.31 shows the energy consumption of configuration 6 with full clock gating.

Area		NAND2 gate equivalent
Combinational	14256	7554
Sequential	7034	3727
Total	21290	11281

Table 8.27: Area usage of decoder configuration 6

Area		NAND2 gate equivalent
Combinational	10063	5332
Sequential	7483	3965
Total	17546	9297

Table 8.28: Area usage of decoder configuration 6 with clock gate insertion

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	252.90	25.47	134.84	413
1	283.43	25.38	134.84	444
2	286.46	26.54	134.84	448

Table 8.29: Power consumption decoder configuration 6

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	122.82	32.14	126.49	281
1	102.71	30.49	126.46	260
2	105.40	31.51	126.46	263

Table 8.30: Power consumption decoder configuration 6 with clock gate insertion

Errors	Total Power (μW)	Runtime (μs)	Total Energy (nJ)
0	281	1.438	0.404
1	260	4.413	1.147
2	263	4.463	1.173

Table 8.31: Energy consumption configuration 6 with clock gating

8.2.7 Configuration 7 Synthesis and Simulation Results

Decoder configuration 7 was implemented using two-parallel syndrome cells. The synthesis results for configuration 7 can be seen in table 8.32. The estimated power consumption is shown in table 8.34. Synthesis and simulation results with full clock gate insertion are shown in table 8.33 and 8.35. Energy consumption for decoder configuration 7 can be seen in table 8.36.

Area		NAND2 gate equivalent
Combinational	13805	7315
Sequential	5893	3122
Total	19690	10437

Table 8.32: Area usage of decoder configuration 7

Area		NAND2 gate equivalent
Combinational	9463	5014
Sequential	6307	3341
Total	15770	8355

Table 8.33: Area usage of decoder configuration 7 with clock gate insertion

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	305.84	44.77	121.77	472
1	320.72	38.41	121.77	481
2	330.26	42.71	121.77	495

Table 8.34: Power consumption decoder configuration 7

Errors	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)
0	197.90	50.91	109.21	358
1	135.30	43.39	109.21	288
2	144.40	46.58	109.21	300

Table 8.35: Power consumption decoder configuration 7 with clock gate insertion

Errors	Total Power (μW)	Runtime (μs)	Total Energy (nJ)
0	358	0.587	0.210
1	288	2.612	0.752
2	300	2.612	0.783

Table 8.36: Energy consumption configuration 7 with clock gating

Chapter 9

Evaluation of Results

The Reed-Solomon system designed in this thesis has not been compared to previously made Reed-Solomon designs. The reason for this is that no other RS(31, 27) code, synthesized with 45nm CMOS technology have been found.

9.1 Encoder

The Reed-Solomon encoder was implemented to be as compact as possible. The constant multipliers implemented in the encoder uses an absolute area of $23 - 40$, compared to the full multipliers that uses an absolute area of 235. By using constant multipliers the total area of the design decreases.

Clock gating was implemented in the encoder design to reduce the power consumption. During encoding, almost all logic elements are clocked. To see the effect of the clock gating, different activity levels had to be simulated. This was described in section 7.3. Figure 9.1 shows the total energy consumption for the encoder with different activity levels, when encoding 20 codewords. When the activity level of the encoding goes below 50%, we see the largest energy saving potential. Energy consumption can be reduced with up to 65%, when implementing clock gating on a encoder that has an activity level of only 5%.

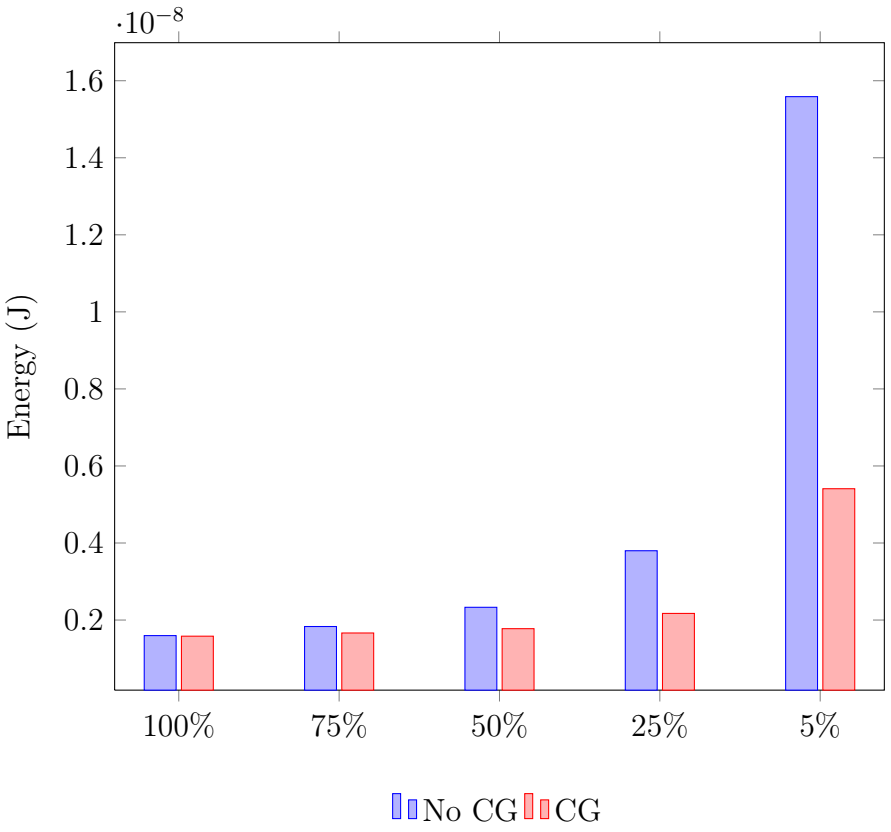

Figure 9.1: Energy consumption of encoder

9.2 Decoder

The different decoder configurations were presented in section 7.4.

The chart in figure 8.4 on page 59 shows that the implementation of the Berlekamp-Massey algorithm for solving the key equation, takes up almost half of the total area of the decoder. This was expected, as the implementation of the Berlekamp-Massey algorithm uses several Galois multipliers and adders. It has also several temporary registers used for storing intermediate result. It is also worth noticing that the memory unit, used for storing the received symbols while decoding, takes up the second most area in the decoder. The "other" clause shown in the chart is a control circuit and multiplexers implemented at the top-level of the decoder. The control circuit and multiplexers are used to control which module that can access the memory modules.

Comparing the synthesis results for configuration 1 and 2 shown in table 8.5 and 8.9, one can see that configuration 2 has a 3,3% lower gate count than configuration 1. The inversionless Berlekamp-Massey algorithm do not need to access the ROM that stores the inverse Galois field values. This reduces some of the logic needed to implement the inversionless Berlekamp-Massey module. Thus, the overall area

usage of the encoder decreases.

When comparing the total power consumption of the two configuration shown in table 8.6 and 8.10 on page 59 and 60, we can seen that configuration 2 uses between 18 - 19μW less power than configuration 1 in all three test cases. Table 8.7 on page 59 and table 8.11 on page 60 shows the power consumption of the two different Berlekamp-Massey algorithms. Comparing these results, we see that the inversionless Berlekamp-Massey implementation uses slightly less power than the standard Berlekamp-Massey algorithm.

Errors	**Total Energy Config 1** (nJ)	**Total Energy Config 2** (nJ)
0	0.629	0.613
1	1.896	1.913
2	1.939	1.941

Table 9.1: Energy consumption of configuration 1 and 2

However, the configuration 2 with the inversionless Berlekamp-Massey implementation uses about 100ns more to decode a codeword. Table 9.1 shows the energy consumption of configuration 1 and 2. Comparing the results for the two designs, configuration 2 uses at average 0,50% more energy than configuration 1, when decoding codeword containing errors.

Decoder configuration 3 was synthesized both with and without clock gate insertion. The synthesis results can be seen in table 8.13 and 8.14 on page 61. Configuration 3 with clock gate insertion has a decrease in gate count with about 20%. Design Compiler optimizes the design when clock gate insertion is being used. This results in lower area usage.

Errors	**Dynamic Power** (μW)	**Dynamic Power With CG** (μW)	**Reduction** (%)
0	308.26	209.96	32
1	342.36	165.14	52
2	356.52	174.92	51
Average			45

Table 9.2: Comparison of dynamic power configuration 3

Table 9.2 shows the dynamic power consumption for decoder configuration 3. Configuration 3 with clock gate insertion has a average 45% reduction in dynamic power, compared to the same configuration with no clock gating.

Configuration 4, which was implemented using a modified syndrome module, has a small reduction in power consumption, compared to configuration 3 as shown in table 9.3. Table 9.4 shows the energy consumption for configuration 3 and 4.

When comping the results for the two designs, we can see that configuration 4 uses more energy when there are one and two errors in the codeword. In these two cases the syndrome module uses twice as much time to calculate the syndromes, because only t syndromes are calculated at the time. When there are no errors in the codeword, configuration 4 has a moderate decrease in energy consumption.

Errors	**Total Power Config 3** (μW)	**Total Power Config 4** (μW)	**Reduction** (%)
0	417	411	1.44
1	451	440	2.44
2	465	450	3.22
Average			2.37

Table 9.3: Comparison of power consumption of configuration 3 and 4

Errors	**Total Energy Config 3** (nJ)	**Total Enrgy Config 4** (nJ)
0	0.391	0.385
1	1.336	1.655
2	1.377	1.693

Table 9.4: Energy consumption of configuration 3 and 4

Table 9.5 and 9.6 shows the reduction of power consumption for configuration 5 and 6 when clock gate insertion is performed. As seen from the results for configuration 5 and 6, clock gating has a good effect, reducing the average power consumption with about 40%.

Errors	**Total Power** (μW)	**Total Power With CG** (μW)	**Reduction** (%)
0	387	257	33
1	441	258	41
2	452	267	41
Average			38

Table 9.5: Reduction of total power for configuration 5

Errors	**Total Power** (μW)	**Total Power With CG** (μW)	**Reduction** (%)
0	413	281	32
1	444	260	41
2	448	263	41
Average			38

Table 9.6: Reduction of total power for configuration 6

Figure 9.2 shows the average dynamic power consumption for the four configuration 3, 5, 6 and 7. Using configuration 3 as a reference, it can be seen that there is a reduction in dynamic power for both configuration 5 and 6. Clock gating has the greatest effect when comparing the non-clock gated designs to the clock gated designs. With the techniques used in configuration 5 and 6, the average dynamic power has been reduced with approximately 30μW in configuration 5, and 40μW in configuration 6. Configuration 7 has the highest dynamic power consumption of the four configurations shown in table 9.2.

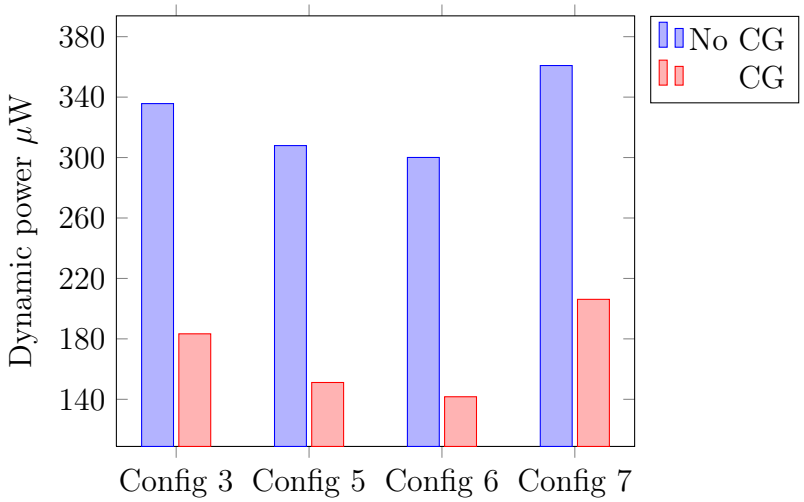

Figure 9.2: Dynamic power consumption for configuration 3, 5, 6 and 7

The total average power consumption for the same four configurations are shown in figure 9.3. As seen in the figure, configuration 5 has the lowest total power consumption. Compared to configuration 3, configuration 5 and 6 have increasing leakage power, and in configuration 6 the leakage power account for almost half the total power consumption. The increase in leakage power from configuration 5 to 6 is approximately 20μW, making configuration 6 use more power than configuration 5. Therefore, the reduction in dynamic power from configuration 5 to 6 has no effect on the total power consumption. Configuration 7 consumes the most power when comparing the four configurations.

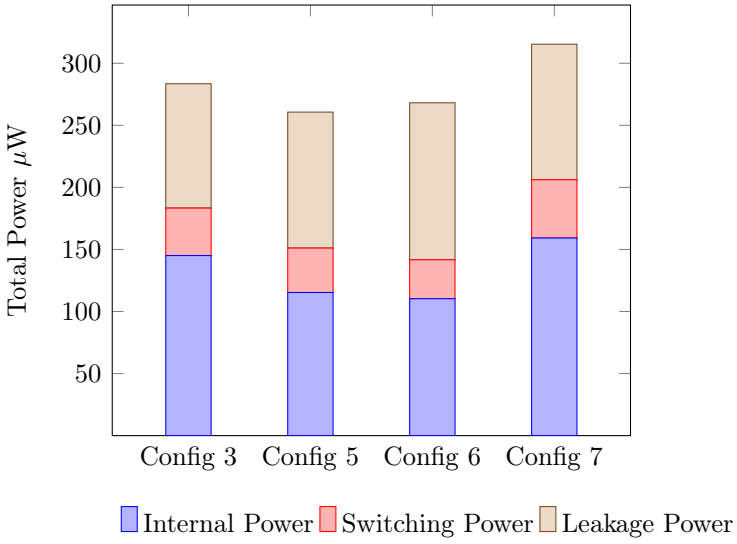

Figure 9.3: Comparison of total power consumption for different configurations

Figure 9.4 shows the energy consumption of configuration 2, 3, 5, 6 and 7 simulated with zero errors, one error and two errors. Configuration 3 has the same implementation as configuration 2, only with clock gating. Configuration 5 and 6 are implemented with techniques to reduce the power consumption of the decoder design, as described in section 5.3 and 5.5. For all three test cases configuration 3 with clock gating uses substantially less energy than configuration 2. The average reduction in energy consumption from configuration 2 to configuration 3 is approximately 58%. Comparing configuration 5 and 6 to configuration 3, we see that there is a increase in energy for all three test cases. The average energy consumption increases 18% from configuration 3 to configuration 5, and 43% from configuration 3 to configuration 6.

Configuration 5 was implemented with a single data input into the decoder and two-parallel syndrome cells. The syndrome module therefore had to wait for at least half the symbols to enter, before it could start calculating the syndromes. This caused the decoding time to increase, resulting in an increase in energy consumption as shown in table 9.4. To remove the wait time a more optimized configuration was designed. As seen in figure 9.2 and 9.3, configuration 7 consumes more power compared to the other configurations. The decoding time however, is decreased because the two-parallel syndrome cells have been properly implemented. This causes the overall energy consumption shown in figure 9.4, to be reduced compared to both configuration 3, 5 and 6. Configuration 7 has a 7% average reduction in energy consumption, compared to configuration 3. Comparing the two last configurations, we see that configuration 7 has 36% lower energy consumption than configuration 6.

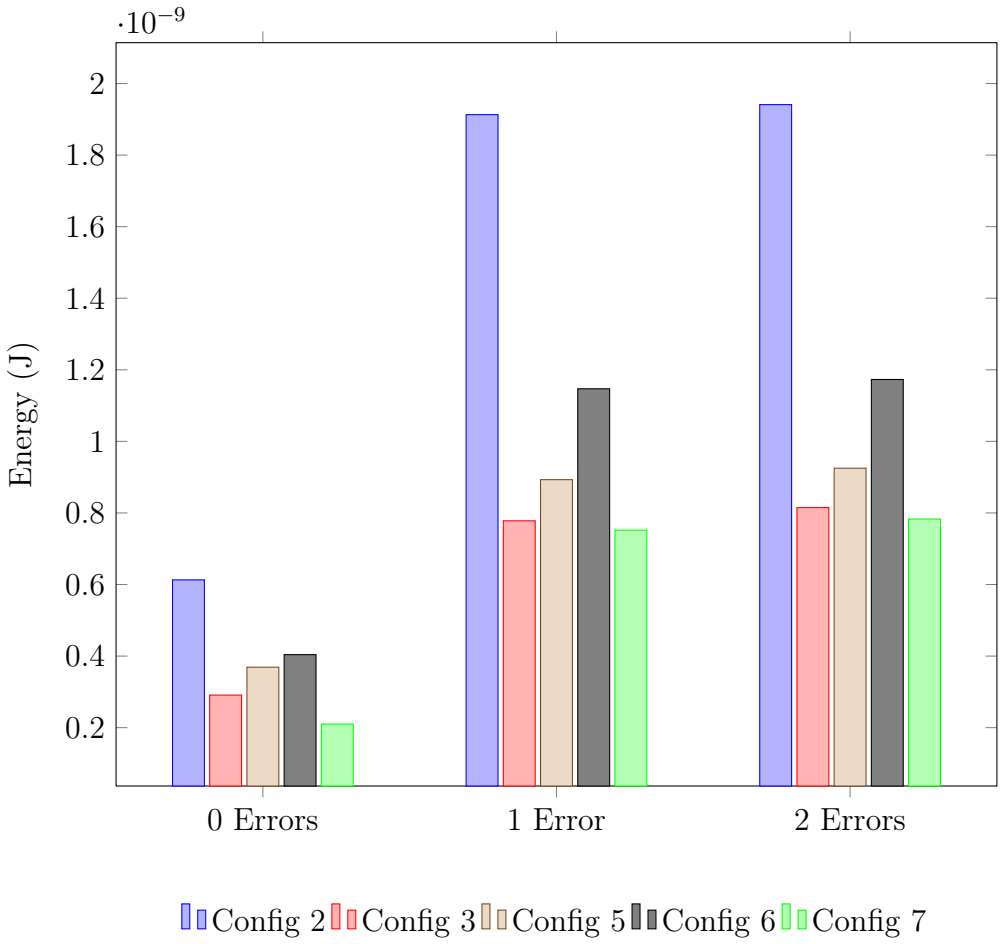

Figure 9.4: Energy consumption for configuration 2, 3, 5, 6 and 7

Looking at the runtime results for the configurations shown in tables 9.7 and 9.8, the average increase in runtime is 29% for configuration 5 and 51% for configuration 6. This contributes greatly to the increase in energy consumption for the two designs, compared to configuration 3.

Errors	**Runtime Config 3** (μs)	**Runtime Config 5** (μs)	**Increase** (%)
0	0.938	1.438	53.30
1	2.963	3.463	16.87
2	2.963	3.463	16.87
Average			29.01

Table 9.7: Runtime comparison configuration 3 and 5

Errors	Runtime Config 3 (μs)	Runtime Config 6 (μs)	Increase (%)
0	0.938	1.438	53.30
1	2.963	4.413	49.54
2	2.963	4.463	50.62
Average			51.15

Table 9.8: Runtime comparison configuration 3 and 6

Table 9.9 shows a comparison of the runtime for configuration 3 and 7. One can see that the parallel syndrome cells help reducing the average decoding time by 20%.

Errors	Runtime Config 3 (μs)	Runtime Config 7 (μs)	Decrease (%)
0	0.938	0.587	37.42
1	2.963	2.612	11.84
2	2.963	2.612	11.84
Average			20.36

Table 9.9: Runtime comparison configuration 3 and 7

Figure 9.5 shows an area comparison of configuration 3, 5, 6 and 7. Configuration 3 with no improvements has the lowest gate count of the four configurations. Configuration 7, which has the lowest energy consumption, has a 12% increase in gate count, compared to configuration 3. This can be explained by the implementation of the two-parallel syndrome cells. To implement these syndrome cells, extra logic is needed.

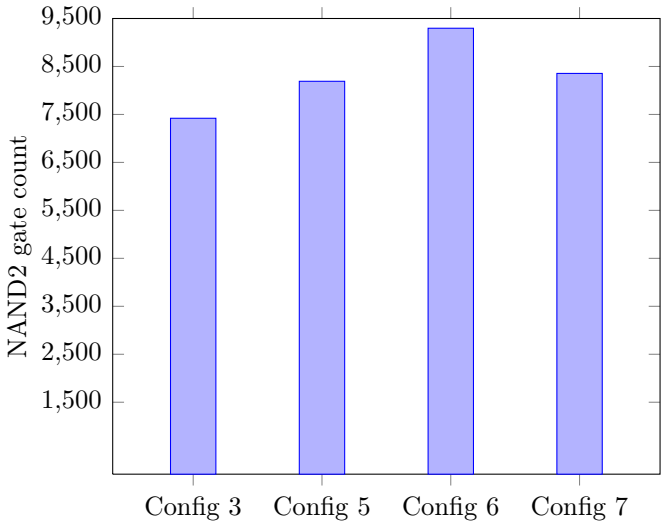

Figure 9.5: Area comparison of decoder configurations

Chapter 10

Discussion

10.1 Encoder

The Reed-Solomon encoder was implemented using the LFSR architecture described in section 5.4. In this thesis, a fixed rate encoder was used to reduce the hardware implementation. By using a fixed rate encoder, a predetermined code generator polynomial can be used. This allows us to use constant multipliers in the encoder implementation, instead of using full multipliers. Full multipliers are more complex and use more area. The constant multipliers implemented in this thesis have an absolute area of $23 - 40$, depending on the multiplying value. To comparison, the full multipliers implemented in this thesis have an absolute area of approximately 235. This is an increase of $500 - 1000\%$ for each multiplier, depending on which value we want to multiply with. The encoder needs four multipliers, so the total area would be: $area\ multiplier \times 4$. From this it is clear that the use of constant multipliers will reduce the total area of the encoder by a great amount, when compared to full multipliers. Synthesis tools have the ability of optimizing and removing unused logic in a design. Full multipliers could have been used instead of constant multipliers. The synthesis tool could then have removed the unused and redundant hardware. By using this approach, we could save time during the implementation phase because only one multiplier has to be made. Using the constant multipliers, however, gives us greater control on how the synthesis tool implement the multipliers.

Clock gate insertion was used in the encoder design to reduce the power consumption. A reduction in power consumption leads to a reduced energy consumption. The energy consumption for the encoder was presented in figure 9.1 on page 70. When the encoder only encodes 5% of the time, the energy consumption can be reduced with over 50% when the design is clock gated. For Reed-Solomon encoders that have a low activity rate, clock gating can be used to reduce the power consumption thus reducing the energy. The energy consumption varies a great amount for the different activity levels. The reason for this is that the same amount of codewords were encoded each time. To get the right relationship between encoding

and idle (i.e activity level), the run times had to increase.

10.2 Decoder

When comparing the standard Berlekamp-Massey algorithm with the inversionless Berlekamp-Massey algorithm, the inversionless algorithm uses at average 15μW less power than the standard algorithm. The decoding process takes longer time with the inversionless Berekamp-Massey algorithm, therefore the total energy consumption of the configuration is higher with this algorithm. The gate count is however lower with the inversionless algorithm with about 3,3%. This can be explained by that the standard Berlekamp-Massey algorithm needs to access the ROM module that stores the inverse values. This requires some extra logic that the inversionless Berlekamp-Massey algorithm do not need. Using the inversionless algorithm, gives us a possibility of reusing some of the hardware. The hardware implemented to calculate the discrepancy can also be used, with some modifications, to calculate the error evaluator polynomial. By doing this, the module that calculates the error evaluator polynomial can be removed. This would possibly reduce the total area of the hardware implementation of the key equation solver. The reduction of hardware components could also lead to a reduction in power consumption. This optimization has not been done in this thesis.

Configuration 3 had an 45% average reduction in dynamic power when the design was fully clock gated. The result from the power estimation was expected, as clock gating can give 40 to 50% reduction in dynamic power consumption. The results show that clock gating is a very efficient technique for reducing dynamic power. Figure 8.5 on page 63 shows the power consumption for configuration 3 without clock gating when decoding. It is interesting to notice that the syndrome module uses over twice as much power, when calculating syndrome values for codewords with no errors. The input stage module also uses more power, when there are no errors in the codeword. It is not known why the syndrome module uses more power in the no errors test case. As seen in the same figure the other modules use less power when there are no errors. This was expected, as these modules do not perform any calculations when no errors have been detected.

In figure 8.7 on page 64 one can see the power distribution between the different modules. As seen from the figure, the syndrome module uses 25% of the total power when decoding codewords with zero errors. With this in mind, configuration 4 was implemented with a syndrome module that only computes the first half of the syndrome values to check if there are errors. If the t first syndrome values are all zero, there is no errors. If they are not zero, rest of the $2t$ syndrome values are calculated. It was expected that this would reduce the energy consumption for codewords with zero errors, since fewer calculation are performed. For codewords with errors, it was expected that the energy consumption would increase. The reason for this is that t plus t values are calculated in sequence, instead of $2t$ values calculated in parallel. This increases the total runtime of the decoding

process. In the zero error test, the reduction in energy consumption was 1,5%. For the test case containing an error, the average increase in energy was 22%. Based on these results, the syndrome module used in configuration 4 should only be used where the codewords rarely contain any errors. The decrease in energy consumption by 1,5% was much less than expected. Only half the calculations are performed in the zero error test case, so it would be fare to assume that the decrease should have been higher. Taking into account that some extra logic was implement to check if t syndromes were zero.

By using two-parallel syndrome cells (configuration 5) and then reduce then number of calculation performed in the Forney module (configuration 6) by only calculating the error values for known error positions, the dynamic power consumption was reduced. Comparing the results to configuration 3 with clock gating, the reduction in dynamic power was approximately 18% for configuration 5 with clock gating, and 23% for configuration 6 with clock gating.

Looking at the total power consumption shown in figure 9.3, the power consumption has only been reduced with 23μW from configuration 3 to configuration 5. This is approximately a 8% reduction. From configuration 3 to configuration 6 the reduction in total power consumption is 15μW, which is a 5% reduction. The reduction in dynamic power do not have major impact on the total power. This is because the the leakage power in configuration 5 and 6 increases, compared to the leakage power in configuration 3. From configuration 5 to configuration 6 the total power has increased with approximately 8μW, even though the dynamic power is reduced from configuration 5 to configuration 6. The leakage power increases with 17μW from configuration 5 to 6, which is a increase of 15%.

This increase in leakage power can be traced back to the extra logic added in configuration 5 and configuration 6. Implementation of two-parallel syndrome cells require some extra hardware. For each cell an extra adder, multiplier and AND gate are implemented, compared to the standard syndrome cell. The modification done in configuration 6 also requires extra logic. In the Chien module extra registers have been implemented to temporary store the error positions, since all the error positions are found before the error values are calculated. Control logic is also needed to control this process. The same is done in the Forney module, since all error values are calculated before the symbols are corrected. The extra hardware needed to implement the registers and control circuitry makes the leakage power increase. This makes the total power increase, even though the calculations are reduced.

As presented in figure 9.4, by clock gating configuration 2 the average energy consumption was reduced with 58%. However, configuration 5 with the two-parallel syndrome cells, and configuration 6 that use modified Chien serach and Forney module, had an increase in energy consumption compared to configuration 3. Configuration 5 had an increase of 18%, and configuration 6 had an increase of 43% in energy consumption. The increase in energy is manly due to the longer decoding time in the two configurations. As was shown in table 9.7 and 9.8, which compared

the runtime of configuration 3, 5 and 6, configuration 3 had the lowest runtime of the three. At average, configuration 5 has a 29% longer runtime then configuration 3, and configuration 6 has a 51% longer runtime than configuration 3.

Configuration 5 uses two-parallel syndrome cells and the same pipelined Chien search, Forney and error correction design, as used in configuration 3. The parallel syndrome architecture implemented in configuration 5 should ideally decrease the runtime. In configuration 5 and 6 only one symbol is shifted into the decoder at the time. The syndrome module therefore have to wait for at least half the codeword to be shifted in, before it can start to calculate the syndrome values. This is done so the syndrome module does not have to stop while calculating the syndromes, since two symbols are send into the syndrome module at the time. This increase in runtime could be reduced by shifting in two symbols into the decoder at each clock cycle. This would eliminate the need for a wait time before the syndrome calculations start, and reduce the decoding time for these configurations.

A more optimized design was implemented to remove the wait time problem in configuration 5. In configuration 7, two symbols are shifted into the decoder at each clock cycle. A wait time is therefore not needed before the syndrome module starts to calculate the syndrome values. This configuration has a 20% decrease in decoding time, compared to configuration 3. The total power consumption is however increased, compared to configuration 3. The average energy consumption for configuration 7 is therefore only reduced with 7%. Implementing the two-parallel syndrome cells adds extra logic to the design. Configuration 7 has an gate count increase of 12%, compared to the gate count for configuration 3. Seeing as energy efficiency is the main focus, this increase in gate count can be manageable.

In configuration 6 the Chien search, Forney and error correction module are not pipelined like they were in configuration 3 and 5. Configuration 6 uses two-parallel syndrome cells and a modified Chien search and Forney module. The Chien search module calculate n values and the Forney module calculates at maximum t values. The modified Forney module used in configuration 6 performs 29 fewer calculations than the original Forney module used in configuration 3 and 5. Because the Forney module has to wait for the Chien search to finish, the decoding time increases. For the three last steps in the decoding process, configuration 6 needs at least $n+t+n$ clock cycles to correct and shift out the codeword. This is more then the pipelined designs, which use $n+5$ clock cycles to correct and shift out the codeword. The reduction in power consumption is to low compared to the increase in decoding time. The effect of reducing the calculations in the Forney module is therefore not good enough.

If the Chien search and Forney module in configuration 6 are optimized to reduce the power consumption even more, the increase of clock cycle in this implementation could be tolerable. For the energy consumption for configuration 6 to be less then the energy consumption for configuration 3 and 7, the power consumption must be reduced by at least 34%.

Chapter 11

Conclusions

In this thesis, a Reed-Solomon encoder and decoder have been designed and synthesized. The designs have been evaluated with respect to energy consumption. To find an energy efficient Reed-Solomon design, different implementation techniques have been compared and discussed. The Reed-Solomon decoder has been the main focus of this thesis, as it is the most complicated part of the Reed-Solomon system and has the most potential for reduction of energy consumption.

A compact RS(31, 27) encoder was implemented using constant multipliers to reduce the area usage, and synthesized with clock gating to reduce the power consumption. For encoders with low activity, the energy consumption can be reduced with $20 - 60\%$ by using clock gating.

Different techniques were evaluated to see which one gave the best results with regards to energy consumption of the decoder.

The energy consumption of the decoder can be reduced by a small amount, if only half of the syndrome values are calculated when checking for errors in a codeword. If the first half of the syndrome do not equal zero, rest of the syndrome have to be calculated. This increases the decoding time, and the energy consumption is therefore increased by 22% when decoding codewords with errors. This implementation technique should therefore only be used in systems that rarely have errors.

By using two-parallel syndrome cells and pipelining the Chien search, Forney and error correction module, the energy consumption can be reduced with 36% compared to a decoder with the same parallel syndrome cells and a full serial implementation of the Chien search, Forney and error correction module. The pipelined design with two-parallel syndrome cells, had a 7% lower energy consumption compared a pipelined design with standard syndrome cells.

Chapter 12

Further Work

The decoding time can be reduced by implementing more parallelism in the Reed-Solomon decoder. Both the Chien search and Forney method can be implemented with a parallel structure [21]. This will reduce the number of clock cycles. For a full parallel implementation of the decoder, the Euclidean algorithm could be used. This algorithm calculates the error locator and error evaluator polynomials in parallel. It would also be interesting to see how the energy consumption of this algorithm compares to the Berlekamp-Massey algorithm.

In order to possibly reduce the energy consumption, reuse of multipliers and adders in the decoder should be explored. All the modules in the decoder have separate multipliers and adders. Many of these modules are performing calculations only a short time period during decoding of a codeword. The multipliers and adders could therefore be shared between several of the modules. This could possibly reduce the total area and the power consumption for the decoder.

To get more relevant synthesis results and power estimations for the Reed-Solomon design, a 90nm or 180nm library should be used when synthesizing. This is the CMOS technology used by Energy Micro.

References

[1] J. C. Moreira and P. G. Farrell, *Essential Of Error-Control Coding.* John Wiley & Sons Ltd, 2006.

[2] A. P. Chandrakasan, S. Sheng and R. W. Brodersen, *Low-Power CMOS Digital Design.* IEEE Journal of solid-state circuit, Vol. 27, No. 4, pp. 473-484, 1992.

[3] C. Piguet, A. Amara and P. Royannez, *Low-Power CMOS Circuits - Technology, logic design and CAD tools, Chapter 11: VHDL for Low Power.* CRC Press Taylor & Francis Group, 2006.

[4] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh and M. Papaefthymiou, *Precomputation-Based Sequential Logic Optimization for Low Power.* IEEE Transactions on very large scale integration (VLSI) systems, Vol. 2, No. 4, 1994.

[5] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications.* Wiley-IEEE Press, 1994.

[6] I. S. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields.* SIAM Journal of Applied Mathematics, Volume 8, No 2, pp. 300-304, 1960.

[7] C.K.P Clarke, *Reed-Solomon Error Correction.* BBC R&D White paper, WHP 031, July 2002.

[8] M. Purser, *Introduction to error-correcting codes.* Artech House, Boston, London, 1995.

[9] Number of primitive polynomials of degree n over GF(2) - OEIS, Date: 6th of May 2013
`http://oeis.org/A011260`

[10] A. Raghupathy and K. J. R. Lui, *Algorithm-Based Low-Power/High-Speed Reed-Solomon Decoder Design.* IEEE Transactions on circuits and systems II: Analog and Digital Signal Processing, Vol. 47, NO. 11, November 2000.

[11] A. Kumar and S. Sawitzki, *High-Throughput and Low-Power Architectures for Reed Solomon Decoder.* IEEE Conference Record Thirty-Ninth Asilomar Conference on Signals, Systems & Computers, pp. 990-994, 2005.

[12] A. Genser, C. Bachmann, C. Steger, J. Hulzink and M. Berekovic, *Low-Power ASIP Architecture Exploration and Optimization for Reed-Solomon Processing.* 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 177-182, 2009.

[13] A. Al Azad, M. Huq and I. R. Rokon, *Efficient Hardware Implementation of Reed Solomon Encoder and Decoder in FPGA using Verilog.* International Conference on Advancements in Electronics and Power Engineering (ICAEPE'2011) Bangkok Dec., 2011.

[14] H. Chia Chang and C. Shung, *New Serial Architecture for the Berlekamp-Massey Algorithm.* IEEE Transactions on Communications, vol.47, no.4, pp. 481-483, Apr 1999

[15] R. Chien, *Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes.* IEEE Transactions on Information Theory, Volume 10, Issue 4, pp. 357-363, 1964.

[16] G. Forney, *On decoding BCH codes.* IEEE Transactions on Information Theory, Volume: 11 , Issue: 4 , pp. 549-557, 1965.

[17] H. Wu, M. A. Hasan, I. F. Blake and S. Gao, *Finite Field Multiplier Using Redundant Representation.* IEEE Transaction on computers, Vol. 51, NO. 11, November 2002.

[18] J. P Deschamps, J. L. Imaña and G. D. Sutter, *Hardware Implementation of Finite-Field Arithmetic.* The McGraw-Hill Companies, Inc., pp. 163-231, 2009.

[19] K. C. C. Wai and S. J. Yang, *Field Programmable Gate Array Implementation of Reed-Solomon Code, RS(255,239).* (Poster Paper) in Proceedings of 2nd IEEE Upstate NY Workshop on Communications and Networking, Rochester, NY., November, 2005.

[20] S. S. Shah, S. Yaqub and F. Suleman, *Self-correcting codes conquer noise Part 2: Reed-Solomon codecs.* EDN Magazine, March 15, 2001.

[21] S. Lee, C. Choi, and H. Lee, *Two-parallel Reed-Solomon Based FEC Architecture for Optical Communications.* IEICE Electronics Express, Vol. 5, No. 10, pp. 374-380, May, 2008.

[22] H. C. Chang, C. C. Lin and C. Y. Lee, *A Low Power Reed-Solomon Decoder For STM-16 Optical Communications.* 2002 IEEE Asia-Pacific Conference on ASIC. Proceedings, pp. 351-354, 2002.

[23] Addition and Multiplication Tables for Galois Fields $GF(2^5)$ Date: 30th of April 2013
`http://www.ee.unb.ca/cgi-bin/tervo/galois3.pl?p=6&C=1&D=1&A=1`

[24] FreePDK45nm, Date: 31th of May 2013
`http://vlsiarch.ecen.okstate.edu/?page_id=12#`

[25] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh and R. Jenkal *FreePDK: An Open-Source Variation-Aware Design Kit.* MSE '07. IEEE International Conference pp. 173-174, 3-4 June 2007

[26] Energy Micro AS - Products, Date: 15th of February 2013
`http://www.energymicro.com/products/`

Appendix A

Galois Field

A.1 Galois Field Representation

A.1.1 Field Elements GF(16)

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

Table A.1: Field elements for GF(16) with $p(x) = x^4 + x + 1$

A.1.2 Field Elements GF(32)

Index form	Polynomial form	Binary form	Decimal form
0	0	00000	0
α^0	1	00001	1
α^1	α	00010	2
α^2	α^2	00100	4
α^3	α^3	01000	8
α^4	α^4	10000	16
α^5	$\alpha^2 + 1$	00101	5
α^6	$\alpha^3 + \alpha$	01010	10
α^7	$\alpha^4 + \alpha^2$	10100	20
α^8	$\alpha^3 + \alpha^2 + 1$	01101	13
α^9	$\alpha^4 + \alpha^3 + \alpha$	11010	26
α^{10}	$\alpha^4 + 1$	10001	17
α^{11}	$\alpha^2 + \alpha + 1$	00111	7
α^{12}	$\alpha^3 + \alpha^2 + \alpha$	01110	14
α^{13}	$\alpha^4 + \alpha^3 + \alpha^2$	11100	28
α^{14}	$\alpha^4 + \alpha^3 + \alpha^2 + 1$	11101	29
α^{15}	$\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1$	11111	31
α^{16}	$\alpha^4 + \alpha^3 + \alpha + 1$	11011	27
α^{17}	$\alpha^4 + \alpha + 1$	10011	19
α^{18}	$\alpha + 1$	00011	3
α^{19}	$\alpha^2 + \alpha$	00110	6
α^{20}	$\alpha^3 + \alpha^2$	01100	12
α^{21}	$\alpha^4 + \alpha^3$	11000	24
α^{22}	$\alpha^4 + \alpha^2 + 1$	10101	21
α^{23}	$\alpha^3 + \alpha^2 + \alpha + 1$	01111	15
α^{24}	$\alpha^4 + \alpha^3 + \alpha^2 + \alpha$	11110	30
α^{25}	$\alpha^4 + \alpha^3 + 1$	11001	25
α^{26}	$\alpha^4 + \alpha^2 + \alpha + 1$	10111	23
α^{27}	$\alpha^3 + \alpha + 1$	01011	11
α^{28}	$\alpha^4 + \alpha^2 + \alpha$	10110	22
α^{29}	$\alpha^3 + 1$	01001	9
α^{30}	$\alpha^4 + \alpha$	10010	18

Table A.2: Field elements for GF(32) with $p(x) = x^5 + x^2 + 1$

Appendix B

Examples

B.1 Implementation Examples

B.1.1 Constant Multiplier

Multiplying with a constant value of 10. $10 = \alpha^6 = \alpha^3 + \alpha$ The 5-bit equivalents for α^7, α^6 and α^5 from table A.2 are:

$$\begin{aligned} \alpha^5 &= \alpha^2 + 1 \\ \alpha^6 &= \alpha^3 + \alpha \\ \alpha^7 &= \alpha^4 + \alpha^2 \end{aligned}$$

Produce a shifted version of the input.

	α^7	α^6	α^5		α^4	α^3	α^2	α^1	α^0
$\times\alpha^3$	a_4	a_3	a_2		a_1	a_0	0	0	0
$\times\alpha$	———	———	a_4		a_3	a_2	a_1	a_0	0
			$a_2 + a_4$	$\rightarrow$	0	0	$a_2 + a_4$	0	$a_2 + a_4$
		a_3	$\longrightarrow$		0	a_3	0	a_3	0
	a_4		$\longrightarrow$		a_4	0	a_4	0	0
					$a_1 + a_3 + a_4$	$a_0 + a_2 + a_3$	$a_1 + a_2$	$a_0 + a_3$	$a_2 + a_4$

Add the values in column α^4, α^3, α^2, α^1 and α^0 to give the input bit contribution for each output bit. The addition is implemented using XOR gates.

$$\begin{aligned} c_0 &= a_2 + a_4 \\ c_1 &= a_0 + a_3 \\ c_2 &= a_1 + a_2 \\ c_3 &= a_0 + a_2 + a_3 \\ c_4 &= a_1 + a_3 + a_4 \end{aligned}$$

B.1.2 Constructing a Full Multiplier

The 5-bit full multiplier used in this thesis was constructed in the following way:

STEP 1: Use equation 5.3 and the two polynomials $a(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$ and $b(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4$ to determine the coefficients of $d(x)$.

$$
\begin{aligned}
d_0 &= (a_0 + b_0) \\
d_1 &= (a_0 + b_1) \oplus (a_1 + b_0) \\
d_2 &= (a_0 + b_2) \oplus (a_1 + b_1) \oplus (a_2 + b_0) \\
d_3 &= (a_0 + b_3) \oplus (a_1 + b_2) \oplus (a_2 + b_1) \oplus (a_3 + b_0) \\
d_4 &= (a_0 + b_4) \oplus (a_1 + b_3) \oplus (a_2 + b_2) \oplus (a_3 + b_1) \oplus (a_4 + b_0) \\
d_5 &= (a_1 + b_4) \oplus (a_2 + b_3) \oplus (a_3 + b_2) \oplus (a_4 + b_1) \\
d_6 &= (a_2 + b_4) \oplus (a_3 + b_3) \oplus (a_4 + b_2) \\
d_7 &= (a_3 + b_4) \oplus (a_4 + b_3) \\
d_8 &= (a_4 + b_4)
\end{aligned}
$$

STEP 2: Then use the relationships of table A.2 in appendix A.1.2 to map the coefficients $d_m - d_{2m-2}$ into the field.

The relationships for α^5, α^6, α^7 and α^8.

$$
\begin{aligned}
\alpha^5 &= \alpha^2 + 1 \\
\alpha^6 &= \alpha^3 + \alpha \\
\alpha^7 &= \alpha^4 + \alpha^2 \\
\alpha^8 &= \alpha^3 + \alpha^2 + 1
\end{aligned}
$$

Resulting polynomial $c(x)$.

$$
\begin{aligned}
c_0 &= d_0 \oplus d_5 \oplus d_8 \\
c_1 &= d_1 \oplus d_6 \\
c_2 &= d_2 \oplus d_5 \oplus d_7 \oplus d_8 \\
c_3 &= d_3 \oplus d_6 \oplus d_8 \\
c_4 &= d_4 \oplus d_7
\end{aligned}
$$

B.2 Decoding Example

Decoding example with the (31, 27) Reed-Solomon code.

Introducing two errors in the 6th (x^{25} term) and 19th (x^{12} term) symbols of a encoded message. The two errors, with value 19 and 24, can be written as an error polynomial as shown below:

$E(x) = 19x^{25} + 24x^{12}$

The received message then becomes:

$$
\begin{aligned}
R(x) &= (26x^{30} + 28x^{29} + 4x^{28} + 29x^{27} + 20x^{26} + 3x^{25} + 8x^{24} + 17x^{23} + 30x^{22} + 30x^{21} + 5x^{20} \\
&\quad + 31x^{19} + 30x^{18} + 15x^{17} + 25x^{16} + 4x^{15} + 13x^{14} + 29x^{13} + 25x^{12} + 30x^{11} + 20x^{10} \\
&\quad + x^9 + 27x^8 + 29x^7 + 21x^6 + 16x^5 + 0 + 14x^3 + 8x^2 + 11x + 11) + (19x^{25} + 24x^{12}) \\
&= 26x^{30} + 28x^{29} + 4x^{28} + 29x^{27} + 20x^{26} + 16x^{25} + 8x^{24} + 17x^{23} + 30x^{22} + 30x^{21} + 5x^{20} \\
&\quad + 31x^{19} + 30x^{18} + 15x^{17} + 25x^{16} + 4x^{15} + 13x^{14} + 29x^{13} + x^{12} + 30x^{11} + 20x^{10} \\
&\quad + x^9 + 27x^8 + 29x^7 + 21x^6 + 16x^5 + 0 + 14x^3 + 8x^2 + 11x + 11
\end{aligned}
$$

The symbols can then be written as shown below, where the values written in bold are the ones with an error.

26 28 4 29 20 **16** 8 17 30 30 5 31 30 15 25 4 13 29 **1** 30 20 1 27 29 21 16 0 14 8 11 11

To calculate the syndrome value S_i, the corresponding root α^i is substituted in for x. The results are then added together. This can be written as a series intermediate steps as shown:

S_0 with $\alpha^0 = 1$	S_1 with $\alpha^1 = 2$	S_2 with $\alpha^2 = 4$	S_3 with $\alpha^3 = 8$
$(0+26) \times 1 = 26$	$(0+26) \times 2 = 17$	$(0+26) \times 4 = 7$	$(0+26) \times 8 = 14$
$(26+28) \times 1 = 6$	$(17+28) \times 2 = 26$	$(7+28) \times 4 = 3$	$(14+28) \times 8 = 4$
$(6+4) \times 1 = 2$	$(26+4) \times 2 = 25$	$(3+4) \times 4 = 28$	$(4+4) \times 8 = 0$
$(2+29) \times 1 = 31$	$(25+29) \times 2 = 8$	$(28+29) \times 4 = 4$	$(0+29) \times 8 = 19$
$(31+20) \times 1 = 11$	$(8+20) \times 2 = 29$	$(4+20) \times 4 = 10$	$(19+20) \times 8 = 29$
$(11+16) \times 1 = 27$	$(29+16) \times 2 = 26$	$(10+16) \times 4 = 7$	$(29+16) \times 8 = 7$
$(27+8) \times 1 = 19$	$(26+8) \times 2 = 1$	$(7+8) \times 4 = 25$	$(7+8) \times 8 = 23$
$(19+17) \times 1 = 2$	$(1+17) \times 2 = 5$	$(25+17) \times 4 = 5$	$(23+17) \times 8 = 21$
$(2+30) \times 1 = 28$	$(5+30) \times 2 = 19$	$(5+30) \times 4 = 3$	$(21+30) \times 8 = 18$
$(28+30) \times 1 = 2$	$(19+30) \times 2 = 26$	$(3+30) \times 4 = 27$	$(18+30) \times 8 = 15$
$(2+5) \times 1 = 7$	$(26+5) \times 2 = 27$	$(27+5) \times 4 = 23$	$(15+5) \times 8 = 26$
$(7+31) \times 1 = 24$	$(27+31) \times 2 = 8$	$(23+31) \times 4 = 5$	$(26+31) \times 8 = 13$
$(24+30) \times 1 = 6$	$(8+30) \times 2 = 9$	$(5+30) \times 4 = 3$	$(13+30) \times 8 = 12$
$(6+15) \times 1 = 9$	$(9+15) \times 2 = 12$	$(3+15) \times 4 = 21$	$(12+15) \times 8 = 24$
$(9+25) \times 1 = 16$	$(12+25) \times 2 = 15$	$(21+25) \times 4 = 21$	$(24+25) \times 8 = 8$
$(16+4) \times 1 = 20$	$(15+4) \times 2 = 22$	$(21+4) \times 4 = 14$	$(8+4) \times 8 = 15$
$(20+13) \times 1 = 25$	$(22+13) \times 2 = 19$	$(14+13) \times 4 = 12$	$(15+13) \times 8 = 16$
$(25+29) \times 1 = 4$	$(19+29) \times 2 = 28$	$(12+29) \times 4 = 14$	$(16+29) \times 8 = 7$
$(4+1) \times 1 = 5$	$(28+1) \times 2 = 31$	$(14+1) \times 4 = 25$	$(7+1) \times 8 = 21$
$(5+30) \times 1 = 27$	$(31+30) \times 2 = 2$	$(25+30) \times 4 = 28$	$(21+30) \times 8 = 18$
$(27+20) \times 1 = 15$	$(2+20) \times 2 = 9$	$(28+20) \times 4 = 5$	$(18+20) \times 8 = 21$
$(15+1) \times 1 = 14$	$(9+1) \times 2 = 16$	$(5+1) \times 4 = 16$	$(21+1) \times 8 = 17$
$(14+27) \times 1 = 21$	$(16+27) \times 2 = 22$	$(16+27) \times 4 = 9$	$(17+27) \times 8 = 26$
$(21+29) \times 1 = 8$	$(22+29) \times 2 = 22$	$(9+29) \times 4 = 26$	$(26+29) \times 8 = 29$
$(8+21) \times 1 = 29$	$(22+21) \times 2 = 6$	$(26+21) \times 4 = 25$	$(29+21) \times 8 = 10$
$(29+16) \times 1 = 13$	$(6+16) \times 2 = 9$	$(25+16) \times 4 = 1$	$(10+16) \times 8 = 14$
$(13+0) \times 1 = 13$	$(9+0) \times 2 = 18$	$(1+0) \times 4 = 4$	$(14+0) \times 8 = 31$
$(13+14) \times 1 = 3$	$(18+14) \times 2 = 29$	$(4+14) \times 4 = 13$	$(31+14) \times 8 = 28$
$(3+8) \times 1 = 11$	$(29+8) \times 2 = 15$	$(13+8) \times 4 = 20$	$(28+8) \times 8 = 17$
$(11+11) \times 1 = 0$	$(15+11) \times 2 = 8$	$(20+11) \times 4 = 19$	$(17+11) \times 8 = 14$
$(0+11) = 11$	$(8+11) = 3$	$(19+11) = 24$	$(14+11) = 5$
$S_0 = 11$	$S_1 = 3$	$S_2 = 24$	$S_3 = 5$

Using the inversionless Berlekamp-Massey shown in algorithm 2 and equation 5.5 and 5.6, we find the error locator polynomial. The syndrome values in the inver-

sionless Berlekamp-Massey algorithm starts with S_1 and not S_0. This however, do not change the calculation method or results. We therefore change the notation and use the following:
$S_1 = S_0 = 11 \quad S_2 = S_1 = 3 \quad S_3 = S_2 = 24 \quad S_4 = S_3 = 5$

Initial condition:
$D^{(-1)} = 0 \qquad \delta = 1 \qquad \sigma^{(-1)}(x) \qquad \tau^{(-1)}(x) = 1 \qquad \Delta^{(0)} = S_1$

$i = 0$

$$\begin{aligned} \sigma_0^{(0)} &= \delta \times \sigma_0^{(-1)} &&= 1 \times 1 = 1 \\ \sigma_1^{(0)} &= \delta \times \sigma_1^{(-1)} + \Delta^{(0)} \times \tau_0^{(-1)} &&= 1 \times 0 + 11 \times 1 = 11 \\ \sigma_2^{(0)} &= \delta \times \sigma_2^{(-1)} + \Delta^{(0)} \times \tau_1^{(-1)} &&= 1 \times 0 + 11 \times 0 = 0 \end{aligned}$$

$\sigma^{(0)}(x) = 11x + 1$

$$\begin{aligned} \Delta_0^{(1)} &= 0 \\ \Delta_1^{(1)} &= \Delta_0^{(1)} + S_2 \times \sigma_0^{(0)} &&= 0 + 3 \times 1 = 3 \\ \Delta_2^{(1)} &= \Delta_1^{(1)} + S_1 \times \sigma_1^{(0)} &&= 3 + 11 \times 11 = 12 \end{aligned}$$

$\Delta^{(1)} = 12$

Updating values: $\Delta^{(0)} \neq 0$ and $2D^{(-1)} < 1$ so..
$D^{(0)} = 0 + 1 - D^{(-1)=1} \qquad \delta = \Delta^{(0)} = 11 \qquad \tau^{(0)}(x) = \sigma^{-1}(x) = 1$

$i = 1$

$$\begin{aligned} \sigma_0^{(1)} &= \delta \times \sigma_0^{(0)} &&= 11 \times 1 = 11 \\ \sigma_1^{(1)} &= \delta \times \sigma_1^{(0)} + \Delta^{(1)} \times \tau_0^{(0)} &&= 11 \times 11 + 12 \times 1 = 3 \\ \sigma_2^{(1)} &= \delta \times \sigma_2^{(0)} + \Delta^{(1)} \times \tau_1^{(0)} &&= 11 \times 0 + 1 \times 0 = 0 \end{aligned}$$

$\sigma^{(1)}(x) = 3x + 11$

$$\begin{aligned} \Delta_0^{(2)} &= 0 \\ \Delta_1^{(2)} &= \Delta_0^{(2)} + S_3 \times \sigma_0^{(1)} &&= 0 + 24 \times 11 = 19 \\ \Delta_2^{(2)} &= \Delta_1^{(2)} + S_2 \times \sigma_1^{(1)} &&= 19 + 3 \times 3 = 22 \end{aligned}$$

$\Delta^{(2)} = 22$

Updating values: $\Delta^{(1)} \neq 0$ and $2D^{(0)} \geq 1$ so..
$D^{(1)} = D^{(0)=1} \qquad \delta = \delta = 11 \qquad \tau^{(1)}(x) = \tau^{(0)}(x) = x$

$$
\begin{array}{lllll}
i=2 & \sigma_0^{(2)} = & \delta \times \sigma_0^{(1)} & = & 11 \times 11 = 15 \\
 & \sigma_1^{(2)} = & \delta \times \sigma_1^{(1)} + \Delta^{(2)} \times \tau_0^{(1)} & = & 11 \times 3 + 22 \times 0 = 29 \\
 & \sigma_2^{(2)} = & \delta \times \sigma_2^{(1)} + \Delta^{(2)} \times \tau_1^{(1)} & = & 11 \times 0 + 22 \times 1 = 22
\end{array}
$$

$$\sigma^{(2)}(x) = 22x^2 + 29x + 15$$

$$
\begin{array}{lllll}
 & \Delta_0^{(3)} = & 0 & & \\
 & \Delta_1^{(3)} = & \Delta_0^{(3)} + S_4 \times \sigma_0^{(2)} & = & 0 + 5 \times 15 = 12 \\
 & \Delta_2^{(3)} = & \Delta_1^{(3)} + S_3 \times \sigma_1^{(2)} & = & 22 + 24 \times 29 = 6 \\
 & \Delta_3^{(3)} = & \Delta_2^{(3)} + S_2 \times \sigma_1^{(2)} & = & 6 + 3 \times 22 = 25
\end{array}
$$

$$\Delta^{(3)} = 25$$

Updating values: $\Delta^{(2)} \neq 0$ and $2D^{(1)} < 3$ so..
$D^{(2)} = 2 + 1 - D^{(1)=2}$ $\delta = \Delta^2 = 22$ $\tau^{(2)}(x) = \sigma^{(1)}(x) = 3x + 11$

$$
\begin{array}{lllll}
i=3 & \sigma_0^{(3)} = & \delta \times \sigma_0^{(2)} & = & 22 \times 15 = 12 \\
 & \sigma_1^{(3)} = & \delta \times \sigma_1^{(2)} + \Delta^{(3)} \times \tau_0^{(2)} & = & 22 \times 29 + 25 \times 11 = 31 \\
 & \sigma_2^{(3)} = & \delta \times \sigma_2^{(2)} + \Delta^{(3)} \times \tau_1^{(2)} & = & 22 \times 22 + 25 \times 3 = 23
\end{array}
$$

$$\sigma^{(3)}(x) = 23x^2 + 31x + 12$$

Then, using equation 4.20 and 5.7 we find the error evaluator polynomial.

$$
\begin{array}{lllll}
i=0 & \Omega_0^{(0)} = & S_1 \times \sigma_0 & = & 11 \times 12 = 27 \\
 & \Omega_0^{(1)} = & \Omega_0^{(0)} + 0 & = & 27 + 0 = 27 \\
 & & & & \\
i=1 & \Omega_1^{(0)} = & S_2 \times \sigma_0 & = & 3 \times 12 = 20 \\
 & \Omega_1^{(1)} = & \Omega_0^{(0)} + S_1 \times \sigma_1 & = & 20 + 11 \times 31 = 19
\end{array}
$$

$$\Omega_0 = 27 \qquad \Omega_1 = 19$$

The error locator polynomial and error evaluator polynomial are then:
$\sigma(x) = 23x^2 + 31x + 12$
$\Omega(x) = 19x + 27$

To find the error positions, the Chien search is used. Each value of the field α^i is substituted into the error locator polynomial. This can be done by multiplying each coefficient of the error locator polynomial with the corresponding power of α and then adding the results to produce the sum, as shown in table B.1. So, σ_2 is multiplied by α^2, σ_1 by α and σ_0 by α^0. Each row in table B.1 can be obtained by

multiplying the previous row with the corresponding power of α. If a sum value becomes zero, there is an error at that position.

	×4	×2	×1	
x	x^2 **term**	x **term**	**unity**	**sum**
α^{-30}	22	27	12	1
α^{-29}	18	19	12	13
α^{-28}	2	3	12	13
α^{-27}	8	6	12	2
α^{-26}	5	12	12	5
α^{-25}	20	24	12	**0**
α^{-24}	26	21	12	3
α^{-23}	7	15	12	4
α^{-22}	28	30	12	14
α^{-21}	31	25	12	10
α^{-20}	19	23	12	8
α^{-19}	6	11	12	1
α^{-18}	24	22	12	2
α^{-17}	15	9	12	10
α^{-16}	25	18	12	7
α^{-15}	11	1	12	6
α^{-14}	9	2	12	7
α^{-13}	1	4	12	9
α^{-12}	4	8	12	**0**
α^{-11}	16	16	12	12
α^{-10}	10	5	12	3
α^{-9}	13	10	12	11
α^{-8}	17	20	12	9
α^{-7}	14	13	12	15
α^{-6}	29	26	12	11
α^{-5}	27	17	12	6
α^{-4}	3	7	12	8
α^{-3}	12	14	12	14
α^{-2}	21	28	12	5
α^{-1}	30	29	12	15
α^{0}	23	31	12	4

Table B.1: Chien search example

Having found the error positions, we use the Forney method to find the error values. The error values are found using equation 4.30. The derivative of $\sigma(x)$ is obtained by setting all the even powers of x to zero and dividing by x, where $x = X_j^{-1}$. This gives:

$$\sigma'(X_j^{-1}) = 31X_j^{-1} / \ X_j^{-1} = 31$$

The error positions where found to be in the 6th (x^{25} term) and 19th (x^{12} term) symbols. By setting $X_j = \alpha^{25}$ and $X_j = \alpha^{12}$ in equation 4.30, we can calculate the corresponding error values. The inverse values are needed when using the Forney method. So, the inverse value of α^{25} is $\alpha^{-25} = \alpha^6 = 10$ and the inverse values of α^{12} is $\alpha^{-12} = \alpha^{19} = 6$.

For the first position, we get:

$$Y_j = \alpha^{25}\frac{19\alpha^{-25} + 27}{31} = 25\frac{19\alpha^{6} + 27}{31} = 25\frac{19 \times 10 + 27}{31} = 25\frac{20}{31}$$

Finite field division can be performed my multiplying with the inverse of the divisor. The inverse of 31 is $\alpha^{-15} = \alpha^{16} = 27$:

$25(20 \times 27) = 19$

For the second position, we get:

$$Y_j = \alpha^{12}\frac{19\alpha^{-12} + 27}{31} = 14\frac{19\alpha^{19} + 27}{31} = 14\frac{19 \times 6 + 27}{31} = 14\frac{30}{31}$$

$14(30 \times 27) = 24$

The error values are added to the received codeword to produce the corrected message.

$$\begin{aligned} C(x) &= (26x^{30} + 28x^{29} + 4x^{28} + 29x^{27} + 20x^{26} + 16x^{25} + 8x^{24} + 17x^{23} + 30x^{22} + 30x^{21} + 5x^{20} \\ &\quad + 31x^{19} + 30x^{18} + 15x^{17} + 25x^{16} + 4x^{15} + 13x^{14} + 29x^{13} + x^{12} + 30x^{11} + 20x^{10} \\ &\quad + x^9 + 27x^8 + 29x^7 + 21x^6 + 16x^5 + 0 + 14x^3 + 8x^2 + 11x + 11) + (19x^{25} + 24x^{12}) \\ &= 26x^{30} + 28x^{29} + 4x^{28} + 29x^{27} + 20x^{26} + 3x^{25} + 8x^{24} + 17x^{23} + 30x^{22} + 30x^{21} + 5x^{20} \\ &\quad + 31x^{19} + 30x^{18} + 15x^{17} + 25x^{16} + 4x^{15} + 13x^{14} + 29x^{13} + 25x^{12} + 30x^{11} + 20x^{10} \\ &\quad + x^9 + 27x^8 + 29x^7 + 21x^6 + 16x^5 + 0 + 14x^3 + 8x^2 + 11x + 11 \end{aligned}$$

The corrected message is then:
26 28 4 29 20 3 8 17 30 30 5 31 30 15 25 4 13 29 25 30 20 1 27 29 21 16 0 14 8 11 11

Appendix C

Test Vectors

C.1 Encoder Test Vectors

C.1.1 Simulation Test Vectors

Input symbols	Parity symbols
24 23 12 20 5 22 1 8 1 3 26 22 10 30 1 14 12 24 25 5 15 14 20 22 24 16 0	27 6 6 0
8 21 20 5 3 15 30 10 18 7 24 8 16 22 28 30 17 4 4 8 26 8 26 7 29 4 0	31 9 14 2
11 6 8 19 15 11 26 18 17 29 9 24 24 12 18 2 1 16 24 29 4 18 15 0 10 8 0	17 2 29 2
5 25 9 16 5 19 8 20 22 23 14 2 7 29 4 26 17 31 2 14 3 30 0 24 26 28 0	21 9 19 8
27 2 12 8 25 13 29 5 8 4 4 27 18 17 4 27 19 11 16 12 2 7 3 5 7 12 0	17 29 5 27
13 1 28 30 15 15 10 28 11 3 24 12 7 12 3 4 30 30 18 1 7 11 26 0 1 24 0	30 25 18 1
5 20 23 20 14 17 9 23 6 21 5 11 20 24 2 29 24 15 13 14 9 16 16 26 25 20 0	30 31 11 29
20 12 25 17 11 30 28 17 19 18 6 9 15 7 27 6 7 5 7 13 9 29 13 5 28 24 0	1 25 15 20
31 14 3 8 13 19 8 19 22 7 3 9 10 13 16 2 8 25 0 29 23 15 18 7 14 28 0	17 28 25 22
30 17 16 7 15 19 21 12 11 31 1 28 29 25 3 8 10 21 4 23 3 20 15 24 22 20 0	2 28 12 7
28 28 10 22 6 0 23 16 15 28 19 19 27 25 18 5 7 28 0 15 5 31 22 16 15 12 0	18 1 22 21
1 21 1 2 16 3 26 26 23 4 21 16 31 20 25 14 13 26 2 4 5 12 26 25 1 16 0	15 21 19 13
12 16 13 21 20 9 13 0 31 5 3 11 6 15 10 30 29 1 23 8 13 17 30 13 31 4 0	25 25 16 28
9 22 21 17 22 21 5 4 31 5 1 17 28 21 6 11 14 31 5 27 20 12 6 13 15 8 0	7 10 27 26
3 18 7 12 18 8 9 19 8 26 31 23 11 18 3 29 28 26 8 19 0 13 10 5 5 12 0	20 22 26 9
13 3 19 15 22 22 20 1 2 10 16 20 13 26 22 30 17 10 3 19 24 13 2 8 4 8 0	8 29 15 22
8 14 16 14 28 16 30 20 30 7 21 9 21 22 2 8 7 21 27 11 24 21 0 19 12 20 0	14 24 11 11
29 0 14 13 14 24 10 25 15 1 5 23 15 4 10 19 6 23 7 29 8 24 6 9 2 4 0	4 30 24 5
18 21 17 13 20 20 21 20 30 6 22 7 3 19 14 14 21 24 11 21 13 26 26 8 19 16 0	28 4 27 21
18 17 27 8 10 3 30 20 15 20 17 20 17 23 16 31 6 3 3 2 12 14 11 24 20 28 0	17 2 9 3

Table C.1: Test vectors used for simulation of encoder

C.2 Decoder Test Vectors

C.2.1 Simulation Test Vectors

Input codeword	26 14 25 25 9 2 1 20 13 28 13 4 17 29 21 12 8 27 30 12 2 5 18 5 9 28 0 26 9 28 12
Decoded codeword	26 14 25 25 9 2 1 20 13 28 13 4 17 29 21 12 8 27 30 12 2 5 18 5 9 28 0 26 9 28 12

Table C.2: Test vector with zero errors

Codeword with error	9 19 14 9 26 18 2 8 5 17 6 24 23 30 24 2 30 31 19 30 10 28 0 15 4 28 0 2 0 15 17
Decoded codeword	9 19 14 9 26 18 2 8 5 17 6 15 23 30 24 2 30 31 19 30 10 28 0 15 4 28 0 2 0 15 17

Table C.3: Test vector with one error

Codeword with error	30 19 26 25 25 22 27 24 6 31 2 20 6 7 10 6 8 30 12 18 19 22 12 27 29 28 0 15 29 1 8
Decoded codeword	6 19 26 25 25 22 27 24 6 31 2 20 6 7 10 6 16 30 12 18 19 22 12 27 29 28 0 15 29 1 8

Table C.4: Test vector with two errors

Appendix D

Scripts

D.1 Matlab Scripts

D.1.1 Encoder Test Vector Script

```
%--------------------------------%
%                                %
%   RS Encoder used to generate  %
%   stimulus for simulation      %
%                                %
%--------------------------------%

home
clear all
%Script for creating 16Byte (128 bit) packets
%padded with zero before encoding to make get 27 symbols of information.

% Open new write file
% fileID = fopen('stimulus.txt', 'w');
% fileID2 = fopen('stimulus_encoded.txt', 'w');


% Setting up Reed-Solomon encoder
j = 2000;    % Number of test vectors to generate
m = 5;       % Number of bits in each symbol
n = 2^m-1;   % Codeword length
k = 27;      % Message length
max_value = 25;   % Full symbols with info, when sending 16 Byte (128 bit)
% 128/5 = 25,6 => 25 symbols and 3 bit


for i=1:j
    data = randi([0 n], max_value, 1); % 125 bit random values
    fileID = fopen('stimulus.txt', 'a');
    fileID2 = fopen('stimulus_encoded.txt', 'a');

    %this gives total of 135 bit
    data_full = data;
    data_full(26) = 28; %add data and pad with two zeroes (00)
    data_full(27) = 0;  %pad last symbol with zero

    % Write information symbols to file
    fprintf(fileID, '%d ', data_full);
    fprintf(fileID, '%d\r\n','');

```

```
    hEnc = comm.RSEncoder(n, k, 'BitInput', false); %Create RS Encoder
    hEnc.GeneratorPolynomialSource = 'Property';
    hEnc.GeneratorPolynomial       = rsgenpoly(n,k,37,0); % Prim.Poly = 37 and A^b
        where b=0
    hEnc.PrimitivePolynomialSource = 'Property';
    hEnc.PrimitivePolynomial       = [1 0 0 1 0 1]; % Prim.Poly = 37

    encodedData = step(hEnc, data_full); % Encode data, adding parity

    % Write encoded data to file
    fprintf(fileID2, '%d ', encodedData);
    fprintf(fileID2, '%d\r\n','');

    fclose(fileID); % Close file at end of loop
    fclose(fileID2); % Close file at end of loop
end
```

D.1.2 Decoder Test Vector Script

```
%--------------------------------%
%                                %
%   RS decoder script            %
%   used to generate             %
%   stimulus for simulation      %
%                                %
%--------------------------------%


home
clear all

m = 5;       % Number of bits in each symbol
n = 2^m-1;   % Codeword length
k = 27;      % Message length
t = (n-k)/2; % Error-correction capability of the code
nw = 1; % Number of words to process each time

%Decoder test gen
for i=1:2000
    data = randi([0 n], k, 1); % 125 bit random values
    fileID = fopen('input.txt', 'a');
    fileID2 = fopen('encoded.txt', 'a');
    fileID3 = fopen('noisy.txt', 'a');
    fileID4 = fopen('decoded.txt', 'a');
    fileID5 = fopen('noise.txt', 'a');

    data_full = data;

    % Write information symbols to file
    fprintf(fileID, '%d ', data_full);
    fprintf(fileID, '%d\r\n','');

    %fprintf(fileID2, '%s', bindata);
    %fprintf(fileID2, '%s\r\n','');

    % Setup for RS encoder
    hEnc = comm.RSEncoder(n, k, 'BitInput', false); %Create RS Encoder
    hEnc.GeneratorPolynomialSource = 'Property';
    hEnc.GeneratorPolynomial       = rsgenpoly(n,k,37,0); % Prim.Poly = 37 and A^b
        where b=0
    hEnc.PrimitivePolynomialSource = 'Property';
    hEnc.PrimitivePolynomial       = [1 0 0 1 0 1]; % Prim.Poly = 37


    %Setup for RS decoder
```

```
    hDec = comm.RSDecoder(n, k, 'BitInput', false); %Create RS Decoder
    hDec.GeneratorPolynomialSource = 'Property';
    hDec.GeneratorPolynomial       = rsgenpoly(n,k,37,0); % Prim.Poly = 37 and A^b
        where b=0
    hDec.PrimitivePolynomialSource = 'Property';
    hDec.PrimitivePolynomial       = [1 0 0 1 0 1]; % Prim.Poly = 37

    encodedData = step(hEnc, data_full); % Encode data, adding parity

    % Write encoded data to file
    fprintf(fileID2, '%d ', encodedData);
    fprintf(fileID2, '%d\r\n','');

    % Generate noise
    t = randi([0 2], 1, 1);
    noise = (1+randi([0 n-1],nw,n)).*randerr(nw,n,t); % t errors per codeword
    noisy = noise';
    noisy = noisy(:);
    cnoisy = gf(encodedData,m) + noisy; % Add noise to the code under gf(m) arithmetic.

    fprintf(fileID5, '%d ', noisy);
    fprintf(fileID5, '%d\r\n','');

    %convert the GF object to a double so we can write it to file
    pcnoisy = double(cnoisy.x);

    fprintf(fileID3, '%d ', pcnoisy);
    fprintf(fileID3, '%d\r\n','');

    [dc nerrs] = step(hDec, cnoisy.x); % Decode the noisy code.

    fprintf(fileID4, '%d ', dc);
    fprintf(fileID4, '%d\r\n','');


    % Check that the decoding worked correctly.
    isequal(dc,data_full);
    nerrs; % Find out how many errors hDec corrected.

    fclose(fileID);  % Close file at end of loop
    fclose(fileID2); % Close file at end of loop
    fclose(fileID3); % Close file at end of loop
    fclose(fileID4); % Close file at end of loop
    fclose(fileID5); % close file at end of loop

end
```

D.2 Design Compiler and Power Compiler Scripts

D.2.1 Synopsys Design Compiler Setup File

```
# .synopsys_dc.setup file
# Define the target search path, technology library, symbol library
# and link library


# Define for 45nm FreePDK cell-library
set 45nm /home/sindredf/master_project/OSU_FreePDK/osu_freepdk_1.0/lib
set target45nm ${45nm}/files
# set synth_lib45nm ${45nm}/
# set target45nm ${45nm]/

```

```
# Define the libraries and search path
set search_path [concat $search_path ./SRC ./SYN/SCR ${target45nm} dw_foundation.sldb]
set target_library ${target45nm}/gscl45nm.db
set link_library [concat "*" $target_library]
#set symbol_library <path/file>

# Optinal lib uncomment if to be used
set synthetic_library dw_foundation.sldb

define_design_lib WORK -path ./SYN/WORK

# Define path directories for file locations
set source_path "./SRC/"
set script_path "./SYN/SCR/"
set log_path "./SYN/RPT/"
set ddc_path "./SYN/DDC/"
set netlist_path "./SYN/NETLIST/"
set saif_path "./SYN/SAIF/"
```

D.2.2 Constraints Script

```
# Create a clock with 25 ns period and 50% duty cycle (40Mhz clock)
create_clock -name "clk" -period 25 -waveform {0 12.5} {clk}

set_clock_uncertainty 0.1 clk
set_clock_latency 0.2 clk
set_clock_transition 0.1 clk
set_dont_touch_network clk

set_dont_touch_network [get_ports clk]
set_dont_touch_network [get_ports reset]
set_ideal_network reset
#set_ideal_network -no_propagate reset

set_load 0.1 [all_outputs]

set_input_delay 0.67 -clock clk [all_inputs]
set_output_delay 0.5 -clock clk [all_outputs]

#set_max_area <value>
```

D.2.3 Compile Script

```
#-----------------------------------------------#
#                                               #
#        Decoder design compile script          #
#                 CONFIG                        #
#                                               #
#                                               #
#                                               #
#-----------------------------------------------#

#Testing with 0, 1 and 2 errors
remove_design -all

analyze -library WORK -format verilog {./SRC/decoder_top.v ./SRC/adder.v
./SRC/AND_gate.v ./SRC/D_reg.v ./SRC/Chien_cell_W.v ./SRC/Chien_cell.v
./SRC/Chien_search.v ./SRC/Codeword_Store.v ./SRC/Error_corr.v
./SRC/Forney.v ./SRC/MUX.v ./SRC/Full_mult.v ./SRC/in_stage.v ./SRC/inverse_GF32.v
./SRC/Mult_x2_5bit.v ./SRC/Mult_x4_5bit.v ./SRC/Mult_x8_5bit.v ./SRC/MUX_1to2.v ./SRC/
    NOR.v
```

```
./SRC/Omega_calc.v ./SRC/BM_alg.v ./SRC/syndrome_calc_S0.v ./SRC/syndrome_calc_S1.v
./SRC/syndrome_calc_S2.v ./SRC/syndrome_calc_S3.v ./SRC/Syndrome_ctrl.v ./SRC/
    clk_gate.v}

elaborate decoder_top -library WORK

# Apply constraints
source ./SYN/SCR/constraints.tcl

compile -incremental

# Run report commands
echo "Decoder config \n" >> ./SYN/RPT/report_config.txt
echo "Report files" >> ./SYN/RPT/report_config.txt

echo "Config QOR REPORT\n" >> ./SYN/RPT/report_config.txt
report_qor >> ./SYN/RPT/report_config.txt

echo "Config AREA HIER\n" >> ./SYN/RPT/report_config.txt
report_area -hierarchy >> ./SYN/RPT/report_config.txt

echo "Config CELL REPORT\n" >> ./SYN/RPT/report_config.txt
report_cell >> ./SYN/RPT/report_config.txt


#----------------------#
# ZERO ERRORS          #
#----------------------#
# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error0.saif -instance_name decoder_top_tb/UUT -verbose

#power reports
echo "Config POWER REPORT with 0 errors\n" >> ./SYN/RPT/report_config.txt
report_power  >> ./SYN/RPT/report_config.txt

echo "Config POWER REPORT HIER with 0 errors\n" >> ./SYN/RPT/report_config.txt
report_power -hier >> ./SYN/RPT/report_config.txt

#----------------------#
# ONE ERROR            #
#----------------------#

# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error1.saif -instance_name decoder_top_tb/UUT -verbose

# Run report commands
#power reports
echo "----------------------------------------\n" >> ./SYN/RPT/report_config.txt
echo "Config POWER REPORT with 1 errors\n" >> ./SYN/RPT/report_config.txt
report_power  >> ./SYN/RPT/report_config.txt

echo "Config POWER REPORT HIER with 1 errors\n" >> ./SYN/RPT/report_config.txt
report_power -hier >> ./SYN/RPT/report_config.txt

#----------------------#
# TWO ERRORS           #
#----------------------#

# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error2.saif -instance_name decoder_top_tb/UUT -verbose

# Run report commands
#power reports
echo "----------------------------------------\n" >> ./SYN/RPT/report_config.txt
echo "Config POWER REPORT with 2 errors\n" >> ./SYN/RPT/report_config.txt
report_power  >> ./SYN/RPT/report_config.txt

echo "Config POWER REPORT HIER with 2 errors\n" >> ./SYN/RPT/report_config.txt
report_power -hier >> ./SYN/RPT/report_config.txt
```

D.2.4 Clock Gate Insertion Compile Script

```
#-------------------------------------------#
#                                           #
#        Decoder design compile script      #
#                 CONFIG                    #
#                 WITH CG                   #
#                                           #
#                                           #
#-------------------------------------------#

#Testing with 0, 1 and 2 errors
remove_design -all

analyze -library WORK -format verilog {./SRC/decoder_top.v ./SRC/adder.v
./SRC/AND_gate.v ./SRC/D_reg.v ./SRC/Chien_cell_W.v ./SRC/Chien_cell.v
./SRC/Chien_search.v ./SRC/Codeword_Store.v ./SRC/Error_corr.v
./SRC/Forney.v ./SRC/MUX.v ./SRC/Full_mult.v ./SRC/in_stage.v ./SRC/inverse_GF32.v
./SRC/Mult_x2_5bit.v ./SRC/Mult_x4_5bit.v ./SRC/Mult_x8_5bit.v ./SRC/MUX_1to2.v ./SRC/
    NOR.v
./SRC/Omega_calc.v ./SRC/BM_alg.v ./SRC/syndrome_calc_S0.v ./SRC/syndrome_calc_S1.v
./SRC/syndrome_calc_S2.v ./SRC/syndrome_calc_S3.v ./SRC/Syndrome_ctrl.v ./SRC/
    clk_gate.v}

elaborate decoder_top -library WORK

# Apply constraints
source ./SYN/SCR/constraints.tcl

set_clock_gating_style -sequential latch

insert_clock_gating

compile -incremental

# Run report commands
echo "Decoder config  WITH CG\n" >> ./SYN/RPT/report_config_CG.txt
echo "Report files" >> ./SYN/RPT/report_config_CG.txt

echo "Config QOR REPORT\n" >> ./SYN/RPT/report_config_CG.txt
report_qor >> ./SYN/RPT/report_config_CG.txt

echo "Config AREA HIER\n" >> ./SYN/RPT/report_config_CG.txt
report_area -hierarchy >> ./SYN/RPT/report_config_CG.txt

echo "Config CELL REPORT\n" >> ./SYN/RPT/report_config_CG.txt
report_cell >> ./SYN/RPT/report_config_CG.txt


#----------------------#
# ZERO ERRORS          #
#----------------------#
# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error0.saif -instance_name decoder_top_tb/UUT -verbose

#power reports
echo "Config POWER REPORT with 0 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power  >> ./SYN/RPT/report_config_CG.txt

echo "Config POWER REPORT HIER with 0 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power -hier >> ./SYN/RPT/report_config_CG.txt

#----------------------#
```

```
# ONE ERROR            #
#----------------------#

# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error1.saif -instance_name decoder_top_tb/UUT -verbose

# Run report commands
#power reports
echo "----------------------------------------\n" >> ./SYN/RPT/report_config_CG.txt
echo "Config POWER REPORT with 1 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power  >> ./SYN/RPT/report_config_CG.txt

echo "Config POWER REPORT HIER with 1 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power -hier >> ./SYN/RPT/report_config_CG.txt

#----------------------#
# TWO ERRORS           #
#----------------------#

# Read the backword annotation SAIF file
read_saif -input ./SYN/SAIF/error2.saif -instance_name decoder_top_tb/UUT -verbose

# Run report commands
#power reports
echo "----------------------------------------\n" >> ./SYN/RPT/report_config_CG.txt
echo "Config POWER REPORT with 2 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power  >> ./SYN/RPT/report_config_CG.txt

echo "Config POWER REPORT HIER with 2 errors\n" >> ./SYN/RPT/report_config_CG.txt
report_power -hier >> ./SYN/RPT/report_config_CG.txt
```