VHDL Implementation of a Modified Reed-Solomon Coding Algorithm with Considerably Reduced Latency for Short-Reach Optical Communications

Bo Zheng

Master of Electrical Engineering

Electrical and Computer Engineering

McGill University

Montreal,Quebec

2017-07

A thesis submitted to McGill University in partial fulfillment of requirements of the degree of Master of Electrical Engineering

ACKNOWLEDGEMENTS

I would first like to thank Prof. Odile Liboiron-Ladouceur who has always been incredibly kind, generous and helpful. She always cares my perspectives like my friend and she always supports me like my family. It was the inspiration she gave me that led me to the world of research. It was the tremendous pleasure of working with her that made that decision the best one I have ever made. I would also like to thank Prof. Warren Gross and Carlo Condo who offered guidance and help in this research. Their comments and corrections are significant to my thesis. My gratitude also goes to Rubana Priti, Yuli Xiong, Bahaa Radi and all other colleagues in my group, not only for the knowledge they shared with me but also for the kindness they treated me with. Last but not least, I'm grateful to my parents and my girlfriend for their support and love. I'm so lucky to have them in my life.

ABSTRACT

Error correction codes have been a necessary part of most optical communication systems. In their implementations, latency is always one of the top challenges, especially for short-reach optical communications. This concern is not only because of the constant demand for faster Internet speed but also because the exponential increase in Internet traffic is mainly driven by emerging applications like streaming video, social networking, and cloud computing, which is all data intensive and require high interaction between the servers. Research teams have come up with some solutions to reduce latency, but most of them considerably increase hardware costs and thus are limited. Nevertheless, a novel class of generalized Reed-Solomon (RS) codes was introduced with faster encoding and decoding algorithm. This low-latency RS codes can run the entire encoding and most parts of the decoding algorithm in parallel with only a slight increase in hardware costs. The speed-up effect is superior and for codes over $GF(2^8)$, for example, the coefficient can be as large as 15. In this thesis, the algorithm is explained in details with numerous examples. A verification of the algorithm in MATLAB using high-level coding technique is introduced, and a BER performance test is discussed. After that, the topic moves forward to the focus of the thesis: its VHDL implementation. Low-latency $RS(255, 225)$ encoders and decoders with different speed-up coefficients are implemented and illustrated in the thesis. The latency results and hardware costs are compared and discussed. Specifically, it is shown that for decoders, a latency reduction from 540 to 70 clock cycles have been achieved with a reasonable increase in hardware costs.

ABRÉGÉ

Les codes de correction d'erreurs ont toujours fait partis de la majorité des systèmes de communication optiques. Dans leur exécution, la latence a toujours été l'un des plus grands problèmes, notamment pour les communications optiques de courte distance. Cette préoccupation n'est pas seulement due à la hausse de la demande de l'Internet plus rapide, mais aussi par le fait que la hausse exponentielle du trafic en ligne est causée par les applications émergentes comme les vidéos en ligne, les médiaux sociaux et le informatique en nuage, c'est-à-dire l'information virtuelle, et ces derniers ont besoin de beaucoup d'espace sur le web et demandent d'interagir souvent entre les machines. Des équipes de recherche ont mis énormément d'effort et ont trouvé des solutions, mais la plupart demandent d'augmenter les coûts des matériaux et donc sont très limités. Toutefois, une nouvelle classe de Reed-Solomon (RS) de codes été introduit avec un algorithme capable de décrypter et d'encoder plus efficacement. Ce code RS avec faible latence est capable d'exécuter un encodage complet en plus de la plupart de l'algorithme du décodage en parallèle avec seulement une petite hausse en coûts. L'augmentation de la vitesse d'exécution est supérieure et le coefficient, par exemple, peut être aussi large que 15 pour les codes an haut de $GF(2^8)$. Dans cette thèse, l'algorithme est expliqué en détail avec plusieurs exemples. Une vérification de l'algorithme sur MATLAB utilisant du codage de niveau avancé est introduite et un examen sur sa performance BER est discutée. Après cela, le sujet avance sur l'aspect important de la thèse: l'implémentation du VHDL. La latence faible $RS(255, 225)$ d'encodage et de décodage avec des coefficients d'augmentation de vitesse ont été implémentée et illustrée. Les résultats de la latence et les coûts de matériaux correspondent sont comparés et discutés. Spécifiquement,

il est démontré par les décodeurs qu'une réduction de la latence de 540 à 70 cycles a été atteinte avec une hausse acceptable des couts des matériaux.

CONTRIBUTION AND CONTENTS OF THE THESIS

The thesis is done solely by the author (Bo Zheng). It starts with a general investigation of error correction codes and short-reach optical communication in Chapter 1. Then latency challenges in this area, as well as solutions developed so far, are introduced. In Chapter 2, conventional RS codes are reviewed along with Galois Fields (GF). Up to this point, readers should be fully prepared to explore low-latency RS codes in Chapter 3. Besides the definition and proof of low-latency RS codes, comprehensive instructions of encoding and decoding algorithms are also provided with examples. Afterward, it is time to discuss implementation. The information first focuses on algorithm verification in MATLAB, and then the testing for BER performance within communication system toolbox is presented. In Chapter 4, the VHDL implementation is presented with the code construction and result analysis. Finally, the results are summarized, and a conclusion is given in Chapter 5 along with future direction for the work.

ABBREVIATIONS

AWGN	additive white Gaussian noise
BCH	Bose, Chaudhuri, and Hocquenghem
BER	bit error rate
BM	Berlekamp-Massey
DFT	discrete Fourier transform
ECC	error correction codes
EDFA	erbium-doped fiber amplifier
EMI	electromagnetic interference
FEC	forward error correction
FPGA	field programmable gate array
FSM	finite state machine
FTTH	fiber to the home
GF	Galois field
GRS	generalized Reed-Solomon
IDFT	inverse discrete Fourier transform
LFSR	linear feedback shift register
LUT	look-up table
MDS	maximum distance separable
PAM	pulse amplitude modulation
RS	Reed-Solomon
SNR	signal-to-noise ratio
VHDL	VHSIC hardware description language
WDM	wavelength division multiplexing

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 Error Correction Codes

Transmission channels in communication systems are not perfect. Even for optical fibers which are well known for being one of the best transmission media, dispersion, scattering, and other impairments can easily undermine the transmitted data. This fact has led to the invention of error correction codes (ECC) which add redundant data and map the message following a particular way to a set of code-words. With ECC, the receiver can acquire the original message to a certain degree of accuracy that is called correction-words by decoding the received-words which refer to the received code-words with potential transmission errors. Coding theory was first introduced in [1] in 1948 by Claude Shannon where he specified the meaning of efficient and reliable information. Two years later, Richard Hamming defined a constructive generating method and the basic parameters of ECC [2]. Nowadays, ECC has become a necessary component in most communication systems. As Figure 1–1 shows, the data to transmit is sent from data source to source encoder where the data are compressed, and redundant bits are removed. Then in channel encoder, ECC is utilized by adding particular redundant bits to data. The modulator converts the sequence of bits out from the channel encoder into symbols suitable for the channel transmission. Passing through the channel which is the physical means of transmissions, the symbols received by the receiver can be corrupted. The demodulator converts the transmitted symbols back to data bits. Afterward, the channel decoder uses the redundancy bits added by the channel encoder to correct transmission errors. The source decoder converts

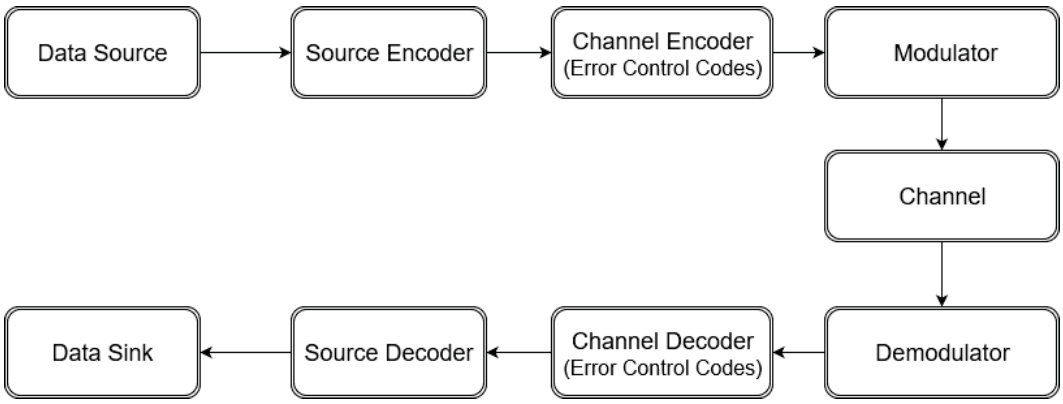


Figure 1–1: A typical communication system.

the data back to uncompressed representations which are finally sent to the sink.

The criterion for designing ECC includes: the probability of decoding errors should be minimized; the transmission of information should be dense and as fast as possible; the reproduced information at the channel decoder output should be reliable; the implementation cost of the encoder and decoder should be reasonable [3]. Structurally, there are two types of ECC: linear block codes and convolutional codes. The main difference between the two categories is that the first one uses only the symbols in the current set of message-words to produce the code-words, while the second one needs to remember some previous set of message-words [4]. Both of the famous Hamming and Reed-Solomon (RS) codes are examples of linear block codes. For several decades, research teams have developed efficient and reliable codes. Figure 1–2 gives an idea about the history of ECC development.

1.2 Error Correction Code in Optical Communication

In [5], Masataka Nakazawa and his team made a very nice description of the evolutionary history of ECC in optical communication. This paragraph is a summary and comments based on that description. Forward error correction (FEC) refers to the technique that uses ECC to estimate the original message.

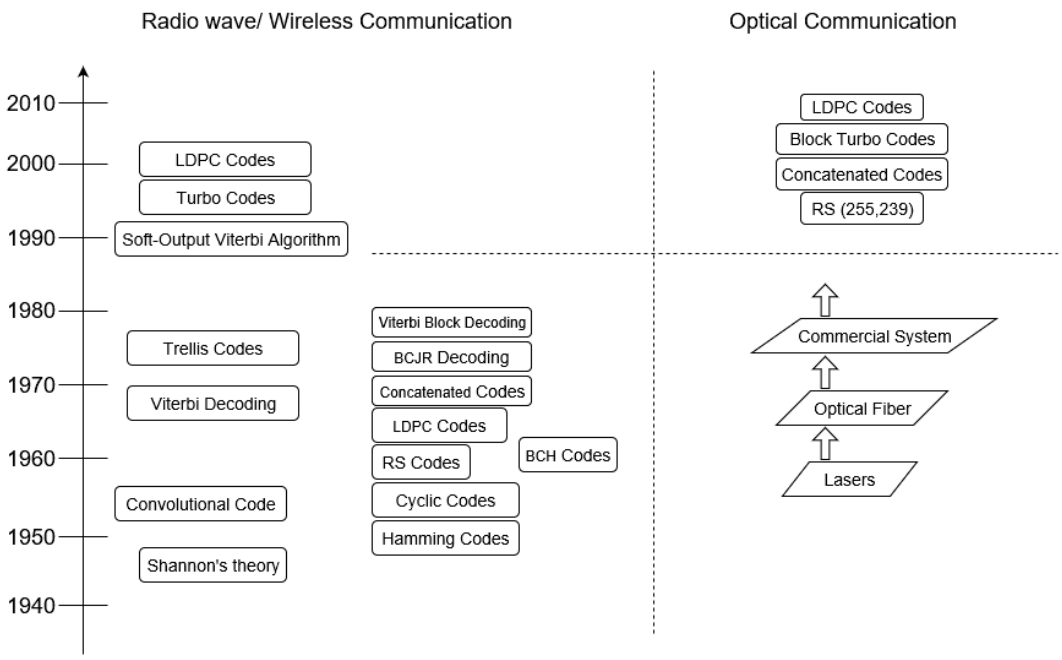


Figure 1–2: History of ECC development, based on [5].

As seen from Figure 1–2, FEC was initially ignored in optical fiber communication systems for a long time, because of its natural high data integrity leading to considerably small bit error ratios (BER) compared to conventional radio and satellite communications. One of the first published practical FEC experiments in optical fiber communications was reported by Grover employing shortened Hamming code (224, 216) in 1988 [6]. Then FEC started to appear in repeaterless submarine cable systems in the early 1990s. After erbium-doped fiber amplifiers (EDFA) had been applied in repeatered submarine systems, a problem with fluctuations in the BER caused by polarization-dependent effects showed up [7]. Nevertheless, RS codes mitigated the performance variation [8], which led to using EEC to gain system margin. These codes are often called the first generation. When wavelength-division multiplexing (WDM) was deployed widely, more powerful error correction codes are desired to satisfy the increase in signal-to-noise ratio (SNR) requirement due to the multiple numbers of multiplexed wavelength. One of the first proposal aiming to this

target was concatenated RS codes with iterative decoding. Ait Sab proposed $RS(255, 239) + RS(255, 223)$ with 22% redundancy and two iterations [9]. The codes developed in this stage are often called the second generation, and they drastically increased the attainable transmission capacity. Due to the continuous explosion of Internet traffic, the motivation was not only for larger capacity but also to reduce the cost. Terabit systems need expensive optical technologies such as ultra-wide band optical amplifiers, complex optical channel equalizers, and special grade premium fibers, while cheaper materials can be used but imply more errors. Therefore, stronger error correction codes based on soft-decision decoding are classified as the third generation. Also based on RS codes, Andrej Puc developed the first demonstration for soft decision in optical communications [10].

1.3 Short-Reach Optical Communication

For a very long time, optical links have been exploited only for long-haul communications. Only sharing among a large number of users can make them cost-effective [11]. Nevertheless, due to the decreasing hardware price and increasing demand for Internet bandwidth, short-reach communication attracts significant attentions now. Currently, fiber-to-the-home (FTTH) is already commercially available in many cities with affordable price. It is believed that fiber optics will eventually spread in any-size networks. Another focus point of short-reach optical communications is high-speed optical interconnects. Especially for data centers where thousands of servers interconnected with high bandwidth packet switches, optical interconnects are the most promising solution to provide high throughput, low latency, and reduced energy consumption [12]. Furthermore, the interconnects become a bottleneck for performance, as the processing speed of chips is continuously increasing fast.

Using optics for board to board, chip to chip and even on-board interconnects are being intensively studying.

Francisco Aznar and his team gave an excellent comparison between optical and electrical communications in [11]. The rest of this paragraph is the summary of their major findings. First, optical fibers do not suffer from electromagnetic interference (EMI) as electrical wires do. Optical carriers have no charge whereas electrical ones with high-speed signal may act as a transmitting antenna and radiate noise, possibly causing interference-related problems with neighboring circuits. Second, electrical transmissions need Galvanic separation that to solve ground loops because of variation of ground potential, while optical transmissions have no such problems because they provide an inherently isolated data path. Third, optical transmissions are safer than electrical ones because no electrical current is conveyed. Fourth, an optical fiber has much lighter weight than an electrical wire. Fifth, glass for optical fibers is obtained from sand and thus is more environmental-friendly than electrical wires which are made from copper and other metals.

1.4 Latency Challenges in Short-Reach Optical Communication

Latency is always one of the most important requirements for any communication systems. As mentioned above, besides the constant demand of higher Internet speed, the latency of optical interconnects among boards, chips, and servers are also urgent to decrease. The exponential increase of Internet traffic is driven by emerging applications like streaming video, social networking and cloud computing, which are all data-intensive and require high interaction between the servers [12]. Furthermore, latency reduction can indirectly decrease the cost of optical communications. Larger error-correction capacity leads to longer decoding time. With lower latency, larger error-correction capacity can

be accommodated in applications with a certain time constraint, which means cheaper but lossier materials can be used in the system.

Research teams have put many efforts in latency reduction of ECC, especially for RS codes as it is one of the most widely used classes. Up to now, several solutions have been proposed. One solution is to modify the formulas to double or triple the number of inputs of some steps so that those steps of decoding algorithm can run in parallel. However, the cost is to double or triple the hardware used in those steps, which obviously makes this solution very limited [13, 14]. Another solution is multi-ECC concatenation. Different type of ECC are concatenated together to achiever better code gain and relatively lower overall latency [15, 16]. However, this category often involves soft-decision decoding algorithms which are not generally suitable for short-reach communications whose power budgets are usually smaller than the power requirements of soft-decision algorithms.

There is one promising solution developed by Amin Shokrollahi in [17]. It is essentially a novel class of generalized RS (GRS) codes, which is referred as low-latency RS codes in this thesis. GRS codes were developed based on RS codes several decades ago and are a generalized version of RS codes. Instead of the set of consecutive roots in a conventional RS codes, Dr. Shokrollahi defined this set of roots to be closed under multiplication with a p-th root of unity over the same Galois field (GF). In this way, a conventional generator polynomial can be split into p generator polynomials, and consequently the encoder use them to generate a set of code-words consisting of p components. All the components can be constructed and decoded at the same time with little extra effort of component-wise Fourier transform. In other words, most of the encoding and decoding procedures can run in p-parallel, where p refers to the speed-up coefficient and can be even more than 10 for many scenarios.

However, the key equation solver and the error evaluation and correction process are not discussed in [17]. In this thesis, we developed a p parallel error evaluation and correction process to make output data-rate consistent with input data-rate. Overall, there are three significant novel features of this design compared with other solutions. First, due to the nature of this proposed design, the increased cost of hardware for this speed-up is low and thus the area and power advantages are outstanding. Second, the speed-up coefficient, that is the scalability, can be very large for this solution. Third, both latency and throughput are considerably improved. The detailed instruction and VHDL implementation of the algorithm constitute the focus of this thesis.

CHAPTER 2
Conventional Reed-Solomon Codes

2.1 The Concept of Reed-Solomon Codes

RS codes developed by Irving S. Reed and Gustave Solomon in 1960 [18] is a special subclass of q-ary non-binary Bose, Chaudhuri, and Hocquenghem (BCH) codes where q refers to the number of elements in the Galois field (GF) on which the codes are built. The details of Galois field are introduced in next section. In other words, instead of using binary signals in BCH codes, RS codes uses non-binary symbols as the unit of data. For example, for binary BCH codes, the encoded data can be [1 0 1 0 1 0 0] but for RS codes, the encoded data can be [3 5 7 2 4 5] (equivalent to [011 101 111 010 100 101]). $RS(n, k)$ means Reed-Solomon codes with the code-word length n and the message-word length k. The message-words are the block of data to transmit, and the code-words are the encoded messages transmitted through the channel. Moreover, $t = (n-k)/2$ is the error-correction capability, that is the maximum number of errors that can be corrected in a code-word with length n. This equation is derived from the fact that the minimum distance for RS coding theory is $2t + 1 = n - k + 1$. The minimum distance is the space between two possible code-words. The reason for $2t + 1$ is from the BCH bound and singleton bound [19]. The lower bound for the distance must be one unit larger than $2t$. In an easier way to see this condition, imagine two points representing two sets of code-words. For each point, draw a circle with the point as the center and radius $r = t$. If the distance between the two points is $2t$, there will be an intersection point between the two circles (Figure 2–1). Therefore, on that point, one cannot decode it correctly. The upper bound is following the

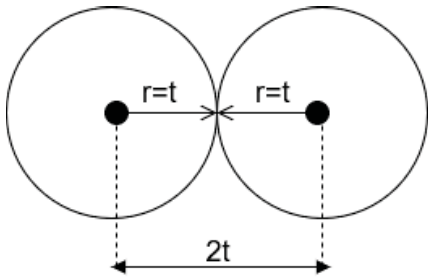


Figure 2–1: Unsatisfied minimum distance for RS codes.

singleton bound that $A_q(n,d) \leq q^{n-d+1}$ where A_q is the maximum number of possible code-words in a q-ary block code of length n and minimum distance d. Furthermore, the fact of satisfying the singleton bound qualifies RS codes as Maximum Distance Separable (MDS) codes [19]. MDS codes are a class of ECC which have the greatest error-correction capacity for given n and k.

Example 1. $RS(7,3)$ handles message-word with length $k = 3$ (for instance, $msg =$ [6 4 2], that is actually [110 100 010]) and produces a code-word with length $n = 7$ (for instance, $c =$[6 4 2 5 7 1 2]). The error-correction capability t for $RS(7,3)$ is 2 $(= (7-3)/2)$, which means that to successfully acquire the correct original message in the receiver, there can be at most 2 symbols in the c corrupted during the transmission in a channel (for instance, $r =$[6 7 2 5 7 0 2]).

For systematic RS codes which most modern RS codes are, the code-words consist of the original message-words and the parity-check bits. In the previous example, we can see the first three digits in $c =$[6 4 2 5 7 1 2] are the original message-words $msg =$[6 4 2] and the last four are the parity-bits. The parity-bits are computed using arithmetics over a specific Galois field.

2.2 Galois Fields

Almost all the arithmetic process of RS codes is based on Galois field (a.k.a. finite field). A Galois field $GF(q)$ is a field that contains a finite number

of elements on which the operations of multiplication, division, addition, and subtraction follow certain rules and are closed in the set, that is, results of the operations are still inside the set. The q from the notation $GF(q)$ represents the number of elements in a specific Galois field and the q must be equal to a power of a prime number p [20]. If the exponential power is 1, that is, q is equal to the prime number p, then $GF(q)$ is called a prime Galois field $GF(p)$. On the other hand, if the exponential power is larger than 1, then the corresponding $GF(q)$ is called extension Galois field $GF(p^m)$. Specific to our context with digital communication applications, RS(n,k) uses an extension Galois field $GF(q) = GF(2^m)$.

Specifically, $GF(q) = GF(2^m) = \{0, 1, \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{(q-2)}\}$, where α is called primitive element satisfying $\alpha^0 = \alpha^{(q-1)} = 1$ and all elements in the field must be distinct. The primitive element α should not be thought as a real number. Instead, think of it as a mathematical symbol, and consistently $\alpha^{(q-1)} = 1$ does not mean $q-1 = 0$. The q here can be any power of 2 ($q = 2^m$). The corresponding arithmetic operations are following the modulo-2 rule (note that if $p = 3$ and thus $q = 3^m$, then module-3 rule would be followed). For example, $\alpha^2 + \alpha^2 + \alpha^2 = \alpha^2$, and $\alpha^2 + \alpha^2 = 0$.

A polynomial over a prime Galois field $GF(2)$ is a univariate polynomial whose coefficients are from $GF(2) = \{0, 1\}$ and the corresponding polynomial arithmetic (addition, subtraction, multiplication, and division) is based on modulo-2. A primitive polynomial of degree m over $GF(2)$ is defined as an irreducible polynomial $p(X)$ of degree m if the smallest positive integer n, for which the primitive polynomial $p(X)$ divides $X^n + 1$, is $n = 2m - 1$ [20]. It should be mentioned that the m here is not necessarily 1 because it is not used to define $GF(2)$. Instead, this m is to later define the extension field of $GF(2)$, that is, $GF(2^m)$.

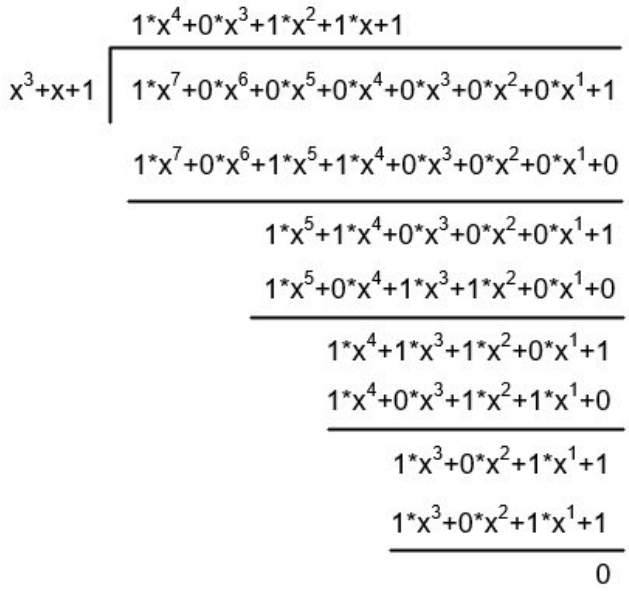


Figure 2–2: Division process for $(x^7+1)/(x^3+x+1)$.

Example 2. For example, $p(X) = X^3 + X + 1$ divides $X^7 + 1$ (Figure 2–2) but does not divide $X^n + 1$ for $1 \leq n < 7$.

There are three ways to represent each extension field $GF(2^m)$: 1) power representation, 2) polynomial representation, and 3) m-tuple representation. These three representations can be converted to each other based on the needs of the circumstances. The three representations of an extension field $GF(2^m)$ can be completely built on a given primitive polynomial $p(x)$ over $GF(2)$. It should be noticed that an extension field $GF(2^m)$ has different primitive polynomials and with each specific primitive polynomial $p(x)$, the representations are different correspondingly. The default primitive polynomials for different order m in MATLAB are shown in Table 2–1.

Example 3. For RS(15,11), the $n = 15$ and thus $q = n + 1 = 16$. Therefore, we are using $GF(16) = GF(2^m) = GF(2^4)$ for this RS code and $m = 4$. For power representation, $GF(2^4) = \{0, 1(1 = \alpha^0 =$

$\alpha^{15}), \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{14}\}$. From Table 2–1 we can find the default primitive polynomial in MATLAB for $m = 4$ is $p(X) = X^4 + X + 1$.

m	Default Primitive Polynomial in MATLAB
1	$X + 1$
2	$X^2 + X + 1$
3	$X^3 + X + 1$
4	$X^4 + X + 1$
5	$X^5 + X^2 + 1$
6	$X^6 + X + 1$
7	$X^7 + X^3 + 1$
8	$X^8 + X^4 + X^3 + X^2 + 1$
9	$X^9 + X^4 + 1$
10	$X^{10} + X^3 + 1$
11	$X^{11} + X^2 + 1$
12	$X^{12} + X^6 + X^4 + X + 1$
13	$X^{13} + X^4 + X^3 + X + 1$
14	$X^{13} + X^4 + X^3 + X + 1$
15	$X^{15} + X + 1$
16	$X^{16} + X^{12} + X^3 + X + 1$

Table 2–1: Default primitive polynomial for different order m in MATLAB.

Then, we can substitute the primitive element α into $p(x)$ and get the equation $0 = \alpha^4 + \alpha + 1$. This equation is used to find out the corresponding polynomial representation in which we use polynomials of the maximum order $(m-1) = 3$ to represent each element. For example, α^4 can be represented by $(\alpha + 1)$ due to the equation $0 = \alpha^4 + \alpha + 1$ (negative sign is equivalent to the positive sign in modulo-2 arithmetic). And then $\alpha^5 = \alpha \cdot \alpha^4 = \alpha \cdot (\alpha + 1) =$

$\alpha^2 + \alpha$ and so on so forth until the element α^{14} is calculated, as $\alpha^{q-1} = 1$ and $q - 2 = 14$.

The m-tuple representation (4-tuple representation in Example 3) is produced according to the polynomial representation. Each bit in the 4-tuple representation represents each coefficient of each polynomial term. For instance, 0111 represents the polynomial representation $\alpha^2 + \alpha + 1$. All three representations for the elements of $GF(2^4)$ generated by the primitive polynomial $p(X) = X^4 + X + 1$ are shown in Table 2–2.

Power Representation	Polynomial Representation	4-Tuple Representation
0	0	0000(=0)
$\alpha^0 (= 1)$	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 2–2: Three representations for the elements of $GF(2^4)$ with the primitive polynomial $p(X) = X^4 + X + 1$.

A polynomial over Galois field $GF(2^m)$ is a univariate polynomial whose coefficients are from $GF(2^m)$ and the corresponding polynomial arithmetic (addition, subtraction, multiplication, and division) is based on modulo-2. In Reed-Solomon codes, the message polynomials, generator polynomials, and the code-word polynomials are all in this category. When a k-length message-words are ready to transmit, it is translated into a message polynomial of order $(k-1)$ over a specific Galois field $GF(2^m)$. It should be noticed that the message-word length is k, that is, there are k coefficients in this message polynomial. A generator polynomial is a polynomial over Galois field to calculate the parity-check bits mentioned in the last section. For each RS(n,k) with a specific value of $(n-k)$, there is a particular generator polynomial of the order $(n-k)$ correspondingly. The code-word polynomial is the final result when we encode a message, and the coefficients are the encoded data to transmit.

Example 4. For RS(7,3), we are using $GF(8) = GF(2^m) = GF(2^3) = \{0, 1(1 = \alpha^0 = \alpha^7), \alpha, \alpha^2, \alpha^3, \cdots, \alpha^6\}$. If the message-word to transmit is [5 6 4], that is, [101 110 100], then the message polynomial over $GF(2^3)$ is $msg(x) = \alpha^6 X^2 + \alpha^4 X + \alpha^2$ (see Table 2–3). Suppose that after a certain encoding process the resultant code-word polynomial is $c(x) = \alpha^6 X^6 + \alpha^4 X^5 + \alpha^2 X^4 + \alpha^5 X^3 + X^2 + \alpha X + \alpha^2$. Then the final resultant code-word is [5 6 4 7 1 2 4], that is, [101 110 100 111 001 010 100]. The next section illustrates how all these polynomials are derived.

The concepts of arithmetics over $GF(2^m)$ are straightforward, while some of them are hard to implement in hardware, which is discussed in Chapter 4. Addition can be computed by simply using polynomial or m-tuple representations of elements. For example, for $GF(2^3), \alpha^3 + \alpha^5 = (\alpha+1) + (\alpha^2+\alpha+1) =$ [011]+[111] $= \alpha^2 =$ [100]. Subtraction is the same as addition due to modulo-2

Power Representation	Polynomial Representation	3-Tuple Representation
0	0	000(=0)
$\alpha^0(=1)$	1	001(=1)
α^1	α	010(=2)
α^2	α^2	100(=4)
α^3	$\alpha+1$	011(=3)
α^4	$\alpha^2+\alpha$	110(=6)
α^5	$\alpha^2+\alpha+1$	111(=7)
α^6	α^2+1	101(=5)

Table 2–3: Three representations for the elements of $GF(2^3)$ with the primitive polynomial $p(X) = X^3 + X + 1$.

rule. For example, $\alpha^3 - \alpha^5 = \alpha^3 + \alpha^5 = \alpha^2$. Inversion can be done using power representation of elements based on the fact that $\alpha^{2^m-1} = 1$. For example, for $GF(2^3), \alpha^{-4} = \frac{1}{\alpha^4} = \frac{\alpha^7}{\alpha^4} = \alpha^3$. Multiplication can be calculated easily by using power representation of elements. First, add up the exponential powers and then calculate the remainder with respect to $(2^m - 1)$, since $\alpha^{2^m-1} = 1$. For example, for $GF(2^3), \alpha^3 \cdot \alpha^5 = \alpha^{(3+5) \mod 7} = \alpha$. On the other hand, a division is simply computed by multiplying the divisor's inversion.

2.3 Narrow-Sense Reed-Solomon Codes

Compared to generalized RS codes, narrow-sense RS codes is a special class which most modern RS applications, as well as books and software such as MATLAB, are using. There are several differences between these two categories. For example, in narrow sense RS codes, the length of code-words is $n = 2^m - 1$ with $GF(2^m)$. While in generalized RS codes, for $GF(2^m)$, the length of code-words can be any values smaller than 2^m. Since the topic of this chapter is about the conventional RS coding algorithm, we here focus on narrow-sense RS codes. Generalized RS will be introduced in Chapter 3.

Let α be a primitive element of $GF(q)$ and let k be an integer with $0 \leq k \leq n = q - 1$. Then $c = [f(1), f(\alpha), f(\alpha^2), \cdots, f(\alpha^{q-2})]$ is a narrow-sense RS(n,k) code-word over $GF(q)$, where $f(x)$ is the message polynomial of order $(k-1)$ [19]. The summary of overall features are listed in Table 2–4.

Block Length of the Code-Words	$n = q - 1$
Number of Parity-Check Symbols in Code-word	$2t = n - k$
Error Correction Capability	$t = (n - k)/2$
Dimension	$k = q - 1 - 2t$
Minimum distance	$d_{min} = 2t + 1$

Table 2–4: Overall features of narrow-sense RS codes, based on [19].

2.3.1 Encoding Algorithm

Despite the definition described above, one mostly encodes and decodes data using a systematic algorithm. In this systematic algorithm, the message words are explicitly shown in the code-words. The structure for the encoding algorithm is shown in Figure 2–3. There are only two main steps. First, a generator polynomial $g(X)$ is calculated. Second, parity-bits are produced using the generator polynomial and the input message polynomial.

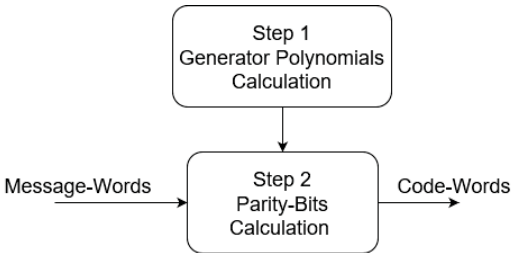


Figure 2–3: Structure of a conventional RS encoding algorithm.

The generator polynomial $g(X)$ of a t-error-correcting RS code is a polynomial with order $2t = (n - k)$ over $GF(q)$ and has $\alpha, \alpha^2, \cdots, \alpha^{2t}$ as all its roots. In other words, $g(X) = 0$ when $X = \alpha, \alpha^2, \cdots, \alpha^{2t}$. With $g_i \in GF(q)$

for $0 \leq i < 2t$, $g(X) = (X-\alpha)(X-\alpha^2)\cdots(X-\alpha^{2t}) = g_0 + g_1X + g_2X^2 + \cdots + g_{2t-1}X^{2t-1} + X^{2t}$.

When we transmit a message, the message data are taken as coefficients of the message polynomial over the same $GF(q)$. With $m_i \in GF(q)$ for $0 \leq i \leq k-1$, that is $m(X) = m_0 + m_1X + m_2X^2 + \cdots + m_{k-1}X^{k-1}$. Then the parity-check bits are calculated by $b(X) = (X^{n-k} \cdot m(X)) \mod g(X)$, which is actually the remainder of the shifted message polynomial divided by the generator polynomial. "*mod*" refers to the arithmetic function "modulo", which means to divide first and acquire the remainder. $g(X)$ is a polynomial, and $(X^{(n-k)}m(X))$ is the shifted message polynomial by $(n-k)$ bits. For RS(7,3), for example, if $m(X) = X^2 + \alpha^3X + \alpha^2$, and $X^{(n-k)} = X^4$, then the shifted message polynomial is $X^6+\alpha^3X^5+\alpha^2X^4$. In other words, for RS(7,3) if the message were [1 3 4] (see Table 2–3), then the shifted message would be [1 3 4 0 0 0 0]. The way to do modulo between two polynomials is demonstrated in Figure 2–2. The resulting coded polynomial is $c(X) = X^km(X) + b(X)$

Example 5. This example is given in [21]. For narrow-sense RS codes (7,3), $n = 7$; $q = 7+1 = 8$; $k = 3$; $2t = 7-3 = 4$; $t = (7-3)/2 = 2$; $m = log_28 = 3$. Therefore, we are using $GF(8) = GF(2^m) = GF(2^3) = \{0, 1(1 = \alpha^0 = \alpha^7), \alpha, \alpha^2, \alpha^3, \cdots, \alpha^6\}$. Then, the generator polynomial is calculated as $g(X) = (X-\alpha)(X-\alpha^2)(X-\alpha^3)(X-\alpha^4) = X^4+\alpha^3X^3+X^2+\alpha X+\alpha^3$. Now assume the message is [1 3 4]. Then the message polynomial is $m(X) = X^2 + \alpha^3X + \alpha^2$. The parity-check polynomial is calculated as $b(X) = (X^6 + \alpha^3X^5 + \alpha^2X^4) \mod (X^4 + \alpha^3X^3 + X^2 + \alpha X + \alpha^3) = \alpha^4X^3 + \alpha^4X^2 + X + \alpha^2$.

Finally, the complete code-word polynomial therefore is $c(X) = X^km(X) + b(X) = X^4m(X) + b(X) = X^6 + \alpha^3X^5 + \alpha^2X^4 + \alpha^4X^3 +$

$\alpha^4 X^2 + X + \alpha^2$. The code-words are $[\alpha^0, \alpha^3, \alpha^2, \alpha^4, \alpha^4, \alpha^0, \alpha^2]$ and equivalently [1 3 4 6 6 1 4] (see Table 2–3).

2.3.2 Decoding Algorithm

The decoding process is divided into the five following standard steps (Figure 2–4):

1. Calculate syndrome S_i for $i = 1, 2, 3, \cdots, 2t$.
2. Determine the error locator polynomial $\sigma(x)$.
3. Find the error locations, that is, the roots of the error locator polynomial (Chien search).
4. Compute the error magnitude.
5. Correct the errors.

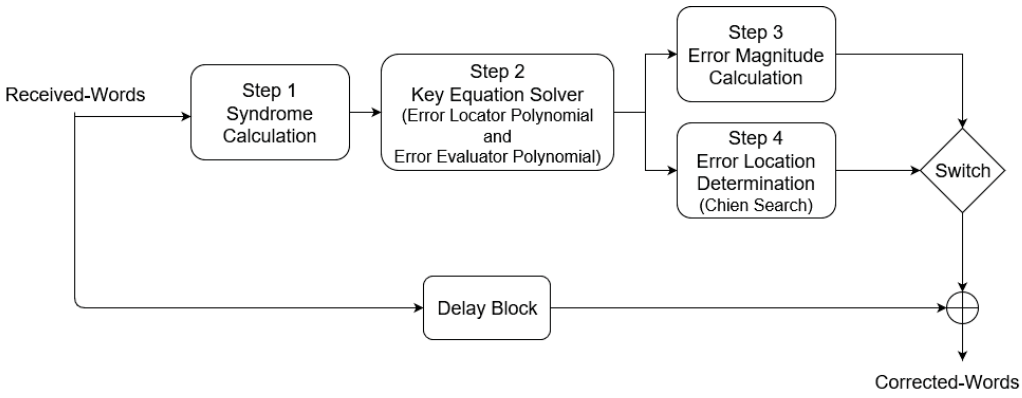


Figure 2–4: Structure of a conventional RS decoding algorithm.

Step 1. Syndrome is a parameter set particularly for the decoding process. Recall the encoding algorithm and the generated code-word has roots $\alpha, \alpha^2, \alpha^3, \cdots, \alpha^{2t}$, while each syndrome can be calculated easily by substituting these roots into the received code-word $r(x)$: $S_i = r(\alpha^i) = r_0 + r_1\alpha^i + r_2(\alpha^i)^2 + r_3(\alpha^i)^3 + \cdots + r_{(n-1)}(\alpha^i)^{(n-1)}$ where $i = 1, 2, \cdots, 2t$.

Let us define an error polynomial $e(x)$ which refers to the error generated in the transmission. Therefore, $r(x) = c(x) + e(x)$, where $c(x)$ is the code-word

polynomial that is the original code-word transmitted from the transmitter. As $c(x)$ has roots $\alpha, \alpha^2, \alpha^3, \cdots, \alpha^{2t}$ such that $c(\alpha^i) = 0$ for $\alpha^i = \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{2t}$, then the syndrome can be defined as $S_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$.

Example 6. Assume we are working on a narrow-sense RS(7,3). Suppose that the code-word polynomial transmitted was $c(x) = x^6 + \alpha^3 x^5 + \alpha^5 x^4 + \alpha^3 x^3 + \alpha^6 x^2 + \alpha^5 x^1 + 1$. The received polynomial was $r(x) = x^6 + \alpha^3 x^5 + x^4 + \alpha^3 x^3 + \alpha^2 x^2 + \alpha^5 x^1 + 1$ and it contains two errors. It should be noticed that r_i refers to the coefficients associated in this polynomial $r(x)$. In this particular example, $r_0 = 1, r_1 = \alpha^5, r_2 = \alpha^2, r_3 = \alpha^3, r_4 = 1, r_5 = \alpha^3, r_6 = 1$.

Then there are $2t = 4$ syndromes and the syndromes are calculated as following:

$S_1 = r(\alpha^1) = r_6 \cdot (\alpha^1)^6 + r_5 \cdot (\alpha^1)^5 + r_4 \cdot (\alpha^1)^4 + r_3 \cdot (\alpha^1)^3 + r_2 \cdot (\alpha^1)^2 + r_1 \cdot \alpha^1 + r_0 = \alpha^6 + \alpha^3 \cdot \alpha^5 + \alpha^4 + \alpha^3 \cdot \alpha^3 + \alpha^2 \cdot \alpha^2 + \alpha^5 \cdot \alpha^1 + 1 = \alpha^4$

$S_2 = r(\alpha^2) = r_6 \cdot (\alpha^2)^6 + r_5 \cdot (\alpha^2)^5 + r_4 \cdot (\alpha^2)^4 + r_3 \cdot (\alpha^2)^3 + r_2 \cdot (\alpha^2)^2 + r_1 \cdot \alpha^1 + r_0 = \alpha^1 2 + \alpha^3 \cdot \alpha^1 0 + \alpha^8 + \alpha^3 \cdot \alpha^6 + \alpha^2 \cdot \alpha^4 + \alpha^5 \cdot \alpha^2 + 1 = 1$

$S_3 = r(\alpha^3) = \alpha^{18} + \alpha^3 \cdot \alpha^{15} + \alpha^{12} + \alpha^3 \cdot \alpha^9 + \alpha^2 \cdot \alpha^6 + \alpha^5 \cdot \alpha^3 + 1 = 1$

$S_4 = r(\alpha^4) = \alpha^{24} + \alpha^3 \cdot \alpha^{20} + \alpha^{16} + \alpha^3 \cdot \alpha^{12} + \alpha^2 \cdot \alpha^8 + \alpha^5 \cdot \alpha^4 + 1 = \alpha^5$

Step 2. With $2t$ roots, we can get $2t$ syndromes and thus $2t$ error polynomials. Therefore, by solving the $2t$ equation system, it is possible to find the error polynomial and thus find the error location. However, it is not easy to solve this system, and we need to construct an error-locator polynomial $\sigma(x)$ via an algorithm referred as the Berlekamp-Massey (BM) algorithm [22].

$\sigma(x) = \sigma_0 + \sigma_1 x + \sigma_2 x^2 + \sigma_3 x^3 + \cdots + \sigma_v x^v$, where $v \leq t$ is the number of errors. It is constructed in a way that its roots are the reciprocals of α^i where i suggests an error location. For example, α^3 means that one error exists in

the coefficient of x^3 in the received polynomial. With this definition and based on Newtons identity [23], it has been proved that the syndromes are related with error locator polynomial $\sigma(x)$ in the way that $S_i = -\sum_{j=1}^{v} \sigma_j S_{i-j}$ where $i = v+1, v+2, \cdots, 2t$ and $v \leq t$ is the number of errors [21].

The above relationship can be implemented using a linear feedback shift register (LFSR) with the syndromes as outputs. Starting with an LFSR that produces S_1, the LFSR is checked to see if it can also produce S_2. If it can, then the LFSR is not changed. Otherwise, the LFSR is modified to produce S_2 as well. Then the LFSR is examined to see if it can also produce S_3. Again, if it can, the LFSR remains unchanged. Otherwise, the LFSR is updated so that it can also produce S_3. The procedure is carried out for $2t$ times [21]. In the end, the LFSR can produce all of the $2t$ syndrome components, and thus we can acquire the error location polynomial $\sigma(x)$.

Concretely, the BM algorithm is defined as follows. $B(x)$ is a supporting polynomial to assist in the updating of the error locator polynomial $\sigma(x)$. Denote L as the length of the LFSR, which represents the number of errors v, that is, the degree of $\sigma(x)$. The upper index i refers to the i-th iteration and the lower index j represents j-th coefficient that is associated with x^j in polynomials. For example, $\sigma_j^{(i)}$ means the j-th coefficient σ_j of $\sigma(x)$ updated at the i-th iteration.

- Initialization:

  $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, L^{(0)} = 0$, and $i = 1$

- Operation on the i-th iteration:

  1. Compute the LFSR output: $\tilde{S}_i = -\sum_{j=1}^{L^{(i-1)}} \sigma_j^{(i-1)} S_{i-j}$.
  2. Calculate the discrepancy: $\Delta_i = S_i - \tilde{S}_i$.
  3. Assign value to the variable $\delta : \delta = \begin{cases} 1, & \text{if } \Delta_i \neq 0 \text{ and } 2L^{(i-1)} \leq i-1 \\ 0, & \text{otherwise} \end{cases}$

4. Update:
$$\begin{bmatrix} \sigma^{(i)}(x) \\ B^{(i)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_i \cdot x \\ \Delta_i^{-1} \cdot \delta & (1-\delta) \cdot x \end{bmatrix} \cdot \begin{bmatrix} \sigma^{(i-1)}(x) \\ B^{(i-1)}(x) \end{bmatrix}$$
$$L^{(i)} = \delta \cdot (i - L^{(i-1)})) + (1-\delta) L^{(j-1)}$$
5. If $i = 2t$, stop. Otherwise, $i = i + 1$ and return to step 1.

Example 7. Continuing with Example 6, assume we are using narrow-sense RS(7,3). Suppose that the code polynomial $c(x) = x^6 + \alpha^3 x^5 + \alpha^5 x^4 + \alpha^3 x^3 + \alpha^6 x^2 + \alpha^5 x^1 + 1$ was transmitted. The received polynomial $r(x) = x^6 + \alpha^3 x^5 + x^4 + \alpha^3 x^3 + \alpha^2 x^2 + \alpha^5 x^1 + 1$ contains two errors. Then there are $2t = 4$ syndromes and the syndromes are calculated as $S_1 = \alpha^4, S_2 = 1, S_3 = 1, S_4 = \alpha^5$.

Now lets use the BM algorithm to find the error location polynomial for the received polynomial:

- Initialization:

  $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, L^{(0)} = 0$, and $i = 1$
- $i = 1$

  $\tilde{S_1} = 0 \Rightarrow \Delta_1 = S_1 - \tilde{S_1} = \alpha^4$

  $2L^{(0)} = 0 = i - 1 \Rightarrow \delta = 1$

  $$\begin{bmatrix} \sigma^{(1)}(x) \\ B^{(1)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha^4 \cdot x \\ \alpha^{-4} & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 - \alpha^4 x \\ \alpha^{-4} \end{bmatrix}$$

  $L^{(1)} = (i - L^{(0)}) = 1$
- $i = 2$

  $\tilde{S_2} = \sigma_1^{(1)} S_1 = \alpha \Rightarrow \Delta_2 = S_2 - \tilde{S_2} = \alpha^3$

$$2L^{(1)} = 2 > i - 1 \Rightarrow \delta = 0$$

$$\begin{bmatrix} \sigma^{(2)}(x) \\ B^{(2)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha^3 \cdot x \\ 0 & x \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^4 x \\ \alpha - 4 \end{bmatrix} = \begin{bmatrix} 1 - \alpha^3 x \\ \alpha^{-4} x \end{bmatrix}$$

$$L^{(2)} = L^{(1)} = 1$$

- $i = 3$

$$\tilde{S}_3 = \sigma_1^{(2)} S_2 = \alpha^3 \Rightarrow \Delta_3 = S_3 - \tilde{S}_3 = \alpha$$

$$2L^{(2)} = 2 > i - 1 \Rightarrow \delta = 1$$

$$\begin{bmatrix} \sigma^{(3)}(x) \\ B^{(3)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha \cdot x \\ \alpha^{-1} & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^3 x \\ \alpha - 4x \end{bmatrix} = \begin{bmatrix} 1 - \alpha^3 x - \alpha^{-3} x^2 \\ \alpha^{-1} - \alpha^2 x \end{bmatrix}$$

$$L^{(3)} = i - L^{(2)} = 2$$

- $i = 4$

$$\tilde{S}_4 = \sigma_1^{(3)} S_3 + \sigma_2^{(3)} S_2 = \alpha^6 \Rightarrow \Delta_4 = S_4 - \tilde{S}_4 = \alpha$$

$$2L^{(3)} = 4 > i - 1 \Rightarrow \delta = 0$$

$$\begin{bmatrix} \sigma^{(4)}(x) \\ B^{(4)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\alpha \cdot x \\ 0 & x \end{bmatrix} \cdot \begin{bmatrix} 1 - \alpha^3 x - \alpha^{-3} x^2 \\ \alpha - 1 - \alpha^2 x \end{bmatrix} = \begin{bmatrix} 1 - \alpha x - \alpha^6 x^2 \\ \alpha^{-1} x - \alpha^2 x^2 \end{bmatrix}$$

$$L^{(4)} = L^{(3)} = 2$$

Therefore, the desired error locator polynomial is $\sigma(x) = \sigma^{(4)}(x) = 1 - \alpha x - \alpha^6 x^2$.

Step 3. With the error locator polynomial $\sigma(x)$, we need to find its roots, whose reciprocals' exponential orders are the error locations. In this step, we must try all n elements $[\alpha^1, \alpha^2, \cdots, \alpha^n]$ of corresponding $GF(2^m)$ to see which elements are the roots. If the errors happened on the parity-bits

of code-words are not concerned, then we only need to try first k elements $[\alpha^1, \alpha^2, \cdots, \alpha^k]$. Substitute each element and if $\sigma(\alpha^i) = 0$ then α^i is a desired root and correspondingly $(n-i)$ is the error locations, that is, r_{n-i} that associated with x^{n-1} in the received-word polynomial. This process is referred as Chien search.

Example 8. Continuing with Example 7, the found error locator polynomial is $\sigma(x) = \sigma^{(4)}(x) = 1 - \alpha x - \alpha^6 x^2$ and it is using $GF(2^3)$. Therefore, we substitute $[\alpha, \alpha^2, \alpha^3, \cdots, \alpha^7]$ ($[\alpha, \alpha^2, \cdots, \alpha^5]$ if the errors on parity-bits are not the concern) into $\sigma(x)$ one by one. In the end of the evaluation process, it is found that

$$\sigma(\alpha^3) = 1 - \alpha \cdot \alpha^3 - \alpha^6 \cdot (\alpha^3)^2 = 0$$
$$\sigma(\alpha^5) = 1 - \alpha \cdot \alpha^5 - \alpha^6 \cdot (\alpha^5)^2 = 0$$

Therefore, α^3 and α^5 are the roots and their reciprocals, α^4 and α^2 (as $\alpha^7 = 1$ in $GF(2^3)$), are the desired error locations, which means that in the received code-word $r(x)$, the coefficients associated with x^4 and x^2 are wrong due to the transmission.

Step 4. To evaluate the error magnitudes, Forneys algorithm is applied. The derivation and proof are illustrated in [24] and in Chapter 7 of [20]. Here, we focus on the implementation of this algorithm.

First, an error evaluator polynomial $\Omega(x)$ is defined as $\Omega(x) = (S(x) \cdot \sigma(x)) \mod x^{2t}$, where $S(x)$ is the syndrome polynomial and $\sigma(x)$ is the error location polynomial. The purpose of "mod x^{2t}" is to eliminate all terms whose order is no less than 2t.

Second, the error magnitude at each error location α^i is calculated as following: $e_i = \frac{\Omega(x)}{\sigma'(x)}|_{x=\alpha^{-i}}$, where $\sigma'(x)$ is the formal derivative of $\sigma(x)$. The

value of $\sigma^{'}(x)$ can be calculated as following:

$\sigma^{'}(x) = \frac{d\sigma(x)}{dx} = \frac{d(\sigma_0+\sigma_1 x+\sigma_2 x^2+\sigma_3 x^3+\cdots+\sigma_v x^v)}{dx} = \sigma_1 + 2\sigma_2 x + 3\sigma_3 x^2 + 4\sigma_4 x^3 + \cdots + v\sigma_v x^{(v-1)}$

As $\sigma(x)$ is a polynomial over $GF(2^m)$, it follows modulo-2 arithmetic and thus:

$$i \cdot \sigma_i = \begin{cases} 0, & \text{if } i \text{ is even} \\ \sigma_i, & \text{if } i \text{ is odd} \end{cases}$$

Therefore, $\sigma'(x)$ can be formed by taking coefficients of the odd power terms of $\sigma(x)$ and assigning them to the next lower power terms. Specifically:

$$\sigma'(x) = \sigma_1 + \sigma_3 x^2 + \sigma_5 x^4 + \cdots$$

Example 9. Lets continue with Example 8. By now, we have found out the syndrome polynomial $S(x) = \alpha^4+x+x^2+\alpha^5 x^3$ and the error location polynomial $\sigma(x) = 1 - \alpha x - \alpha^6 x^2$. We also know the error locations are α^4 and α^2 (which actually refers to the coefficients associated with x^4 and x^2 in the received-word polynomial). Then the error evaluator polynomial is computed as following:

$$\begin{aligned} \Omega(x) &= (S(x) \cdot \sigma(x)) \mod x^{2t} \\ &= (\alpha^4 + x + x^2 + \alpha^5 x^3) \cdot (1 - \alpha x - \alpha^6 x^2) \mod x^4 \\ &= \alpha^4 + \alpha^4 x \end{aligned}$$

And the derivative of error location polynomial is: $\sigma^{'}(x) = -\alpha$. Then the error values are calculated as following:

$$e_2 = \frac{\Omega(x)}{\sigma^{'}(x)}|_{x=\alpha^{-2}} = \frac{\alpha 4 + \alpha^4 \cdot \alpha^{-2}}{-\alpha} = -1 = 1$$

$$e_4 = \frac{\Omega(x)}{\sigma'(x)}|_{x=\alpha^{-4}} = \frac{\alpha 4 + \alpha^4 \cdot \alpha^{-4}}{-\alpha} = -\alpha^4 = \alpha^4$$

Therefore, the error polynomial is $e(x) = x^2 + \alpha^4 x^4$, which satisfies $c(x) = r(x) + e(x)$.

Step 5. After we find out the error polynomial $e(x)$, we can just add it to the received-word polynomial $r(x)$ and then we can have the original code-word polynomial $c(x)$.

Example 10. Lets continue with Example 9. Since we have $e(x) = x^2 + \alpha^4 x^4$ and $r(x) = x^6 + \alpha^3 x^5 + x^4 + \alpha^3 x^3 + \alpha^2 x^2 + \alpha^5 x^1 + 1$, it is easy to add them up and get the following result:

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

So the original code-words we got is $[\alpha^0, \alpha^3, \alpha^5, \alpha^3, \alpha^6, \alpha^5, \alpha^0]$, that is, [1 3 7 3 5 7 1]. By checking with the information provided in Example 6, the result is correct.

CHAPTER 3
Low-Latency Reed-Solomon Codes

3.1 Concept and Proof

The content of this section is mainly based on [17] in which Dr. Shokrollahi introduced a special class of generalized RS codes which allow for faster encoding and decoding. The underlying idea is primarily a clever choice of the root set of the codes so that this set is closed under multiplication with a p-th root of unity over the base GF. The biggest advantage is that this class of generalized RS codes can be constructed as p components with length n/p in a similar manner of conventional RS codes so that the entire encoding process and most parts of the decoding process can all run in parallel on these p constituent codes. In other words, these processes can speed up by a factor of almost p, with only a slight increase in hardware costs.

Before stating the newly proposed algorithm, the definition of GRS codes should be examined first. For $1 \leq n \leq q$,

$$GRS_k(\boldsymbol{\alpha}, \boldsymbol{v}) = \{(v_0 f(\alpha_0), v_1 f(\alpha_1), \cdots, v_{n-1} f(\alpha_{n-1})) | f \in GF_q[x]_{<k}\}$$

where $\boldsymbol{\alpha}$ is a set of distinct elements of $GF(q)$, $\boldsymbol{v}$ is a set of nonzero elements of $GF(q)$. $f \in GF_q[x]_{<k}$ is an univariate polynomial over $GF(q)$ with degree less than k. Just like narrow-sense RS codes, GRS codes are also MDS and the minimum distance is $(n-k+1)$ as well [25].

Recall the definition of narrow-sense RS codes in Chapter 2. It is easy to see that narrow-sense RS codes is a special class of GRS codes with $n = q-1$, $\alpha_i = \alpha^i$, and $v_i = 1$ for $0 \leq i \leq n-1$.

The definition of low-latency RS codes developed by Dr. Shokrollahi is given as following:

$C(n, k; \alpha, \rho)$ is a code over $GF(q)$ with block-length $n = p \cdot m$, dimension k, and minimum distance $(n - k + 1)$. Its codewords are of the form $(v_0 f(\alpha_0), v_1 f(\alpha_1), \cdots, v_{n-1} f(\alpha_{n-1}))$, where $f \in GF_q[x]_{<k}$, $\alpha_i = \rho^{(i \mod p)} \alpha^{\lfloor i/p \rfloor}$, α is the primitive element of $GF(q)$, ρ is p-th root of unity of the same field, and $v_i = \frac{1}{\prod_{j \neq i} (\alpha_i - \alpha_j)}$.

Example 11. Assume that we are using $GF(2^4)$, and working on $RS(15, 9; \alpha, \rho)$ with $p = 3$. Then

$$\rho^p = \rho^3 = \alpha^{15} = 1 \Rightarrow \rho = \alpha^5$$

According to $\alpha_i = \rho^{(i \mod p)} \alpha^{\lfloor i/p \rfloor}$, we can get the relationship in Table 3–1. As we can see, although the selected roots are not consecutive, the set essentially contains all the roots of a conventional narrow-sense $RS(15, 9)$ codes and certainly it is closed under multiplication.

$\alpha_0 = \rho^0 \alpha^0 = \alpha^0$	$\alpha_8 = \rho^2 \alpha^2 = \alpha^{12}$
$\alpha_1 = \rho^1 \alpha^0 = \alpha^5$	$\alpha_9 = \rho^0 \alpha^3 = \alpha^3$
$\alpha_2 = \rho^2 \alpha^0 = \alpha^{10}$	$\alpha_{10} = \rho^1 \alpha^3 = \alpha^8$
$\alpha_3 = \rho^0 \alpha^1 = \alpha^1$	$\alpha_{11} = \rho^2 \alpha^3 = \alpha^{13}$
$\alpha_4 = \rho^1 \alpha^1 = \alpha^6$	$\alpha_{12} = \rho^0 \alpha^4 = \alpha^4$
$\alpha_5 = \rho^2 \alpha^1 = \alpha^{11}$	$\alpha_{13} = \rho^1 \alpha^4 = \alpha^9$
$\alpha_6 = \rho^0 \alpha^2 = \alpha^2$	$\alpha_{14} = \rho^2 \alpha^4 = \alpha^{14}$
$\alpha_7 = \rho^1 \alpha^2 = \alpha^7$	

Table 3–1: Conversion between roots of $C(15, 9; \alpha, \rho)$ and elements of $GF(2^4)$.

In order to utilize the parallel running property, the code-words need to be capable to be partitioned into p parts. A set of (coefficients of) polynomial vectors $[H_0, H_1, \cdots, H_{p-1}]$ is defined, where $H_j \in GF_q[x]_{<m}$ are univariate polynomials over $GF(q)$ of degree less than m, such that a set of Fourier transform equations: for $i = 0, 1, \cdots, p-1$, $FT(H)_i \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} H_j = 0 \mod g_i$ are true. g_i is a set of generator polynomials and is defined as $g_i \equiv \Pi_{j \equiv i \mod p}^{0 \le j \le n-k} (x - \alpha^j)$.

Now the task is to prove that $[H_0, H_1, \cdots, H_{p-1}]$ is a codeword set belonging to $C(n, k; \alpha, \rho)$. First, we define a root matrix of order $(k-1)$ as $V_k(\alpha_0, \alpha_2, \cdots, \alpha^{n-1})$ and a diagonal matrix as v:

$$V_k(\alpha_0, \alpha_2, \cdots, \alpha^{n-1}) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{k-1} & \alpha_1^{k-1} & \cdots & \alpha_{n-1}^{k-1} \end{bmatrix}$$

$$v = \mathrm{diag}(v_0, v_1, \cdots, v_{n-1}) = \begin{bmatrix} v_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & v_1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & v_{n-2} & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_{n-1} \end{bmatrix}$$

With simple calculation, we can get:

$$G = V_k(\alpha_0, \alpha_2, \cdots, \alpha^{n-1}) \cdot v = \begin{bmatrix} v_0 & v_1 & \cdots & v_{n-1} \\ v_0\alpha_0 & v_1\alpha_1 & \cdots & v_{n-1}\alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_0\alpha_0^{k-1} & v_1\alpha_1^{k-1} & \cdots & v_{n-1}\alpha_{n-1}^{k-1} \end{bmatrix}$$

It is obvious that the codewords $(v_0 f(\alpha_0), v_0 f(\alpha_0), \cdots, v_{n-1} f(\alpha_{n-1}))$ of $C(n, k; \alpha, \rho)$ are the row-span of G. In other words, G is the generator matrix of $C(n, k; \alpha, \rho)$.

On the other hand, as the generator polynomial set for $[H_0, H_1, \cdots, H_{p-1}]$ is defined as follows: for $i = (0, 1, \cdots, p-1)$, $g_i \equiv \Pi_{j \equiv i \bmod p}^{0 \le j \le n-k}(x - \alpha^j)$, so a set of root matrix is defined: for $i = (0, 1, \cdots, p-1)$, $V_i \equiv V_m(\alpha^i, \alpha^{i+p}, \cdots, \alpha^{i+p\cdot(r_i-1)})$ where r_i is the number of integers between 0 and $(n-k-1)$ which are congruent to i modulo p. As $FT(H)_i \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} H_j = 0 \mod g_i$, so $FT(H)_i \cdot V_i = \Sigma_{j=0}^{p-1} \rho^{i \cdot j} H_j \cdot V_i = 0$, which leads to $([H_0, H_1, \cdots, H_{p-1}] \cdot P) = 0$ and P is a matrix defined as for $i, j = (0, 1, \cdots, p-1)$, $P_{i,j} = \rho^{i \cdot j} V_j$. In other words, P is the transpose of the parity-check matrix of codewords $[H_0, H_1, \cdots, H_{p-1}]$.

When we substitute $V_i \equiv V_m(\alpha^i, \alpha^{i+p}, \cdots, \alpha^{i+p\cdot(r_i-1)})$ into P, with simple calculations and permutations of columns, the matrix P has the form

$$P = \begin{bmatrix} V_{n-k}(1, \alpha, \cdots, \alpha^{m-1})^T \\ V_{n-k}(\rho, \rho\alpha, \cdots, \rho\alpha^{m-1})^T \\ V_{n-k}(\rho^2, \rho^2\alpha, \cdots, \rho^2\alpha^{m-1})^T \\ \vdots \\ V_{n-k}(\rho^{p-1}, \rho^{p-1}\alpha, \cdots, \rho^{p-1}\alpha^{m-1})^T \end{bmatrix}$$

which is $V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T$. Therefore it suggests that $([H_0, H_1, \cdots, H_{p-1}] \cdot V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T) = 0$ and $V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$ is the parity-check matrix of codewords $[H_0, H_1, \cdots, H_{p-1}]$.

At this point, all we need to do is to prove $V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$ is also the parity-check matrix of $C(n, k; \alpha, \rho)$, that is, $(V_k(\alpha_0, \alpha_2, \cdots, \alpha^{n-1}) \cdot v \cdot V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T) = G \cdot V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T = 0$.

$$G \cdot V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T =$$

$$\begin{bmatrix} v_0 & v_1 & \cdots & v_{n-1} \\ v_0\alpha_0 & v_1\alpha_1 & \cdots & v_{n-1}\alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_0\alpha_0^{k-1} & v_1\alpha_1^{k-1} & \cdots & v_{n-1}\alpha_{n-1}^{k-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{n-k-1} & \alpha_1^{n-k-1} & \cdots & \alpha_{n-1}^{n-k-1} \end{bmatrix}^T =$$

$$\begin{bmatrix} v_0 + \cdots + v_{n-1} & \cdots & v_0\alpha_0^{n-k-1} + \cdots + v_{n-1}\alpha_{n-1}^{n-k-1} \\ v_0\alpha_0 + \cdots + v_{n-1}\alpha_{n-1} & \cdots & v_0\alpha_0^{n-k} + \cdots + v_{n-1}\alpha_{n-1}^{n-k} \\ \vdots & \ddots & \vdots \\ v_0\alpha_0^{k-1} + \cdots + v_{n-1}\alpha_{n-1}^{k-1} & \cdots & v_0\alpha_0^{n-2} + \cdots + v_{n-1}\alpha_{n-1}^{n-2} \end{bmatrix}$$

As we can see, if the above matrix is 0, then every entry needs to be 0, which means $v_0\alpha_0^i + \cdots + v_{n-1}\alpha_{n-1}^i = 0$ for all $i = 0, 1, \cdots, n-2$. This condition is equivalent to proving the following:

$$V_{n-1}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) \cdot \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{n-2} & \alpha_1^{n-2} & \cdots & \alpha_{n-1}^{n-2} \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = 0$$

To prove this equality, the inverse of $V_n(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$ needs to be reviewed first. Let's define $\lambda_i(x) = v_i \prod_{j \neq i}(x - \alpha_j)$. Recall that $v_i = \frac{1}{\prod_{j \neq i}(\alpha_i - \alpha_j)}$ in the definition of $C(n, k; \alpha, \rho)$. Therefore, we have $\lambda_i(\alpha_i) = 1$ and $\lambda_i(\alpha_j) = 0$ for $j \neq i$. Let $\lambda_{i,j}$ denote the coefficient of x^j of $\lambda_i(x)$, so that $\lambda_{i,n-1} = v_i$.

Then, the inverse of $V_n(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$ is

$$\left[\begin{array}{cccc|c} \lambda_{0,0} & \lambda_{0,1} & \cdots & \lambda_{0,n-2} & \lambda_{0,n-1} \\ \lambda_{1,0} & \lambda_{1,1} & \cdots & \lambda_{1,n-2} & \lambda_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \lambda_{n-1,0} & \lambda_{n-1,1} & \cdots & \lambda_{n-1,n-2} & \lambda_{n-1,n-1} \end{array}\right] = (A|\Gamma)$$

where $\Gamma = (\lambda_{0,n-1}, \lambda_{1,n-1}, \cdots, \lambda_{n-1,n-1})^T = (v_0, v_1, \cdots, v_{n-1})^T$. Therefore,

$$\begin{aligned}(A|\Gamma) \cdot V_n(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) = I_n &= V_n(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) \cdot (A|\Gamma) \\ &= (\frac{V_{n-1}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})}{B}) \cdot (A|\Gamma)\end{aligned}$$

which suggests that

$$V_{n-1}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) \cdot A = I_{n-1}$$

$$V_{n-1}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) \cdot \Gamma = V_{n-1}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1}) \cdot (v_0, v_1, \cdots, v_{n-1})^T = 0$$

Thus, it proves the following:

$$V_k(\alpha_0, \alpha_2, \cdots, \alpha^{n-1}) \cdot v \cdot V_{n-k}(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})^T = 0$$

Therefore, it is proved that P is also the transpose of the parity-check matrix of $C(n, k; \alpha, \rho)$ and thus $[H_0, H_1, \cdots, H_{p-1}]$ is a codeword set belonging to $C(n, k; \alpha, \rho)$, which is a class of generalized RS codes.

3.2 Algorithm Description

This section is mainly based on [17]. For each step, theorems and concepts are first introduced and then an example is given for better understanding. The definitions of parameters and variables already given in the preceding chapters may not be presented in this section again. It is highly recommended to read at least Chapter 2 before moving to this chapter.

Recall in the definition of $C(n, k; \alpha, \rho)$, speed-up coefficient p is defined in $n = p \cdot m$, which implies that code-word length n is divisible by p. However, no constraints are added to message-word length k and parity-bit length r. In fact, the coding algorithm would be slightly different dependent on if k and r are divisible by p. For this section, assume that n, k, and r are all divisible by p. Circumstances of arbitrary values of k and r will be introduced in the next section. Specifically, we define $n = p \cdot m$, $k = p \cdot t$, $r = (n - k) = p \cdot l$, α is the primitive elements of $GF(q)$, ρ is p-th roots of unity, and thus $\rho = \alpha^m$, and $m = t + l$.

3.2.1 Encoding Algorithm

Design Structure. The overall structure for low-latency RS encoding algorithm is shown in Figure 3–1. There are five steps. A block of message-words is split into p components and then go through a discrete Fourier transformer (DFT). With p generator polynomials pre-calculated, p blocks of parity bits are produced based on the output of the DFT block. Finally, the message-words along with the parity-bits are grouped together following a particular rule to output the final code-words. In the following paragraphs, each step is explained in greater details.

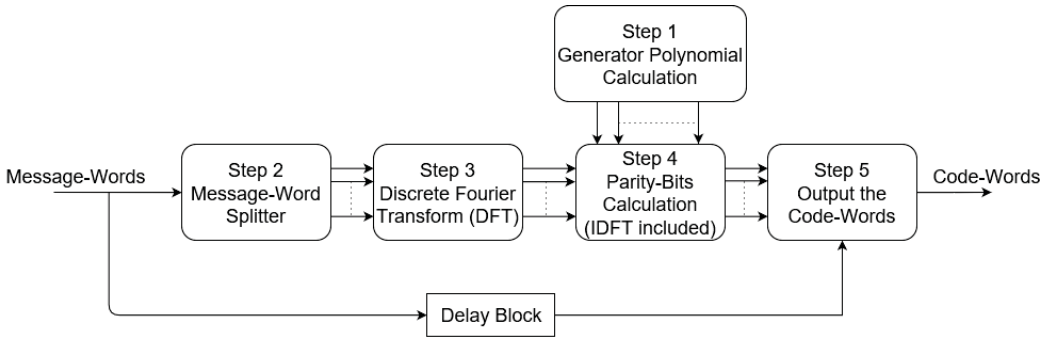


Figure 3–1: Structure of low-latency RS encoding algorithm.

Step 1. Produce the set of generator polynomials as follows: for $i = (0, 1, \cdots, p-1)$, $g_i \equiv \Pi_{j \equiv i \mod p}^{0 \le j \le n-k}(x - \alpha^j)$.

Example 12. Assume we are using $GF(2^4) = GF(16)$ (Table 2–2), and working on $RS(15, 9; \alpha, \rho)$ with $p = 3$. So $r = n - k = 15 - 9 = 6$ and thus the generator polynomials are:

$$\begin{aligned}
g_0 \equiv \Pi^{0\le j\le 6}_{j\equiv 0 \bmod 3}(x-\alpha^j) = (x-\alpha^0)(x-\alpha^3) &= x^2 + \alpha^{14}x + \alpha^3 \\
&= [\alpha^0, \alpha^{14}, \alpha^3] = [1, 9, 8] \\
g_1 \equiv \Pi^{0\le j\le 6}_{j\equiv 1 \bmod 3}(x-\alpha^j) = (x-\alpha^1)(x-\alpha^4) &= x^2 + x + \alpha^5 \\
&= [\alpha^0, \alpha^0, \alpha^5] = [1, 1, 6] \\
g_2 \equiv \Pi^{0\le j\le 6}_{j\equiv 2 \bmod 3}(x-\alpha^j) = (x-\alpha^2)(x-\alpha^5) &= x^2 + \alpha^1 x + \alpha^7 \\
&= [\alpha^0, \alpha^1, \alpha^7] = [1, 2, 11]
\end{aligned}$$

Step 2. Arrange the block of message-words M into a matrix and label each row vector as M_i, where $i = 0, 1, \cdots, p-1$. The matrix is constructed as following: for $i = 0, 1, \cdots, p-1$, $j = 0, 1, \cdots, t-1$, $M_{matrix}(i, j) = M((j+1)\cdot p - i)$.

Example 13. Continuing with Example 12, assume the block of message-words M is $[11, 3, 2, 5, 6, 4, 10, 12, 8]$, which in power representation is equal to $[\alpha^7, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^6, \alpha^3]$, then the matrix is as following:

$$M_{matrix} = \begin{bmatrix} 2 & 4 & 8 \\ 3 & 6 & 12 \\ 11 & 5 & 10 \end{bmatrix} \text{ and thus } \begin{cases} M_0 = [2, 4, 8] = [\alpha^1, \alpha^2, \alpha^3] \\ M_1 = [3, 6, 12] = [\alpha^4, \alpha^5, \alpha^6] \\ M_2 = [11, 5, 10] = [\alpha^7, \alpha^8, \alpha^9] \end{cases}$$

Step 3. Define the Fourier transform of M_i as done with H_i before. Specifically, for $i = 0, 1, \cdots, p-1$, $FT(M)_i = \Sigma_{j=0}^{p-1} \rho^{i \cdot j} M_j$.

Example 14. Continuing with Example 13, as ρ is 3-th ($p = 3$) root of unity, so $\rho = \alpha^5$ and thus

$$\begin{aligned}
FT(M)_0 &= M_0 + M_1 + M_2 \\
&= [\alpha^1 + \alpha^4 + \alpha^7, \alpha^2 + \alpha^5 + \alpha^8, \alpha^3 + \alpha^6 + \alpha^9] \\
&= [\alpha^9, \alpha^{10}, \alpha^{11}] \\
FT(M)_1 &= M_0 + \rho M_1 + \rho^2 M_2 \\
&= [\alpha^1 + \rho\alpha^4 + \rho^2\alpha^7, \alpha^2 + \rho\alpha^5 + \rho^2\alpha^8, \alpha^3 + \rho\alpha^6 + \rho^2\alpha^9] \\
&= [\alpha^6, \alpha^7, \alpha^8] \\
FT(M)_2 &= M_0 + \rho^2 M_1 + \rho M_2 \\
&= [\alpha^1 + \rho^2\alpha^4 + \rho\alpha^7, \alpha^2 + \rho^2\alpha^5 + \rho\alpha^8, \alpha^3 + \rho^2\alpha^6 + \rho\alpha^9] \\
&= [\alpha^2, \alpha^3, \alpha^4]
\end{aligned}$$

Step 4. We define the parity bits $[h_0, h_1, \cdots, h_{(p-1)}]$ so that $[x^l M_0 + h_0, x^l M_1 + h_1, \cdots, x^l M_{(p-1)} + h_{(p-1)}] = [H_0, H_1, \cdots, H_{(p-1)}] \in C(n, k; \alpha, \rho)$.

Let $f_i := x^l FT(M)_i \bmod g_i$, where $i = 0, 1, \cdots, p-1$, and the purpose of x^l is to shift $FT(M)_i$ by l degrees. Recall that vectors essentially represent polynomials and each vector element is the coefficient associated with each order of x. From left to right, the order is monotonically decreasing. This step is actually the same process as conventional narrow-sense RS encoding algorithms except that the message polynomial $m(x)$ is replaced by $FT(M)_i$.

Recall that $[x^l M_0 + h_0, x^l M_1 + h_1, \cdots, x^l M_{(p-1)} + h_{(p-1)}] = [H_0, H_1, \cdots, H_{(p-1)}]$ and $FT(H)_i \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} H_j = 0 \mod g_i$. Therefore, for $i = 0, 1, \cdots, p-$

1, $f_i = -\Sigma_{j=0}^{p-1}\rho^{i\cdot j}h_j \mod g_i$. With this expression, we can calculate h_j from f_i using inverse Fourier transform. Concretely, for $i = (0, 1, \cdots, p-1)$, $h_i = -\frac{1}{p}(\Sigma_{j=0}^{p-1}\rho^{-i\cdot j}f_j)$.

Example 15. Continuing with Example 14, so

$$f_0 := x^l FT(M)_0 \mod g_0 = [\alpha^9, \alpha^{10}, \alpha^{11}, 0, 0] \mod [\alpha^0, \alpha^{14}, \alpha^3] = [\alpha^4, 0]$$

$$f_1 := x^l FT(M)_1 \mod g_1 = [\alpha^6, \alpha^7, \alpha^8, 0, 0] \mod [\alpha^0, \alpha^0, \alpha^5] = [\alpha^{13}, \alpha^{11}]$$

$$f_2 := x^l FT(M)_2 \mod g_2 = [\alpha^2, \alpha^3, \alpha^4, 0, 0] \mod [\alpha^0, \alpha^1, \alpha^7] = [\alpha^0, \alpha^6]$$

Notice that the method of solving "**mod**" is shown in Figure 2–2.

$$\begin{aligned}
h_0 = -\frac{1}{3}(\Sigma_{j=0}^{2}\rho^{-0\cdot j}f_j) &= -(1/3)(f_0 + f_1 + f_2) \\
&= [\alpha^4 + \alpha^{13} + \alpha^0, 0 + \alpha^{11} + \alpha^6] \\
&= [\alpha^{12}, \alpha^1] = [15, 2] \\
h_1 = -\frac{1}{3}(\Sigma_{j=0}^{2}\rho^{-1\cdot j}f_j) &= -(1/3)(f_0 + \rho^2 f_1 + \rho f_2) \\
&= [\alpha^4 + \alpha^{10}\cdot\alpha^{13} + \alpha^5\cdot\alpha^0, 0 + \alpha^{10}\cdot\alpha^{11} + \alpha^5\cdot\alpha^6] \\
&= [0, \alpha^1] = [0, 2] \\
h_2 = -\frac{1}{3}(\Sigma_{j=0}^{2}\rho^{-2\cdot j}f_j) &= -(1/3)(f_0 + \rho f_1 + \rho^2 f_2) \\
&= [\alpha^4 + \alpha^5\cdot\alpha^{13} + \alpha^{10}\cdot\alpha^0, 0 + \alpha^5\cdot\alpha^{11} + \alpha^{10}\cdot\alpha^6] \\
&= [\alpha^6, 0] = [12, 0]
\end{aligned}$$

Notice that $-\frac{1}{3} = 1$ in the context of a field of characteristic 2.

Step 5. Finally we put everything together to get the final code-words $c = [H_0, H_1, \cdots, H_{(p-1)}] = [x^l M_0 + h_0, x^l M_1 + h_1, \cdots, x^l M_{(p-1)} + h_{(p-1)}]$. The ordering follows a two-level rule.the first-level rule is that the highest order has the smallest index. As the message-words M have higher order than all

h_i, so M is in front of all h_i. The same way is followed by elements inside each group. The second-level rule is that among all h_i, the ones with bigger i have smaller index. Specifically, for the speed-up coefficient p and $l = (n-k)/p$, the code-words are

$$c = [M(0), M(1), \cdots, M(k-1), h_{(p-1)}(0), h_{(p-2)}(0), \cdots, h_0(0), h_{(p-1)}(1),$$
$$h_{(p-2)}(1), \cdots, h_0(1), \cdots, h_{(p-1)}(l-1), h_{(p-2)}(l-1), \cdots, h_0(l-1)]$$

Example 16. Continuing with Example 15, the message-words $M = [\alpha^7, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^6, \alpha^3]$, and the set of parity-bits are $h_0 = [\alpha^{12}, \alpha^1]$, $h_1 = [0, \alpha^1]$, $h_2 = [\alpha^6, 0]$, then the final code-words c is $[\alpha^7, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^6, \alpha^3, \alpha^6, 0, \alpha^{12}, 0, \alpha^1, \alpha^1]$, that is, in 4-tuple representation $[11, 3, 2, 5, 6, 4, 10, 12, 8, 12, 0, 15, 0, 2, 2]$.

3.2.2 Decoding Algorithm

As Figure 3–2 shows, there are seven steps in the low-latency RS decoding algorithm. Received-words are first partitioned into p components followed by a process in discrete Fourier transformer (DFT). Then the syndromes are computed and sent to the key equation solver. The key equation solver is implemented by a conventional Berlekamp-Massey algorithm, which produces an error-locator polynomial and an error-evaluator polynomial. With these results, locations and error magnitudes are calculated in Step 5 and Step 6, respectively. Finally, in Step 7 errors are eliminated, and the correction-words are output. In the following paragraphs, each step is illustrated in greater details.

Step 1. For $i = 0, 1, \cdots, p-1$, define $L_i(x) = \Sigma_{(j=0)}^{(m-1)} r_{(p \cdot j+i)} x^j$. This step is to divide the block of received-words into p components, which is similar to what we did in the encoding algorithm.

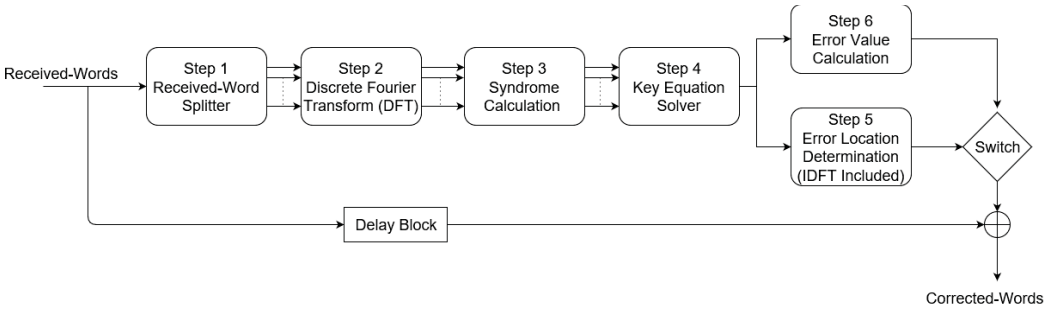


Figure 3–2: Structure of low-latency RS decoding algorithm.

Example 17. Continue with Example 16. The original code-words is

$$c = [\alpha^7, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^6, \alpha^3, \alpha^6, 0, \alpha^{12}, 0, \alpha^1, \alpha^1]$$

that is, in 4-tuple representation

$$c = [11, 3, 2, 5, 6, 4, 10, 12, 8, 12, 0, 15, 0, 2, 2]$$

Assume the received-word is

$$r = [6, 3, 2, 5, 6, 4, 10, 11, 8, 12, 0, 15, 6, 2, 2]$$

that is, in power representation

$$r = [\alpha^5, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^7, \alpha^3, \alpha^6, 0, \alpha^{12}, \alpha^5, \alpha^1, \alpha^1]$$

Comparing with the original code-words produced in the encoder, the errors are at $r(0), r(7)$ and $r(12)$, that is, in term of polynomial coefficients, r_{14}, r_7 and r_2. Then

$$\begin{aligned} L_0(x) = \Sigma_{(j=0)}^{4} r_{(3*j+0)} x^j &= r_0 + r_3 x + r_6 x^2 + r_9 x^3 + r_{12} x^4 \\ &= [r_{12}, r_9, r_6, r_3, r_0] \\ &= [\alpha^1, \alpha^2, \alpha^3, \alpha^{12}, \alpha^1] = [2, 4, 8, 15, 2] \\ L_1(x) = \Sigma_{(j=0)}^{4} r_{(3*j+1)} x^j &= r_1 + r_4 x + r_7 x^2 + r_{10} x^3 + r_{13} x^4 \end{aligned}$$

$$
\begin{aligned}
&= [r_{13}, r_{10}, r_7, r_4, r_1] \\
&= [\alpha^4, \alpha^5, \alpha^7, 0, \alpha^1] = [3, 6, 11, 0, 2] \\
L_2(x) = \Sigma_{(j=0)}^{4} r_{(3*j+2)} x^j &= r_2 + r_5 x + r_8 x^2 + r_{11} x^3 + r_{14} x^4 \\
&= [r_{14}, r_{11}, r_8, r_5, r_2] \\
&= [\alpha^5, \alpha^8, \alpha^9, \alpha^6, \alpha^5] = [6, 5, 10, 12, 6]
\end{aligned}
$$

Step 2. In this step, apply Fourier transform on L_j as the same way before. Concretely, $FT(L)_i = \Sigma_{j=0}^{p-1} \rho^{i \cdot j} L_j$.

Example 18. Continue with Example 17, then

$F_0(x) = FT(L)_0 = \Sigma_{j=0}^{p-1} \rho^{0 \cdot j} L_j = L_0 + L_1 + L_2 = [\alpha^{10}, \alpha^{10}, \alpha^{14}, \alpha^4, \alpha^5]$

$F_1(x) = FT(L)_1 = \Sigma_{j=0}^{p-1} \rho^{1 \cdot j} L_j = L_0 + \rho L_1 + \rho^2 L_2 = [\alpha^{14}, \alpha^7, \alpha^2, \alpha^{13}, \alpha^{12}]$

$F_2(x) = FT(L)_2 = \Sigma_{j=0}^{p-1} \rho^{2 \cdot j} L_j = L_0 + \rho^2 L_1 + \rho L_2 = [\alpha^6, \alpha^3, \alpha^8, \alpha^0, \alpha^7]$

Step 3. In this step, calculate the syndromes by substituting roots of generator polynomials. Recall that for conventional RS codes, syndromes are calculated by substituting the roots of generator polynomial $g(x)$ into the received-word polynomial $r(x)$ because the parity-bits polynomial $b(x)$ is acquired by $b(x) = msg(x) \cdot x^{n-k} \bmod g(x)$ where $msg(x)$ is the message-word polynomial and thus the code-word polynomial $c(x) = (msg(x) \cdot x^{n-k} + b(x))$ has the same roots of $g(x)$. This conclusion leads to the definition of syndromes that is $S_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$ where $e(x)$ is the error polynomial. For low-latency RS codes, on the other hand, there are p generator polynomials $g_i(x)$ each with $\frac{n-k}{p}$ roots. In Step 4 of Section 3.2.1, there are equations that

$f_i := x^l FT(M)_i \bmod g_i$, where $FT(M)_i$ is obtained by Fourier transform on message-word polynomials M_i and f_i are sent to IDFT to generate parity bits h_i. Therefore, polynomials $u_i(x) = x^l FT(M)_i(x) + f_i(x)$ have the same roots of $g_i(x)$. In other words, to calculate the syndromes in low-latency RS decoding algorithm, we should substitute the roots of $g_i(x)$ into polynomial $u_i(x)$. To generate $u_i(x)$ we should acquire $FT(M)_i$ and f_i by applying Fourier transform on both M_i and h_i. As the code-words c generated from low-latency RS encoding algorithm consists of M_i and h_i, therefore, we just need to simply apply Fourier transform on the split received-words L_i which was illustrated in the last step, and then substitute the roots of g_i to the corresponding output from Fourier transform. Specifically, for $i = 0, 1, \cdots, n-k-1$, the syndromes are defined as follows:

$$s_i = \begin{cases} F_0(\alpha^i) & \text{if } i = 0 \bmod p \\ F_1(\alpha^i) & \text{if } i = 1 \bmod p \\ \vdots & \vdots \\ F_{p-1}(\alpha^i) & \text{if } i = p-1 \bmod p \end{cases}$$

Example 19. Continuing with Example 18, we have $n - k - 1 = 15 - 9 - 1 = 5$. The syndromes are

$$s_0 = F_0(\alpha^0) = \alpha^{10} \cdot (\alpha^0)^4 + \alpha^{10} \cdot (\alpha^0)^3 + \alpha^{14} \cdot (\alpha^0)^2 + \alpha^4 \cdot (\alpha^0)^1 + \alpha^5 = \alpha^6$$

$$s_1 = F_1(\alpha^1) = \alpha^{14} \cdot (\alpha^1)^4 + \alpha^7 \cdot (\alpha^1)^3 + \alpha^2 \cdot (\alpha^1)^2 + \alpha^{13} \cdot (\alpha^1)^1 + \alpha^{12} = \alpha^9$$

$$s_2 = F_2(\alpha^2) = \alpha^6 \cdot (\alpha^2)^4 + \alpha^3 \cdot (\alpha^2)^3 + \alpha^8 \cdot (\alpha^2)^2 + \alpha^0 \cdot (\alpha^2)^1 + \alpha^7 = \alpha^4$$

$$s_3 = F_0(\alpha^3) = \alpha^{10} \cdot (\alpha^3)^4 + \alpha^{10} \cdot (\alpha^3)^3 + \alpha^{14} \cdot (\alpha^3)^2 + \alpha^4 \cdot (\alpha^3)^1 + \alpha^5 = \alpha^4$$

$$s_4 = F_1(\alpha^4) = \alpha^{14} \cdot (\alpha^4)^4 + \alpha^7 \cdot (\alpha^4)^3 + \alpha^2 \cdot (\alpha^4)^2 + \alpha^{13} \cdot (\alpha^4)^1 + \alpha^{12} = \alpha^{11}$$

$$s_5 = F_2(\alpha^5) = \alpha^6 \cdot (\alpha^5)^4 + \alpha^3 \cdot (\alpha^5)^3 + \alpha^8 \cdot (\alpha^5)^2 + \alpha^0 \cdot (\alpha^5)^1 + \alpha^7 = \alpha^4$$

Step 4. With the syndromes computed, we can move forward to the key equation solver, which uses BM algorithm in the same way as we did for conventional narrow-sense RS codes. Nevertheless, in this part, there is one difference that the error-locator polynomial and the error-evaluator polynomial are generated together at the same time, utilizing the relationship that $\gamma(x) = (S(x) \cdot \sigma(x)) \mod x^{2t}$, which is discussed in Chapter 2.

The modified BM algorithm is explained below based on [26]. $\sigma(x)$ is the error-locator polynomial; $B(x)$ is the error-locator support polynomial; $\gamma(x)$ is the error-evaluator polynomial; $A(x)$ is the error-evaluator support polynomial; L is a integer variable that indicates the degree of $\sigma(x)$; k is a integer variable; Δ is the discrepancy. The upper index i refers to the i-th iteration and the lower index j represents j-th coefficient that is associated with x^j in polynomials. For example, $\sigma_j^{(i)}$ means the j-th coefficient σ_j of $\sigma(x)$ that is updated at the i-th iteration.

- Initialization:

  $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, \gamma^{(0)}(x) = 0, A^{(0)}(x) = x^{-1}, L^{(0)} = 0$ and $k = 0$
- The algorithm iterates for $r = n - k$ steps. At the (k+1)-th iteration, follow the procedures below:
  1. Calculate the discrepancy: $\Delta^{(k+1)} = \sum_{j=0}^{L^{(k)}} \sigma_j^{(k)} S_{k-j}$.
  2. Update the values of $\sigma(x)$ and $\gamma(x)$. Specifically,

$$\sigma^{(k+1)}(x) = \sigma^{(k)}(x) - \Delta^{(k+1)} \cdot B^{(k)}(x) \cdot x$$

$$\gamma^{(k+1)}(x) = \gamma^{(k)}(x) - \Delta^{(k+1)} \cdot A^{(k)}(x) \cdot x$$

3. Update the values of $B(x), A(x)$ and L. This step is not necessary for the last iteration. Specifically,

$$B^{(k+1)}(x) = \begin{cases} x \cdot B^{(k)}(x) & \text{if } \Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)} \leq k \\ \sigma^{(k)}(x)/\Delta^{(k+1)} & \text{otherwise} \end{cases}$$

$$A^{(k+1)}(x) = \begin{cases} x \cdot A^{(k)}(x) & \text{if } \Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)} \leq k \\ \gamma^{(k)}(x)/\Delta^{(k+1)} & \text{otherwise} \end{cases}$$

$$L^{(k+1)} = \begin{cases} L^{(k)} & \text{if } \Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)} \leq k \\ k+1-L^{(k)} & \text{otherwise} \end{cases}$$

4. If $k = 2t-1$, then stop. Otherwise, $k = k+1$ and return to Procedure 1.

Example 20. Continue with Example 14. The syndromes are $s_0 = \alpha^6, s_1 = \alpha^9, s_2 = \alpha^4, s_3 = \alpha^4, s_4 = \alpha^{11}, s_5 = \alpha^4$, then

- Initialization:

  $\sigma^{(0)}(x) = 1, B^{(0)}(x) = 1, \gamma^{(0)}(x) = 0, A^{(0)}(x) = x^{-1}, L^{(0)} = 0$ and $k = 0$

- Iteration $k+1 = 1$

$$\Delta^{(1)} = \sum_{j=0}^{L^{(0)}} \sigma_j^{(0)} s_{0-j} = \sigma_0^{(0)} \cdot s_0 = 1 \cdot \alpha^6 = \alpha^6$$

$$\sigma^{(1)}(x) = \sigma^{(0)}(x) - \Delta^{(1)} \cdot B^{(0)}(x) \cdot x = 1 - \alpha^6 \cdot 1 \cdot x = 1 - \alpha^6 x$$

$$\gamma^{(1)}(x) = \gamma^{(0)}(x) - \Delta^{(1)} \cdot A^{(0)}(x) \cdot x = 0 - \alpha^6 \cdot x^{-1} \cdot x = \alpha^6$$

As $\Delta^{(1)} = \alpha^6 \neq 0$ and $2L^{(0)} = 0$, thus

$$B^{(1)}(x) = \sigma^{(0)}(x)/\Delta^{(1)} = \alpha^{-6}$$

$$A^{(1)}(x) = \gamma^{(0)}(x)/\Delta^{(1)} = 0$$

$$L^{(1)} = k + 1 - L^{(0)} = 0 + 1 - 0 = 1$$

- Iteration $k + 1 = 2$

$$\Delta^{(2)} = \sum_{j=0}^{L^{(1)}} \sigma_j^{(1)} s_{1-j} = \sigma_0^{(1)} \cdot s_1 + \sigma_1^{(1)} \cdot s_0 = 1 \cdot \alpha^9 + \alpha^6 \cdot \alpha^6 = \alpha^8$$

$$\begin{aligned}\sigma^{(2)}(x) &= \sigma^{(1)}(x) - \Delta^{(2)} \cdot B^{(1)}(x) \cdot x \\ &= 1 - \alpha^6 x - \alpha^8 \cdot \alpha^{-6} \cdot x = 1 - \alpha^3 x\end{aligned}$$

$$\gamma^{(2)}(x) = \gamma^{(1)}(x) - \Delta^{(2)} \cdot A^{(1)}(x) \cdot x = \alpha^6 - \alpha^8 \cdot 0 \cdot x = \alpha^6$$

As $\Delta^{(2)} = \alpha^8 \neq 0$ and $2L^{(1)} = 2 > 1 = k$, thus

$$B^{(2)}(x) = x \cdot B^{(1)}(x) = \alpha^{-6} x$$

$$A^{(2)}(x) = x \cdot A^{(1)}(x) = 0$$

$$L^{(2)} = L^{(1)} = 1$$

- Iteration $k + 1 = 3$

$$\Delta^{(3)} = \sum_{j=0}^{L^{(2)}} \sigma_j^{(2)} s_{2-j} = \sigma_0^{(2)} \cdot s_2 + \sigma_1^{(2)} \cdot s_1 = 1 \cdot \alpha^4 + \alpha^3 \cdot \alpha^9 = \alpha^6$$

$$\begin{aligned}\sigma^{(3)}(x) &= \sigma^{(2)}(x) - \Delta^{(3)} \cdot B^{(2)}(x) \cdot x \\ &= 1 - \alpha^3 x - \alpha^6 \cdot \alpha^{-6} x \cdot x = 1 - \alpha^3 x - x^2\end{aligned}$$

$$\gamma^{(3)}(x) = \gamma^{(2)}(x) - \Delta^{(3)} \cdot A^{(2)}(x) \cdot x = \alpha^6 - \alpha^6 \cdot 0 \cdot x = \alpha^6$$

As $\Delta^{(3)} = \alpha^6 \neq 0$ and $2L^{(2)} = 2 = k$, thus

$$B^{(3)}(x) = \sigma^{(2)}(x)/\Delta^{(3)} = (1 - \alpha^3 x) \cdot \alpha^{-6} = \alpha^{-6} - \alpha^{-3} x$$

$$A^{(3)}(x) = \gamma^{(2)}(x)/\Delta^{(3)} = \alpha^6 \cdot \alpha^{-6} = 1$$

$$L^{(3)} = k + 1 - L^{(2)} = 2 + 1 - 1 = 2$$

- Iteration $k + 1 = 4$

$$\begin{aligned}
\Delta^{(4)} &= \sum_{j=0}^{L^{(3)}} \sigma_j^{(3)} s_{3-j} = \sigma_0^{(3)} \cdot s_3 + \sigma_1^{(3)} \cdot s_2 + \sigma_2^{(3)} \cdot s_1 \\
&= 1 \cdot \alpha^4 + \alpha^3 \cdot \alpha^4 + \alpha^0 \cdot \alpha^9 = \alpha^1 \\
\sigma^{(4)}(x) &= \sigma^{(3)}(x) - \Delta^{(4)} \cdot B^{(3)}(x) \cdot x \\
&= 1 - \alpha^3 x - x^2 - \alpha^1 \cdot (\alpha^{-6} - \alpha^{-3} x) \cdot x = 1 - \alpha^{12} x - \alpha^6 x^2 \\
\gamma^{(4)}(x) &= \gamma^{(3)}(x) - \Delta^{(4)} \cdot A^{(3)}(x) \cdot x = \alpha^6 - \alpha^1 \cdot 1 \cdot x = \alpha^6 - \alpha^1 x
\end{aligned}$$

As $\Delta^{(4)} = \alpha^1 \neq 0$ and $2L^{(3)} = 4 > 3 = k$, thus

$$\begin{aligned}
B^{(4)}(x) &= x \cdot B^{(3)}(x) = \alpha^{-6} x - \alpha^{-3} x^2 \\
A^{(4)}(x) &= x \cdot A^{(3)}(x) = x \\
L^{(4)} &= L^{(3)} = 2
\end{aligned}$$

- Iteration $k + 1 = 5$

$$\begin{aligned}
\Delta^{(5)} &= \sum_{j=0}^{L^{(4)}} \sigma_j^{(4)} s_{4-j} = \sigma_0^{(4)} \cdot s_4 + \sigma_1^{(4)} \cdot s_3 + \sigma_2^{(4)} \cdot s_2 \\
&= 1 \cdot \alpha^{11} + \alpha^{12} \cdot \alpha^4 + \alpha^6 \cdot \alpha^4 = \alpha^7 \\
\sigma^{(5)}(x) &= \sigma^{(4)}(x) - \Delta^{(5)} \cdot B^{(4)}(x) \cdot x \\
&= 1 - \alpha^{12} x - \alpha^6 x^2 - \alpha^7 \cdot (\alpha^{-6} x - \alpha^{-3} x^2) \cdot x \\
&= 1 - \alpha^1 2x - \alpha^{11} x^2 + \alpha^4 x^3 \\
\gamma^{(5)}(x) &= \gamma^{(4)}(x) - \Delta^{(5)} \cdot A^{(4)}(x) \cdot x = \alpha^6 - \alpha^1 x - \alpha^7 \cdot x \cdot x \\
&= \alpha^6 - \alpha^1 x - \alpha^7 x^2
\end{aligned}$$

As $\Delta^{(5)} = \alpha^7 \neq 0$ and $2L^{(4)} = 4 = k$, thus

$$\begin{aligned} B^{(5)}(x) &= \sigma^{(4)}(x)/\Delta^{(5)} = (1 - \alpha^{12}x - \alpha^6 x^2) \cdot \alpha^{-7} \\ &= \alpha^{-7} - \alpha^5 x - \alpha^{-1} x^2 \\ A^{(5)}(x) &= \gamma^{(4)}(x)/\Delta^{(5)} = (\alpha^6 - \alpha^1 x) \cdot \alpha^{-7} = \alpha^{-1} - \alpha^{-6} x \\ L^{(5)} &= k + 1 - L^{(4)} = 4 + 1 - 2 = 3 \end{aligned}$$

- Iteration $k + 1 = 6$

$$\begin{aligned} \Delta^{(6)} &= \sum_{j=0}^{L^{(5)}} \sigma_j^{(5)} s_{5-j} = \sigma_0^{(5)} \cdot s_5 + \sigma_1^{(5)} \cdot s_4 + \sigma_2^{(5)} \cdot s_3 + \sigma_3^{(5)} \cdot s_2 \\ &= 1 \cdot \alpha^4 + \alpha^{12} \cdot \alpha^{11} + \alpha^{11} \cdot \alpha^4 + \alpha^4 \cdot \alpha^4 \\ &= \alpha^1 \\ \sigma^{(6)}(x) &= \sigma^{(5)}(x) - \Delta^{(6)} \cdot B^{(5)}(x) \cdot x \\ &= 1 - \alpha^{12}x - \alpha^{11}x^2 + \alpha^4 x^3 - \alpha^1 \cdot (\alpha^{-7} - \alpha^5 x - \alpha^{-1} x^2) \cdot x \\ &= 1 - \alpha^8 x - \alpha^1 x^2 - \alpha^1 x^3 \\ \gamma^{(6)}(x) &= \gamma^{(5)}(x) - \Delta^{(6)} \cdot A^{(5)}(x) \cdot x \\ &= \alpha^6 - \alpha^1 x - \alpha^7 x^2 - \alpha^1 \cdot (\alpha^{-1} - \alpha^{-6} x) \cdot x \\ &= \alpha^6 + \alpha^4 x + \alpha^6 x^2 \end{aligned}$$

In conclusion, the error-locator polynomial is $\sigma(x) = \sigma^{(6)}(x) = 1 - \alpha^8 x - \alpha^1 x^2 - \alpha^1 x^3$, and the error-evaluator polynomial is $\gamma(x) = \gamma^{(6)}(x) = \alpha^6 + \alpha^4 x + \alpha^6 x^2$.

Step 5. With the error-locator polynomial computed, we can move forward to finding out the error locations. This step can be done using the conventional Chien search algorithm except that to check if r_{n-i} is an error location, the root to substitute is α_i instead of α^i where $i = 0, 1, \cdots, n-1$.

The change is because α_i is used as roots in the definition of $C(n, k; \alpha, \rho)$. Nevertheless, based on the parallel feature of proposed low-latency RS codes, Dr. Shokrollahi developed a modified Chien Search which can run in parallel as well. The novel idea is to split the error-locator polynomial into p components and substitute roots to all of them. The computed results from all the components are then sent to IDFT. If there is any 0 among the p values generated from IDFT, then the same number of error positions could be located. The most significant advantage of this method is that we only need to substitute $m = n/p$ roots instead of n roots.

Specifically, assume there are v ($v \leq \frac{n-k}{2}$) errors in the received word and the error-locator polynomial is found that $\sigma(x) = \sigma_0 + \sigma_1 x + \cdots + \sigma_v x^v$. For $i = [0, 1, \cdots, m-1]$ ($i = [l, l+1, \cdots, m-1]$, where $l = (n-k)/p$, if errors on parity-bits are not the concern) and $k = (0, 1, \cdots, p-1)$, define $a_k = \Sigma_{j=k \bmod p}^{0 \leq j \leq v} \sigma_j \alpha^{-i \cdot j}$, and then apply inverse Fourier transform on a_k to obtain $w_k = IDFT(a)_k = \Sigma_{j=0}^{p-1} \rho^{-j \cdot k} a_j$. If for some k, the value w_k is zero, then the index of one error location is $y = (p \cdot i + k)$. Repeat the procedures for all value of i. The proof of this method is illustrated below.

First, we note that $\sigma(\alpha^{-i}\rho^j) = 0$ if and only if $(p \cdot i + j)$ is an error location, because d is an error position only when $\sigma(\alpha_d^{-1}) = 0$, and recall that a_d is defined as $\alpha_d = \rho^{(d \bmod p)} \alpha^{\lfloor d/p \rfloor}$. Therefore, d must be equal to $(p \cdot i + j)$. Define a polynomial $b(x) = \sigma(x) \bmod (x^p - \alpha^{-p \cdot i})$. Then

$$\begin{aligned} b(\alpha^{-i}\rho^{-j}) &= \sigma(\alpha^{-i}\rho^{-j}) \bmod (\alpha^{-p \cdot i}\rho^{-p \cdot j} - \alpha^{-p \cdot i}) \\ &= \sigma(\alpha^{-i}\rho^{-j}) \bmod (\alpha^{-p \cdot i} \cdot 1 - \alpha^{-p \cdot i}) \\ &= \sigma(\alpha^{-i}\rho^{-j}) \bmod 0 \\ &= \sigma(\alpha^{-i}\rho^{-j}) \end{aligned}$$

Besides,

$$\begin{aligned} b(x) &= \sigma(x) \bmod (x^p - \alpha^{-p\cdot i}) \\ &= \Sigma_{j\equiv 0 \bmod p}^{0\le j\le v} \sigma_j \alpha^{-ij} + x(\Sigma_{j\equiv 1 \bmod p}^{0\le j\le v} \sigma_j \alpha^{-i(j-1)}) + \cdots \\ &+ x^{p-1}(\Sigma_{j\equiv p-1 \bmod p}^{0\le j\le v} \sigma_j \alpha^{-i(j-p+1)}) \\ &= a_0 + a_1 \frac{x}{\alpha^{-i}} + \cdots + a_{p-1}(\frac{x}{\alpha^{-i}})^{p-1} \end{aligned}$$

Therefore, $b(\alpha^{-i}x) = a_0 + a_1 x + \cdots + a_{p-1}x^{p-1}$. Define $a(x) = b(\alpha^{-i}x)$ and thus $w_j = a(\rho^{-j})$ and it is equal to 0 if and only if $b(\alpha^{-i}\rho^{-j}) = \sigma(\alpha^{-i}\rho^{-j}) = 0$, that is, $(p \cdot i + j)$ is an error location.

Example 21. Continuing with Example 20, the error-locator polynomial is $\sigma(x) = \sigma^{(6)}(x) = 1 - \alpha^8 x - \alpha^1 x^2 - \alpha^1 x^3 = \sigma_0 + \sigma_1 x + \sigma_2 x^2 + \sigma_3 x^3$ and $v = 3$. Then

- For $i = 0$

  $a_0 = \Sigma_{(j=0 \bmod 3)}^{0\le j\le 3} \sigma_j \alpha^{-0\cdot j} = \sigma_0 \cdot \alpha^0 + \sigma_3 \cdot \alpha^0 = 1 + \alpha^1 = \alpha^4$

  $a_1 = \Sigma_{(j=1 \bmod 3)}^{0\le j\le 3} \sigma_j \alpha^{-0\cdot j} = \sigma_1 \cdot \alpha^0 = \alpha^8$

  $a_2 = \Sigma_{(j=2 \bmod 3)}^{0\le j\le 3} \sigma_j \alpha^{-0\cdot j} = \sigma_2 \cdot \alpha^0 = \alpha^1$

  $w_0 = \Sigma_{j=0}^{2} \rho^{-j\cdot 0} a_j = \alpha_0 + a_1 + a_2 = \alpha^4 + \alpha^8 + \alpha^1 = \alpha^2$

  $w_1 = \Sigma_{j=0}^{2} \rho^{-j\cdot 1} a_j = \alpha_0 + \rho^2 a_1 + \rho a_2 = \alpha^4 + \alpha^{10} \cdot \alpha^8 + \alpha^5 \cdot \alpha^1 = \alpha^{10}$

  $w_2 = \Sigma_{j=0}^{2} \rho^{-j\cdot 2} a_j = \alpha_0 + \rho a_1 + \rho^2 a_2 = \alpha^4 + \alpha^5 \cdot \alpha^8 + \alpha^{10} \cdot \alpha^1 = 0$

  Therefore, $r_{3\cdot i+k} = r_{3\cdot 0+2} = r_2 = r(12)$ is an error location.

- For $i = 1$

$a_0 = \Sigma_{(j=0 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-1 \cdot j} = \sigma_0 \cdot \alpha^0 + \sigma_3 \cdot \alpha^{-3} = 1 + \alpha^{-2} = \alpha^6$

$a_1 = \Sigma_{(j=1 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-1 \cdot j} = \sigma_1 \cdot \alpha^{-1} = \alpha^8 \cdot \alpha^{-1} = \alpha^7$

$a_2 = \Sigma_{(j=2 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-1 \cdot j} = \sigma_2 \cdot \alpha^{-2} = \alpha^1 \cdot \alpha^{-2} = \alpha^{-1} = \alpha^{14}$

$w_0 = \Sigma_{j=0}^{2} \rho^{-j \cdot 0} a_j = \alpha_0 + a_1 + a_2 = \alpha^6 + \alpha^7 + \alpha^{14} = \alpha^{11}$

$w_1 = \Sigma_{j=0}^{2} \rho^{-j \cdot 1} a_j = \alpha_0 + \rho^2 a_1 + \rho a_2 = \alpha^6 + \alpha^{10} \cdot \alpha^7 + \alpha^5 \cdot \alpha^{14} = \alpha^7$

$w_2 = \Sigma_{j=0}^{2} \rho^{-j \cdot 2} a_j = \alpha_0 + \rho a_1 + \rho^2 a_2 = \alpha^6 + \alpha^5 \cdot \alpha^7 + \alpha^{10} \cdot \alpha^{14} = \alpha^{14}$

- For $i = 2$

$a_0 = \Sigma_{(j=0 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-2 \cdot j} = \sigma_0 \cdot \alpha^0 + \sigma_3 \cdot \alpha^{-6} = 1 + \alpha^{-5} = \alpha^5$

$a_1 = \Sigma_{(j=1 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-2 \cdot j} = \sigma_1 \cdot \alpha^{-2} = \alpha^8 \cdot \alpha^{-2} = \alpha^6$

$a_2 = \Sigma_{(j=2 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-2 \cdot j} = \sigma_2 \cdot \alpha^{-4} = \alpha^1 \cdot \alpha^{-4} = \alpha^{-3} = \alpha^{12}$

$w_0 = \Sigma_{j=0}^{2} \rho^{-j \cdot 0} a_j = \alpha_0 + a_1 + a_2 = \alpha^5 + \alpha^6 + \alpha^{12} = \alpha^8$

$w_1 = \Sigma_{j=0}^{2} \rho^{-j \cdot 1} a_j = \alpha_0 + \rho^2 a_1 + \rho a_2 = \alpha^5 + \alpha^{10} \cdot \alpha^6 + \alpha^5 \cdot \alpha^{12} = 0$

$w_2 = \Sigma_{j=0}^{2} \rho^{-j \cdot 2} a_j = \alpha_0 + \rho a_1 + \rho^2 a_2 = \alpha^5 + \alpha^5 \cdot \alpha^6 + \alpha^{10} \cdot \alpha^{12} = \alpha^4$

Therefore, $r_{3 \cdot i + k} = r_{3 \cdot 2 + 1} = r_7 = r(7)$ is an error location.

- For $i = 3$

$a_0 = \Sigma_{(j=0 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-3 \cdot j} = \sigma_0 \cdot \alpha^0 + \sigma_3 \cdot \alpha^{-9} = 1 + \alpha^{-8} = \alpha^9$

$a_1 = \Sigma_{(j=1 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-3 \cdot j} = \sigma_1 \cdot \alpha^{-3} = \alpha^8 \cdot \alpha^{-3} = \alpha^5$

$a_2 = \Sigma_{(j=2 \mod 3)}^{0 \le j \le 3} \sigma_j \alpha^{-3 \cdot j} = \sigma_2 \cdot \alpha^{-6} = \alpha^1 \cdot \alpha^{-6} = \alpha^{-5} = \alpha^{10}$

$$w_0 = \Sigma_{j=0}^{2} \rho^{-j\cdot 0} a_j = \alpha_0 + a_1 + a_2 = \alpha^9 + \alpha^5 + \alpha^{10} = \alpha^7$$

$$w_1 = \Sigma_{j=0}^{2} \rho^{-j\cdot 1} a_j = \alpha_0 + \rho^2 a_1 + \rho a_2 = \alpha^9 + \alpha^{10} \cdot \alpha^5 + \alpha^5 \cdot \alpha^{10} = \alpha^9$$

$$w_2 = \Sigma_{j=0}^{2} \rho^{-j\cdot 2} a_j = \alpha_0 + \rho a_1 + \rho^2 a_2 = \alpha^9 + \alpha^5 \cdot \alpha^5 + \alpha^{10} \cdot \alpha^{10} = \alpha^7$$

- For $i = 4$

$$a_0 = \Sigma_{(j=0 \mod 3)}^{0\le j\le 3} \sigma_j \alpha^{-4\cdot j} = \sigma_0 \cdot \alpha^0 + \sigma_3 \cdot \alpha^{-12} = 1 + \alpha^{-11} = \alpha^1$$

$$a_1 = \Sigma_{(j=1 \mod 3)}^{0\le j\le 3} \sigma_j \alpha^{-4\cdot j} = \sigma_1 \cdot \alpha^{-4} = \alpha^8 \cdot \alpha^{-4} = \alpha^4$$

$$a_2 = \Sigma_{(j=2 \mod 3)}^{0\le j\le 3} \sigma_j \alpha^{-4\cdot j} = \sigma_2 \cdot \alpha^{-8} = \alpha^1 \cdot \alpha^{-8} = \alpha^{-7} = \alpha^8$$

$$w_0 = \Sigma_{j=0}^{2} \rho^{-j\cdot 0} a_j = \alpha_0 + a_1 + a_2 = \alpha^1 + \alpha^4 + \alpha^8 = \alpha^2$$

$$w_1 = \Sigma_{j=0}^{2} \rho^{-j\cdot 1} a_j = \alpha_0 + \rho^2 a_1 + \rho a_2 = \alpha^1 + \alpha^{10} \cdot \alpha^4 + \alpha^5 \cdot \alpha^8 = \alpha^5$$

$$w_2 = \Sigma_{j=0}^{2} \rho^{-j\cdot 2} a_j = \alpha_0 + \rho a_1 + \rho^2 a_2 = \alpha^1 + \alpha^5 \cdot \alpha^4 + \alpha^{10} \cdot \alpha^8 = 0$$

Therefore, $r_{3\cdot i+k} = r_{3\cdot 4+2} = r_{14} = r(0)$ is an error location.

In conclusion, the three errors are at r_2, r_7, r_{14}, that is, $r(12), r(7), r(0)$.

Step 6. After acquiring the error locations, we can find out the particular error value on each location using Forney's algorithm [24]. The formula is shown below. However, comparing with conventional narrow-sense RS codes, there are two differences here. First, the roots to substitute into the formula is $(\alpha_i)^{(-1)}$ instead of $(\alpha^i)^{(-1)}$, because α_i is used as roots in the definition of $C(n, k; \alpha, \rho)$. Second, there is an extra x multiplied with the denominator of the formula due to the nature of generalized RS codes [27].

Concretely, let the error-locator polynomial be $\sigma(x) = \sigma_0 + \sigma_1 x + \cdots + \sigma_v x^v$ and the error-evaluator polynomial be $\gamma(x) = \gamma_0 + \gamma_1 x + \cdots + \gamma_{(v-1)} x^{(v-1)}$. Thus

$$e_j = \left.\frac{\gamma(x)}{x \cdot \sigma'(x)}\right|_{x=(\alpha_j)^{-1}} = \left.\frac{\gamma(x)}{\sigma_o(x)}\right|_{x=(\alpha_j)^{-1}}$$

Where $\sigma'(x)$ is the derivative of $\sigma(x)$. Recall Section 2.3.2 that $\sigma'(x) = \Sigma^{0\le i\le v}_{(i=1 \bmod 2)} \sigma_i x^{(i-1)}$. Thus $\sigma_o(x) = x \cdot \sigma'(x) = \Sigma^{0\le i\le v}_{(i=1 \bmod 2)} \sigma_i x^i$.

Example 22. Continuing with Example 21, the three error locations are found as r_2, r_7, r_{14}, the error-locator polynomial is $\sigma(x) = \sigma^{(6)}(x) = 1 - \alpha^8 x - \alpha^1 x^2 - \alpha^1 x^3$, and the error-evaluator polynomial is $\gamma(x) = \gamma^{(6)}(x) = \alpha^6 + \alpha^4 x + \alpha^6 x^2$. From Table 3–1, we can find that $\alpha_2 = \alpha^{10}, \alpha_7 = \alpha^7, \alpha_{14} = \alpha^{14}$. Then,

$$\sigma_o(x) = x \cdot \sigma'(x) = \Sigma^{0\le i\le 3}_{(i=1 \bmod 2)} \sigma_i x^i = \sigma_1 x^1 + \sigma_3 x^3 = \alpha^8 x + \alpha^1 x^3$$

$$e_2 = \left.\frac{\gamma(x)}{\sigma_o(x)}\right|_{x=(\alpha_2)^{-1}} = \frac{\alpha^6 + \alpha^4 \cdot \alpha^{-10} + \alpha^6 \cdot (\alpha^{-10})^2}{\alpha^8 \cdot \alpha^{-10} + \alpha^1 \cdot (\alpha^{-10})^3} = \frac{\alpha^2}{\alpha^{12}} = \alpha^{-10} = \alpha^5$$

$$e_7 = \left.\frac{\gamma(x)}{\sigma_o(x)}\right|_{x=(\alpha_7)^{-1}} = \frac{\alpha^6 + \alpha^4 \cdot \alpha^{-7} + \alpha^6 \cdot (\alpha^{-7})^2}{\alpha^8 \cdot \alpha^{-7} + \alpha^1 \cdot (\alpha^{-7})^3} = \frac{\alpha^3}{\alpha^8} = \alpha^{-5} = \alpha^{10}$$

$$e_{14} = \left.\frac{\gamma(x)}{\sigma_o(x)}\right|_{x=(\alpha_{14})^{-1}} = \frac{\alpha^6 + \alpha^4 \cdot \alpha^{-14} + \alpha^6 \cdot (\alpha^{-14})^2}{\alpha^8 \cdot \alpha^{-14} + \alpha^1 \cdot (\alpha^{-14})^3} = \frac{\alpha^{12}}{\alpha^{14}} = \alpha^{-2} = \alpha^{13}$$

Step 7. Finally, we can obtain the correction-words by adding the error value to the received value at the error locations. $r_i^{'} = r_i + e_i$ where $i \in E$ and E denotes the set of v error positions.

Example 23. Continue with Example 22. The received-words are $[r_{14}, r_{13}, \cdots, r_2, r_1, r_0] = [\alpha^5, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^7, \alpha^3, \alpha^6, 0, \alpha^{12}, \alpha^5, \alpha^1,$

$\alpha^1]$. There are three error locations r_2, r_7 and r_{14}. The three error values are $e_2 = \alpha^5$, $e_7 = \alpha^{10}$ and $e_{14} = \alpha^{13}$.

Then the three corrected values are

$$r'_2 = \alpha^5 + \alpha^5 = 0$$

$$r'_7 = \alpha^7 + \alpha^{10} = \alpha^6$$

$$r'_{14} = \alpha^5 + \alpha^{13} = \alpha^7$$

and the corrected code-word is thus

$$r' = [\alpha^7, \alpha^4, \alpha^1, \alpha^8, \alpha^5, \alpha^2, \alpha^9, \alpha^6, \alpha^3, \alpha^6, 0, \alpha^{12}, 0, \alpha^1, \alpha^1]$$

which is consistence with the original code-word c generated from Example 16.

3.3 Algorithm Verification in MATLAB and BER Performance

In this section, implementations of the low-latency RS codes in MATLAB using high-level programming techniques are illustrated. Before we start to discuss the codes and results, there are two things to be clarified. First, both of the encoder and decoder example given in this section can handle almost any arbitrary code-word length n, message-word length k and parity-bit length $r = n - k$. The only constraint is that they all need to be divisible by the speed-up coefficient p. Solutions to arbitrary k and r are discussed in the next section. Second, we set $n = q - 1$ in this implementation to make it easier to understand, but it does not always have to be the case, and it would not affect the algorithm as long as $1 \leq n \leq q$ due to the nature of GRS. The functionality of this newly proposed algorithm was verified, and BER performance was measured using Communication System Toolbox in MATLAB.

First of all, two MATLAB function files defining DFT and IDFT should be introduced, as they are essential functions used both in low-latency RS encoders and decoders. They are simple to realize in MATLAB with high-level coding technique. Figure 3–3 shows the DFT function and follows the same formula given in previous sections. The output generated is a matrix and each row represents a $FT(X)_i$ where $i = 0, 1, \cdots, p-1$. The only inconsistent between the codes and formula given before is that the index of a matrix in MATLAB must start with $(1, 1)$ instead of $(0, 0)$. Therefore, we have to change i, j in the equation to $(i-1), (j-1)$, respectively. Figure 3–4 shows the IDFT function and it is same as DFT function except that a minus sign is added to the exponential order of $((i-1) \cdot (j-1))$.

DFT.m

```
1   function F=DFT(InputPoly,p,a,q)
2   %InputPoly refers to the input polynomial
3    %p refers to the speed-up coefficient
4    %a refers to the primitive element of GF(2^GFPower)
5    %q refers to the p-th root of unity
6    [~,m]=size(InputPoly); %to find out the length of input polynomial
7    F=a.*zeros(p,m);
8    for i=1:p
9        for j=1:p
10           F(i,:)=F(i,:)+q^((i-1)*(j-1))*InputPoly(j,:);
11       end
12   end
13   end
```

Figure 3–3: MATLAB codes of DFT function.

3.3.1 Implementation of the Encoder in MATLAB

The encoder structure follows the one shown in Figure 3–1. Similar to the steps defined in Section 3.2.1, the MATLAB codes of this encoder are divided into six parts. In the following paragraphs, each part is explained in details, and more comments can be found in the screenshots of their MATLAB codes.

The first part is to initialize parameters (Figure 3–5). The function name is ***LowLatRSenc*** which stands for "Low-Latency Reed-Solomon Encoder".

IDFT.m

```
function F=IDFT(InputPoly,p,a,q)
%InputPoly refers to the input polynomial
%p refers to the speed-up coefficient
%a refers to the primitive element of GF(2^GFPower)
%q refers to the p-th root of unity
[~,m]=size(InputPoly);%to find out the length of input polynomial
F=a.*zeros(p,m);
for i=1:p
    for j=1:p
        F(i,:)=F(i,:)+q^(-(i-1)*(j-1))*InputPoly(j,:);
    end
end
end
```

Figure 3–4: MATLAB codes of IDFT function.

There are three inputs in which "msg" refers to user-defined message-words in the form of an integer vector, "n" represents user-defined code-word length and "p" is the user-defined speed-up coefficient. As we can see from Figure 3–5, all needed parameters can be calculated from the three inputs. Due to internal setting of MATLAB, it cannot interpret integers as elements of Galois field. Therefore, input message-words needs to be converted from integer type to Galois field type using command "msgGF=gf(msg,GFPower)", while they still have the same numerical values after conversion. "GFPower" means the Galois field used is $GF(2^{GFPower})$.

LowLatRSenc.m

```
function c=LowLatRSenc(msg,n,p)
[~,k]=size(msg);
r=(n-k);
GFPower=log2(n+1); % the degree of Galois field (GF 2^m)
m=n/p;
t=k/p;
L=r/p;

a=gf(2,GFPower);% to produce the primitive element of GF(2^GFPower)
q=a^m;%to produce the p-th root of unity.
msgGF=gf(msg,GFPower); %to convert msg-words from integer type to GF type
```

Figure 3–5: MATLAB codes of low-latency RS encoder: Part 1.

The second part is to produce the set of generator polynomials (Figure 3–6). The result acquired in this step is a matrix "g", and each row represents

a generator polynomial. Its first row refers to the generator polynomial g_0; its second row refers to g_1 and so on and so forth. Each row is initialized as 1 at first, and then a loop makes each row multiply with $(x - \alpha^i)$, where α^i are the roots of each generator polynomial. Based on the method stated in Step 1 of Section 3.2.1, i is set dependent on the loop counter, row number, and the speed-up coefficient p.

LowLatRSenc.m

```
13
14      %to produce p generator polynomials
15      %in this case, the sizes of all p polynomials are L+1
16 -    g=a*zeros(p,L+1);
17 -    g(:,end)=1; %initialize each row to 1, so it can work with muliplication loop
18 -    for i=1:p
19 -        for j=1:L
20 -            int1=conv(g(i,:),[1 a^(p*(j-1)+(i-1))]);
21              % conv(x,y) refers to the muliplication of 2 polynomials.
22 -            g(i,:)=int1(end-L:end);
23              % to eliminate unwanted "0" terms due to muliplication of 2 polynomials
24 -        end
25 -    end
26
```

Figure 3–6: MATLAB codes of low-latency RS encoder: Part 2.

Part 3 shown in Figure 3–7 is to split message-words to p components following the way stated in Step 2 of Section 3.2.1. The only inconsistence is that the index of a matrix in MATLAB starts with $(1, 1)$ instead of $(0, 0)$.

LowLatRSenc.m

```
25
26 -    msgPoly=a^0*zeros(p,t); %so that the type of this default vector is GF not double
27      %to split the message-word in to 3 parts
28      %e.g. the message [d_k-1 d_k-2 d_k-3 d_k-4 d_k-5 ... d_k-9] is divided into three row
29      %as follows:msg1=[d_k-3 d_k-6 d_k-9]; msg2=[d_k-2 d_k-5 d_k-8]; msg3=[d_k-1 d_k-4 d_k-7]
30 -    for i=1:p
31 -        for j=1:t
32 -            msgPoly(i,j)=msgGF(j*p-(i-1));
33 -        end
34 -    end
35
```

Figure 3–7: MATLAB codes of low-latency RS encoder: Part 3.

Part 4 shown in Figure 3–8 is to generate the Fourier transform of message-words using the DFT function stated at the beginning of Section 3.3.

Part 5 shown in Figure 3–9 is to generate the set of parity bits following the method stated in Step 4 of Section 3.2.1. Specifically, x^l is first produces

LowLatRSenc.m

```
39      %Fourier transform of message polynomials
40 -    FT_msg=DFT(msgPoly,p,a,q);
41
```

Figure 3–8: MATLAB codes of low-latency RS encoder: Part 4.

by a multiplication loop. Then with command "conv", each of p components resulted from DFT is shifted by x^l. Afterward, f_i are calculated from modulo arithmetic and finally, the parity bits are computed by IDFT function stated at the beginning of Section 3.3.

LowLatRSenc.m

```
42 -    x=gf([1 0],GFPower); %to produce x
43 -    xl=a^0; %to set default xl=1
44 -    for i=1:L  %to generate x^L
45 -        xl=conv(xl,x); % conv(x,y) refers to the muliplication of 2 polynomials.
46 -    end
47 -    FT_xl=a*zeros(p,t+L);
48 -    for i=1:p %to shift each of p message polynomials by x^L
49 -        FT_xl(i,:)=conv(xl,FT_msg(i,:));
50 -    end
51
52      %to generate the remainder f0,f1,f2
53 -    f=a*zeros(p,t+L);
54 -    for i=1:p
55 -        [~,f(i,:)]=deconv(FT_xl(i,:),g(i,:));
56          % deconv(x,y) refers to the division of 2 polynomials and f(i,:) here
57          % represents the remainder of this division.
58 -    end
59
60      %use Inverse Fourier Transform to generate p parity-bit components,
61 -    h=IDFT(f,p,a,q);
62
```

Figure 3–9: MATLAB codes of low-latency RS encoder: Part 5.

Part 6 shown in Figure 3–10 is to generate the final code-words by putting message-words and parity bits together following the way stated in Step 5 of Section 3.2.1.

The functionality of this encoder has been tested, and it worked properly. Figure 3–11 shows the code-words generated by this encoder in MATLAB with the initial conditions stated in Example 12. As we can see, the result is consistent with Example 16.

```
69 -    c=a*zeros(1,n);
70 -    for i=1:p
71 -        for j=1:t
72 -            c(i+p*(j-1))=msgPoly(end-(i-1),j);
73 -        end
74 -    end
75      %we cannot simply use c(1:k)=msg becuase type of msg is integer while c is over GF.
76
77 -    for i=1:p
78 -        for j=1:L
79 -            c(k+i+p*(j-1))=h(end-(i-1),end-L+j);
80      %the size of h is [p,size of f row vector] due to "deconv" function. However, there are
81      %many zeros ahead in each row vector of h. only the last L elements of each row are the
82      %meaningful elements.
83 -        end
84 -    end;
85 -    c_double=double(c.x); %convert the code-words from type gf to type double
86 -    end
87
```

Figure 3–10: MATLAB codes of low-latency RS encoder: Part 6.

```
Workspace                                    Command Window
>> msg=[11 3 2 5 6 4 10 12 8];
>> n=15;
>> p=3;
>> code_words=LowLatRSenc(msg,n,p)

code_words = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

  Columns 1 through 9

        11         3         2         5         6         4        10        12         8

  Columns 10 through 15

        12         0        15         0         2         2

fx >>
```

Figure 3–11: Output of low-latency RS encoder in MATLAB with circumstances in Example 12.

3.3.2 Implementation of the Decoder in MATLAB

The implementation of a decoder is much more complicated than that of a encoder. The decoder structure follows the one shown in Figure 3–2. The low-latency decoder consists five component MATLAB function files: 1) ***LowLatRSsyn*** for syndrome calculation, 2) ***LowLatRSbm*** for key equation solver using Berlekamp-Massey algorithm, 3) ***LowLatRSchien*** for error location computation by modified Chien search, 4) ***LowLatRSerrValue*** for

error value calculation and finally, 5) ***LowLatRSdec*** as the top file for putting components together and producing correction-words. In the following paragraphs, each file is explained in details, and more comments can be found in the screenshots of their MATLAB codes.

LowLatRSsyn. Figure 3–12 shows the MATLAB codes for syndrome calculation, which essentially combines Step 1, 2 and 3 of Section 3.2.2. The input is the received-words denoted as y, the message-word length denoted as k, and the speed-up coefficient p. The output is a vector that contains all computed syndromes $(s_0, s_1, \cdots, s_(n-k-1))$. The whole process starts with calculating all needed parameters based on the three inputs. Then, the received-words are divided into p components followed by a discrete Fourier transform that produces a matrix F. Each row of F refers to a polynomial F_i where $i = 0, 1, \cdots, p-1$. Afterward, we set up the roots which are then substituted into F_i to calculate syndromes. Command "polyval(x,y)" is a function to evaluate polynomials x by substituting y. Recall that the exponential order of roots decides in which polynomial the particular root is substituted. Specifically, α^i is substituted into $F_{i \bmod p}$ where $i = 0, 1, \cdots, p-1$. However, since the index of a vector in MATLAB must start with 1, we have to change i to $(i-1)$ in the codes and then plus 1 after modulo function "mod".

LowLatRSbm. ***LowLatRSbm*** is designed as a component to compute error-locator polynomial and error-evaluator polynomial by utilizing the BM algorithm. The MATLAB codes are shown in Figure 3–13. This process exactly follows Step 4 in Section 3.2.2. For this function, there are two inputs: syndrome vector denoted as "syndrome" which includes all syndromes $(s_0, s_1, \cdots, s_{(n-k-1)})$ and "GFPower" which defines Galois field $GF(2^{GFPower})$. There are two outputs: an error-locator polynomial denoted as "errorLoc" and an error-evaluator polynomial denoted as "errorEva". As Figure **??** shows, the

LowLatRSsyn.m

```
function syndrome=LowLatRSsyn(y,p,CorrectCapbility)
[~,n]=size(y); %to get the code-word length n
GFPower=log2(n+1); % GF(2^GFPower)
m=n/p; %to get the length of each of p components of received-words
a=gf(2,GFPower); %to get the primitive element of GF(2^GFPower)
q=a^m; %to get the p-th root of unity.

L=a.*zeros(p,m);
%to seperate the received-words into p components and each row is one componet
for i=1:p
    for j=1:m
        L(i,j)=y(p*j-i+1);
%Watch out the index here. y(1) is y_(n-1) which is the coefficient of x^(n-1),
%that is, the highest order coefficient in the received-words polynomial.
    end
end

F=DFT(L,p,a,q); %Fourier transform of the received-words

root_exp=zeros(1,2*CorrectCapbility);
for i=1:(2*CorrectCapbility)
    root_exp(i)=i-1;
    %root_exp is the exponential order of the root, from 0 to (2*CorrectCapbility-1)
end
root=a.^root_exp;
%root is the actual root to produce the syndromes, from a^0 to a^(2*CorrectCapbility-1)

syndrome=a*zeros(1,2*CorrectCapbility);
for i=1:(2*CorrectCapbility)
    syndrome(i)=polyval(F((rem((i-1),p)+1),:),root(i));
    %syndrome is a row vector containing all 2t syndromes(s_0,s_1,...,s(2*CorrectCapbility-1))
end
end
```

Figure 3–12: MATLAB codes of syndrome calculation in low-latency RS decoder.

process starts with calculating all important parameters based on the two inputs and initializing all matrices and vectors just following the BM algorithm, illustrated previously. Afterwards, a loop (Figure 3–13) runs $(n-k)$ iterations, which is to calculate discrepancy "delta" and update the error-locator polynomial "sigma", the error-evaluator polynomial "errEva" and all other supporting variables and polynomials.

LowLatRSchien. Figure 3–14 shows the MATLAB codes for the modified Chien search function. Its purpose is to find out the number of errors denoted as "errNum" and their locations denoted as "errLocation". It should

LowLatRSbm.m

```
function [errorLoc,errorEva]=LowLatRSbm(syndrome,GFPower)
a=gf(2,GFPower);%define the primitive element of the gf.
x=gf([1 0],GFPower);%define "x", which is frequently used in the expressions later.
[~,r]=size(syndrome);%the length of syndrome vector "r"=(n-k)=2* error-correction capability
%sigma is the set of error locator polynomial at different iterations
sigma=a*zeros(r+1,(r/2+1)); %each row refers to sigma at a specific iteration
sigma(1,end)=a^0; %with the first row refer the sigma at initialzation
B=a*zeros(r+1,(r/2+1));%B is the set of supporting polynomial for error locator
B(1,end)=a^0;
%errEva is the set of error evaluator polynomial at different iterations
errEva=a*zeros(r+1,(r/2));
A=a*zeros(r+1,(r/2)); %A is the set of supporting polynomial for error evaluator
%note that the lengths of errEva and A polynomial are both one term less than sigma and B
L=zeros(1,r+1);
delta=a*zeros(1,r+1);
%initialize the parameters L and delta. note that delta(1) is actually not
%used. Defining delta in this way is to make the later computations simpler.

for i=2:r+1 %starting from 2 not 1 because 1 is actual iteration 0 (initialization) in the original algorithm.
    k=i-1;  %I still make k 1 less than i for the index consistance so that the codes
            %can be confined to the original algorithm in the paper as much as possible.
    for j=0:L(k)
        delta(i)=delta(i)+sigma(k,end-j)*syndrome(k-j);
    end
    int1=conv((delta(i)*B(k,:)),x);
    sigma(i,:)=sigma(k,:)+int1(end-(r/2+1)+1:end);
% Compared with B, the size of "int1" is increased, due to the nature of "conv" function,
%but addition needs two vectors with same length. so I define and manipulate this intermedia
    if i==2
        errEva(i,end)=delta(i);
%because in the original algorithm, the initial value of A(x) is x^(-1).It's impossible to represent with GF polynomial
    else
        int2=conv(delta(i)*A(k,:),x);
        errEva(i,:)=errEva(k,:)+int2(end-r/2+1:end);
    end
    if delta(i)==0 || 2*L(k)>(k-1)
%Recall that we started iteration at i=2, but at each iteration still k=i-1, we use (k-1) instead of k because
%the "k" here represents a value not index.that's why we need to keep it as same as stated in the original method.
        int3=conv(x,B(k,:));
        B(i,:)=int3(end-(r/2+1)+1:end);
        if i==2
            A(i,end)=a^0;
        else
            int4=conv(x,A(k,:));
            A(i,:)=int4(end-r/2+1:end);
        end
        L(i)=L(k);
    else
        B(i,:)=conv(delta(i)^(-1),sigma(k,:));
        A(i,:)=conv(delta(i)^(-1),errEva(k,:));
        L(i)=i-1-L(k);
%Recall that we started iteration at i=2 we use (i-1) instead i because here i represents a value
%not index.that's why we need to keep it as same as stated in the original method.
    end
end
errorLoc=sigma(end,:);
errorEva=errEva(end,:);
end
```

Figure 3–13: MATLAB codes of key equation solver in low-latency RS decoder.

be clarified that the computed locations are the index of error terms in the received-words. For instance, "errLocation"$= [2, 7]$ means r_2 and r_7 in the

received-words r are the terms with error. This function requires four inputs: the error-locator polynomial denoted as "sigma", "GFPower" which defines Galois field $GF(2^{GFPower})$, the speed-up coefficient denoted as p and the code-word length denoted as n. This process follows Step 5 in Section 3.2.2. It starts with calculating all needed parameters based on the four inputs. Then there is a loop runs for $m = n/p$ iterations as we need to check m roots. In each iteration, p values are generated because the error-locator polynomial is split into p components and a root is substituted into all the components at each iteration.

LowLatRSchien.m

```
function [errLocation,errNum]=LowLatRSchien(sigma,GFPower,p,n)
[~,v]=size(sigma);%v is the degree of error locator polynomial.
m=n/p; %m is the length of each code-word component.
a=gf(2,GFPower); %define the primitive element of the GF(2^GFPower).
q=a^m;%define the p-th root of unity.
b=a*zeros(m,p);
% m iterations => m rows, error-locator polynomial is divided into p components, so at each iteration p values generated.
w=a*zeros(m,p);

for i=0:(m-1) % m iterations
    for j=0:(p-1)
% p polynomial-evaluation values generated at each iteration because error-location polynomial is divided into p components.
        for k=0:fix((v-1)/p)%this loop is include all terms of each component when do the evaluation.
        %fix((v-1)/p) is to get the integer part of the division.
        %which is to find the minimum number of terms a compoent could receive from error-locator polynomial
            if (k*p+j)<=(v-1) % to ensure the index not over limit
                b(i+1,j+1)=b(i+1,j+1)+sigma(end-(k*p+j))*a^(-i*(k*p+j));
                %b(i+1,j+1) refers to the evaluation result of (j+1) component at (i+1)-th iteration.
                %sigma(end-k*p+j) refers to the coefficient associated with x^(k*p+j)
            end
        end
    end
    % for each iteration, we have to finish polynomial evaluation for all p components before we calculate w, which is a IDFT.
    w(i+1,:)=(IDFT(b(i+1,:)',p,a,q))';
    % the first "'" is because the inputs to the IDFT should be the p values in a column vector. The second "'" is because the
    %outputs from IDFT should be stored in w as a row vector, as each row refers to a set of results at one iteration.
end

%calculate w for all iterations first, then find out if there is any 0
[row,col]=find(w==0);
%compute the error locations with row number and column number of each 0 term
errLocation=(p*(row-1)+(col-1))';
%computed errLocation is the index of term with error, eg. errorlocation(i)=7 means y_7 is
%an error location and y_7 is the coefficient associated with x^7.
%"'" here is to convert a column vector into a row vector to simplify the rest processes of decoder.

[~,errNum]=size(errLocation);
%errNum is the number of errors found
end
```

Figure 3–14: MATLAB codes of Chien search in low-latency RS decoder.

LowLatRSerrValue. Figure 3–15 shows the MATLAB codes for error evaluation function. The outputs of this function are error values on the corresponding error locations that computed in the modified Chien search. There are seven inputs required: the error-locator polynomial denoted as "sigma", the error-evaluator polynomial denoted as "errEva", the error locations denoted as "errorlocation", the number of errors denoted as "errNum", "GFPower" that defines $GF(2^{GFPower})$, the speeding-up coefficient denoted as "p", and the component length of code-words denoted as "m". The process follows Step 6 in Section 3.2.2. As usual, it starts with computing all the necessary parameters. Then it defines the σ_o polynomial which is equal to $(x \cdot \sigma'(x))$ by just assigning the odd-degree terms of σ to σ_o. Afterwards, we need to find the values of $\alpha_{errorlocation}$ over "$GF(2^{GFPower})$" based on the formula $\alpha_i = \rho^{(i \mod p)}\alpha^{\lfloor i/p \rfloor}$. Finally, we substitute these values and use command "polyval" to evaluate the polynomials and obtain the error values.

LowLatRSerrValue.m

```
function errVal=LowLatRSerrValue(sigma, errEva, errorlocation, errNum, GFPower, p, m)
a=gf(2,GFPower); %find the primitive element of the gf.
q=a^m; %to get the p-th root of unity.
oddNum=fix((errNum+1)/2); %to find how many odd-degree terms there are in sigma(x).

sigma_o=a*zeros(1,(errNum+1)); %sigma_o= x * sigma'(x)
for i=0:(oddNum-1)
    sigma_o(end-2*i-1)=sigma(end-2*i-1);
    %assign the odd-degree term value of u to sigma_o. remain other terms of sigma_o to 0.
end

errLoc=a*zeros(1,errNum);
for i=1:(errNum)
    errLoc(i)=q^(rem(errorlocation(i),p))*a^(fix(errorlocation(i)/p));
    %find out the corresponding elements of GF respective to
    %a_errorlocation based on the formula a_i=q^(i mod p)*a^(fix(i/p))
end

errVal=a*zeros(1,errNum);
for j=1:(errNum)
    errVal(j)=polyval(errEva,(errLoc(j)^(-1)))/polyval(sigma_o,(errLoc(j)^(-1)));
    %this is to implement the expression e=v((a_errorlocation)^(-1))/u0((a_errorlocation)^(-1)),
end

end
```

Figure 3–15: MATLAB codes of error evaluation in low-latency RS decoder.

LowLatRSdec. Figure 3–16 shows the MATLAB codes for the top file which connects all functions together and generates the final correction-words. The low-latency decoder requires three inputs: the received-words denoted as "r", the message-word length denoted as "k" and the speed-up coefficient denoted as "p". First, the syndrome calculation function processes the received-words to generate $(n-k)$ syndromes. Then the BM algorithm function produces an error-locator polynomial and an error-evaluator polynomial based on the syndromes. With this information, the modified Chien search function finds out the error positions and the number of errors and passes them to the error evaluation function, in which error values at corresponding locations are computed. In the end, correction-words are output with all errors corrected. More details and explanation are shown in the figures.

LowLatRSdec.m

```
function [correction_word_double, errNum]=LowLatRSdec(r, k, p)
[~, n]=size(r);
% t is the error correction capability
GFPower=log2(n+1);
m=n/p;

syndrome=LowLatRSsyn(r, k, p); %to calculate syndromes, output a vector [s_0, s_1, ... s_(n-k-1)]
[errorLoc, errorEva]=LowLatRSbm(syndrome, GFPower);%to generate error-locator polynomial and error-evaluator polynomial
[errLocation, errNum]=LowLatRSchien(errorLoc, GFPower, p, n);%to compute error locations and the number of errors.
errValue=LowLatRSerrValue(errorLoc, errorEva, errLocation, errNum, GFPower, p, m);
%to compute the error values on corresponding error positions

corrected=gf(r, GFPower); %first copy the received-words, but have to convert them to GF elements for later correction.
for i=1:errNum
    %correct the terms with error in received-words by adding (=minusing in GF(2^GFPower)) the error values
    corrected(end-errLocation(i))=corrected(end-errLocation(i))+errValue(i);
    %errLocation is the index.
    %e.g. errorlocation(i)=7 means r_7 which associates with x^7 is a term with error.
    %while for vectors in MATLAB the highest-order coefficient has the lowest index. e.g 2x^3+9x^2+10 => [2 9 0 10]
    %So I used "end-errLocation(i)" here.
end
correction_word_double=double(corrected.x); %convert the correction-words from type gf to type double
end
```

Figure 3–16: MATLAB codes of top file in low-latency RS decoder.

The functionality of this decoder has been tested and approved. Figure 3–17 shows the generated correction-words under the circumstance stated in Example 17. Compared with Example 22, the results are consistent.

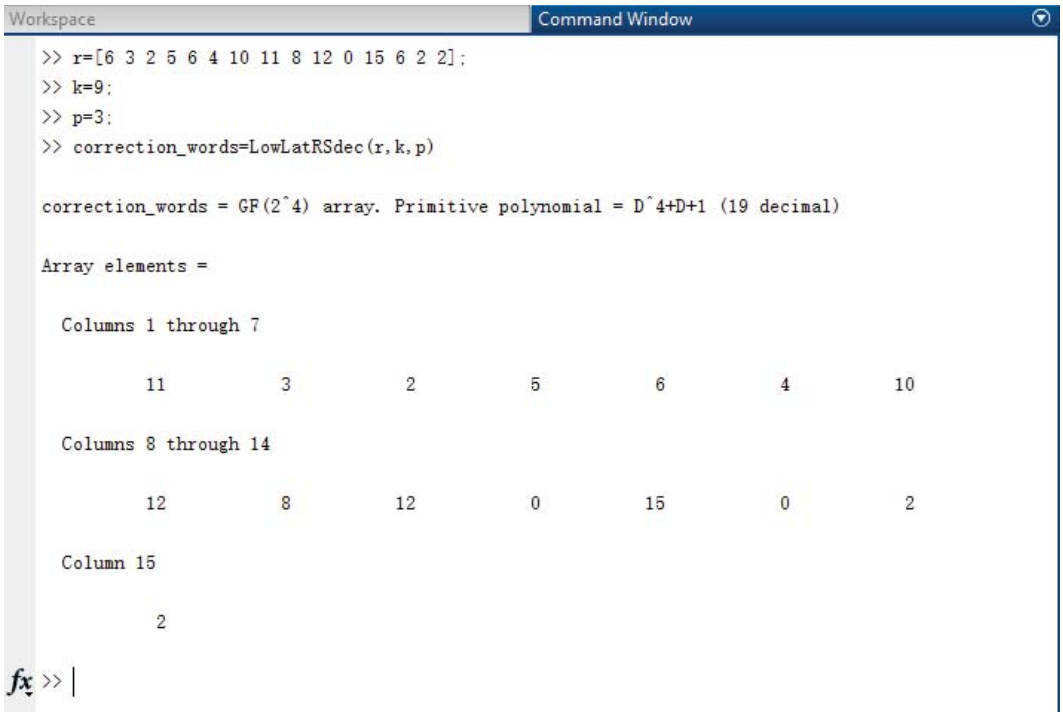


Figure 3–17: Output of low-latency RS decoder in MATLAB with circumstances in Example 17.

3.3.3 BER Performance

BER is one of the most significant criteria to measure the performance of ECC. With millions of applications, RS codes have proved its ability in BER improvement. As BER performance depends on many conditions such as the normalized signal-to-noise ratio E_b/N_o, channel characteristics, modulation type and so on, it is difficult to make an absolute comparison. Moreover, running BER test-bench in MATLAB is very time consuming due to the requirement of a substantial number of transmissions and only extremely stable testing environment could succeed. Therefore, the purpose of this test is to roughly verify the BER performance and functionality of low-latency RS codes. This test is conducted with Communication System Toolbox in MATLAB, which considerably simplified MATLAB coding. We can simply insert any functions and communication components by writing one or two lines of commands.

Figure 3–18 shows the structure of testbench and Figure 3–19 shows the MATLAB codes. There are two parts: configuration and the testbench. The process starts with generating random message-words with length $k = 225$, then the low latency RS encoder encodes the message-words and passes the code-words with length $N = 255$ to the modulator for PAM-4 modulation. The data are then transmitted through an additive-white-Gaussian-noise (AWGN) channel. At the receiver end, the data are first demodulated and then passed the received-words to low latency RS decoder where the correction-words are produced. Then an analyzer makes a comparison between the original message-words and the message part of the correction-words from the decoder. Figure 3–20 shows the results generated from this testbench, and it suggests that the channel BER is $5.3877 \cdot 10^{-4}$, the coded BER is 0, the number of RS decoding error is 0 and the total number of bit transmission is 500000400. Channel BER is referred as the bit error rate due to channel transmissions. Coded BER is referred as the bit error rate after the low-latency RS decoding process. Decoding errors are referred as the data errors still existed after decoding process. It should be clarified that the coded BER being 0 only suggests there is no RS decoding error, during this particular number of transmissions. When the transmission number becomes large enough, then there would be errors after decoding, and the coded BER would not be 0. Therefore, we cannot assert that BER for low latency RS codes is 0. Instead, we can safely say that BER for low latency RS codes is smaller than 10^{-8} under this circumstance.

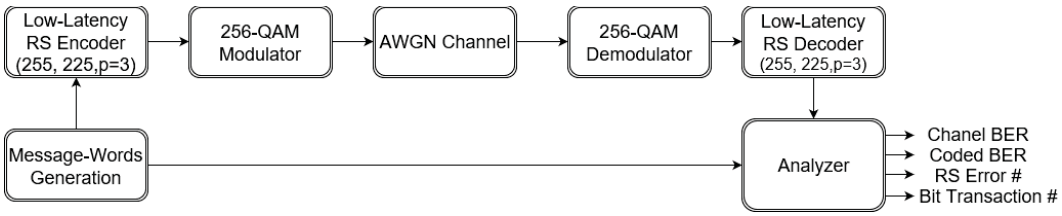


Figure 3–18: Structure of the communication test-bench in MATLAB.

LowLatRS_BERtest.m

```
%%-------------------------beginning of configuration-----------------------------
%set a single uncoded Eb/No value, and set the simulation stop criteria.
if ~exist('initFlagRSDemo', 'var') || initFlagRSDemo
  EbNoUncoded = 10; % dB
  % Set the simulation stop criteria
  targetErrors = 500;
  maxNumTransmissions = 5e8;
end
%% Rectangular 4-PAM Modulation
% Create a rectangular 4-PAM modulator System object. Set the SymbolMapping to 'Gray' for Gray
% coding and set the BitInput property to true to specify that the modulator's input is binary bits.
M = 4; % Modulation order
hMod = comm.PAMModulator(M, 'SymbolMapping', 'Gray',...
    'BitInput', true);
% Create a rectangular 64-QAM demodulator System object with same settings
hDemod = comm.PAMDemodulator(M, 'SymbolMapping', 'Gray',...
  'BitOutput', true);
%% AWGN Channel
% Create an additive white Gaussian noise (AWGN) channel System object. Set the NoiseMethod property
%to 'Signal to noise ratio (Eb/No)' to specify the noise level using the energy per bit to noise power
%spectral density ratio (Eb/No) in dB.
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)');
% We assume no upsampling so the number of samples per symbol is 1. The signal power for 4-PAM is 4
% Watts, so we set the SignalPower property of the channel to this value.
hChan.SamplesPerSymbol = 1;
hChan.SignalPower = 4;
% The number of bits per symbol is equal to log2(M). We need to set the BitsPerSymbol property of the AWGN
% channel System object so that it knows how to distribute noise evenly across the symbol samples.
hChan.BitsPerSymbol = log2(M);
%% Error rate measurement
% Create two error rate measurement System objects, one to measure the channel bit error rate (BER), and
% the other to measure the coded BER. Since the inputs and outputs of the modulator and demodulator are integer
% symbols, and we want to measure bit error rates, we also need to create integer to bit converters.
hChanBERCalc = comm.ErrorRate; % Error rate measurement System object for channel BER
hCodedBERCalc = comm.ErrorRate;% Error rate measurement System object for coded BER
hIntToBit1 = comm.IntegerToBit('BitsPerInteger', 8);%integer-->bit converter
hIntToBit2 = comm.IntegerToBit('BitsPerInteger', 8);
hBitToInt = comm.BitToInteger('BitsPerInteger', 8);%bit-->integer converter
%%-----------------------end of configuration-----------------------------------------
%%-----------------------beginning of test-bench--------------------------------------------------------
N=255; %set code-word length
K=225; %set message-word length
p=3; %set the speed-up coefficient
EbNoCoded=EbNoUncoded+10*log10(K/N); %compute coded E_b/N_o (SNR per bit)
hChan.EbNo=EbNoCoded;%set channel E_b/N_o to the coded E_b/N_o
chanErrorStats=zeros(3,1); %initialize a vector for holding channel error analyzer results
codedErrorStats=zeros(3,1);%initialize a vector for holding coding error analyzer results
while (codedErrorStats(2)<targetErrors)&&(codedErrorStats(3)<maxNumTransmissions)
%set up a loop that breaks only when the number of error exceeds 500 or
%the number of bit transcations of message-word exceed 5*10^8.
    data=randi([0 N],K,1); % randomly generate a column vector as message-words
    encData_int=LowLatRSenc(data',N,p);
    %convert the column vector to a row vector and encode the message-words. Note that it's a row vector.
    encData=(encData_int)';%convert the row vector into column vector
    encDataBit=step(hIntToBit1,encData);%convert encData from integer to bit
    modData=step(hMod,encDataBit);% conduct a 4-PAM modulation to the code-words.
    chanOutput=step(hChan,modData);%mimic that the data pass through a channel
    demodDataBit=step(hDemod,chanOutput);%demodulate the data, generate a column vector
    demodData=step(hBitToInt,demodDataBit);%convert the demodulated data from bits to integer
    [estData_int,errs]=LowLatRSdec(demodData',K,p);%convert the demodulated data into row vector and decode the data
    estData_int2=estData_int(1:K);%we only want the message-word part for later comparison. Recall k=225
    estData=(estData_int2)';%convert the data from row vector to a column vector.
   %analyze the results
   chanErrorStats(:,1)=step(hChanBERCalc,encDataBit,demodDataBit);
   codedErrorStats(:,1)=step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
chanelBitErrorRate=chanErrorStats(1)
codedBitErrorRate=codedErrorStats(1)
RS_Errors=codedErrorStats(2)
totalNumberofBitTransactions=codedErrorStats(3)
```

Figure 3–19: MATLAB codes of the BER performance test-bench.

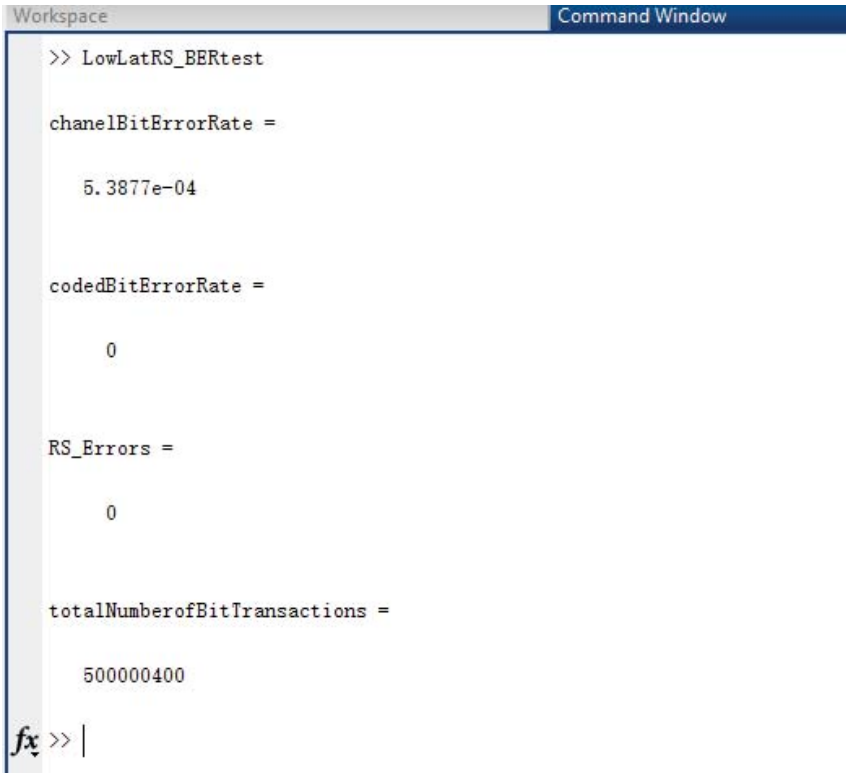


Figure 3–20: Results from the communication test-bench for BER performance.

3.4 Solutions to Arbitrary k and r

The low-latency RS coding algorithm needs to be modified when at least one of k and r is not divisible by p. In [17], Dr. Shokrollahi has provided the related information. This section sums up the solutions for different cases.

3.4.1 $k \neq 0$ mod p, $r = 0$ mod p

In this case, conceptual zeros can be added to the message-words to enforce k divisible by p. Specifically, let $k = c \bmod p$, and the message-words are $[d_{k-1}, d_{k-2}, \cdots, d_0]$. Then $(p-c)$ conceptual zeros are added to the end of message-words to generate $[0_{p-c-1}, \cdots, 0_1, 0_0, d_{k-1}, d_{k-2}, \cdot, d_0]$. With this adjusted message-words, we can simply encode them as before. Certainly, the conceptual zeros added are not transmitted after encoding process. As for decoding, conceptual zeros are added to the end of received-words again and followed with old decoding process illustrated in Section 3.2.2.

3.4.2 $r \neq 0$ **mod** p

When $r \neq 0 \bmod p$, the situation becomes more complicated. Let $k = c \bmod p$ and $r = d \bmod p$. First fix k by adding conceptual zeros if $k \neq 0 \bmod p$. Then produce the generator polynomial $g_i \equiv \Pi_{j \equiv i \bmod p}^{0 \leq j \leq r+p-d}(x - \alpha^j)$. Afterwards, calculate $(r+p-d)$ parity bits in the same way as before and drop the lowest coefficients of the resulting polynomials $h_0, h_1, \cdots, h_{p-d-1}$, that is, the last $(p-d)$ terms of the code-words. Specifically, $n = k+r+2p-c-d$ and the generated code-words are $[0_0, 0_1, \cdots, 0_{p-c-1}, c_{n-p-c-1}, c_{n-p-c-2}, \cdots, c_{p-d-1}, c_{p-d-2}, \cdots, c_0]$. While only $[c_{n-p-c-1}, c_{n-p-c-2}, \cdots, c_{p-d}]$ are transmitted.

As for the receiver side, denote the received-words as $[y_{n-p-c-1}, y_{n-p-c-2}, \cdots, y_{p-d}]$. Define $z = [z_{n-1}, z_{n-2}, \cdots, z_0]$ and set $z_i = (\Pi_{j=0}^{p-d-1}(\alpha_i - \rho^j) \cdot y_i)$ for $i = (p-d, p-d+1, \cdots, n-p+c-1)$ and $z_i = 0$ for others. Then take z as the "received-words" and follow the old decoding algorithm as before until reaching the error evaluation part. Recall the key equation $\gamma(x) = \sigma(x) \cdot s(x) \bmod x^{n-k}$ discussed in Chapter 2, where $\gamma(x)$ is the error-evaluator polynomial, $\sigma(x)$ is the error-locator polynomial and $s(x)$ is the syndrome polynomial. Since the syndromes are aquired from z instead of y. Then the error evaluation formula should be modified correspondingly as $e_j = (\Pi_{i=0}^{p-d-1}(\alpha_j - \rho^i)) \cdot \frac{\gamma(x)}{\sigma_o(x)}\Big|_{x=(\alpha_j)^{-1}}$. Certainly, we will ignore any errors with locations $j = (0, 1, \cdots, p-d-1)$ and $j = (n-p-c, \cdots, n-1)$.

CHAPTER 4
VHDL Implementation of Low-Latency RS(255,225)

4.1 VHDL Implementation of Galois Field Arithmetics

Galois field arithmetics are the basis for RS coding algorithms. As hardware has no awareness of any field elements, their arithmetics have to be defined with binary bits. Specific to $RS(255, 225)$, $GF(2^8)$ is assumed to be used. As it is a field of characteristic 2, subtraction is exactly equivalent to addition. Moreover, since division can always be taken as multiplication by a reciprocal, therefore only addition, multiplication and inversion needs to be implemented.

4.1.1 VHDL Implementation of a Galois Field Adder

Addition is the simplest arithmetic over Galois field to implement. Recall the polynomial representations of Galois field elements introduced in Chapter 2. Let $\mathbf{a}, \mathbf{b}, \mathbf{c}$ be three elements of $GF(2^8)$ and there is a relationship that $\mathbf{c} = \mathbf{a} + \mathbf{b}$. Assume their polynomial representations are

$$\mathbf{a} = a_7 \cdot \alpha^7 + a_6 \cdot \alpha^6 + a_5 \cdot \alpha^5 + a_4 \cdot \alpha^4 + a_3 \cdot \alpha^3 + a_2 \cdot \alpha^2 + a_1 \cdot \alpha^1 + a_0 \cdot \alpha^0$$

$$\mathbf{b} = b_7 \cdot \alpha^7 + b_6 \cdot \alpha^6 + b_5 \cdot \alpha^5 + b_4 \cdot \alpha^4 + b_3 \cdot \alpha^3 + b_2 \cdot \alpha^2 + b_1 \cdot \alpha^1 + b_0 \cdot \alpha^0$$

where all coefficients a_i, b_i are binary. Therefore,

$$\begin{aligned}
\mathbf{c} &= \mathbf{a} + \mathbf{b} \\
&= (a_7 + b_7) \cdot \alpha^7 + (a_6 + b_6) \cdot \alpha^6 + \cdots + (a_1 + b_1) \cdot \alpha^1 + (a_0 + b_0) \cdot \alpha^0 \\
&= c_7 \cdot \alpha^7 + c_6 \cdot \alpha^6 + \cdots + c_1 \cdot \alpha^1 + c_0 \cdot \alpha^0
\end{aligned}$$

As $GF(2^8)$ is a field of characteristic 2, so the coefficients $c_i = a_i + b_i$ are also binary and a truth table is obtained as Table 4–1. Obviously, this truth table can be implemented by the logic gate "XOR". Consequently, $c_i = a_i$ XOR b_i for $i = (0, 1, \cdots, 7)$ and $\mathbf{c} = \mathbf{a}$ XOR $\mathbf{b}$. Figure 4–1 shows VHDL codes for an addition function over $GF(2^8)$ and Figure 4–2 shows the corresponding circuit diagram.

a_i	b_i	$c_i = a_i + b_i$
0	0	0
0	1	1
1	0	1
1	1	0

Table 4–1: Addition with characteristic 2.

```
constant GFPower: integer:=8; --Galois Field power. is GF(2^8) here.
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);

--addition function for GF(2^8)
function add (b, c: in Galois_Field_element) return Galois_Field_element is
variable d: Galois_Field_element;

begin
d(0):=(b(0) xor c(0));
d(1):=(b(1) xor c(1));
d(2):=(b(2) xor c(2));
d(3):=(b(3) xor c(3));
d(4):=(b(4) xor c(4));
d(5):=(b(5) xor c(5));
d(6):=(b(6) xor c(6));
d(7):=(b(7) xor c(7));
return d;
end function add;
```

Figure 4–1: VHDL codes of an addition function over $GF(2^8)$.

4.1.2 VHDL Implementation of a Galois Field Multiplier

Hardware implementation of multiplication over a Galois field is relatively complicated. It starts with conventional multiplication between their polynomial representations and then the product polynomial is modulo the primitive

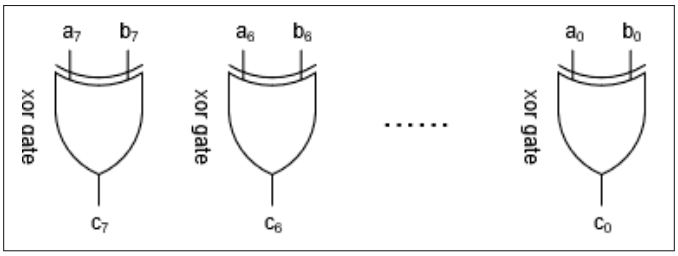


Figure 4–2: Implementation diagram of an adder over $GF(2^8)$.

irreducible polynomial. Mathematically, $w(\alpha) = u(\alpha) \cdot v(\alpha) \bmod P(\alpha)$ where $u(\alpha)$, $v(\alpha) \in GF(2^m)$ are the two operands, $w(\alpha)$ is the product, and $P(\alpha)$ is the primitive polynomial on which $GF(2^m)$ is built (details about primitive polynomials are stated in Chapter 2).

Example 24. It is easy to calculate $\alpha^3 \cdot \alpha^5 = \alpha^8 = \alpha^7 \cdot \alpha^1 = 1 \cdot \alpha^1 = \alpha^1$ over $GF(2^3)$ using the high-level method stated in Chapter 2. Now the task is to calculate the same equation using the method of hardware implementation. Look up into Table 2–3 and find that the polynomial representations for α^3 and α^5 are $\alpha^3 = (1 + \alpha)$ and $\alpha^5 = (1 + \alpha + \alpha^2)$, respectively. The corresponding primitive polynomial is $P(\alpha) = (1 + \alpha + \alpha^3)$. Then the multiplication should be performed as following:

$$\begin{aligned}
\alpha^3 \cdot \alpha^5 &= (1 + \alpha) \cdot (1 + \alpha + \alpha^2) \bmod (1 + \alpha + \alpha^3) \\
&= 1 + \alpha + \alpha^2 + \alpha + \alpha^2 + \alpha^3 \bmod (1 + \alpha + \alpha^3) \\
&= 1 + \alpha^3 \bmod (1 + \alpha + \alpha^3) \\
&= \alpha
\end{aligned}$$

Therefore, the results are consistent.

As there is no existing function for polynomial multiplication and modulo in hardware implementation, it is necessary to solve the equation to a general form with only arithmetics between binary coefficients of operands' polynomial

representations. To better explain the solving method, $GF(2^4)$ is taken as an example. Let $\mathbf{a}$, $\mathbf{b}$ be two elements of $GF(2^4)$. Table 2–1 shows the corresponding primitive polynomial is $P(\alpha) = \alpha^4 + \alpha + 1$. As any terms of the product of $\mathbf{a} \cdot \mathbf{b}$ with order less than 4 will be unchanged when modulo $P(\alpha)$, then

$$\begin{aligned}
\mathbf{w} &= \mathbf{a} \cdot \mathbf{b} \bmod P(\alpha) \\
&= (a_3\alpha^3 + \cdots + a_0\alpha^0) \cdot (b_3\alpha^3 + \cdots + b_0\alpha^0) \bmod (\alpha^4 + \alpha + 1) \\
&= (w_6\alpha^6 + w_5\alpha^5 + w_4\alpha^4) \bmod (\alpha^4 + \alpha + 1) + w_3\alpha^3 + w_2\alpha^2 + w_1\alpha^1 + w_0\alpha^0
\end{aligned}$$

where $w_i = \Sigma_{j+k=i}^{0 \le j,k \le 3} a_j b_k$. The multiplication between the coefficients a_j, b_k is implemented by a "AND" gate just like the conventional binary multiplication.

Recall that any element of a Galois field $GF(2^m)$ has a polynomial presentation with order smaller than that of its corresponding primitive polynomial $P(\alpha)$. Therefore, if the $[\alpha^{14}, \alpha^{13}, \cdots, \alpha^8]$ in $(w_{14}\alpha^{14} + w_{13}\alpha^{13} + \cdots + w_8\alpha^8)$ are converted to their polynomial representations with order smaller than that of $P(\alpha)$, then the modulo arithmetic part in the above equation will be gone. Up to that point, the only task left is to combine like terms. Concretely, by checking Table 2–2, we can find that:

$$\alpha^4 = \alpha^1 + \alpha^0$$

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

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

The above polynomials are substituted into the equation of $\mathbf{w}$. After combining like terms and replacing w_i with expressions of a_i and b_i, the result is

$$\mathbf{w} = (w_6\alpha^6 + w_5\alpha^5 + w_4\alpha^4) \bmod (\alpha^4 + \alpha + 1) + w_3\alpha^3 + w_2\alpha^2 + w_1\alpha^1 + w_0\alpha^0$$

$$= w_6(\alpha^3 + \alpha^2) + w_5(\alpha^2 + \alpha) + w_4(\alpha^1 + \alpha^0) + w_3\alpha^3 + w_2\alpha^2 + w_1\alpha^1 + w_0\alpha^0$$

$$= (w_6 + w_3)\alpha^3 + (w_6 + w_5 + w_2)\alpha^2 + (w_5 + w_4 + w_1)\alpha^1 + (w_4 + w_0)$$

$$= (a_3b_3 + a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)\alpha^3$$

$$+ (a_3b_3 + a_3b_2 + a_2b_3 + a_2b_0 + a_1b_1 + a_0b_2)\alpha^2$$

$$+ (a_3b_2 + a_2b_3 + a_3b_1 + a_2b_2 + a_1b_3 + a_1b_0 + a_0b_1)\alpha^1$$

$$+ (a_3b_1 + a_2b_2 + a_1b_3 + a_0b_0)\alpha^0$$

As mentioned before, the addition between binary coefficients a_i, b_i is implemented by a "XOR" gate and the multiplication is implemented by an "AND" gate. Therefore, the corresponding VHDL codes for multiplication over $GF(2^4)$ can be programmed as Figure 4–3.

```
constant GFPower: integer:=4;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);

function mul (b, c: in Galois_Field_element) return Galois_Field_element is variable w: Galois_Field_element;
   begin
   w(0):=(b(0) and c(0)) xor (b(1) and c(3)) xor(b(2)and c(2)) xor (b(3)and c(1));
   w(1):=(b(0) and c(1)) xor (b(1)and c(0)) xor (b(1)and c(3)) xor (b(2)and c(2)) xor
         (b(3)and c(1)) xor (b(2) and c(3)) xor (b(3) and c(2));
   w(2):=(b(0)and c(2)) xor (b(1)and c(1)) xor (b(2)and c(0)) xor (b(2) and c(3)) xor
         (b(3) and c(2)) xor (b(3) and c(3));
   w(3):=(b(0)and c(3)) xor (b(1)and c(2))xor (b(2) and c(1))xor(b(3)and c(0))xor(b(3)and c(3));
   return w;

end function mul;
```

Figure 4–3: VHDL codes of a multiplication function over $GF(2^4)$.

Specific to $GF(2^8)$ for $RS(255, 225)$, let $\mathbf{a}, \mathbf{b}$ be two elements. Table 2–1 shows that for $m = 8$ the primitive polynomial $P(\alpha) = \alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1$. Then

$$\mathbf{w} = \mathbf{a} \cdot \mathbf{b} \bmod P(\alpha)$$

$$= (a_7\alpha^7 + \cdots + a_0\alpha^0) \cdot (b_7\alpha^7 + \cdots + b_0\alpha^0) \bmod P(\alpha)$$

$$= (w_{14}\alpha^{14} + w_{13}\alpha^{13} + \cdots + w^8\alpha^8) \bmod P(\alpha) + w^7\alpha 7 + w^6\alpha^6 + \cdots + w^0\alpha^0$$

where $w_i = \Sigma_{j+k=i}^{0\le j,k\le 7} a_j b_k$.

As same as the last example, by checking Table 4–2, $[\alpha^{14}, \alpha^{13}, \cdots, \alpha^{8}]$ of $(w_{14}\alpha^{14} + w_{13}\alpha^{13} + \cdots + w_{8}\alpha^{8})$ can be converted into their polynomial representations with order smaller than that of $P(\alpha)$, then the modulo arithmetic part in the above equation will be gone. Notice that the full element table for $GF(2^8)$ is not given in this thesis due to its considerably long length. However, it can be accessed easily from the Internet and many books on ECC such as [20]. After conversion, the only task left is to combine like terms. Following the same multiplication process over $GF(2^4)$, the VHDL implementation of the multiplication over $GF(2^8)$ is constructed as shown in Figure 4–4.

Power Representation	Polynomial Representation	4-Tuple Representation
0	0	00000000(=0)
$\alpha^0(=1)$	1	00000001(=1)
α^1	α	00000010(=2)
α^2	α^2	00000100(=4)
α^3	α^3	00001000(=8)
α^4	α^4	00010000(=16)
α^5	α^5	00100000(=32)
α^6	α^6	01000000(=64)
α^7	α^7	10000000(=128)
α^8	$\alpha^4 + \alpha^3 + \alpha^2 + 1$	00011101(=29)
α^9	$\alpha^5 + \alpha^4 + \alpha^3 + \alpha$	00111010(=58)
α^{10}	$\alpha^6 + \alpha^5 + \alpha^4 + \alpha^2$	01110100(=116)
α^{11}	$\alpha^7 + \alpha^6 + \alpha^5 + \alpha^3$	11101000(=232)
α^{12}	$\alpha^7 + \alpha^6 + \alpha^3 + \alpha^2 + 1$	11001101(=205)
α^{13}	$\alpha^7 + \alpha^2 + \alpha + 1$	10000111(=135)
α^{14}	$\alpha^4 + \alpha + 1$	00010011(=19)

Table 4–2: Part of the elements table of $GF(2^8)$ with the primitive polynomial $P(X) = X^8 + X^4 + X^3 + X^2 + 1$.

```
function mul (b, c: in Galois_Field_element) return Galois_Field_element is variable w: Galois_Field_element;
--multiplication function for GF(2^4)
   begin
   w(0):=(b(0) and c(0)) xor (b(1) and c(7)) xor (b(2) and c(6)) xor (b(3) and c(5)) xor (b(4) and c(4)) xor
         (b(5) and c(3)) xor (b(5) and c(7)) xor (b(6) and c(2)) xor (b(6) and c(6)) xor (b(6) and c(7)) xor
         (b(7) and c(1)) xor (b(7) and c(5)) xor (b(7) and c(6)) xor (b(7) and c(7));

   w(1):=(b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(7)) xor (b(3) and c(6)) xor (b(4) and c(5)) xor
         (b(5) and c(4)) xor (b(6) and c(3)) xor (b(6) and c(7)) xor (b(7) and c(2)) xor (b(7) and c(6)) xor
         (b(7) and c(7));

   w(2):=(b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(7)) xor (b(2) and c(0)) xor (b(2) and c(6)) xor
         (b(3) and c(5)) xor (b(3) and c(7)) xor (b(4) and c(4)) xor (b(4) and c(6)) xor (b(5) and c(3)) xor
         (b(5) and c(5)) xor (b(5) and c(7)) xor (b(6) and c(2)) xor (b(6) and c(4)) xor (b(6) and c(6)) xor
         (b(6) and c(7)) xor (b(7) and c(1)) xor (b(7) and c(3)) xor (b(7) and c(5)) xor (b(7) and c(6));

   w(3):=(b(0) and c(3)) xor (b(1) and c(2)) xor (b(1) and c(7)) xor (b(2) and c(1)) xor (b(2) and c(6)) xor
         (b(2) and c(7)) xor (b(3) and c(0)) xor (b(3) and c(5)) xor (b(3) and c(6)) xor (b(4) and c(4)) xor
         (b(4) and c(5)) xor (b(4) and c(7)) xor (b(5) and c(3)) xor (b(5) and c(4)) xor (b(5) and c(6)) xor
         (b(5) and c(7)) xor (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
         (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(4)) xor (b(7) and c(5));

   w(4):=(b(0) and c(4)) xor (b(1) and c(3)) xor (b(1) and c(7)) xor (b(2) and c(2)) xor (b(2) and c(6)) xor
         (b(2) and c(7)) xor (b(3) and c(1)) xor (b(3) and c(5)) xor (b(3) and c(6)) xor (b(3) and c(7)) xor
         (b(4) and c(0)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(4) and c(6)) xor (b(5) and c(3)) xor
         (b(5) and c(4)) xor (b(5) and c(5)) xor (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(4)) xor
         (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor (b(7) and c(7));

   w(5):=(b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(2) and c(7)) xor (b(3) and c(2)) xor
         (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(1)) xor (b(4) and c(5)) xor (b(4) and c(6)) xor
         (b(4) and c(7)) xor (b(5) and c(0)) xor (b(5) and c(4)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor
         (b(6) and c(3)) xor (b(6) and c(4)) xor (b(6) and c(5)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor
         (b(7) and c(4));

   w(6):=(b(0) and c(6)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor (b(3) and c(7)) xor
         (b(4) and c(2)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor (b(5) and c(1)) xor (b(5) and c(5)) xor
         (b(5) and c(6)) xor (b(5) and c(7)) xor (b(6) and c(0)) xor (b(6) and c(4)) xor (b(6) and c(5)) xor
         (b(6) and c(6)) xor (b(7) and c(3)) xor (b(7) and c(4)) xor (b(7) and c(5));

   w(7):=(b(0) and c(7)) xor (b(1) and c(6)) xor (b(2) and c(5)) xor (b(3) and c(4)) xor (b(4) and c(3)) xor
         (b(4) and c(7)) xor (b(5) and c(2)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor (b(6) and c(1)) xor
         (b(6) and c(5)) xor (b(6) and c(6)) xor (b(6) and c(7)) xor (b(7) and c(0)) xor (b(7) and c(4)) xor
         (b(7) and c(5)) xor (b(7) and c(6));
   return w;
end function mul;
```

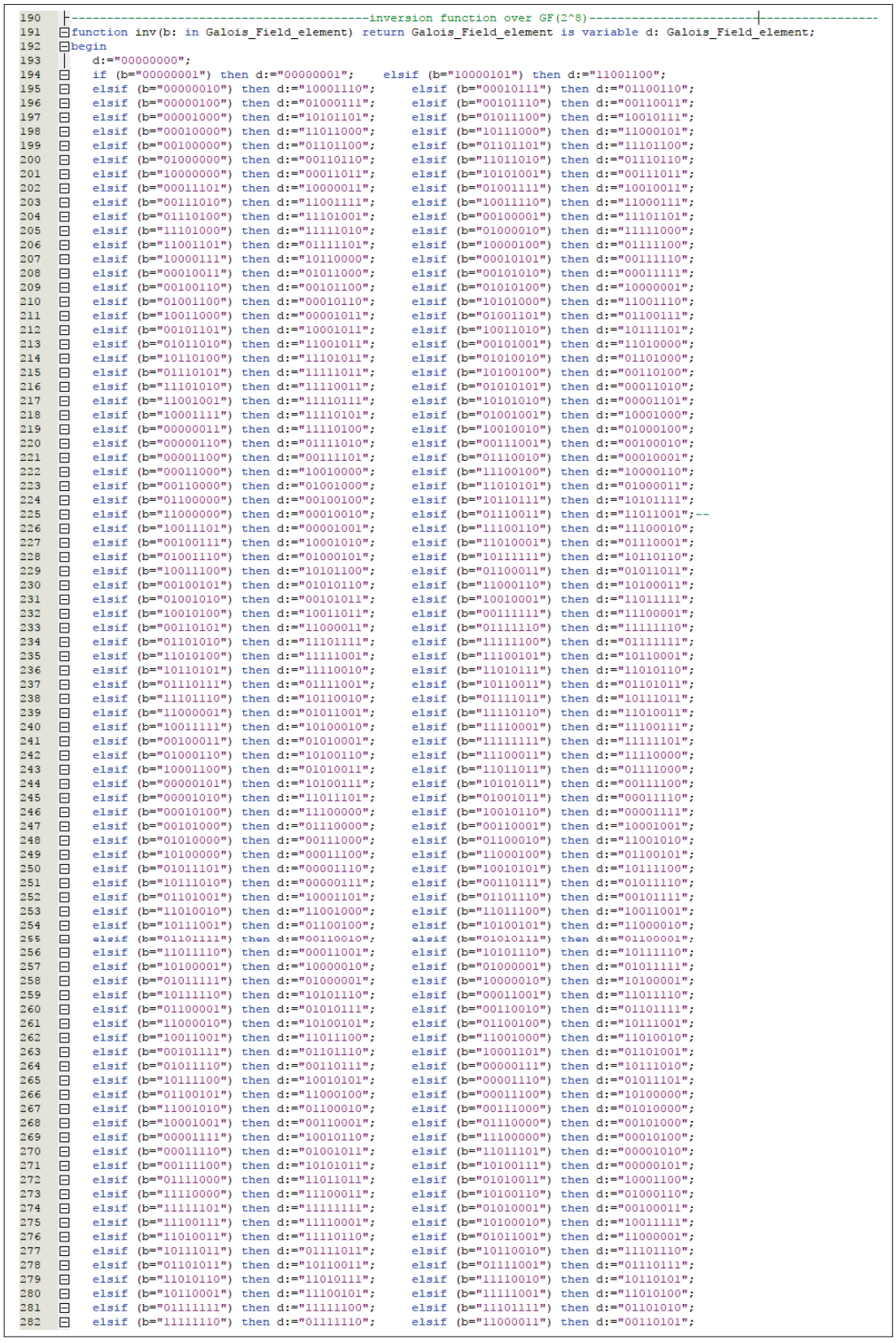
Figure 4–4: VHDL codes of a multiplication function over $GF(2^8)$.

4.1.3 VHDL Implementation of a Galois Field Inverter

Inversion over a Galois field is significantly difficult to implement on hardware. For fields with small to medium size, it is much simpler and faster to use a look-up table (LUT). Figure 4–5 and 4–6 show the inversion function for $GF(2^8)$.

4.2 Encoder Implementation

Prior to the discussion of the novel low-latency RS encoder, implementation of a conventional RS encoder should be roughly introduced. Recall the algorithm stated in Chapter 2. To a specific encoder, the generator polynomial can be pre-calculated and the only step to implement is $b(X) = (X^{n-k} \cdot m(X)) \mod g(X)$. This equation can be solved by a linear feedback shift register (LFSR) which is shown in Figure 4–7. The rectangular box b_i represents the



```
190   ├-----------------------------------------inversion function over GF(2^8)-----------------------┼----------------
191   ⊟function inv(b: in Galois_Field_element) return Galois_Field_element is variable d: Galois_Field_element;
192   ⊟begin
193   │    d:="00000000";
194   ⊟    if (b="00000001") then d:="00000001";    elsif (b="10000101") then d:="11001100";
195   ⊟    elsif (b="00000010") then d:="10001110";     elsif (b="00010111") then d:="01100110";
196   ⊟    elsif (b="00000100") then d:="01000111";     elsif (b="00101110") then d:="00110011";
197   ⊟    elsif (b="00001000") then d:="10101101";     elsif (b="01011100") then d:="10010111";
198   ⊟    elsif (b="00010000") then d:="11011000";     elsif (b="10111000") then d:="11000101";
199   ⊟    elsif (b="00100000") then d:="01101100";     elsif (b="01101101") then d:="11101100";
200   ⊟    elsif (b="01000000") then d:="00110110";     elsif (b="11011010") then d:="01110110";
201   ⊟    elsif (b="10000000") then d:="00011011";     elsif (b="10101001") then d:="00111011";
202   ⊟    elsif (b="00011101") then d:="10000011";     elsif (b="01001111") then d:="10010011";
203   ⊟    elsif (b="00111010") then d:="11001111";     elsif (b="10011110") then d:="11000111";
204   ⊟    elsif (b="01110100") then d:="11101001";     elsif (b="00100001") then d:="11101101";
205   ⊟    elsif (b="11101000") then d:="11111010";     elsif (b="01000010") then d:="11111000";
206   ⊟    elsif (b="11001101") then d:="01111101";     elsif (b="10000100") then d:="01111100";
207   ⊟    elsif (b="10000111") then d:="10110000";     elsif (b="00010101") then d:="00111110";
208   ⊟    elsif (b="00010011") then d:="01011000";     elsif (b="00101010") then d:="00011111";
209   ⊟    elsif (b="00100110") then d:="00101100";     elsif (b="01010100") then d:="10000001";
210   ⊟    elsif (b="01001100") then d:="00010110";     elsif (b="10101000") then d:="11001110";
211   ⊟    elsif (b="10011000") then d:="00001011";     elsif (b="01001101") then d:="01100111";
212   ⊟    elsif (b="00101101") then d:="10001011";     elsif (b="10011010") then d:="10111101";
213   ⊟    elsif (b="01011010") then d:="11001011";     elsif (b="00101001") then d:="11010000";
214   ⊟    elsif (b="10110100") then d:="11101011";     elsif (b="01010010") then d:="01101000";
215   ⊟    elsif (b="01110101") then d:="11111011";     elsif (b="10100100") then d:="00110100";
216   ⊟    elsif (b="11101010") then d:="11110011";     elsif (b="01010101") then d:="00011010";
217   ⊟    elsif (b="11001001") then d:="11110111";     elsif (b="10101010") then d:="00001101";
218   ⊟    elsif (b="10001111") then d:="11110101";     elsif (b="01001001") then d:="10001000";
219   ⊟    elsif (b="00000011") then d:="11110100";     elsif (b="10010010") then d:="01000100";
220   ⊟    elsif (b="00000110") then d:="01111010";     elsif (b="00111001") then d:="00100010";
221   ⊟    elsif (b="00001100") then d:="00111101";     elsif (b="01110010") then d:="00010001";
222   ⊟    elsif (b="00011000") then d:="10010000";     elsif (b="11100100") then d:="10000110";
223   ⊟    elsif (b="00110000") then d:="01001000";     elsif (b="11010101") then d:="01000011";
224   ⊟    elsif (b="01100000") then d:="00100100";     elsif (b="10110111") then d:="10101111";
225   ⊟    elsif (b="11000000") then d:="00010010";     elsif (b="01110011") then d:="11011001";--
226   ⊟    elsif (b="10011101") then d:="00001001";     elsif (b="11100110") then d:="11100010";
227   ⊟    elsif (b="00100111") then d:="10001010";     elsif (b="11010001") then d:="01110001";
228   ⊟    elsif (b="01001110") then d:="01000101";     elsif (b="10111111") then d:="10110110";
229   ⊟    elsif (b="10011100") then d:="10101100";     elsif (b="01100011") then d:="01011011";
230   ⊟    elsif (b="00100101") then d:="01010110";     elsif (b="11000110") then d:="10100011";
231   ⊟    elsif (b="01001010") then d:="00101011";     elsif (b="10010001") then d:="11011111";
232   ⊟    elsif (b="10010100") then d:="10011011";     elsif (b="00111111") then d:="11100001";
233   ⊟    elsif (b="00110101") then d:="11000011";     elsif (b="01111110") then d:="11111110";
234   ⊟    elsif (b="01101010") then d:="11101111";     elsif (b="11111100") then d:="01111111";
235   ⊟    elsif (b="11010100") then d:="11111001";     elsif (b="11100101") then d:="10110001";
236   ⊟    elsif (b="10110101") then d:="11110010";     elsif (b="11010111") then d:="11010110";
237   ⊟    elsif (b="01110111") then d:="01111001";     elsif (b="10110011") then d:="01101011";
238   ⊟    elsif (b="11101110") then d:="10110010";     elsif (b="01111011") then d:="10111011";
239   ⊟    elsif (b="11000001") then d:="01011001";     elsif (b="11110110") then d:="11010011";
240   ⊟    elsif (b="10011111") then d:="10100010";     elsif (b="11110001") then d:="11100111";
241   ⊟    elsif (b="00100011") then d:="01010001";     elsif (b="11111111") then d:="11111101";
242   ⊟    elsif (b="01000110") then d:="10100110";     elsif (b="11100011") then d:="11110000";
243   ⊟    elsif (b="10001100") then d:="01010011";     elsif (b="11011011") then d:="01111000";
244   ⊟    elsif (b="00000101") then d:="10100111";     elsif (b="10101011") then d:="00111100";
245   ⊟    elsif (b="00001010") then d:="11011101";     elsif (b="01001011") then d:="00011110";
246   ⊟    elsif (b="00010100") then d:="11100000";     elsif (b="10010110") then d:="00001111";
247   ⊟    elsif (b="00101000") then d:="01110000";     elsif (b="00110001") then d:="10001001";
248   ⊟    elsif (b="01010000") then d:="00111000";     elsif (b="01100010") then d:="11001010";
249   ⊟    elsif (b="10100000") then d:="00011100";     elsif (b="11000100") then d:="01100101";
250   ⊟    elsif (b="01011101") then d:="00001110";     elsif (b="10010101") then d:="10111100";
251   ⊟    elsif (b="10111010") then d:="00000111";     elsif (b="00110111") then d:="01011110";
252   ⊟    elsif (b="01101001") then d:="10001101";     elsif (b="01101110") then d:="00101111";
253   ⊟    elsif (b="11010010") then d:="11001000";     elsif (b="11011100") then d:="10011001";
254   ⊟    elsif (b="10111001") then d:="01100100";     elsif (b="10100101") then d:="11000010";
255   ⊟    elsif (b="01101111") then d:="00110010";     elsif (b="01010111") then d:="01100001";
256   ⊟    elsif (b="11011110") then d:="00011001";     elsif (b="10101110") then d:="10111110";
257   ⊟    elsif (b="10100001") then d:="10000010";     elsif (b="01000001") then d:="01011111";
258   ⊟    elsif (b="01011111") then d:="01000001";     elsif (b="10000010") then d:="10100001";
259   ⊟    elsif (b="10111110") then d:="10101110";     elsif (b="00011001") then d:="11011110";
260   ⊟    elsif (b="01100001") then d:="01010111";     elsif (b="00110010") then d:="01101111";
261   ⊟    elsif (b="11000010") then d:="10100101";     elsif (b="01100100") then d:="10111001";
262   ⊟    elsif (b="10011001") then d:="11011100";     elsif (b="11001000") then d:="11010010";
263   ⊟    elsif (b="00101111") then d:="01101110";     elsif (b="10001101") then d:="01101001";
264   ⊟    elsif (b="01011110") then d:="00110111";     elsif (b="00000111") then d:="10111010";
265   ⊟    elsif (b="10111100") then d:="10010101";     elsif (b="00001110") then d:="01011101";
266   ⊟    elsif (b="01100101") then d:="11000100";     elsif (b="00011100") then d:="10100000";
267   ⊟    elsif (b="11001010") then d:="01100010";     elsif (b="00111000") then d:="01010000";
268   ⊟    elsif (b="10001001") then d:="00110001";     elsif (b="01110000") then d:="00101000";
269   ⊟    elsif (b="00001111") then d:="10010110";     elsif (b="11100000") then d:="00010100";
270   ⊟    elsif (b="00011110") then d:="01001011";     elsif (b="11011101") then d:="00001010";
271   ⊟    elsif (b="00111100") then d:="10101011";     elsif (b="10100111") then d:="00000101";
272   ⊟    elsif (b="01111000") then d:="11011011";     elsif (b="01010011") then d:="10001100";
273   ⊟    elsif (b="11110000") then d:="11100011";     elsif (b="10100110") then d:="01000110";
274   ⊟    elsif (b="11111101") then d:="11111111";     elsif (b="01010001") then d:="00100011";
275   ⊟    elsif (b="11100111") then d:="11110001";     elsif (b="10100010") then d:="10011111";
276   ⊟    elsif (b="11010011") then d:="11110110";     elsif (b="01011001") then d:="11000001";
277   ⊟    elsif (b="10111011") then d:="01111011";     elsif (b="10110010") then d:="11101110";
278   ⊟    elsif (b="01101011") then d:="10110011";     elsif (b="01111001") then d:="01110111";
279   ⊟    elsif (b="11010110") then d:="11010111";     elsif (b="11110010") then d:="10110101";
280   ⊟    elsif (b="10110001") then d:="11100101";     elsif (b="11111001") then d:="11010100";
281   ⊟    elsif (b="01111111") then d:="11111100";     elsif (b="11101111") then d:="01101010";
282   ⊟    elsif (b="11111110") then d:="01111110";     elsif (b="11000011") then d:="00110101";
```

Figure 4–5: VHDL codes of a inversion function over $GF(2^8)$: Part 1.

```
283     elsif (b="11100001") then d:="00111111";     elsif (b="10011011") then d:="10010100";
284     elsif (b="11011111") then d:="10010001";     elsif (b="00101011") then d:="01001010";
285     elsif (b="10100011") then d:="11000110";     elsif (b="01010110") then d:="00100101";
286     elsif (b="01011011") then d:="01100011";     elsif (b="10101100") then d:="10011100";
287     elsif (b="10110110") then d:="10111111";     elsif (b="01000101") then d:="01001110";
288     elsif (b="01110001") then d:="11010001";     elsif (b="10001010") then d:="00100111";
289     elsif (b="11100010") then d:="11100110";     elsif (b="00001001") then d:="10011101";
290     elsif (b="11011001") then d:="01110011";     elsif (b="00010010") then d:="11000000";
291     elsif (b="10101111") then d:="10110111";     elsif (b="00100100") then d:="01100000";
292     elsif (b="01000011") then d:="11010101";     elsif (b="01001000") then d:="00110000";
293     elsif (b="10000110") then d:="11100100";     elsif (b="10010000") then d:="00011000";
294     elsif (b="00010001") then d:="01110010";     elsif (b="00111101") then d:="00001100";
295     elsif (b="00100010") then d:="00111001";     elsif (b="01111010") then d:="00000110";
296     elsif (b="01000100") then d:="10010010";     elsif (b="11110100") then d:="00000011";
297     elsif (b="10001000") then d:="01001001";     elsif (b="11110101") then d:="10001111";
298     elsif (b="00001101") then d:="10101010";     elsif (b="11110111") then d:="11001001";
299     elsif (b="00011010") then d:="01010101";     elsif (b="11110011") then d:="11101010";
300     elsif (b="00110100") then d:="10100100";     elsif (b="11111011") then d:="01110101";
301     elsif (b="01101000") then d:="01010010";     elsif (b="11101011") then d:="10110100";
302     elsif (b="11010000") then d:="00101001";     elsif (b="11001011") then d:="01011010";
303     elsif (b="10111101") then d:="10011010";     elsif (b="10001011") then d:="00101101";
304     elsif (b="01100111") then d:="01001101";     elsif (b="00001011") then d:="10011000";
305     elsif (b="11001110") then d:="10101000";     elsif (b="00010110") then d:="01001100";
306     elsif (b="10000001") then d:="01010100";     elsif (b="00101100") then d:="00100110";
307     elsif (b="00011111") then d:="00101010";     elsif (b="01011000") then d:="00010011";
308     elsif (b="00111110") then d:="00010101";     elsif (b="10110000") then d:="10000111";
309     elsif (b="01111100") then d:="10000100";     elsif (b="01111101") then d:="11001101";
310     elsif (b="11111000") then d:="01000010";     elsif (b="11111010") then d:="11101000";
311     elsif (b="11101101") then d:="00100001";     elsif (b="11101001") then d:="01110100";
312     elsif (b="11000111") then d:="10011110";     elsif (b="11001111") then d:="00111010";
313     elsif (b="10010011") then d:="01001111";     elsif (b="10000011") then d:="00011101";
314     elsif (b="00111011") then d:="10101001";     elsif (b="00011011") then d:="10000000";
315     elsif (b="01110110") then d:="11011010";     elsif (b="00110110") then d:="01000000";
316     elsif (b="11101100") then d:="01101101";     elsif (b="01101100") then d:="00100000";
317     elsif (b="11000101") then d:="10111000";     elsif (b="11011000") then d:="00010000";
318     elsif (b="10010111") then d:="01011100";     elsif (b="10101101") then d:="00001000";
319     elsif (b="00110011") then d:="00101110";     elsif (b="01000111") then d:="00000100";
320     elsif (b="01100110") then d:="00010111";     elsif (b="10001110") then d:="00000010";
321     elsif (b="11001100") then d:="10000101";
322     end if;
323     return d;
324   end inv;
```

Figure 4–6: VHDL codes of a inversion function over $GF(2^8)$: Part 2.

parity-bits register. The encoding progress starts with the Switch 1 connected to the adder and Switch 2 connected to the LFSR input. In the first k clock cycles, the coefficients of $m(x)$ enter the circuit one coefficient per clock cycle and the highest-order coefficient m^{k-1} enters first. During these clock cycles, the parity-bits are being calculated, and the output of LFSR is the same as the input. After the calculation finished at the k-th clock cycle, the coefficients of $b(x)$ of the previous formula are stored in each shift register b_i. At this point, Switch 1 turns to "0" input and Switch 2 turns to the register b_{n-k-1} so that the LFSR shifts the coefficients of $b(x)$ out one coefficient per clock cycle. As there are $n-k$ registers, this procedure goes on for $n-k$ clocks, and it leads to the end of the encoding process.

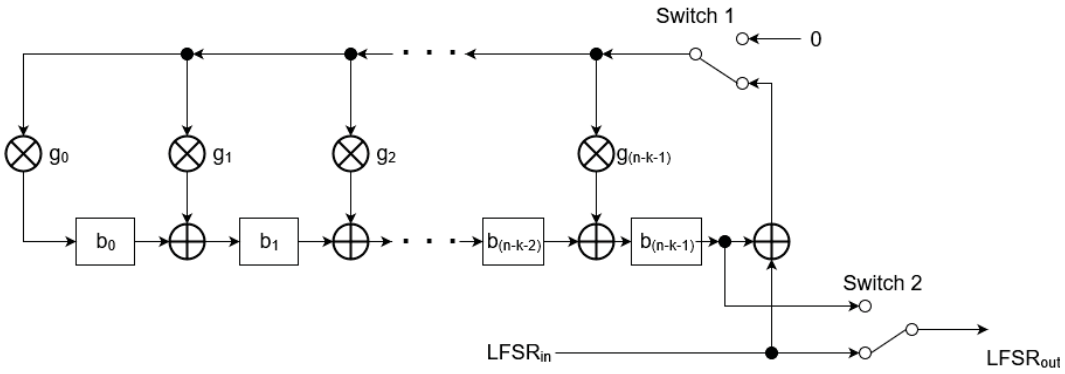


Figure 4–7: Structure of a linear shift back register (LFSR).

In this research, three low-latency $RS(255, 225)$ encoders with different speed-up coefficients p have been implemented in VHDL and tested. The implementation logic follows the structure shown in Figure 3–1. Compared to a conventional RS encoder, a low-latency RS encoder uses multiple shorter LFSRs which can run in parallel in its Step 4. However, there is a cost which is message-words have to be pre-processed before entering the LFSRs and the outputs from LFSRs have to be post-processed to generate the real parity bits.

4.2.1 Implementations with Speed-Up Coefficient $p = 3$

Figure 4–8 shows a low-latency $RS(255, 225)$ encoder with speed-up coefficient $p = 3$. Each message-word is an 8-bit vector, and there are $1800 (= 8 \cdot 225)$ bits in total in a set of message-words. At each clock cycle, three message-words enter the circuit and are split as three inputs of the DFT. The DFT then generates three outputs and send them into three LFSRs. Notice that each LFSR has a distinct generator polynomial. The outputs from LFSRs are applied with an inverse Fourier transform in the IDFT to produce the real parity-bits. With access to all computed parity bits and the delayed message-word input, the output controller gives the final code-words by three words at each clock. In the following paragraphs, each part of the implementation is

discussed in details, and more comments are shown in the screenshots of their VHDL codes.

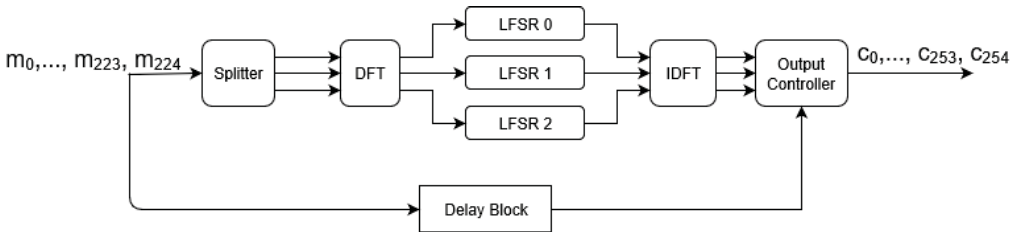


Figure 4–8: Low-latency $RS(255, 225)$ encoder with speed-up coefficient $p = 3$.

Set-Up. As shown in Figure 4–9, there are four input ports of the encoder: a global reset denoted as "**reset_n**", a clock signal denoted as "**clk**", a port for input data denoted as "**data_in**" and a signal denoted as "**input_strobe**" which is to activate the encoding process. Notice that the length of "**data_in**" is 24 bits because three message-words enter the circuit at each clock cycle. There are three outputs: a signal denoted as "**output_strobe**" that suggests if the first three code-words are ready, a signal denoted as "**enc_done_p**" that indicates if the last three code-words have been sent out and the code-word output also of 24-bit length denoted as "**data_out_p**". Figure 4–10 shows the declaration of the LFSR component and other signals. The signals with suffix "**_p**" are all register signals and their counterparts which are without the suffix "**_p**" are combinational logic signals used for register update.

Defining Functions. The adder and multiplier over $GF(2^8)$ have been discussed in Section 4.1 and the corresponding VHDL codes are shown in Figure 4–1 and 4–4, respectively. Figure 4–11 shows the definition of DFT and IDFT functions. It follows the formula $DFT(M)_i \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} M_j$ and $IDFT(f)_i = \Sigma_{j=0}^{p-1} \rho^{-j \cdot i} f_j$. Notice that "**q**" refers to p-th root of unity and "**q2**" refers to q^2.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity LowLatRSenc_n255k225_p3 is

port (reset_n: in std_logic; --global reset input
      data_in: in std_logic_vector(23 downto 0); --3*8=24
      --the input is a set of bits in sequence.
      --MSB(the highest-order coefficients of data polynomial) enters first.
      clk: in std_logic;  --clock signal
      input_strobe: in std_logic; --a signal which activates encoding procedure.
      output_strobe: out std_logic; --a signal that suggests the output is ready to be sent out
      enc_done_p: out std_logic;--a signal which indicates the encoding process has been done.
      data_out_p:out std_logic_vector(23 downto 0)); --3*8=24
end;
```

Figure 4–9: Ports Declaration of RS encoder with $p = 3$.

```
architecture RTL of LowLatRSenc_n255k225_p3 is

COMPONENT LowLatRSenc_LFSR_n85k75
--is a linear feedback shift register which is actually the same as conventional narrow-sense RS encoder.
--we need 3 LFSR in this low-latency algorithm because p=3.
   port (reset_n: in std_logic;
    data_in: in std_logic_vector (7 downto 0);
    clk: in std_logic;
    input_strobe: in std_logic;
    g0: in std_logic_vector (7 downto 0);
    g1: in std_logic_vector (7 downto 0);
    g2: in std_logic_vector (7 downto 0);
    g3: in std_logic_vector (7 downto 0);
    g4: in std_logic_vector (7 downto 0);
    g5: in std_logic_vector (7 downto 0);
    g6: in std_logic_vector (7 downto 0);
    g7: in std_logic_vector (7 downto 0);
    g8: in std_logic_vector (7 downto 0);
    g9: in std_logic_vector (7 downto 0);
    --the highest-order coefficient of generator polynomial is ommited because it's always 1.
    data_out_p: out std_logic_vector (7 downto 0));
END COMPONENT ;
----------basic parameters---------------------------------
constant N: integer:=255; --code-word length
constant K: integer:=225; --message-word length
constant R: integer:=30; --parity-bits length =255-225=30
constant m: integer:=85; --m=N/p=255/3=85
constant t: integer:=75; --t=K/p=225/3=75
constant L: integer:=10; --L=R/p=30/3=10
constant p: integer:=3; --speed-up coefficient
constant GFPower: integer:=8; --Galois Field power to define GF(2^8).
----------pre-define useful types-------------------------------------------
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0); --type for elements in GF(2^8)
type state_type is (Idle,Calculation,XferParity,WrapUp);--(Finite state machine)
type FTout_type is array (0 to (p-1)) of Galois_Field_element;
type LFSRout_type is array (0 to (p-1)) of Galois_Field_element;
----------constants-----------------------------------------------------------
constant zero: Galois_Field_element :="00000000";
constant one: Galois_Field_element :="00000001";
constant q: Galois_Field_element:="11010110"; --set the 3rd root of unity
constant q2: Galois_Field_element:="11010111"; --square of the 3rd root of unity
----------data vectors--------------------------------------------------------
signal DFTout: FTout_type; --output of the DFT, that is, input of each LFSR
signal IDFTout: FTout_type; --output of the IDFT, that is, parity-bits h0,h1,h2
signal LFSRout: LFSRout_type; --output of the LFSR
signal data_inbuf,data_inbuf_p:std_logic_vector((p*GFPower-1) downto 0); --delayed data_in
signal data_out:std_logic_vector(p*GFPower-1 downto 0);--combinational logic signal used to update data_out_p
----------control signals-----------------------------------------------------
signal state,state_p:state_type;--for Finite state machine
signal counter,counter_p:integer:=0;
signal countEn,countEn_p:std_logic; --enable counting. counter only count when countEn='1'
signal countRst,countRst_p:std_logic;--reset counting. counter=0 whenever countRSt='1'
signal XferEn,XferEn_p:std_logic;--enable transferring parity bits out
signal enc_done:std_logic;--combinational logic signal used to update enc_done_p
```

Figure 4–10: Signal declaration of RS encoder with $p = 3$.

Component Instantiations. As Figure 4–12 shows, with $p = 3$, three LFSRs are used. They all share a same global clock and reset. The "**input_strobe**" signal activate all the three at the same time. The data input of each LFSR

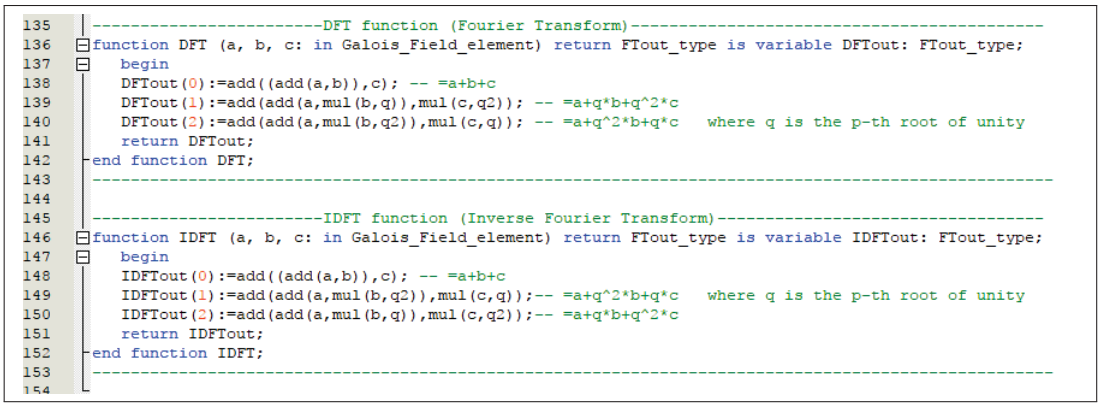


Figure 4–11: DFT and IDFT functions of RS encoder with $p = 3$.

is one of the three output from the DFT. Each output from them is a vector denoted as "**LFSRout**". Notice that the generator polynomial coefficients $[g_0, g_1, g_2, \cdots, g_9]$ are different for each LFSR, because in the algorithm, there is p distinct generator polynomials. With the formula given in Chapter 3, each generator polynomial is pre-calculated. The features and inside structure of LFSR component are explained with details later.

Register Update. Figure 4–13 shows the VHDL codes for register update. At each rising edge of the clock, the register signals are updated to their combinational logic counterparts. When the global reset is active at a clock's rising edge, all register signals go to zero.

Finite State Machine. A finite state machine (FSM) is utilized to control the overall process (Figure 4–14). There are four state: "**Idle**", "**Calculation**", "**XferParity**" and "**WrapUp**". In state "**Idle**", all the control signals are set to zero except "**countRst**" which resets the counter when active. When "**input_strobe**" becomes to 1, it means the encoding process is activated. Then "**countEn**" turns to 1 which allows the counter to do increment and the state goes to "**Calculation**", which means LFSRs start their computing process. When "**counter_p**" reaches t where $t = k/p$ is the component length of message-words, "**xferEn**" becomes 1 and the state enters

```
154  begin
155    LFSR0:  LowLatRSenc_LFSR_n85k75
156  port map(reset_n=>reset_n,
157      data_in=>DFTout(0),
158      clk=>clk,
159      input_strobe=>input_strobe,
160      --all generator coefficients are pre-calculated by MATLAB
161      g0=>"10101001", g1=>"01010011", g2=>"11100010",
162      g3=>"01010001", g4=>"11001010", g5=>"01110100",
163      g6=>"10011111", g7=>"00111001", g8=>"00010011",
164      g9=>"01000011",
165      data_out_p=>LFSRout(0));
166
167    LFSR1:  LowLatRSenc_LFSR_n85k75
168  port map(reset_n=>reset_n,
169      data_in=>DFTout(1),
170      clk=>clk,
171      input_strobe=>input_strobe,
172      g0=>"01001101", g1=>"11101111", g2=>"10001000",
173      g3=>"11101111", g4=>"11110000", g5=>"00100110",
174      g6=>"00000101", g7=>"11010101", g8=>"01001100",
175      g9=>"10000110",
176      data_out_p=>LFSRout(1));
177
178    LFSR2:  LowLatRSenc_LFSR_n85k75
179  port map(reset_n=>reset_n,
180      data_in=>DFTout(2),
181      clk=>clk,
182      input_strobe=>input_strobe,
183      g0=>"01110010", g1=>"00010010", g2=>"11001110",
184      g3=>"10001010", g4=>"11010110", g5=>"10110100",
185      g6=>"01010000", g7=>"11100110", g8=>"00101101",
186      g9=>"00010001",
187      data_out_p=>LFSRout(2));
188
```

Figure 4–12: LFSR component instantiations of RS encoder with $p = 3$.

```
211  process(clk) --register update
212    begin
213      if(clk'event and clk='1') then --only update at the rising edge of clock
214        if (reset_n='1') then --reset function
215          data_inbuf_p<=(others=>'0');
216          data_out_p<=(others=>'0');
217          counter_p<=0;
218          state_p<=Idle;
219          countRst_p<='1';
220          countEn_p<='0';
221          xferEn_p<='0';
222          enc_done_p<='0';
223        else
224          data_inbuf_p<=data_inbuf;
225          data_out_p<=data_out;
226          counter_p<=counter;
227          state_p<=state;
228          countRst_p<=countRst;
229          countEn_p<=countEn;
230          xferEn_p<=xferEn;
231          enc_done_p<=enc_done;
232        end if;
233      end if;
234  end process;
```

Figure 4–13: Register update of RS encoder with $p = 3$.

"**XferParity**", which means the intermediate parity bits calculated by LFSRs are ready to shift out. When "**counter_p**" reaches m where $m = n/p$ is the component length of code-words, "**countRst**" goes to 1 and "**enc_done**" is activated to suggest that the encoding progress has been done.

```
215   ----------------------------Finite state machine--------------------------------------
216   process(state_p,input_strobe,counter_p)
217   begin
218      case state_p is
219         when Idle =>
220            if(input_strobe='1') then --input_strobe initialates encoding process
221               countRst<='0';
222               countEn<='1';
223               xferEn<='0';
224               enc_done<='0';
225               state<=Calculation;
226            else
227               countRst<='1';
228               countEn<='0';
229               xferEn<='0';
230               enc_done<='0';
231               state<=Idle;
232            end if;
233
234         when Calculation => --calculating parity bits.
235            if (counter_p=t) then
236               countRst<='0';
237               countEn<='1';
238               xferEn<='1';
239               enc_done<='0';
240               state<=XferParity;
241            else
242               countRst<='0';
243               countEn<='1';
244               xferEn<='0';
245               enc_done<='0';
246               state<=Calculation;
247            end if;
248
249         when XferParity => --transfer parity bits out of LFSR
250            if(counter_p=m) then
251                  --when counter=m, all parity bits are transferred out. and encoding is finished
252               countRst<='1';
253               countEn<='0';
254               XferEn<='0';
255               enc_done<='1';
256               state<=WrapUp;
257            else
258               countRst<='0';
259               countEn<='1';
260               xferEn<='1';
261               enc_done<='0';
262               state<=XferParity;
263            end if;
264
265         when WrapUp =>
266               --this state is to de-active enc_done signal. so that enc_done is only high for one single clock
267            countRst<='0';
268            countEn<='0';
269            XferEn<='0';
270            enc_done<='0';
271            state<=Idle;
272
273         when others =>
274            countRst<='1';
275            countEn<='0';
276            xferEn<='0';
277            enc_done<='0';
278            state<=Idle;
279      end case;
280   end process;
281
```

Figure 4–14: Finite state machine of RS encoder with $p = 3$.

Counter Implementation. Figure 4–15 shows the counter behavior. When "**input_strobe**" or "**countRst_p**" is activated, the counter is compelled to be 0. The counter only counts when "**countEn_p**" is 1.

```
283  process(input_strobe,countRst_p,countEn_p,counter_p)
284  begin
285     if(input_strobe='1' or countRst_p='1') then
286     --reset the counter both at the beginning and the end of encoding process
287        counter<=0;
288     elsif (countEn_p='1') then
289        counter<=counter_p+1;
290     else
291        counter<=0;
292     end if;
293  end process;
```

Figure 4–15: Counter behavior of RS encoder with $p = 3$.

Preprocessing before LFSR. Figure 4–16 show the preprocessing before the LFSR. Besides creating a copy of delayed inputs, the other task is to apply Fourier transform on three components of the input message-words. The outputs from the DFT are sent to three LFSRs. "**p*GFPower**" refers to the total number of bits in "**data_in**". Specific to this encoder with $p = 3$, every input at each clock cycle has three message-words and thus has 24 bits binary data. To split the input into $p = 3$ components, just set the corresponding index correctly.

```
295  data_inbuf<=data_in;  --copy of data_in when new encoding process is activated.
296  DFTout<=DFT(data_in((p*GFPower-17)downto(p*GFPower-24)),data_in((p*GFPower-9)downto(p*GFPower-16))
297              ,data_in((p*GFPower-1)downto(p*GFPower-8)));
298  --p*GFpower=3*8=24. so the three components are data_in(7 downto 0), data_in(15 downto 8), and
299  --data_in(23 downto 16)
300
```

Figure 4–16: Preprocessing of RS encoder with $p = 3$.

LFSR Implementation. The three LFSRs used in this encoder follow the same structure shown in Figure 4–7. As the working principle has been introduced at the beginning of Section 4.2, let us directly look into its VHDL implementation.

As message-words are split into $p = 3$ components, each LFSR is designed to totally take in $t = k/p = 225/3 = 75$ message-words and generate $l = r/p = (n - k)/p = (255 - 225)/3 = 10$ parity-bit symbols. Figure 4–17 shows the port declaration of LFSR component. Two things should be noticed. First, both of "**data_in**" and "**data_out_p**" are a vector with 8 bits and the generator polynomial coefficients are inputs to be filled when instantiating the component. Figure 4–18 shows the signal declaration. The signals with suffix "**_p**" are all register signals. The VHDL codes for register update are shown in Figure 4–19. A counter is defined and its behavior is shown in Figure 4–20. Control signals are defined in Figure 4–21. "**DoCalc_p**" enables parity bits calculation. "**xferPB_p**" enables computed parity bits to be shifted out. The registers accommodating parity bits are denoted as "**parity_reg_p**". Each register update follows exactly as the circuit in Figure 4–7 and the corresponding VHDL codes are shown in Figure 4–22. The output data of an LFSR is the same as the input data for the first 75 clock cycles and then becomes the value stored in the last parity bits register (Figure 4–23). Specific to low latency $RS(255, 225)$ with $p = 3$, the outputs of LFSRs for the first 75 clock cycles are not concerned. Also, it should be emphasized that the parity-bits computed by LFSRs are essentially intermediate parity bits, which need to be post-processed to produce the real parity-bits for the final code-words.

Output controller. Inverse Fourier transform needs to be applied on the intermediate parity bits in order to generate the real parity bits. Recall that code-words consist of message-words and parity bits. During the first 75 cycles, the output data are simply the delayed input message-words. When "**xferEn_p**" is activated, the output data are the results generated from IDFT. The VHDL codes for the post-processing is shown in Figure 4–24.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity LowLatRSenc_LFSR_n85k75 is
6  --this file is the vhdl code for linear feedback shift register (LFSR) used in low latency RS(15,9,p=3) encoder
7  port (reset_n: in std_logic;
8      data_in: in std_logic_vector (7 downto 0); --the input is one word (8 bits) at a time.
9      clk: in std_logic;
10     input_strobe: in std_logic;
11     g0: in std_logic_vector (7 downto 0);
12     g1: in std_logic_vector (7 downto 0);
13     g2: in std_logic_vector (7 downto 0);
14     g3: in std_logic_vector (7 downto 0);
15     g4: in std_logic_vector (7 downto 0);
16     g5: in std_logic_vector (7 downto 0);
17     g6: in std_logic_vector (7 downto 0);
18     g7: in std_logic_vector (7 downto 0);
19     g8: in std_logic_vector (7 downto 0);
20     g9: in std_logic_vector (7 downto 0);
21     data_out_p: out std_logic_vector (7 downto 0)); --the output is one word (8 bits) at a time
22 end;
23
```

Figure 4–17: Port Declaration of LFSR component.

```
25 architecture RTL of LowLatRSenc_LFSR_n85k75 is
26 ----------------------------basic parameters----------------------------------------
27 constant data_size:Galois_Field_element:="01001011"; --data_size=t=k/p
28 constant coded_size:Galois_Field_element:="01010101";--coded_size=m=n/p
29 constant errCap: integer:=5; --error correction capacity
30 constant GFPower: integer:=8;--Galois Field power to define GF(2^8).
31 ----------------------------pre-defined useful types--------------------------------
32 subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
33 type parity_reg_type is array (0 to (2*errCap-1)) of Galois_Field_element; --10 parity-bits registers
34 ----------------------------constant-----------------------------------------------
35 constant zero: Galois_Field_element :="00000000";
36 constant one: Galois_Field_element :="00000001";
37 ----------------data vectors-------------------------------------------------------
38 signal sum: Galois_Field_element; --the sum of last register + input data
39 signal parity_reg,parity_reg_p: parity_reg_type; --parity-bits registers
40 signal data_out:std_logic_vector (7 downto 0); --combinational logic signal for data_out_p
41 signal counter,counter_p: std_logic_vector ((GFPower-1) downto 0); -- define a counter
42 ----------------control signals----------------------------------------------------
43 signal counterEn,counterEn_p: std_logic; --counter only counts when counterEn is activated
44 signal DoCalc,DoCalc_p: std_logic; --when DoCalc is high, LFSR is calculating parity bits
45 signal xferPB,xferPB_p: std_logic; --when all parity bits calculation finished,
46                                    --xferPB is high and parity bits are shifted out
47
```

Figure 4–18: Signal Declaration of LFSR component.

```
110 begin
111 ----------register update----------------------------
112 process (clk)
113 begin
114    if (clk'event and clk='1') then
115       if(reset_n='1') then
116          DoCalc_p<='0';
117          xferPB_p<='0';
118          counterEn_p<='0';
119          counter_p<=(others=>'0');
120          for i in 0 to (2*errCap-1) loop
121             parity_reg_p(i)<=(others=>'0');
122          end loop;
123          data_out_p<=(others=>'0');
124       else
125          DoCalc_p<=DoCalc;
126          xferPB_p<=xferPB;
127          counterEn_p<=counterEn;
128          counter_p<=counter;
129          parity_reg_p<=parity_reg;
130          data_out_p<=data_out;
131       end if;
132    end if;
133 end process;
```

Figure 4–19: Register update of LFSR component.

```
134   ----------counter----------
135   process(input_strobe,counterEn_p,counter_p)
136   begin
137      if (input_strobe='1') then
138         counter<= one;
139      elsif (counterEn_p='1') then
140         counter<=counter_p+1;
141      else
142         counter<=(others=>'0');
143      end if;
144   end process;
145
```

Figure 4–20: Counter behavior of LFSR component.

```
146   ----------combinational logic for control signals----------
147   process(input_strobe,counter_p,DoCalc_p)
148   begin
149      DoCalc<=DoCalc_p;
150      if (input_strobe='1') then
151         DoCalc<='1';
152      elsif (counter_p=data_size) then
153         DoCalc<='0';
154      end if;
155   end process;
156
157   process(input_strobe,counter_p,counterEn_p)
158   begin
159      counterEn<=counterEn_p;
160      if (input_strobe='1') then
161         counterEn<='1';
162      elsif (counter_p=coded_size) then
163         counterEn<='0';
164      end if;
165   end process;
166
167   process(counter_p,xferPB_p)
168   begin
169      xferPB<=xferPB_p;
170      if (counter_p=data_size) then
171         xferPB<='1';
172      elsif (counter_p=coded_size) then
173         xferPB<='0';
174      end if;
175   end process;
176
```

Figure 4–21: Combinational logic for control signals of LFSR component.

```
179   ------------------parity bits register----------------------------
180   process(reset_n,input_strobe,DoCalc_p,parity_reg_p,sum,g0,
181           g1,g2,g3,g4,g5,g6,g7,g8,g9,xferPB_p)
182   begin
183      parity_reg<=parity_reg_p;
184      if(input_strobe='1') then
185         for i in 0 to (2*errCap-1) loop
186            parity_reg(i)<=(others=>'0');
187         end loop;
188      elsif (DoCalc_p='1') then
189         parity_reg(9)<=add(parity_reg_p(8),mul(sum,g9));
190         parity_reg(8)<=add(parity_reg_p(7),mul(sum,g8));
191         parity_reg(7)<=add(parity_reg_p(6),mul(sum,g7));
192         parity_reg(6)<=add(parity_reg_p(5),mul(sum,g6));
193         parity_reg(5)<=add(parity_reg_p(4),mul(sum,g5));
194         parity_reg(4)<=add(parity_reg_p(3),mul(sum,g4));
195         parity_reg(3)<=add(parity_reg_p(2),mul(sum,g3));
196         parity_reg(2)<=add(parity_reg_p(1),mul(sum,g2));
197         parity_reg(1)<=add(parity_reg_p(0),mul(sum,g1));
198         parity_reg(0)<=mul(sum,g0);
199      elsif (xferPB_p='1') then
200         for i in (2*errCap-1) downto 1 loop
201            parity_reg(i)<=parity_reg_p(i-1);
202         end loop;
203         parity_reg(0)<=(others=>'0');
204      end if;
205   end process;
206
```

Figure 4–22: Combinational logic for parity-bits registers of LFSR component.

```
207   ---------------output of LFSR------------------
208   process(DoCalc_p,data_in,parity_reg_p)
209   begin
210      if (DoCalc_p='1') then
211         data_out<=data_in;
212      else
213         data_out<=parity_reg_p((2*errCap-1));
214      end if;
215   end process;
216
217   sum<=add(parity_reg_p(2*errCap-1),data_in);
218   end architecture;
```

Figure 4–23: Combinational logic for the output data of LFSR component.

```
301   process(counter_p)
302   begin
303      if (counter_p=1)then
304         output_strobe<='1'; --a signal that suggests the first output is ready to be sent out
305      else
306         output_strobe<='0';
307      end if;
308   end process;
309
310   process(counter_p,xferEn_p,LFSRout,data_inbuf_p)
311   begin
312      if (counter_p>0 and counter_p<t+1) then
313         data_out<=data_inbuf_p;
314      elsif (xferEn_p='1') then --we only want the last ten output from LFSR
315         data_out((p*GFPower-17)downto(p*GFPower-24))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2))(0);
316         data_out((p*GFPower-9)downto(p*GFPower-16))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2))(1);
317         data_out((p*GFPower-1)downto(p*GFPower-8))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2))(2);
318      else
319         data_out<=(others=>'0');
320      end if;
321   end process;
322   end architecture;
```

Figure 4–24: Post-processing of low-latency $RS(255, 225)$ encoder with $p = 3$.

4.2.2 Implementations with Speed-Up Coefficients $p = 5$ and $p = 15$

Both of low-latency $RS(255, 225)$ encoders with $p = 5$ and $p = 15$ follow the same implementation logic and style. In this subsection, only the difference on implementation are listed and discussed. VHDL codes of the different parts are shown in Appendix A.

Set-Up. The sizes of "**data_in**" and "**data_out_p**" are different for each p value because p value decides how many message-words are transmitted at each clock cycle and correspondingly, the number of bits is equal to $(p \cdot GFPower)$, that is, 40 bits for $p = 5$ and 120 bits for $p = 15$. The LFSR component declaration is certainly changed because for different p there are different LFSR length and thus, the number of generator polynomial coefficients is different. As for the signal declaration, as long as basic parameters are changed, all other modification will happen automatically such as pre-defined types, data signals, because they are defined by the basic parameters. One more modification is to add more constants for the higher order of q because DFT and IDFT functions for bigger p are defined based on them.

Defining Function. For different p values, DFT and IDFT functions need to be modified based on the formula $DFT(M)_i \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} M_j$ and $IDFT(f)_i = \Sigma_{j=0}^{p-1} \rho^{-j \cdot i} f_j$.

Component Instantiations. There are always p LFSRs instantiated. Also in each instantiation, modification should be made based on the component declaration which has been discussed above.

Preprocessing before LFSR. The only thing that needs to be revised is the expression for "**DFTout**", based on the change in the DFT function and the size of "**data_in**". Recall that the DFT function always has p inputs, and each input is in the type of "**Galois_Field_element**".

LFSR Implementation. There are three parts to modify for a distinct p value. First, the number of polynomial generator coefficients are changed due to the length change of LFSR. Second, "**data_size**", "**coded_size**" and "**errCap**"of basic parameters need to change based on the equations $t = k/p$, $m = n/p$ and $l = r/p$, respectively. Third, the combinational logic of "**parity_reg**" is changed by deleting the expressions of its last-part components, as the LFSR becomes shorter for larger a p value.

Output Controller. The only place to modify is the expressions of "**data_out**", due to its size change and the change in the number of IDFT function inputs and outputs.

4.3 Synthesis and Simulations Results for the Encoder

This section include both FPGA synthesis results and ASIC synthesis results. For better explanation, the conventional RS encoder can be thought as a low-latency RS encoder with $p = 1$ which has no DFT and IDFT components.

The FPGA used is Altera DE2-115 board, specifically Cyclone IV E with device number "EP4CE115F23C8L". The software used for synthesis and simulations is Quartus II 15.0. The synthesis results are shown in Table 4–3. The functional simulation results for encoders with each p value are shown Figure 4–25, 4–26 and 4–27. The timing simulation results are shown in Appendix C. The results have been verified by the encoder implemented in MATLAB. For better presentation, all data signals are represented in hexadecimal form, and the 225 input message-words are all set to 3, that is "03" in hexadecimal. As three message-words are input at each clock for $p = 3$, so "**data_in**" is "030303" for all input clocks. Similarly, "**data_in**" is "0303030303" for $p = 5$ and "030303030303030303030303030303" for $p = 15$.

As the results shown, the low-latency RS encoders can speed up the encoding algorithm by a factor almost equal to p, which proves the major feature

$RS(255, 225)$ Encoders with Error-Correction Capacity 15				
Altera DE2-115 FPGA				
Encoder Name	Total Clock Cycles	Max Frequenc (MHz)	Hardware Costs	
			Registers	Combinational Logic
Conventional RS Encoder	256	249.53	323	179
Low-Latency RS Encoder ($p = 3$)	87	204.21	362	210
Low-Latency RS Encoder ($p = 5$)	53	236.46	410	1031
Low-Latency RS Encoder ($p = 15$)	19	206.78	650	1914

Table 4–3: Comparison of FPGA synthesis results for different encoders.

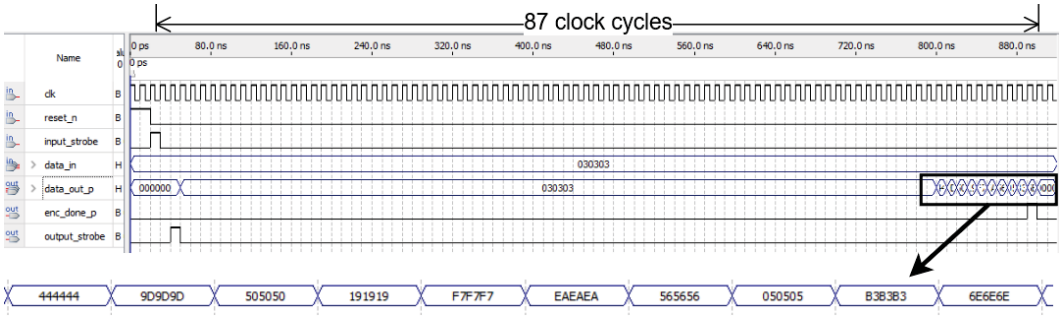


Figure 4–25: Functional simulation of low-latency $RS(255, 225)$ encoder with $p = 3$.

claimed in [17]. The maximum clock frequency decreases with p increasing because larger p causes more complicated DFT and IDFT components, which leads to longer critical path and thus smaller maximum clock frequency. However, one thing should be noticed that for $p = 5$, the encoder's maximum clock frequency is surprised high. While this abnormal fact did not show up in ASIC synthesis results. The possible reason can be the routing rules of FPGA responding better to shorter LFSRs, and when $p = 5$ this benefit surpasses

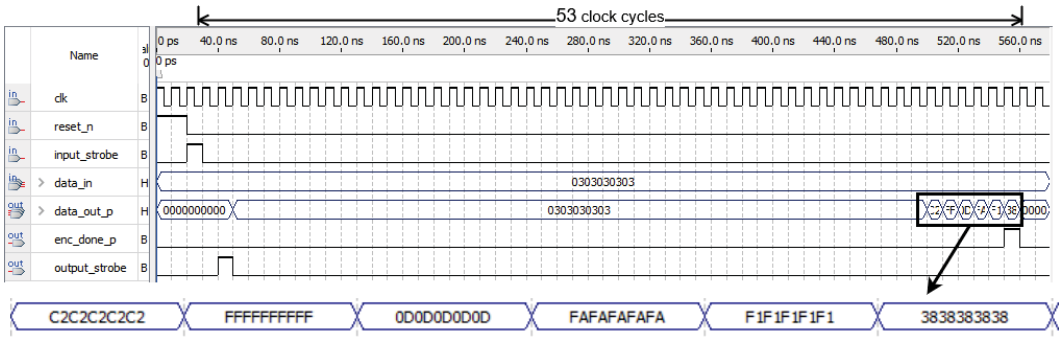


Figure 4–26: Functional simulation of low-latency $RS(255, 225)$ encoder with $p = 5$.

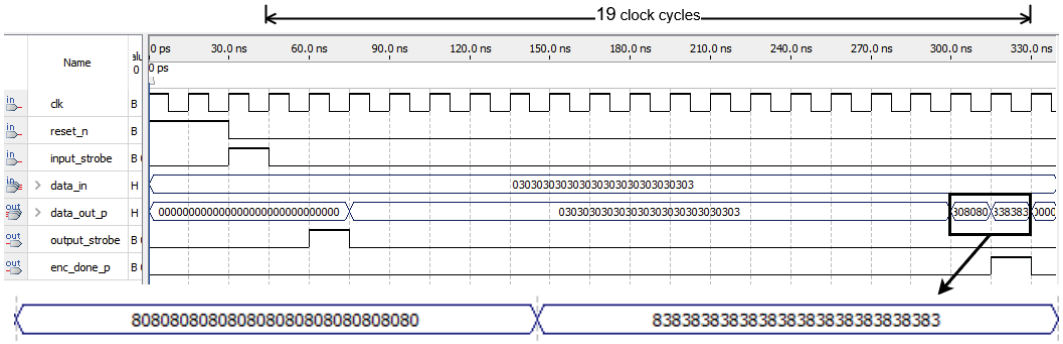


Figure 4–27: Functional simulation of low-latency $RS(255, 225)$ encoder with $p = 15$.

the drawback from DFT and IDFT components. As for the hardware cost, the resource used by the low-latency RS decoder with $p = 15$ is about five times of that used by a conventional RS encoder. The reason can be explained in term of the structures. A low-latency RS encoder or decoder is essentially made by breaking a conventional RS encoder or decoder into p parts and connecting them to DFTs and IDFTs. As the hardware cost of the conventional RS encoder is very small, the hardware costs of the DFT and IDFT components that are proportional to p, have a significant impact on the total hardware costs. Nevertheless, this situation does not exist for low-latency decoder implementation because the conventional RS decoders are much more complicated. Moreover, the hardware costs of a low-latency RS encoder are

still small. For the encoder with $p = 15$, the resource usage is less than 2% of FPGA according to the synthesis report from Quartus II.

$RS(255, 225)$ Encoders with Error-Correction Capacity 15				
65nm TSMC				
Encoder Name	Area (μm^2)	Power Comsumption (mW)	Max Frequency GHz	Data Rate Gbps
Conventional RS Encoder	$0.798 \cdot 10^4$	5.601	2.77	22.16
Low-Latency RS Encoder ($p = 3$)	$0.629 \cdot 10^4$	3.97	2.71	65.04
Low-Latency RS Encoder ($p = 5$)	$1.36 \cdot 10^4$	13.34	2.68	107.2
Low-Latency RS Encoder ($p = 15$)	$4.79 \cdot 10^4$	23.18	2.08	249.6

Table 4–4: Comparison of ASIC synthesis results for different encoders.

The technology used for ASIC synthesis is 65 nm TSMC and the results are shown in Table 4–4. The data rate increases about linearly with p, which is expected as it is one of the major features of the proposed RS algorithm. Another expectation got verified is that max clock frequency decreases also about linearly with p, which is explained in last paragraph. As for areas, the results have consistent trend to the hardware costs from FPGA synthesis. Specifically, the encoder with larger p costs more hardware and area. However, one exception is that the area for conventional RS encoder slightly larger than the low-latency one with $p = 3$. There are two possible reasons responsible for this inconsistence. First, some arithmetic logical elements are omitted when the one long LFSR of the conventional RS encoder is split into three shorter LFSRs of the low-latency one with $p = 3$. Second, the rules of ASIC

routing may result in a larger area for the longer sequential structure, which is a subject to confirm in the future work. The power consumption results of the encoders are expected and can be explained by dynamic power formula of CMOS transistors, as it occupies about 80%-90% of total power. The formula is $P = CV^2FA$ where C is the capacitance, V is the supply voltage, F is the clock frequency, and A is the activity factor modeling the average switching activity. Correspondingly, the power consumption results are mainly following the trend of area as the change in hardware costs is large, whereas the clock frequency difference is relatively small.

In conclusion, although low-latency with small p value can indeed improve the conventional design, large p value also raises the hardware cost and power consumption significantly because the simplicity of the encoder structure makes the added DFT and IDFT components affect considerably.

4.4 Decoder Implementation

In this research, three low-latency $RS(255, 225)$ decoders with different speed-up coefficient p have been implemented in VHDL and tested. The implementation logic follows the structure shown in Figure 3–2. The entire implementation consists of four VHDL files. One is the top file, and the others are components. All the details are discussed in the rest of the section.

4.4.1 Implementation with Speed-Up Coefficient p=3

Figure 4–28 is a block diagram for low-latency $RS(255, 225)$ decoder with $p = 3$. At each clock, three received-words enter a splitter and are split into three single received-word. Then they enter into three syndrome calculators. This process repeats for $n/p = 85$ clock cycles and each syndrome calculator finally generates $r/p = 10$ syndromes. The 30 syndromes are organized as a single vector and enter Berlekamp-Massey algorithm in which a vector containing the error-locator polynomial coefficients and another vector containing

the error-evaluator polynomial coefficients are computed after $r = 30$ clock cycles. Afterward, the modified Chien search and correction process is activated. The error-locator polynomial is split into three parts, and each one becomes an error-location searcher. At each clock cycle, the results generated from error location searchers are then applied with inverse Fourier transform to see if any error locations are found. At the same time, error evaluation is also running based on error-evaluator polynomial and error-locator polynomial. To be consistent with the number of error locations that are checked at each clock cycle, three same error-evaluation processors are used. At each clock cycle, the output controller sent out a vector containing three correction-words. When any error location is found, the corresponding error value is added to the received-word at that location, and the sum is sent out as a correction-word. Otherwise, the correction-words are just as same as the received-words. After another $n/p = 85$ clock cycles, all the correction-words are sent out. In the following paragraphs, VHDL codes for each file are discussed with details.

LowLatRSdec_n255k225_p3_syn.vhd. Figure 4–29 shows the port declaration. There are four inputs. Besides "**reset**" and clock signal "**clk**", another input "**data_in**" is a vector with 24 bits because three 8-bit received-words are received at each clock cycle. "**input_strobe**" is also an input signal to activate the decoding process. The three outputs are "**syndrome_out**" which is a vector with 240 bits (30 8-bit syndromes), "**error_present_out**" which suggests if there is any error in the received-words, and "**synd_calc_done**" which suggests the end of the syndrome calculations and activates the Berlekamp-Massey algorithm.

Figure 4–30 shows the signal declaration. The signals with suffix "**_p**" are all register signals and their counterparts are the combinational logic signals for register update. A constant vector "**alpha**" is created to include

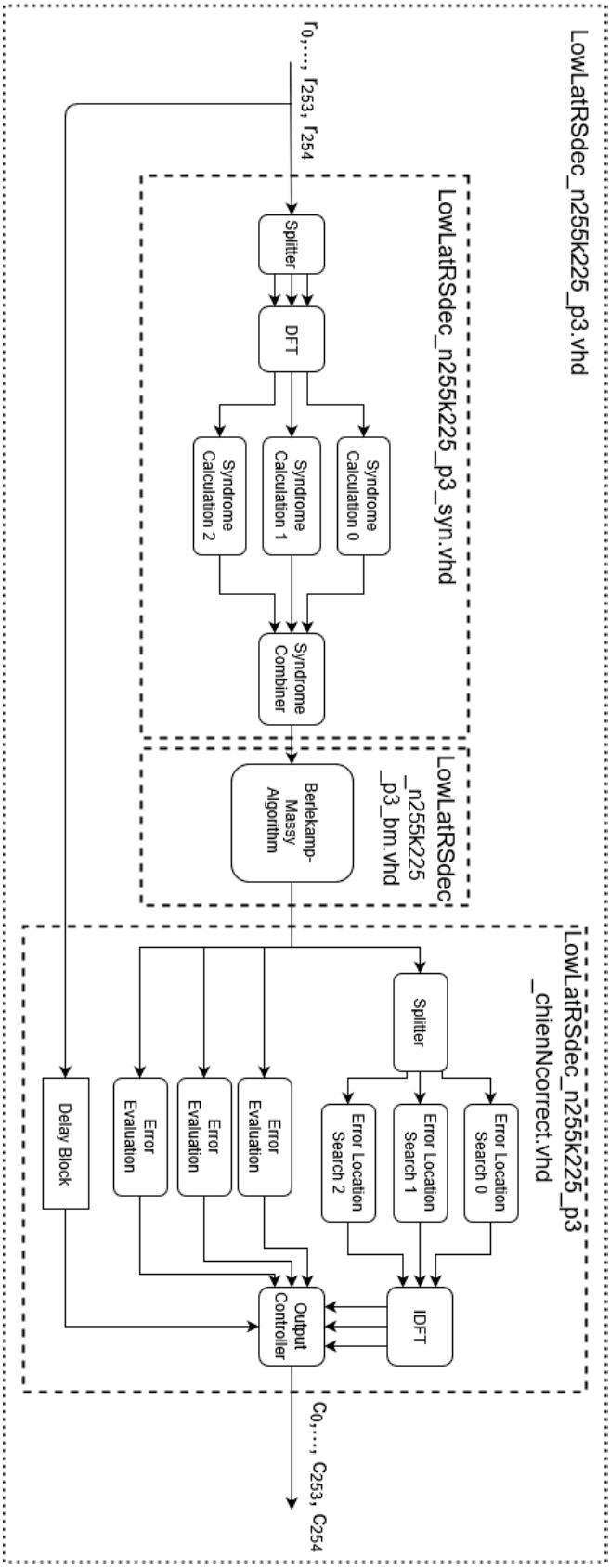


Figure 4–28: Structure of low-latency $RS(255, 225)$ decoder with $p = 3$.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity LowLatRSdec_n255k225_p3_syn is
6   port (reset: in std_logic;
7       data_in: in std_logic_vector(23 downto 0);
8       clk: in std_logic;
9       input_strobe: in std_logic;
10      synd_calc_done: out std_logic;
11      error_present_out:out std_logic;
12      syndrome_out: out std_logic_vector (239 downto 0)
13      );-- 30 syndromes and each syndrome has 8 bits=>total 240 bits
14  end;
```

Figure 4–29: Port declaration of syndrome component.

all elements in $GF(2^8)$. In data signals part, "**intS_p**" refers to the intermediate syndromes and it is a two-dimensional vector where each row refers to all the ten syndromes calculated from a particular syndrome calculator. "**intEP_p**" refers to intermediate error-present signal whose size is consistent with "**intS_p**". As for control signals, "**xferSyn_p**" is a signal to activate output process of the final syndrome vector.

The declaration of addition, multiplication and DFT functions are the same as those of the encoder which has been already described in the last section.

Figure 4–31 shows the process for register update. All register signals are updated to their combinational logic counterparts at each clock's rising edge except when "**reset**" is 1, in which case all register signals become zero.

Figure 4–32 shows a finite state machine. There are four state: "**Idle**", "**Initialization**", "**Calculation**" and "**OutputReady**". When "**input_strobe**" is high, the state transfers from "**Idle**" to "**Initialization**" in which the counter is enabled and all signals are set up. This state is only up for one clock cycle and followed by "**Calculation**" state in which the control signal "**DoCalc**" is activated and intermediate syndrome calculation starts. After

```
architecture RTL of LowLatRSdec_n255k225_p3_syn is
---------basic parameters-------------------------------------------------
constant GFPower: integer:=8;
constant N: integer:=255; --codeword length
constant K: integer:=225; --message word length
constant R: integer:=30; --parity bits length =255-225
constant m: integer:=85; --m=N/p=255/3=85
constant t: integer:=75; --t=K/p=225/3=75
constant L: integer:=10; --L=R/p=30/3=10
constant p: integer:=3;--speed-up coefficient
constant bitNum: integer:=p*GFpower; --number of bits in data_in
-------------predefined useful types-------------------------------------
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
type alpha_type is array (0 to 254) of Galois_Field_element;
type intS_subtype is array (0 to (L-1)) of Galois_Field_element;
type intS_type is array (0 to p-1) of intS_subtype;
type intEP_subtype is array(0 to (L-1)) of std_logic;
type intEP_type is array(0 to p-1) of intEP_subtype;
type FT_type is array (0 to (p-1)) of Galois_Field_element;
type state_type is (Idle, Initialization, Calculation, OutputReady);
--------constants-----------------------------------------------------------------------------------
constant zero: Galois_Field_element :="00000000";
constant alpha:alpha_type:=     --all the elements of GF(2^8), starts with alpha(0)=a^0.
("00000001","00000010","00000100","00001000","00010000","00100000","01000000","10000000"
,"00011101","00111010","01110100","11101000","11001101","10000111","00010011",
"00100110","01001100","10011000","00101101","01011010","10110100","01110101","11101010",
"11001001","10001111","00000011","00000110","00001100","00011000","00110000",
"01100000","11000000","10011101","00100111","01001110","10011100","00100101","01001010",
"10010100","00110101","01101010","11010100","10110101","01110111","11101110",
"11000001","10011111","00100011","01000110","10001100","00000101","00001010","00010100",
"00101000","01010000","10100000","01011101","10111010","01101001","11010010",
"10111001","01101111","11011110","10100001","01011111","10111110","01100001","11000010",
"10011001","00101111","01011110","10111100","01100101","11001010","10001001",
"00001111","00011110","00111100","01111000","11110000","11111101","11100111","11010011",
"10111011","01101011","11010110","10110001","01111111","11111110","11100001",
"11011111","10100011","01011011","10110110","01110001","11100010","11011001","10101111",
"01000011","10000110","00010001","00100010","01000100","10001000","00001101",
"00011010","00110100","01101000","11010000","10111101","01100111","11001110","10000001",
"00011111","00111110","01111100","11111000","11101101","11000111","10010011",
"00111011","01110110","11101100","11000101","10010111","00110011","01100110","11001100",
"10000101","00010111","00101110","01011100","10111000","01101101","11011010","10101001",
"01001111","10011110","00100001","01000010","10000100","00010101","00101010",
"01010100","10101000","01001101","10011010","00101001","01010010","10100100","01010101",
"10101010","01001001","10010010","00111001","01110010","11100100","11010101",
"10110111","01110011","11100110","11010001","10111111","01100011","11000110","10010001",
"00111111","01111110","11111100","11100101","11010111","10110011","01111011",
"11110110","11110001","11111111","11100011","11011011","10101011","01001011","10010110",
"00110001","01100010","11000100","10010101","00110111","01101110","11011100",
"10100101","01010111","10101110","01000001","10000010","00011001","00110010","01100100",
"11001000","10001101","00000111","00001110","00011100","00111000","01110000",
"11100000","11011101","10100111","01010011","10100110","01010001","10100010","01011001",
"10110010","01111001","11110010","11111001","11101111","11000011","10011011",
"00101011","01010110","10101100","01000101","10001010","00001001","00010010","00100100",
"01001000","10010000","00111101","01111010","11110100","11110101","11110111",
"11110011","11111011","11101011","11001011","10001011","00001011","00010110","00101100",
"01011000","10110000","01111101","11111010","11101001","11001111","10000011",
"00011011","00110110","01101100","11011000","10101101","01000111","10001110");
constant q: Galois_Field_element:="11010110"; --set the 3rd root of unity
constant q2: Galois_Field_element:="11010111"; --square of the 3rd root of unity
---------------data signals-------------------------------------------
signal syndrome,syndrome_p:std_logic_vector (239 downto 0);
signal counter,counter_p: integer:=0;
signal intS,intS_p: intS_type; --intermediate syndromes. two-dimensional vector
signal intEP: intEP_type; --intermediate error_present, the size is consistent to intS and intS_p
signal DFTout: FT_type;
--------------control signals-----------------------------------------
signal state,state_p: state_type;
signal error_present,error_present_p:std_logic;
signal initialize,initialize_p:std_logic;
signal countEn,countEn_p:std_logic;
signal countRst,countRst_p:std_logic;
signal xferSyn,xferSyn_p: std_logic;
signal DoCalc,DoCalc_p: std_logic;

```

Figure 4–30: Signal declaration of syndrome component.

```
begin
----------------register update-----------------------
process(clk)
begin
   if (clk'event and clk='1') then
      if(reset='1') then
         xferSyn_p<='0';
         initialize_p<='0';
         DoCalc_p<='0';
         countRst_p<='1';
         countEn_p<='0';
         state_p<= Idle ;
         counter_p<=0;
         for i in 0 to p-1 loop
            for j in 0 to (L-1) loop
               intS_p(i)(j)<=zero;
            end loop;
         end loop;
         syndrome_p<=(others=>'0');
         error_present_p<='0';
      else
         xferSyn_p<=xferSyn;
         initialize_p<=initialize;
         DoCalc_p<=DoCalc;
         countRst_p<=countRst;
         countEn_p<=countEn;
         state_p<= state ;
         counter_p<=counter;
         intS_p<=intS;
         syndrome_p<=syndrome;
         error_present_p<=error_present;
      end if;
   end if;
end process;
```

Figure 4–31: Register update of syndrome component.

$n/p = 85$ clock cycles, the calculation process is finished, the control signal "**xferSyn**" is up and the state turns to "**OutputReady**" where all intermediate syndromes are ready to be organized into a single syndrome vector. One clock cycle later, the state goes to "**Idle**" automatically.

Figure 4–33 shows the counter behavior. As usual, the counter only counts when "**countEn_p**" is high.

Figure 4–34 shows all other combinational logic processes for data signals. "**DFTout**" is acquired by applying Fourier transform on the input received-words. Each intermediate syndrome is calculated using a conventional recursive multiplication shown in Figure 4–35 and it follows the formula given in Step 3 of Section 3.2.2. Each "**intEP**" suggests if the intermediate syndrome at the same index is a zero. For the final output, "**syndrome**" and "**error-present**" are both generated by connecting all the corresponding intermediate data signals and one clock cycle later, these values are passed

```
196   ------------------finite state machine---------------
197   process(state_p,input_strobe)
198   begin
199      case state_p is
200         when Idle =>
201            if (input_strobe='1') then
202               xferSyn<='0';
203               initialize<='1';
204               DoCalc<='0';
205               countRst<='0';
206               countEn<='1';
207               state<=Initialization;
208            else
209               xferSyn<='0';
210               initialize<='0';
211               DoCalc<='0';
212               countRst<='1';
213               countEn<='0';
214               state<=Idle;
215            end if;
216
217         when Initialization =>
218               xferSyn<='0';
219               initialize<='0';
220               DoCalc<='1';
221               countRst<='0';
222               countEn<='1';
223               state<=Calculation;
224
225         when Calculation=>
226            if (counter_p=m-1) then
227               initialize<='0';
228               xferSyn<='1';
229               DoCalc<='0';
230               countRst<='1';
231               countEn<='0';
232               state<=OutputReady;
233            else
234               xferSyn<='0';
235               initialize<='0';
236               DoCalc<='1';
237               countRst<='0';
238               countEn<='1';
239               state<=Calculation;
240            end if;
241
242         when OutputReady =>
243            xferSyn<='0';
244            initialize<='0';
245            DoCalc<='0';
246            countRst<='1';
247            countEn<='0';
248            state<=Idle;
249
250         when others=>
251            xferSyn<='0';
252            initialize<='0';
253            DoCalc<='0';
254            countRst<='1';
255            countEn<='0';
256            state<=Idle;
257      end case;
258   end process;
```

Figure 4–32: Finite State Machine of syndrome component.

```
260  process(countRst_p,countEn_p,counter_p)
261  begin
262     if(countRst_p='1')then --reset when reset='1' (not '0'!!).
263        counter<=0;
264     elsif (countEn_p='1') then
265        counter<=counter_p+1;
266     else
267        counter<=0;
268     end if;
269  end process;
```

Figure 4–33: Counter process of syndrome component.

to "**syndrome_out**" and "**error_present_out**". Besides, "**synd_calc_done**" which indicates the end of syndrome calculations and activates the Berlekamp-Massey algorithm, is a copy of "**xferSyn_p**" and only up for one clock as well.

LowLatRSdec_n255k225_p3_bm.vhd. Figure 4–36 shows the port declaration. "**synd_calc_done**" is the input control signal indicating that the syndrome calculations have been done. When it is equal to 1, Berlekamp-Massey algorithm is activated. "**error_present**" is an input signal generated from last component and indicating if there is any error in the received-words. "**syndrome**" is a 240-bit input vector containing all syndromes calculated from last component. There are three outputs: an error-locator polynomial denoted as "**lambda_poly**", an error-evaluator polynomial denote as "**omega_poly**" and a control signal "**startChien_p**" which indicates the end of Berlekamp-Massey algorithm and activates the next decoding process: modified Chien search.

Figure 4–37 shows the signal declaration. "**keepOldL**" is a control signal standing for the condition $\Delta^{(k+1)} \neq 0$ and $2L^{(k)} \leq k$. The number of elements insides "**Omega_p**" and "**Ax_p**" are all 16 instead of 15 because the zero-index element is taken as the coefficient associated with x^{-1}. "**SReg_p**" is a set of register storing syndrome values. Its size is 1.5 times of the number

```
DFTout<=DFT(data_in((bitNum-17)downto(bitNum-24)),data_in((bitNum-9)downto(bitNum-16)),
         data_in((bitNum-1)downto(bitNum-8)));

process(initialize_p,DoCalc_p,DFTout,intS_p)
begin
   intS<=intS_p;
   if (initialize_p='1') then
      for i in 0 to p-1 loop
         for j in 0 to (L-1) loop
            intS(i)(j)<=DFTout(i);
         end loop;
      end loop;
   elsif (DoCalc_p='1') then
      for i in 0 to p-1 loop
         for j in 0 to L-1 loop
            intS(i)(j)<=add(mul(alpha(i+j*p),intS_p(i)(j)),DFTout(i));
         end loop;
      end loop;
   end if;
end process;

process(intS_p)
begin
   for i in 0 to p-1 loop
      for j in 0 to (L-1) loop
         intEP(i)(j)<='1';
         if (intS_p(i)(j)=zero) then
            intEP(i)(j)<='0';
         end if;
      end loop;
   end loop;
end process;

process(initialize_p,xferSyn_p,intS_p,syndrome_p,error_present_p,intEP)
begin
   syndrome<=syndrome_p;
   error_present<=error_present_p;
   if(initialize_p='1') then
      syndrome<=(others=>'0');
      error_present<='0';
   elsif (xferSyn_p='1') then
      syndrome<=intS_p(2)(9)& intS_p(1)(9) & intS_p(0)(9) & intS_p(2)(8)& intS_p(1)(8) & intS_p(0)(8)
               & intS_p(2)(7)& intS_p(1)(7) & intS_p(0)(7) & intS_p(2)(6)& intS_p(1)(6) & intS_p(0)(6)
               & intS_p(2)(5)& intS_p(1)(5) & intS_p(0)(5) & intS_p(2)(4)& intS_p(1)(4) & intS_p(0)(4)
               & intS_p(2)(3)& intS_p(1)(3) & intS_p(0)(3) & intS_p(2)(2)& intS_p(1)(2) & intS_p(0)(2)
               & intS_p(2)(1)& intS_p(1)(1) & intS_p(0)(1) & intS_p(2)(0) & intS_p(1)(0) & intS_p(0)(0);
               -- =(s_(n-k-1),...,s2,s1,s0)
      error_present<=intEP(0)(0) or intEP(0)(1) or intEP(0)(2) or intEP(0)(3) or intEP(0)(4) or intEP(0)(5)
                     or intEP(0)(6) or intEP(0)(7) or intEP(0)(8) or intEP(0)(9) or intEP(1)(0) or intEP(1)(1)
                     or intEP(1)(2) or intEP(1)(3) or intEP(1)(4) or intEP(1)(5) or intEP(1)(6) or intEP(1)(7)
                     or intEP(1)(8) or intEP(1)(9) or intEP(2)(0) or intEP(2)(1) or intEP(2)(2) or intEP(2)(3)
                     or intEP(2)(4) or intEP(2)(5) or intEP(2)(6) or intEP(2)(7) or intEP(2)(8) or intEP(2)(9);
                  -- to check if there is any error appeared in the received code-word
   end if;
end process;

syndrome_out<=syndrome_p;
error_present_out<=error_present_p;
synd_calc_done<=xferSyn_p;
end architecture;
```

Figure 4–34: Combinator logic of syndrome component.

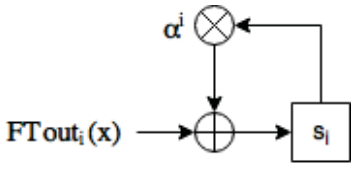


Figure 4–35: Recursive Multiplication Block.

of syndromes, because 15 extra slots need to be left in advance for back-shift of syndrome values according to the formula $\Delta^{(k+1)} = \sum_{j=0}^{L^{(k)}} \sigma_j^{(k)} S_{k-j}$. "**Convolution_Term**", "**Convolution_term_multiplier** and "**post_convol**

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   entity LowLatRSdec_n255k225_p3_bm is
5   port (reset: in std_logic;
6       clk: in std_logic;
7       synd_calc_done: in std_logic;
8       error_present:in std_logic;
9       syndrome: in std_logic_vector (239 downto 0);--30*8=240
10      lambda_poly:out std_logic_vector(127 downto 0);--error locator polynomial 16*8=128
11      omega_poly: out std_logic_vector(119 downto 0); --error evaluator polynomial 15*8=120
12      startChien_p: out std_logic);
13  end;
```

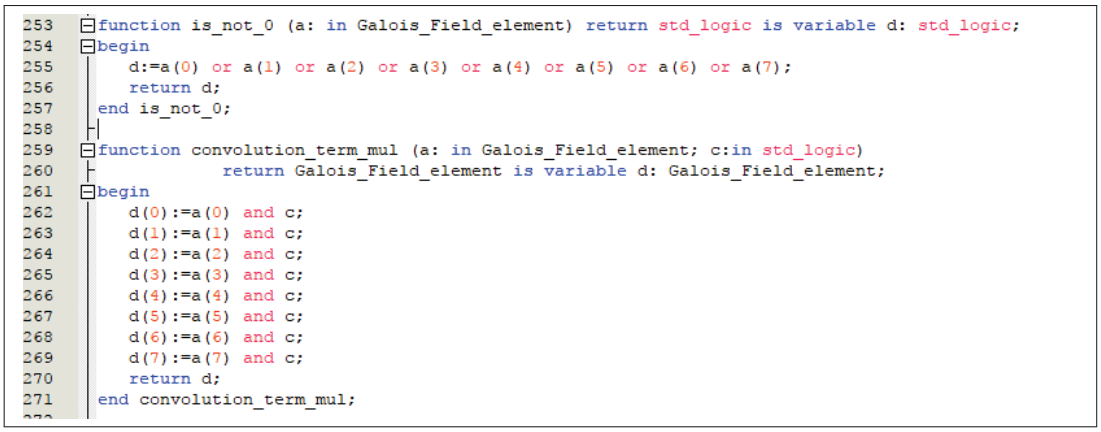
Figure 4–36: Port declaration of BM component.

ution_term" are three supporting vectors for computing "**Delta**". More details are introduced in later paragraphs.

```
17  architecture RTL of LowLatRSdec_n255k225_p3_bm is
18  ----------basic parameters--------------------------------
19  constant GFPower: integer:=8;
20  constant N: integer:=255;
21  constant errCap: integer :=15; --error-correction
22  constant R: integer:=30;
23  ---------------------pre-defined useful types---------------------------
24  subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
25  type SRegType is array (0 to (3*errCap-1)) of Galois_Field_element;
26  type Poly1Type is array(0 to errCap) of Galois_Field_element;
27  type Poly2Type is array(0 to 2*errCap) of Galois_Field_element;
28  type Poly3Type is array(1 to errCap) of Galois_Field_element;
29  type state_type is (Idle,Initialization,Update);
30  ----------------constants-----------------------------------
31  constant zero: Galois_Field_element :="00000000";
32  constant one: Galois_Field_element :="00000001";
33  ---------------control signals-------------------------------
34  signal startChien: std_logic;
35  signal state,state_p: state_type;
36  signal initialize,initialize_p: std_logic;
37  signal storeNewPolys,storeNewPolys_p: std_logic;
38  signal countEn,countEn_p:std_logic;
39  signal KeepOldL:std_logic;-- = if $\Delta^{(k+1)} \neq 0 \text{ and } 2L^{(k)}\leq k$ is true
40  ---------------data signals-----------------------------------
41  signal counter,counter_p:std_logic_vector(7 downto 0);
42  signal SReg,SReg_p: SRegType; --syndrome register
43  signal Delta:Galois_Field_element;
44  signal Convolution_Term: Poly1Type;
45  signal Convolution_term_multiplier: std_logic_vector(errCap downto 0);
46  signal post_convolution_term: poly1Type;
47  signal L,L_p:std_logic_vector (7 downto 0); --maximum L=15
48  signal TwoL:std_logic_vector(7 downto 0); --maximum 2L=30
49  signal Bx,Bx_p: Poly1Type;--coefficient set of error-locater supporting polynomial
50  signal lambda,lambda_p: Poly1Type; --coefficient set of error-locater polynomial
51  signal xDeltaBx: Poly3Type; -- = x*Delta*B(x)
52  signal Omega,Omega_p: Poly1Type;--coefficient set of error-evaluator polynomial
53  signal Ax,Ax_p: Poly1Type; --coefficient set of error-evaluator supporting polynomial
54  signal xDeltaAx: Poly3Type;  -- = x*Delta*A(x)
```

Figure 4–37: Signal declaration of BM component.

For function definitions, as addition, multiplication and inversion have been given in the previous section, the VHDL codes are not shown here again. There are two new functions shown in Figure 4–38. "**is_not_0**" is a function to check if a 8-bit vector is zero. "**convolution_term_mul**" is a multiplication function between a 8-bit vector and another single bit.



```
253  function is_not_0 (a: in Galois_Field_element) return std_logic is variable d: std_logic;
254  begin
255      d:=a(0) or a(1) or a(2) or a(3) or a(4) or a(5) or a(6) or a(7);
256      return d;
257  end is_not_0;
258
259  function convolution_term_mul (a: in Galois_Field_element; c:in std_logic)
260              return Galois_Field_element is variable d: Galois_Field_element;
261  begin
262      d(0):=a(0) and c;
263      d(1):=a(1) and c;
264      d(2):=a(2) and c;
265      d(3):=a(3) and c;
266      d(4):=a(4) and c;
267      d(5):=a(5) and c;
268      d(6):=a(6) and c;
269      d(7):=a(7) and c;
270      return d;
271  end convolution_term_mul;
```

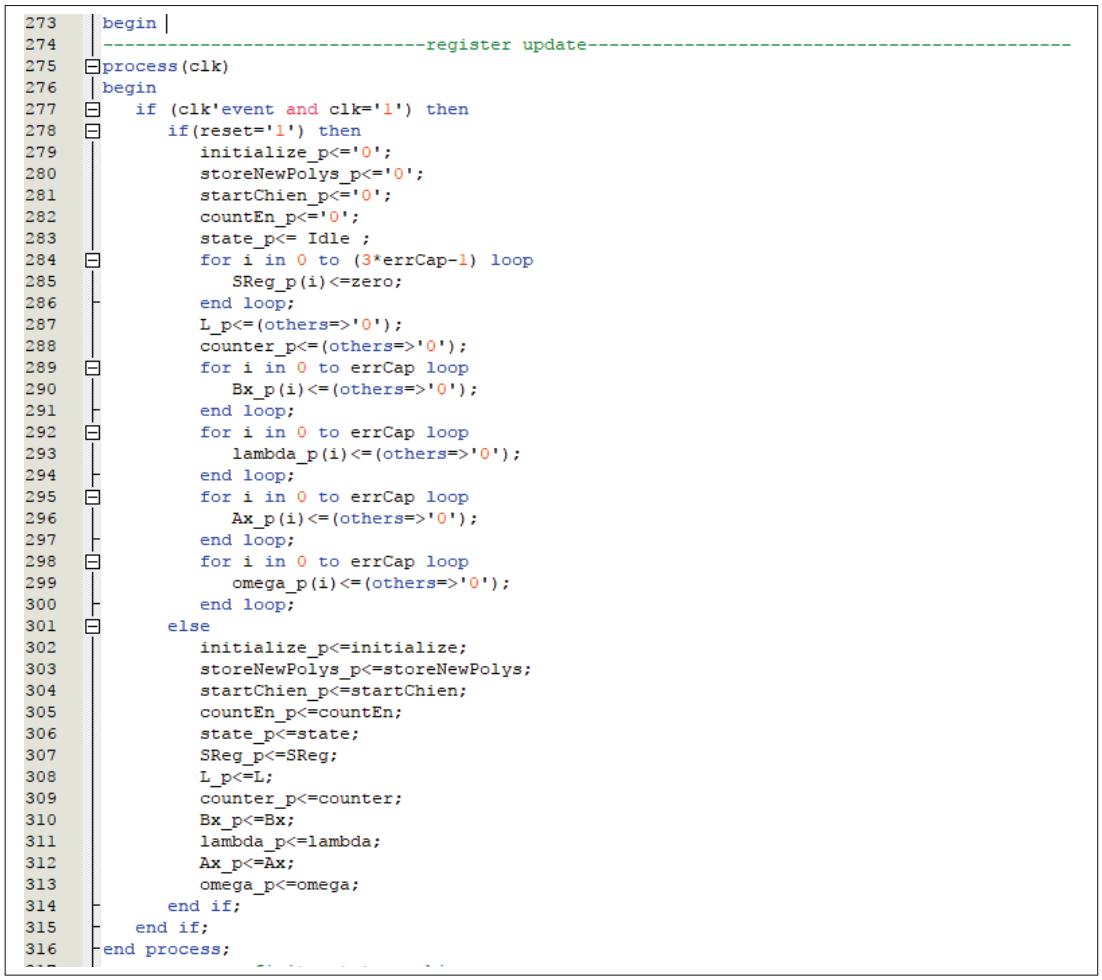
Figure 4–38: Newly defined functions of BM component.

Figure 4–39 shows the register update. At each clock's rising edge, all register signals go to zero, when "**reset**" is high. Otherwise, the register signals are updated to their combinational logic counter parts.

Figure 4–40 shows the finite state machine. When all syndromes are calculated from the last component, "**synd_calc_done**" turns up. Then "**initialize**" goes to 1 and the state transfers from "**Idle**" to "**Initialization**". If "**error_p resent**" is 0, which means there are no error in the received-words, then "**startChien**" turns to 1 and the state goes back to "**Idle**". Otherwise, "**storeNewPolys**" and "**countEn**" become 1 and the state goes to "**Update**". During this state, calculation iterations are under progress and the counter keeps counting. When "**counter_p**" reaches $r = 30$, "**startChien**" is up which indicates the calculation process is finished and the states goes to "**Idle**".

Figure 4–41 shows the counter behavior. As usual, the counter counts only when "**countEn_p**" is 1.

Figure 4–42 shows the calculation process of "**Delta**". In initialization state, 30 syndromes are assigned to the last 30 elements of "**SReg**". While the first 15 elements are assigned to zero. When "**storeNowPolys_p**" is equal to 1, at every clock cycle, the elements of "**SReg**" shift back by one position.



```
273   begin
274   ------------------------------register update-----------------------------------------------
275   process(clk)
276   begin
277      if (clk'event and clk='1') then
278         if(reset='1') then
279            initialize_p<='0';
280            storeNewPolys_p<='0';
281            startChien_p<='0';
282            countEn_p<='0';
283            state_p<= Idle ;
284            for i in 0 to (3*errCap-1) loop
285               SReg_p(i)<=zero;
286            end loop;
287            L_p<=(others=>'0');
288            counter_p<=(others=>'0');
289            for i in 0 to errCap loop
290               Bx_p(i)<=(others=>'0');
291            end loop;
292            for i in 0 to errCap loop
293               lambda_p(i)<=(others=>'0');
294            end loop;
295            for i in 0 to errCap loop
296               Ax_p(i)<=(others=>'0');
297            end loop;
298            for i in 0 to errCap loop
299               omega_p(i)<=(others=>'0');
300            end loop;
301         else
302            initialize_p<=initialize;
303            storeNewPolys_p<=storeNewPolys;
304            startChien_p<=startChien;
305            countEn_p<=countEn;
306            state_p<=state;
307            SReg_p<=SReg;
308            L_p<=L;
309            counter_p<=counter;
310            Bx_p<=Bx;
311            lambda_p<=lambda;
312            Ax_p<=Ax;
313            omega_p<=omega;
314         end if;
315      end if;
316   end process;
```

Figure 4–39: Register update of BM component.

The elements are multiplied with the error-locator polynomial following the formula $\sigma_j^{(k)} \cdot S_{k-j}$. Then the case statement based "**L_p**" and the calculation process of "**post_convolution_term**" determine every term of the formula $\sum_{j=0}^{L^{(k)}} \sigma_j^{(k)} S_{k-j}$. Finally, "**Delta**" is obtained by adding these terms together.

Figure 4–43 shows the update of error-locator polynomial. $\Delta^{(k+1)} \cdot B^{(k)}(x) \cdot x$ is first calculated as "**xDeltaBx**". Then the new error-locator polynomial is computed following the formula $\sigma^{(k+1)}(x) = \sigma^{(k)}(x) - \Delta^{(k+1)} \cdot B^{(k)}(x) \cdot x$.

Figure 4–44 shows the update of error-evaluator polynomial. The process follows the same style as an error-locator polynomial.

```
317   --------------finite state machine----------------------------------------
318   process(state_p,synd_calc_done,error_present,counter_p)
319   begin
320      case state_p is
321         when Idle =>
322            if (synd_calc_done='1') then
323               initialize<='1';
324               storeNewPolys<='0';
325               startChien<='0';
326               countEn<='0';
327               state<= Initialization;
328            else
329               initialize<='0';
330               storeNewPolys<='0';
331               startChien<='0';
332               countEn<='0';
333               state<=Idle;
334            end if;
335
336         when Initialization =>
337            if(error_present='0') then
338               initialize<='0';
339               storeNewPolys<='0';
340               startChien<='1';
341               countEn<='0';
342               state<= Idle;
343            else
344               initialize<='0';
345               storeNewPolys<='1';
346               startChien<='0';
347               countEn<='1';
348               state<= Update;
349            end if;
350
351         when Update=>
352            if (counter_p="011101") then --30-1=29=011101
353               initialize<='0';
354               storeNewPolys<='0';
355               startChien<='1';
356               countEn<='0';
357               state<= Idle;
358            else
359               initialize<='0';
360               storeNewPolys<='1';
361               startChien<='0';
362               countEn<='1';
363               state<= Update;
364            end if;
365            ------------------------------------------------------------------
366         when others=>
367            initialize<='0';
368            storeNewPolys<='0';
369            startChien<='0';
370            countEn<='0';
371            state<= Idle;
372      end case;
373   end process;
```

Figure 4–40: Finite state machine of BM component.

Figure 4–45 shows the update of the order of error-locator polynomial "**L**", the error-locator supporting polynomial "**Bx**", and the error-evaluator supporting polynomial "**Ax**".

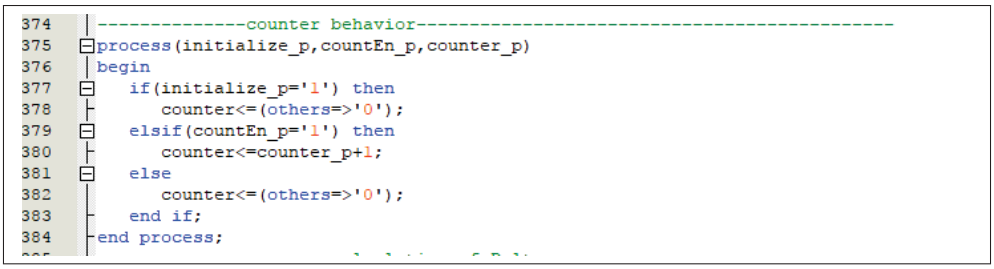


Figure 4–41: Counter behavior of BM component.

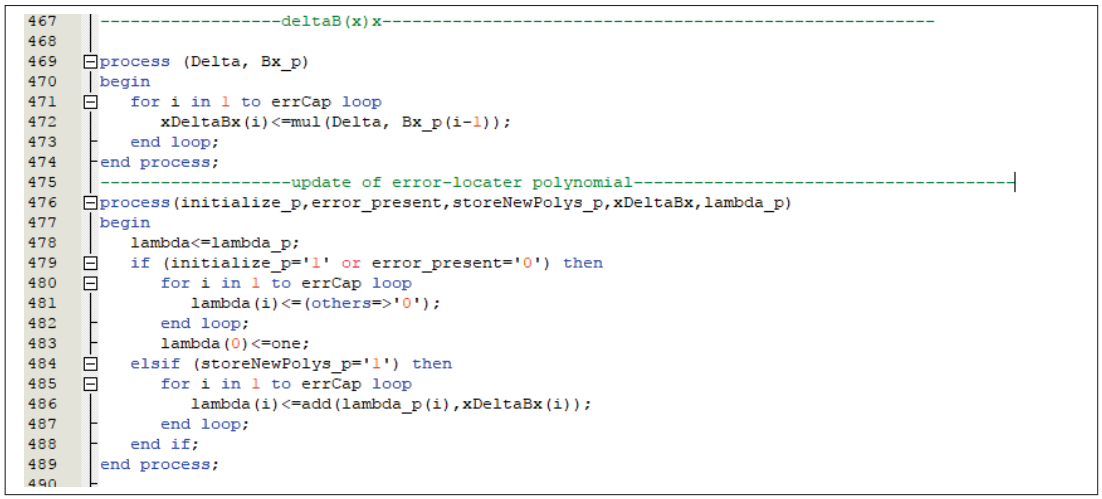
Figure 4–46 shows the final output process. The error-locator polynomial "**lambda_poly**" is acquired by connecting every element of the signal "**lambda_p**". The error-evaluator polynomial "**omega_poly**" is obtained by connecting every element except the first one of the signal "**omega_p**", because the element with zero index is taken as the coefficient associated with x^{-1}, which is mentioned in signal declaration.

LowLatRSdec_n255k225_p3_chienNcorrect.vhd. Figure 4–47 shows the port declaration. Besides the error-locator polynomial "**lambda_poly**" and the error-evaluator polynomial "**omega_poly**", "**startChien**" is another input to activate the processes of this component. Notice that this component records the received-words when they enter the decoder and the purpose is to reuse them for error correction and output generation. There are three outputs: "**dec_done_p**" which indicates the end of decoding algorithm, "**output_strobe**" which indicates the first correction-word output is ready, and "**data_out_p**" which is the correction-word output. With $p = 3$, three correction-words are generated at one clock.

Figure 4–48 shows the signal declaration. The figure does not include the constants: "**zero**", "**one**" and "**alpha**", because they are also defined in the syndrome component. For the basic parameters, "**lamLength**", "**derLemLength**" and "**omeLength**" are the number of coefficients in the error-locator polynomial, the derivative of error-locator polynomial, and the error-evaluator

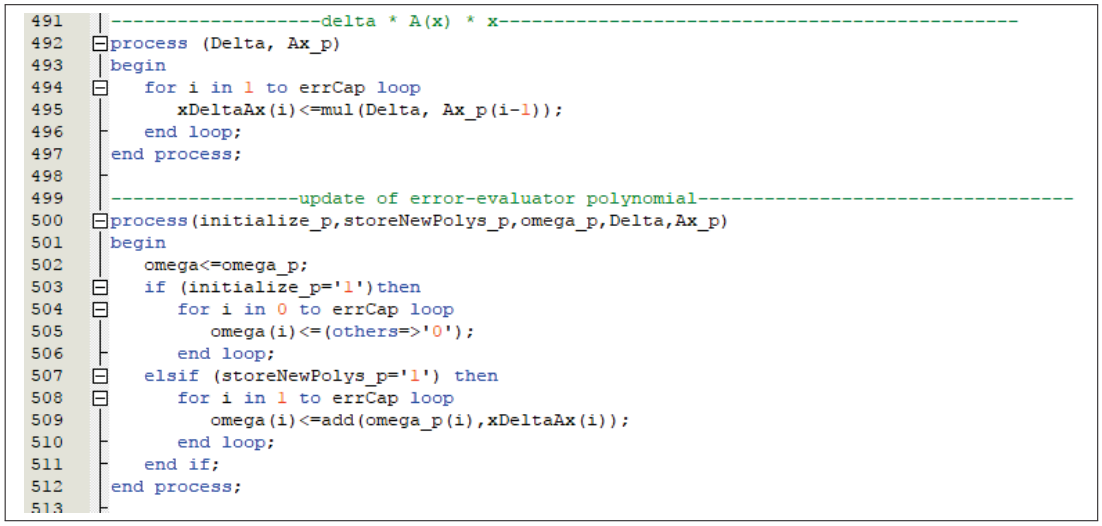
```
----------------------calculation of Delta---------------------------------------
process(initialize_p,SReg_p,storeNewPolys_p,syndrome)
begin
   SReg<=SReg_p;
   if (initialize_p='1') then
      for i in 0 to (errCap-1) loop
         SReg(i)<=zero;
      end loop;
      for i in 0 to (2*errCap-1) loop
         SReg(i+errCap)<=syndrome ((((i+1)*GFPower)-1) downto (i*GFPower));
      end loop;
   elsif (storeNewPolys_p='1') then
      for i in 0 to (3*errCap-2) loop
         SReg(i) <=SReg_p(i+1); --shift back
      end loop;
         SReg(3*errCap-1)<=zero;
   end if;
end process;

process (SReg_p,lambda_p)
begin
   for i in 0 to errCap loop
      convolution_Term(i)<=mul(SReg_p(errCap-i),lambda_p(i));
   end loop;
end process;

process(L_p)
begin
   case L_p is
      when "00000000" =>  --maximum L=15
         Convolution_term_multiplier<="0000000000000001"; --there are 16 iterms in Bx
      when "00000001" =>
         Convolution_term_multiplier<="0000000000000011";
      when "00000010" =>
         Convolution_term_multiplier<="0000000000000111";
      when "00000011" =>
         Convolution_term_multiplier<="0000000000001111";
      when "00000100" =>
         Convolution_term_multiplier<="0000000000011111";
      when "00000101" =>
         Convolution_term_multiplier<="0000000000111111";
      when "00000110" =>
         Convolution_term_multiplier<="0000000001111111";
      when "00000111" =>
         Convolution_term_multiplier<="0000000011111111";
      when "00001000" =>
         Convolution_term_multiplier<="0000000111111111";
      when "00001001" =>
         Convolution_term_multiplier<="0000001111111111";
      when "00001010" =>
         Convolution_term_multiplier<="0000011111111111";
      when "00001011" =>
         Convolution_term_multiplier<="0000111111111111";
      when "00001100" =>
         Convolution_term_multiplier<="0001111111111111";
      when "00001101" =>
         Convolution_term_multiplier<="0011111111111111";
      when "00001110" =>
         Convolution_term_multiplier<="0111111111111111";
      when "00001111" =>
         Convolution_term_multiplier<="1111111111111111";
      when others =>
         Convolution_term_multiplier<="1111111111111111";
   end case;
end process;

process(Convolution_term_multiplier,convolution_Term)
begin
   for i in 0 to errCap loop
      post_convolution_term(i)<=convolution_term_mul(convolution_Term(i),Convolution_term_multiplier(i));
   end loop;
end process;

process(post_convolution_term)
begin
   Delta<=add(post_convolution_term(0),add(post_convolution_term(1),add(post_convolution_term(2),
            add(post_convolution_term(3),add(post_convolution_term(4),add(post_convolution_term(5),
            add(post_convolution_term(6),add(post_convolution_term(7),add(post_convolution_term(8),
            add(post_convolution_term(9),add(post_convolution_term(10),add(post_convolution_term(11),
            add(post_convolution_term(12),add(post_convolution_term(13),add(post_convolution_term(14),
            post_convolution_term(15))))))))))))));
end process;
```

Figure 4–42: Delta calculation of BM component.



```
467   ------------------deltaB(x)x-----------------------------------------------------------
468
469   process (Delta, Bx_p)
470   begin
471      for i in 1 to errCap loop
472         xDeltaBx(i)<=mul(Delta, Bx_p(i-1));
473      end loop;
474   end process;
475   --------------------update of error-locater polynomial-------------------------------------
476   process(initialize_p,error_present,storeNewPolys_p,xDeltaBx,lambda_p)
477   begin
478      lambda<=lambda_p;
479      if (initialize_p='1' or error_present='0') then
480         for i in 1 to errCap loop
481            lambda(i)<=(others=>'0');
482         end loop;
483         lambda(0)<=one;
484      elsif (storeNewPolys_p='1') then
485         for i in 1 to errCap loop
486            lambda(i)<=add(lambda_p(i),xDeltaBx(i));
487         end loop;
488      end if;
489   end process;
```

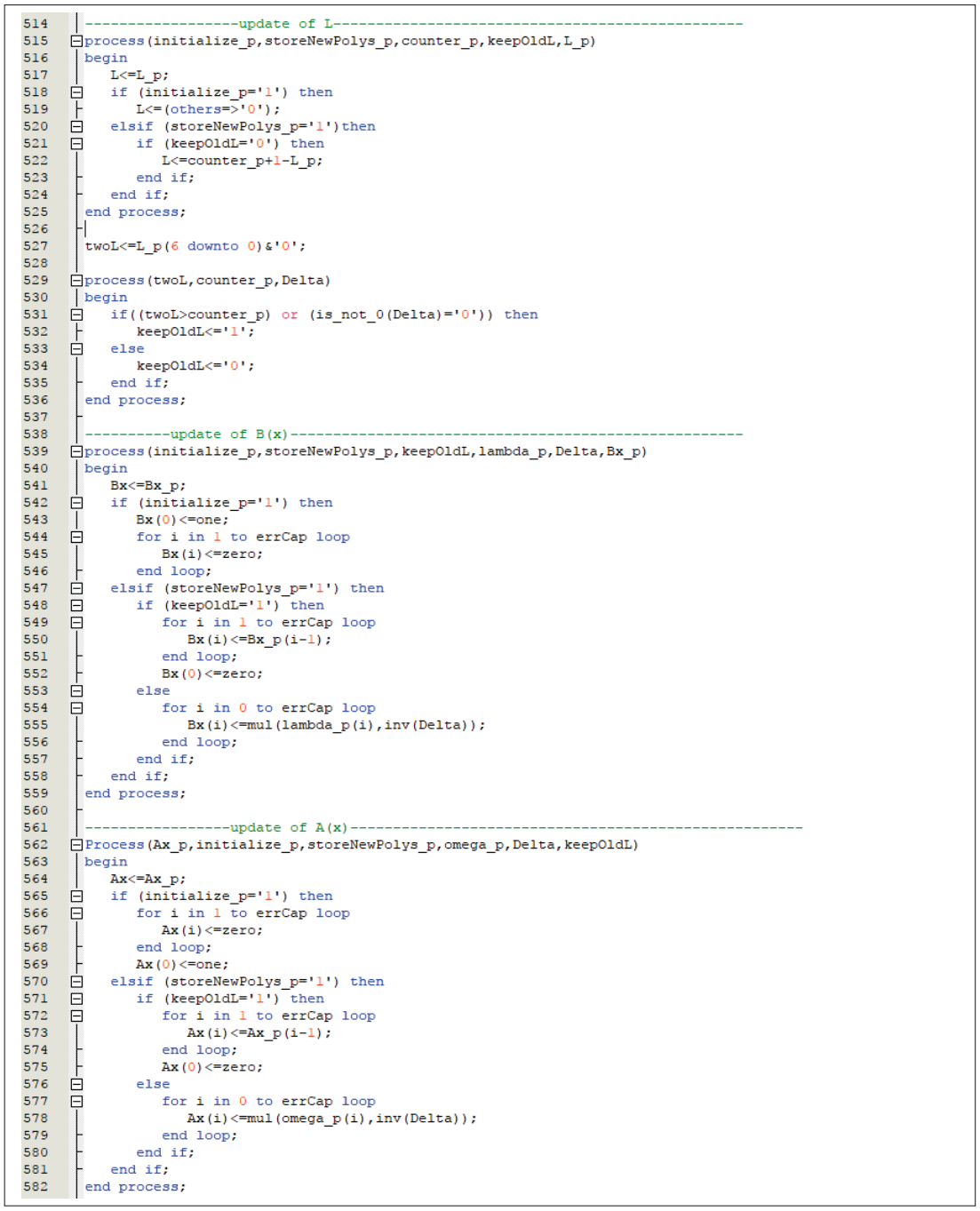
Figure 4–43: Update of error-locator polynomial of BM component.



```
491   --------------------delta * A(x) * x-----------------------------------------------
492   process (Delta, Ax_p)
493   begin
494      for i in 1 to errCap loop
495         xDeltaAx(i)<=mul(Delta, Ax_p(i-1));
496      end loop;
497   end process;
498
499   -----------------update of error-evaluator polynomial----------------------------------
500   process(initialize_p,storeNewPolys_p,omega_p,Delta,Ax_p)
501   begin
502      omega<=omega_p;
503      if (initialize_p='1')then
504         for i in 0 to errCap loop
505            omega(i)<=(others=>'0');
506         end loop;
507      elsif (storeNewPolys_p='1') then
508         for i in 1 to errCap loop
509            omega(i)<=add(omega_p(i),xDeltaAx(i));
510         end loop;
511      end if;
512   end process;
```

Figure 4–44: Update of error-evaluator polynomial of BM component.

polynomial, respectively. "**chienLength**" refers to the number of coefficients used in the Chien search. It should be equal to "**lamLength**", but in order to simplify the codes, it is forced to be divisible by $p = 3$ by adding zero coefficients. Therefore, the "**chienLength**" is 18 instead of 16 in this case. For the data-signal part, the signals with suffix "**_product**" refer to the single-term multiplication in the process of polynomial evaluations. The signals with suffix "**_sum**" refer to the final results of corresponding polynomial evaluations. The specific structures are shown in Figure 4–49, 4–50 and



```
514   ------------------update of L-------------------------------------------------
515   process(initialize_p,storeNewPolys_p,counter_p,keepOldL,L_p)
516   begin
517      L<=L_p;
518      if (initialize_p='1') then
519         L<=(others=>'0');
520      elsif (storeNewPolys_p='1')then
521         if (keepOldL='0') then
522            L<=counter_p+1-L_p;
523         end if;
524      end if;
525   end process;
526
527   twoL<=L_p(6 downto 0)&'0';
528
529   process(twoL,counter_p,Delta)
530   begin
531      if((twoL>counter_p) or (is_not_0(Delta)='0')) then
532         keepOldL<='1';
533      else
534         keepOldL<='0';
535      end if;
536   end process;
537
538   ----------update of B(x)-----------------------------------------------------
539   process(initialize_p,storeNewPolys_p,keepOldL,lambda_p,Delta,Bx_p)
540   begin
541      Bx<=Bx_p;
542      if (initialize_p='1') then
543         Bx(0)<=one;
544         for i in 1 to errCap loop
545            Bx(i)<=zero;
546         end loop;
547      elsif (storeNewPolys_p='1') then
548         if (keepOldL='1') then
549            for i in 1 to errCap loop
550               Bx(i)<=Bx_p(i-1);
551            end loop;
552            Bx(0)<=zero;
553         else
554            for i in 0 to errCap loop
555               Bx(i)<=mul(lambda_p(i),inv(Delta));
556            end loop;
557         end if;
558      end if;
559   end process;
560
561   -----------------update of A(x)-----------------------------------------------------
562   Process(Ax_p,initialize_p,storeNewPolys_p,omega_p,Delta,keepOldL)
563   begin
564      Ax<=Ax_p;
565      if (initialize_p='1') then
566         for i in 1 to errCap loop
567            Ax(i)<=zero;
568         end loop;
569         Ax(0)<=one;
570      elsif (storeNewPolys_p='1') then
571         if (keepOldL='1') then
572            for i in 1 to errCap loop
573               Ax(i)<=Ax_p(i-1);
574            end loop;
575            Ax(0)<=zero;
576         else
577            for i in 0 to errCap loop
578               Ax(i)<=mul(omega_p(i),inv(Delta));
579            end loop;
580         end if;
581      end if;
582   end process;
```

Figure 4–45: Update of other supporting signals of BM component.

4–51. "**DR_counter**" is an inner counter for data recording process. While "**counter**" is the formal counter for the subject component. For the control-signal part, when "**xferdata**" is 1, the decoder output a vector containing

```
584   ------------------final output-----------------------------------------------
585   omega_poly<=omega_p(15) & omega_p(14) & omega_p(13) & omega_p(12) & omega_p(11) & omega_p(10) & omega_p(9)
586         & omega_p(8) & omega_p(7) & omega_p(6) & omega_p(5) & omega_p(4) & omega_p(3) & omega_p(2) & omega_p(1);
587
588   lambda_poly<=lambda_p(15) & lambda_p(14) & lambda_p(13) & lambda_p(12) & lambda_p(11) & lambda_p(10) &
589                 lambda_p(9) & lambda_p(8) & lambda_p(7) & lambda_p(6) & lambda_p(5) & lambda_p(4) &
590                 lambda_p(3) & lambda_p(2) & lambda_p(1) & lambda_p(0);
591
592   end architecture;
593
```

Figure 4–46: Final output of BM component.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity LowLatRSdec_n255k225_p3_chienNcorrect is
6   port (reset: in std_logic;
7       clk: in std_logic;
8       data_input_strobe:in std_logic;
9       startChien: in std_logic;
10      omega_poly: in std_logic_vector(119 downto 0);
11      lambda_poly: in std_logic_vector(127 downto 0);
12      data_in:in std_logic_vector(23 downto 0);
13      dec_done_p:out std_logic;
14      output_strobe:out std_logic;
15      data_out_p:out std_logic_vector(23 downto 0));
16  end;
```

Figure 4–47: Port declaration of Chien search and correction component.

three correction-words at each clock. More details and explanation can be found in the figures.

For functions declaration, as addition, multiplication, inversion, and IDFT have been covered already in the previous section, they are not shown here again.

Figure 4–52 shows the register update. Same as before, at each clock's rising edge, all register signals go to zero, when "**reset**" is high. Otherwise, the register signals are updated to their combinational logic counterparts.

Figure 4–53 shows the finite state machine. There are five states. When "**data_input_strobe**" is up, "**recorddata**" becomes 1 which enables copying the input received-words, and the state goes from "**Idle**" to "**DataRecording**". The data recording process finishes when the inner counter "**DR_counter**" reach $m - 1 = 84$, because all received-words enters in 85 clock cycles. Then the state is back to "**Idle**". When "**startChien**" turns to 1, "**initialize**" is up to set up all signals and the state goes to "**Initialization**". The

```
architecture RTL of LowLatRSdec_n255k225_p3_chienNcorrect is
----------basic parameters------------------------------------------------------------
constant GFPower: integer:=8;--to define GF(2^8)
constant N: integer:=255; --codeword length
constant K: integer:=225; --message word length
constant R: integer:=30; --parity bits length =255-225
constant errCap: integer:=R/2; --error-correction capacity
constant m: integer:=85; --m=N/p=255/3=85
constant t: integer:=75; --t=K/p=225/3=75
constant L: integer:=10; --L=R/p=30/3=10
constant p: integer:=3;--speed-up coefficient
constant lamLength: integer:=16;
constant derLamLength: integer:=8;
constant omeLength: integer:=15;
constant chienLength: integer:=18;
--the real value for chienLength is 16,
--but force it to be divisible by p=3 for simplify VHDL codes
----------------pre-defined useful types-----------------------------------------------------
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
type alpha_type is array (0 to 254) of Galois_Field_element;
type state_type is (Idle, DataRecording, Initialization, Calculation, WrapUp);
type FTout_type is array (0 to (p-1)) of Galois_Field_element;
type data_delay_type is array (m-1 downto 0) of std_logic_vector(p*GFPower-1 downto 0);
type chienProd_type is array (0 to ChienLength-1) of Galois_Field_element;
type dLambProd_subtype is array (0 to derLamLength-1) of Galois_Field_element;
type omegProd_subtype is array (0 to omeLength-1) of Galois_Field_element;
type dLambProd_type is array (0 to p-1) of dLambProd_subtype;
type omegProd_type is array (0 to p-1) of omegProd_subtype;
type ChienSum_type is array (0 to p-1) of Galois_Field_element;
type dLambSum_type is array (0 to p-1) of Galois_Field_element;
type omegSum_type is array (0 to p-1) of Galois_Field_element;
type correction_type is array (0 to p-1) of Galois_Field_element;
type u_type is array (0 to LamLength-1) of Galois_Field_element;
type v_type is array (0 to omeLength-1) of Galois_Field_element;

--------------------data signals-----------------------------------
signal data_delay,data_delay_p: data_delay_type;
--record the received words for later chien search and correction process.
signal data_out_int,data_out_int_p:data_delay_type;
signal dec_done: std_logic;
signal data_out: std_logic_vector(GFPower*p-1 downto 0);
signal DR_counter,DR_counter_p:integer range 0 to m:=0;--counter for recording the received words
signal counter,counter_p:integer range 0 to m:=0;--counter for chien search and correction process
signal FTout: FTout_type; --fourier transform results using in chien search
signal u:u_type; --a vector containing all coefficient of error-locator polynomial
signal v:v_type; --a vector containing all coefficient of error-evaluator polynomial
--each product represents a term evaluation in corresponding polynomial equation
signal chien_product,chien_product_p: ChienProd_type; --for chien search
signal dLamd_product,dLamb_product_p: dLambProd_type;
--for odd terms of error locator polynomial evaluation
signal omeg_product,omeg_product_p: omegProd_type;
-- sum is the polynomial evaluation result
signal Chien_sum:ChienSum_type; --sum of chienProds. Basically, it's evaluation of the polynomial
signal dLamd_sum:dLambSum_type;
-- sum of dLamb_products => evaluation of the odd terms of error-locater polynomial
signal omeg_sum:omegSum_type;
-- sum of omeg_products => evaluation of the error-evaluator polynomial
signal correction:correction_type;
-- equals to error value + received-word
------------------control signals-------------------------------------------------
signal state,state_p: state_type;--used in Finite State Machine
signal initialize,initialize_p:std_logic; --to set up signals
signal DoCalc,DoCalc_p:std_logic; --enable error-location and error-value calculation
signal dec_almost_done,dec_almost_done_p:std_logic;--one clock early than the end of decoding process
signal Detector0En,Detector0En_p:std_logic; --enable zero detection on FTout
signal recorddata,recorddata_p:std_logic; --record received-words
signal countEn,countEn_p:std_logic; --enable signal for counter
signal xferdata,xferdata_p:std_logic;--start output transfer
```

Figure 4–48: Signal declaration of Chien search and correction component.

"**Initialization**" state lasts only for one clock cycle and turns on "**countEn**" which enables counter's increment, "**DoCalc**" which enables calculation, and "**Deteror0En**" which enables zero detection for the output of IDFT. Meanwhile, the state goes to "**Calculation**". When "**counter_p**" reaches $m-2 = 83$,

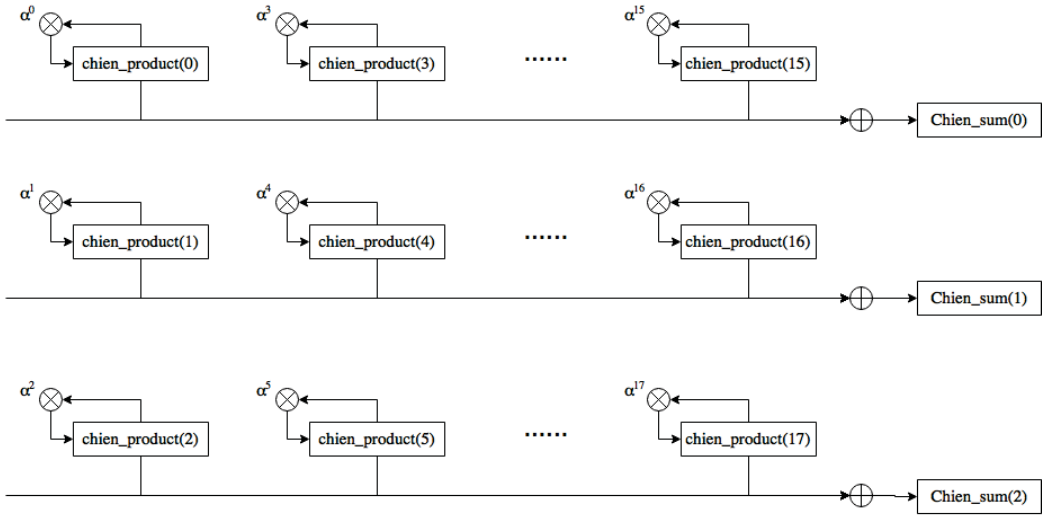


Figure 4–49: Evaluation block of the modified Chien search

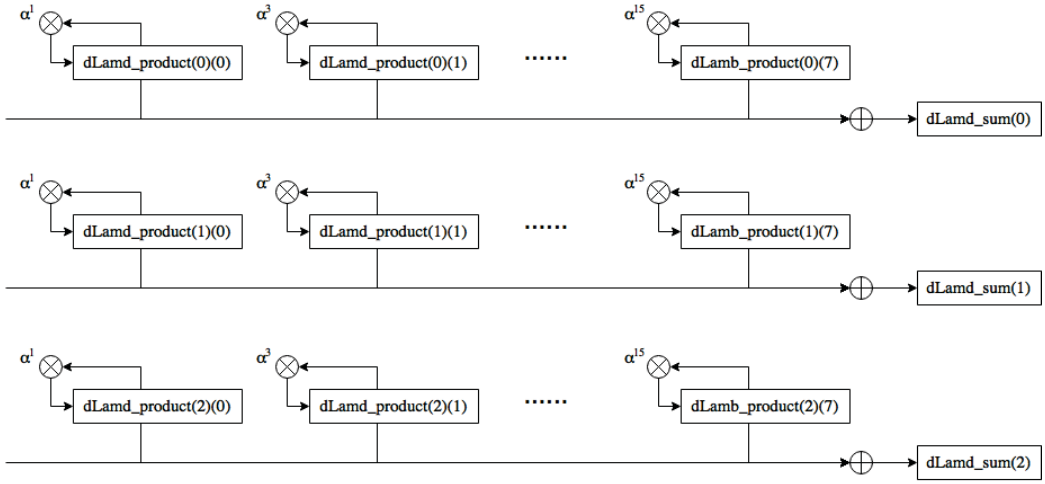


Figure 4–50: Evaluation block of the odd terms of error-locator polynomial.

"**DoCalc**" turns off and the state enters "**WrapUp**" in which all state control signals goes off except "**dec_almost_done**". After one clock cycle, the state goes back to "**Idle**".

Figure 4–54 shows the data-recording process. While the inner counter "**DR_counter**" counts from 0 to 84, the input received-words are copied by "**data_delay**".

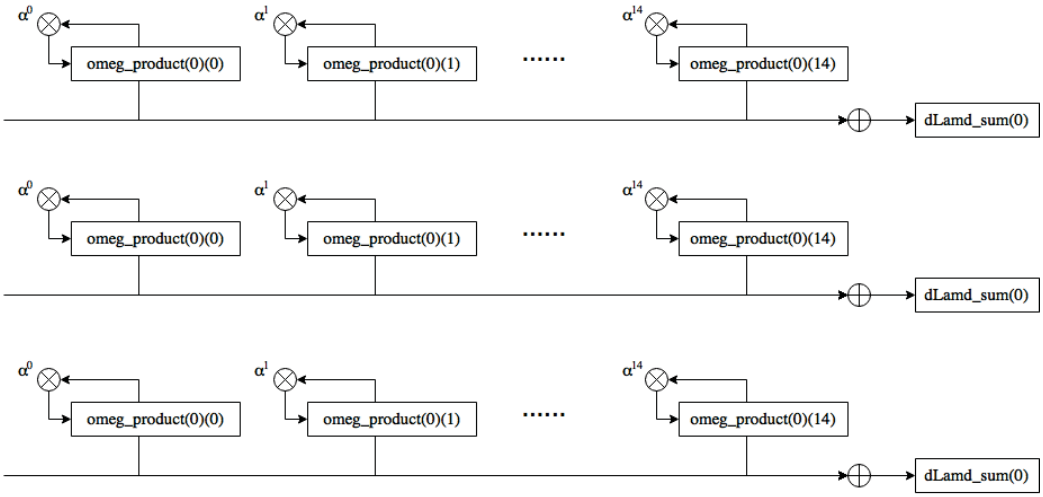


Figure 4–51: Evaluation block of the error-evaluator polynomial.

Figure 4–55 shows the behavior of formal counter for Chien search and correction process. The counter is reset to zero when the process starts and ends.

Figure 4–56 shows the coefficient extraction of the error-locator polynomial and the error-evaluator polynomial.

Figure 4–57 shows the computation of modified Chien search. The polynomial evaluation is implemented just like the way shown in Figure 4–49 and then it is followed by an inverse Fourier transform.

Figure 4–58 shows the evaluation of the odd terms of the error-locator polynomial following the way in Figure 4–50 and an evaluation of the error-evaluator polynomial follows the way shown in Figure 4–51. Both of these evaluations are used for error-value calculation. Recall Step 6 in Section 3.2.2, the odd terms of the error-locator polynomial is equal to the derivative of error-locator polynomial multiplied by x.

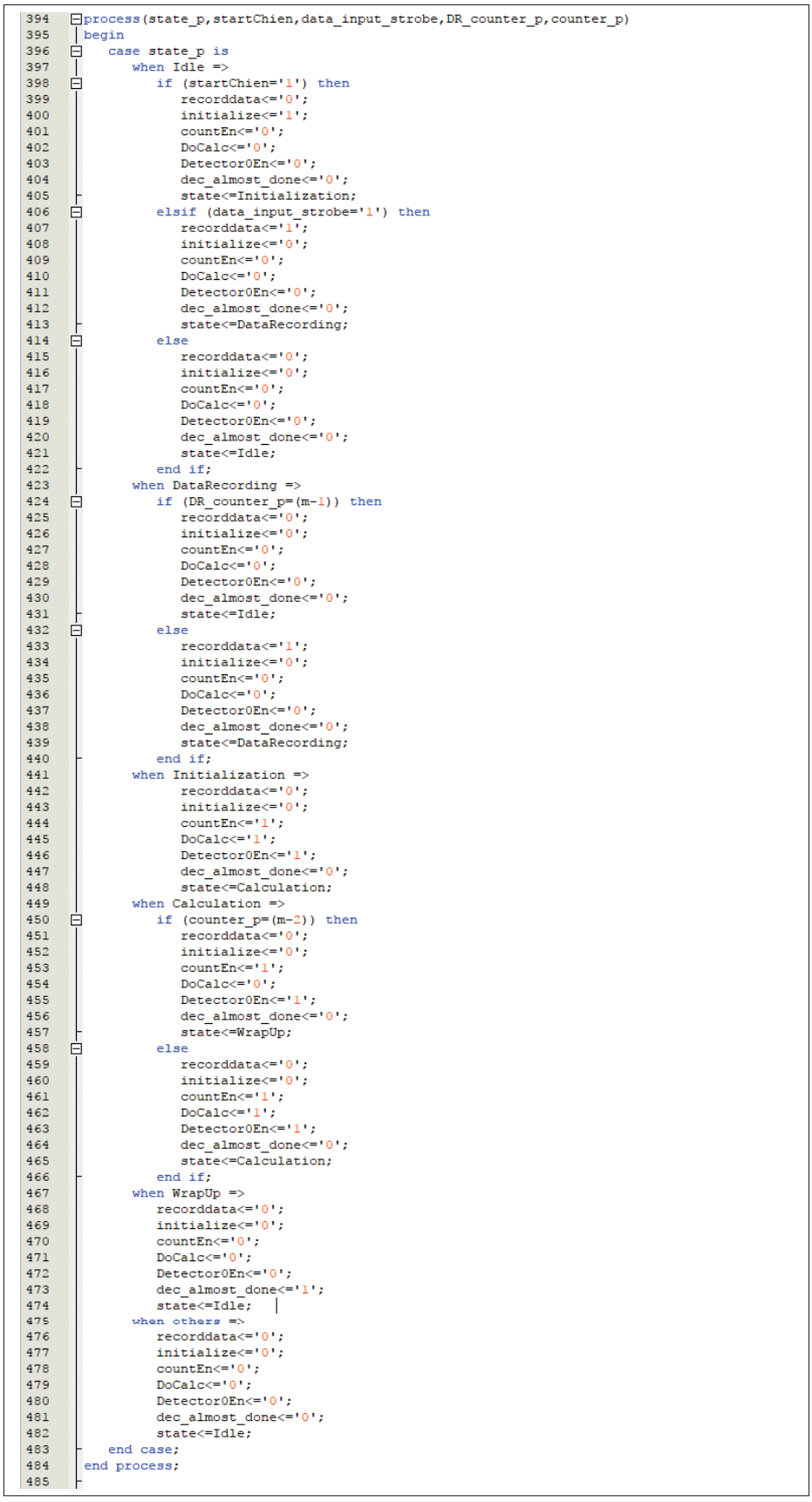
Figure 4–59 shows the correction process. For every three error location checked in one clock, three corresponding correction values are calculated, but

```
336   ------------------------register update------------------------------------------------------------
337   process(clk)
338   begin
339      if (clk'event and clk='1') then
340         if (reset='1') then
341            recorddata_p<='0';
342            initialize_p<='0';
343            countEn_p<='0';
344            DoCalc_p<='0';
345            Detector0En_p<='0';
346            dec_almost_done_p<='0';
347            state_p<=Idle;
348            DR_counter_p<=0;
349            for i in 0 to (m-1) loop
350               data_delay_p(i)<=(others=>'0');
351            end loop;
352            counter_p<=0;
353            for i in 0 to chienLength-1 loop
354               Chien_product_p(i)<=(others=>'0');
355            end loop;
356            for i in 0 to p-1 loop
357               for j in 0 to derLamLength-1 loop
358                  dLamb_product_p(i)(j)<=(others=>'0');
359               end loop;
360            end loop;
361            for i in 0 to p-1 loop
362               for j in 0 to omeLength-1 loop
363                  omeg_product_p(i)(j)<=(others=>'0');
364               end loop;
365            end loop;
366            for i in 0 to (m-1) loop
367               data_out_int_p(i)<=(others=>'0');
368            end loop;
369            xferdata_p<='0';
370            data_out_p<=(others=>'0');
371            dec_done_p<='0';
372         else
373            recorddata_p<=recorddata;
374            initialize_p<=initialize;
375            countEn_p<=countEn;
376            DoCalc_p<=DoCalc;
377            Detector0En_p<=Detector0En;
378            dec_almost_done_p<=dec_almost_done;
379            state_p<=state;
380            DR_counter_p<=DR_counter;
381            data_delay_p<=data_delay;
382            counter_p<=counter;
383            Chien_product_p<=Chien_product;
384            dLamb_product_p<=dLamd_product;
385            omeg_product_p<=omeg_product;
386            data_out_int_p<=data_out_int;
387            xferdata_p<=xferdata;
388            data_out_p<=data_out;
389            dec_done_p<=dec_done;
390         end if;
391      end if;
392   end process;
```

Figure 4–52: Register update of Chien search and correction component.

they are used as intermediate outputs only when the corresponding locations are determined by zero detection for IDFT outputs.

Figure 4–60 shows the final output process of decoder. "**xferdata_p**" is just one-clock delayed "**Detector0En_p**" and when it is 1, "**data_out**", the combinational logic counter part of final output, is assigned three correction-words at each clock. "**output_strobe**" is a output signal to indicate the start



```
394  process(state_p,startChien,data_input_strobe,DR_counter_p,counter_p)
395  begin
396     case state_p is
397        when Idle =>
398           if (startChien='1') then
399              recorddata<='0';
400              initialize<='1';
401              countEn<='0';
402              DoCalc<='0';
403              Detector0En<='0';
404              dec_almost_done<='0';
405              state<=Initialization;
406           elsif (data_input_strobe='1') then
407              recorddata<='1';
408              initialize<='0';
409              countEn<='0';
410              DoCalc<='0';
411              Detector0En<='0';
412              dec_almost_done<='0';
413              state<=DataRecording;
414           else
415              recorddata<='0';
416              initialize<='0';
417              countEn<='0';
418              DoCalc<='0';
419              Detector0En<='0';
420              dec_almost_done<='0';
421              state<=Idle;
422           end if;
423        when DataRecording =>
424           if (DR_counter_p=(m-1)) then
425              recorddata<='0';
426              initialize<='0';
427              countEn<='0';
428              DoCalc<='0';
429              Detector0En<='0';
430              dec_almost_done<='0';
431              state<=Idle;
432           else
433              recorddata<='1';
434              initialize<='0';
435              countEn<='0';
436              DoCalc<='0';
437              Detector0En<='0';
438              dec_almost_done<='0';
439              state<=DataRecording;
440           end if;
441        when Initialization =>
442              recorddata<='0';
443              initialize<='0';
444              countEn<='1';
445              DoCalc<='1';
446              Detector0En<='1';
447              dec_almost_done<='0';
448              state<=Calculation;
449        when Calculation =>
450           if (counter_p=(m-2)) then
451              recorddata<='0';
452              initialize<='0';
453              countEn<='1';
454              DoCalc<='0';
455              Detector0En<='1';
456              dec_almost_done<='0';
457              state<=WrapUp;
458           else
459              recorddata<='0';
460              initialize<='0';
461              countEn<='1';
462              DoCalc<='1';
463              Detector0En<='1';
464              dec_almost_done<='0';
465              state<=Calculation;
466           end if;
467        when WrapUp =>
468           recorddata<='0';
469           initialize<='0';
470           countEn<='0';
471           DoCalc<='0';
472           Detector0En<='0';
473           dec_almost_done<='1';
474           state<=Idle;
475        when others =>
476           recorddata<='0';
477           initialize<='0';
478           countEn<='0';
479           DoCalc<='0';
480           Detector0En<='0';
481           dec_almost_done<='0';
482           state<=Idle;
483     end case;
484  end process;
485
```

Figure 4–53: Finite state machine of Chien search and correction component.

```
486   ----------------------data recording process----------------------------------------
487   process(data_input_strobe,recorddata_p,DR_counter_p)
488   begin
489      if(data_input_strobe='1') then
490         DR_counter<=0;
491      elsif (recorddata_p='1') then
492         DR_counter<=DR_counter_p+1;
493      else
494         DR_counter<=0;
495      end if;
496   end process;
497
498   process(data_delay_p,data_in,recorddata_p,DR_counter_p)
499   begin
500      data_delay<=data_delay_p;
501      if (recorddata_p='1') then
502         data_delay(m-1-DR_counter_p)<=data_in;
503      end if;
504   end process;
505
```

Figure 4–54: Data recording of Chien search and correction component.

```
506   ----------------------------counter behavior-------------------------------------------------
507   process(counter_p,initialize_p,dec_almost_done_p,countEn_p)
508   begin
509      if (initialize_p='1') then
510         counter<=0;
511      elsif (dec_almost_done_p='1') then
512         counter<=0;
513      elsif (countEn_p='1') then
514         counter<=counter_p+1;
515      else
516         counter<=0;
517      end if;
518   end process;
519
```

Figure 4–55: Counter behavior of Chien search and correction component.

```
520   ----------------polynomial-coefficient extraction-------------------------------------
521   --assign error-locator polynomial coefficients to u
522   process(lambda_poly)
523   begin
524      for i in 0 to lamlength-1 loop
525         u(i)<=lambda_poly((i+1)*GFPower-1 downto i*GFPower);
526      end loop;
527   end process;
528
529   --assign error-evaluator polynomial coefficients to v
530   process(omega_poly)
531   begin
532      for i in 0 to omeLength-1 loop
533         v(i)<=omega_poly((i+1)*GFPower-1 downto i*GFPower);
534      end loop;
535   end process;
```

Figure 4–56: Coefficient extraction of Chien search and correction component.

of the final correction-words output, whereas "**dec_done**" is a output signal indicating the end.

LowLatRSdec_n255k225_p3.vhd. Figure 4–61 shows the entire codes for the top file. There are four inputs: the global reset denoted as "**reset_n**", the clock signal denoted as "**clk**", the received-words denoted as

```
537   --------------------modified Chine search------------------------------------------------------------------
538   process(initialize_p,DoCalc_p,Chien_product_p,u)
539   begin
540         Chien_product<=Chien_product_p;
541
542      if (initialize_p='1') then
543         for i in 0 to 15 loop
544            Chien_product(i)<=mul(u(i),alpha((N-m+1)*i rem N));
545         end loop;
546
547      elsif (DoCalc_p='1') then
548         for j in 0 to 15 loop
549            Chien_product(j)<=mul(Chien_product_p(j),alpha(j));
550         end loop;
551      end if;
552   end process;
553
554   process (Chien_product_p)
555   begin
556      for i in 0 to (p-1) loop
557         Chien_sum(i)<=add(add(Chien_product_p(i),Chien_product_p(i+p)),add(add(Chien_product_p(i+2*p),
558                        Chien_product_p(i+3*p)),add(Chien_product_p(i+4*p),Chien_product_p(i+5*p))));
559      end loop;
560   end process;
561
562   FTout<= IDFT(Chien_sum(0),Chien_sum(1),Chien_sum(2));
563
```

Figure 4–57: Modified Chien search of Chien search and correction component.

"**data_in**" and the decoding activation signal denoted as "**input_strobe**". There are five outputs: a signal "**output_strobe**" that suggests the output is ready, the correction-words "**data_out**" and three component-process-ending indicators: "**synd_calc_done_out**", "**startChien_out**" and "**dec_done**".

The signal and component declaration are both presented. The whole architecture only includes the assignment of three process-ending indicator signals and the three component instantiations connected with each other following the structure shown in Figure 4–28.

4.4.2 Implementations with Speed-Up Coefficient $p = 5$ and $p = 15$

Both of low-latency $RS(255, 225)$ decoders with $p = 5$ and $p = 15$ follows the same implementation logic and style. In this subsection, only the difference on implementation is listed and discussed. VHDL codes of the different parts are shown in Appendix B.

Syndrome Component. First, as p changes, the size of "**data_in**" is different, and it is always equal to $(p{\cdot}GFPower)$. Second, the basic parameters have to be revised based on p. This change actually adjusts most of the parts in this component, correspondingly. Third, DFT function has to be changed

```
564   ------------------------------Evaluation of the odd terms of Error Locator polynomial------------------
565   process(initialize_p,DoCalc_p,dLamb_product_p,u)
566   begin
567      dLamd_product<=dLamb_product_p;
568      if (initialize_p='1') then
569         for i in 0 to p-1 loop
570            for j in 0 to DerLamLength-1 loop
571               dLamd_product(i)(j)<=mul(u(2*j+1),alpha((((N-m+1)-i*m)*(2*j+1)) rem N));
572            end loop;
573         end loop;
574      elsif (DoCalc_p='1') then
575         for i in 0 to p-1 loop
576            for j in 0 to DerLamLength-1 loop
577               dLamd_product (i)(j)<=mul(dLamb_product_p(i)(j),alpha(2*j+1));
578            end loop;
579         end loop;
580      end if;
581   end process;
582
583   process(dLamb_product_p)
584   begin
585      for i in 0 to p-1 loop
586         dLamd_sum(i)<=add(add(add(dLamb_product_p(i)(0),dLamb_product_p(i)(1)),
587                           add(dLamb_product_p(i)(2),dLamb_product_p(i)(3))),add(add(dLamb_product_p(i)(4),
588                           dLamb_product_p(i)(5)),add(dLamb_product_p(i)(6),dLamb_product_p(i)(7))));
589      end loop;
590   end process;
591
592   --------------------Evaluation of Error Evaluator Polynomials----------------------------------------
593   process(initialize_p,DoCalc_p,omeg_product_p,v)
594   begin
595      omeg_product<=omeg_product_p;
596      if (initialize_p='1') then
597         for i in 0 to p-1 loop
598            for j in 0 to omeLength-1 loop
599               omeg_product(i)(j)<=mul(v(j),alpha((((N-m+1)-i*m)*j) rem N));
600            end loop;
601         end loop;
602      elsif (DoCalc_p='1') then
603         for i in 0 to p-1 loop
604            for j in 0 to omeLength-1 loop
605               omeg_product (i)(j)<=mul(omeg_product_p(i)(j),alpha(j));
606            end loop;
607         end loop;
608      end if;
609   end process;
610
611   process(omeg_product_p)
612   begin
613      for i in 0 to p-1 loop
614         omeg_sum(i)<=add(add(add(add(omeg_product_p(i)(0),omeg_product_p(i)(1)),
615                           add(omeg_product_p(i)(2),omeg_product_p(i)(3))),
616                           add(add(omeg_product_p(i)(4),omeg_product_p(i)(5)),
617                           add(omeg_product_p(i)(6),omeg_product_p(i)(7)))),
618                           add(add(add(omeg_product_p(i)(8),omeg_product_p(i)(9)),
619                           add(omeg_product_p(i)(10),omeg_product_p(i)(11))),
620                           add(add(omeg_product_p(i)(12),omeg_product_p(i)(13)),
621                           omeg_product_p(i)(14))));
622      end loop;
623   end process;
```

Figure 4–58: Polynomial evaluation of Chien search and correction component.

following the formula $DFTout(i) \equiv \Sigma_{j=0}^{p-1} \rho^{i \cdot j} L_j$. Forth, the expressions for final outputs "**syndrome**" and "**error_present**" have to be changed due to the size change of "**intS_p**" and "**intEP**".

Berlekamp-Massey Algorithm Component. As this component is independent on p, nothing needs to be changed.

Chien Search and Correction Component. First, as p changes, the sizes of "**data_in**" and "**data_out_p**" need to be revised. Both of them

```
624   --------------------correction evaluation by adding error value to delayed message-words-----------
625   process(dLamd_sum,omeg_sum,data_out_int_p,counter_p)
626   begin
627      for i in 0 to p-1 loop
628         correction(i)<=add(mul(inv(dLamd_sum(i)),omeg_sum(i)),
629                           data_out_int_p(m-1-counter_p)((i+1)*GFPower-1 downto i*GFPower));
630      end loop;
631   end process;
632
633   ---------------------define intermediate output-----------------------------------------------
634   process(data_delay_p,data_out_int_p,initialize_p,Detector0En_p,FTout,correction,counter_p)
635   begin
636      data_out_int<=data_out_int_p;
637      if (initialize_p='1') then
638         data_out_int<=data_delay_p;
639
640      elsif(Detector0En_p='1' and (FTout(0)=zero or FTout(1)=zero or FTout(2)=zero))then
641         for i in 0 to p-1 loop
642            if (FTout(i)=zero) then
643               data_out_int(m-1-counter_p)((i+1)*GFPower-1 downto (i*GFPower))<=correction(i);
644            end if;
645         end loop;
646      end if;
647   end process;
648
649   ------------------------------start Xfering data_out-------------------------------------------
```

Figure 4–59: Correction process of Chien search and correction component.

```
649   ------------------------------start Xfering data_out-------------------------------------------
650   xferdata<=Detector0En_p; --so xferdata_p is one clock delayed DetectorEn_p
651
652   -----------------------------------final data_out-----------------------------------------------
653   process(xferdata_p,data_out_int_p,counter_p)
654   begin
655      if (xferdata_p='1') then
656         data_out<=data_out_int_p(m-counter_p);
657      else
658         data_out<=(others=>'0');
659      end if;
660   end process;
661
662   --------------------------------output_strobe-----------------------------------------------------
663   process(counter_p)
664   begin
665      if (counter_p=1) then
666         output_strobe<='1';
667      else
668         output_strobe<='0';
669      end if;
670   end process;
671
672   -------------------------------dec_done-----------------------------------------------------------
673   dec_done<=dec_almost_done_p;
674   end architecture;
```

Figure 4–60: Final output process of Chien search and correction component.

are equal to $(p \cdot GFPower)$. Second, the basic parameters have to be revised based on p. Notice that, "**chienLength**" is always forced to be larger than "**lamLength**"(= 16) and divisible by p. Therefore, "**chienLength**" is equal to 20 for $p = 5$ and "**chienLength**" is equal to 30 for $p = 15$. The

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.std_logic_unsigned.all;
 4 entity LowLatRSdec_n255k225 is
 5 port (reset_n: in std_logic;
 6     clk: in std_logic;
 7     data_in: in std_logic_vector(23 downto 0); --three 8-bit received-words
 8     input_strobe: in std_logic;   --the signal which activates decoding process.
 9     output_strobe:out std_logic; --the signal which indicates outputs are ready to be sent out
10     data_out: out std_logic_vector(23 downto 0); --three 8-bit correction-words
11     synd_calc_done_out: out std_logic;
12     startChien_out: out std_logic;
13     dec_done:out std_logic);
14 end;
15 architecture RTL of LowLatRSdec_n255k225 is
16  signal synd_calc_done:std_logic; --indicates syndrome computing has been done
17  signal error_present:std_logic; --indicate if there is any error in received-words
18  signal syndrome: std_logic_vector (239 downto 0); --(s_r&s_r-1&s_r-2&...&s_1&s_0)
19  signal startChien:std_logic; --activate Chien Search
20  signal lambda_poly:std_logic_vector(127 downto 0); --error locator polynomial. MSB is on the Left
21  signal omega_poly:std_logic_vector(119 downto 0); --error evaluator polynomial. MSB is on the Left
22  signal dec_done_int:std_logic; --intermediate signal for dec_done
23 component LowLatRSdec_n255k225_p3_syn
24 port (reset: in std_logic;
25     data_in: in std_logic_vector(23 downto 0);
26     clk: in std_logic;
27     input_strobe: in std_logic;
28     synd_calc_done: out std_logic;
29     error_present_out:out std_logic;
30     syndrome_out: out std_logic_vector (239 downto 0)
31     );-- 30 syndromes and each syndrome has 8 bits=>total 240 bits
32  end component;
33 component LowLatRSdec_n255k225_p3_bm --key equation solver (berlekamp-Massey Algorithm)
34 port (reset: in std_logic;
35     clk: in std_logic;
36     synd_calc_done: in std_logic;
37     startChien_p: out std_logic;
38     error_present:in std_logic;
39     syndrome: in std_logic_vector (239 downto 0);--30*8=240
40     lambda_poly:out std_logic_vector(127 downto 0);--error locator polynomial 16*8=128
41     omega_poly: out std_logic_vector(119 downto 0) --error evaluator polynomial 15*8=120
42     );
43  end component;
44 component LowLatRSdec_n255k225_p3_chienNcorrect --chien search, error evaluation and correction
45 port (reset: in std_logic;
46     clk: in std_logic;
47     data_input_strobe:in std_logic;
48     startChien: in std_logic;
49     omega_poly: in std_logic_vector(119 downto 0);
50     lambda_poly: in std_logic_vector(127 downto 0);
51     data_in:in std_logic_vector(23 downto 0);
52     dec_done_p:out std_logic;
53     output_strobe:out std_logic;
54     data_out_p:out std_logic_vector(23 downto 0));
55  end component;
56  begin
57  syndrome_cal: LowLatRSdec_n255k225_p3_syn
58     port map(reset=>reset_n,
59           data_in=>data_in,
60           clk=>clk,
61           input_strobe=>input_strobe,
62           synd_calc_done=>synd_calc_done,
63           error_present_out=>error_present,
64           syndrome_out=>syndrome);
65  BMSolver: LowLatRSdec_n255k225_p3_bm
66     port map(reset=>reset_n,
67     clk=>clk,
68     synd_calc_done=>synd_calc_done,
69     startChien_p=>startChien,
70     error_present=>error_present,
71     syndrome=>syndrome,
72     lambda_poly=>lambda_poly,
73     omega_poly=>omega_poly);
74  ChienNCorrect: LowLatRSdec_n255k225_p3_chienNcorrect
75     port map(reset=>reset_n,
76     clk=>clk,
77     data_input_strobe=>input_strobe,
78     startChien=>startChien,
79     omega_poly=>omega_poly,
80     lambda_poly=>lambda_poly,
81     data_in=>data_in,
82     dec_done_p=>dec_done_int,
83     output_strobe=>output_strobe,
84     data_out_p=>data_out);
85
86  synd_calc_done_out<=synd_calc_done;
87  startChien_out<=startChien;
88  dec_done<=dec_done_int;
89 end architecture;
```

Figure 4–61: Top file of low-latency $RS(255, 225)$ decoder with $p = 3$.

change of the basic parameters actually adjusts most of parts in this component correspondingly. Third, IDFT function has to change following the formula $IDFTout(i) = \Sigma_{j=0}^{p-1} \rho^{-j \cdot i} f_j$. Forth, the expression of "**Chien_sum(i)**" needs to be revised due to the size change in "**chienLength**".

Top File. The only thing to revise is the sizes of input received-words and output correction words included in a port declaration and component instantiations.

4.5 Synthesis and Simulations Results for the Decoder

This section includes both FPGA synthesis results and ASIC synthesis results. For better explanation, the conventional RS decoder can be thought as a low-latency RS decoder with $p = 1$ which has no DFT and IDFT components.

The FPGA used for synthesis is an Altera DE2-115 board, specifically Cyclone IV E with device number "EP4CE115F23C8L". The software used for synthesis and simulations is Quartus II 15.0. The comparison of synthesis and performance results among the conventional decoder and low-latency decoders with different p is shown in Table 4–5. The functional simulation results for decoders with each p value are shown Figure 4–62, 4–63 and 4–64. The timing simulation results are shown in Appendix D. The results have been verified with the low-latency decoder implemented in MATLAB. For better presentation, all data signals are represented in hexadecimal form. The received-words are produced by adding random errors to the code-words generated in Figure 4–25, 4–26 and 4–27. These errors mimic the data corruption happened in transmissions due to the noise effect in the channel.

It is easy to see that the simulation results generated by low-latency decoders (Figure 4–62, 4–63 and 4–64) are consistent with Figure 4–25, 4–26 and 4–27. Moving on to Table 4–5, the clock cycles used by the low-latency decoders are dramatically decreased compared to the conventional one. It is

$RS(255, 225)$ Decoders with Error-Correction Capacity 15				
Implemented using Altera DE2-115 FPGA				
Decoder Name	Clock Cycles	Max Frequency (MHz)	Hardware Costs	
			Registers	Combinational Logic
Conventional RS Decoder	546	78.60	5791	7102
Low-Latency RS Decoder ($p = 3$)	206	78.77	5981	8667
Low-Latency RS Decoder ($p = 5$)	138	72.94	6363	10954
Low-Latency RS Decoder ($p = 15$)	71	69.74	7418	17639

Table 4–5: Comparison FPGA synthesis results for different decoders.

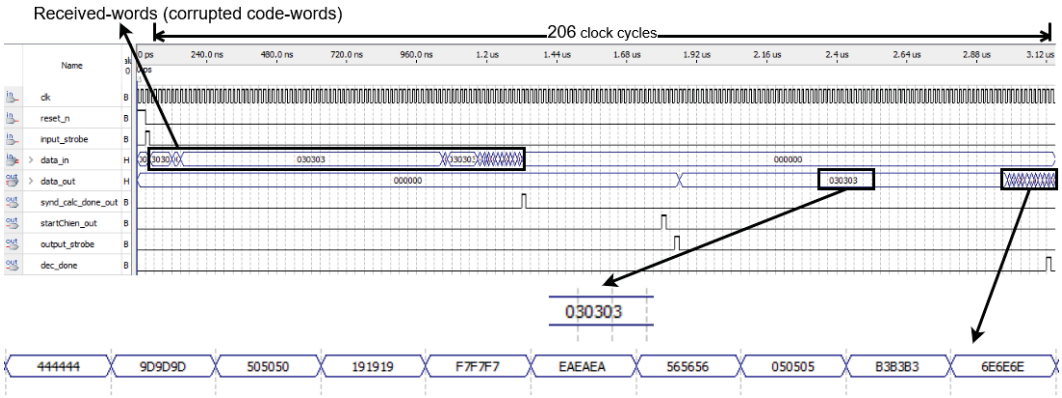


Figure 4–62: Functional simulation of low-latency $RS(255, 225)$ decoder with $p = 3$.

expected that the clock number change is not simply proportional to p as the low-latency encoders do, because the same BM algorithm is used in all low-latency and conventional decoders and does not speed up. It should be noticed that all the tests in this section correct the errors on parity bits, too. As we discussed in Step 5 (the modified Chien search) in Section 3.2.2, if we only correct transmission errors on message part of the code-words, which is also a

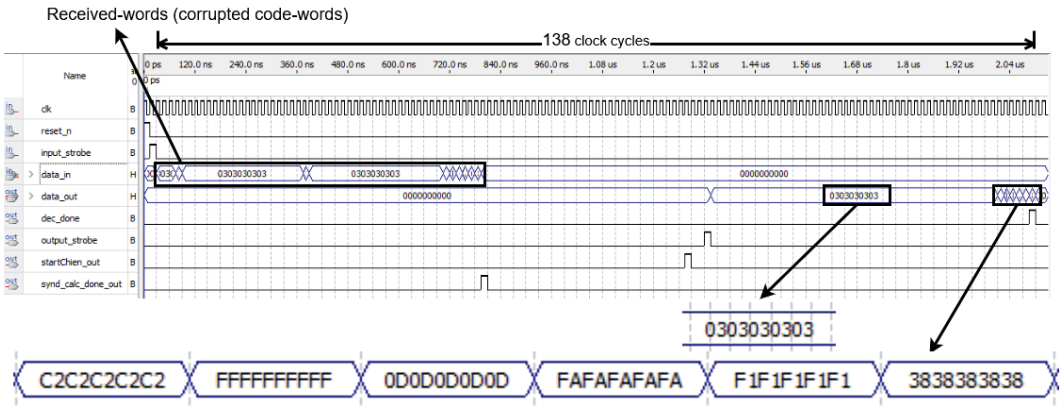


Figure 4–63: Functional simulation of low-latency $RS(255, 225)$ decoder with $p = 5$.

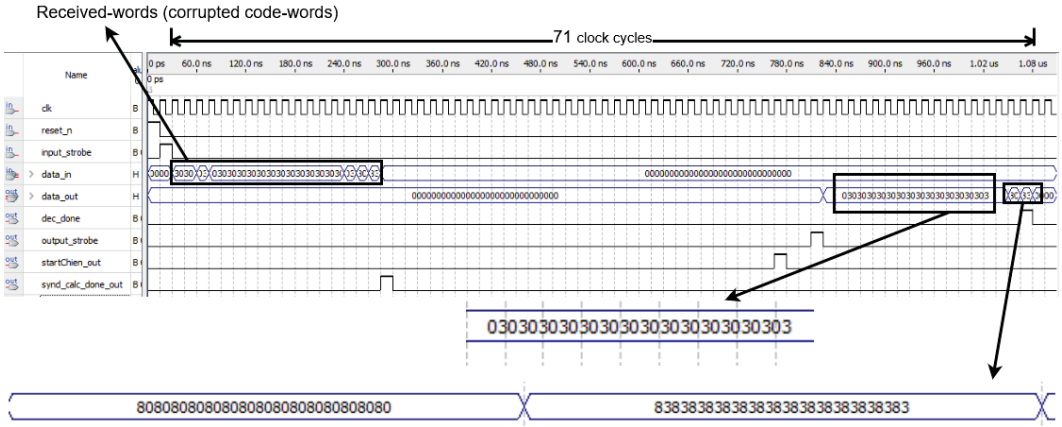


Figure 4–64: Functional simulation of low-latency $RS(255, 225)$ decoder with $p = 15$.

common case in real life, the number of roots used to find error locations would be further reduced by $(n-k)/p$. Equivalently, the clock cycles would be further reduced by $(n - k)/p$. As for hardware cost, the results are reasonal as well. On one hand, the increase in hardware cost was relatively small for p raising, which proved one of the major features that asserted in [17]. On the other hand, the hardware cost of the low-latency decoder with $p = 15$ still reaches a significant level. These results can be interpreted with two facts. First, as the conventional RS decoders have a very complicated structure already, the added DFT and IDFT components did not affect the hardware cost as much as those in the encoders did. Second, as the conventional error-evaluation

$RS(255, 225)$ Decoders with Error-Correction Capacity 15				
65 nm TSMC				
Encoder Name	Area (μm^2)	Power Comsumption (mW)	Max Frequency GHz	Data Rate Gbps
Conventional RS Decoder	$1.60 \cdot 10^5$	77.60	0.895	7.16
Low-Latency RS Decoder ($p = 3$)	$1.03 \cdot 10^5$	32.91	0.886	21.3
Low-Latency RS Decoder ($p = 5$)	$1.08 \cdot 10^5$	43.41	0.880	35.2
Low-Latency RS Decoder ($p = 15$)	$2.52 \cdot 10^5$	145.22	0.877	105.24

Table 4–6: Comparison of ASIC synthesis results for different decoders.

circuit is simply replicated by p times to generate p error-values at each clock, these extra replicated circuits cost more hardware with larger p. The results of maximum clock frequency also satisfy the expectation. As p raises, the DFT and IDFT become more complicated and thus the critical path get increased, which leads to smaller maximum clock frequency.

The technology used for ASIC synthesis is 65 nm TSMC and the results are shown in Table 4–6. Compared to the FPGA synthesis, the maximum clock frequency have a very similar trend. The results for data rate are also expected and they are directly proportional to p due to p-parallel input circuits. Specifically, it can reach more than 100 Gbps for $p = 15$. The area results for low-latency RS decoders are consistent with their hardware costs from FPGA synthesis. However, one exception of this consistence is that conventional RS decoder has a very large area in ASIC synthesis, whereas in FPGA synthesis it costs almost the same amount of hardware as low-latency RS decoder with

$p = 3$. It should be noted that this same scenario also happens in the encoder part (Section 4.3). Similarly, there are two possible reasons. First, a number of logic elements are omitted when the long sequential circuits such as the Chien search, are split into p shorter ones. Second, the rules of ASIC routing may result in a larger area for the longer sequential structure, which is a subject to confirm in the future work. The power consumption results are reasonable. Just like the explanation based on the formula $P = CV^2FA$ that is stated in the encoder section (Section 4.3), the results are mainly following the trend of area as the change in hardware cost is large, whereas the clock frequency difference is relatively small. One thing should be noted is that, the power consumption for $p = 15$ reaches more than 145 mw, which may not fit in some applications of short-reach optical communication.

In conclusion, low-latency RS decoders can efficiently speed up the decoding process without heavily increasing the hardware cost. However, the power consumption can possibly be a concern to some specific applications if the p is very large.

CHAPTER 5
Conclusion

In this thesis, ECC in short-reach optical communication is introduced, which leads to a discussion of latency challenge. After briefly looking into current solutions, a novel class of GRS codes developed by Amin Shokrollahi in [17], that is, low-latency RS codes are studied in detail. Both the conventional and newly proposed low-latency RS coding algorithms are illustrated with details and examples. An implementation of low-latency encoders and decoders using the high-level coding technique in MATLAB are provided and analyzed. The best part of this MATLAB implementation is that it was designed for almost any arbitrary parameters of low-latency RS codes. The BER performance was also verified for low-latency $RS(255, 225)$ codes using MATLAB Communication System Toolbox. However, due to the limit of testing conditions, we can only prove that its BER is lower than 10^{-8}. After MATLAB implementation, low-latency $RS(255, 225)$ encoders and decoders with $p = (3, 5, 15)$ were built in VHDL. All the details of VHDL implementation are examined in Chapter 4 along with the synthesis and performance results. The results suggest that low-latency RS encoder can reduce the latency by a factor of almost p, compared to conventional RS encoders. However, although the hardware cost of a low-latency RS encoder is still relatively small, its increasing ratio is large. The reason can be explained in term of the structures. A low-latency RS encoder or decoder is essentially made by breaking a conventional RS encoder or decoder into p parts and connecting them to DFTs and IDFTs. As the hardware cost of the conventional RS encoder is very small, the hardware costs of the DFT and IDFT components which are proportional

to p, dominate the total hardware cost. The situation in low-latency decoder implementation is very different because the conventional RS decoder is much more complicated. The results show that clock cycles have been largely reduced with only a relatively small increase in hardware cost. Specific to the comparison between the conventional RS decoder and low-latency RS decoder with $p = 15$, the clock cycles is reduced by a factor 769% and its hardware cost is only increased by 57.5%. Power performance is also studied through ASIC synthesis. Although low-latency RS encoders consume much more energy compared to the conventional RS encoder, the increase in power consumption of low-latency RS decoders is very small with respect to p growth.

The future work of this research focuses on two aspects. One aspect is to improve the key equation solver in the decoding process. Currently, all encoding and decoding processes except the key equation solver can run in p parallel. This bottleneck is much desired to be solved. Marc Fossorier has developed a new algorithm in [28] recently. The original BM algorithm is modified based on Gaussian elimination so that two discrepancies can be computed in parallel at each step. It is very promising to integrate this algorithm into low-latency RS decoders. The other aspect is to build a generator which can automatically construct low-latency RS encoders and decoders with arbitrary parameters. Usability is also a significant factor for a type of ECC to be widespread. So far, even with the existing VHDL codes of low-latency $RS(255, 225)$ encoders and decoders, it is still inconvenient to construct low-latency codes with different parameters, especially when k and r are not divisible by p (details in Section 3.4). Therefore, work should be done to standardize all the procedures so that low latency RS encoders and decoders can be generated with only user-defined parameter inputs.

Appendix A

Due to the space limit, the following is only the parts of the novel low-latency RS encoder with $p = 5$ that are different from the implementation for $p = 3$. The explanation are given in Section 4.2.2.

```
entity LowLatRSenc_n255k225_p5 is
port (reset_n: in std_logic;
      --global reset input which can reset every signal.
      data_in: in std_logic_vector(39 downto 0); --3*8=24.
      clk: in std_logic;
      input_strobe: in std_logic;
      output_strobe: out std_logic;
      enc_done_p: out std_logic;
      data_out_p:out std_logic_vector(39 downto 0)); --3*8=24
end;
architecture RTL of LowLatRSenc_n255k225_p5 is

COMPONENT LowLatRSenc_LFSR_n51k45
   port (reset_n: in std_logic;
   data_in: in std_logic_vector (7 downto 0);
   clk: in std_logic;
   input_strobe: in std_logic;
   g0: in std_logic_vector (7 downto 0);
   g1: in std_logic_vector (7 downto 0);
   g2: in std_logic_vector (7 downto 0);
   g3: in std_logic_vector (7 downto 0);
   g4: in std_logic_vector (7 downto 0);
   g5: in std_logic_vector (7 downto 0);
   data_out_p: out std_logic_vector (7 downto 0));
END COMPONENT ;
----------basic parameters---------------------------------
constant N: integer:=255; --codeword length
constant K: integer:=225; --message word length
constant R: integer:=30; --parity bits length =255-225
constant m: integer:=51; --m=N/p=255/5=51
constant t: integer:=45; --t=K/p=225/5=45
constant L: integer:=6; --L=R/p=30/5=6
constant p: integer:=5; --speedup coefficient
constant GFPower: integer:=8; --Galois Field power. is GF(2^4) here.
------------constants-----------------------------------------------------------
constant zero: Galois_Field_element :="00000000";
constant one: Galois_Field_element :="00000001";
constant q: Galois_Field_element:="00001010"; --set the 5-th root of unity
constant q2: Galois_Field_element:="01000100"; --square of the 5-th root of unity
constant q3: Galois_Field_element:="10010010";
constant q4: Galois_Field_element:="11011101";
```

Figure 5–1: Set up for $p = 5$

```
-------------------------DFT function (Fourier Transform)--------------------------------
function DFT (a,b,c,d,e: in Galois_Field_element) return FTout_type is variable DFTout: FTout_type;
   begin
   DFTout(0):=add(add(add(a,b),add(d,e)),c);
   DFTout(1):=add(add(add(a,mul(q,b)),add(mul(d,q3),mul(e,q4))),mul(c,q2));
   DFTout(2):=add(add(add(a,mul(q2,b)),add(mul(d,q),mul(e,q3))),mul(c,q4));
   DFTout(3):=add(add(add(a,mul(q3,b)),add(mul(d,q4),mul(e,q2))),mul(c,q));
   DFTout(4):=add(add(add(a,mul(q4,b)),add(mul(d,q2),mul(e,q))),mul(c,q3));
   return DFTout;
end function DFT;
-------------------------IDFT function (Inverse Fourier Transform)---------------------------------
function IDFT (a,b,c,d,e: in Galois_Field_element) return FTout_type is variable IDFTout: FTout_type;
   begin
   IDFTout(0):=add(add(add(a,b),add(d,e)),c);
   IDFTout(1):=add(add(add(a,mul(q4,b)),add(mul(d,q2),mul(e,q))),mul(c,q3));
   IDFTout(2):=add(add(add(a,mul(q3,b)),add(mul(d,q4),mul(e,q2))),mul(c,q));
   IDFTout(3):=add(add(add(a,mul(q2,b)),add(mul(d,q),mul(e,q3))),mul(c,q4));
   IDFTout(4):=add(add(add(a,mul(q,b)),add(mul(d,q3),mul(e,q4))),mul(c,q2));
   return IDFTout;
end function IDFT;
```

Figure 5–2: Defining functions for $p = 5$

```
151   LFSR0:  LowLatRSenc_LFSR_n51k45
152   port map(reset_n=>reset_n,
153      data_in=>DFTout(0),
154      clk=>clk,
155      input_strobe=>input_strobe,
156      g0=>"00001111",
157      g1=>"11110011",
158      g2=>"00011001",
159      g3=>"11011100",
160      g4=>"11111100",
161      g5=>"11000100",
162      data_out_p=>LFSRout(0));
163   LFSR1:  LowLatRSenc_LFSR_n51k45
164   port map(reset_n=>reset_n,
165      data_in=>DFTout(1),
166      clk=>clk,
167      input_strobe=>input_strobe,
168      g0=>"11100111",
169      g1=>"00001011",
170      g2=>"10001101",
171      g3=>"10101110",
172      g4=>"11010111",
173      g5=>"10010101",
174      data_out_p=>LFSRout(1));
175   LFSR2:  LowLatRSenc_LFSR_n51k45
176   port map(reset_n=>reset_n,
177      data_in=>DFTout(2),
178      clk=>clk,
179      input_strobe=>input_strobe,
180      g0=>"01111111",
181      g1=>"01111101",
182      g2=>"00111000",
183      g3=>"00011001",
184      g4=>"01111011",
185      g5=>"00110111",
186      data_out_p=>LFSRout(2));
187   LFSR3:  LowLatRSenc_LFSR_n51k45
188   port map(reset_n=>reset_n,
189      data_in=>DFTout(3),
190      clk=>clk,
191      input_strobe=>input_strobe,
192      g0=>"10110110",
193      g1=>"00011011",
194      g2=>"10100111",
195      g3=>"11001000",
196      g4=>"11110001",
197      g5=>"01101110",
198      data_out_p=>LFSRout(3));
199   LFSR4:  LowLatRSenc_LFSR_n51k45
200   port map(reset_n=>reset_n,
201      data_in=>DFTout(4),
202      clk=>clk,
203      input_strobe=>input_strobe,
204      g0=>"10000110",
205      g1=>"01000111",
206      g2=>"10100010",
207      g3=>"00001110",
208      g4=>"11100011",
209      g5=>"11011100",
210      data_out_p=>LFSRout(4));
```

Figure 5–3: Component Instantiations for $p = 5$

```
DFTout<=DFT(data_in((p*GFPower-33)downto(p*GFPower-40)),data_in((p*GFPower-25)downto(p*GFPower-32)),
         data_in((p*GFPower-17)downto(p*GFPower-24)),data_in((p*GFPower-9)downto(p*GFPower-16)),
         data_in((p*GFPower-1)downto(p*GFPower-8)));
```

Figure 5–4: Preprocess for $p = 5$

```
--------------------------basic parameters----------------------------------------
constant errCap: integer:=3; --error correction capacity
constant data_size:std_logic_vector (7 downto 0):="00101101";
constant coded_size:std_logic_vector (7 downto 0):="00110011";
constant GFPower: integer:=8;
```

Figure 5–5: LFSR basic parameters for $p = 5$

```
process(reset_n,input_strobe,DoCalc_p,parity_reg_p,sum,g0,g1,g2,g3,g4,g5,xferPB_p)
begin
   parity_reg<=parity_reg_p;
   if(input_strobe='1') then
      for i in 0 to (2*errCap-1) loop
         parity_reg(i)<=(others=>'0');
      end loop;
   elsif (DoCalc_p='1') then
      parity_reg(5)<=add(parity_reg_p(4),mul(sum,g5));
      parity_reg(4)<=add(parity_reg_p(3),mul(sum,g4));
      parity_reg(3)<=add(parity_reg_p(2),mul(sum,g3));
      parity_reg(2)<=add(parity_reg_p(1),mul(sum,g2));
      parity_reg(1)<=add(parity_reg_p(0),mul(sum,g1));
      parity_reg(0)<=mul(sum,g0);
   elsif (xferPB_p='1') then
      for i in (2*errCap-1) downto 1 loop
         parity_reg(i)<=parity_reg_p(i-1);
      end loop;
      parity_reg(0)<=(others=>'0');
   end if;
end process;
```

Figure 5–6: LFSR parity-bits register update for $p = 5$

```
process(counter_p,xferEn_p,LFSRout,data_inbuf_p)
begin
   if (counter_p>0 and counter_p<t+1) then
      data_out<=data_inbuf_p;
   elsif (xferEn_p='1') then
      data_out((p*GFPower-33)downto(p*GFPower-40))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2),LFSRout(3),LFSRout(4))(0);
      data_out((p*GFPower-25)downto(p*GFPower-32))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2),LFSRout(3),LFSRout(4))(1);
      data_out((p*GFPower-17)downto(p*GFPower-24))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2),LFSRout(3),LFSRout(4))(2);
      data_out((p*GFPower-9)downto(p*GFPower-16))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2),LFSRout(3),LFSRout(4))(3);
      data_out((p*GFPower-1)downto(p*GFPower-8))<=IDFT(LFSRout(0),LFSRout(1),LFSRout(2),LFSRout(3),LFSRout(4))(4);
   else
      data_out<=(others=>'0');
   end if;
end process;
```

Figure 5–7: Output control for $p = 5$

Appendix B

Due to the space limit, the following is only the parts of the novel low-latency RS decoder with $p = 5$ that are different from the implementation for $p = 3$. The explanation are given in Section 4.4.2.

```
entity LowLatRSdec_n255k225_p5_syn is
port (reset_n: in std_logic;
    data_in: in std_logic_vector(39 downto 0);  --5 * 8=40
    clk: in std_logic;
    input_strobe: in std_logic;
    synd_calc_done: out std_logic;
    error_present_out:out std_logic;
    syndrome_out: out std_logic_vector (239 downto 0) );
end;
architecture RTL of LowLatRSdec_n255k225_p5_syn is
---------basic parameters-----------------------------------------------
constant GFPower: integer:=8;
constant N: integer:=255; --codeword length
constant K: integer:=225; --message word length
constant R: integer:=30; --parity bits length =255-225
constant m: integer:=51; --m=N/p=255/5=51
constant t: integer:=45; --t=K/p=225/5=45
constant L: integer:=6; --L=R/p=30/5=6
constant p: integer:=5;--speed-up coefficient
constant bitNum: integer:=p*GFpower;
```

Figure 5–8: Set up of syndrome component for $p = 5$

```
------------------------DFT function (Fourier Transform)--------------------------------
function DFT (a,b,c,d,e: in Galois_Field_element) return FTout_type is variable DFTout: FTout_type;
  begin
    DFTout(0):=add(add(add(a,b),add(d,e)),c);
    DFTout(1):=add(add(add(a,mul(q,b)),add(mul(d,q3),mul(e,q4))),mul(c,q2));
    DFTout(2):=add(add(add(a,mul(q2,b)),add(mul(d,q),mul(e,q3))),mul(c,q4));
    DFTout(3):=add(add(add(a,mul(q3,b)),add(mul(d,q4),mul(e,q2))),mul(c,q));
    DFTout(4):=add(add(add(a,mul(q4,b)),add(mul(d,q2),mul(e,q))),mul(c,q3));
    return DFTout;
end function DFT;
```

Figure 5–9: DFT function of syndrome component for $p = 5$

```
process(initialize_p,xferSyn_p,intS_p,syndrome_p,error_present_p,intEP)
begin
   syndrome<=syndrome_p;
   error_present<=error_present_p;
   if(initialize_p='1') then
      syndrome<=(others=>'0');
      error_present<='0';
   elsif (xferSyn_p='1') then
      syndrome<=intS_p(4)(5)& intS_p(3)(5) & intS_p(2)(5) & intS_p(1)(5)& intS_p(0)(5) & intS_p(4)(4) &
                intS_p(3)(4)& intS_p(2)(4) & intS_p(1)(4) & intS_p(0)(4) & intS_p(4)(3)& intS_p(3)(3) &
                intS_p(2)(3) & intS_p(1)(3)& intS_p(0)(3) & intS_p(4)(2) & intS_p(3)(2)& intS_p(2)(2) &
                intS_p(1)(2) & intS_p(0)(2)& intS_p(4)(1) & intS_p(3)(1) & intS_p(2)(1)& intS_p(1)(1) &
                intS_p(0)(1) & intS_p(4)(0)& intS_p(3)(0) & intS_p(2)(0) & intS_p(1)(0) & intS_p(0)(0);
      error_present<=intEP(0)(0) or intEP(0)(1) or intEP(0)(2) or intEP(0)(3) or intEP(0)(4) or intEP(0)(5) or
                     intEP(1)(0) or intEP(1)(1) or intEP(1)(2) or intEP(1)(3) or intEP(1)(4) or intEP(1)(5) or
                     intEP(2)(0) or intEP(2)(1) or intEP(2)(2) or intEP(2)(3) or intEP(2)(4) or intEP(2)(5) or
                     intEP(3)(0) or intEP(3)(1) or intEP(3)(2) or intEP(3)(3) or intEP(3)(4) or intEP(3)(5) or
                     intEP(4)(0) or intEP(4)(1) or intEP(4)(2) or intEP(4)(3) or intEP(4)(4) or intEP(4)(5);
   end if;
end process;
```

Figure 5–10: Final output of syndrome component for $p = 5$

```
entity LowLatRSdec_n255k225_p5_chienNcorrect is
port (reset: in std_logic;
    clk: in std_logic;
    data_input_strobe:in std_logic;
    startChien: in std_logic;
    omega_poly: in std_logic_vector(119 downto 0);
    lambda_poly: in std_logic_vector(127 downto 0);
    data_in:in std_logic_vector(39 downto 0);
    dec_done_p:out std_logic;
    output_strobe:out std_logic;
    data_out_p:out std_logic_vector(39 downto 0));
end;

architecture RTL of LowLatRSdec_n255k225_p5_chienNcorrect is
-----------basic parameters-----------------------------------------------------------
constant GFPower: integer:=8;
constant N: integer:=255; --codeword length
constant K: integer:=225; --message word length
constant R: integer:=30; --parity bits length =255-225
constant errCap: integer:=R/2;
constant m: integer:=51; --m=N/p=255/5=51
constant t: integer:=45; --t=K/p=225/5=45
constant L: integer:=6; --L=R/p=30/6=5
constant p: integer:=5;--speed-up coefficient
constant lamLength: integer:=16;
constant derLamLength: integer:=8;
constant omeLength: integer:=15;
constant chienLength: integer:=20;
```

Figure 5–11: Set up of Chien search and correction component for $p = 5$

```
-------------------------------------------------------IDFT-------------------------------------------------------
function IDFT (a,b,c,d,e: in Galois_Field_element) return FTout_type is variable IDFTout: FTout_type;
  begin
    IDFTout(0):=add(add(add(a,b),add(d,e)),c);
    IDFTout(1):=add(add(add(a,mul(q4,b)),add(mul(d,q2),mul(e,q))),mul(c,q3));
    IDFTout(2):=add(add(add(a,mul(q3,b)),add(mul(d,q4),mul(e,q2))),mul(c,q));
    IDFTout(3):=add(add(add(a,mul(q2,b)),add(mul(d,q),mul(e,q3))),mul(c,q4));
    IDFTout(4):=add(add(add(a,mul(q,b)),add(mul(d,q3),mul(e,q4))),mul(c,q2));
    return IDFTout;
end function IDFT;
```

Figure 5–12: IDFT function of Chien search and correction component for $p = 5$

```
process (Chien_product_p)
begin
   for i in 0 to (p-1) loop
      Chien_sum(i)<=add(add(Chien_product_p(i),Chien_product_p(i+p)),
                        add(Chien_product_p(i+2*p),Chien_product_p(i+3*p)));
   end loop;
end process;

FTout<= IDFT(Chien_sum(0),Chien_sum(1),Chien_sum(2),Chien_sum(3),Chien_sum(4));
```

Figure 5–13: Chien sum of syndrome component for $p = 5$

Appendix C

The following figures show the timing simulation results corresponding to the low-latency RS encoders described in Section 4.3.

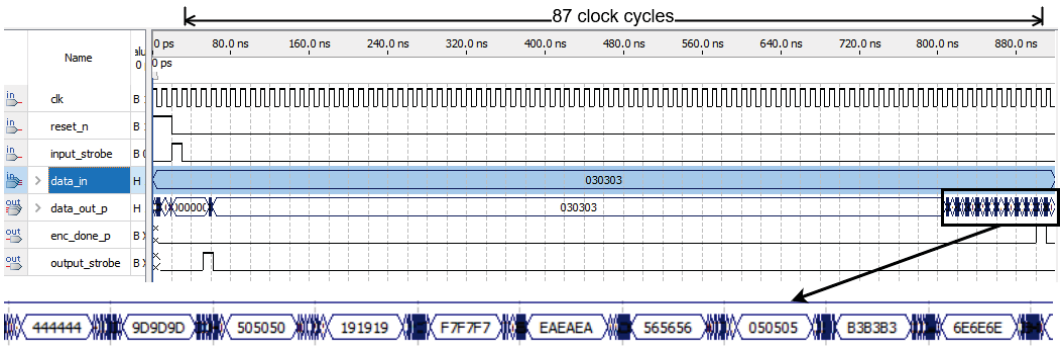


Figure 5–14: Timing simulation of low-latency $RS(255, 225)$ encoder with $p = 3$

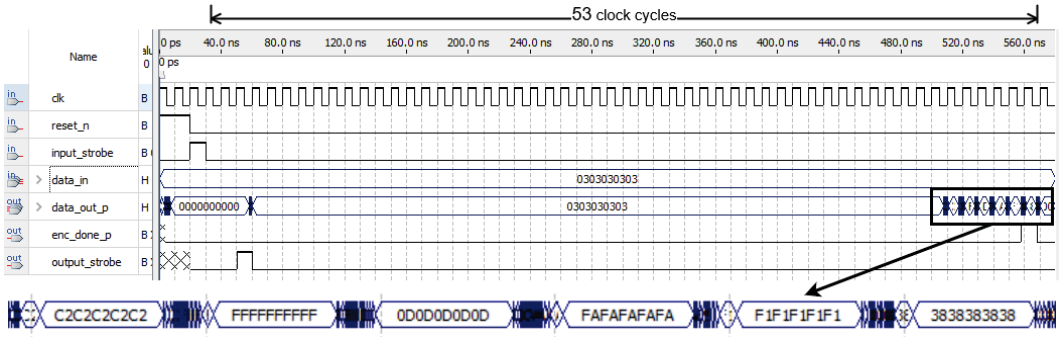


Figure 5–15: Timing simulation of low-latency $RS(255, 225)$ encoder with $p = 5$

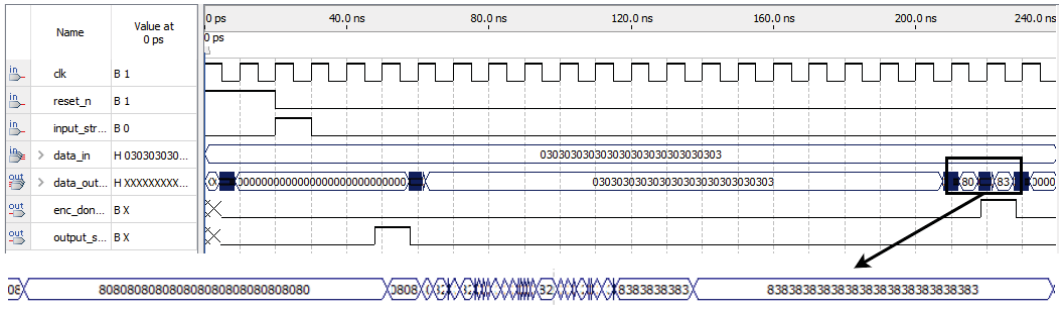


Figure 5–16: Timing simulation of low-latency $RS(255, 225)$ encoder with $p = 15$

Appendix D

The following figures show the timing simulation results corresponding to the low-latency RS decoders described in Section 4.5.

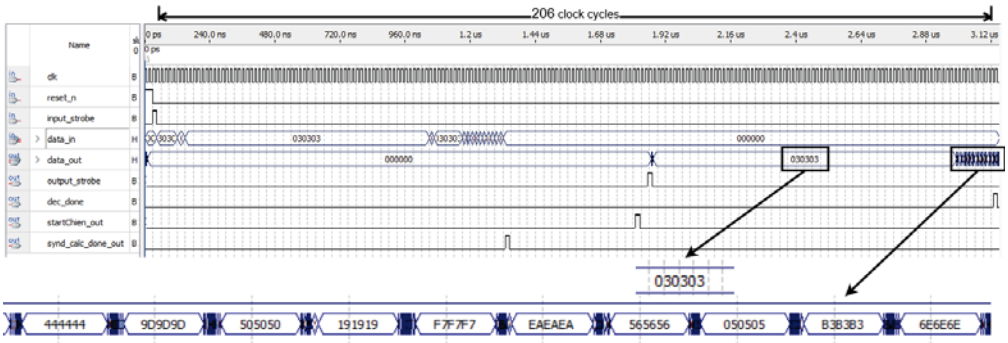


Figure 5–17: Timing simulation of low-latency $RS(255, 225)$ decoder with $p = 3$

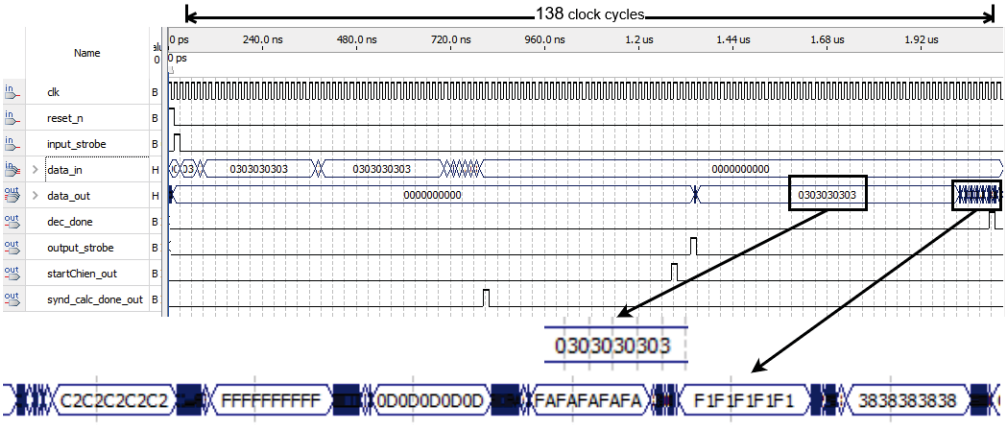


Figure 5–18: Timing simulation of low-latency $RS(255, 225)$ decoder with $p = 5$

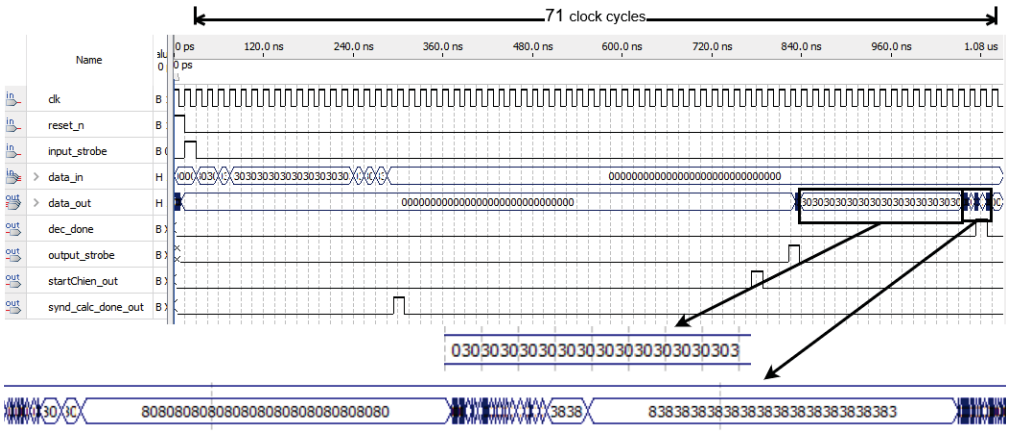


Figure 5–19: Timing simulation of low-latency $RS(255, 225)$ decoder with $p = 15$

References

[1] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. XXVII, no. 3, pp. 379–423, 1948.

[2] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. XXIX, no. 2, pp. 147–160, 1950.

[3] B. Sklar, *Digital Communications: Fundamentals and Applications*. Prentice-Hall, Upper Saddle River, 1988.

[4] J. A. Alzubi, O. A. Alzubi, and T. M. Chen, *Forward Error Correction Based on Algebraic-Geometric Theory*. Springer, 2014.

[5] M. Nakazawa, K. Kikuchi, and T. Miyazaki, *High Spectral Density Optical Communication Technologies*. Springer, 2010.

[6] W. D. Grover, "Forward error correction in dispersionlimited lightwave systems," *IEEE Journal of Lightwave Technology*, vol. 6, no. 5, pp. 643–645, 1988.

[7] S. Yamamoto, H. Taga, N. Edagawa, and H. Wakabayashi, "Observation of BER degradation due to fading in long distance optical amplifier system," *IEEE Electronics Letters*, vol. 29, no. 2, pp. 209–210, 1993.

[8] S. Yamamoto, H. Takahira, and M. Tanaka, "5 Gbit/s optical transmission terminal equipment using forward error correcting code and optical amplifier," *IEEE Electronics Letters*, vol. 30, no. 3, pp. 254–255, 1994.

[9] O. A. Sab and J. Fang, "Concatenated forward error correction schemes for long-haul DWDM optical transmission systems," in *25th European Conference on Optical Communication (ECOC)*, 1999.

[10] A. Puc, F. Kerfoot, A. Simons, and D. L. Wilson, "Concatenated FEC experiment over 5000 km long straight line WDM test bed," in *The Optical Networking and Communication Conference and Exhibition*, 1999.

[11] F. Aznar, S. Celma, and B. Calvo, *CMOS Receiver Front-ends for Gigabit Short-Range Optical Communications*. Springer, 2013.

[12] C. Kachris and I. Tomkos, "A survey on optical interconnects for data centers," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 4, 2012.

[13] 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, 2008.

[14] J. D. Lee and M. H. Sunwoo, "Three-parallel Reed-Solomon Decoder using S-DCME for high-speed communications," *Journal of Signal Processing Systems*, vol. 66, pp. 15–24, 2012.

[15] R. Zhou, R. L. Bidan, R. Pyndiah, and A. Goalic, "Low-complexity high-rate Reed-Solomon block turbo codes," *IEEE Transactions on communications*, vol. 55, no. 9, pp. 1656–1660, 2007.

[16] P. P. Ankolekar, R. Isaac, and J. W. Bredow, "Multibit error-correction methods for latency-constrained flash memory systems," *IEEE Transactions on Device and Materials Reliability*, vol. 10, no. 1, pp. 33–39, 2010.

[17] A. Shokrollahi, "A class of generalized RS-codes with faster encoding and decoding algorithms," in *2013 Information Theory and Applications Workshop (ITA)*, 2013.

[18] I. S. Reed and G. Solomon, "Polynomial codes over certain fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[19] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[20] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Pearson-Prentice Hall, 2004.

[21] Y. Jiang, *A Practical Guide to Error-Control Coding*. Artech House, 2010.

[22] J. Massey, "Shift-register synthesis and bch decoding," *IEEE Communications Surveys and Tutorials*, vol. 15, pp. 122–127, 1969.

[23] T. K. Moon, *Error Control Coding: Mathematical Methods and Algorithms*. John Wiley and Sons, 2005.

[24] G. D. Forney, "On decoding BCH codes," *IEEE Transaction on Information Theory*, vol. IT-11, pp. 549–557, 1965.

[25] G. L. Guardia, "Asymmetric quantum Reed-Solomon and generalized Reed-Solomon codes," *Quantum Information Processing*, vol. 11, no. 2, pp. 591–604, 2012.

[26] V. Glavac, "A VHDL code generator for Reed-Solomon encoders and decoders," Master's thesis, Concordia University, Montreal, Quebec, Canada, April 2003.

[27] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, Prentice Hall, 1995.

[28] M. Fossorier, "Gaussian elimination decoding of t-error correcting Reed-Solomon codes in t steps and $o(t^2)$ complexity," *IEEE Communications Letters*, vol. 19, no. 7, 2015.