國立交通大學

電機學院 IC 設計產業研發碩士班

碩 士 論 文

適用於多模式高速通訊系統之
低複雜度里德所羅門編解碼模組

Low Complexity Reed-Solomon CODEC for Multi-Mode High-Speed Communication Systems

研 究 生：蘇建毓

指導教授：溫瓌岸　教授

中 華 民 國 九 十 七 年 一 月

適用於多模式高速通訊系統之低複雜度里德所羅門編解碼模組
Low Complexity Reed-Solomon CODEC for Multi-Mode High-Speed Communication Systems

研 究 生：蘇建毓　　　　Student：Jian-Yuh Su

指導教授：溫瓌岸 博士　　　Advisor：Dr. Kuei-Ann Wen

國 立 交 通 大 學
電機學院 IC 設計產業研發碩士班
碩 士 論 文

A Thesis
Submitted to College of Electrical and Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Industrial Technology R & D Master Program on
IC Design

January 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年一月

國立交通大學
電機學院IC設計產業研發碩士班

論文口試委員會審定書

本校 電機學院IC設計產業研發碩士班 蘇建毓 君

所提論文：

（中文） 適用於多模式高速通訊系統之低複雜度里德索羅門編解碼模組

（英文） Low-complexity Reed-Solomon CODEC for Multi-mode High-speed Communication Systems

合於碩士資格水準、業經本委員會評審認可。

口試委員：[illegible] [illegible]

[illegible]

指導教授：[illegible]

專班主任：[illegible]

中 華 民 國 97 年 1 月 10 日

適 用 於 多 模 式 高 速 通 訊 系 統 之 低 複 雜 度 里 德 所 羅 門 編 解 碼 模 組

學生：蘇建毓　　　　　　　　　　　　　　　　　　指導教授：溫瓌岸 博士

國立交通大學電機學院產業研發碩士班

摘　　　要

本論文主要討論設計一應用於多模式高速通訊系統之低複雜度里德所羅門編解碼模組。所提出的演算法以係數最簡化為方針。由 PGZ 演算法，Berlekamp 演算法，iBMA，到提出的 SiBM 演算法，皆有詳盡的推導及整合比對。尤其著名的廣義牛頓特性，是 Berlekamp 系列演算法的重要精神，在本論文以三種方法來推導驗證，這些皆在第二章以數學表示法呈現。

所提出的架構有主要幾個特色，以解碼器的三個主要運算區塊來做介紹：
一、在錯誤症(syndrome)計算區塊，藉由非常簡單的 t 解碼器，來支配十六個症狀值單元動作，以達到可以運用在多模操作上。
二、在關鍵方程式求解器(KES)，為了克服大部分序列式 BM 架構，計算差值接著決定更新校正方程式的關鍵路徑的瓶頸，以及避免在特殊錯誤位置及錯誤值組合所產生的資料危障。此設計藉由判定時序改變，控制迭代及資料流，取代延時(stall)。使得計算錯誤位置方程式的時脈週期數小於 $2t(t+1)$，約為 $t\cdot\delta_i+3t$ 個週期數(δ_i 為第 i 迭代的 degree)。且藉由在演算法上將係數同化，以致於在設計共用電路時，甚至不需額外硬體來選擇資料流，因此某些關鍵路徑瓶頸得以克服。更進一步利用特製位址線，和較簡單的控制電路聯結技巧，省去許多電路。
三、在 CSEE 區塊，為符合多模式操作，通常在輸入端加上有限場乘法器，使 Chien's Search 跳至指定搜尋起點及產生對應的錯誤值。本論文提出一個共用補償器的方法，並將其和合併到 KES 中，大幅減少硬體，使得由單模(255,239)至多模僅需增加少許的邏輯閘數，電路示意圖於第三章呈現。

此架構實作於 Xinlinx VirtexE xcv2000e FPGA 和 UMC 0.18 1P6M 製程，經過 204.8 億隨機位元驗證，皆正常運作。在最高時脈頻率(Clock Rate) 730Mhz 下，數據傳輸率(Data Rate)可達 5.84bps，此時僅約 11596 個邏輯閘數。

Low Complexity Reed-Solomon CODEC for Multi-Mode High-Speed Communication Systems

student：Jian-Yuh Su　　　　　　　　　　Advisors：Dr. Kuei-Ann Wen

Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

ABSTRACT

In this thesis, a high-speed and low-complexity design of multi-mode Reed-Solomon codec is proposed. In the beginning of deliberating algorithms, the policy is to simplify the coefficients of equations, so as to construct a simpler structure.

The proposed RS decoder has some major features introduced as following：

1. In the SC block, a simpler t-decoder is exploited to dominate the sixteen cells, so as to answer to multi-mode applications.
2. In the KES, the “Decision Variations” is proposed to break the main speed bottleneck of iBMA in the iterative computation of discrepancies followed by updating the correction polynomial and to prevent the special-case data hazard in the most serial structures. Also, for the sake of keeping the critical path in the reusing hardware is still $T_{ff}+T_{mult}+T_{\text{xor}}$, the assimilative coefficient knack is adopted. Further, we use the purpose-built address line to simplify the hardware complexity of storage element
3. In Chien’s Search and Forney’s block, fifteen Compensators used to adjust the starting point of search are reduced to one and combined in the KES block.

After 204.8 hundred-million bits transmission and verification regular, the proposed RS decoder for multi-mode applications (n<=255, t<=8) is implemented by Xinlinx VirtexE xcv2000e FPGA and Synopsys DC with UMC018 library. The design possesses higher speed and lower gate count than present decoder design. The data rate of the proposed decoder is 5.84bps at the maximum clock rate of 730MHz with 11596 gates.

誌　謝

在完成本篇論文之際，首先要感謝的是我的指導教授溫瓌岸教授。在就讀研究所的期間，承蒙教授不斷給予研究方向上的指導、督促與研究資源；在此向溫教授致上最誠摯的謝意。此外，亦感謝參與口試審核委員會的交大研發長李鎮宜教授與台大電機系所吳安宇教授，提供寶貴的建議與指教。

接下來要感謝的就是實驗室的學長姐們的指導與照顧：美芬學姐、嘉笙學長、哲生學長、文安學長、晧名學長、立協學長、彥凱學長、懷仁學長、志德學長、俊憲學長等。學長姐們的幫助與指導讓我獲益良多。

還有感謝實驗室的同學們－世基、書旗、昱瑞、建龍、家岱、漢健、閎仁，兩年來在課業和日常生活上總是相互的扶持，一起高興、一起嬉笑、一起度過難關。還有給予我信心與勇氣的學弟們－佳欣、磊中、國爵、柏麟、俊彥、士賢、謙若，同時也要感謝實驗室的助理們－苑佳、淑怡、慶宏、恩齊、怡倩、智伶、宛君、嘉誠，幫忙實驗室裡大大小小的事，讓我們能更專心於研究工作。

最後，我最要感謝的是家人與女友無怨無悔的付出與支持，感謝好友們給我的鼓勵，感謝關心我與幫助我的人，……，要感謝的人族繁不及備載，謹此這篇研究論文，與我的家人及好友分享我的收穫與喜悅。

蘇建毓

2008 年 1 月

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the communication system and storage system, the integrity of data transmission is necessary. But the error probability increases after interference introduce. Consequently, the correctness of data transmission is reduced. So, many digital signaling applications in communication system and storage system employ Forward Error Correction (FEC). FEC is a technique in which redundant information is added to the signal to allow the receiver to detect and correct errors that may have occurred in transmission. Figure 1.1 shows the connection of FEC in the communication system.

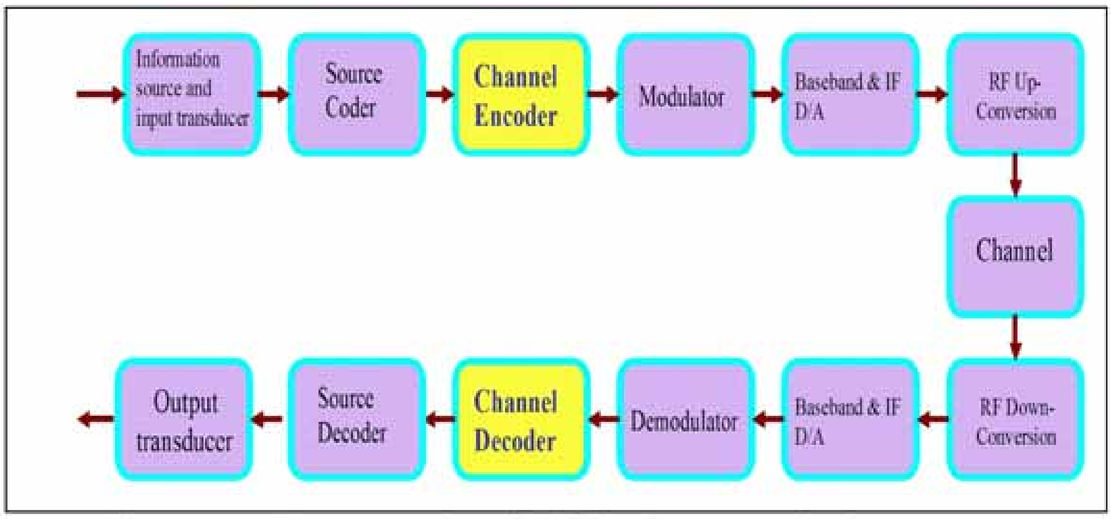


Figure 1.1：The connection of FEC in the communication system

Many different types of codes have been devised for this purpose. Due to the well characteristic of error correction capability for both random errors and burst errors, Reed-Solomon codes [1] have been widespread used for error correction in various kinds of digital transmission and storage systems. It has been proved to be a good compromise between efficiency and complexity.

In general, the main categories of error correcting codes are block codes and convolution codes. Reed-Solomon codes are belonging to a block code, meaning the message to be transmitted is divided into separate blocks of data and each block has parity protection information added to it to form a self-contained codeword. It is also a systematic code, which means the encoding process does not vary the message symbols. The protection symbols are added as a separate part of the block. Figure 1.2 is a sketch map of this.

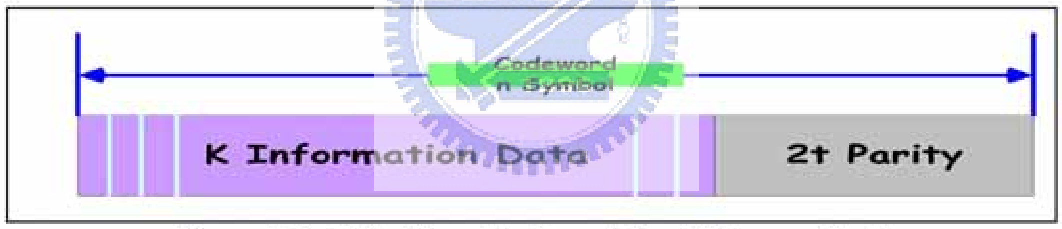


Figure 1.2：The Constitution of Reed-Solomon Code.

In Reed-Solomon codes, any two code-words operation produces another codeword. So, Reed-Solomon code is also a linear code. And because of cyclically shifting the symbols of a codeword produces another code-word, it is cyclic. The classification of Reed-Solomon Codes in the FEC is shown in Figure1.3. Furthermore, it belongs to the family of Bose-Chaudhuri-Hocquenghem (BCH) codes [2, 3], but is distinguished by having multi-bit symbols. It makes the code particularly good at dealing with bursts of errors. In virtue of that although a symbol may have all its bits in error, this counts as only one symbol error in terms of the correction capacity of the code. It is the important factor in adopting RS code in many practical applications such as digital audio and video [4], magnetic and optical recording [5,6], computer memory, cable modem[7], xDSL[8,9], wireless and satellite communications systems.

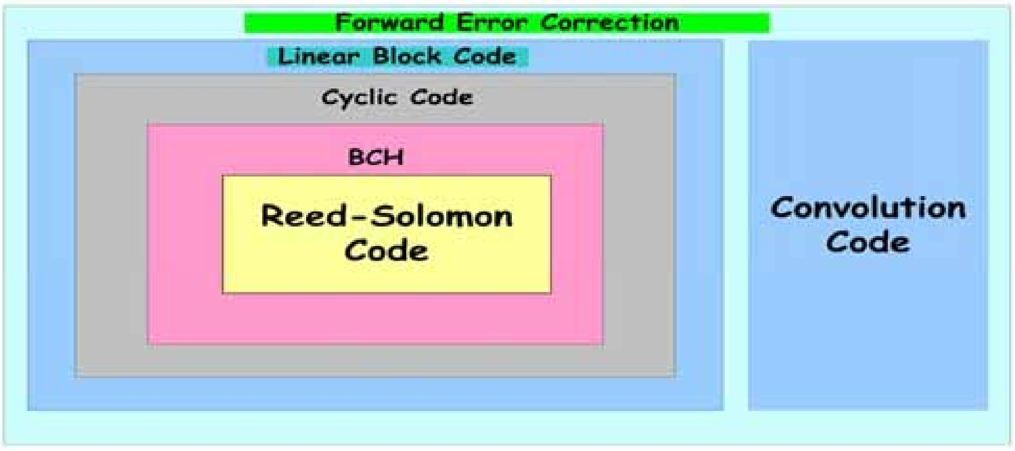


Figure 1.3：The classification of Reed-Solomon Codes in the FEC.

It provides different levels of protection by choosing different parameters for a code and affects the complexity of implementation. Therefore, a RS code can be expressed as (n, k) code, where n is the codeword length in symbols and k is the number of information symbols in the message.

In general, $n \leq 2^m - 1$ .Where m is the order of Galois Field.

If $n < 2^m - 1$, this is referred to as a **Shortened Code**. There are $n - k$ parity symbols and t symbol errors can be corrected in a block,

where $t = (n\text{-}k)/2$……. …for $n\text{-}k$ even

or $t = (n\text{-}k\text{-}1)/2$ ……..for $n\text{-}k$ odd.

Modified RS codes are frequently used for providing different code rates and correcting capabilities. Shortening and puncturing is the major modification of RS codes. Ordinary, the standard error-and-erasure decoder is employed to decode the shortened and punctured RS codes. In this thesis, our design focus on the error-only decoder for the shorten code.

1.2 Motivation

As mentioned above, for burst errors and random errors, Reed-Solomon codes are popular to provide data integrity and exploited in numerous digital systems such as those for deep space, digital subscriber loop (cable modem and xDSL), gigabit (1000base-T) and 10G Ethernet (Fiber, Copper), digital audio and video broadcast (DAB/DVB), magnetic and optical recording (CD, DVD), computer memory (HDD), wireless as well as in satellite communications systems, and etc.

Table 1.1：Some parameters list of specification with regard to RS codes.

Application	Specification： (n, k, t)	
HDD	(72, 64, 4)	(36, 32, 2)
CD	(32, 28, 2)	(28, 24, 2)
DVD	(208, 192, 8)	(182, 172, 5)
DVB ITU J83(A,B)	(204, 188, 8)	
STM-16 OC-192	(255, 239, 8)	
CCSDS	(15~255, n-16, 8)	
x-DSL & Cable modem	(~255, n-2t, t=1~8)	
WiMAX	(~255, ~239, ~8)	

The data processing topology in each application is based on different error property. And because of the difference at processing topology, choosing different parameters such as the number of bit in a symbol, the number of symbol in a codeword and in a information data as well as in the redundant data for a code to provide different levels of protection is necessary. The constitution of parameter is numerous and each application has its own specification parameters. Table1.1 shows a list of specification parameters with regard to Reed-Solomon codes.

The parameter affects the complexity of implementation. For the sake of various purpose applications or more than one parameter in a system, we must settle on the suitable VLSI design orientation.

As showing in Figure1.4, the ASIC having the best performance and then is the reconfigurable hardware. If a design is perfect, the performance of reconfigurable hardware is very close to ASIC. The remainder, even the processor or the DSP, may have smaller area but the processing speed of this approach is slow and can only be dealt with low bit rate data.

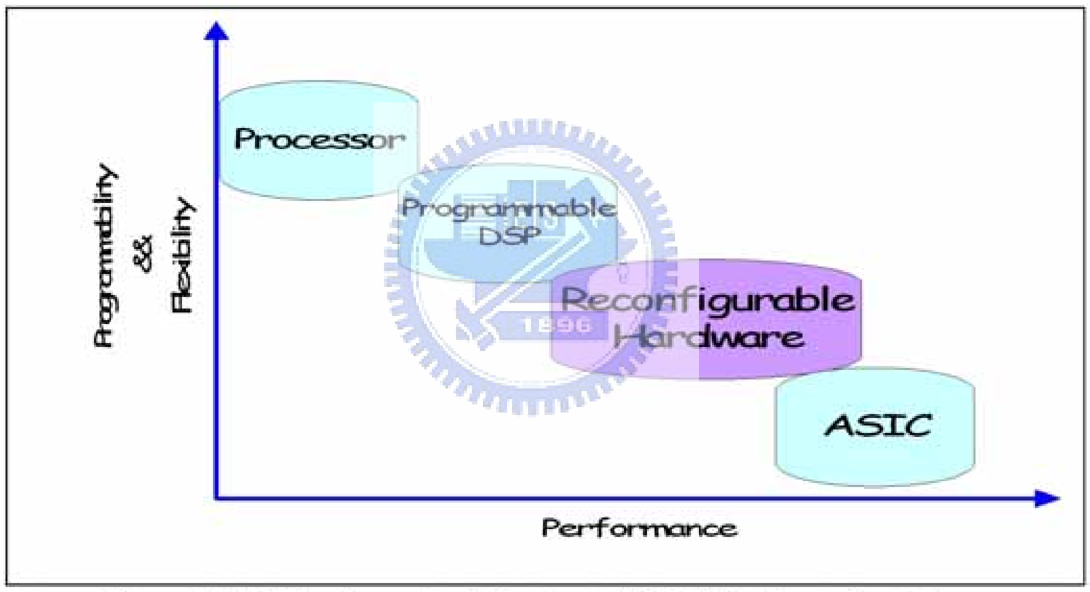


Figure 1.4：The characteristic chart of VLSI design orientation.

The circuit must process large amounts of data within a limited time in order to handle video and audio signals. Continual demand for ever higher data rates makes it necessary to devise very high-speed implementations of decoders for Reed–Solomon codes.

At the same time, we need a system to suit numerous different characters of channel that have different specification to guarantee the quality on transmission. So, the multi-mode and high-speed system is preferred. In other words, the system must be high-speed enough and we don't need to redesign from the scratch.

Hitherto, the reported decoder implementations [10, 11, 12] have quoted data rates of ranging from 144 Mb/s to 3.2 Gb/s. The high throughputs have been achieved by architectural reformations such as pipelining, parallel processing, and etc. More than half of the architectures [11, 13, 14, 15] take advantage of the extended Euclidean (**eE**) algorithm for computing the greatest common divisor between two polynomials [16]. A key benefit of the **eE** algorithm is regularity. Furthermore, the critical path delay in these architectures is at most $T_{mult} + T_{add} + T_{mux}$, where T_{mult} is the delays of the finite-field multiplier. Similarly, T_{add} and T_{mux} are the delays of adder, and multiplexer respectively. This is fast enough for most applications. Relatively, fewer architectures are based on the Berlekamp–Massey (**BM**) algorithm [16, 17, 18, 19]. Because of the irregularity and longer critical path delay, the BM algorithm is almost utilized for low-complexity applications rather than high-speed applications.

In this thesis, we show that it is possible to exploit the **BM** algorithm to achieve extremely high-speed applications. Wonderfully, it cannot only operate at higher data rate, but also has lower gate complexity for multi-mode applications.

1.3 Organization

Although the logic structure of Reed-Solomon codes is simple, the mathematical foundation is complicated. It is important to have a reasonable understanding of at any rate what needs to be done, if not why it is done. In this thesis, a brief mathematical representation as well as schematic architecture of the encoding and decoding processes will be introduced.

In Chapter 2, we explain some essential background with regard to the Galois field GF(2^m), the process of encoding, and the mathematical representation of decoding. In decoding, we first verify the method of Peterson-Gorenstein-Zierler(PGZ) algorithm, and the conventional Berlekamp-Massey(BM) algorithm. Then, the inversion-less Berlekamp-Massey(iBM) algorithm is compared. Further, the serial and inversion-less Berlekamp-Massey(SiBM) algorithm is proposed. Moreover, we will verify the Chien's search and the Forney's algorithm that are used for calculating the error locations and magnitudes respectively.

In Chapter 3, we demonstrate the architecture design of the multi-mode Reed-Solomon codec. The pipeline topology and logic design of each block will be discussed. It includes the multi-mode syndrome calculator that is the first block of RS decoder, and the SiBM architecture with high-speed and low-complexity property in the key equation solver, which is the second block in the RS decoder and is based on the inversion-less Berlekamp-Massey algorithm. Then we explain how we decrease the critical path and simplify the circuit. Moreover, we show the integrated architecture of the Chien-Search and Forney's block, which is the final operation of the Rees-Solomon decoder for finding the error location and error magnitude simultaneously.

For shorten codes, a common compensator is employed to adjust the starting point of search and is combined in the SiBM architecture. In other word, the coefficients of error locator polynomial and the error evaluator polynomial that are feed into Chien Search and Forney Block respectively are adjusted by a common compensator in the SiBM block. It reduces the penalty of that from single-mode to multi-mode greatly.

Chapter 4 shows the implementations of Xinlinx VirtexE xcv2000e FPGA and Synopsys DC with UMC018 library. Also, the design flow, the verification step, the simulation result and the synthesis result are included.

In Chapter 5, we show the advantages of our design and make some comparisons with conventional design graphically. Also, the comparison with publications of RS decoder will be discussed.

Chapter 6 concludes with a summary of contributions and the future works.

Chapter 2

Mathematic Representation of RS codec

The mathematical foundation of Reed-Solomon codes is complicated, but it is important to have a reasonable understanding of in any case what needs to be done, if not why it is done. In this chapter, we will introduce the principle of Reed-Solomon code with regard to our design mathematically.

2.1 Galois Field GF(2^m)

The Reed-Solomon code is defined over the finite field. To proceed further requires some knowledge of the theory of finite fields, otherwise known as Galois fields after the French mathematician.

2.1.1 Galois field element

The construction of a Galois field is a set of elements that are based on a primitive element, usually denoted α, and take the values

$\{0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, ..., \alpha^{2^m-2}\}$ .... (2.1)

to compose a collection of 2^m elements. Thus, the field is known as GF(2^m). Furthermore, n = 2^m- 1, the final element of the 2^m collection can be described as α^{n-1}.

The powers of α form shown in (2.1) are called the index form, and each field element can also be represented by a polynomial form

$$\{ q_{m-1}x^{m-1} + q_{m-2}x^{m-2} + ... + q_1x^1 + q_0 \} \quad (2.2)$$

where the coefficients q_{m-1} to q_0 take the values 0 or 1. So, we can describe a field element using the binary number $(q_{m-1}q_{m-2}......q_1q_0)$ and the 2^m field elements correspond to the 2^m combinations of the *m*-bit number.

The value of α is often chosen to be 2, even if other values can be used. When α is chosen, higher powers can then be obtained by multiplying by α at each step. Notice that the multiplication in a Galois field is different from those that we use in a general field. This will be described later in Section 2.1.4.

2.1.2 Primitive Polynomial p(x)

A weighty part of the Galois field, and therefore of a Reed-Solomon code, is the field primitive polynomial, *p*(*x*). This is a polynomial of degree *m* which is irreducible, that is, a polynomial with no proper factors. It forms part of the process of multiplying two field elements together. For GF(256), the polynomial

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \quad (2.3)$$

is irreducible and therefore will be utilized in the following sections. For a Galois field of a particular size, there is sometimes a choice of adapted polynomials. Notice that choosing different primitive polynomial will produce different results.

2.1.3 Constructing the Galois Field

Before encoding a Reed-Solomon code, we need to construct the Galois field. Let

α be a primitive element in GF(2^m). As shown in (2.1), the 2^m consecutive powers of α must be distinct. The nonzero elements in the Galois field can be constructed by using the principle of that the primitive element α is a root of the field primitive polynomial. In other word, $p(\alpha) = 0$.

Therefore, we can construct the finite field GF(256) by using $p(\alpha) = 0$ in (2.3). Meaning that, we can write $\alpha^8 = \alpha^4 + \alpha^3 + \alpha^2 + 1$. Multiplying by α at each stage, using $\alpha^4 + \alpha^3 + \alpha^2 + 1$ to substitute for α^8 and adding the resulting terms can be used to obtain the complete field as shown in Table 2.1. This shows the field element values in both index and polynomial forms along with the decimal shorthand versions of the polynomial representation.

index form	polynomial form								decimal
	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
0	0	0	0	0	0	0	0	0	0
α^0	0	0	0	0	0	0	0	1	1
α^1	0	0	0	0	0	0	1	0	2
α^2	0	0	0	0	0	1	0	0	4
α^3	0	0	0	0	1	0	0	0	8
α^4	0	0	0	1	0	0	0	0	16
α^5	0	0	1	0	0	0	0	0	32
α^6	0	1	0	0	0	0	0	0	64
α^7	1	0	0	0	0	0	0	0	128
α^8	0	0	0	1	1	1	0	1	29
α^9	0	0	1	1	1	0	1	0	58
~~									
α^{254}	1	0	0	0	1	1	1	0	142

Table 2.1：The construction of the GF(256) elements

If the process shown in Table 2.1 is continued over α^{254} , it is found that $\alpha^{255} = \alpha^{254} * \alpha^1 = 1 = \alpha^0$, $\alpha^{256} = \alpha^1$, ... Consequently, the sequence repeats with all the values remaining valid field elements.

2.1.4 The operation of Galois Field

The addition of two elements in the finite field is an exclusive-OR operation. Meaning that when we add two field elements, we add the two polynomials: $(a_{m-1} x^{m-1} + ... + a_1 x^1 + a_0) + (b_{m-1} x^{m-1} + ... + b_1 x^1 + b_0) = c_{m-1} x^{m-1} + ... + c_1 x^1 + c_0$, where $c_i = a_i \oplus b_i$ and "$\oplus$" is an exclusive-OR operation.

The multiplication of two elements is defined as $\alpha^i \cdot \alpha^j = \alpha^{(i+j) \text{ modulo } (2^m-1)}$. In practice, we often do multiplication of two polynomials straightforwardly. If both of polynomials with degree m-1 results in a polynomial with degree 2m-2, which is therefore not a valid element of GF(2^m). Therefore, multiplication in a Galois field is defined as the product modulo the field primitive polynomial $p(x)$.

	x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
Multiplicand			1	0	0	0	0	0	0	1
Multiplicator		$\otimes$	0	0	0	0	0	1	1	0
		1	0	0	0	0	0	0	1	0
$\oplus$	1	0	0	0	0	0	0	1	0	0
Not Valid Field Element➔	1	1	0	0	0	0	0	1	1	0
						1	1	1	0	1
$\oplus$					1	1	1	0	1	0
Finial Result(Valid Field Element)			0	0	1	0	0	0	0	1

Figure 2.1 : An example of multiplication in Finite Field

Instead of division, the corresponding primitive polynomial is substituting for the nonzero terms that exceed the degree of the field, and adding the remaining terms as shown in Figure 2.1.

2.2 Encoding of Reed-Solomon Codes

The RS (*n*, *k*, *t*) is a code whose codeword are blocks of $n \leq 2^m - 1$ symbols, and each element in GF (2^m) can be represented by *m*-bits. Meaning that there are n symbols and each symbol has m-bits in a codeword.

For the sake of correcting $t = \lfloor \text{n-k} / 2 \rfloor$ symbol errors, 2*t* parity symbols are calculated and appended to a group of information symbols during the encoding process. Therefore, each codeword consists of *k information* symbols and *2t* = *n* – *k* *redundant* symbols. To proceed further, we have to construct the code generator polynomial g(x).

2.2.1 Code Generator Polynomial

The Reed-Solomon code is constructed by forming the code generator polynomial *g*(*x*), consisting of 2*t* factors, the roots of which are consecutive elements of the Galois field which can be calculated by the following equation:

$$g(x) = \prod_{i=0}^{2t-1} (x + \alpha^{b+i}) = (x + \alpha^{b+0})(x + \alpha^{b+1}) \ldots\ldots (x + \alpha^{b+2t-1}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.4)$$

where b is an integer constant which is called the power of the first consecutive root in g(x), and is typically zero or one. However, other choices sometimes simplify the decoding process slightly. While each is valid, it results in a completely different code and requiring changes in both the coder and decoder operation. If the chosen value of *b* is near 2^m-1, then some of the roots may reach or exceed $\alpha^{2^m - 1}$. In this case the index values modulo 2^m-1 will be substituted. Moreover, the factors are often written as $(x - \alpha^i)$, which emphases that $g(x) = 0$ when $x = \alpha^i$ and those familiar with Galois fields realize that $-\alpha^i$ is exactly the same as α^i [20].

It is also, the code generator polynomial takes the form：

$$g(x)=\sum_{i=0}^{2t} g_i x^i = g_0 + g_1x^1 + g_2x^2 + \ldots + g_{2t-1}x^{2t-1} + (g_{2t}=1)x^{2t} \quad \text{.....................} \quad (2.5)$$

which is often employed to construct a single-mode encoder.

2.2.2 Forming the codeword

Let $\{ I_{k-1,} I_{k-2}, ..., I_{1,} I_0 \}$ indicate k *information symbols* and each symbol with m-bit that are to be passed along a communication channel or stored in memory. These symbols are considered as elements of the Galois field, and encoded into a *codeword* $\{ C_{n-1,} C_{n-2}, ..., C_{1,} C_0 \}$ of n symbols that can correct t = $\lfloor$n-k / 2$\rfloor$ symbols of errors.

The k information symbols are preferable represented in the form of the *information polynomial* $I(x) = I_{k-1}x^{k-1} + I_{k-2}x^{k-2} + ... + I_1x^1 + I_0$ .... (2.6)

Then, will be transformed into a *codeword polynomial*, as following shows：

$$C(x) = C_{n-1}x^{n-1} + C_{n-2}x^{n-2} + ... + C_1x^1 + C_0 \quad \text{...} \quad (2.7)$$

At least, there are three kind of process to encode the codeword. To proceed, we discuss the three kinds of encoding process.

Method 1 ：$C(x) = I(x) \cdot g(x)$

The codeword polynomials C(x) are constructed of that information polynomial I(x) multiplied by g(x), the *generator polynomial* of the code, which is defined as (2.4) or (2.5).

The concept of this is very simple. Because of the 2t consecutive elements $\alpha^{b}, \alpha^{b+1}, \alpha^{b+2}, ..., \alpha^{b+2t-1}$ are roots of g(x), and C(x) is a multiple of g(x). Consequently, C(α^{b+i}) = 0, for $0 \le i \le 2t-1$ and any codeword polynomials C(x).

Method 2 ：$C(x) = I(x) \cdot x^{2t} + p(x) = g(x)q(x)$

According to the equation above, the codeword C(x) is consisted of that information symbols I(x) followed by 2t parity-check (remainder) symbols p(x). Moreover, C(x) is a multiple of g(x) by multiplying g(x) with a polynomial q(x). In other words, $I(x) \cdot x^{2t} = g(x)q(x) + p(x)$, where we define q(x) and p(x) as the quotient and remainder respectively while the polynomial $I(x) \cdot x^{2t}$ is divided by g(x). Because of the lowest degree term in $I(x) \cdot x^{2t}$ is $I_0 \cdot x^{2t}$ and deg [p(x)] < 2t, the codeword construction as shown in Figure 2.2 and therefore called a systematic encoding.

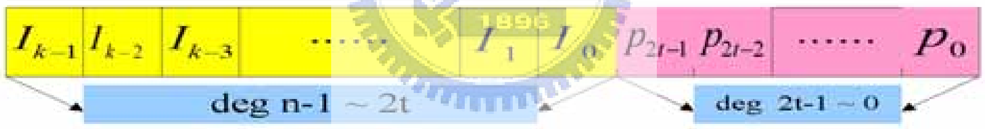


Figure 2.2：The Construction of a Codeword in Systematic Encoding

The advantage of systematic encoding is that we may see the content of a codeword roughly when the error is not serious, but it is not readable by using Method 1. Because of a process in dividing g(x) is needed.

Method 3 ： Factorization form of g(x) (or encoder)

The resulting codeword after encoding is the same as Method 2 (i.e. ***C(x)*** = $I(x) \cdot x^{2t}$ + I(x) mod g(x)), but the process has some difference in g(x). Method 2 uses (2.5) to form g(x), i.e. I(x) modulo (2.5). In this case, Method 3 uses (2.4) to form g(x), that is, I(x) modulo (2.4). In the concept of DSP, we can use the z-transform 1/g(z) as a digital filter to explain it as shown in (2.8).

$$\frac{1}{g(z)} = \frac{1}{1+\alpha^{b}z^{-1}} \cdot \frac{1}{1+\alpha^{b+1}z^{-1}} \cdots\cdots \frac{1}{1+\alpha^{b+2t-1}z^{-1}} \quad \text{(2.8)}$$

It is suitable for multi-mode applications and can reduce the hardware greatly by combining encoder and syndrome calculator. Unfortunately, there is a little penalty in increasing the critical path. The schematic of both Method 2 and Method 3 will be discussed in Chapter 3.

2.3 Decoding of Reed-Solomon Codes

The RS code belongs to a non-binary BCH codes. In the decoding of binary BCH code, we just need to calculate the error locations and then the error values must be 1. Nevertheless, a step involving the computation of error values is required in the decoding of RS codes.

There are various algorithms in decoding the RS codes. Roughly, we could divide it into three categories, i.e. Time Domain Decoding, Frequency Domain Decoding, and Transform Approach Domain Decoding, as shown in Figure 2.3：

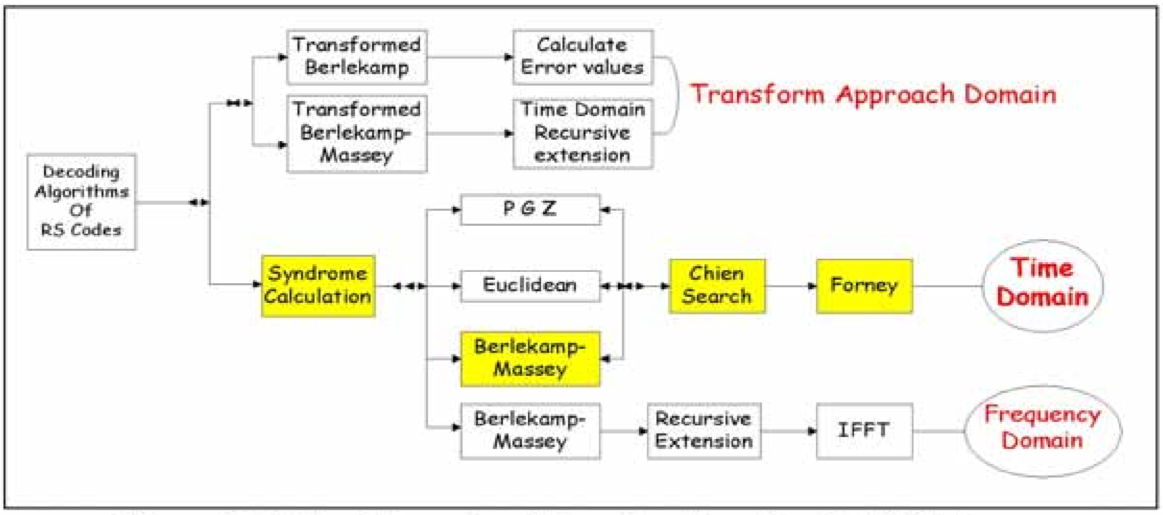


Figure 2.3：The Categories of Decoding Algorithm in RS Codes

As shown in the yellow mark of Figure 2.3, our design focuses on the

Berlekamp-Massey algorithms. We will introduce the time domain decoding mathematically in the following sections.

To proceed further requires the understanding in the constitution of the receiving codeword. The transmitted code polynomial C(x) has described in (2.7). Then let

$$R(x) = R_{n-1}x^{n-1} + \cdots\cdots + R_1 x + R_0 \quad \ldots\ldots (2.9)$$

be the corresponding received polynomial. Assume that υ errors have been induced by channel noises during transmission, in the form of an error polynomial, E(x). Thus the received polynomial, R(x), is given by：

$$R(x) = C(x) + E(x) \quad \ldots\ldots (2.10)$$

where $E(x) = e_{n-1}x^{n-1} + \cdots\cdots + e_1 x + e_0$ and $e_i = R_i \oplus C_i$ is a m-bit symbol in GF(2^m) with the positions of the errors in the codeword being determined by the degree of x for that term. It is also, the error pattern E(x) contains υ errors at locations $x^{L1}, x^{L2}, \cdots\cdots, x^{Lv}$ with error values $\varepsilon_1, \varepsilon_2, \cdots\cdots, \varepsilon_\upsilon$, the error polynomial E(x) can also be represented as

$$E(x) = \varepsilon_1 x^{L1} + \varepsilon_2 x^{L2} + \cdots\cdots + \varepsilon_v x^{Lv} \quad \ldots\ldots (2.11)$$

Clearly, to determine E(x), we need to know the υ pairs (ε_i, x^{Li})'s. If no more than t = $\lfloor \text{n-k} / 2 \rfloor$ of the ε_i values are valid (nonzero), the correction capacity of the code is not exceeded and the errors are correctable.

2.3.1 Syndrome Calculation

When receiving a codeword of Reed-Solomon code, we have to check if there is any error in it. As the transmitted codeword C(x) is always divisible by the generator polynomial g(x) and that this property extends to the individual factors (x+α^i) of the generator polynomial, (2.10) can be denoted as

$$\mathrm{R}(\alpha^i) = \mathrm{C}(\alpha^i) + \mathrm{E}(\alpha^i) = 0 + \mathrm{E}(\alpha^i) = \mathrm{E}(\alpha^i) \quad \text{……………………………………. (2.12)}$$

For each ***i*** =b, b+1,......,b+2t-1, the value R(α^i) is defined as the syndrome value. That is, from (2.9) and *for **i** =b, b+1,......,b+2t-1*

$$S_i = R(\alpha^i) = R_{n-1}(\alpha^i)^{n-1} + R_{n-2}(\alpha^i)^{n-2} + \cdots\cdots + R_1\alpha^i + R_0 \quad \text{……………………… (2.13)}$$

This means that each of the syndrome values can also be obtained by substituting $x = \alpha^i$ in the received polynomial. Moreover, in (2.12) and (2.13) between the syndromes S_i and the error polynomial E(x) can be used to produce a set of simultaneous equations from which the errors can be found. If choosing b=0 and then substituting $\alpha^0, \alpha^1, \cdots, \alpha^{2t-1}$ for x into (2.11), the 2t syndrome equations can be rewritten as：

$$\left.\begin{array}{l} S_0 = R(\alpha^0) = \varepsilon_1 \cdot (\alpha^0)^{L1} + \varepsilon_2 \cdot (\alpha^0)^{L2} + \cdots\cdots + \varepsilon_v \cdot (\alpha^0)^{Lv} \\ S_1 = R(\alpha^1) = \varepsilon_1 \cdot (\alpha^1)^{L1} + \varepsilon_2 \cdot (\alpha^1)^{L2} + \cdots\cdots + \varepsilon_v \cdot (\alpha^1)^{Lv} \\ \cdots\cdots \\ S_{2t-1} = R(\alpha^{2t-1}) = \varepsilon_1 \cdot (\alpha^{2t-1})^{L1} + \varepsilon_2 \cdot (\alpha^{2t-1})^{L2} + \cdots\cdots + \varepsilon_v \cdot (\alpha^{2t-1})^{Lv} \end{array}\right\} \quad \text{……….. (2.14)}$$

To clear (2.14), we let $\beta_i = \alpha^{Li}$ to simplify error-location numbers and substitute it into corresponding equations in (2.14). Then, the equations can be rewritten as：

$$\left.\begin{array}{l} S_0 = R(\alpha^0) = \varepsilon_1 \cdot 1 + \varepsilon_2 \cdot 1 + \cdots\cdots + \varepsilon_v \cdot 1 \\ S_1 = R(\alpha^1) = \varepsilon_1\beta_1^{\,1} + \varepsilon_2\beta_2^{\,1} + \cdots\cdots + \varepsilon_v\beta_v^{\,1} \\ \cdots\cdots \\ S_{2t-1} = R(\alpha^{2t-1}) = \varepsilon_1\beta_1^{\,2t-1} + \varepsilon_2\beta_2^{\,2t-1} + \cdots\cdots + \varepsilon_v\beta_v^{\,2t-1} \end{array}\right\} \quad \text{……………………….. (2.15)}$$

By computing the syndromes, we can determine if we need to correct the codeword or not. If all the syndrome components are equal to zero, there is no error occurs and the data is valid. Oppositely, if not all syndromes are equal to zero, we have to do some process for data recovery. First of all, we need to find error location polynomial by some algorithms described in next section.

2.3.2 The Key Equation Solver

We define the Key Equation Solver Block including following two steps：

1. Determine the error-location polynomial $\Lambda(x)$.
2. Determine the error-value evaluator $\Omega(x)$.

The error locator polynomial with v actual errors is defined as：

$$\Lambda(x) = (1+\beta_1 x)(1+\beta_2 x)......(1+\beta_v x) = 1+\Lambda_1 x+\Lambda_2 x^2+......+\Lambda_v x^v \quad .... (2.16)$$

Then, the error-value evaluator polynomial is

$$\Omega(x) = \Lambda(x)S(x) \bmod x^{2t} = \Omega_0+\Omega_1 x^1+......+\Omega_{v-1}x^{v-1} \quad .... (2.17)$$

where S(x) is defined as $S(x) = S_0 + S_1 x^1 + ... + S_{2t-1}x^{2t-1}$ .... (2.18)

To find error-location numbers and error magnitudes, we need to get the coefficients Λ_i's in (2.17) and Ω_i's in (2.18) first. The computation of error locator polynomial is the most complex block in the whole decoding process. As shown in Figure 2.3, there are various algorithms for this propose.

In this section, we introduce some of the time-domain decoding algorithms with regard to our design.

2.3.2.1 The Peterson-Gorenstein-Zierler Algorithm

The Peterson-Gorenstein-Zierler decoding algorithm is the first explicit algorithm developed in 1960 for binary BCH code, and is the basic concept to let us know how to decode Reed-Solomon codes.

The error-location numbers are the reciprocals of the roots of $\Lambda(x)$. Let X_L be one of the error-location number and therefore to substitute the reciprocal X_L^{-1} into

(2.16), we can derive $1+\Lambda_1 X_L^{-1}+\Lambda_2 X_L^{-2}+......+\Lambda_v X_L^{-v}=0$ .... (2.19)

Now, let ε_{jL} be the error magnitude with regard to the error location and then multiplying $\varepsilon_{jL} X_L^{j}$ in the both sides of (2.19), we get

$$\varepsilon_{jL} X_L^{j}(1+\Lambda_1 X_L^{-1}+\Lambda_2 X_L^{-2}+......+\Lambda_v X_L^{-v})=0$$

$$\Rightarrow \varepsilon_{jL}(X_L^{j}+\Lambda_1 X_L^{j-1}+\Lambda_2 X_L^{j-2}+......+\Lambda_v X_L^{j-v})=0 \quad (2.20)$$

Substituting L = {1, 2,…,v} into (2.20) and adding the resulting terms, we get

$$\sum_{L=1}^{v}\varepsilon_{jL}(X_L^{j}+\Lambda_1 X_L^{j-1}+\Lambda_2 X_L^{j-2}+......+\Lambda_v X_L^{j-v})=0$$

$$\Rightarrow \sum_{L=1}^{v}\varepsilon_{jL}X_L^{j}+\Lambda_1\sum_{L=1}^{v}\varepsilon_{jL}X_L^{j-1}+\Lambda_2\sum_{L=1}^{v}\varepsilon_{jL}X_L^{j-2}+......+\Lambda_v\sum_{L=1}^{v}\varepsilon_{jL}X_L^{j-v}=0$$

$$\Rightarrow S_j+\Lambda_1 S_{j-1}+\Lambda_2 S_{j-2}+......+\Lambda_v S_{j-v}=0$$

$$\Rightarrow \Lambda_1 S_{j-1}+\Lambda_2 S_{j-2}+......+\Lambda_v S_{j-v}=-S_j \quad (2.21)$$

Let v = t and substituting j = (t), (t+1),……,(2t-1) into (2.21), we get

$$\zeta\cdot\Lambda=\begin{bmatrix} S_{t-1} & S_{t-2} & \cdots S_0 \\ S_t & S_{t-1} & \cdots S_1 \\ & \vdots & \\ S_{2t-2} & S_{2t-1} & \cdots S_{t-1}\end{bmatrix}\begin{bmatrix}\Lambda_1\\ \Lambda_2\\ \vdots\\ \Lambda_t\end{bmatrix}=-\begin{bmatrix}S_t\\ S_{t+1}\\ \vdots\\ S_{2t-1}\end{bmatrix} \quad (2.22)$$

Then, the coefficients Λ_i of error locator polynomial in (2.16) can be solved by matrix inversion of (2.22). If the error count is equal to t, matrix ζ is nonsingular. Otherwise, if there are less than t errors, ζ is singular. We have to delete the right-most columns as well as lowest rows repeatedly until the matrix ζ is nonsingular.

As soon as the error-location numbers β_L 's have been calculated, the error magnitudes ε_i 's can be derive by matrix inversion of (2.23) that is similar to (2.15).

$$\beta \cdot \varepsilon = \begin{bmatrix} \beta_1 & \beta_2 & \cdots \beta_v \\ \beta_1^2 & \beta_2^2 & \cdots \beta_v^2 \\ & \vdots & \\ \beta_1^v & \beta_2^v & \cdots \beta_v^v \end{bmatrix} \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_v \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_v \end{bmatrix} \quad \text{(2.23)}$$

Now we know the error locations as well as error values, the decoding process is done. Summarized, the PGZ decoding algorithm including follow 9 steps：

1. Compute the syndrome values $S_0, S_1, ..., S_{2t-1}$, where $S_i = R(\alpha^i)$.

2. Construct the syndrome matrix ζ , and compute the determinant of this matrix. If the determinant is nonzero, go to step 4.

3. Simplify the syndrome matrix by deleting the rightmost column as well as the bottom row from the original syndrome matrix. Go to step 2.

4. Do matrix inversion operation of (2.22), and we get Λ_i 's.

5. Construct the error-location polynomial $\Lambda(x)$ such as (2.16), and find its roots. If it has the same roots or no root, go to step 9.

6. Construct the matrix β and (2.23), do matrix inversion to get the error magnitudes ε_i 's.

7. Correcting the received codeword by adding the error magnitudes with regard to the corresponding error-location numbers.

8. Finish the decoding process and send the codeword out.

9. Declare a decoding fail or request a resend command.

The PGZ algorithm is uncomplicated in concept, but the hardware complexity of a T*T matrix is increasing cube with T. In order to replace the matrix operation and decode the RS code more efficient, we will introduce the Berelekamp-Massey algorithm in the following.

2.3.2.2 The Berlekamp-Massey Algorithm

In this section, we re-derive the equations similar to the expansion of the matrix in (2.22) and solve it by the Belekamp's algorithm [21]. Then, the Berelekamp-Massey algorithm **(BMA)** and the inversion-less Berelekamp- Massey algorithm [22] **(iBMA)** will be introduced. Finally, the Serial and inversion-less Berlekamp–Massey **(SiBM)** algorithm will be proposed.

Note that there are some differences from Lin's text book [23] or some other technical literature by choosing different integer constant "b" which is called the power of the first consecutive root in g(x) , and different starting point of iteration "i". By this way, we can optimize our design in the hardware consideration. So, we let b=0 as well as i = (1 ~ 2t) from stem to stern and the advantages will be mentioned in the end.

- **The Berlekamp's algorithm :**

We can also define the syndrome polynomial S(x) as

$$S(x)=\sum_{i=0}^{\infty} S_i x^i \quad\ldots\ldots (2.24)$$

Notice that only the coefficients of the first 2t terms above are known. It is also,

$$S_i=\sum_{L=1}^{v} \varepsilon_L \beta_L{}^i \qquad for\ \ 0\le i\le\infty \quad\ldots\ldots (2.25)$$

If we extend (2.25) from i=0 to i=(2t-1), we can find that the 2t such S_i 's are simply the 2t equalities of (2.15). Substituting S_i of (2.25) into (2.24), we can put S(x) in the following form :

$$S(x)=\sum_{i=0}^{\infty} x^i \sum_{L=1}^{v} \varepsilon_L \beta_L{}^i=\sum_{L=1}^{v}\varepsilon_L\sum_{i=0}^{\infty}(\beta_L x)^i \quad\ldots\ldots (2.26)$$

and where $\sum_{i=0}^{\infty}(\beta_L x)^i = \dfrac{1}{1-\beta_L x}$ .... (2.27)

That is, (2.26) can be re-written as $S(x) = \sum_{L=1}^{v} \dfrac{\varepsilon_L}{1-\beta_L x}$ .... (2.28)

Then, we can derive $\Lambda(x)S(x)$ in the following form：

$$\begin{aligned}\Lambda(x)S(x) &= \left\{\prod_{k=1}^{v}(1-\beta_k x)\right\}\cdot\left\{\sum_{L=1}^{v}\frac{\varepsilon_L}{1-\beta_L x}\right\} = \sum_{L=1}^{v}\frac{\varepsilon_L}{1-\beta_L x}\cdot\prod_{k=1}^{v}(1-\beta_k x)\\ &= \sum_{L=1}^{v}\varepsilon_L \prod_{k=1,k\neq L}^{v}(1-\beta_k x)\end{aligned} \quad \text{.............. (2.29)}$$

Besides, consider the product $\Lambda(x)S(x)$ of two polynomials directly, we get

$$\left.\begin{aligned}&\Lambda(x)S(x) = (1+\Lambda_1 x+\Lambda_2 x^2+\ldots+\Lambda_v x^v)\cdot(S_0+S_1x^1+\ldots+S_{2t-1}x^{2t-1}+\ldots)\\ &= S_0+(S_1+\Lambda_1S_0)x+(S_2+\Lambda_1S_1+\Lambda_2S_0)x^2+\ldots+(S_{v-1}+\Lambda_1S_{v-2}+\ldots+\Lambda_{v-1}S_0)x^{v-1}\\ &+(S_v+\Lambda_1S_{v-1}+\ldots+\Lambda_vS_0)x^v+\ldots+(S_{2t-1}+\Lambda_1S_{2t-2}+\ldots+\Lambda_vS_{2t-1-v})x^{2t-1}+\ldots\end{aligned}\right\} \ldots$$

.... (2.30)

If (v = t) and assume the coefficients of the terms that greater than x^{v-1} are equal to zero, the coefficients with red mark in (2.30) are similar to the expansion of (2.22). Fortunately, the assumption above is valid. Why?

Note that if we unfold (2.29), the equation is just a polynomial of degree v-1 and is equal to (2.30). Meaning that the coefficients of the terms that greater than x^{v-1} in (2.30) are equal to zero, the assumption holds.

Furthermore, as (2.29), (2.30) and the definition of $\Omega(x)$ in (2.17), we can derive

$$\left.\begin{aligned}\Omega(x) = \Lambda(x)S(x) \bmod x^{2t} &= S_0+(S_1+\Lambda_1S_0)x+(S_2+\Lambda_1S_1+\Lambda_2S_0)x^2+\\ &\ldots+(S_{v-1}+\Lambda_1S_{v-2}+\ldots+\Lambda_{v-1}S_0)x^{v-1}\end{aligned}\right\} \text{........ (2.31)}$$

and $\Omega(x) = \sum_{L=1}^{v}\varepsilon_L \prod_{k=1,k\neq L}^{v}(1-\beta_k x)$ .... (2.32)

Equation (2.32) will be exploited to derive the error magnitudes in the Forney's algorithm later. Now, let us proceed to find the coefficients Λ_i's of error-location polynomial $\Lambda(x)$.

As mentioned above, the degree of $\Omega(x)$ is v-1, the coefficient from x^v to x^{2t-1}

in the expansion of $\Lambda(x)S(x)$ must be zero. By setting these coefficients to zero, we have exactly the same set of (2.22). That is,

$$\left.\begin{array}{l} S_v + \Lambda_1 S_{v-1} + \ldots + \Lambda_v S_0 = 0 \\ S_{v+1} + \Lambda_1 S_v + \ldots + \Lambda_v S_1 = 0 \\ \qquad\qquad \vdots \\ S_{2t-1} + \Lambda_1 S_{2t-2} + \ldots + \Lambda_v S_{2t-1-v} = 0 \end{array}\right\} \cdots\cdots (2.33)$$

This set of equations is known as the generalized Newton's identities. Similar to the operation of the PGZ algorithm in Section 2.3.2.1, find the simplest set of matrix ζ, our objective is to find the minimum-degree polynomial $\Lambda(x)$ whose coefficients satisfy these generalized Newton's identities.

Instead of the complex matrix inversion, we can compute the error location polynomial $\Lambda(x)$ iteratively in the 2t iterations of Berlekamp's algorithm. Suppose we have just finished the i-th step and found the solution

$$\Lambda^{(i)}(x) = \Lambda_0^{(i)} + \Lambda_1^{(i)}x + \Lambda_2^{(i)}x^2 + \ldots + \Lambda_{\delta i}^{(i)}x^{\delta i} \cdots\cdots (2.34)$$

that is the minimum-degree polynomial whose coefficients satisfy the first i Newton's identities of (2.33) and where δi is the degree of $\Lambda^{(i)}(x)$. To determine $\Lambda^{(i+1)}(x)$, we have to check whether the coefficient of $\Lambda^{(i)}(x)$ satisfy the next generalized Newton's identity. We compute the following quantity：

$$\Delta_i = S_i + \Lambda_1^{(i)}S_{i-1} + \ldots + \Lambda_{\delta i}^{(i)}S_{i-\delta i} \cdots\cdots (2.35)$$

This quantity Δ_i is called the i-th discrepancy. If Δ_i =0, the coefficients of $\Lambda^{(i)}(x)$ satisfy the (i+1)th Newton's identity. In this event, we set $\Lambda^{(i+1)}(x) = \Lambda^{(i)}(x)$. On the contrary, $\Delta_i \neq 0$, we must add a correction term to $\Lambda^{(i)}(x)$ to obtain $\Lambda^{(i+1)}(x)$. To make this correction, we go back to the step prior to the i-th step and determine a polynomial C(x) which is the minimum-degree polynomial at the γ-th step of iteration such that the γ-th discrepancy $\Delta_\gamma \neq 0$ and γ - δ_γ has the largest value. Then,

$$\Lambda^{(i+1)}(x) = \Lambda^{(i)}(x) + \frac{\Delta_i}{\Delta_\gamma} \cdot C(x) \cdot x^{(i-\gamma)} \quad \text{......} \quad (2.36)$$

which is the solution at the (i+1)-th step of the iteration process and the coefficients satisfy the first i+1 Newton's identities. Continue the foregoing process until 2t steps have been completed, we get $\Lambda(x) = \Lambda^{(2t)}(x)$ is the true error-location polynomial whose coefficients satisfy the set of the generalized Newton's identities given by (2.33).

In the beginning, the initial condition of $\Lambda(x)$ and Δ_i is shown as bright yellow mark of Table 2.2. To carry out the iteration of finding $\Lambda(x)$, we proceed to fill out the table form i=1 to i=2t by determining Δ_i in (2.35) and computing $\Lambda^{(i+1)}(x)$ as (2.36) that have described above. After 2t iterations, the error-location polynomial $\Lambda(x)$ is equal to $\Lambda^{(2t)}(x)$. Then, we can derive the coefficients of $\Omega(x)$ by substituting S_i's as well as Λ_i's into (2.31) and calculating it.

i	$\Lambda^{(i)}(x)$	Δ_i	δ_i	i - δ_i
-1	$1 = C(x)$	1	0	-1
0	1	S_0	0	0
1	$1+S_0x$	$S_1+\Lambda_1S_0$	⋮	⋮
2	⋮	⋮		
3				
⋮				
2t				

Table 2.2：Iterative procedure for finding the error-location polynomial $\Lambda(x)$

Having calculated the coefficient values of $\Lambda(x)$ and $\Omega(x)$, it is possible to find the error-location numbers and error magnitudes by Chien's Search algorithm and

Forney's algorithm respectively. Nevertheless, we stop a story at a climax to keep the listeners in suspense for the moment and proceed to discuss the Berlekamp-Massey algorithm.

Conventional Berlekamp-Massey algorithm (BMA)：

After one year, in 1968, the Berlekamp's algorithm was re-explained by Massey with shortest linear feedback shift register (LFSR) concept that has regular property for decoding key equation and is so-called Berlekamp-Massey algorithm.

By Berlekamp-Massey algorithm, (2.22) or (2.33) is just an autoregressive filter and the Λ_i 's are regarded as tap coefficients. As Figure 2.4 shows, it can be implemented by a so-called LFSR and the discrepancy Δ_i is employed to verify the LFSR to generate the corresponding syndrome sequence at each step.

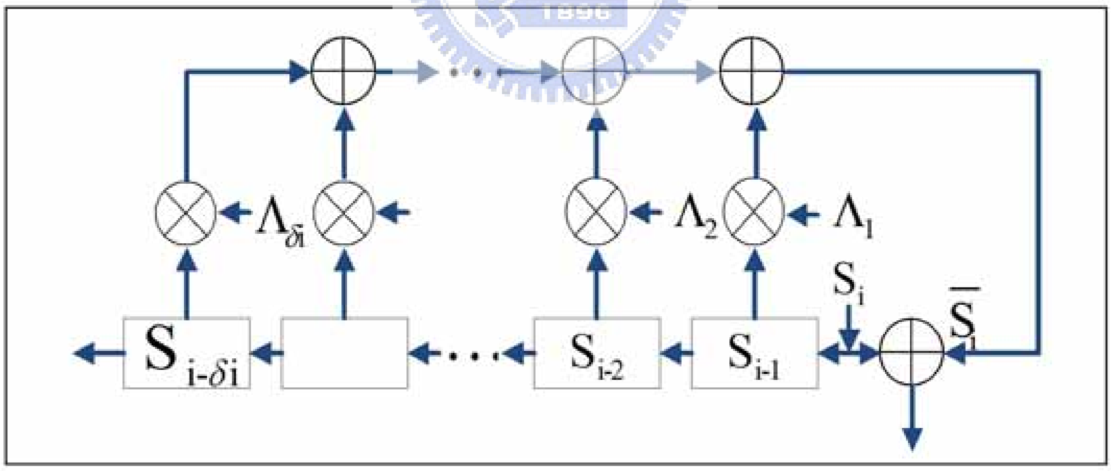


Figure 2.4：The LFSR Structure of Berlekamp-Massey algorithm.

At the i-th step, we get $\overline{S_i}$ and the discrepancy

$$\Delta_i = S_i - (-\overline{S_i}) = \sum_{j=0}^{\delta_i} \Lambda_j^{(i)} S_{i-j} \qquad (2.37)$$

If you are attentive, you could easily find the extension of (2.37) is exactly equal to

(2.35). Now, let us eliminate the discrepancy, i.e. calculating $\Lambda^{(i+1)}(x)$ that satisfy

$$\overline{\Delta_i} = \sum_{j=0}^{\delta_i} \Lambda_j^{(i+1)} S_{i-j} = 0 \quad \text{(2.38)}$$

As a result of $\Delta_i + \Delta_i = 0$, we can derive

$$\sum_{j=0}^{\delta_i} \Lambda_j^{(i)} S_{i-j} + \frac{\Delta_i}{\Delta_\gamma} \cdot \sum_{j=0}^{\delta_\gamma} \Lambda_j^{(\gamma)} S_{\gamma-j} = 0 \quad \text{(2.39)}$$

and can be re-written as

$$\sum_{j=0}^{\delta_i} (\Lambda_j^{(i)} + \frac{\Delta_i}{\Delta_\gamma} \cdot \Lambda_{j-(i-\gamma)}^{(\gamma)}) S_{i-j} = 0 \quad \text{(2.40)}$$

Compare with (2.38) and (2.40), we can get

$$\Lambda_j^{(i+1)} = \Lambda_j^{(i)} + \frac{\Delta_i}{\Delta_\gamma} \cdot \Lambda_{j-(i-\gamma)}^{(\gamma)} \quad \text{(2.41)}$$

and can be re-explained as

$$\Lambda^{(i+1)}(x) = \Lambda^{(i)}(x) + \frac{\Delta_i}{\Delta_\gamma} \cdot \Lambda^{(\gamma)}(x) \cdot x^{(i-\gamma)} \quad \text{(2.42)}$$

In fact, (2.42) is equivalent to (2.36) by setting $\Lambda^{(\gamma)}(x) = C(x)$ and $C(x)$ is the so-called correction polynomial.

As Berlekamp's algorithm, setting the initial condition followed by calculating (2.37) and (2.41) iteratively from i=1 to i=2t, we can get the error-location polynomial. Summarized, the entire BMA is shown as:

```
1. Setting the initial condition:
   $\delta_0 = 0,\ \Lambda^{(0)}(x) = 1,\ \Delta_0 = S_0$
   $C^{(0)}(x) = \Lambda^{(-1)}(x) = 1,\ \Delta_C = \Delta_{-1} = 1$
2. For (i=1 to 2t)
       $\Lambda^{(i)}(x) = \Lambda^{(i-1)}(x) + \frac{\Delta_{i-1}}{\Delta_C} \cdot C^{(i-1)}(x) \cdot x$
       $\Delta_i = \sum_{j=0}^{\delta_i} \Lambda_j^{(i)} S_{i-j} = \Lambda_0^{(i)} S_i + \Lambda_1^{(i)} S_{i-1} + \ldots + \Lambda_{\delta i}^{(i)} S_{i-\delta i}$
```

If $\Delta_{i-1} = 0$ or $i \le 2 \cdot \delta_{i-1}$

$$\delta_i = \delta_{i-1},\ C^{(i)}(x) = C^{(i-1)}(x) \cdot x,\ \Delta_C = \Delta_C$$

Else

$$\delta_i = i - \delta_{i-1},\ C^{(i)}(x) = \Lambda^{(i-1)}(x),\ \Delta_C = \Delta_{i-1}$$

where δ_i is the an auxiliary degree variable, $\Lambda^{(i)}(x)$ is the i-th dummy error-location polynomial, and Δ_i is the i-th dummy discrepancy. Besides, $C^{(i)}(x)$ is the correction polynomial for $\Lambda^{(i)}(x)$, and Δ_C is a previous nonzero discrepancy. If the discrepancy Δ_i is equal to zero, it represent that we don't update the error-location polynomial $\Lambda^{(i+1)}(x)$ at step (i+1). After 2t iterations, the error-location polynomial $\Lambda(x)$ is equivalent to the dummy polynomial $\Lambda^{(2t)}(x)$.

So, we can say that the BMA is just a shortest-LFSR version of the Berlekamp's algorithm. If there is something confusing, please refer to Table 2.2 of the Berlekamp's algorithm.

The inversion-less Berlekamp-Massey algorithm (iBMA):

Why an inversion-less version is needed? Let us look back to (2.42), the equation used for updating the error-location polynomial, the conventional BMA is including a division operation in the finite field that is more complex and very time-consuming. In contrast with the conventional Berlekamp-Massey algorithm, the inversion-less Berlekamp-Massey algorithm needs no divider or inverse ROM for updating the polynomial. To avoid the division operation, the inversion-less Berlekamp-Massey algorithm (iBMA) is essential.

It should be noted that the iBMA finds the multiple of $\Lambda(x)$ instead of $\Lambda(x)$ and therefore $\Lambda_0 \neq 1$ anymore, we need to store the coefficient Λ_0 when we using the iBMA. Then, this algorithm is described as follows:

```
1. Setting the initial condition:
    $\delta_0 = 0, \boxed{\Lambda^{(0)}}(x) = 1,\ \boxed{\Delta_0} = S_0$
    $\boxed{C^{(0)}}(x) = \Lambda^{(-1)}(x) = 1,\ \boxed{\Delta_C} = \Delta_{-1} = 1$
2. For i = (1 ~ 2t)
        $\boxed{\Lambda^{(i)}}(x) = \boxed{\Delta_C} \cdot \boxed{\Lambda^{(i-1)}}(x) + \boxed{\Delta_{i-1}} \cdot \boxed{C^{(i-1)}}(x) \cdot x$
        $\boxed{\Delta_i} = \sum_{j=0}^{\delta_i} \boxed{\Lambda_j^{(i)}} S_{i-j} = \boxed{\Lambda_0^{(i)}} S_i + \boxed{\Lambda_1^{(i)}} S_{i-1} + \ldots + \boxed{\Lambda_{\delta i}^{(i)}} S_{i-\delta i}$
If $\boxed{\Delta_{i-1}} = 0$ or $i \leq 2 \cdot \delta_{i-1}$
        $\delta_i = \delta_{i-1},\ \boxed{C^{(i)}}(x) = \boxed{C^{(i-1)}}(x) \cdot x,\ \boxed{\Delta_C} = \boxed{\Delta_C}$
    Else
        $\delta_i = i - \delta_{i-1},\ \boxed{C^{(i)}}(x) = \boxed{\Lambda^{(i-1)}}(x),\ \boxed{\Delta_C} = \boxed{\Delta_{i-1}}$
```

where $\boxed{\Lambda^{(i)}}(x)$ represents a multiple of $\Lambda^{(i)}(x)$ and is similarly to $\boxed{\Delta_i}$, $\boxed{C^{(i)}}(x)$ and $\boxed{\Delta_C}$. Furthermore, the error magnitude polynomial $\Omega(x)$ could be calculated with $\Lambda(x)$ simultaneously or substituting S_i's as well as Λ_i's into (2.31) and calculating it after the 2t iteration above.

For computing $\Omega(x)$ with $\Lambda(x)$ simultaneously, we need to set the initial condition $\boxed{\Omega^{(0)}}(x) = 1$ and $\boxed{\gamma^{(0)}}(x) = 0$, then calculate

```
for i = 1~2t
    $\boxed{\Omega^{(i)}}(x) = \boxed{\Delta_C} \cdot \boxed{\Omega^{(i-1)}}(x) + \boxed{\Delta_{i-1}} \cdot \boxed{\gamma^{(i-1)}}(x) \cdot x$
    if $\boxed{\Delta_{i-1}} = 0$ or $i \leq 2 \cdot \delta_{i-1}$ ➔ $\boxed{\gamma^{(i)}}(x) = \boxed{\gamma^{(i-1)}}(x) \cdot x$
    else $\boxed{\gamma^{(i)}}(x) = \boxed{\Omega^{(i-1)}}(x)$
```

where $\boxed{\Delta_C}$ as well as $\boxed{\Delta_{i-1}}$ are the same with the iteration of $\Lambda(x)$ and $\boxed{\Omega^{(i)}}(x)$ is a multiple of $\Omega^{(i)}(x)$. It is also, $\Lambda(x)=\boxed{\Lambda^{(2t)}}(x)$ and $\Omega(x)=\boxed{\Omega^{(2t)}}(x)$.

In the iBMA, the finite filed inverter is replaced by a multiplier and it can be proved that it doesn't have any influence on computing the correct result by induction.

2.3.2.3 Proposed Serial and inversion-less Berlekamp-Massey algorithm (SiBM)：

Most implementations of Reed-Solomon decoder choose the iBMA for their error-location updating block in the past. By this algorithm, the parallel architecture is formed and always require 2t~3t finite field multipliers [24,25]. If we do some transformation mathematically, similar to the form of (2.41), we can get different equation set which is so-called SiBM algorithm and is helpful for implementation.

Then, the equation of error-location polynomial $\boxed{\Lambda^{(i)}}(x)$ at i-th step can be re-written as

$$\boxed{\Lambda_j^{(i)}}=\boxed{\Delta_C}\cdot\boxed{\Lambda_j^{(i-1)}}+\boxed{\Delta_{i-1}}\cdot\boxed{C_{j-1}^{(i-1)}} \quad for \quad j=0\sim\delta_i \ldots\ldots\ldots\ldots\ldots\ldots (2.43)$$

where $\boxed{\Lambda_j^{(i-1)}}$ is the j-th term of the (i-1)-th iteration, and so is $\boxed{C_{j-1}^{(i-1)}}$. Notice that $\boxed{C_{-1}^{(i-1)}}=0$, when j=0. Similarly, the error magnitude polynomial $\boxed{\Omega^{(i)}}(x)$ can be represented as

$$\boxed{\Omega_j^{(i)}}=\boxed{\Delta_C}\cdot\boxed{\Omega_j^{(i-1)}}+\boxed{\Delta_{i-1}}\cdot\boxed{\Upsilon_{j-1}^{(i-1)}} \quad for \quad j=0\sim\delta_i \ldots\ldots\ldots\ldots\ldots\ldots (2.44)$$

and the discrepancy $\boxed{\Delta_i}$ can be re-written as

$$\boxed{\Delta_i}=\sum_{j=0}^{\delta_i}\boxed{\Lambda_j^{(i)}}S_{i-j}=0+\sum_{j=1}^{\delta_i+1}\boxed{\Lambda_{j-1}^{(i)}}S_{i-j+1} \ldots\ldots\ldots\ldots\ldots\ldots (2.45)$$

Why we change above equation from i = (0 ~ δ_i) to i = (1~ δ_i +1)? This is the critical path consideration. At j=0 sub-iteration, we must calculate $\boxed{\Lambda_0^{(i)}}$ first, so as to get the first term of $\boxed{\Delta_i}$, i.e. $\boxed{\Delta_0^{(i)}}$. To reduce the timing path, we let $\boxed{\Delta_0^{(i)}}$ to be calculated at j = 1 and re-write (2.45) as

$$\boxed{\Delta_{j-1}^{(i)}} = \boxed{\Delta_{j-2}^{(i)}} + \boxed{\Lambda_{j-1}^{(i)}} S_{i-j+1} \quad \textit{for} \quad j = 1 \sim \delta_i + 1 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(2.46)}$$

Notice that in the reason of we excessive emphasis $\boxed{\Delta_0^{(i)}} = \boxed{\Lambda_0^{(i)}} S_i$, we must let $\boxed{\Delta_{-1}^{(i)}} = \boxed{\Delta_{-2}^{(i)}} + \boxed{\Lambda_{-1}^{(i)}} \cdot S_{i+1} = 0$.

Summarized, the entire SiBM is shown as

```
1. Setting the initial condition:
   $\delta_0 = 0, \boxed{\Lambda^{(0)}}(x) = 1,\ \boxed{\Delta^{(0)}} = \boxed{\Delta_0} = S_0$
   $\boxed{C^{(0)}}(x) = \Lambda^{(-1)}(x) = 1,\ \boxed{\Delta_C} = \Delta_{-1} = 1$
   $\boxed{\Omega^{(0)}}(x) = 1,\ \boxed{\gamma^{(0)}}(x) = 0$
2. for i = (1 ~ 2t)
   $\boxed{\Delta_{-1}^{(i)}} = \boxed{C_{-1}^{(i-1)}} = \boxed{\gamma_{-1}^{(i-1)}} = 0$
     for j = $(0 \sim \delta_i)$
       $\boxed{\Lambda_j^{(i)}} = \boxed{\Delta_C} \cdot \boxed{\Lambda_j^{(i-1)}} + \boxed{\Delta^{(i-1)}} \cdot \boxed{C_{j-1}^{(i-1)}}$
       $\boxed{\Omega_j^{(i)}} = \boxed{\Delta_C} \cdot \boxed{\Omega_j^{(i-1)}} + \boxed{\Delta^{(i-1)}} \cdot \boxed{\Upsilon_{j-1}^{(i-1)}}$
       $\boxed{\Delta_{j-1}^{(i)}} = \boxed{\Delta_{j-2}^{(i)}} + \boxed{\Lambda_{j-1}^{(i)}} S_{i-j+1}$
     for j = $(\delta_i + 1)$ $\rightarrow$ $\boxed{\Delta^{(i)}} = \boxed{\Delta_{\delta_i}^{(i)}} = \boxed{\Delta_{\delta_i - 1}^{(i)}} + \boxed{\Lambda_{\delta_i}^{(i)}} S_{i-\delta_i}$
   if $\boxed{\Delta^{(i-1)}} = 0$ or $i \le 2 \cdot \delta_{i-1}$
     $\delta_i = \delta_{i-1},\ \boxed{C^{(i)}}(x) = \boxed{C^{(i-1)}}(x) \cdot x,\ \boxed{\Delta_C} = \boxed{\Delta_C},\ \boxed{\gamma^{(i)}}(x) = \boxed{\gamma^{(i-1)}}(x) \cdot x$
```

```
else
    $\delta_i = i - \delta_{i-1}$, $\boxed{C^{(i)}}(x) = \boxed{\Lambda^{(i-1)}}(x)$, $\boxed{\Delta_C} = \boxed{\Delta_{i-1}}$, $\boxed{\gamma^{(i)}}(x) = \boxed{\Omega^{(i-1)}}(x)$
```

The j = (δ_i +1) sub-iteration can be combined into the j=0 of the next i-th iteration. Alternatively, the shadow mark above can be deleted and calculated by (2.47) after the 2t*(δ_i +1+1) iterations for computing Λ_j's finished.

$$\Omega^{(i)} = \sum_{k=0}^{i} \Lambda_k^{(2t)} S_{i-k} \quad \textit{for} \;\; i = 0\sim(\delta_i\text{-}1),\ k = (0\sim i) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \; (2.47)$$

where $\Omega^{(i)}$ is the i-th coefficient of $\Omega(x)$. Recently, a decomposed architecture which only use three finite field multipliers has been proposed to reduce the hardware complexity significantly in [26~28], and this architecture is based on the iBMA.

Once the coefficient values of $\Lambda(x)$ and $\Omega(x)$ has been calculated, it is possible to find the error-location numbers and error magnitudes by Chien's Search algorithm and Forney's algorithm respectively.

2.3.3 The Chien's Search

We can determine the roots of $\Lambda(x)$ in GF(2^m) by substituting the elements of GF(2^m) into $\Lambda(x)$ cyclically. If $\Lambda(\alpha^i) = 0$, then $\exists$ L s.t. $1 + \beta_L \alpha^i = 0$, i.e. $\beta_L = \alpha^{-i} = \alpha^{(2^m-1)-i}$ is an error-location number.

Since the first symbol of the codeword corresponds to the x^{n-1} term, the search begins with the value

$\alpha^{-(n-1)} = \alpha^1$ *for n =* 2^m *- 1*, (io = 0)

or $\alpha^{-(n-1)} = \alpha^{io} \cdot \alpha^1$ *for n =* 2^m *- 1*- io

where io means index offset, then $\alpha^{-(n-2)} = \alpha^{io}\alpha^2 = \alpha^{io+2}$, and continues to $\alpha^0 = \alpha^{2^m-1}$, which corresponds to the last symbol of the code word.

The roots of $\Lambda(\mathrm{x})$, and hence the numbers $\beta_1 \sim \beta_v$ of error-location, are found by trial and error, known as the Chien's search [29], in which all the possible values of the roots (the field elements α^i, io+1 $\leq i \leq$ 2^m -*1*) are substituted into (2.16) and the results are evaluated.

2.3.4 The Forney's algorithm

Once the error-location numbers $\beta_1 \sim \beta_v$ have been found by Chien's Search, the next step is to find the corresponding error magnitudes $\varepsilon_1 \sim \varepsilon_v$. Recall the PGZ algorithm in Section 2.3.2.1, when the error locations $\beta_1 \sim \beta_v$, are substituted into (2.23), the first v equations can be solved by matrix inversion to produce the error values $\varepsilon_1 \sim \varepsilon_v$.

There is an alternative means, according to Forney's algorithm [27], the error magnitude of location β_L can be calculated as follow：

$$\varepsilon_L = \frac{\beta_L^{-(b-1)}\Omega(\beta_L^{-1})}{\Lambda'(\beta_L^{-1})} = \frac{\beta_L^{-b}\Omega(\beta_L^{-1})}{\Lambda_{odd}(\beta_L^{-1})} \quad \cdots\cdots (2.48)$$

How the equation above is derived from? Let us go back to (2.32) and β_L is one of the error-location numbers, then (2.32) can be rewritten as

$$\Omega(\mathrm{x}) = \sum_{L=1}^{v} \varepsilon_L \prod_{k=1,k\neq L}^{v} (1+\beta_k\beta_L^{-1}) \quad \cdots\cdots (2.49)$$

and the derivative of error-location polynomial $\Lambda(\mathrm{x})$ shown in (2.16) is

$$\Lambda'(\beta_L^{-1}) = \frac{d}{dx}\prod_{k=1}^{v}(1+\beta_k \mathrm{x}) = \sum_{i=1}^{v}\beta_i \prod_{k=1,k\neq i}^{v}(1+\beta_k\beta_L^{-1}) = \beta_L \cdot \prod_{k=1,k\neq L}^{v}(1+\beta_k\beta_L^{-1}) \ldots (2.50)$$

Moreover, the formal derivative polynomial can be described as

$$\Lambda'(\mathrm{x}) = \Lambda_1 + 2\Lambda_2\mathrm{x} + 3\Lambda_3\mathrm{x}^2 + \ldots + v\Lambda_v\mathrm{x}^{v-1} = \Lambda_1 + 3\Lambda_3\mathrm{x}^2 + \ldots = \Lambda_{odd}/\,\mathrm{x} \ldots (2.51)$$

i.e.

$$\Lambda'(\beta_L^{-1}) = \Lambda_{odd}/\beta_L^{-1} \quad \text{......} \quad (2.52)$$

From (2.49) to (2.52), the error magnitude ε_L at the position β_L can be obtained by

$$\varepsilon_L = \frac{\beta_L \cdot \Omega(\beta_L^{-1})}{\Lambda'(\beta_L^{-1})} = \frac{\Omega(\beta_L^{-1})}{\Lambda_{odd}(\beta_L^{-1})} \quad \text{......} \quad (2.53)$$

If b=0 is substituting into (2.48), the (2.48) will be equivalent to (2.53). There is a little convenient for computing the error magnitude ε_L in that $\Omega(\beta_L^{-1})$ is divided by the sum of odd terms of $\Lambda(x)$ directly without multiplying β_L^{-1} anymore. Therefore, this is why we let b=0 from stem to stern.

The alternative form of error-value evaluator is slightly different and defined as

$$Z(x) = \Lambda(x) + x \cdot \Omega(x) \quad \text{......} \quad (2.54)$$

and hence,

$$\varepsilon_L = \frac{Z(\beta_L^{-1})}{\prod_{k=1, k \neq L}^{v} (1+\beta_k \beta_L^{-1})} \quad \text{......} \quad (2.55)$$

This description for evaluating ε_L was derived by Berlekamp.

Chapter 3

Schematic-view of Proposed RS CODEC Architecture

The architecture of RS codec is separated into encoder and decoder. All of them are based on finite field arithmetic. As mentioned in Chapter 2, the RS decoder consists of syndrome calculator (Sec. 2.3.1), key-equation solver (Sec. 2.3.2), Chien-search block (Sec. 2.3.3) and error-value evaluator (Sec. 2.3.4).

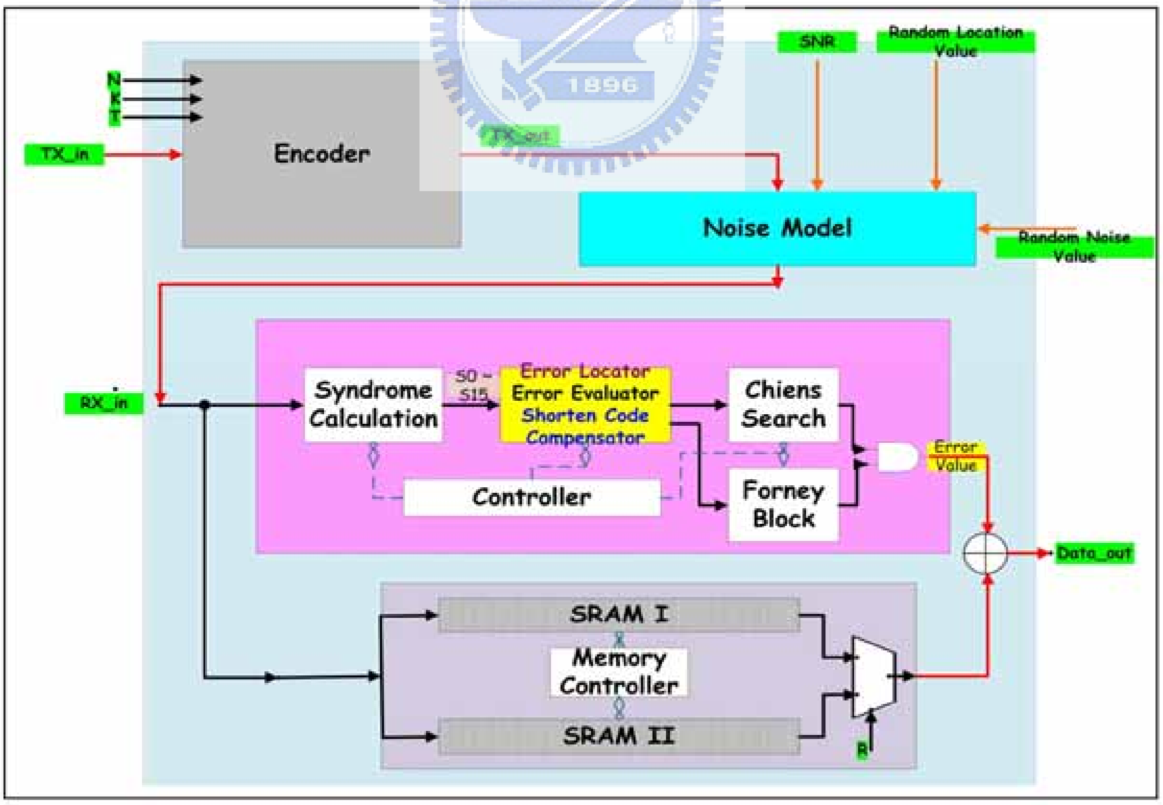


Figure 3.1：System overview of RS codec design

After long deliberation, we decided to adopt the three stage pipeline strategy and those will be described later. As shown in Figure 3.1, the first block, the syndrome calculator computes the syndrome vector and feed into the key equation solver (KES) which is the second block of RS decoder.

In the third block, the error locations are calculated by Chien's search and the error values are evaluated simultaneously according to Forney's algorithm. Meanwhile, the FIFO buffer used to keeps the received codeword is adding by the corresponding error magnitude and the output data are corrected in successive clock period.

For the shorten code, a common compensator employed to adjust the index offset which has been described in section 2.3.3 is combined in the KES block and shown in section 3.6 schematically.

Additionally, there is a block between encoder and decoder, called noise model, is designed for random patterns (or symbols) simulation and will be introduced in the next chapter. In this chapter, we focus on the entire architecture of multi-mode RS encoder and decoder design.

3.1 Pipeline Topology

In general, the pipeline topology of RS decoder is divided into two stages and three stages.

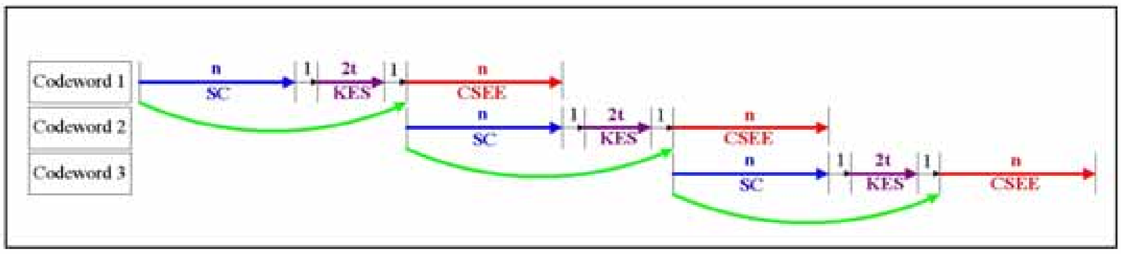


Figure 3.2：The two-stage pipeline

The two-stage pipeline is often used in parallel structure [19], thus in Figure 3.2, the second block is performed in the first stage after the first block finished the calculating of the syndrome values, and just occupies 2t clock cycles.

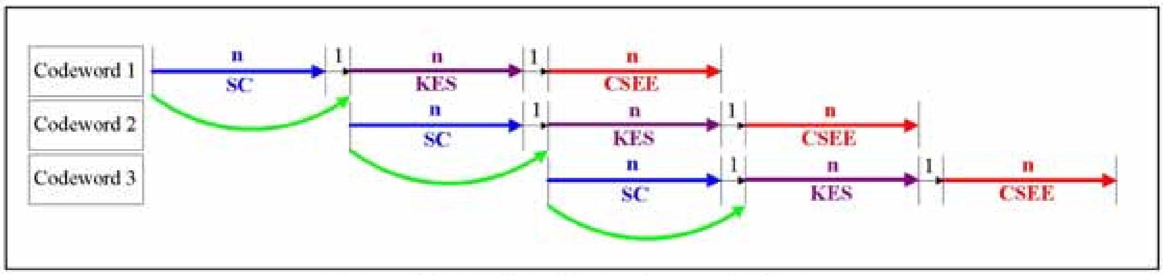


Figure 3.3：The three-stage pipeline

Figure 3.3 shows the three-stage pipeline, the second block is performed in the second stage and is often using the serial structure to reduce the hardware complexity.

For the mass data transmission, the strategy of two-stage pipeline costs more time to wait the second block finish. For example, for 10K bytes (40 code-words) transmission and (n=255, t=8), the two-stage pipeline needs (255+1+2*8+1) *40+255=11175 clock cycles but the three-stage one only has (255+1)*40+ 255+1+255=10751 cycles. In other words, the decoding speed of three-stage pipeline is fast than the two-stage one in the ratio 4%. Similarly, for 25K bytes transmission, the ratio is 5.5% fast, and that for higher data transmission with higher ratio. Deservedly, the three-stage pipeline is adopted for our high efficient design.

3.2 The t-Decoder

For multi-mode applications, we must to dominate sixteen cells of encoder and those of SC block taking effect or not by determining if a codeword has been transmitted n symbols and which correction capability "t" is required.

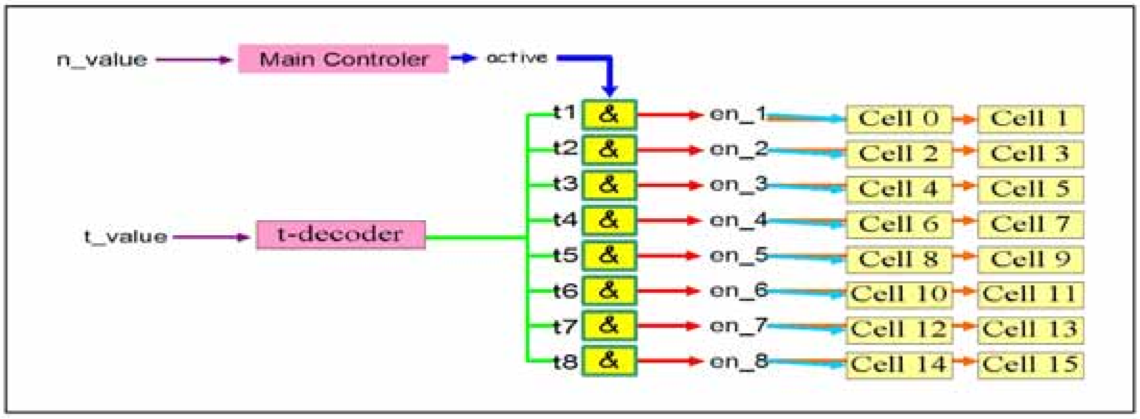


Figure 3.4：The connection of 16 cells in the encoder or SC block

As Figure 3.4 shows, the signal en_1 is ANDing "active" by "t1" and is used for dominating the cell 0 as well as the cell 1, and these extend to en_2 ,…, en_8 for cell 2~cell 15. The waveform of "active" is true in the n symbols input period and the t1~t8 is outputted from the "t-decoder". When t=8, the output of "t-decoder" is setting to 11111111, and that in {t, output} = {7, 01111111}, {6, 00111111}, …… , {1, 00000001}. In this manner, there are eight comparators and one 8-to-1 multiplexer required. However, a design only occupies 1/5 times the size above is proposed as below.

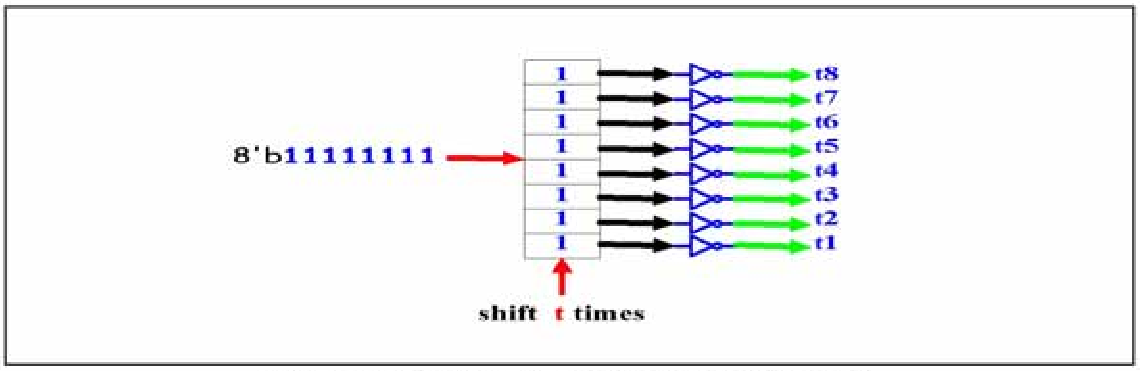


Figure 3.5：The simpler "t-decoder" design.

As Figure 3.5 shows, there are just one shift register and eight 1-bit NOT-gates in the "t-decoder" design. Each bit of the output value is connecting to t1~t8 paralleled to dominate the 16 cells in the encoder or SC block.

3.3 Encoder Architecture

Recall in Chapter 2, the *n* codeword symbols are generated form the *k* information symbols to utilize the generator polynomial $g(x)$ described as (2.4) or (2.5). As shown in Figure 2.2, a codeword can be obtained in a systematic form by adding $2t$ parity-check symbols. In general, there are two different architectures in encoding as described in Method 2 and Method 3 of Section 2.2.2.

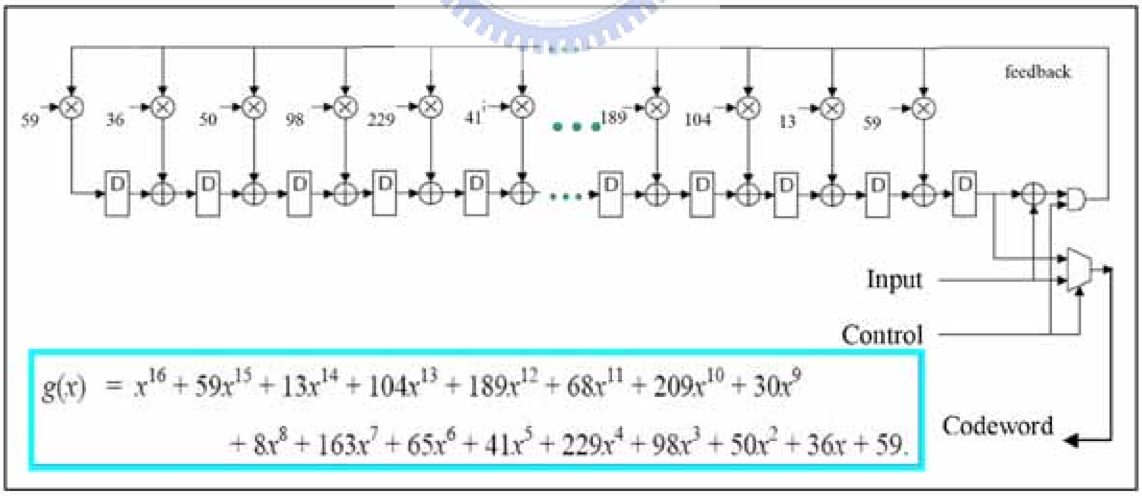

feedback
59
36
50
98
229
41
189
104
13
59
D
Input
Control
Codeword
$g(x) = x^{16} + 59x^{15} + 13x^{14} + 104x^{13} + 189x^{12} + 68x^{11} + 209x^{10} + 30x^{9}$
$+ 8x^{8} + 163x^{7} + 65x^{6} + 41x^{5} + 229x^{4} + 98x^{3} + 50x^{2} + 36x + 59.$

Figure 3.6：The structure of RS encoder formed by Method 2.

Figure 3.6 shows the pipelined calculation (division) performed by using Method 2. The tap-coefficient of the 2t stage LFSR is corresponding to the coefficient of g(x) formed in (2.5) and rewritten in the bottom of Figure 3.6 by setting b=0 and t=8. All the data paths shown above provided for 8-bit values. Over the message input period, the selector passes the input values directly to the output and the AND gate is enabled by the control signal. After *k* calculation steps have been completed (in *k* consecutive clock periods), the remainder is contained in the D-type register. The control waveform then changes so that the AND gate prevents further feedback to the multipliers and the *2t* remainder symbol values are clocked out of the registers and routed to the output by the selector.

An alternative structure, the fractional form according to Method 3, is constructed by using α^i as its tap-coefficients and where α^i is the 2t successive power of α as shown in (2.4) or (2.8). Figure 3.7 shows the α^i-based structure.

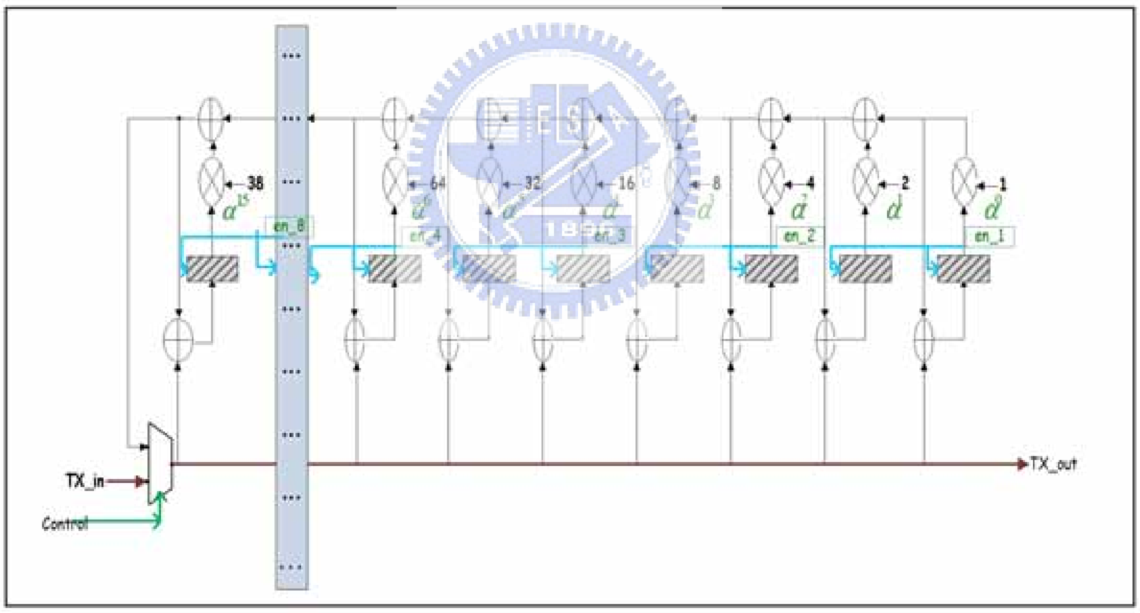


Figure 3.7： The α^i-based structure of RS encoder formed by Method 3.

Comparing Method 2 with Method 3, Method 2 is suitable for single-mode application and Method 3 is suitable for multi-mode application. For multi-mode applications, the tap-coefficients of Method 2 can not be simplified to const multipliers and additional registers is required to store different set coefficients of g(x). On the contrary, in the Method 3, we only need to enable or disable the branch (register), then the α^i-based encoder can be employed for the multi-mode application. But the critical path of the α^i-base encoder is increasing with t and is slower than those in Method 2.

Additionally, the major advantage of α^i-base architecture is that the hardware can be shared in the both encoding processes and syndrome calculation, discussed in the next section. For the timing comparison outcome, the result suggests to choose Method 2. For area comparison results, the α^i-base architecture is the better choice. For convenience, we choose the α^i-base encoding in our design in virtue of its elastic property.

3.4 Syndrome Calculator (SC)

According to the decoding algorithm mentioned in Section 2.3, we need to determine if the series of symbols contained in a data block form a valid codeword for the particular RS code chosen, by finding the syndrome values first. The SC block accepts the received symbols transmitted over a noise channel from encoder and outputted to the KES block. The syndrome values S_i can be defined as $R(\alpha^i)$. By Horner's rule, (2.13) can be re-written as：

$$S_i = (((...(R_{n-1}\alpha^i + R_{n-2}) \alpha^i + ... + R_1) \alpha^i + R_0 \qquad (3.1)$$

In the main, the process turns out to begin with multiplying the first coefficient R_{n-1} by α^i. Then each subsequent coefficient is added to the previous product and the resulting sum multiplied by α^i until finally R_0 is added. This has the advantage that the multiplication is always by the same value α^i at each stage. The calculation can be completed by using GF constant multipliers, exclusive-or gates, and registers. As Figure 3.8 shows, the SC block diagram, the pervious partial syndrome value is multiplied by α^i in each clock period and accumulates with the received symbol.

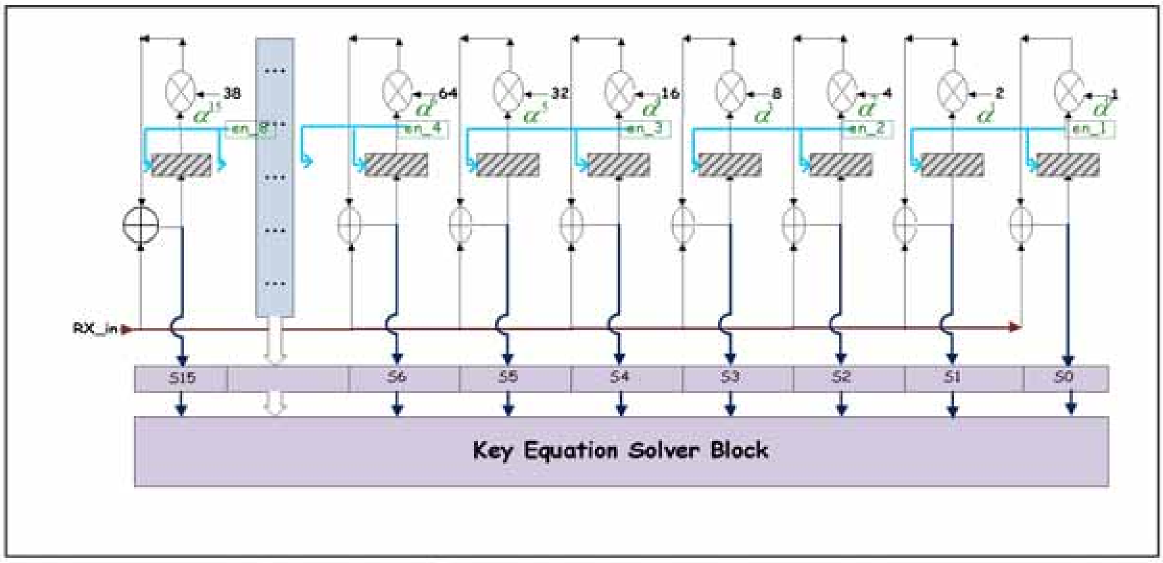


Figure 3.8： Syndrome calculator block

If you are penetrative, you can find that Figure 3.7 and 3.8 are similarly except the additional exclusive-or gates at the top of Figure 3.7. So, we can combine them by inserting multiplexers between two exclusive-or gates.

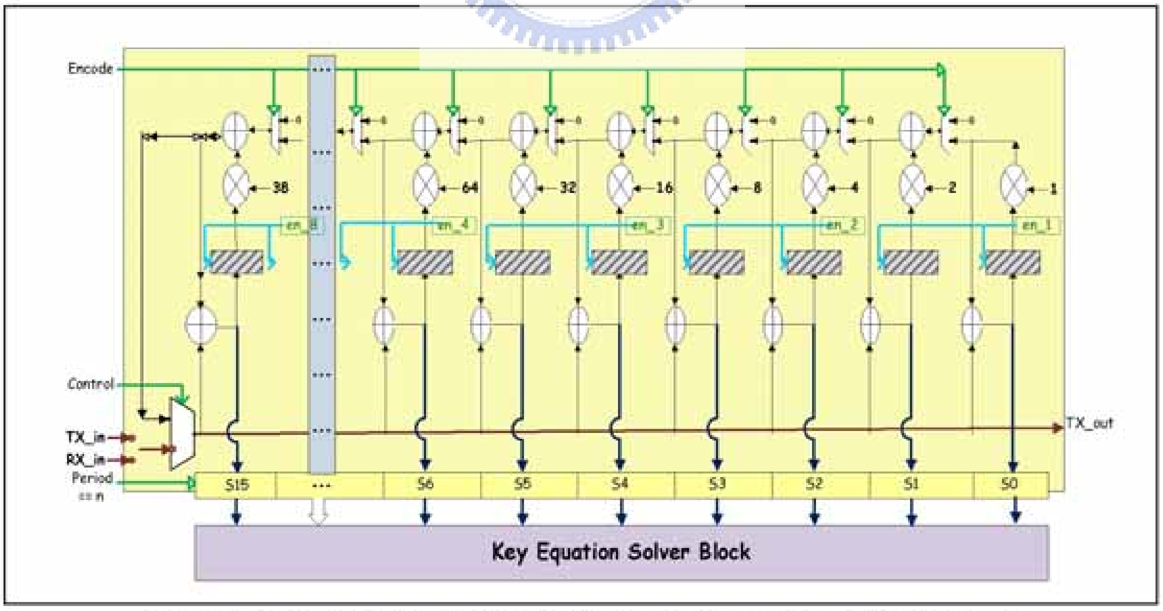


Figure 3.9： Combined Multi-Mode Encoder & SC Architecture

Thus in 3.9, the combined architecture can not only for encoder purpose but also for syndrome calculation.

The SC block makes it possible to compute the syndromes within n symbol period and the syndrome symbols, S_i (0~15), are outputted parallel to the KES block.

When the syndrome values are all equal to zero, the following decoding procedure will be terminated. If not, the error-locator polynomial as well as the error-evaluator will be calculated in the second block.

3.5 Key Equation Solver (KES)

According to the proposed SiBM algorithm mentioned in Section 2.3.2.3, the SiBM architecture will be proposed in this section. The $\Lambda(x)$ can be obtained by (2.43) and can be performed by using the arrangement of Figure 3.10 in which all data paths are 8 bits wide.

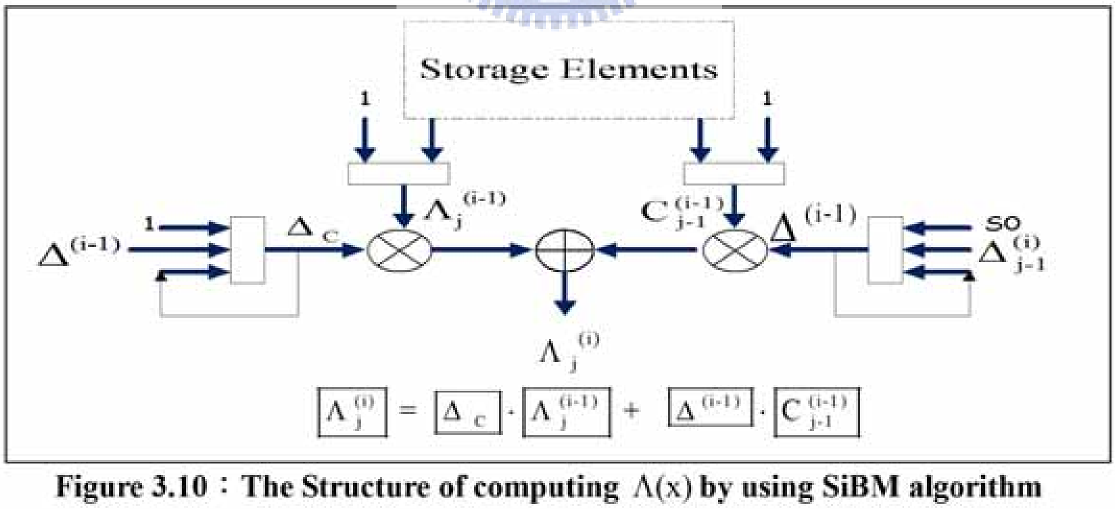
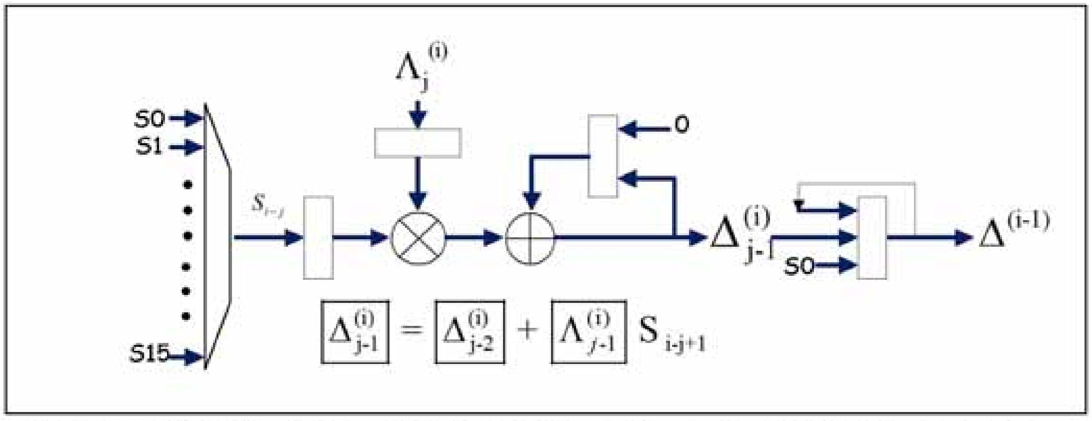


Figure 3.10：The Structure of computing $\Lambda(x)$ by using SiBM algorithm

The concept is too simple to interpret anymore for us. By this arrangement, the critical path delay is just $T_{ff} + T_{mult} + T_{add}$ at most if the output terminal $\Lambda_j^{(i)}$ is followed by a FIFO register.

It is also, according to (2.45), the circuit for computing $\Delta_{j-1}^{(i)}$ at step j of iteration i is shown in Figure 3.11.



$\Lambda_j^{(i)}$
S0
S1
S15
S_{i-j}
0
$\Delta_{j-1}^{(i)}$
S0
$\Delta^{(i-1)}$
$\boxed{\Delta_{j-1}^{(i)}} = \boxed{\Delta_{j-2}^{(i)}} + \boxed{\Lambda_{j-1}^{(i)}} S_{i-j+1}$

Figure 3.11 : The decomposed architecture for computing i-th discrepancy

For j =1~δ_i of each iteration, $\Lambda_{j-1}^{(i)}$ is fed and multiplied by a corresponding syndrome value S_{i-j+1} which is outputted from SC block and selected by a multiplexer. Then, the product is added by the pervious partial discrepancy $\Delta_{j-2}^{(i)}$ which is stored in the register and initialize to zero when j=1. Finally, the discrepancy of the i-th iteration, i.e. $\Delta^{(i)}$, is calculated and summed up in the step j=δ_i+1 which is combined in the first step of next iteration. In other word, the discrepancy $\Delta^{(i)}$ is finished in the step j=0 and outputted while j=1 of the (i+1)th iteration.

In general, the serial architecture of iBMA[12, 26~28] needs 2t iterations and each iteration needs (t+1) steps to compute $\Lambda(x)$, $\Omega(x)$ and $\Delta^{(i)}$. In practice, they can be calculated simultaneously and five **finite field multipliers** (**FFM**) are needed. That is, totally 2t*(t+1) steps (cycles) is required if five FFMs is used. But the FFM is more complex and cost a lot of area. Alternatively, we can use (2.47) to compute $\Omega(x)$ as soon as $\Lambda(x)$ have been calculated after 2t iterations. Then, we can not only save much power consumption on iteratively computing the medium solution of $\Omega(x)$ but also reduce two FFMs.

Suppose we have just finished the computation of $\Lambda(x)$, that is, $\Lambda_{\delta_{2t}}^{(2t)}$ has just been calculated in the finial step of the 2t iteration. For calculating $\Omega(x)$ by (2.47) and sharing the hardware that is shown in Figure 3.10 and 3.11, we must let the inputs of the FFM in Figure 3.11 corresponding to $\Lambda_k^{(2t)}$ and S_{i-k} of (2.47). Fortunately, it is not difficult to achieve. For the corresponding $\Lambda_k^{(2t)}$, we let $\Delta_C=1$ and $C_{j-1}^{(i-1)}=0$ in Figure 3.10 during the computation of $\Omega(x)$. Then, the value of the position $\Lambda_j^{(i-1)}$ is equal to that of the position $\Lambda_j^{(i)}$, that is, $\Lambda_0^{2t},\Lambda_1^{2t},\cdots,\Lambda_i^{2t}$ is fed into Figure 3.11 in proper order and answer to $\Lambda_k^{(2t)}$ of (2.47). Coincidentally, the position S_{i-j} of Figure 3.11 also fit in with S_{i-k} of (2.47) and $\Omega(x)$ can be calculated after additional $\delta_{2t}(\delta_{2t}+1)/2$ cycles.

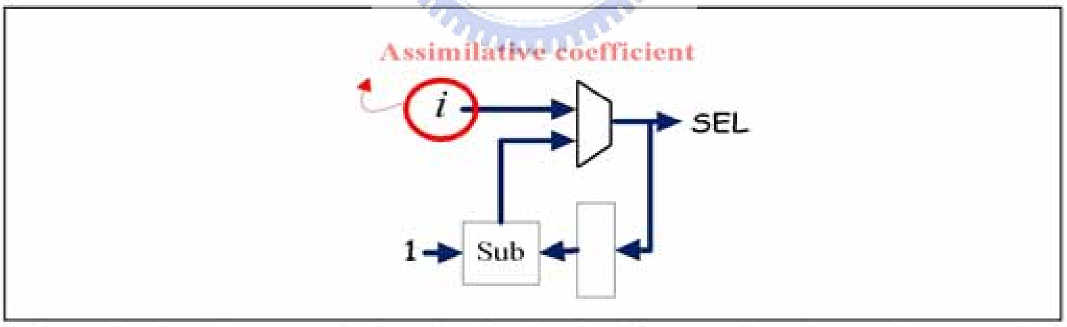


Figure 3.12：The control circuit of selection line to select the syndrome values

Figure 3.12 shows the control circuit of the selection line in which the multiplexer of Figure 3.11 is using to select the syndrome values. The initial coefficient of syndrome is exactly the iteration number " i " and is decreasing by one in each successive clock period, even if the circuit is performed the computation of $\Lambda(x)$ or $\Omega(x)$.

In fact, this coincidence is artificiality by setting the computation of Λ(x) from i=1 to i=2t, the power of the first consecutive root in g(x) equal to zero (b=0), and some retiming skills. The artificial modification is an example of my so-called the "**assimilative coefficient**" skill.

If n is small enough than 2t*(t+1), the input data must stall for waiting the KES block finish. Actually, only δ_i+1 steps is valid at each iteration (recall that δ_i is the degree of i iteration), the rest of t-δ_i steps resulted in zero. It stands to reason that we can save many steps if we skip them, but how we can evaluate the degree before the iteration ending and determine the ending step without increasing the critical path delay is very difficult.

Recall the summation in Section 2.3.2.3, if $\boxed{\Delta^{(i-1)}} = 0$ or $i \le 2 \cdot \delta_{i-1}$, the degree of the i-th iteration δ_i is equal to pervious degree δ_{i-1}, else $\delta_i = i - \delta_{i-1}$. According to the schematic of Figure 3.11, the (i-1)-th discrepancy is finished in the first cycle of i-th iteration and updated in the second cycle (j=1). Thus for keeping the critical path is still $T_{ff} + T_{mult} + T_{add}$ at most, the decision (if $\boxed{\Delta^{(i-1)}} = 0$ or $i \le 2 \cdot \delta_{i-1}$) must be performed in the j=1.

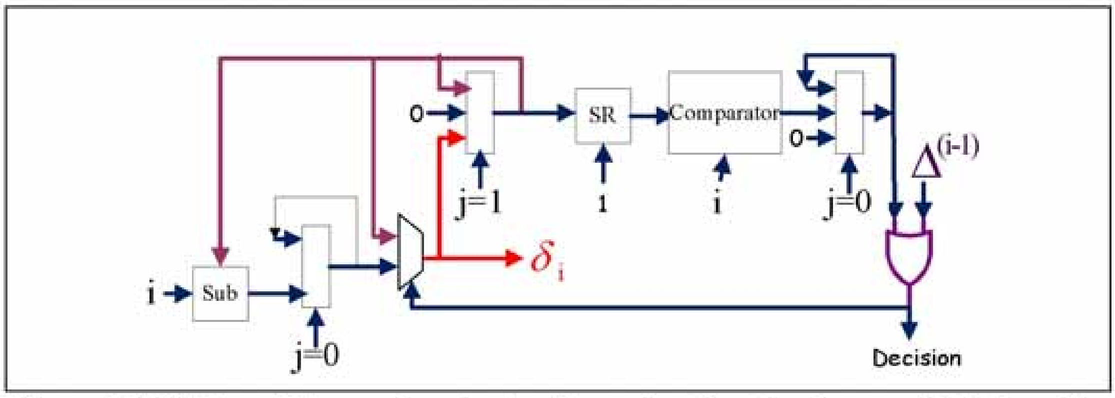
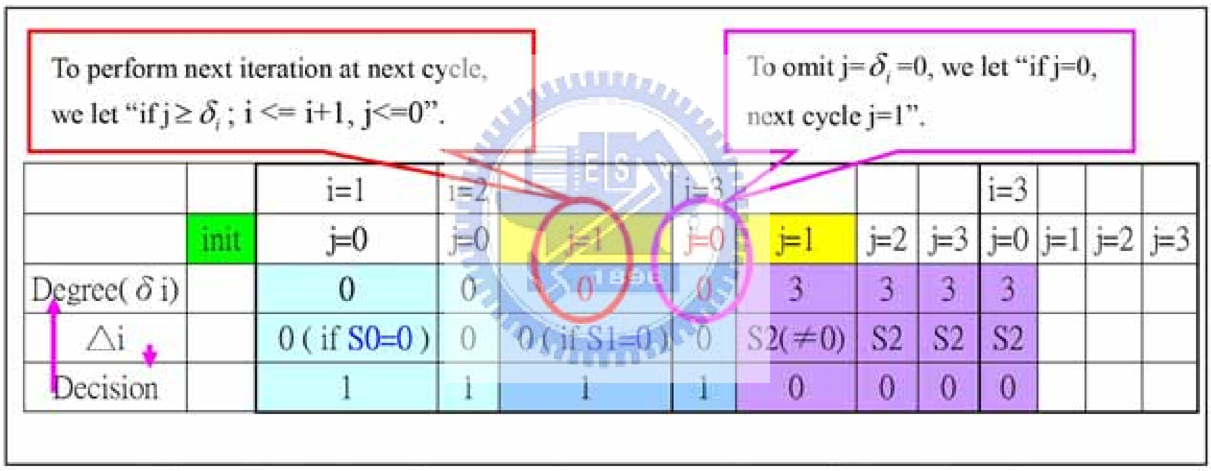


Figure 3.13：The arbiter and evaluator for evaluating the degree of i-th iteration

As Figure 3.13 shows, the waveform of "Decision" has transition only when j=1.

Then, the degree δ_i can be evaluated at the same cycle and three registers are inserted to keep the critical path delay not exceeding $T_{ff} + T_{mult} + T_{add}$.

In general, when j=δ_i, the finial step of the i-th iteration, we must set i= i+1 and reset j=0 in the next cycle by determining if j is equal to δ_i. But the degree of i-th iteration δ_i is evaluated in the j=1 in our design, there are two exception may cause some unexpected results. One is when j=0, δ_i is still equal to zero, but δ_i is no longer equal to zero in the next cycle, the iteration will be terminated and gone to the next iteration. The other is when {j, δ_i}={0,0} is ignored, then {j, δ_i}={1,0} is happened, the iteration ought to stop but never stop in the following cycle.


To perform next iteration at next cycle, we let "if j ≥ δ_i ; i <= i+1, j<=0".
To omit j=δ_i=0, we let "if j=0, next cycle j=1".

		i=1	i=2		i=3				i=3			
	init	j=0	j=0	j=1	j=0	j=1	j=2	j=3	j=0	j=1	j=2	j=3
Degree(δ i)		0	0	0	0	3	3	3	3			
$\triangle$i		0 (if S0=0)	0	0 (if S1=0)	0	S2($\neq$0)	S2	S2	S2			
Decision		1	1	1	1	0	0	0	0			

Figure 3.14：Iteration control to prevent the special case hazard

As Figure 3.14 shows, we have to control the flow to go to next iteration at next cycle when j=δ_i *or* {j, δ_i}={1,0}, but ignore {j, δ_i}={0,0} even though the j=δ_i holds.

Figure 3.15 shows the storage element for storing and providing all data required in Figure 3.10 and 3.11. By **retiming the decision** (Figure 3.13) and **controlling the iteration** (Figure 3.14) mentioned above, the critical path bottleneck in the iterative computation of discrepancies followed by evaluating the degree and updating of the error-locator correction polynomial C(x) can be fetched through.

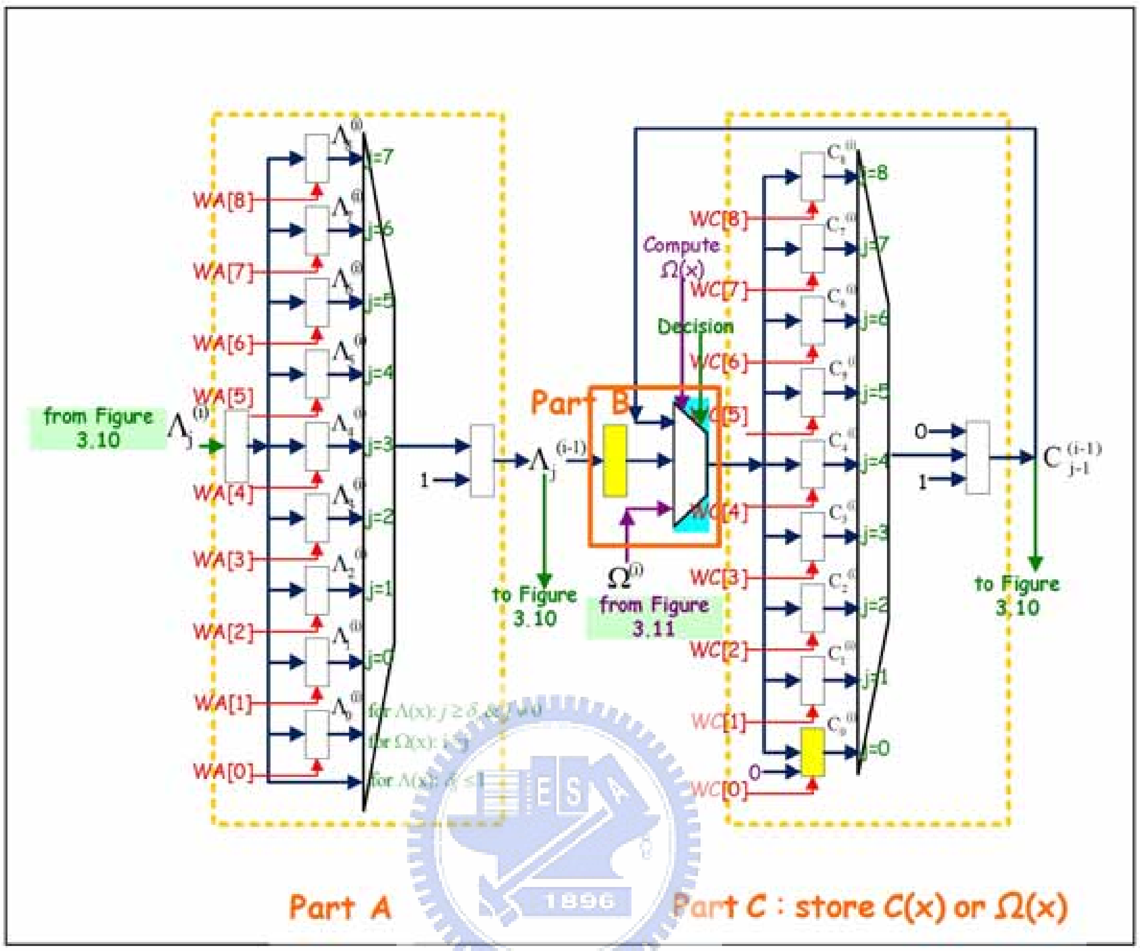


Figure 3.15： **Dataflow controller and storage element design**

In the i-th iteration, the computation result of Figure 3.10, the coefficients $\Lambda_j^{(i)}$'s are stored in the Part A and outputted in the next iteration as the $\Lambda_j^{(i-1)}$'s in turn. For computing $\Lambda(x)$, the Part C is storing the correction polynomial. Otherwise, for calculating $\Omega(x)$, the Part C is storing the $\Omega^{(i)}$'s to save 8 registers. Deservedly, the Part B is used to do this selection and to decide which one is the proper correction polynomial coefficient by the waveform of "Decision".

Note that the additional register in the Part B and the register in the Part C with the same yellow mark indicate that they are both used for matching the "Decision" variation and preventing the special-case data hazard of serial architecture. In the conventional design, the "Decision" waveform has original transition in the j=0 and the additional register does not exist, the special case data hazard always arises in that the "Decision" waveform of previous iteration is true, the degree of correction polynomial $C^{(i-1)}(x)$ is δ_{i-1}; if the "Decision" waveform is still true in this iteration, $\delta_i=\delta_{i-1}$ and $C^{(i)}(x)=C^{(i-1)}(x)\cdot x$, the degree of $C^{(i)}(x)$ must be δ_i+1. Then, when j=0 in the next iteration, the $C^{(i-1)}_{\delta_{i-1}}$ is just outputted from Part C and feedback to the multiplexer of Part B, the serial architecture have to store $C^{(i-1)}_{\delta_{i-1}}$ in the position $C^{(i)}_{\delta_i+1}$ of Part C. At the same time, if the "Decision" waveform is false, the multiplexer select the value $\Lambda_0^{(i-1)}$ and store in the position $C^{(i)}_0$, the data of $C^{(i)}_{\delta_i+1}$ is empty. Even if the "Decision" waveform is true, the data $C^{(i-1)}_{\delta_{i-1}}$ is storing in the position $C^{(i)}_0$, not the expecting position $C^{(i)}_{\delta_i+1}$. In other words, the data hazard is happened when the degree of $C^{(i)}(x)$ is exceeding the degree δ_i.

Fortunately, if we **control the dataflow** and do "Decision" variation, the special case data hazard will be eliminated. That is, the "Decision" waveform has transition in the j=1, when j=0 and the "Decision" waveform is "true", the multiplexer selecting $C^{(i-1)}_{\delta_{i-1}}$ and storing in the position $C^{(i)}_{\delta_i+1}$ in the next cycle. If the waveform change to false in the j=1, the multiplexer selecting the value $\Lambda_0^{(i-1)}$ and storing in the position $C^{(i)}_0$. Oppositely, if the waveform keep true in the j=1, we let the value in the register of the Part C with yellow mark be equal to zero and $C^{(i-1)}_0$ be stored in the position $C^{(i)}_1$ for answering to $C^{(i)}(x)=C^{(i-1)}(x)\cdot x$.

To store the data in the proper position, the purpose-built address line is used in the Part A and Part C as drawn in red. By these address lines, the storage element can not only insert the data stream more flexibility but also save many comparators and multiplexers. Therefore, the shorten code compensator can be combined in the KES block easily by this arrangement.

3.6 Shorten Code Compensator

Recall in Section 2.3.3, for the shorten code applications, the codeword length "n" is not always equal to 2^m-1. For skipping the non-valid cycles, the search must begins with the position X^{n-1}, that is, we must check if $\Lambda(\alpha^{-(n-1)})$ is equal to zero or not initially. Suppose n = 2^m-1-io, where "io" is called the *index offset* and then $\alpha^{-(n-1)}$ = $\alpha^{io}\cdot\alpha^{1}$. In other words, for checking $\Lambda(x)$ is equal to zero or not, all the coefficients in $\Lambda(x)$ must multiplied by a compensative value $(\alpha^{io})^P$ firstly, then multiplied by $(\alpha^{1})^P$ in each successive clock period, and these procedures extend to the coefficients of $\Omega(x)$ for finding the error magnitude. Note that P is the power corresponding to the coefficient.

It turns out to be 15 FFMs needed for multiplying 15 coefficients at one cycle, when the compensator is locating in the CSEE block. Alternatively, we can use 2 FFMs, one for multiplying $(\alpha^{io})^P$ with corresponding coefficient and the other for producing the proper value $(\alpha^{io})^P$ by multiplying α^{io} each cycle, there will be additional 15 cycles required.

However, there is a better choice, combining the compensator into KES block and taking effect as soon as the coefficient is valid. As Figure 3.16 shows, the common compensator with dim yellow mark for multiplying all coefficients is merged with overall KES block.

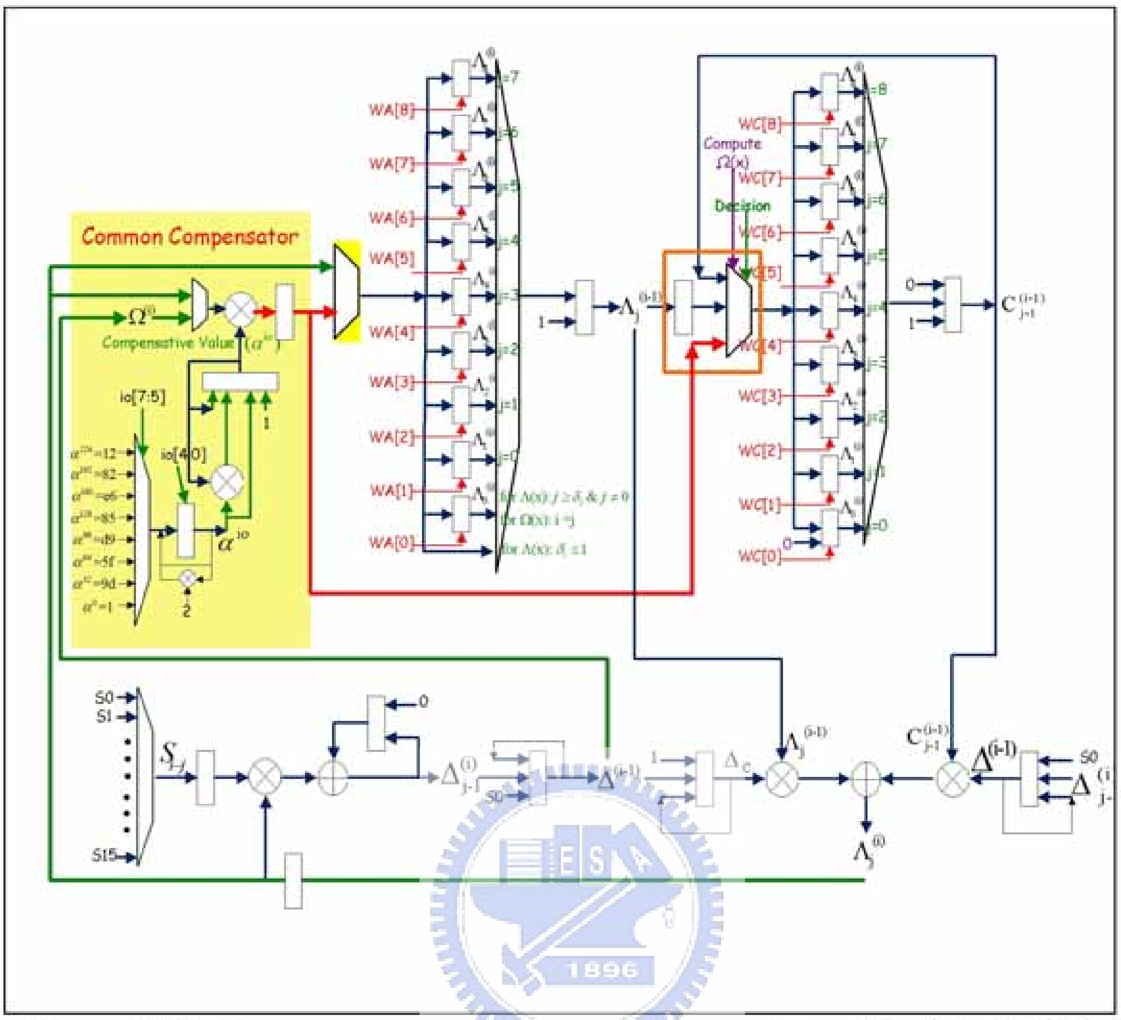


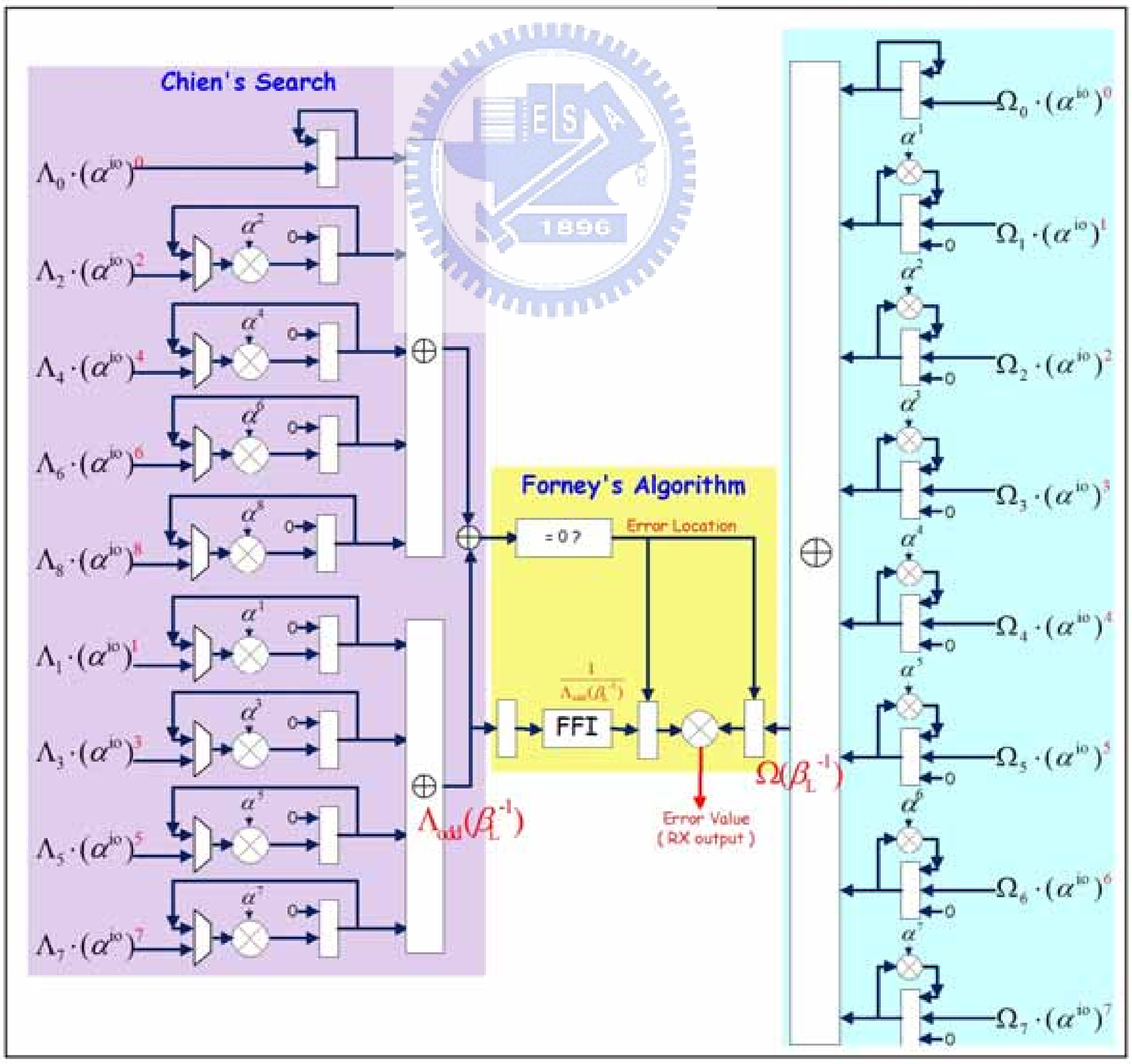
Figure 3.16：The common compensator and its connection with whole KES block

Initially, the compensative value is set to 1 for multiplying Λ_0 or Ω_0. Then, $(\alpha^{\mathrm{io}})^1$ is set to multiply Λ_1 or Ω_1, and this property extend to $(\alpha^{\mathrm{io}})^2 \ldots (\alpha^{\mathrm{io}})^{\delta_{2t}}$ in turn. Note that since Ω_i is not outputted continuously, we must keep the compensative value not to change until next Ω_i is valid. Where α^{io} is calculated in the beginning and before the KES block taking effect.

By this arrangement, we can not only save more than 4000 gates but also reduce many processing cycles and the critical path delay is still $T_{ff} + T_{mult} + T_{add}$ at most. It is also, the Chien's search and the Forney's block can be performed as normal condition in that we have adjusted the starting point from X^{2^m-1} to X^{n-1}.

3.7 Chien's Search and Error Evaluator(CSEE) Block

According to Section 2.3.3, to try the first position X^{n-1} in the codeword, we need to substitute $x=\alpha^{-(n-1)}=\alpha^{io}\cdot\alpha^{1}$ into the error locator polynomial $\Lambda(x)$. Recall in Section 3.6, the coefficients of $\Lambda(x)$ and $\Omega(x)$ are multiplied by appropriate power of α^{io} and stored in the storage element. Then, as shown in the purple mark of Figure 3.17, the value of each term is calculated by loading $\Lambda_i\cdot(\alpha^{io})^i$ and multiplying it by the appropriate α^i.



Chien's Search
$\Lambda_0\cdot(\alpha^{io})^0$
$\Lambda_2\cdot(\alpha^{io})^2$
$\Lambda_4\cdot(\alpha^{io})^4$
$\Lambda_6\cdot(\alpha^{io})^6$
$\Lambda_8\cdot(\alpha^{io})^8$
$\Lambda_1\cdot(\alpha^{io})^1$
$\Lambda_3\cdot(\alpha^{io})^3$
$\Lambda_5\cdot(\alpha^{io})^5$
$\Lambda_7\cdot(\alpha^{io})^7$
$\Lambda_{odd}(\beta_L^{-1})$
Forney's Algorithm
=0?
Error Location
$\frac{1}{\Lambda_{odd}(\beta_L^{-1})}$
FFI
Error Value (RX output)
$\Omega(\beta_L^{-1})$
$\Omega_0\cdot(\alpha^{io})^0$
$\Omega_1\cdot(\alpha^{io})^1$
$\Omega_2\cdot(\alpha^{io})^2$
$\Omega_3\cdot(\alpha^{io})^3$
$\Omega_4\cdot(\alpha^{io})^4$
$\Omega_5\cdot(\alpha^{io})^5$
$\Omega_6\cdot(\alpha^{io})^6$
$\Omega_7\cdot(\alpha^{io})^7$

Figure 3.17：The CSEE Block

Adding the value of the individual terms together produces the value of $\Lambda(\alpha^{-(n-1)})$. For subsequent position, the power of α to be substituted will advance by one for the x term and by two for the x^2 term, similar to the other terms. So, at each successive clock period, the next value of the term is produced by multiplying the previous result by the appropriate power of α and adding the values of the individual terms together produces the value of $\Lambda(x)$ for each position in turn.

Detecting zero values of the sum identifies the error positions and then using the Forney's Algorithm to calculate the corresponding error values. According to (2.53), these value is obtained by computing the quotient of two polynomials, $\Omega(x)$ and $\Lambda_{odd}(x)$, that is, multiplied $\Omega(x)$ by $\frac{1}{\Lambda_{odd}(x)}$ and hence one finite field inverter (FFI) is needed, as drawn in the yellow bottom color and where the value of $\Omega(x)$ is calculated from the cyan mark in the right hand.

For the critical path consideration, three register is inserted. When detecting the error position, the error value is calculated by $\frac{\Omega(\beta_L^{-1})}{\Lambda_{odd}(\beta_L^{-1})}$ and outputted form the decoder. Otherwise, two inputs in the multiplier are setting to zero and hence the output value is zero. Note that the value of $\Omega(\beta_L^{-1})$ is calculated after the error position β_L detected by one clock period for matching the effect of inserting three registers. That is why the hardware of computing $\Omega(x)$ is quite different to that of computing $\Lambda(x)$.

The outputted values from decoder, the error magnitudes from CSEE block, will be added with the FIFO buffer that is employed to keep the received codeword. Then, the codeword are corrected in successive clock periods.

Chapter 4

Implementation & Verification

This chapter will show the design flow and the verification step of our design. Besides, the simulation result, synthesis result, performance and comparison will be discussed.

4.1 Design Flow

As soon as the specification of our multi-mode RS code is determined, we begin to design our circuit. The design flow and verifications are illustrated in Figure 4.1.

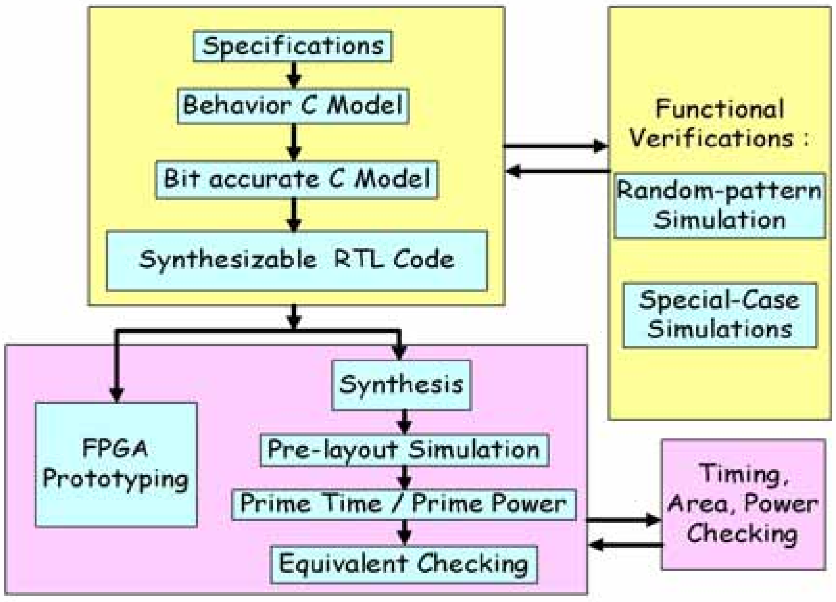
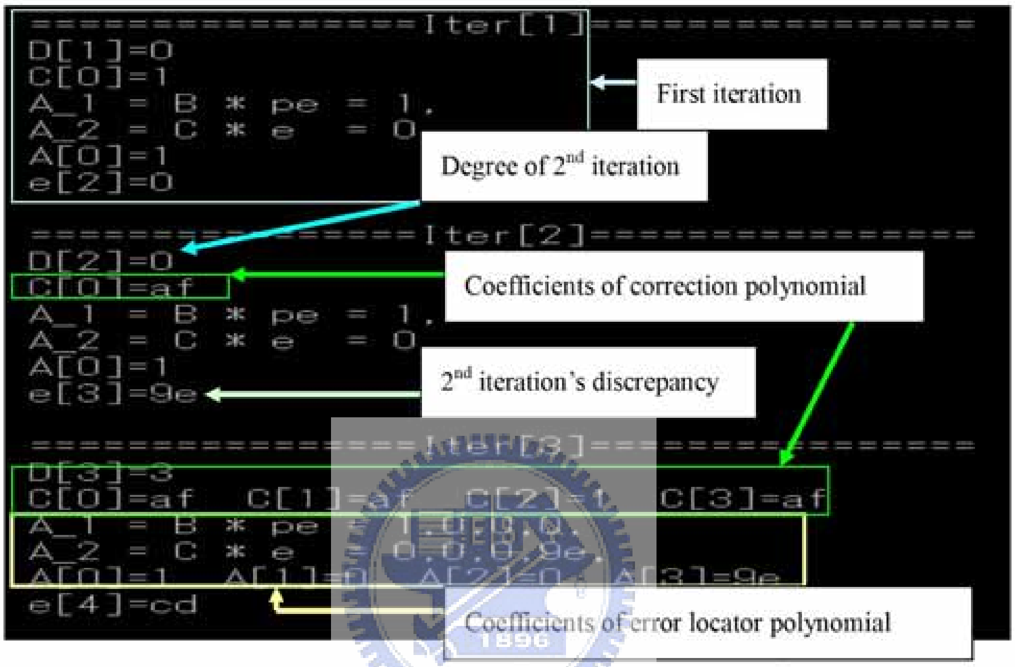


Figure 4.1：Design Flow and verifications

In the beginning, the behavior C model is used to make sure the understanding of whole encoding and decoding step. Secondly, a bit accurate C model based on the decided architecture is built and output some useful data to file as Figure 4.2 shows.



```
================Iter[1]================
D[1]=0
C[0]=1
A_1  =  B  *  pe  =  1,
A_2  =  C  *  e   =  0,
A[0]=1
e[2]=0

================Iter[2]================
D[2]=0
C[0]=af
A_1  =  B  *  pe  =  1,
A_2  =  C  *  e   =  0,
A[0]=1
e[3]=9e

================Iter[3]================
D[3]=3
C[0]=af   C[1]=af   C[2]=1   C[3]=af
A_1  =  B  *  pe  =  1,0,0,0,
A_2  =  C  *  e   =  0,0,0,9e,
A[0]=1   A[1]=0   A[2]=0   A[3]=9e
e[4]=cd
```

First iteration
Degree of 2nd iteration
Coefficients of correction polynomial
2nd iteration's discrepancy
Coefficients of error locator polynomial

Figure 4.2：Parts of output file of bit-accurate C model.

Then the synthesizable RTL code written by Verilog Hardware Description Language (HDL) is simulated and compared with the output file of bit-accurate C model. The results of simulation must ensure that they are the same as those of the C program. After all functions have been verified, the hardware is implemented by Xinlinx VirtexE xcv2000e FPGA and synthesized by UMC 0.18um cell library. Because the equivalent checking is regular, we get some valid information about timing, area and power.

4.2 Functional Verifications

As soon as the results of synthesizable RTL code are verified by comparing with the bit-accurate C model, the next step is to do functional verifications. To make sure all functionality regular, we perform the random simulation and special-case simulation.

4.2.1 Random Simulation

For random simulation, the block diagram of overall system can be connected as the arrangement of Figure 3.1. The codeword is outputted from encoder and interfered by the noise model, and then the impure information is fed into decoder as well as FIFO buffer. The noise model can be implemented as shown in Figure 4.3.

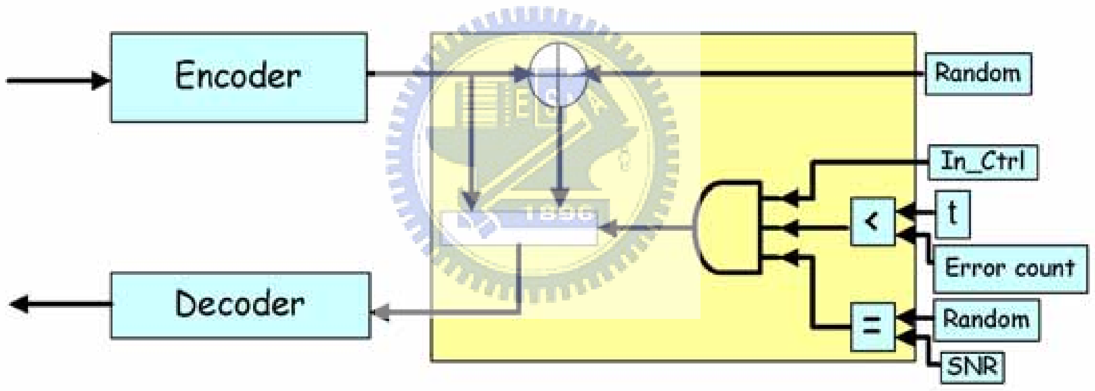


Figure 4.3：The Noise Model

Thus in Figure 4.3, the data symbol that fed into decoder is coming from the output of encoder directly or added by a random error magnitude. For example, as (n, t) = (255, 8) and the SNR is setting to 11111, if random (location) value is exactly equal to 11111 as well as the error count is less than the error capability t during the n input period, the output symbol of encoder will be added by a random value. Otherwise, the output of encoder will be fed into decoder directly. The error probability is about $256\times\frac{1}{2^5}=8$ times per codeword and the additional control signal (Error count < t) is using to make sure the error count not exceed the correction capability t. After 1 hundred-million code-words (204.8 hundred-million bits) transmission, a flag used to count the number of decode-fail is equal to zero in our design. Meaning that the functionality in (255, 239, 8) of our design is regular, and so is for other coding rate.

4.2.2 Special-Case Simulation

Because of the hardware sharing in the serial structure, some combinations of error magnitudes as well as error locations may cause unforeseen data hazard. Especially when the discrepancy of SiBM algorithm caused continuous zero, the degree of correction polynomial is increasing by one in each the iteration. Then, the valid data may appear at the same time and is selected by a multiplexer, that is, one data must be loss in the selection. For example, if $S_0=0$, the initial discrepancy is setting to $S_0=0$; then as $S_1=0$ exactly, the discrepancy of this iteration is also equal to zero, the data hazard will occur during the next iteration. So, the special-case simulation is performed as following：

Codeword 1：8 arbitrarily errors
Codeword 2：4 errors, $S_0=0$，$S_1=0$
    Ex: The continuous error values are 1, 2, 0, 0, 0, 0, 1, 2
Codeword 3：no errors
Codeword 4：8 errors, $S_0=0$
    Ex: The 8 errors with the same values：1, 1, 1, 1, 1, 1, 1, 1
Codeword 5：2 errors, $S_0=0$
    Ex: The 2 errors with the same values：0, 1, 0, 1, 0, 0, 0, 0
Codeword 6：2 arbitrarily errors
Codeword 7：1 arbitrarily errors

The special-case simulation is performed before and after synthesis, the results are the same in our design. Figure 4.4 shows one of the results in the gate-level simulation.

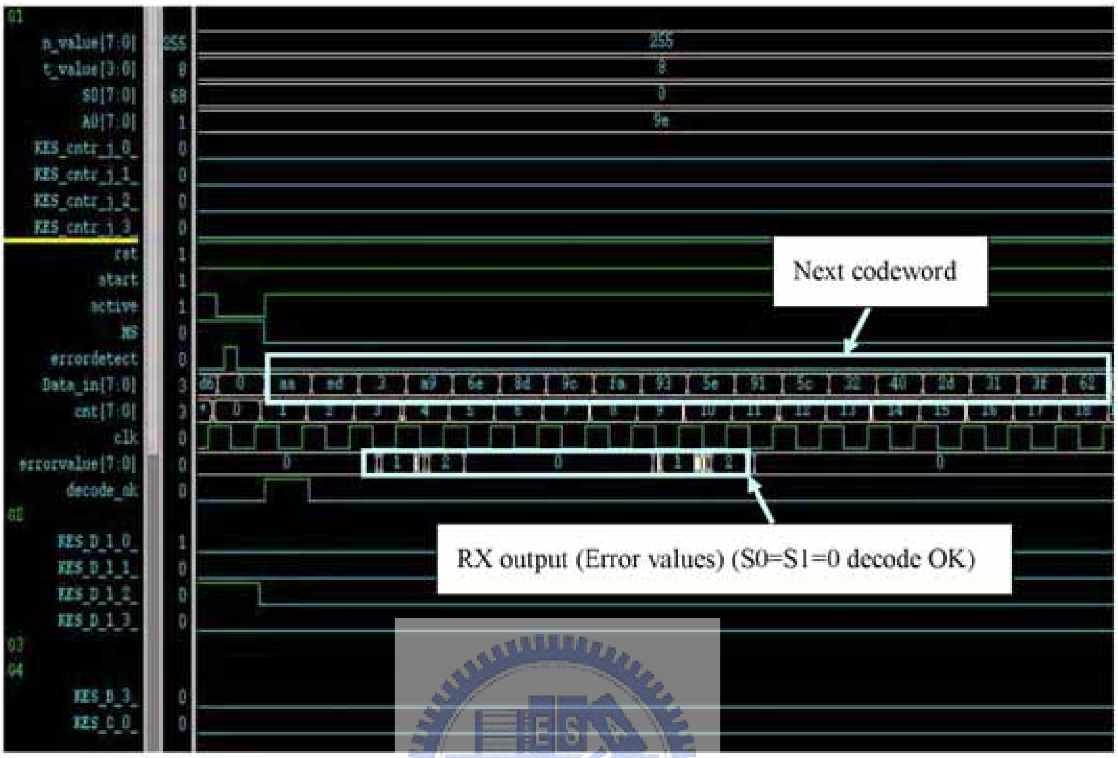


Figure 4.4：Gate-level simulation

4.3 FPGA prototyping

Test Environment：

Hardware Description Langue：verilog

Compiler and Simulation Tools：Xilinx ISE 7.1i

FPGA Chip：Xilinx VirtexE xcv2000e

FPGA Board：BG560-7

Pattern Generator：Agilent 16522A

Logic Analyzer：Agilent 16557D

The input patterns are stored in the pattern generator and sent to FPGA. Then, we check the result of logic analyzer by the waveform or dump the resulting file for checking. Also, the random simulation and special-case simulation are included. Figure 4.5 shows the connection of FPGA board, pattern generator, and logic analyzer.

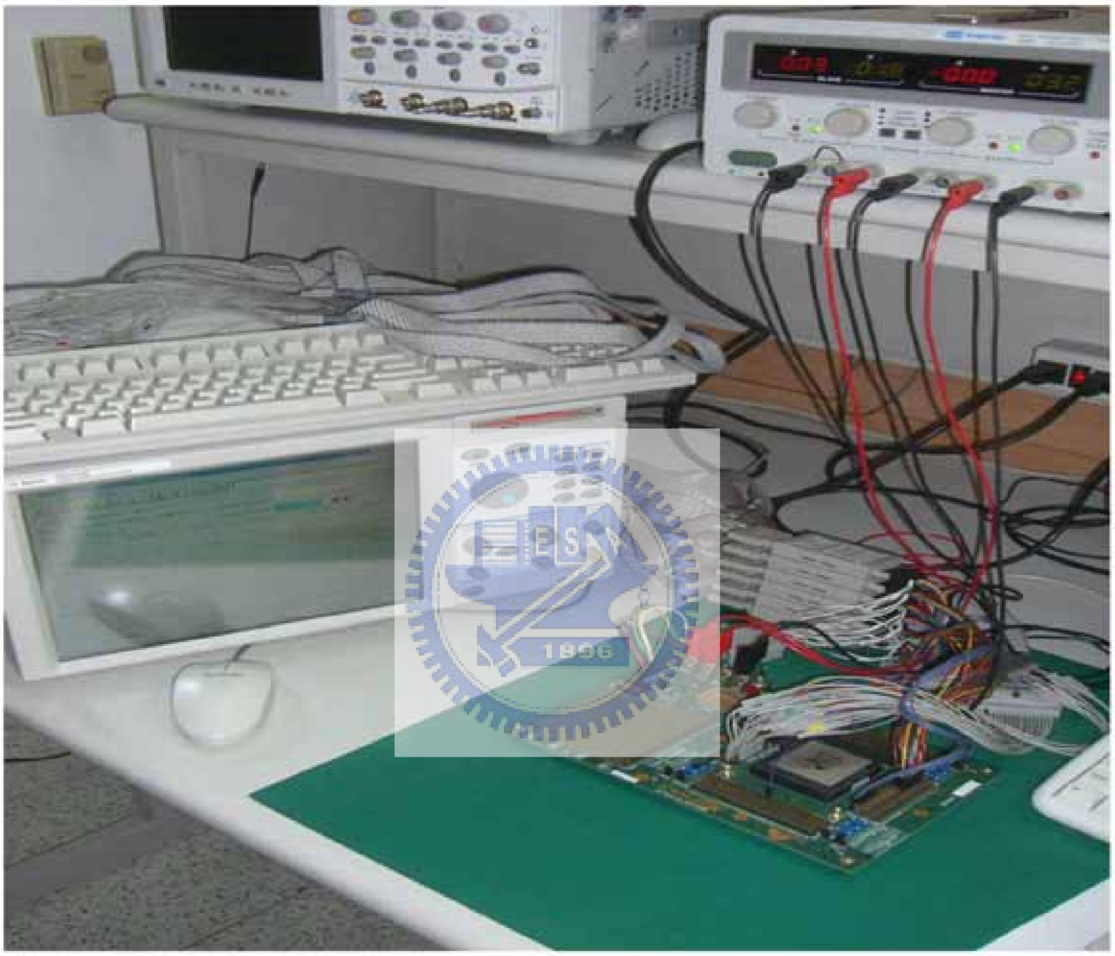


Figure 4.5：The connection of FPGA Board, pattern generator, and logic analyzer

The synthesis report of Xilinx ISE is tabulated in Table 4.1 and Figure 4.6 shows one of the simulation results in the special-case simulation.

Table 4. 1：The Synthesis Report

Total equivalent gate count for design	13531
Clock period	8.763ns

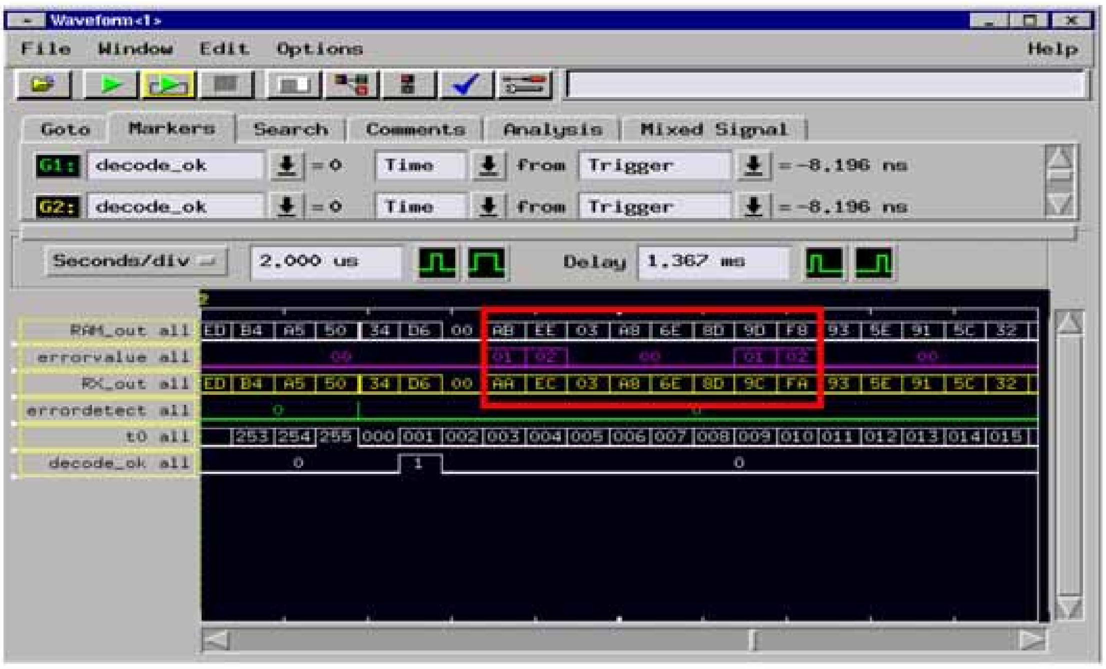

Waveform<1>
File Window Edit Options Help
Goto Markers Search Comments Analysis Mixed Signal
G1: decode_ok = 0 Time from Trigger = -8.196 ns
G2: decode_ok = 0 Time from Trigger = -8.196 ns
Seconds/div 2.000 us
Delay 1.367 ms
RAM_out all
errorvalue all
RX_out all
errordetect all
t0 all
decode_ok all
ED B4 A5 50 34 D6 00 AB EE 03 A8 6E 8D 9D F8 93 5E 91 5C 32
00 01 02 00 01 02 00
ED B4 A5 50 34 D6 00 AA EC 03 A8 6E 8D 9C FA 93 5E 91 5C 32
0
253 254 255 000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
0 1 0

Figure 4.6：One of the simulation results in the special-case simulation.

4.4 Synthesis Result

We use SYNOPSYS design compiler to synthesize the register-level Verilog file with UMC018 library. The gate counts and the power of each module are shown in Table 4.2 while the clock period is setting to 1.37ns.

Table 4.2：The performance of each synthesized module.

Module Name	Timing(ns)	Area(gates)	Power(mW)
SC Block	1.37ns	3817.0	34.9314
KES	1.37ns	5794.5	27.5
CSEE	1.37ns	3839.9	24.9064
RX_Controller	1.37ns	278.0	2.3808
Total	1.37ns	11596.8	74.9115

Chapter 5

Comparisons

After the detailed description of our design shown in the previous chapter, you may have some doubts and be interested in what the diversity is in our design. This chapter will show you the clear comparisons in algorithm, architecture, and performance with regard to our decoder design graphically.

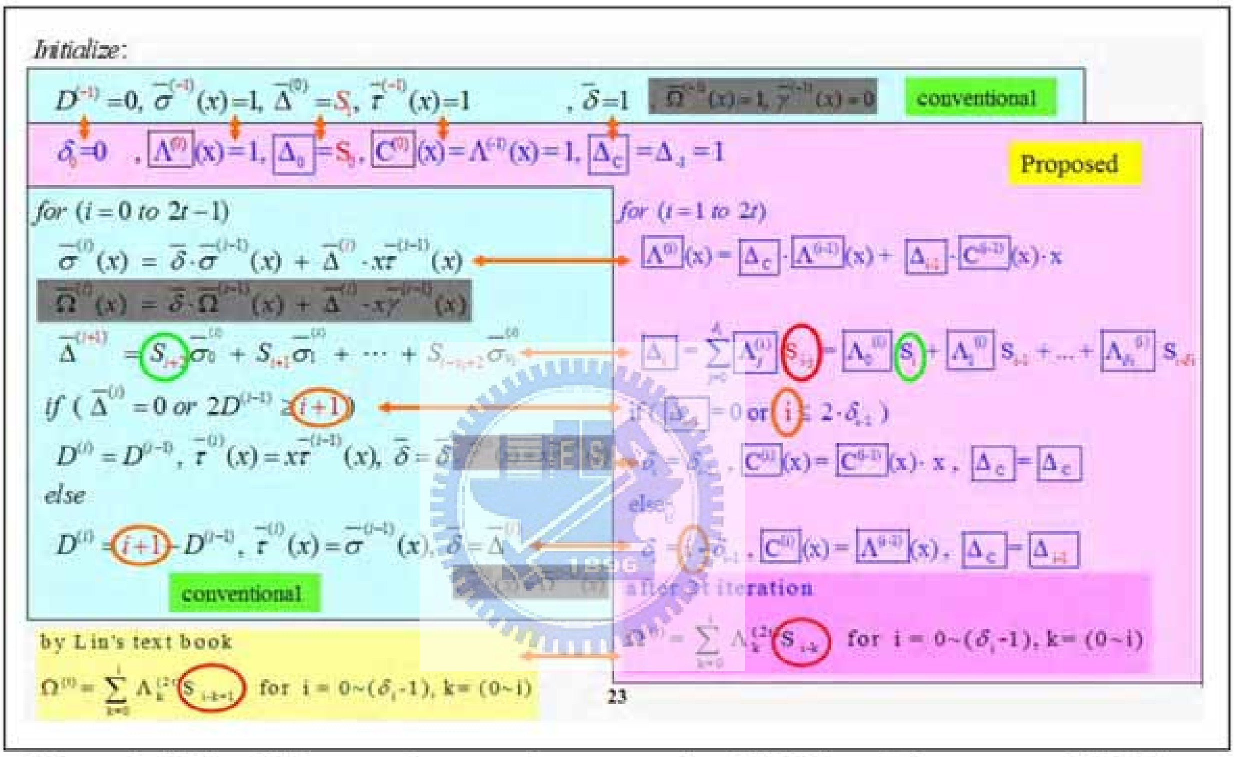
5.1 Algorithm Compare

Our design is based on the iBM algorithm and do some modifications, then the SiBM algorithm is proposed. So, we show the difference between the conventional iBMA and the proposed iBMA, followed by the comparison of the proposed SiBMA with proposed iBMA in this section.

5.1-1 Conventional iBMA versus Proposed iBMA

Figure 5.1 shows the comparison of iBMA, the conventional algorithm with cyan bottom color and the proposed algorithm with pink bottom color. Then, the differences between these two algorithms are signed in red. Thus in Figure 5.1, the major advantage of our design is that we make the selection of the syndromes (S_0~S_{15})

in the computation of Δ_i the same condition as in the computation of $\Omega^{(i)}$; it can not only save the hardware complexity but also reduce the critical path delay. However, in the conventional algorithm, the selection is different in the form of S_{i+2-j} and S_{i+1-k} as shown.


Initialize:
$D^{(-1)}=0,\ \bar{\sigma}^{(-1)}(x)=1,\ \bar{\Delta}^{(0)}=S_1,\ \bar{\tau}^{(-1)}(x)=1,\ \bar{\delta}=1$
$\bar{\Omega}^{(-1)}(x)=1,\ \bar{\gamma}^{(-1)}(x)=0$
conventional
$\delta_0=0,\ \boxed{\Lambda^{(0)}}(x)=1,\ \boxed{\Delta_0}=S_0,\ \boxed{C^{(0)}}(x)=\Lambda^{(-1)}(x)=1,\ \boxed{\Delta_C}=\Delta_{-1}=1$
Proposed
for $(i=0$ to $2t-1)$
$\bar{\sigma}^{(i)}(x)=\bar{\delta}\cdot\bar{\sigma}^{(i-1)}(x)+\bar{\Delta}^{(i)}\cdot x\bar{\tau}^{(i-1)}(x)$
$\bar{\Omega}^{(i)}(x)=\bar{\delta}\cdot\bar{\Omega}^{(i-1)}(x)+\bar{\Delta}^{(i)}\cdot x\bar{\gamma}^{(i-1)}(x)$
$\bar{\Delta}^{(i+1)}=S_{i+2}\bar{\sigma}_0^{(i)}+S_{i+1}\bar{\sigma}_1^{(i)}+\cdots+S_{i-v_i+2}\bar{\sigma}_{v_i}^{(i)}$
if $(\bar{\Delta}^{(i)}=0$ or $2D^{(i-1)}\geq i+1)$
$D^{(i)}=D^{(i-1)},\ \bar{\tau}^{(i)}(x)=x\bar{\tau}^{(i-1)}(x),\ \bar{\delta}=\bar{\delta}$
else
$D^{(i)}=i+1-D^{(i-1)},\ \bar{\tau}^{(i)}(x)=\bar{\sigma}^{(i-1)}(x),\ \bar{\delta}=\bar{\Delta}^{(i)}$
conventional
by Lin's text book
$\Omega^{(i)}=\sum_{k=0}^{i}\Lambda_k^{(2t)}S_{i-k+1}$ for $i=0\sim(\delta_t-1),\ k=(0\sim i)$
for $(i=1$ to $2t)$
$\boxed{\Lambda^{(i)}}(x)=\boxed{\Delta_C}\cdot\boxed{\Lambda^{(i-1)}}(x)+\boxed{\Delta_{i-1}}\cdot\boxed{C^{(i-1)}}(x)\cdot x$
$\boxed{\Delta_i}=\sum_{j=0}^{\delta_i}\boxed{\Lambda_j^{(i)}}S_{i-j}=\boxed{\Lambda_0^{(i)}}S_i+\boxed{\Lambda_1^{(i)}}S_{i-1}+\ldots+\boxed{\Lambda_{\delta_i}^{(i)}}S_{i-\delta_i}$
if $(\boxed{\Delta_{i-1}}=0$ or $i\leq 2\cdot\delta_{i-1})$
$\delta_i=\delta_{i-1},\ \boxed{C^{(i)}}(x)=\boxed{C^{(i-1)}}(x)\cdot x,\ \boxed{\Delta_C}=\boxed{\Delta_C}$
else
$\delta_i=i-\delta_{i-1},\ \boxed{C^{(i)}}(x)=\boxed{\Lambda^{(i-1)}}(x),\ \boxed{\Delta_C}=\boxed{\Delta_{i-1}}$
after 2t iteration
$\Omega^{(i)}=\sum_{k=0}^{i}\Lambda_k^{(2t)}S_{i-k}$ for $i=0\sim(\delta_t-1),\ k=(0\sim i)$
23
Figure 5.1：The differences between the conventional iBMA and the proposed iBMA.

5.1-2 Proposed iBMA versus Proposed SiBMA

In order to construct the serial structure of the proposed iBMA, the SiBMA is proposed. It should be noted that Figure 5.2 shows that each terms of the discrepancy in the SiBMA calculated after the calculation of $\boxed{\Lambda_j^{(i)}}$ by one clock cycle as well as the 'Decision Retiming' skill are both used to reduce the critical path delay. Then, the data stream for updating the correction polynomial C(x) is controlled to prevent special-case data hazard of serial structure. The comparison between proposed iBMA and SiBMA are shown in Figure 5.2.

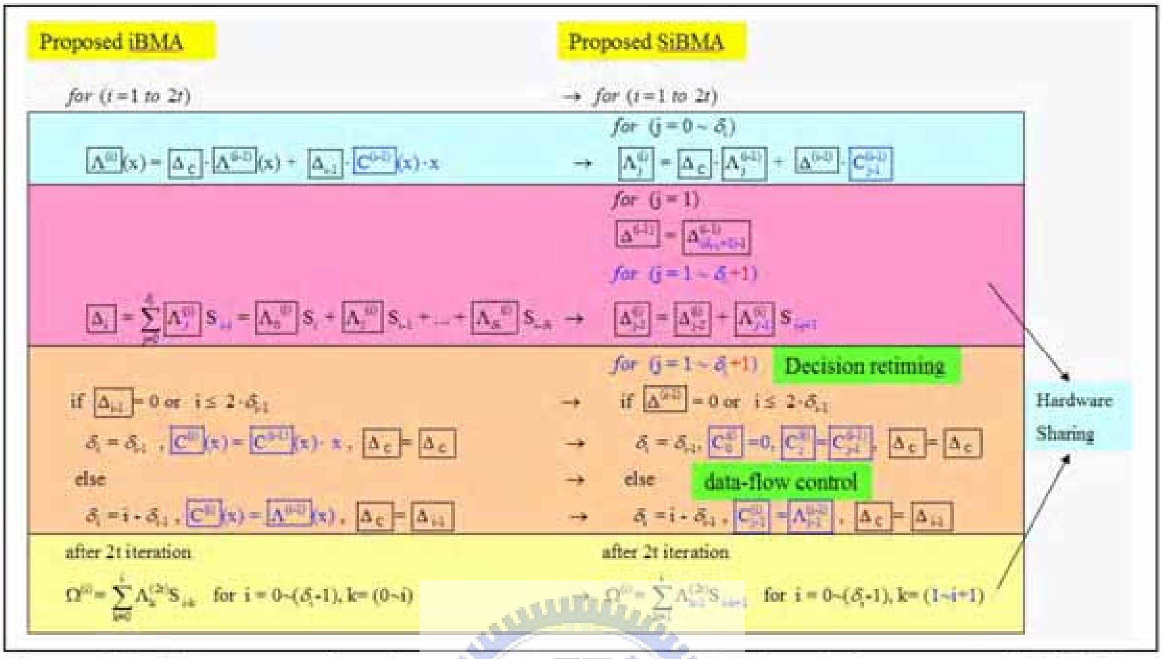


Figure 5.2：The comparison between the proposed iBMA and the proposed SiBMA.

5.2 Architecture Compare

In this section, the architecture of our design will be compared with conventional ones. The traditional parallel to serial skill and the simpler design in our creative work are both used to reduce the hardware complexity. Also, the critical path bottleneck of conventional BM series structure will be shown and be compared with our design to illustrate the speed-up rate.

5.2-1 Parallel to Serial

As many serial structures in the publications, our design also takes the advantage of traditional VLSI skill to transform the parallel structure into serial structure to reduce the hardware complexity. Figure 5.3 shows the computation of error-locator polynomial $\Lambda(x)$, our design save more than 2t FFMs (additional 4t register, and t XOR gates are needed), it counts as 4800(=2*8*300) gates at least. In other words, we save 88%(=8/9) hardware complexity in our design.

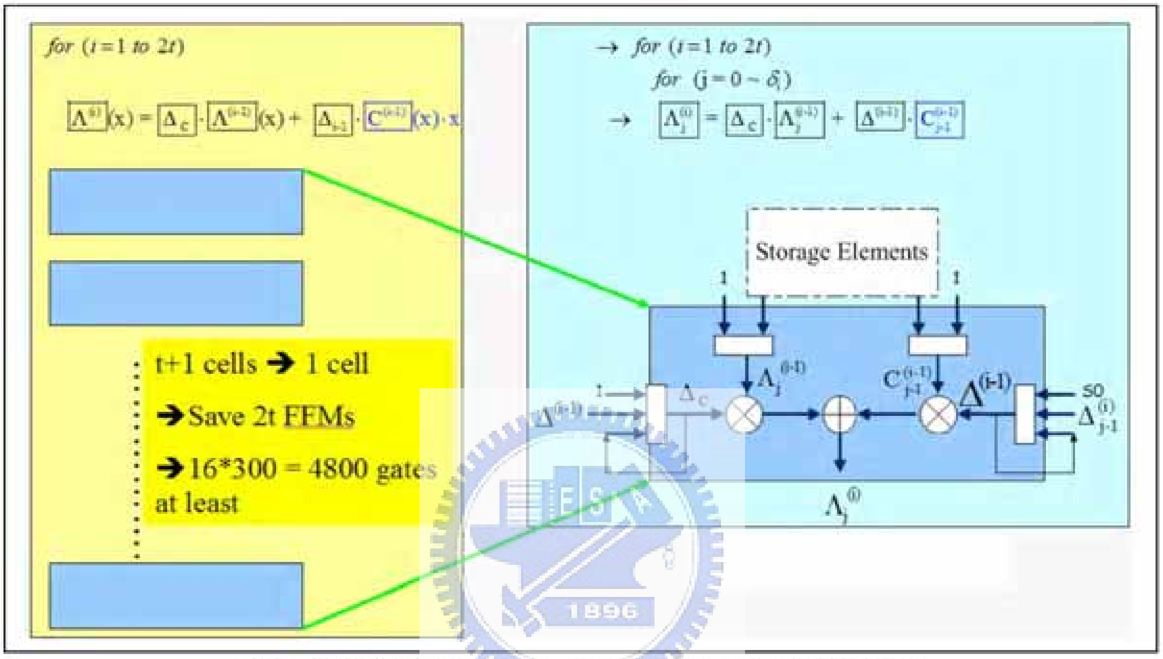
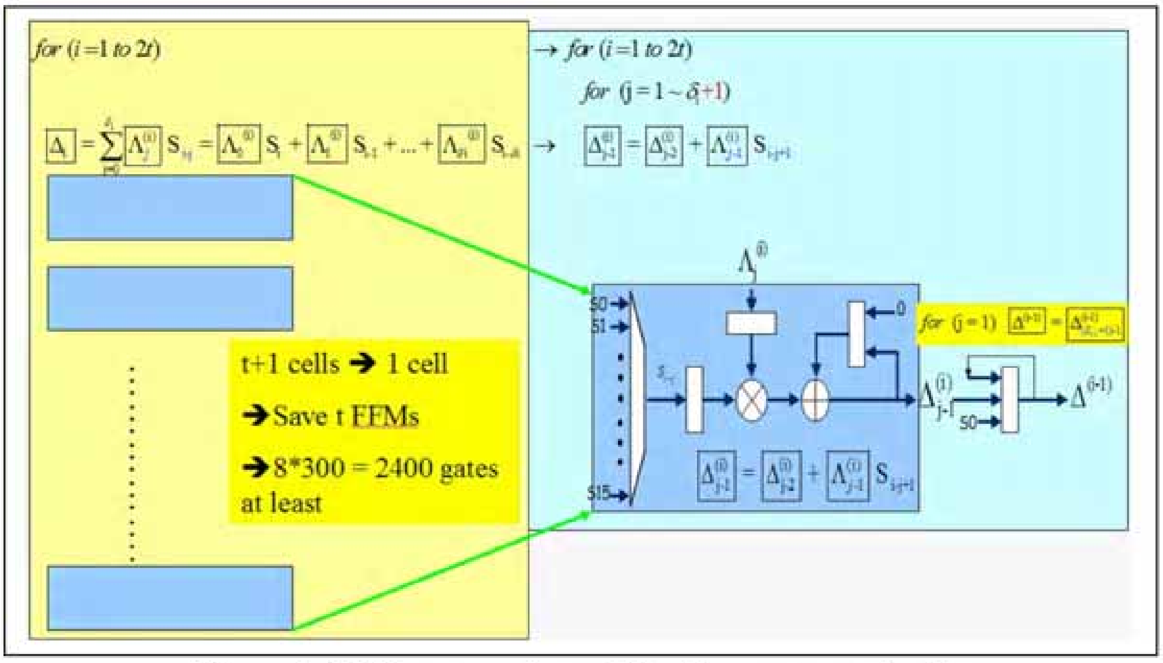


Figure 5.3：The comparison of the error-locators.

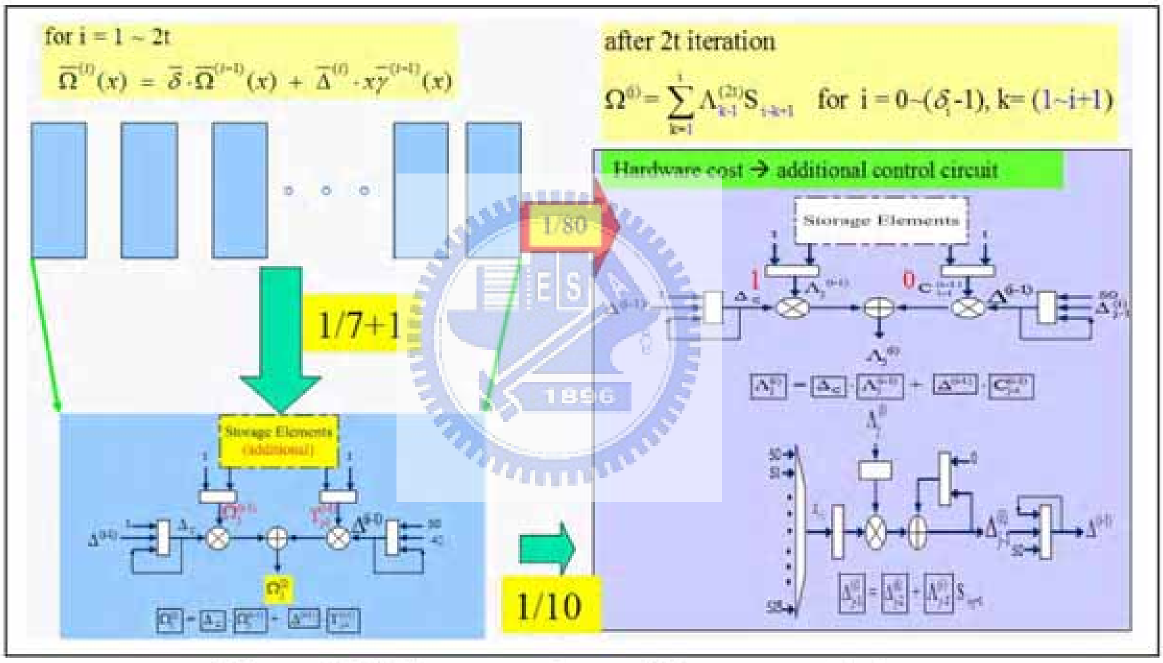


for (i=1 to 2t)
Δi = Σ Λj(i) Si-j = Λ0(i) Si + Λ1(i) Si-1 + ... + Λδi(i) Si-δi
t+1 cells → 1 cell
→Save t FFMs
→8*300 = 2400 gates at least
→ for (i=1 to 2t)
for (j=1 ~ δi+1)
→ Δj-1(i) = Δj-2(i) + Λj-1(i) Si-j+1
Λj(i)
S0
S1
S15
0
for (j=1) Δ(i-1) = Δ(i-1)
Δj-1(i)
S0
Δ(i-1)
Δj-1(i) = Δj-2(i) + Λj-1(i) Si-j+1

Figure 5.4：The comparison of the discrepancy calculator.

The computation of discrepancy is shown in Figure 5.4, and the reduced hardware is also in the ratio 88%.

As the comparison of Figure 5.5, the hardware complexity of the error-evaluator for evaluating $\Omega(x)$ in the serial structure is 1/8 times the size of that in the parallel structure. If we further combine the error-evaluator into the circuit that we design to calculate the discrepancy, the hardware cost will be additional control circuit at most. Thus, we save 99%(=79/80) hardware complexity in our design.

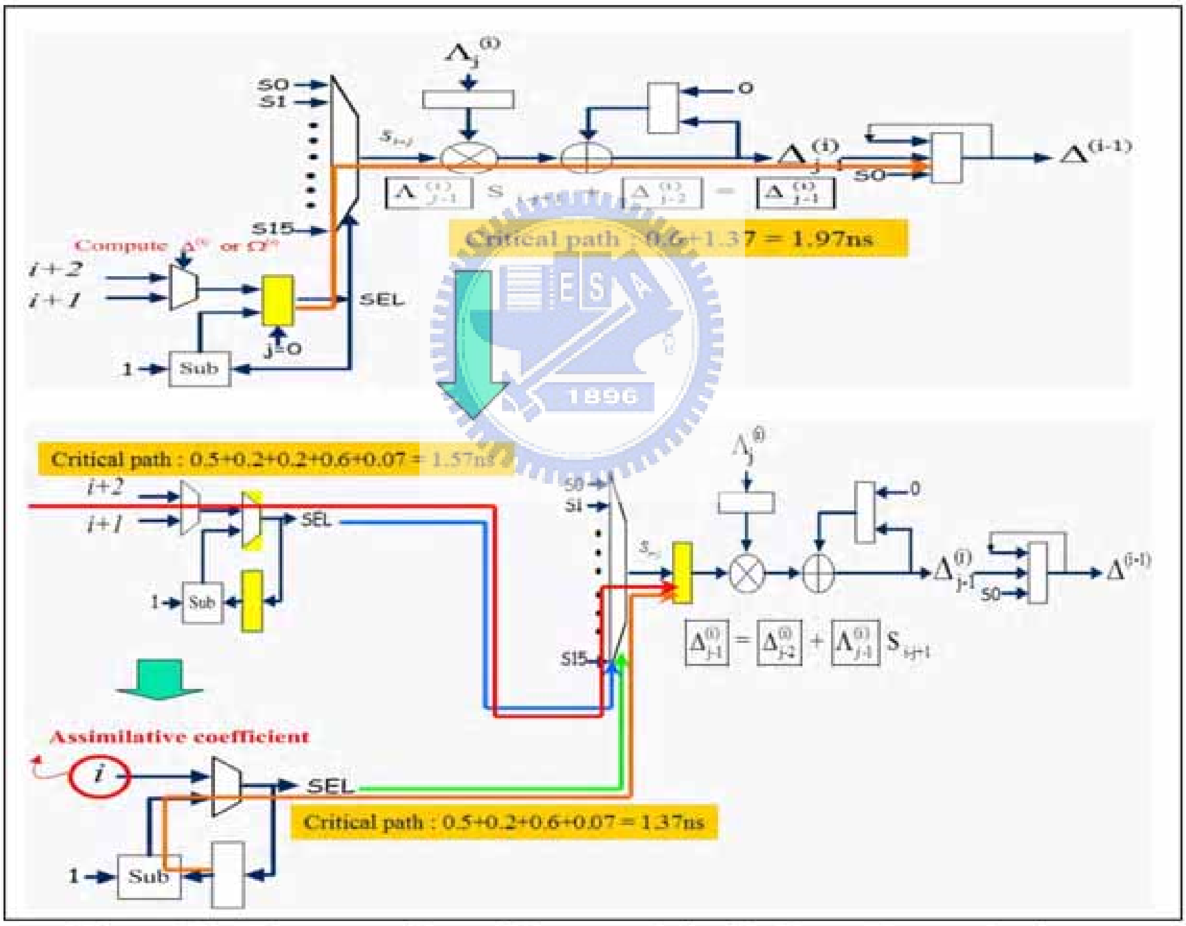


for i = 1 ~ 2t
$\overline{\Omega}^{(i)}(x) = \overline{\delta}\cdot\overline{\Omega}^{(i-1)}(x) + \overline{\Delta}^{(i)}\cdot x\overline{\gamma}^{(i-1)}(x)$
after 2t iteration
$\Omega^{(i)} = \sum_{k=1}^{i} \Lambda_{k-1}^{(2t)} S_{i-k+1}$ for i = 0~(δ_i-1), k= (1~i+1)
Hardware cost → additional control circuit
1/80
1/7+1
1/10
Storage Elements
Storage Elements (additional)

Figure 5.5：The comparison of the error-evaluators.

5.2-2 Speed up

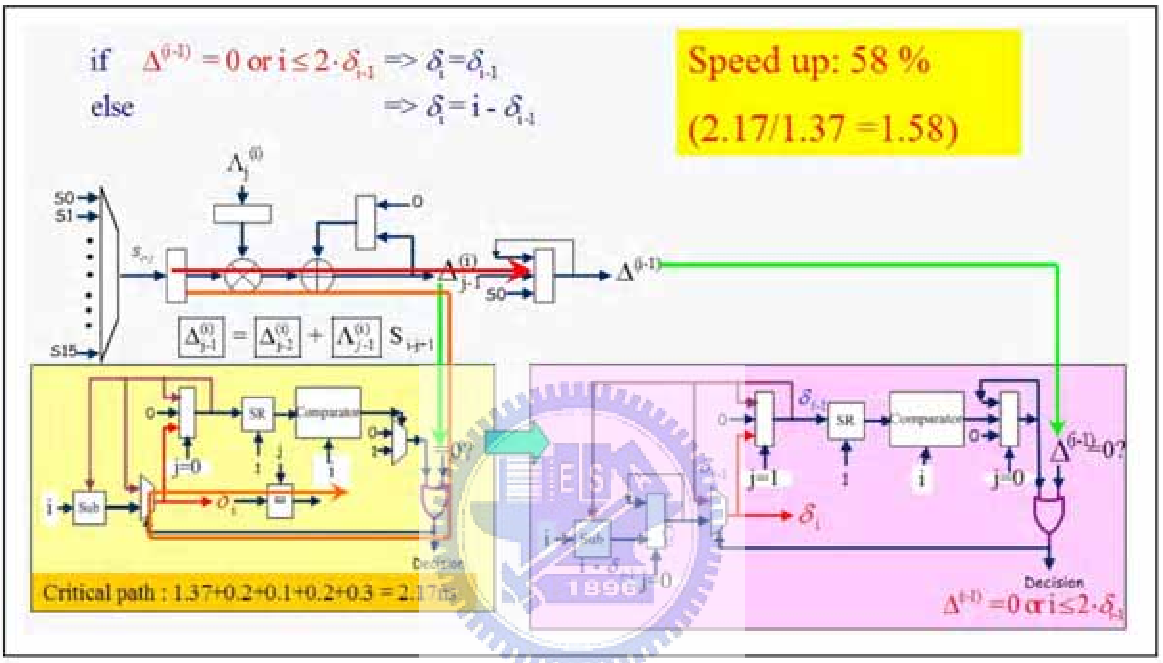
Some publications shown the critical path delay is $T_{ff} + T_{mult} + T_{xor} + T_{mux\_2t\text{-}to\text{-}1}$, as drawn at the top of Figure 5.6. The control circuit of selection line must settle on a proper initial value in the j=0, and decrease by one in successive clock period. Meaning that one register in the control circuit is needed and then the initial value is valid while j=1. If additional register is following the 16-to-1 multiplexer, the initial syndrome value cannot be multiplied by Λ_0 while j=1. The timing mismatch can be solved by some modifications. The middle of Figure 5.6 shows the efficient design and the reduced critical path is in the ratio 25% (=1.97/1.57). Nevertheless, there is a little longer than desirable cycle time, so the 'Assimilative Coefficient' is adopted as shown at the bottom of Figure 5.6. Then, the overall speed-up rate is 45% (=1.97/1.37).



Critical path : 0.6+1.37 = 1.97ns
Compute Δ or Ω
i+2
i+1
SEL
Sub
j=0
Critical path : 0.5+0.2+0.2+0.6+0.07 = 1.57ns
Assimilative coefficient
Critical path : 0.5+0.2+0.6+0.07 = 1.37ns
S0
S1
S15

Figure 5.6：The comparison of the control circuits for syndrome selection.

In the normal condition, the decision of the degree in the i-th iteration and the data stream for updating C(x) must be determined in the first cycle of the i-th iteration. It will increase the critical path delay, so the 'Decision Variation' is adopted to fetch through this bottleneck. As illustrated in the figure 5.7, the normal condition drawn with yellow bottom-color is having the critical path delay about 2.17ns. However, the structure in our elaborate design keeping the maximum path delay is still 1.37ns as shown in the right side with mauve bottom-color. In other words, we have speed up the clock rate in the ratio 58%.



if $\Delta^{(i-1)} = 0$ or $i \le 2\cdot\delta_{i-1}$ => $\delta_i = \delta_{i-1}$
else => $\delta_i = i - \delta_{i-1}$
Speed up: 58 %
(2.17/1.37 =1.58)
$\Delta^{(i)}_{j-1} = \Delta^{(i)}_{j-2} + \Lambda^{(i)}_{j-1} S_{i-j+1}$
SR
Comparator
Sub
Critical path : 1.37+0.2+0.1+0.2+0.3 = 2.17ns
Decision
$\Delta^{(i-1)} = 0$ or $i \le 2\cdot\delta_{i-1}$

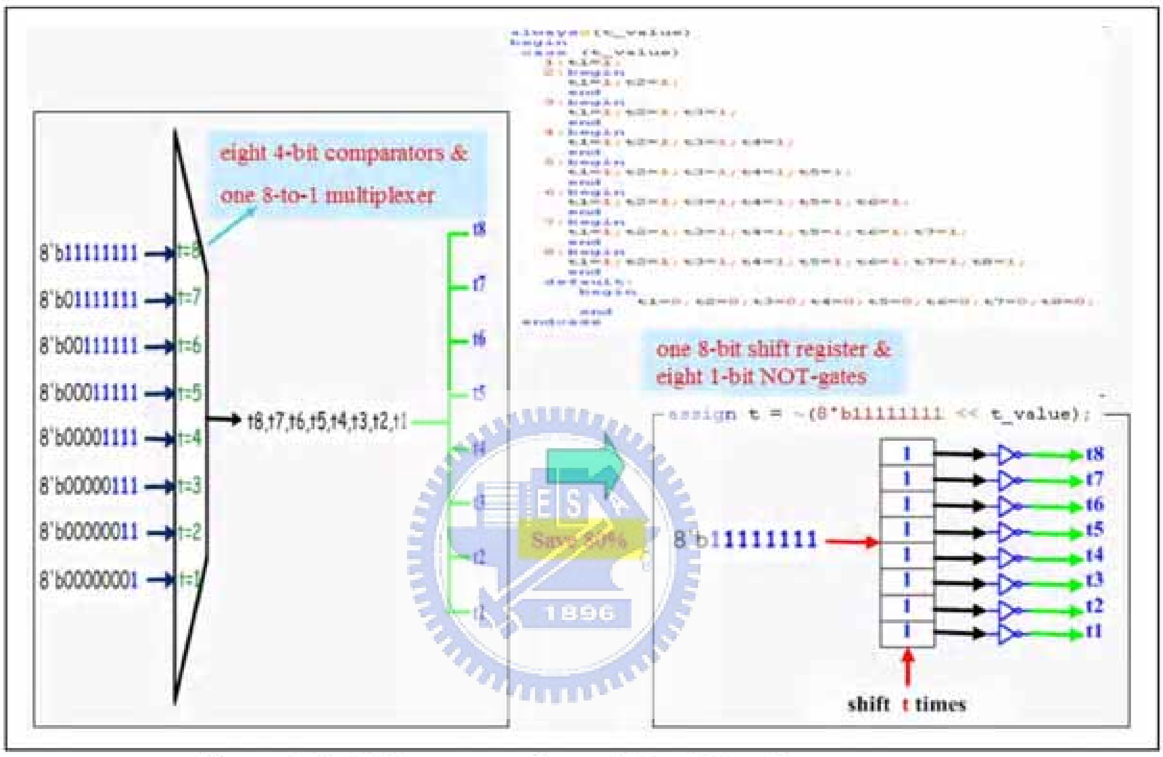
Figure 5.7：The comparison between the normal condition and the "Decision Variation".

5.2-3 Simpler Design

In this section, we list some of our simpler design that have never remarked in other technical literature. Especially, the common compensator and the purpose-built address line are employed to reduce the hardware complexity greatly.

The t-decoder

As mentioned in Section 3.2, for multi-mode applications, the t-decoder is exploited to dominate the sixteen cells in the encoder or in the syndrome calculator. We make some comparisons between conventional and our design as shown in Figure 5.8. The hardware complexity is greatly reduced in the ratio 80%. It is also, the RTL code shown that our design is just occupying one line.



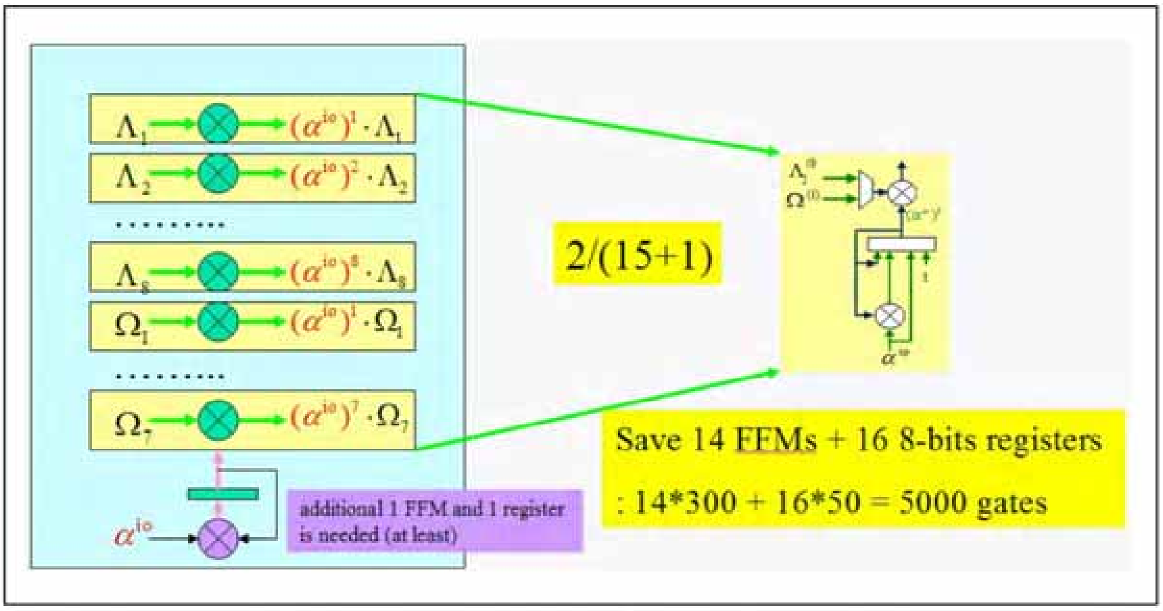
```
always@(t_value)
begin
  case  (t_value)
    1: t1=1;
    2: begin
       t1=1; t2=1;
       end
    3: begin
       t1=1; t2=1; t3=1;
       end
    4: begin
       t1=1; t2=1; t3=1; t4=1;
       end
    5: begin
       t1=1; t2=1; t3=1; t4=1; t5=1;
       end
    6: begin
       t1=1; t2=1; t3=1; t4=1; t5=1; t6=1;
       end
    7: begin
       t1=1; t2=1; t3=1; t4=1; t5=1; t6=1; t7=1;
       end
    8: begin
       t1=1; t2=1; t3=1; t4=1; t5=1; t6=1; t7=1; t8=1;
       end
    default:
       begin
              t1=0; t2=0; t3=0; t4=0; t5=0; t6=0; t7=0; t8=0;
       end
endcase
```

```
assign t = ~(8'b11111111 << t_value);
```

Figure 5.8 : The comparison of the t-decoders.

The common compensator

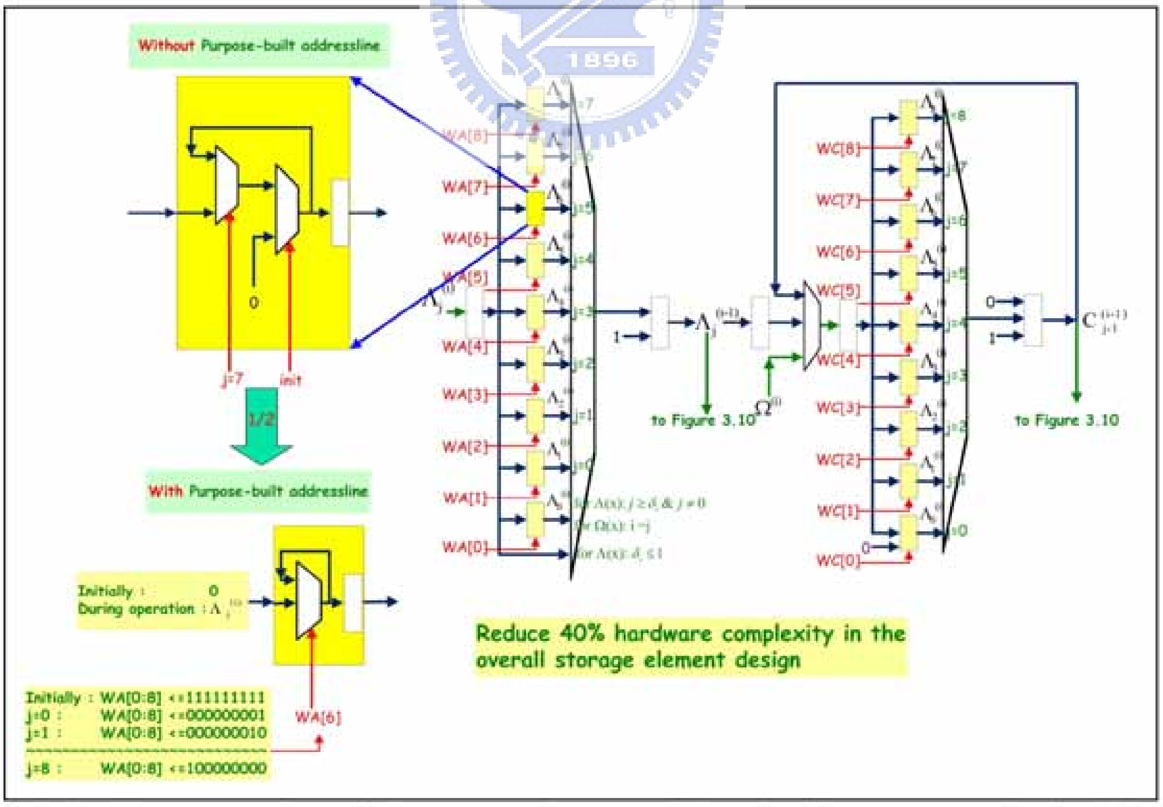
Recall in Section 3.6, the common compensator is employed to adjust starting point of Chien's Search and to evaluate the corresponding error value. Thus in Figure 5.9, there are (15+1) FFMs in the conventional compensator design but only 2 FFMs in the common-compensator design. In other words, we reduce the hardware complexity in the ratio 88% (=14/16).



$\Lambda_1 \rightarrow (\alpha^{io})^1 \cdot \Lambda_1$
$\Lambda_2 \rightarrow (\alpha^{io})^2 \cdot \Lambda_2$
$\Lambda_8 \rightarrow (\alpha^{io})^8 \cdot \Lambda_8$
$\Omega_1 \rightarrow (\alpha^{io})^1 \cdot \Omega_1$
$\Omega_7 \rightarrow (\alpha^{io})^7 \cdot \Omega_7$
α^{io}
additional 1 FFM and 1 register is needed (at least)
2/(15+1)
Save 14 FFMs + 16 8-bits registers
: 14*300 + 16*50 = 5000 gates

Figure 5.9 : The comparison between the conventional design and the common compensators.

Purpose-built address line



Without Purpose-built addressline
j=7
init
1/2
With Purpose-built addressline
Initially : 0
During operation : $\Lambda_j^{(i)}$
Initially : WA[0:8] <=111111111
j=0 : WA[0:8] <=000000001
j=1 : WA[0:8] <=000000010
j=8 : WA[0:8] <=100000000
WA[6]
WA[8]
WA[7]
WA[6]
WA[5]
WA[4]
WA[3]
WA[2]
WA[1]
WA[0]
to Figure 3.10
WC[8]
WC[7]
WC[6]
WC[5]
WC[4]
WC[3]
WC[2]
WC[1]
WC[0]
to Figure 3.10
Reduce 40% hardware complexity in the overall storage element design

Figure 5.10 : The comparison of the storage element with and without purpose-built address line.

The storage element shown in Figure 5.10 is the lowest hardware complexity in the multi-mode design that I have tried. The purpose-built address line is utilized to reduce the hardware cost and to increase the flexibility of inserting data stream. If the register with yellow mark is controlled by the purpose-built address line, the hardware cost will be about half of the without one as shown in the left side of this picture. The overall storage element design can be reduced in the ratio of 40%, and any data stream can be stored into this element easily by setting this address line in a proper condition. It is why the output data stream of common compensator described in Section 3.6 and shown in Figure 5.9 can be inserted in this element.

5.3 Performance Compare

Fortunately, our design can not only operate at higher clock rates but also have lower hardware complexity for multi-mode applications. Table 5.1 shows the comparison with other design of publications. In this table, the hardware complexity of our decoder design is the lowest, and clock rate is the fastest except the third column synthesized in the more advanced technology and with huge area.

Table 5.1：The comparison with publications.

m=8	Single Mode					Multi-Mode	
Publications	[30] IEE2001	[31] IEEE 2003	[32] IEEE2003	[33] IEEE 2005 PrME	[12] ISCAS2006	[12] ISCAS2006	Proposed
KES Unit	43100	44700	102500	17000	9566	10405	5794.5

Decoder (w/o FIFO)	NA	54000	115500	24600	20224	22931	11596.8
Clock Rate (MHz)	300	300	770	625	400	400	730
Technology	0.16	0.13	0.13	0.13	0.18	0.18	0.18

Chapter 6

Conclusions & Future work

6.1 Summary

The main characters of our design focus on the high performance multi-mode systems. In this work, a SiBM algorithm for solving the key equations is proposed. Then, a propose SiBM architecture is presented and many VLSI knacks are employed to reduce the hardware complexity as well as the critical path delay. For instance, a simpler t-decoder, a common compensator, and the assimilative coefficient skill as well as the purpose-built address line are employed in the multi-mode system, the hardware complexity is reduced. For speeding up the critical path delay, the "Decision Variation" is exploited in the KES block.

All functionalities of our design are verified by one hundred-million random code-words and special patterns simulation. Furthermore, the FPGA simulations are also regular, that is, our design can work well in the real world. For comparing with other designs, we must evaluate the performance by the design complier. For the valid estimation, the gate-level simulation is performed. After check all functionalities of synthesized gate-level hardware, the report shows the clock period is 1.37ns with 11596 gates. That is, the data rate can reach 5.84Gbps at maximum clock rate

730MHz.

6.2 Future Works

To make a comprehensive survey of our design, the contributions almost focus on the architecture design. This is just because we too take the gate-count to heart in the beginning of research to disregard other good ideas, such as reconfigurable property, universal architecture or a well-designed soft decision decoding. In fact, these contributions are greater than the low-complexity ones while the technology with giant strides. So, we should make more efforts in the research and design of innovating algorithms in the future.

Bibliography

[1] Reed, I. S. and Solomon, G., 1960. Polynomial Codes over Certain Finite Fields, J. SIAM.,Vol. 8, pp. 300-304, 1960.

[2] Bose, R. C. and Chaudhuri, D. K. R., 1960. On a class of error-correcting binary group codes. Inf. Control, Vol. 3, 1960.

[3] Hocquenghem, A., 1959. Codes correcteurs d'erreurs, Chiffres., Vol. 2, 1959.

[4] S. Whitaker, J. Canaris, and K. Cameron, "*Reed-Solomon VLSI codec for advanced television*," IEEE Trans. Circuits System Video Technol., vol. 1, pp.230-236, June 1991.

[5] I.S. Reed, R. He, X. Chen, and T.K. Truong, "*Application of Grobner bases for decoding Reed-Solomon codes used on CDs*," IEE Proceedings-Computers and Digital Techniques, vol. 145, issue. 6, pp. 369-376, Nov. 1998.

[6] H.C. Chang, C.B. Shung, "*A (208,192;8) Reed-Solomon decoder for DVD application*," IEEE International Conference, pp. 957-960, vol. 2, 1998.

[7] "*Annex B to ITU-T Recommendation J.83, Digital multi-programme systems for television sound and data services for cable distribution*," Oct. 1995.

[8] Walter Y. Chen, "*DSL: Simulation Techniques and Standards Development for Digital Subscriber Line Systems*," Macmillan Technical Publishing, Indianapolis, 1998.

[9] Dennis J. Rauschmayer, "*ADSL/VDSL Principles: A Pracitical and Precise Study of Asymmetric Digital Subscriber Lines and Very High Speed Digital Subscriber Lines*," Macmillan Technical Publishing, Indianapolis, 1999.

[10] H.-C. Chang and C. Shung, "A Reed-Solomon product code (RS-PC) decoder for

DVD applications," in *Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1998, pp. 390–391.

[11] W. Wilhelm, "A new scalable VLSI architecture for Reed–Solomon decoders, "*IEEE J. Solid-State Circuits*, vol. 34, pp. 388–396, Mar. 1999.

[12] Ming-Der Shieh, Yung-Kuei Lu, Shen-Ming Chung, and Jun-Hong Chen," Design and Implementation of Efficient Reed-Solomon Decoders for Multi-Mode Applications," in Digital Object Identifier 10.1109/ISCAS.2006.1692579, 21-24 May 2006 Page(s):4 pp.

[13] E. R. Berlekamp, G. Seroussi, and P. Tong, *Reed–Solomon Codes and Their Applications*, S. B.Wicker and V. K. Bhargava, Eds. Piscataway, NJ: IEEE Press, 1994. A hyper-systolic Reed–Solomon decoder.

[14] S. Kwon and H. Shin, "An area-efficient VLSI architecture of a Reed–Solomon decoder/encoder for digital VCRs," *IEEE Trans. Consumer Electron.*, pp. 1019–1027, Nov. 1997.

[15] P. Tong, "A 40 MHz encoder-decoder chip generated by a Reed-Solomon code compiler," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 13.5.1–13.5.4, May 1990.

[16] R. E. Blahut, *Theory and Practice of Error-Control Codes*. Reading, MA: Addison-Wesley, 1983.

[17] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968. (revised ed.—Laguna Hills, CA: Aegean Park, 1984).

[18] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122–127, Mar. 1969.

[19] D. V. Sarwate, and N. R. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders", IEEE Transactions on VLSI systems, vol. 9, no. 5, pp. 641-655. October 2001.

[20] C.K.P Clarke, Reed-Solomon error correction, BBC Research & Development, July 2002.

[21] E. Berlekamp, Algebraic Coding Theory, New York: McGraw-Hill, 1968.

[22] J. Massey, "Shift-register synthesis and BCH decoding," IEEE Trans. Inform. Theory, vol. IT-15, pp122-127, Jan. 1969

[23] S. Lin and D. Costello, Error Control Coding: Fundamentals and Applications, Englewood Cliffs., NJ: Prentice-Hall, 1983.

[24] I. Reed, M. Shih, and T.-K. Truong, "VLSI design of inverse-free Berlekamp-Massey algorithm", Proc. Inst. Elect. Eng. pt. E, vol. 138, pp. 295–298, Sept.1991.

[25] K.-Y. Liu, "Architecture for VLSI Design of Reed-Solomon decoders", IEEE Trans. Comput., vol. C-33, pp. 178–189, Feb. 1984.

[26]Hsie-Chia Chang, and Shung, C.B, "New serial architecture for the Berlekamp-Massey algorithm," IEEE Trans. on Commun., Page(s):481 – 483, April, 1999.

[27]Chien-Ching Lin, Fuh-Ke Chang, Hsie-Chia Chang, and Chen-Yi Lee, "An Universal VLSI Architecture for Bit-parallel computation in GF(2m)," *IEEE Asia-Pacific Conference Circuits and Systems*, 6-9 Dec, 2004

[28]Fuh-Ke Chang , Chien-Ching Lin, Hsie-Chia Chang, and Chen-Yi Lee, "An Universal Architecture for Reed-Solomon Erasure Decoder," *IEEE ASSCC*, 1-3 Nov, 2005

[29] G. Forney, "On decoding BCH codes," IEEE Trans. Inform. Theory, vol. IT-11, pp.549–557, Oct. 1965

[30] H. Lee, "Modified Ecuclidean Algorithm Block for High-Speed Reed-Solomon Decoder." IEE Electronics Letters, vol.37, no. 14, pp903-904, July 2001.

[31] H. Lee, "An Area-Efficient Euclidean Algorithm Block for Reed-Solomon

Decode," proc. IEEE Computer Society Annual Symposium on VLSI, pp.209-210, Feb. 2003.

[32] H. Lee, "High-Speed VLSI Architecture for parallel Reed-Solomon Decoder," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 11, no. 2, pp.288-294, April 2003.

[33] Hanho Lee, "A High-Speed Low-Complexity Reed-Solomon Decoder for Optical Communications" IEEE Transactions on circuits and systems, vol. 52, NO. 8, August 2005.

Vita

蘇建毓，民國六十八年出生於高雄市。民國九十年自私立義守大學電子工程學系取得學士學位，九二年退役後於電子業界工作二年餘，並於九四年進入國立交通大學IC設計產業研發碩士班就讀。作者從事之研究領域為無線通訊系統及儲存元件之通道解碼器架構設計與硬體實作。民國九十七年一月取得碩士學位，學位論文題目為：「適用於多模式高速通訊系統之低複雜度里德所羅門編解碼模組」。