FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Verilog Implementation of a Forward Error Correcting Reed Solomon Encoder and Decoder

Artur Jorge Alves Antunes

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Jose Carlos Alves

Second Supervisor: Rui Rainho Almeida

July 28, 2017

Abstract

Nowadays the consumers' needs for faster communication systems and higher image and sound fidelity has forced the industry to adopt techniques that compress data, in order to increase the usable channel capacity. On the other hand, errors in compressed data have catastrophic results, and to that end, Forward Error Correcting codes are used do detect and correct errors that may have happened during data transmission.

Reed-Solomon has been widely used in consumer devices, and the need for higher bit-rates requires special implementations that are capable of handling high frequency parallel interfaces at full capacity. This dissertation provides a Verilog implementation and a SystemVerilog validation of a parametrized Reed-Solomon encoder and decoder with a parallel interface.

In order to achieve this, mathematical expansion of difference equations and sequential logic parallelization of existing algorithms and techniques are used.

With this approach it is guaranteed synthesis for TSMC $28\,nm$ at $450\,MHz$ using RS(255,251), with a nine symbol interface, the encoder and the decoder have a spacial impact of 3186 and 33507 equivalent NANDs respectively.

A compute capacity of up to $120\,Gbits/s$ and $60\,Gbits/s$, on the encoder and decoder respectively, is achieved at higher clock speeds.

"Nothing happens to anybody which
he is not fitted by nature to bear."

Marcus Aurelius

Contents

List of Figures

List of Tables

Abbreviations and Symbols

HD	High-Definition
HDMI	High-Definition Multimedia Interface
RS	Reed-Solomon
FEC	Forward Error Correcting
GF	Galois/Finite Field
BCH	Bose–Chaudhuri–Hocquengheme
DUT	Device Under Test
DPI	SystemVerilog Direct Programming Interface
LUT	Look-up Table

Chapter 1

Introduction

1.1 Context

Since the appearance of the first television, the public has been demanding for higher resolutions and more functionality. It was not long after that the people realized that the capabilities of the televisions of the time were not good enough to cover their needs. Simply using a television as an endpoint to display a broadcast media, like a radio, was not keeping up with the demands. From this, communication technologies flourished.

The public was using third party hardware along side the television to play games, access extra channels and later on to access the internet. Televisions went from simply displaying monochromatic images to provide sound and reproducing vivid and colourful three-dimensional scenes.

So as to allow for those luxuries the interconnections between devices had to be carefully engineered, which ended up making most interface technologies, mainly multimedia ones, converge to an increase of the required bandwidth. This necessity for more bandwidth is easily justifiable by the need for higher resolutions and faster display panels, among other factors. For example the HDMI version 2.0b can sustain resolutions up to 3840x2160 (Ultra HD) more commonly known as 4K, and the latest monitors can handle up to $144\,Hz$ refresh rates.

1.2 Motivation

In order to keep up with the market needs, multimedia interfaces have to find ways so that the overall communication capacity is improved, like the use of compression methods, among others. Compression, as the name indicates, allows for a given message to have the same, or almost the same, information but in a smaller package.

For example, a video may be represented as a series of discrete sequential images, or frames. One way to compress it is to use only one image and infer the subsequent frames, from this first image. That means if an error occurs in the first image, every subsequent frame will appear with errors, until a new full image is displayed. Such methods when used in noisy channels, such as

wireless communications or poorly designed cables, generate small errors in the data transmission, which may cause great impact in the user experience.

In order to prevent this, systems like Reed Solomon with Forward Error Correction are implemented. Error Correction Codes like this add redundant data during transmission, which is then used to detect and, if possible, correct errors that may have occurred during transmission.

1.3 Goals

Reed-Solomon has been used for decades, however, every design is different and there may be systems where existent designs might not be fully compatible or may not comply with the design restrictions. For this implementation, Synopsys has presented a set of requirements that were not present in other implementations. Synopsys required both Reed Solomon encoder and decoder, which had to process nine bytes per clock cycle, and respect a set of control signals and polynomials that will be specified and explained later in this document.

Both the encoder and the decoder should be synthesizable for TSMC 28 nm architecture achieve frequencies above 450 MHz while minimizing both cell area and power consumption. The latency is non-important, at least not its actual value, as long at it is consistent, independently of the situation presented to the modules. The modules should be able to work in real-time, that is the input bit rate should be the same as the output.

During this dissertation, a functional validation of the modules with an appropriate SystemVerilog testbench, and C implementation to use as a golden model was also required.

Chapter 2

Coding Theory

Ever since the beginning of communications systems, noise and interference have been great obstacles in the pursuit of the theoretical limit for a channel capacity, that is the theoretical limit for how much information can flow in a given channel. Forward Error Correcting codes have played a great role in this mission. By adding redundant data alongside the information data, it is possible to recover some errors that may have occurred during data transfer. For further efficiency mathematical tools are used, like the Galois Fields, which represent the symbol space used in Reed-Solomon. This chapter will cover an overview of Linear Block Codes, where Reed-Solomon are inserted, as well as Galois Fields arithmetic deductions and Reed-Solomon encoding and decoding generalizations.

2.1 Galois Fields

Fields, Rings and Groups represent sets of numbers alongside operations whose results are always present in the same set. A Group has only two valid operations addition and subtraction. A Ring is always a Group but also supports other operations such as multiplication. A Field is the more general presented concept, it integrates the previously mentioned operations as well as inversion. For example, the set of real numbers ($\Re$) forms a Field. On the other hand, the set of natural numbers ($\aleph$) can only form a Ring, since none if its numbers' inversions result in a natural number.

Galois or Finite Fields (GF), unlike the set of real numbers, only have a finite amount of elements in its set and the number of elements is always a power of a prime. For example, $\{0,1,2,3,4\}$ and $\{0,1,2,3\}$ may form Galois Fields, since they have a power of a prime number of elements, 5 and 2^2 respectively. On the other hand, $\{0,1,2,3,4,5\}$ has six elements which is not a power of a prime. They are represented as $GF(p^m)$, $GF(5)$ and $GF(2^2)$ in the previous examples, and there are two main classes. One is called "prime fields" if $m = 1$, and "extended fields" otherwise.[1]

In the prime fields, most of the operations are fairly simple. Assuming:

$$a,b,c,d,e \in GF(p) \tag{2.1}$$

Addition, subtraction and multiplications are computed as if they were normal numbers, however for finite fields, the division's remainder by p is taken:

$$a+b \equiv c \bmod p[1] \tag{2.2}$$

$$a-b \equiv d \bmod p[1] \tag{2.3}$$

$$a \cdot b \equiv e \bmod p[1] \tag{2.4}$$

Inversion, at the other hand is a bit more complicated since, by definition:

$$a^{-1} \cdot a \equiv 1 \bmod p[1] \tag{2.5}$$

Finding a^{-1} is not as trivial as the other operations, however, there are iterative methods that calculate it, like the Euclidean algorithm. If the field is not very large, a brute force search, or a lookup table are also valid alternatives.

When $m > 1$, in the extended fields, things get a bit more complicated. They can not be simply represented as numbers like the prime fields. The arithmetic rules stated earlier no longer apply. However by representing the Galois Field elements as polynomials instead of simple numbers, some operations are still kept fairly simple. For example, considering the $GF(2^2)$, which was stated before that a set of elements could be $\{0,1,2,3\}$, taking the polynomial representation, the set becomes $\{0,1,x,x+1\}$, which may be simplified by looking at the polynomials coefficients as an array, $\{[0\ 0],[0\ 1],[1\ 0],[1\ 1]\}$, which directly maps elements to the binary representation of the original number representation showed first. This means that the Galois Fields can always be represented as a set of number, however, the operations must take in consideration that they are actually polynomials. [1] Assuming:

$$A(x),B(x),C(x) \in GF(p^m) \tag{2.6}$$

Addition and subtraction are very similar to those of prime fields, however numbers are treated as polynomials, so instead of the result being the remainder of the division by p, a normal operation must be applied to the polynomials and taken the remainder of the division of each coefficient by p:

$$C(x) = A(x) \pm B(x) \equiv \sum_{i=0}^{m-1} c_i \cdot x^i [1] \tag{2.7}$$

where,

$$c_i \equiv (a_i \pm b_i) \bmod p[1] \tag{2.8}$$

Multiplication and inversion becomes more complicated, because since the elements of the extended fields are actually polynomials and multiplication of two polynomials results in a polynomial whose order is the sum of the multiplicands polynomials' orders. Which means, that by considering a $GF(p^m)$ which has polynomials of order up to $m-1$ and by multiplying two polynomials of that order, the result would be of order $2m-2$, which is not part of the available set. A polynomial of order m is needed, more specifically a primitive polynomial, p(x). [1] Now the Galois multiplication can be defined as:

$$A(x) \cdot B(x) \equiv C(x) \bmod p(x)[1] \tag{2.9}$$

Inversion does not change that much, the only difference is that $p(x)$ must used, instead of p:

$$A^{-1}(x) \cdot A(x) \equiv p(x)[1] \tag{2.10}$$

With the introduction of $p(x)$, comes one more form of displaying the elements of the field itself. If there is an element of $GF(p^m)$, then the Galois Field set may be expressed as:

$$\{\alpha^{\infty}, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{p^m-2}\}[2] \tag{2.11}$$

where $\alpha \neq 0 \neq 1$ and is denoted as the field root. By taking this representation, multiplication and inversion becomes doable:

$$\alpha^a \cdot \alpha^b = \alpha^{a+b} = \alpha^{c \bmod (p^m-1)}[2] \tag{2.12}$$

$$\alpha^{(-a) \bmod (p^m-1)} \cdot \alpha^b = \alpha^a = 1 \bmod p(x)[2] \tag{2.13}$$

where,

$$\alpha^a, \alpha^b, \alpha^c \in GF(p^m)[1] \tag{2.14}$$

However in this representation, additions and subtractions are not easy, neither is converting between representations for higher order Galois Fields, since their size increases exponentially with m. Using 2 as value for p is generally the most interesting scenario in digital systems, for obvious reasons, as it simplifies element representation as well as some operations. [2] Throughout the remaining of the document, $p = 2$,$m = 4$,$p(x) = x^4 + x + 1$, will be used an example.

The equation 2.7 may be simplified, since calculating the remainder of the division by 2 corresponds to truncating to the least significant bit, the Galois addition and subtraction may be simplified to the same equation, but where c_i is now given by:

$$c_i \equiv a_i \; xor \; b_i[2] \tag{2.15}$$

From the above it is inferable that, in fact, the addition and the subtraction are actually the same operation and, by representing the number in polynomial form, the addition is simply an exclusive-or. For example, take the elements $15(x^3+x^2+x+1)$ and $8(x^3)$:

$$\begin{aligned} 15+8 &= (x^3+x^2+x+1)+(x^3) \\ &= [1\ 1\ 1\ 1]\ xor\ [1\ 0\ 0\ 0] \\ &= [0\ 1\ 1\ 1] \\ &= 7 \end{aligned} \tag{2.16}$$

The multiplication, on the other hand, is more complex. However, by the definition, it is still possible to arrive to an algorithm that makes multiplication easily synthesizable. Starting by an example:

$$\begin{aligned} 15\cdot 8 &= (x^3+x^2+x+1)\cdot(x^3) \bmod p(x) \\ &= x^6+x^5+x^4+x^3 \bmod p(x) \\ &= [1\ 1\ 1\ 1\ 0\ 0\ 0] \bmod [1\ 0\ 0\ 1\ 1] \end{aligned} \tag{2.17}$$

$$\begin{array}{r|l} \mathbf{1\,1\,1\,1\,0\,0\,0} & \mathbf{1\,0\,0\,1\,1} \\ \hline 1\,1\,0\,1\,0\,0 & 1 \\ 1\,0\,0\,1\,0 & 1\,1 \\ \mathbf{1} & 1\,1\,1 \end{array} \tag{2.18}$$

$$15\cdot 8 = 1 \tag{2.19}$$

Here it is noticeable a two-step sequential process. However, due to the high throughput required, it is necessary to execute this multiplication combinationally, or in a pipelined version, which is feasible in three steps. Assuming:

$$\begin{aligned} A(x), B(x) &\in GF(2^4) \\ C(x) &\in GF(2^4) \end{aligned} \tag{2.20}$$

The first step remains the same, but combinationally. By simply doing a generic polynomial multiplication, which may be described as follows:

$$C(x) = A(x)\cdot B(x) = \sum_{i=0}^{6} c_i \cdot x^i \tag{2.21}$$

$$\begin{array}{lllllllll}
c_6 = & a_3 \cdot b_3 & & & & & & & \\
c_5 = & a_3 \cdot b_2 & + & a_2 \cdot b_3 & & & & & \\
c_4 = & a_3 \cdot b_1 & + & a_2 \cdot b_2 & + & a_1 \cdot b_3 & & & \\
c_3 = & a_3 \cdot b_0 & + & a_2 \cdot b_1 & + & a_1 \cdot b_2 & + & a_0 \cdot b_3 \\
c_2 = & & & a_2 \cdot b_0 & + & a_1 \cdot b_1 & + & a_0 \cdot b_2 & \\
c_1 = & & & & & a_1 \cdot b_0 & + & a_0 \cdot b_1 & \\
c_0 = & & & & & & & a_0 \cdot b_0 &
\end{array} \tag{2.22}$$

This can be simplified into a more generic and compact version:

$$c_{i+j} a_i \cdot b_j, \quad i = 0,1,\ldots,m-1, \quad j = 0,1,\ldots,m-1 \tag{2.23}$$

The second step is a bit more complicated to implement in a single step, therefore it has been split into two, where first an auxiliary polynomial ($D(x)$) is calculated. Considering the primitive polynomial to be:

$$p(x) = x^m + \sum_{i=0}^{m-1} p_i \cdot x^i \tag{2.24}$$

The following table is formed from both the primitive polynomial $p(x)$ and the previously calculated $C(x)$, this was obtained from a generalization of the long division and actually corresponds to most of it.

$$\begin{array}{ccccccc}
c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\
d_3 \cdot p_3 & d_3 \cdot p_2 & d_3 \cdot p_1 & d_3 \cdot p_0 & & & \\
 & d_2 \cdot p_3 & d_2 \cdot p_2 & d_2 \cdot p_1 & d_2 \cdot p_0 & & \\
 & & d_1 \cdot p_3 & d_1 \cdot p_2 & d_1 \cdot p_1 & d_1 \cdot p_0 & \\
 & & & d_0 \cdot p_3 & d_0 \cdot p_2 & d_0 \cdot p_1 & d_0 \cdot p_0
\end{array} \tag{2.25}$$

where each coefficient of $d(x)$ is the sum of the column before the one it is first used:

$$\begin{aligned}
d_3 &= 0 \\
d_2 &= c_6 + d_3 \cdot p_3 \\
d_1 &= c_5 + d_3 \cdot p_2 + d_2 \cdot p_3 \\
d_0 &= c_4 + d_3 \cdot p_1 + d_2 \cdot p_2 + d_1 \cdot p_3
\end{aligned} \tag{2.26}$$

This can be simplified into a more generic version:

$$d_i = c_{i+m} + \sum_{j=m-1}^{i+1} d_j \cdot p_{m+i-j}, \quad i = m-1,\ldots,1,0 \tag{2.27}$$

After arriving to this polynomial, the final result, $E(x)$, is achieved with a similar equation:

$$e_i = c_i + \sum_{j=0}^{i} d_j \cdot p_{i-j}, \quad i = 0,1,\ldots,m-1 \tag{2.28}$$

where,

$$E(x) = \sum_{i=0}^{m-1} e_i \cdot x^i \tag{2.29}$$

and,

$$E(x) \in GF(2^m) \tag{2.30}$$

The Galois Inversion of a number a is not easily achieved, by definition, $a \cdot a^{-1} = 1$, one can simply multiply every single possible number by a and try to find one that results in 1. This may be implemented either sequentially or combinationally, however, since the system requires very high throughput, a sequential implementation and a combinational one is very area inefficient, a lookup table is more suitable, on the other hand, even this becomes unreasonable for very large Galois Fields. [2]

2.2 Block Codes

Linear Block codes represent a great class of forward error correcting codes where the transmitted word data, generally denoted as being composed as N symbols, where each symbol is a set of M bits, is segmented in two blocks, information and redundancy, that is, the information data, set of K symbols is kept intact and the redundancy is added afterwards, as described in figure 2.1. Several Examples of block codes include Hamming Codes, BCH Codes and Reed-Solomon Codes. [3]

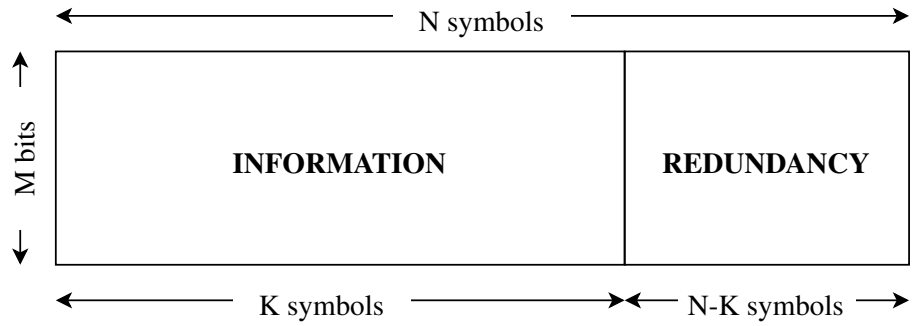

Figure 2.1: Block Code[3]

2.3 Reed-Solomon

The Reed Solomon codes differentiate themselves from the remaining BCH codes by the number of bits present in their symbols and the relationship between the symbol width and the length of

the encoded message. For example, if the symbols are composed of 8 bits, then the message must be 255, that is, $N = 2^M - 1$. However there are versions of Reed Solomon that do not follow this rule. Shortened versions of Reed Solomon append symbols either at the beginning or at the end of the information message, which is then encoded as a normal code, but these added symbols are removed before transmission. During the decoding process they are appended to the received message in order to normalize decoding procedure. [2]

In Reed Solomon, the redundancy symbols are created with the resource of a generator polynomial of order $2T$, where $T = \lfloor (N-K)/2 \rfloor$, is the number of correctable symbols. If the error count in the received message is larger than T, the information can no longer be retrieved without errors. This generator polynomial, commonly denoted as $g(x)$ has the particularity of having all of its roots as consecutive elements of the Galois Field in the exponential form: [2]

$$\begin{aligned} g(x) &= (x+\alpha^b)\cdot(x+\alpha^{b+1})\cdot\ldots\cdot(x+\alpha^{b+2T-1}) \\ &= x^{2T} + g_{2T-1}\cdot x^{2T-1} + \ldots + g_2\cdot x^2 + g_1\cdot x + g_0 [2] \end{aligned} \tag{2.31}$$

where α is the Galois Field root.

2.3.1 Encoding

Assuming a message:

$$m(x) = m_{K-1}\cdot x^{K-1} + \ldots + m_2\cdot x^2 + m_1\cdot x + m_0 \tag{2.32}$$

that has to be encoded, the encoded message $t(x)$ may be achieved by:

$$\begin{aligned} t(x) &= x^{2T}\cdot m(x) + m(x) \bmod g(x) \\ &= t_{N-1}\cdot x^{N-1} + \ldots + t_2\cdot x^2 + t_1\cdot x + t_0 \end{aligned} \tag{2.33}$$

This is just the message $m(x)$ and the remainder of the division of $m(x)$ by $g(x)$, the parity symbols, appended at the end. This has the by-product of ensuring that $t(x)$ is always divisible by $g(x)$, which is quite useful during the decoding [2]. In equation 2.18 is presented an algorithm for calculating the remainder of a polynomial division, however, its complexity grows exponentially with the order of the polynomials, and having a polynomial of 255 coefficients would make the algorithm very inefficient. Therefore a different approach is necessary. The most common algorithm is based on Linear Feedback Shift-Register (LFSR) which is presented in figure 2.2.

In this algorithm, by initializing the registers X to zero and shifting in through the input the message $m(x)$, starting with higher order coefficient, after the entire message has been shifted in, the content of the registers X is nothing more than $m(x) \bmod g(x)$. However the main disadvantage of this algorithm is that it only supports a single input, not good enough for the desired scenario.

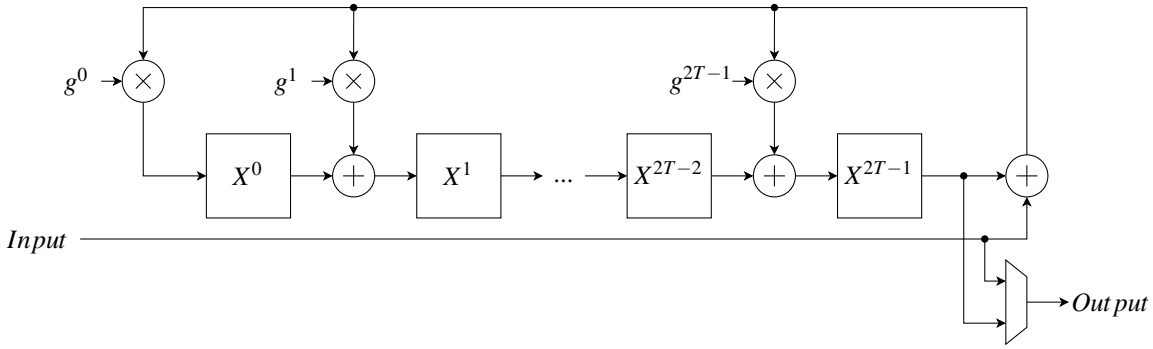

Figure 2.2: Generalized Linear Feedback Shift-Register

On the other hand, it is a good starting point for a parallel encoder, as it is possible to obtain generic equations. For ease of use, the *Input* will be denoted as I:

$$\begin{aligned} X_n^0 &= (X_{n-1}^{2T-1} + I_n) \cdot g_0 \\ X_n^1 &= (X_{n-1}^{2T-1} + I_n) \cdot g_1 + X_{n-1}^0 \\ &\dots \\ X_n^{2T-1} &= (X_{n-1}^{2T-1} + I_n) \cdot g_{2T-1} + X_{n-1}^{2T-2} \end{aligned} \tag{2.34}$$

From this, and by assuming that $X^i = 0$ and $i < 0$, it is possible to generalize this set of equations in a single more generic and compact equation:

$$X_n^i = (X_{n-1}^{2T-1} + I_n) \cdot g_i + X_{n-1}^{i-1}, \quad i = 0, 1, \dots, 2T-1 \tag{2.35}$$

Since this equation has been obtained, is possible to generate a new equation for two inputs, simply by going back in time, by replacing X_{n-1}^i with $(X_{n-2}^{2T-1} + I_{n-1}) \cdot g_i + X_{n-2}^{i-1}$.

$$\begin{aligned} X_n^i &= ((X_{n-2}^{2T-1} + I_{n-1}) \cdot g_{2T-1} + X_{n-2}^{2T-2} + I_n) \cdot g_i + (X_{n-2}^{2T-1} + I_{n-1}) \cdot g_{i-1} + X_{n-2}^{i-2} \\ &= (X_{n-2}^{2T-1} + I_{n-1}) \cdot (g_{2T-1} \cdot g_i + g_{i-1}) + (X_{n-2}^{2T-2} + I_n) \cdot g_i + X_{n-2}^{i-2} \end{aligned} \tag{2.36}$$

The equations start to get exponentially complex as time goes back, as expected, going back in time is not easy. However a pattern is already noticeable:

$$X_n^i = X_{n-\omega}^{i-\omega} + \sum_{j=1}^{\omega} \gamma_i^j \cdot (I_{n+j-\omega} + x_{2T-j}^{n-\omega}), \quad i = 0, 1, \dots, 2T-1 \tag{2.37}$$

where ω is the number of inputs and γ is a constant generated from the generator polynomial, but there is not enough information to derive this constant with certainty. It is needed to go back

one more clock:

$$\begin{aligned} X_n^i = {} & (X_{n-3}^{2T-1} + I_{n-2}) \cdot (g^{2T-1} \cdot (g^{2T-1} \cdot g^i + g^{i-1}) + g^{2T-2} \cdot g^i + g^{i-2}) \\ & + (X_{n-3}^{2T-2} + I_{n-1}) \cdot (g^{2T-1} \cdot g^i + g^{i-1}) \\ & + (X_{n-3}^{2T-3} + I_{n-0}) \cdot g^i \\ & + X_{n-3}^{i-3} \end{aligned} \tag{2.38}$$

Now it is finally possible to infer the equation for γ, in order to help readability and to better match the input format, a change of notation is required where n will no longer make part:

$$X^i = X^{i-\omega} + \sum_{j=1}^{\omega} \gamma_i^{\omega-j+1} \cdot (I_{j-1} + X^{2T-j}), \quad i = 0, 1, \ldots, 2T-1 \tag{2.39}$$

$$\gamma_i^{\omega} = g^{i-\omega+1} + \sum_{j=1}^{\omega} g^{2T-j} \cdot \gamma_i^{\omega-j}, \quad \gamma_i^0 = 0,\ i = 0, 1, \ldots, 2T-1 \tag{2.40}$$

From this, it is possible to generate an encoder with a configurable number of parallel input ports. In the limit this is reduced to an LFSR, when $\omega = 1$, and if the number of inputs is the same as the number of symbols to encode ($\omega = K$), it is possible to encode one message per clock.

2.3.2 Decoding

Assuming a received polynomial $r(x)$ which is composed of the encoded polynomial $t(x)$ altered by an error polynomial $E(x)$ with a degree lower than N and with at most $0 < v \leq T$ non zero coefficients, such that:

$$r(x) = t(x) + e(x) \tag{2.41}$$

where

$$e(x) = Y_1 \cdot x^{e_1} + Y_2 \cdot x^{e_2} + \ldots + Y_v \cdot x^{e_v} \tag{2.42}$$

then the original message can still be decoded correctly. [3]

Since, as stated before, $t(x)$ is always divisible by $s(x)$:

$$t(x) \bmod g(x) = 0 \Rightarrow r(x) \bmod g(x) = e(x) \bmod g(x) \tag{2.43}$$

If $e(x) \bmod g(x) \neq 0$ then $\exists i \in \{0, 1, \ldots, 2T-1\}$ such that $r(x) \bmod (x + \alpha^{b+i}) \neq 0$, that is, if $e(x)$ is not divisible by $g(x)$, then it must not be divisible by least one of the roots of $g(x)$. The syndrome of the received message may then be defined as:

$$S_{b+i} = r(x) \bmod (x + \alpha^{b+i}) = r(\alpha^{b+i}) \tag{2.44}$$

as per the polynomial remainder theorem. [2]

By representing the received code word as:

$$r(x) = (\ldots((r_{i-1} \cdot x + r_{i-2}) \cdot x + r_{i-3}) \cdot x + \ldots + r_1) \cdot x + r_0, \tag{2.45}$$

And with equation 2.44, is possible to calculate the Syndrome iteratively. This is commonly known as the Horner's method. [3]

From equation 2.42 the syndromes become:

$$S_{b+i} = Y_1 \cdot (\alpha^{b+i})^{e_1} + Y_2 \cdot (\alpha^{b+i})^{e_2} + \ldots + Y_v \cdot (\alpha^{b+i})^{e_v} \tag{2.46}$$

To ease the readability, S_{b+i} and $(\alpha^{b+i})^{e_j}$ will be simply referred as S_i and X_j^i respectively:

$$S_i = Y_1 \cdot X_1^i + Y_2 \cdot X_2^i + \ldots + Y_v \cdot X_v^i = \sum_{j=1}^{v} Y_j \cdot X_j^i \tag{2.47}$$

By converting the previous equation into a matrix form:

$$\begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ S_{2t-1} \end{bmatrix} = \begin{bmatrix} X_1^0 & X_2^0 & \ldots & X_v^0 \\ X_1^1 & X_2^1 & \ldots & X_v^1 \\ \vdots & \vdots & \ddots & \vdots \\ X_1^{2t-1} & X_2^{2t-1} & \ldots & X_v^{2t-1} \end{bmatrix} \cdot \begin{bmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_v \end{bmatrix} \tag{2.48}$$

it becomes clear that the system is non-linear and has more that one possible solutions, since not only the Y_j and X_j^i are unknown, but also v which may be any value from 1 to T. To solve this, a new polynomial is formulated, where its roots are error locations and is denominated as the error locator polynomial. [2] The inverse of this polynomial:

$$\begin{aligned} \Lambda(x) &= (1 + X_1 \cdot x) \cdot (1 + X_2 \cdot x) \ldots (1 + X_v \cdot x) \\ &= 1 + \Lambda_1 \cdot x + \Lambda_2 \cdot x^2 + \ldots + \Lambda_v \cdot x^v [3] \end{aligned} \tag{2.49}$$

is used instead, since it makes the arithmetic deductions easier. [2]

By writing out all the equations that make $\Lambda(x) = 0$:

$$\begin{aligned} 1 + \Lambda_1 \cdot X_1^{-1} + \Lambda_2 \cdot X_1^{-2} + \ldots + \Lambda_v \cdot X_1^{-v} &= 0 \\ 1 + \Lambda_1 \cdot X_2^{-1} + \Lambda_2 \cdot X_2^{-2} + \ldots + \Lambda_v \cdot X_2^{-v} &= 0 \\ \ldots & \\ 1 + \Lambda_1 \cdot X_v^{-1} + \Lambda_2 \cdot X_v^{-2} + \ldots + \Lambda_v \cdot X_v^{-v} &= 0 \end{aligned} \tag{2.50}$$

multiplying each equation by $Y_j \cdot X_j^i$

$$\begin{aligned} Y_1 \cdot X_1^{i+v} + \Lambda_1 \cdot Y_1 \cdot X_1^{i+v-1} + \Lambda_2 \cdot Y_1 \cdot X_1^{i+v-2} + \ldots + \Lambda_v \cdot Y_1 \cdot X_1^i = 0 \\ Y_2 \cdot X_2^{i+v} + \Lambda_1 \cdot Y_2 \cdot X_2^{i+v-1} + \Lambda_2 \cdot Y_2 \cdot X_2^{i+v-2} + \ldots + \Lambda_v \cdot Y_2 \cdot X_2^i = 0 \\ \ldots \\ Y_v \cdot X_v^{i+v} + \Lambda_1 \cdot Y_v \cdot X_v^{i+v-1} + \Lambda_2 \cdot Y_v \cdot X_v^{i+v-2} + \ldots + \Lambda_v \cdot Y_v \cdot X_v^i = 0 \end{aligned} \tag{2.51}$$

and finally adding all the equations together, the following set of $2T - v$ equation is formed:

$$\sum_{j=1}^{v} Y_j \cdot X_j^{i+v} + \Lambda_1 \cdot \sum_{j=1}^{v} Y_j \cdot X_j^{i+v-1} + \Lambda_2 \cdot \sum_{j=1}^{v} Y_j \cdot X_j^{i+v-2} + \ldots + \Lambda_v \cdot \sum_{j=1}^{v} Y_j \cdot X_j^i = 0 \tag{2.52}$$

By comparison with equation 2.47 is possible to simplify the previous equation to:

$$S_{i+v} + \Lambda_1 \cdot S_{i+v-1} + \Lambda_2 \cdot S_{i+v-2} + \ldots + \Lambda_v \cdot S_i = 0, \quad i = 0, 1, \ldots, 2T - v - 1 \tag{2.53}$$

Since this system has more equations than variables, v variables and $2T - v - 1$ equations, only the first v equations are necessary [3], and by representing them in matrix form:

$$\begin{bmatrix} S_v \\ S_{v+1} \\ \vdots \\ S_{2v-1} \end{bmatrix} = \begin{bmatrix} S_{v-1} & S_{v-2} & \ldots & S_0 \\ S_{v-1} & S_{v-1} & \ldots & S_1 \\ \vdots & \vdots & \ddots & \vdots \\ S_{2v-2} & S_{2v-3} & \ldots & S_{v-1} \end{bmatrix} \cdot \begin{bmatrix} \Lambda_1 \\ \Lambda_2 \\ \vdots \\ \Lambda_v \end{bmatrix} \tag{2.54}$$

is possible to infer that the system is linear, and by using the result of this into the system in equation 2.48, that also becomes linear. On the other hand, to solve either of them, v must be known. The solution is to take the smallest v possible, that makes the system solvable, since the likelihood of having errors decreases as the number of errors increase. That is, it is more likely to have 1 error, than to have 2 errors. [2]

According to [4], a new polynomial called error evaluator polynomial ($\omega(x)$), is defined as:

$$\sigma(x) \cdot S(x) \equiv \omega(mod x^{2t}) \tag{2.55}$$

One of the most known algorithms for determining both the error locator and the error evaluator polynomials is the Berleykamp algorithm, the following, presented in [4], is an inversion-less version, which, unlike the original one, does not require an inverter during the iterative step:

```
/*                          ASSUMPTIONS                                  */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1} (x - α^k).      */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword.     */

PROCEDURE Berlekamp2;
BEGIN
FOR k := 0,1,...,2t-1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
D := 0;
δ := 1;
σ^0(x) := τ^0(x) := ω^0(x) := 1;
γ^0(x) := 0;
FOR k := 0,1,...,2t-1 DO
BEGIN
   Δ := Σ_{i=0}^{k} σ_i^k S_{k-i};
   σ^{k+1} := δσ^k - Δxτ^k
   ω^{k+1} := δω^k - Δxγ^k
   IF Δ = 0 OR (2D > k+1) THEN
      τ^{k+1} := xτ^k;
      γ^{k+1} := xγ^k;
   ELSE
      D := k+1-D;
      δ := Δ;
      τ^{k+1} := σ^k;
      γ^{k+1} := ω^k;
   ENDIF;
END;
σ(x) := σ^2 t(x);
ω(x) := ω^2 t(x);
FOR k := 0,1,...,n-1 DO
   IF σ(α^{-k}) = 0 THEN s_k := s_k - α^{k(L-2)} ω(α^{-k})/σ'(α^{-k}) ENDIF;
END;
```

For a given position j in the received message, it's possible to test if an error has occurred, by evaluating the error locator polynomial, $\Lambda(x)$, for $x = X_j^{-1}$. If $\Lambda(X_j^{-1}) = 0$, then an error as occurred in the position j. The Forney algorithm as described in [3], calculates the error from the following formula:

$$Y_j = X_j^{1-b} \cdot \frac{\Omega(X_j^{-1})}{\Lambda'(X_j^{-1})} [3] \tag{2.56}$$

According to [4], the derivative of a polynomial in a base 2 Galois field can be obtained from the odd terms:

$$\Lambda^{'}(x) = \sum_{i=0}^{2t} i \cdot \Lambda_i x^{i-1} = \sum_{odd\, i} \Lambda_i x^{i-1} \tag{2.57}$$

From this is possible to simplify the other equation:

$$\Lambda^{'}(x) = \sum_{odd i} \Lambda_i x^{i-1} = x^{-1} \cdot \sum_{odd i} \Lambda_i x^i = x^{-1} \cdot \Lambda_{odd}(x) \tag{2.58}$$

$$Y_j = X_j^{1-b} \cdot \frac{\Omega(X_j^{-1})}{\Lambda'(X_j^{-1})} = X_j^{1-b} \cdot \frac{\Omega(X_j^{-1})}{X_j \cdot \Lambda_{odd}(X_j^{-1})} = X_j^{-b} \cdot \frac{\Omega(X_j^{-1})}{\Lambda_{odd}(X_j^{-1})} \tag{2.59}$$

2.4 State of the Art

Since the first major appearance of Reed-Solomon encoder and decoders in the 1977 Voyager mission where a bit-rate of $1\,Mbit/s$ was archived, Reed-Solomon has been widely used in commercial applications, ranging from the compact disc (CD), to storage devices, digital television (DTV), digital video broadcasting (DVB) [5] [6]. The number of related articles and papers on this is enormous, however only a handful present parallel algorithms. In this section a small selection of the state of the art will be presented.

A parallel architecture presented in [7] is capable of processing a number of positive integer symbols per clock cycle, however it computes several different messages at once but only one symbol of each message per clock cycle, unlike this implementation which processes several symbols of the same message at once. Due to the article's age, the usable clock frequencies were quite low ($50\,MHz$) and the available technologies were around $1\,\mu m$, therefore for a four symbol parallel circuit would only reach up to $1.6\,Gbit/s$, which is very low for today's standards. On the other hand, the cell count usage is still useful as it is mostly architecture dependent. This paper reported a cell count for a four symbol encoder of 1490 gates and 34410 gates for the decoder with the extra memory included. Without it the paper reports a cell count of 18506.

A comparison of several architectures is provided in [8] where they are compared in terms of gate count, latency and path delay, as well as an implementation of one of the specified designs for $0.12\,\mu m$ and $0.18\,\mu m$ using an ASIC synthesis tool. Although this paper provides a good base for the available architectures it does not bring valid results for comparison since the implementations are only able to decode one symbol at a time and the used Reed-Solomon polynomial is larger that the focus of this implementation. Although not very useful, from this paper is possible to note that the implemented algorithm is able to reach frequencies up to $200\,MHz$ and bit-rates of $1.6\,Gbit/s$.

An high speed implementation for FPGA is exposed in [9], where, as the name of the thesis implies, presents an implementation capable of reaching $40\,Gbit/s$ in a FPGA. The algorithm presented in the referred thesis is capable of frequencies up to $169\,MHz$ in a Xilinx's Virtex4 FPGA and can only process one symbol per clock cycle. The high bit rate is archived by having 32 decoders in parallel. The results here are not very relevant for this thesis, as it implements a different algorithm for FPGA, making any meaningful comparison not fair. However it is still relevant as it shows the flexibility of Reed-Solomon codecs to reach high frequencies if the architecture is properly chosen and implemented.

Chapter 3

Implementation

In this chapter a Verilog implementation of the formulas derived in the previous chapter will be presented. It starts with a description of the project stages and an overview of each stage as well as a presentation of the available parameters and the interface imposed by Synopsys, after which the implementation details regarding the Galois arithmetic are described. Its main focus is the implementation of both the encoder and the decoder where a low level diagram and description is presented when possible. This chapter ends with the implementation details of an audio demonstration.

3.1 Project Flow

The development of this project was sectioned in several stages depicted in figure 3.1.

The first step in this project was a study of the available architectures. In this stage, the architectures available in the state of art were studied and briefly analysed in order to estimate which would have the lowest area requirements, without missing out on performance. The mathematics behind the selected architecture are present in the previous chapter, as well as the mathematical deductions required in order to properly implement a parallel interface.

The design of the testbench itself was split in several stages, where beforehand a C Model was design and validated against the Octave implementations, only then the SystemVerilog testbench was designed. More details on this stages is present on the next chapter.

The main stages, and the most time consuming ones in this project were the design and validation of the Verilog implementation itself using the VCS simulation tool from Synopsys. The implementation is covered in more detail in this chapter, whilst the the validation details as previously mentioned are detailed in the next chapter.

After proper validation of the implementation with the already mentioned testbench, the design was synthesised, formally verified and a static timing analysis performed with a tool set from Synopsys. The designed was synthesised using the Design Compiler, formally validated with the Formality tool, and finally the Static Timing Analysis was performed with PrimeTime.

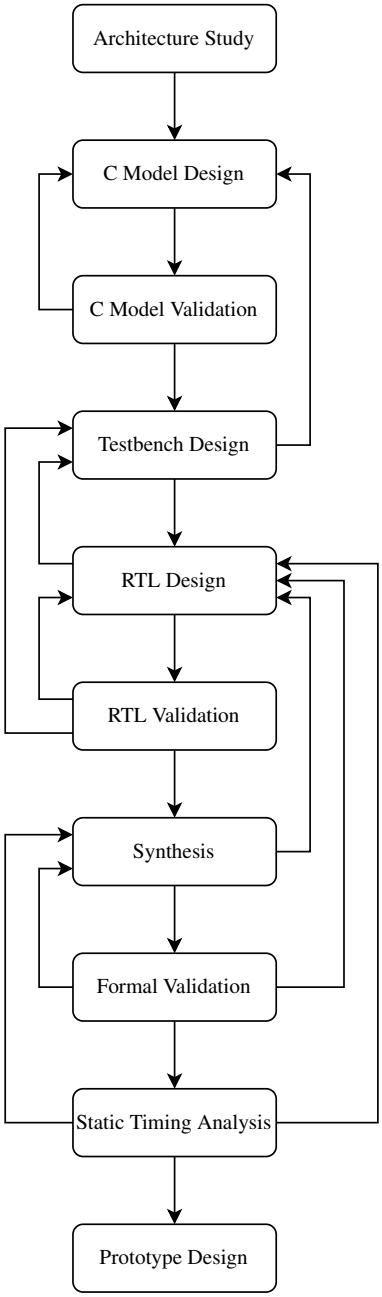

Figure 3.1: Project Flow Diagram

In order to verify that the design was working as expected and the validation environment was providing correct results, a prototype was designed and implemented in a FPGA. This prototype was validated and synthesised with Xilinx tools. The prototype design is covered in more detail at the end of this chapter.

3.2 Parameters and constants

This implementation is almost fully parametrized, both the encoder and the decoder share the same parameters. The Reed-Solomon generator polynomial should also be parametrized. However, since symbols are used instead of bits, each symbol might have more than one bit, and the Reed-Solomon generator polynomial coefficients are symbols, a problem arises due to the fact that Verilog does not support multidimensional parameters. To solve this several options were considered, from concatenating all the symbols in a larger symbol, to using pins. On the other hand, since the decoder requires the polynomial roots, not the polynomial itself, passing the polynomial coefficients is not very efficient, even though the the synthesis tools do not translate this calculations into hardware. A middle ground is found by passing the first polynomial root. According to equation 2.31, the first polynomial root is easily relatable to both, the remaining roots and the polynomial coefficients.

The parameter space of this implementation is given by the following table:

Table 3.1: Encoder and Decoder Parameters

Parameter Name	Width (bits)	Default	Description
GF_M	32	8	Galois Field symbol width
GF_P	GF_M+1	285	Galois Field primitive polynomial
RS_N	GF_M	255	Reed Solomon word code length (information + parity)
RS_K	GF_M	251	Reed Solomon word message length (information)
RS_FCR	GF_M	0	Reed Solomon polynomial first consecutive root exponent
BUS_WIDTH	32	9	Number of symbols processed per clock cycle

Although the most important parameters are covered, other settings are not supported in this implementation, as it only supports base 2 Galois fields and the field root is not customizable, being stuck at 2 as well.

As previously mentioned, some calculations have to be made during synthesis, in order to allow the tool to properly implement this, some care has to be taken when writing the hardware description. In order to properly initialize the required constants, they were declared as a *reg* and initialized inside a *always*@$(*)$ statement. This kind of description would never be evaluated during simulation, which is easily fixed by a preprocessor statement, like so:

```
1  `ifdef RSFEC_DEBUG
2  initial begin : SETUP_PROC
3  `else
4  always @(*) begin : SETUP_PROC
5  `endif // RSFEC_DEBUG

7    (...)

9  end //SETUP_PROC
```

Several constants are initialized within this block, in line 7, and explained ahead when necessary. For further reference, this initialization block will simply be described as $SETUP$.

Since several modules will use this initialization block, having to define it over and over again is error prone. Defining it in a separate file and including it within the modules, if necessary, is more flexible, less error prone and easier to maintain. On the other hand, this file must be explicitly excluded from both compilation and synthesis, since by including it with a preprocessor statement it gets indirectly compiled and synthesised.

3.3 Interface

Due to requirements imposed by Synopsys, the module it self has to follow a specific interface. This interface is represented in figure 3.2, and is composed of a parallel data bus, and three control signals, *start*, *end* and *valid*. The parallel data bus, denoted as *data*, is composed of BUS_WIDTH buses of $SYMBOL\_WIDTH$ bits each, where in the specification required by Synopsys is defined with $BUS\_WIDTH = 9$ and $SYMBOL\_WIDTH = 8$. The control signals behaviour is easily extractable from figure 3.2, the *valid* signal is set when there are usable data present in the data bus, the *start* and the *end* signals are set during the first and the last clock cycle of the message, respectively.

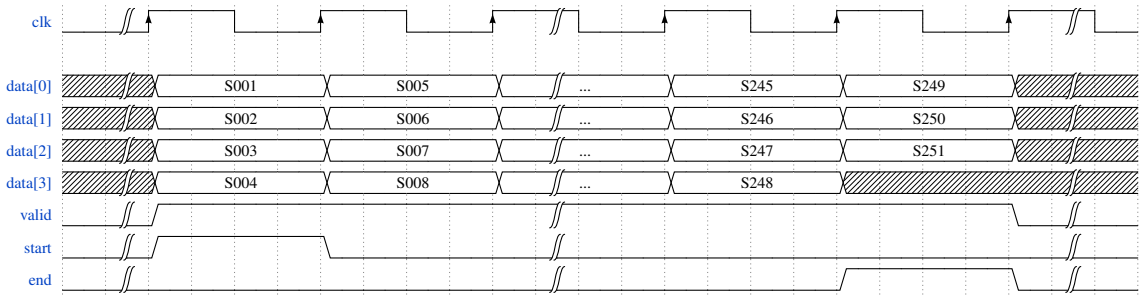

Figure 3.2: Interface with $BUS\_WIDTH = 4$

If the messages are continuous, that is, there are always valid data in the bus, then the valid signal is always on, and the beginning and the end of each message is only delimited by the *start* and *end* signals. This behaviour is visible in figure 3.3. .

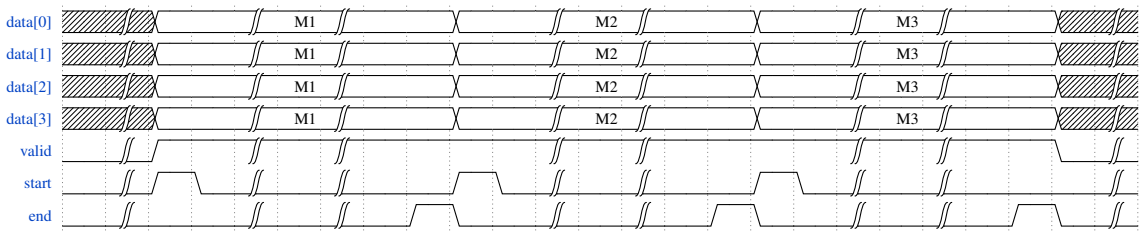

Figure 3.3: Interface with $BUS\_WIDTH = 4$ at maximum capacity

With $RS(251,255)$ and $BUS\_WIDTH = 9$, the input of the encoder input is composed of $\lceil 251/9 \rceil = 28$ clock cycles per message, the same as the decoder output, while both the decoder input and the encoder output are defined as a set of $\lceil 255/9 \rceil = 29$ clock cycles per message.

3.4 Galois Arithmetic

Galois operations are widely used throughout both the encoder and the decoder, its implementation has a great impact in the designs performance and consumption. Nowadays the synthesisers available on the market are very powerful and are great at propagating constants, therefore there is only the need for generic operators.

The adder is fairly simple, since, as described before, it is the exclusive-or of both operands.

The multiplication is quite complicated, however, since most of the multiplications have a constant as one of the operands, and the polynomial is known beforehand, passed as a parameter to the modules, the synthesiser has a lot of room for optimizations. The multipliers may be implemented in several different ways, however the design, as will be shown later, has several feedback loops, therefore, the multiplier has to be combinational, which means it must be implemented either with combinational logic, or with a lookup table. A lookup table is not feasible for larger fields, as its size increases exponentially. By defining the multiplier as a Verilog function, the description becomes easier to read, and gives more liberty to the synthesiser to implement as it finds more convenient. Since the multiplier is used in several modules, like the constants initialized in section 3.2, this function is declared in a separate file and included if needed.

The inversion is even more complex than the multiplication Unlike the multipliers, the inverters could be implemented using a sequential process, however, due to bandwidth requirements, this is either not feasible or unproductive, since it is possible to have several inverters and commutate between them, as needed. An implementation using a lookup table is less error prone and far easier do design. This lookup table is initialized within the block like the one previously described.

3.5 Encoder

The encoder organization is fairly simple, it is constituted of a single sub block, the parallel encoder, and a set of registers, on both the output and the input, as described in figure 3.4. Therefore, by design, the encoder has a minimum of two clocks of latency. The parallel encoder itself, has no latency, there is a direct path from the inputs, to its outputs, which has the purpose of simplifying its design.

Due to the concatenated form of the input bus, to facilitate readability, it is convenient to unpack the input bus into an array of bytes, like so:

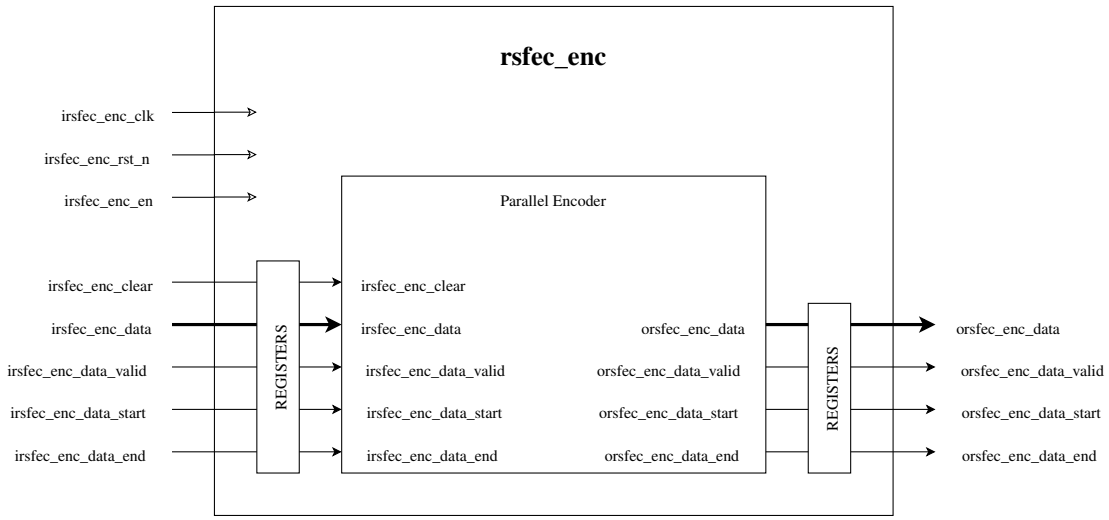

Figure 3.4: Encoder Top Diagram

```
always @(*) begin : irsfec_enc_data_unpack_PROC
  integer vi;
  for ( vi = 0; vi < BUS_WIDTH; vi = vi+1) begin
    rirsfec_enc_data[vi] = irsfec_enc_data[GF_M*vi +: GF_M];
  end
end // irsfec_enc_data_unpack_PROC

always @(*) begin : orsfec_enc_data_pack_PROC
  integer vi;
  for ( vi = 0; vi < BUS_WIDTH; vi = vi+1) begin
    orsfec_enc_data[GF_M*vi +: GF_M] = rorsfec_enc_data[vi];
  end
end // orsfec_enc_data_pack_PROC
```

Note that, in the previous snippet, GF_M is a parameter for the number of bits per symbol, and the variables used are declared as follows:

```
input  wire [GF_M*BUS_WIDTH-1:0] irsfec_enc_data       ,
output reg  [GF_M*BUS_WIDTH-1:0] orsfec_enc_data       ,
reg   [GF_M-1:0] rirsfec_enc_data       [BUS_WIDTH-1:0] ;
reg   [GF_M-1:0] rorsfec_enc_data       [BUS_WIDTH-1:0] ;
```

In order to improve readability, the $rirsfec\_enc\_data$ input signal, $rorsfec\_enc\_data$ output signal and BUS_WIDTH were replaced by single letters, I, O and ω respectively, and the constant $\delta = RS\_K \bmod \omega$ represents the number of valid bytes present at the input during the last clock cycle of the message being encoded. The figure 3.6 models an implementation of the parallel encoder, where the empty cloud represents the equations deduced in the previous chapter, that is, equations 2.39 and 2.40 describe the next state of the registers X, while inputs are being fed

into the module, while $irsfec\_enc\_data\_valid$ is set. Otherwise, the registers behave like a shift-register, each time the circuit outputs a set of symbols, the registers X are right shifted ω symbols, that is, $X_i \Leftarrow X_{i+\omega}$.

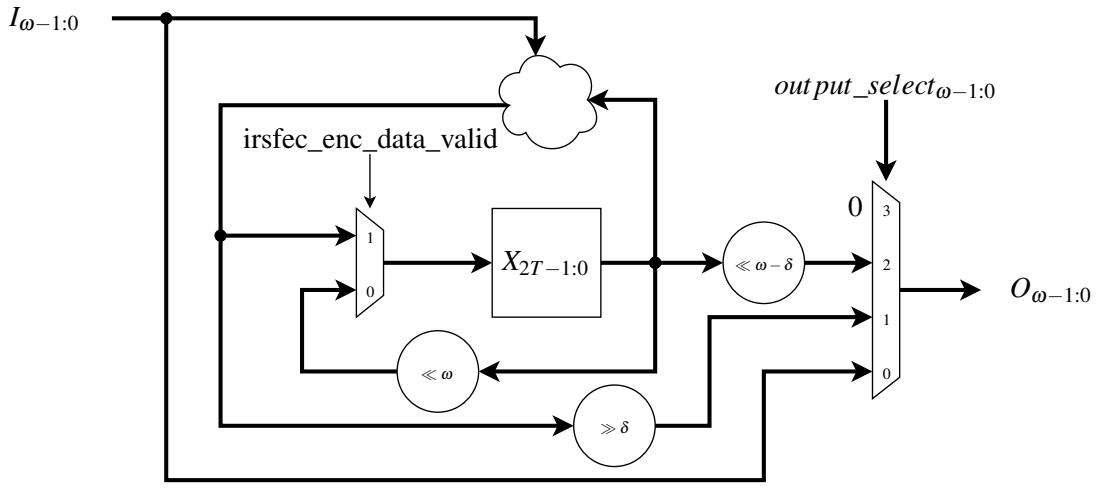

Figure 3.5: Parallel Encoder Diagram

During the last clock cycle of a valid message, there might be unused symbols in the interface, they must be filled with Reed-Solomon parity symbols. A left shift of δ symbols right after being calculated, before being clocked into the registers X, perfectly aligns them with the input signal. After the last clock cycle of valid message, if there are still more parity symbols to be clocked out, they must be left shifted $\omega - \delta$ symbols, in order to the compensate for the ones already outputted. A mux controlled by the signal $output\_select$, generated in the controller depicted in figure 3.6, where the cloud computes the following algorithm:

```
1  if (CNT < RS_K/ω))
2    output_select_{ω-1:0} = 0
3  else if (CNT > RS_K/ω))
4    output_select_{ω-1:0} = 2
5  else for i = 0:ω-1
6    output_select_i = (i < δ)
```

Where CNT is a simple counter that is used to count the number of clock cycles passed since the beginning of the message. All the other control signals are calculated from this counter and the input control signals.

3.6 Decoder

The decoder itself is split into three main sections, the Syndrome calculator, the Berleykamp algorithm, and the Chien search and Forney algorithm, described and generalized in section 2.3.2. These sections have a single controller responsible for all of them, as well as to manage the external memory used to store the message while it is being decoded. This structure is depicted in figure 3.7.

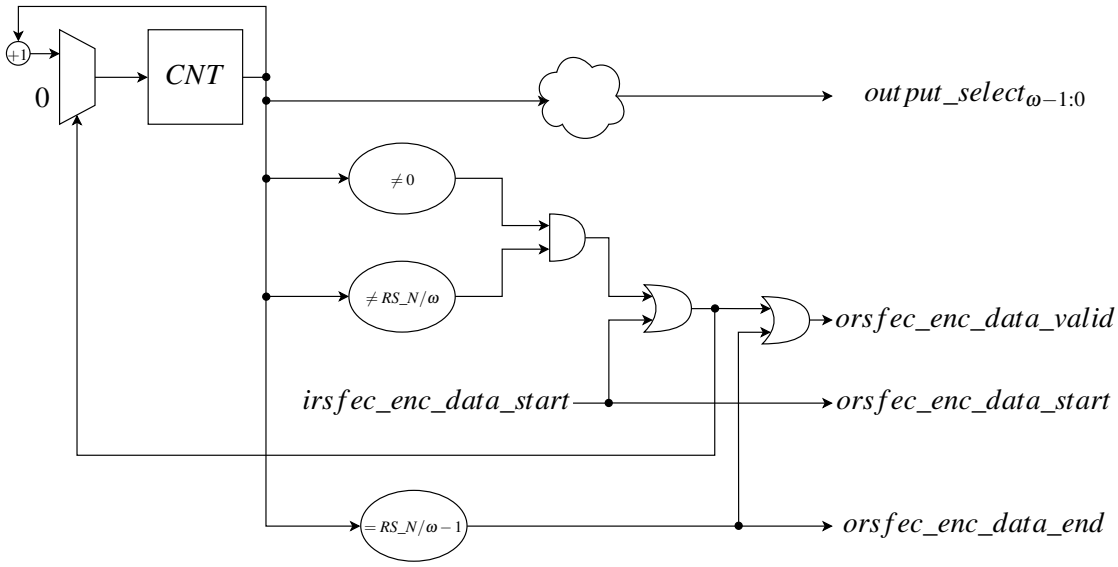

Figure 3.6: Encoder Controler Diagram

3.6.1 Syndromes

From equation 2.45 it is possible to visualize an interactive way of calculating the syndrome by using a LBSR, which prevents the need to calculate x^{2t-1}. On the other hand, this the inputs to be presented sequentially, but in the present system, there may be up $BW(BUS\_WIDTH)$ inputs per clock cycle. In order to bypass this situation, the received message if seen as a polynomial it can be represented as the sum of several polynomials:

$$r(x) = r_0 + \sum_{j=1}^{BW-1} (r_{N-BW+j} \cdot x^{N-BW} + \ldots + r_{j+2} \cdot x^2 + r_{j+1} \cdot x + r_j) \cdot x^j \tag{3.1}$$

Where each polynomial can be represented in the Horner method, like the equation 2.45, which results in the following equation:

$$r(x) = r_0 + \sum_{i=j}^{BW-1} ((\ldots(r_{N-BW+j} \cdot x^{BW} + r_{N-2BW+j}) \cdot x^{BW} + \ldots + r_{j+BW}) \cdot x^{BW} + r_j) \cdot x^j \tag{3.2}$$

As presented in equation 2.44, the syndromes can be calculated by replacing x in equation 3.2 by each of the Reed-Solomon polynomial generator roots (α^{b+i}) as follows:

$$S_{b+i} = r_0 + \sum_{j=1}^{BW-1} ((\ldots(r_{N-BW+j} \cdot \alpha^{(b+i)\cdot BW} + r_{N-2BW+j}) \cdot \alpha^{(b+i)\cdot BW} + \ldots + r_{j+BW}) \cdot \alpha^{(b+i)\cdot BW} + r_j) \cdot \alpha^{(b+i)\cdot j} \tag{3.3}$$

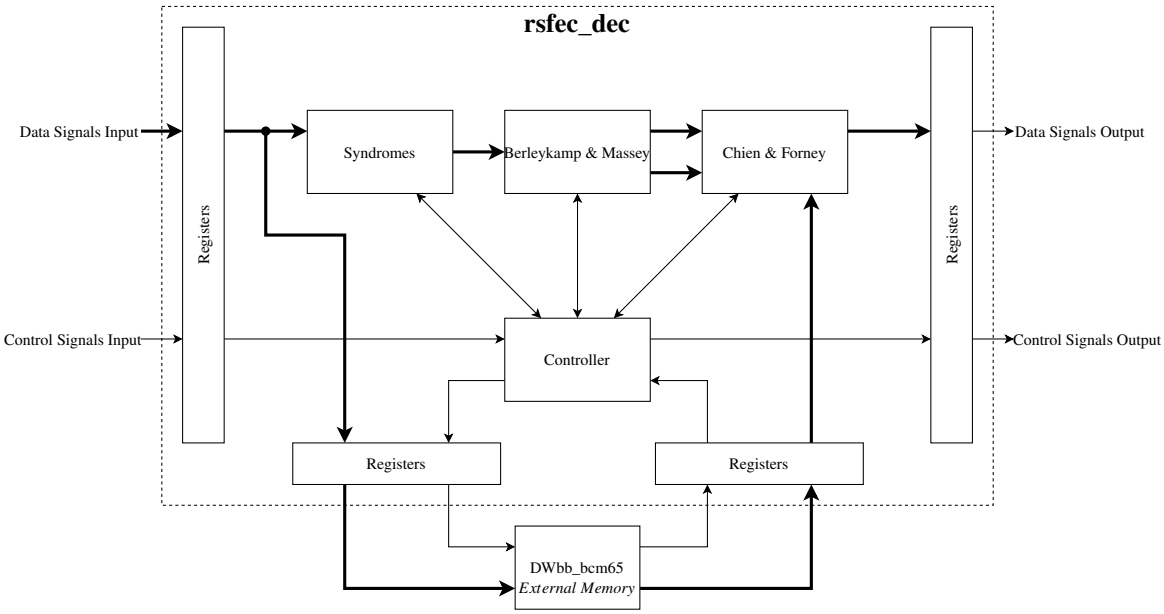

Figure 3.7: Decoder Top Diagram

The powers of α are properly initialized in the $SETUP$ block previously mentioned and form a matrix with the following content:

$$LP\_R = \begin{bmatrix} \alpha^{(b+0)\cdot 0} & \alpha^{(b+0)\cdot 1} & \alpha^{(b+0)\cdot 2} & \dots & \alpha^{(b+0)\cdot (BW-1)} \\ \alpha^{(b+1)\cdot 0} & \alpha^{(b+1)\cdot 1} & \alpha^{(b+1)\cdot 2} & \dots & \alpha^{(b+1)\cdot (BW-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha^{(b+2T-1)\cdot 0} & \alpha^{(b+2T-1)\cdot 1} & \alpha^{(b+2T-1)\cdot 2} & \dots & \alpha^{(b+2T-1)\cdot (BW-1)} \end{bmatrix} \tag{3.4}$$

With this constants properly initialized, a matrix of fed-back registers that iteratively add the input message multiplied by a constant depending on position of the register in the matrix can be depicted in figure 3.8. During the last clock cycle, the number of inputs might not be fully utilized, therefore, the input is bypassed and added to a delayed version of the previously accumulated result and the registers reset so that new syndromes can be calculated without a gap of invalid data in the input signal.

3.6.2 Berley

The equation (7) simply represent the Syndrome calculation, and the equations (29 - 30) will be implemented using a slightly simpler version in the sub-block of the Chien search and Forney algorithm, which requires $\omega(x) = \Lambda(x) \cdot S(x)$, however, in the presented algorithm $\omega(x) = \Lambda(x) \cdot (1 + x \cdot S(x))$. In order to normalize $\omega(x)$, the equation (28) is replaced by $\omega(x) = (\omega^{2t}(x) + \Lambda^{2t}(x))/x$.

The equations (8 – 12) are initialization steps, next the algorithm iterates over the syndromes, S_k, until the difference Δ goes to zero. Δ is calculated by convolving the syndromes with the current

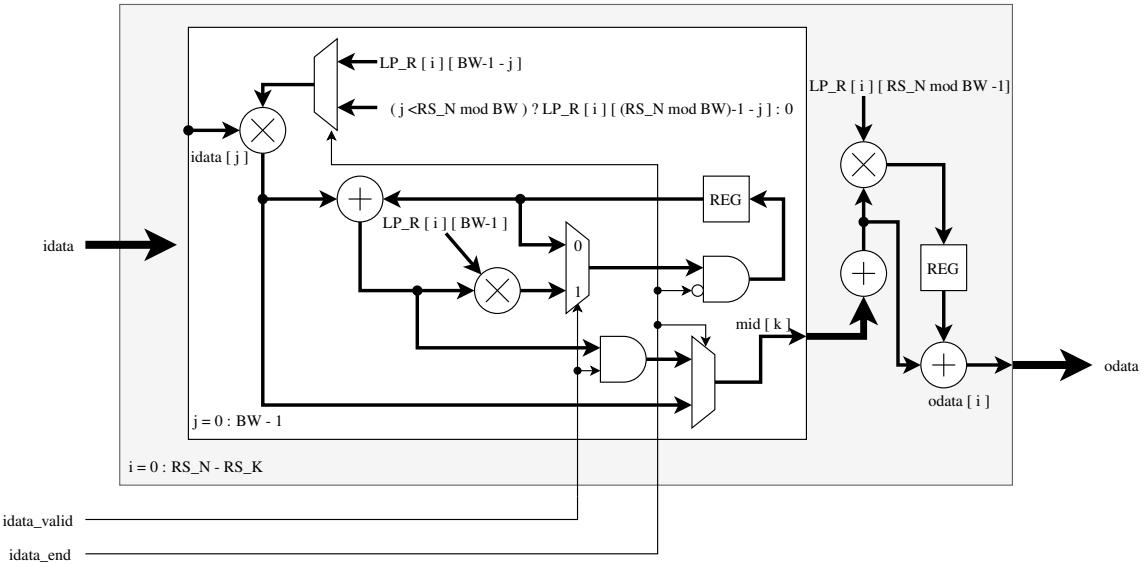

Figure 3.8: Decoder Syndrome Diagram

estimate of the locator polynomial, $\Lambda(x)$, as presented in equation (14). This can be achieved by definition, as seen on figure 3.9.

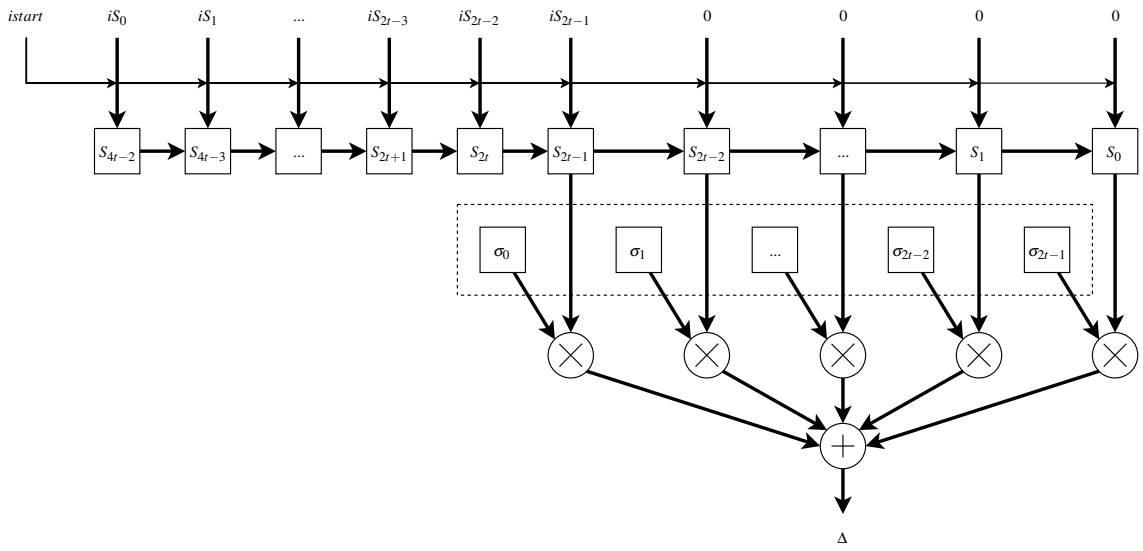

Figure 3.9: Decoder Berleykamp Delta Diagram

After calculating Δ, and updating the locator and the evaluator polynomials, $\delta(x)$ and $\omega(x)$ respectively, according to the equations (15 – 16), there are two options, whether the algorithm as arrived to a conclusion or not. This may be estimated from equation (17), being represented in the following diagram as $flag = (\Delta == 0 || 2D > k+1)$, where D and δ are updated if $flag == 1$, as described in the equations (21 – 22). This may be seen in figure 3.10.

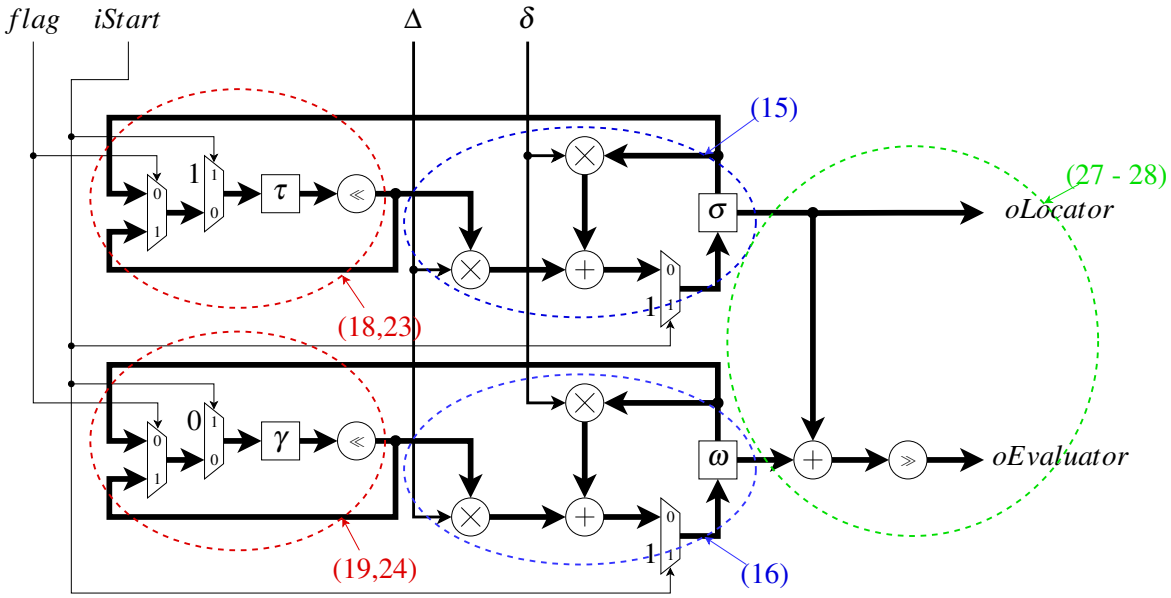

Figure 3.10: Decoder Berleykamp Algorithm Diagram

3.6.3 Chien & Forney

From equation 2.59 is noticeable five main steps, calculate X_j^{-b}, X_j^{-1}, from this, both $\Omega(X_j^{-1})$ and $\Lambda_{odd}(X_j^{-1})$, and finally invert the last one and multiply them.

Since $X_j^{-b} = X_1^{-b} \cdot X_{j-1}^{-b}$, starting with $j = 0$, it is possible to generate a sequence of X_j^{-b} with a register with a loopback multiplier. However, since the design must be able to compute BUS_WIDTH symbols at a time, a set with the same number of loopback multipliers is necessary, which means that the loopback multiplier constant has to be $(X_j^{-b})^{BW}$. The top left section of the figure 3.11 shows an implementation of this.

Due to the need to calculate polynomials of order $2T-1$ it is necessary to calculate $X_j^1, X_j^2, ..., X_j^{2T-1}$ for each symbol, and since it is required to compute BUS_WIDTH symbols per clock cycle, this becomes a crucial section of the design. In order to reduce area and power consumption, an effort was made to implement this section only using multiplications with constants. By representing the powers of the polynomial argument for the first clock cycle in a matrix form:

$$\begin{bmatrix} 0 & 0 & \dots & 0 \\ 1 & 2 & \dots & (2T-1) \\ \vdots & \vdots & \ddots & \vdots \\ BW-1 & 2\cdot(BW-1) & \dots & (2T-1)\cdot(BW-1) \end{bmatrix} \tag{3.5}$$

And comparing it with the expected powers in the next clock cycle:

$$\begin{bmatrix} BW & 2\cdot BW & \dots & (2T-1)\cdot BW \\ BW+1 & 2\cdot(BW+1) & \dots & (2T-1)\cdot(BW+1) \\ \vdots & \vdots & \ddots & \vdots \\ 2\cdot BW-1 & 2\cdot 2\cdot(BW-1) & \dots & (2T-1)\cdot 2\cdot(BW-1) \end{bmatrix} \tag{3.6}$$

It is possible to infer a pattern by subtracting both matrices. The following matrix is obtained:

$$\begin{bmatrix} BW & 2\cdot BW & \dots & (2T-1)\cdot BW \\ BW & 2\cdot BW & \dots & (2T-1)\cdot BW \\ \vdots & \vdots & \ddots & \vdots \\ BW & 2\cdot BW & \dots & (2T-1)\cdot BW \end{bmatrix} \tag{3.7}$$

This implementation is depicted in figure 3.11 in the top left corner, with the reset values for the register being described in 3.5 and the loopback multipliers constant in 3.7.

The remaining steps, calculate the polynomials and multiply them, is displayed on the same figure along side its bottom.

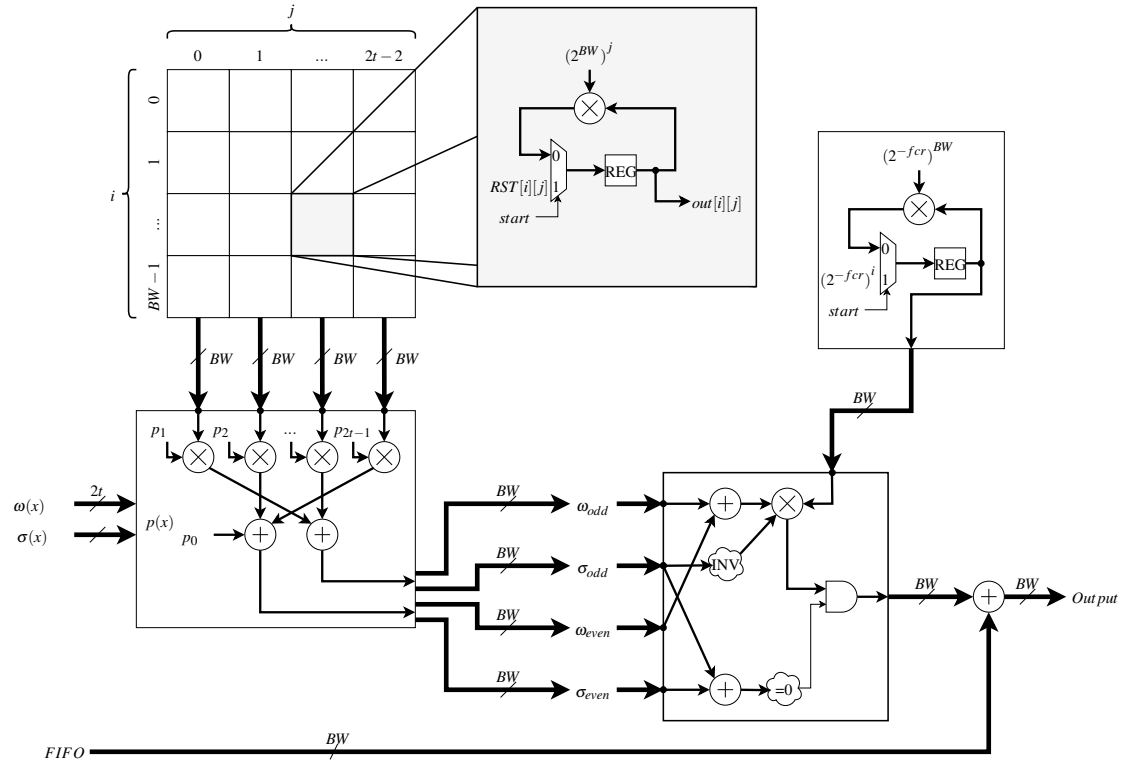

Figure 3.11: Decoder Chien Search and Forney Algoritm Diagram

An attempt of improving this block was also implemented, in which all the constants in the top left corner were replaced by their logarithmic form, and the multipliers replaced by adders. Right before the adders, when calculating the polynomials' values, lookup tables were used to convert back from logarithmic form. Despite the effort, the results were not satisfactory, and the attempt was dropped.

3.6.4 Controller

Although the controller could be segmented and implemented within the other modules themselves, by separating it from the remaining modules, and implementing it in a separate module, it

becomes makes the design more flexible and allows for a easier calibration during the implantation phase of the project. Like the controller present in the encoder, depicted in figure 3.6, it was implemented by counting clock cycles, however in this case a more complicated set-up is used.

There are a total of four counters within this block. The first one is used to initialize the DWbb_bcm65 fifo, its value is reset to 3 during the asynchronous reset, and counts down to 0. During which the output $orsfec\_dec\_bcm65\_init\_n$ is cleared. The remaining counters are implemented like so:

```
always @(posedge irsfec_dec_clk or negedge irsfec_dec_rst_n) begin
  if (!irsfec_dec_rst_n) begin
    rcnt1 <= {LP_CNT1_WIDTH{1'b0}};
  end else begin
    if (!irsfec_dec_en) begin
      rcnt1 <= rcnt1;
    end else begin
      if (rcnt1_rst) begin
        rcnt1 <= LP_CNT1_SET;
      end else if (rcnt1 == {LP_CNT1_WIDTH{1'b0}}) begin
        rcnt1 <= rcnt1;
      end else begin
        rcnt1 <= rcnt1 - 1;
      end
    end
  end
end
```

Where the LP_CNT1_WIDTH is the width of the register, which is determined by:

$$LP\_CNT1\_WIDTH = log2(LP\_CNT1\_SET + 1) \tag{3.8}$$

LP_CNT1_SET is the value for which the counter sets, when $rcnt1\_rst$ is active. The set values for the counters $CNT1$, $CNT2$ and $CNT3$ are RS_K/BUS_WIDTH, $RS\_N/BUS\_WIDTH - RS\_K/BUS\_WIDTH - 1 + RS\_N - RS\_K$ and $1 + RS\_K/BUS\_WIDTH$, respectively. LP_CNT1_SET is the number of clock cycles for which the data is pushed to the FIFO minus one, which is compensated by combinational logic. LP_CNT2_SET is the number of clock cycles that the block waits before it starts pushing data, the number of clock cycles used for the parity data plus the number of clock cycles necessary for the Berleykamp algorithm to operate properly. LP_CNT1_SET is the number of clock cycles for which the data is popped from the FIFO. The generation of the control signals is seen in figure 3.12, where $A = LP\_CNT3\_SET$.

3.6.5 Demonstration

After proper implementation and validation, which will be covered in the next chapter, a demonstration was implemented using a Xilinx Spartan-6 FPGA development kit from Digilent. This demonstration was based on a previous design provided by the supervisor, were the audio codec

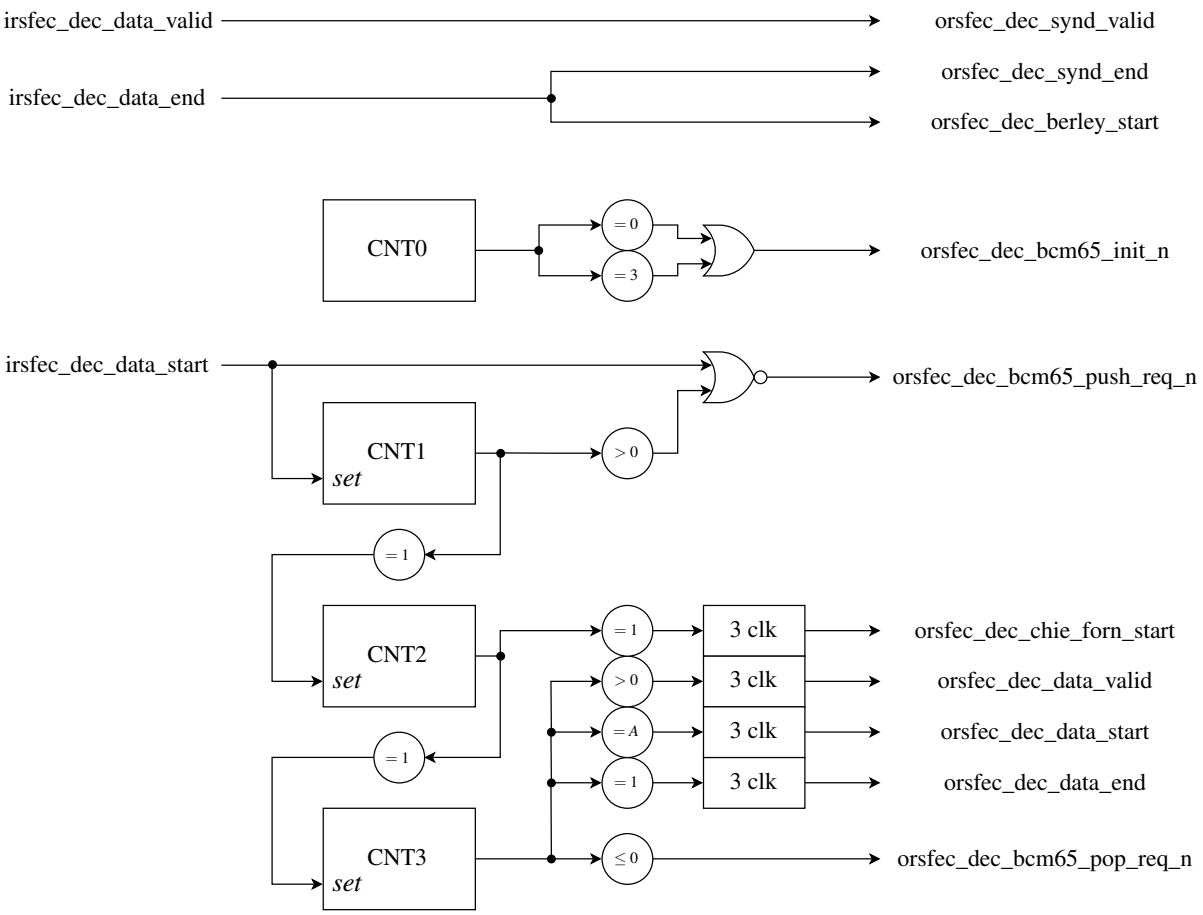

Figure 3.12: Decoder Controller Diagram

existent in the development kit is configured to both acquire and reproduce audio samples at a frequency of 48 KHz, through its ADC and DAC. The implemented Reed-Solomon modules and an exclusive-or which adds errors between the encoder and the decoder were inserted into the audio path as depicted in figure 3.13.

This block receives a set of audio samples from the $leftIn$ and the $rightIn$ signals and outputs a new set of samples into the $leftOut$ and $rightOut$ signals each time a pulse on the $smpl\_rdy$ appears at its input. This control signal denotes when the external audio codec in the board supplies a new sample. It also reads from a memory outside the block the error mask to be applied to the data. This memory is filled with an error pattern generated by a computer application. The samples are composed of 16 bits per channel but the default bus width for both the encoder and the decoder is 8 bytes per clock cycle, in order to simplify the design of this demonstration the last byte of the encoder input interface is a constant zero. This means that in each clock cycle two samples are processed.

Several new blocks are introduced here. The encoder and the decoder are the ones presented before, with the default parameters.The *mem_in* block works like a FIFO, but with different buses widths on each side and without the internal counter on the output side, therefore the read address

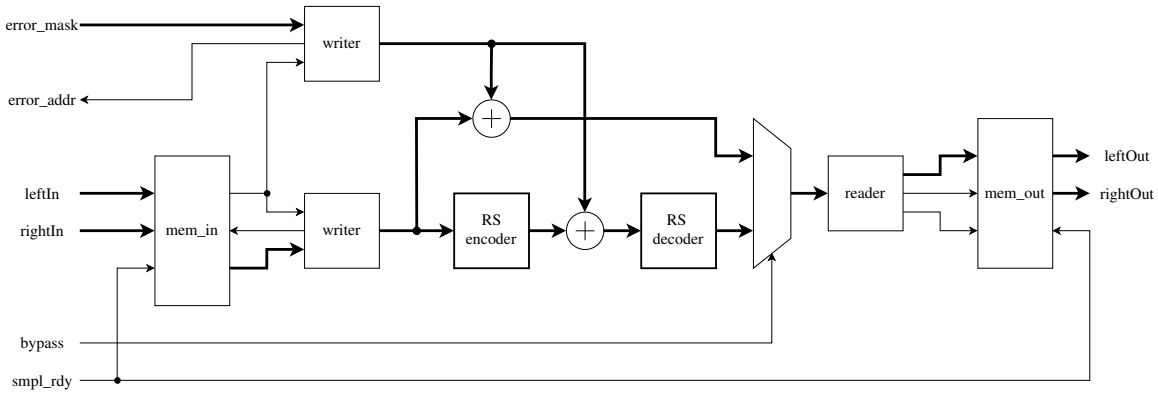

Figure 3.13: Demo Wrapper Diagram

is passed as an input. A signal is generated here denoting that the memory has been filled with 56 samples.

This signal starts the block *writer* which reads the input memory, two samples at a time and outputs them in the specified interface for 28 clock cycles, zero padding the extra byte each cycle. This block is replicated in order to read the error mask and generate a secondary parallel stream with it.

Errors which are added between the encoder and the decoder and to a secondary path that bypassed them. The enabled path is selected by a mux controlled by an input signal attached to a board switch. The mux output is passed to the *reader* block.

The *reader* block does the opposite of the *writer* block, it reads the samples from the interface and writes them to the *mem_out* block.

From the point of the view of the top design this module adds a delay of 56 samples, the required amount to fill the input memory. The samples are read from the input memory, encoded, decoded and written back to the output memory before the next sample is ready.

Chapter 4

Validation Environment

In this chapter a SystemVerilog validation environment based on C implementation of a golden model is described, where the a verification plan is descrived as well as the testbench overall structure and where and how the C model was integrated in the validation environment.

4.1 Verification Plan

In order to guarantee that the design works as expected, prior to the design of the hardware, a verification plan was specified, a golden model and a testbench implemented. Note that the testbench was designed with the HDMI specification and code coverage in mind, and therefore, only validates the designed for limited set of parameters.

The testbench checks the design for the following points:

- If the encoder and the decoder follow the specified output interface correctly;
- If the encoder successfully encodes a specified message;
- If the decoder successfully corrects a specified error mask;
- If the decoder successfully calculates the syndromes of a specified error mask;
- If the decoder successfully calculates the locator polynomial of a specified error mask;
- If the decoder successfully calculates the evaluator polynomial of a specified error mask;
- If the decoder output number of corrected errors is coherent with the specified error mask;
- If the Galois multiplier produces expected results;

The test vectors used are composed of:

- All the same symbols, with a random error count and values;
- All the messages with one symbol not zero, with a random error count and values;

- Random symbols, with all the possible single errors;
- Random symbols, with all the possible dual errors locations, with random values;
- Random symbols, with a random error count and values;

4.2 Testbench

The test bench is divided in two major groups, one for the decoder and another for the encoder. Each group is composed of the DUT itself and a golden model, alongside all the logic necessary to synchronize and compare them.

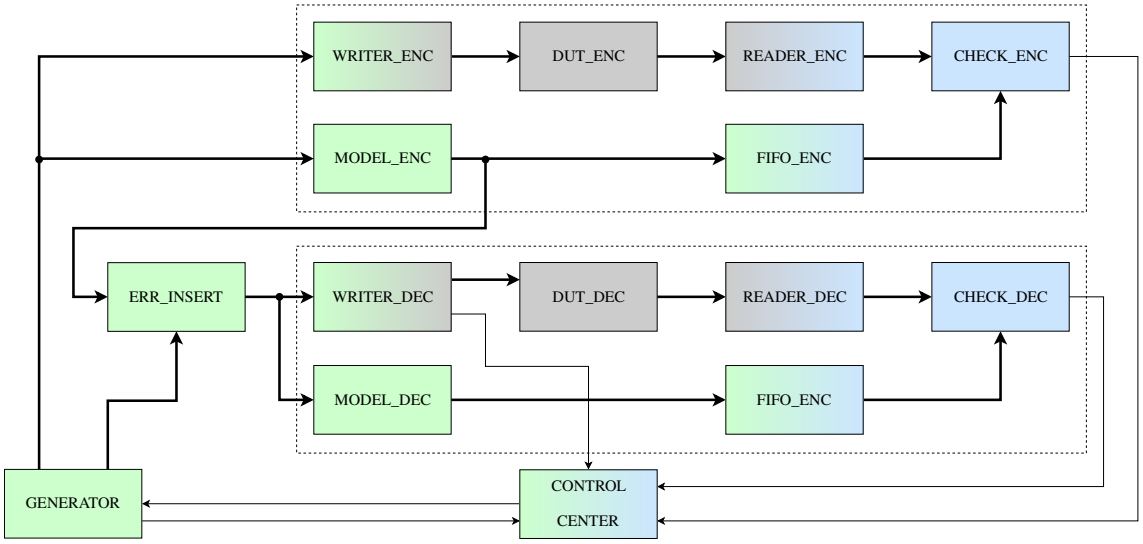

Figure 4.1: Testbench Diagram

A block named writer receives an array of symbols, the test message, and transforms that into a series of test vectors, synchronous to the testbench clock, designed to follow the interface required by the DUT. A similar interface is present at the DUT output, the block reader collects the output vectors and generates a ready signal if the interface was properly followed.

The golden model is implemented in a C function imported with the help of SystemVerilog DPI and encapsulated inside a block, simply named model, which calculates the expected values and pushes them to a $fifo$. A block named *check*, upon receiving the ready signal pops a set of values from the $fifo$, and compares them with the values interpreted by the reader, sending the result to the control center.

The control center, is a block which counts the number of errors detected, stopping the simulation if necessary and generating the start signal. This signal is responsible to start a new test. Upon receiving this signal the writer and model blocks start operating, using the array generated by the block generator.

The generator not only generates new test vectors but also controls which tests are being deployed. It generates two arrays of symbols, one to serve as input to the encoder group and another with a mask of the errors used to test the decoder.

A block denoted as *err_insert*, adds the error mask to the output of the encoder model, this results serves as an input to the decoder group of blocks.

In order to test both the encoder and the decoder in their maximum throughput scenario, the control center must enable the start signal before the arrival of the ready signal. To do so, it generates the start signal right after the writer block from the decoder group has finished stimulating the decoder. The control center also has a timeout for each the group, in case the reader does not detect a valid sequence outputted by the DUT.

The diagram 4.1 represents a simplified schematic of the process previously described. Note that in green are represented the blocks that are activated by start signal, in blue the ones activated by the ready signal and in grey the block synchronous with the testbench clock cycle.

4.3 Golden Model

Unlike the design itself, the golden model does not implement the interface signals, however it was implemented so that the testbench is able to verify not only if the inputs and the outputs are as expected, but if the syndromes, the error locator polynomial and the error evaluator polynomial are also being properly generated.

The algorithm used for the golden model was the same used for the Verilog implementation, and it was implemented in *C*. Since this was done before the testbench itself and the knowledge of the SystemVerilog DPI interface was not existent, a flexible programming was required in order to soften the adaptation from the Octave validation scripts to the SystemVerilog DPI. By implementing the most important functions within C libraries the adaptation to either necessity becomes feasible through wrappers.

The Octave validation scripts used system calls to execute pre-compiled C programs that implemented both the encoder and decoder through the previously mentioned library. Although not an efficient method, it proved to be good enough for its propose. This scripts used the octave functions *rsenc* and *rsdec* from the *communications package*, available in [10], to verify if the given input vector generated the adequate output. On the decoder side, the syndromes and the locator and evaluator polynomials were not verified as they may change depending on the algorithm used.

The SystemVerilog DPI (Direct Programming Interface) allows function calls between C and SystemVerilog. In this case it is only required that the SystemVerilog testbench calls C functions. As stated before, the required C functions were implemented in a library and a wrapper for the required functions was deployed. The following snippet illustrates the required declaration on the SystemVerilog side:

```
1  import "DPI-C" function
2    byte c_rs_encode(input longint v1 [], output longint v2 []);
```

The arguments for this functions are both arrays, one used as and input and the other used as an output. Since the arguments are both arrays, a C structure is required to as a medium, called $svOpenArrayHandle$ and defined in $svdpi.h$. Which along side a set of functions allows data to be read and written in the arrays by the C wrapper.

Chapter 5

Results and Conclusions

5.1 Overall Result

Throughout the project, several stages of the design were completed. From the verification plan, results of code coverage were obtained which are show in figure 5.1, and noted that both the encoder and decoder where fully covered.

Figure 5.1: Code Coverage Results

After completing the functional verification phase, synthesis for the target frequency and libraries was executed for both encoder and decoder, where the following results were obtained:

Table 5.1: Results for HDMI Spec, TSCM 28 nm @ 450 MHz

	Area (NANDs)	**Power Leakage** (mW)
Encoder	3186	10.6478
Decoder	33507	99.9320

Table 5.2: Results for four symbols per clock, TSCM 28 nm @ 450 MHz

	Area (NANDs)	**Power Leakage** (mW)
Encoder	1738	5.5091
Decoder	18240	52.9306

As stated in section 2.4, a parallel encoder and decoder are presented in [7]. The results of this alternative implementation are found in tables 5.3 and 5.4, for the encoder and decoder respectively. Note that in these tables the H stands for the number of symbols processed in each clock cycle, the gate count is an estimate of the number of gates for theoretical circuits, not actual implementations, and its unit is unclearif it stands for a two input NAND, as the paper does not specify it.

Table 5.3: Results taken from an alternative implementation of the encoder[7]

Complexity	Conventional encoder	Parallel encoder				
		H=2	H=3	H=4	H=5	H=6
Gate count	646	1068	1490	1912	2334	2756

Table 5.4: Results taken from an alternative implementation of the decoder[7]

Decoder modules	Conventional decoder	Parallel decoder		
		H=2	H=3	H=4
Syndrome calculator	641	1633	2622	3617
Error-locator poly. calculator	4439	4439	4439	4439
Error location detector	446	1180	1770	2360
Error evaluator	3857	7714	7714	7714
Error correction circuit	94	188	282	376
Delay	14672	15120	15624	15904
Total gate count	24149	30274	32451	34410
Partial gate count	9477	15154	16827	18506

The results turned out to be in the same order of magnitude of the ones already existent in the state of the art, although the implemented decoding algorithms were quite different. This is

to be expected, since most of the area used is on the interfaces themselves. After calculating the syndromes and until the error is found and corrected the algorithms are invariant to parallelization.

Both the encoder and the decoder passed without errors the formality and primetime post-synthesis tools.

5.2 Parametrization Effect

By varying the *BUS_WIDTH* parameter of the encoder and the decoder is possible to generate a set of graphics that help understand the effect of parallelization in the current implementation, as well as their benefits. Note that the dynamic power are only estimations, and should only be used as a relative metric. The baudrate per area was calculated using the area occupied by the design at the maximum detected frequency for its configuration. This frequency was found by applying a sweep to the target frequency and searching for the highest synthesizable frequency.

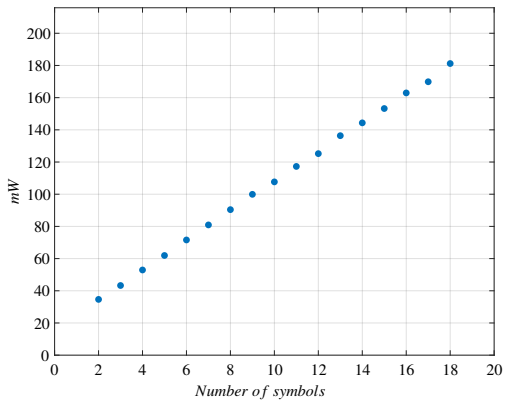

Figure 5.2: Decoder leakage power

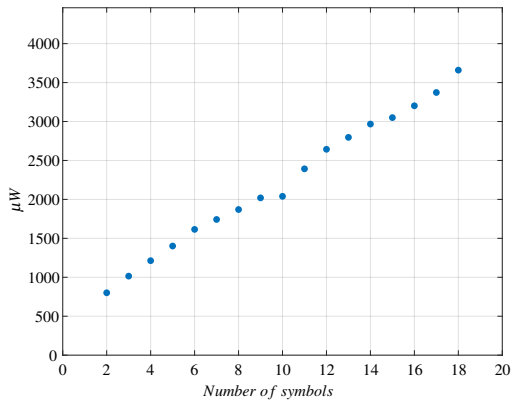

Figure 5.3: Decoder dynamic power

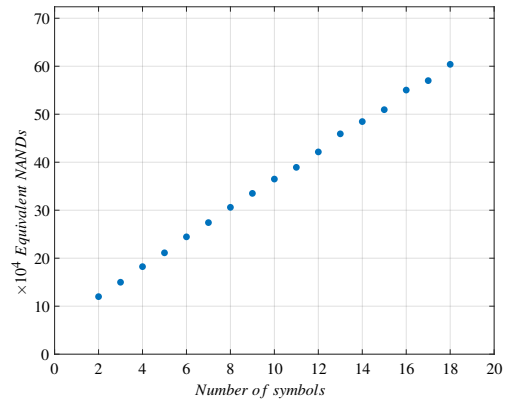

Figure 5.4: Decoder area

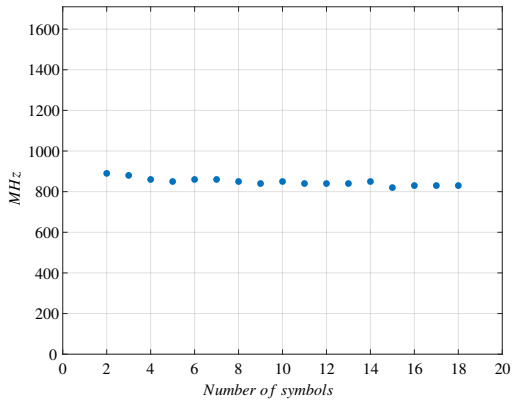

Figure 5.5: Decoder maximum frequency

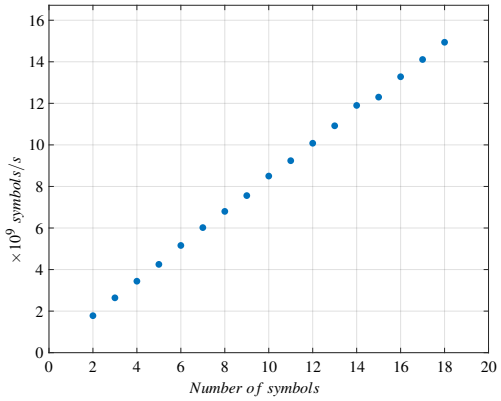

Figure 5.6: Decoder baudrate

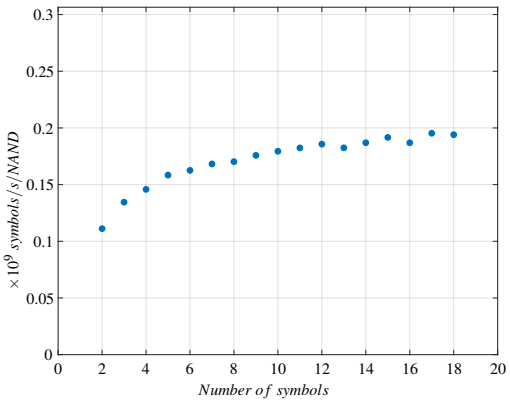

Figure 5.7: Decoder baudrate per area unit

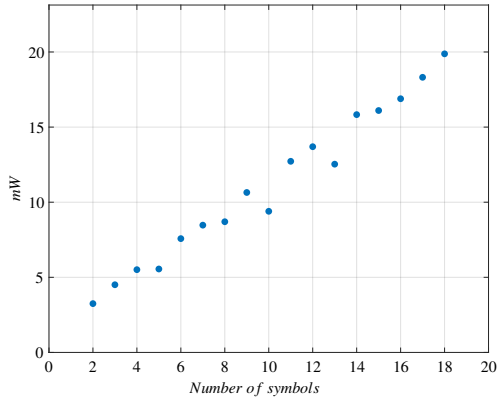

Figure 5.8: Encoder leakage power

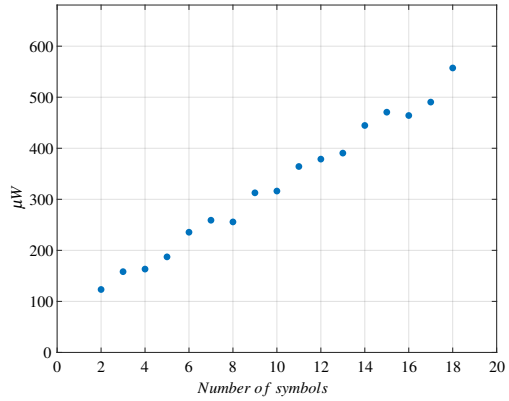

Figure 5.9: Encoder dynamic power

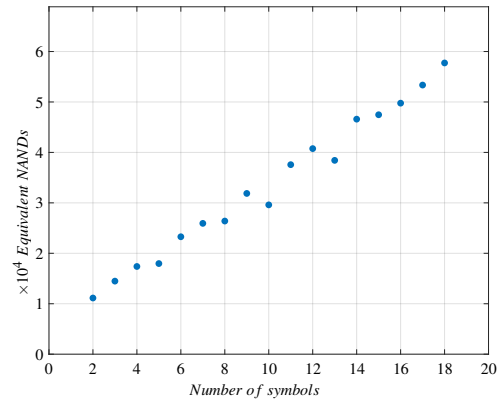

Figure 5.10: Encoder area

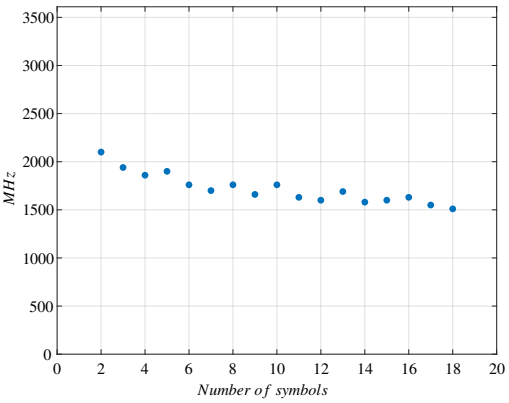

Figure 5.11: Encoder maximum frequency

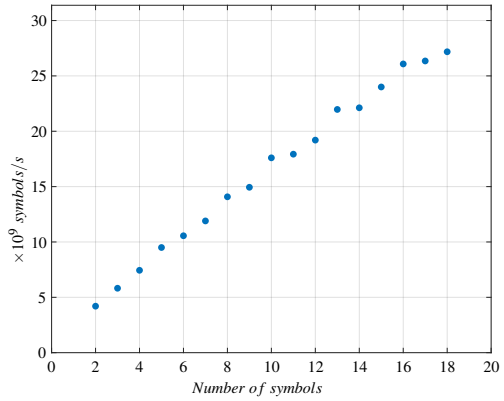

Figure 5.12: Encoder baudrate

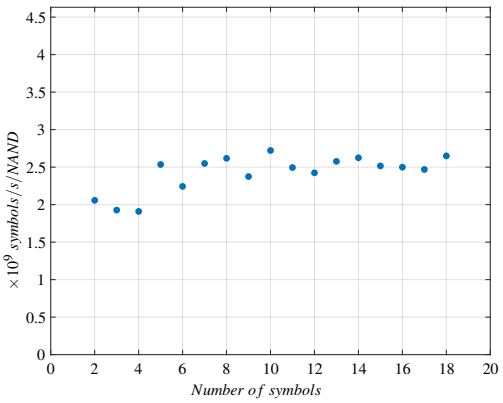

Figure 5.13: Encoder baudrate per area unit

The area occupied by the designs and its power consumption turned out to be a linear function of the number of processed symbols per clock and with a positive gradient as expected. The maximum frequency, as expected, shows a linear decreasing trend as the BUS_WIDTH increases, however not as a accentuated as the variation of the other two, as it seems. Since the operations can be organized as a tree, the critical path does not have much variation.

The plot of the baudrate per area presents an interesting result, the area efficiency is continuously increasing, although with lower and lower gains.

5.3 Conclusions and Future Work

The main objective of this dissertation was fully achieved. An implementation in Verilog for TSMC $28\,nm$ at $450\,MHz$ and above, that can handle computing nine symbols per clock using the specified interface and fulfils the functional requirements imposed by the HDMI specification. Not only was this completed, but the implementation itself is also almost fully parametrized.

The implementation results in line with the state of the art cell count, as well as being able to handle frequencies much higher than the ones imposed by Synopsys, making it able to handle throughputs of $15\,Gsymbols/s$, or $120\,Gbits/s$ on the encoder side, and $7.5\,Gsymbols/s$, or $60\,Gbits/s$, for the validated nine symbols per clock cycle and eight bits per symbol.

Although the objectives were completed, many improvements could still be applied The implementation has not been properly validated for every parameter combination, as the focus was the HDMI specification.

There are also many variants that should be explored in search for a more efficient architecture. The following options should be considered:

Sequential Inverter A Sequential inverter might be more area and power efficient than the implemented LUT, on the other hand the parallel interface and the required high baudrate could become a challenge.

Dual Basis Multiplier A dual basis multiplier should provide better efficiency, however this is only applicable to smaller Galois Fields [4]

Syndromes based on remainder The Syndrome calculation using a polynomial remainder parallel calculator, like the encoder, would reduce the register count, on the other hand, it could also increase the combinational logic area. However, due to its smaller size compared to the remaining circuit, little gain would be obtained.

Pipelined Chien & Forney Currently the decoder as the critical path is located in the Chien & Forney, a pipelined version would improve its performance. On the other hand, the required frequency is not high enough to justify the increased area caused by the addition of registers, at least for the HDMI specification's requirements.

References

[1] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *CHES*, volume 4727, pages 450–466. Springer, 2007.

[2] William A Geisel. Tutorial on reed-solomon error correction coding. 1990.

[3] CKP Clarke. Reed-solomon error correction. *BBC R&D White Paper, WHP*, 31, 2002.

[4] Douglas L Whiting. Bit-serial reed solomon decoders in vlsi. 1984.

[5] Daniel J Costello and G David Forney. Channel coding: The road to channel capacity. *Proceedings of the IEEE*, 95(6):1150–1177, 2007.

[6] Sangeeta Singh and S. Sujana. ASIC Implementation Of Reed Solomon Codec For Burst Error Detection And Correction. volume Vol.2 - Issue 4 (April - 2013). IJERT, April 2013.

[7] Tomoko K Matsushima, T Matsushima, and S Hirasawa. Parallel architecture for high-speed reed-solomon codec. In *Telecommunications Symposium, 1998. ITS'98 Proceedings. SBT/IEEE International*, volume 2, pages 468–473. IEEE, 1998.

[8] Akash Kumar and Sergei Sawitzki. High-throughput and low power architectures for reed solomon decoder. *a. kumar at tue. nl, Eindhoven University of Technology and sergei. sawitzki at philips. com*, 2005.

[9] Kenny Chung Chung Wai. Fpga implementation of reed solomon codec for 40gbps forward error correction in optical networks. 2006.

[10] Olaf Till Søren Hauberg. The 'communications' package, 2015. URL: https://octave.sourceforge.io/communications/index.html.