National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

Versatile Reed-Solomon Decoders

Yousef Rajableh-Shayan

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

August, 1990

© Yousef R. Shayan, 1990

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-64738-8

Canada

ABSTRACT

VERSATILE REED-SOLOMON DECODERS

Yousef Rajabieh-Shayan, Ph. D.,
Concordia University, 1990.

This thesis is concerned with the development of versatile Reed-Solomon (RS) decoder structures. Reed-Solomon decoding algorithms can be divided into two categories: i) syndrome-based algorithms which can be further divided into algebraic decoding and transform decoding algorithms, ii) the time-domain decoding algorithms which are time-domain equivalent of syndrome-based algorithms. These two categories of algorithms are compared from the structural point of view and it is shown that time-domain algorithms are the best candidates for designing versatile hardware decoders. However, for the software decoders, it is advantageous to use the syndrome-based algorithms.

The decoding algorithms of RS codes require algebraic operations over Galois fields. Parallel-in/parallel-out multipliers and inverters in Galois fields are considered and least complex structures for the multiplier are introduced both in standard and normal basis. A new normal basis multiplier is presented which is less complex compared to the Massey-Omura multiplier. A universal multiplier is also introduced which can multiply two elements of Galois field 2^m, $m = 4,5,6,7,8$.

In this thesis, the time-domain algorithm based on transform decoder is restructured and two versatile decoder structures are presented. Both of these structures are simple and modular, and hence suitable for Very-Large-Scale-Integration (VLSI) design. These versatile decoders can be used for decoding any primitive RS code defined in a specific Galois field. The error correction capability of these versatile decoders is configurable and they can correct both errors and erasures. The structure of a universal RS decoder is also presented. The time-domain decoding algorithm based on algebraic decoder is modified to reduce the complexity of the universal decoder. This universal decoder can decode any primitive RS code in Galois field 2^m, for m =4,5,6,7,8. The error correction capability and the size for the Galois field of this decoder are configurable. To increase the versatility of the decoder, a method is also introduced for decoding the RS codes generated by any generator polynomial.

Dedicated to My Parents

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my thesis supervisor, Dr. Tho Le-Ngoc, for his continued guidance, suggestions and encouragement during the course of this research. I am very grateful to Dr. V. K. Bhargava for many valuable discussions, and for providing me with recent literature.

I express my deep appreciation to my wife, Fariba, for her love, patience, and encouragement during my studies.

I would also like to thank Dr. M. R. Soleymani and Mr. S. H. Jamali for proof reading of the text of this work and helping in final preparation of this thesis.

This work was supported in part by Canada NSERC under Grants A5987, A32671 and Quebec FCAR under Grant ER-0106.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

BCH	Bose-Chaudhuri-Hocquenghem
CLB	Configurable-Logic-Blocks
CMOS	Complementary-Metal-Oxide-Semiconductor
CPU	Central-Processing-Unit
FFT	Fast-Fourier-Transform
GF	Galois-Field
GCD	Greatest-Common-Divisor
IC	Integrated-Circuit
IOB	Input/Output-Blocks
LCA	Logic-Cell-Array
LCM	Least-Common-Multiple
LFSR	Linear-Feedback-Shift-Register
MDS	Maximum-Distance-Separable
MSI	Medium-Scale-Integration
NOT	Inverter-Gate
ROM	Read-Only-Memory
RS	Reed-Solomon
SI	Segment-Identifier
SSI	Small-Scale-Integration
VLSI	Very-Large-Scale-Integration
XOR	Exclusive-Or-Gate

LIST OF SYMBOLS

k	Number of Information Symbols
m	Size of Each Symbol
n	Number of Code Word Symbols of Primitive Reed-Solomon Codes
r	Iteration Number
t	Error Correction Capability of the Code
α	The Primitive Element in Galois Field 2^m
ρ	Number of Erasures
ν	Number of Errors
$\underline{c}$	Code Word Vector
$\underline{d}$	Information Vector
$\underline{e}$	Error or Errata Vector
$\underline{v}$	Received Vector to the Decoder
$\underline{C}$	Code Word Vector in the Frequency Domain
$\underline{D}$	Information Vector in the Frequency Domain
$\underline{E}$	Error or Errata Vector in the Frequency Domain
$\underline{\gamma}$	Erasure Locator Vector
$\underline{\lambda}$	Error or Errata Locator Vector
$\underline{\omega}$	Error or Errata Evaluator Vector
$c(x)$	Code Word Polynomial
$d(x)$	Information Polynomial
$e(x)$	Error or Errata Locator Polynomial
$g(x)$	Generator Polynomial of the Reed-Solomon Code

$p(x)$	Parity Polynomial
$v(x)$	Received Polynomial to the Decoder
$S(x)$	Syndrome Polynomial
$I(x)$	Irreducible Polynomial over GF(2)
$\Gamma(x)$	Erasure Locator Polynomial
$\Lambda(x)$	Error or Errata Locator Polynomial
$\Omega(x)$	Error or Errata Evaluator Polynomial
S_j	j the Syndrome
X_l	Error Locator Number for the Error at Location l
Y_l	Error Value for the Error at Location l

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

1.1. Motivation

There are many communication channels which are affected by disturbances that cause transmission errors to cluster into bursts. For example, on telephone lines, a stroke of lightning or a human made electrical disturbance frequently affects many adjacent transmitted digits. Other examples of bursty noise are intentional jamming, and fading. Storage media such as film, magnetic tape and magnetic disk also suffer from burst errors. Burst errors can occur on these systems through scratches, defects, etc.

A typical data transmission system (or a data storage system) can be represented by the block diagram as shown in Fig. 1.1. The channel encoder, according to some rules, adds some redundancy to the digital information sequence $\underline{d}$ and transforms it into a longer binary sequence $\underline{c}$ called a code word. The channel encoder is implemented to cope with the noisy condition in which the resulting code sequence is to be transmitted or stored. The function of the modulator is to assign a signal to each output digit of the channel encoder. The output of the modulator enters the channel and is disturbed by noise. The demodulator makes a decision for each received signal, to determine the transmitted digit, e.g., 0 or 1 in the case of binary transmission. Thus, the output of the demodulator is a sequence $\underline{v}$ of (binary) digits. Because of the channel noise disturbance, the received sequence $\underline{v}$ might not match the code word $\underline{c}$. The code used for channel encoding should be such that the code words have the capability of combating the transmission errors $\underline{e}$. The channel decoder, based on the received sequence $\underline{v}$, and the rules of channel encoding, tries to correct the errors. If the errors introduced in the channel are not more than the error correction capability of the code, the decoder corrects all the errors and outputs the transmitted information sequence $\underline{d}$.

The choice of code depends on the characteristics of the channel. For channels which are disturbed by burst errors, non-binary Bose-Chaudhuri-Hocquenghem (BCH) codes are suitable. Reed-Solomon (RS) codes, discovered by Reed and Solomon in 1960 [1], are an important subclass of non-binary BCH codes. They are also an important subclass of Maximum Distance Separable (MDS) codes. Reed-Solomon codes have a very large burst error correcting capability when the symbols are transmitted bit by bit. These codes can also correct a large number of random errors and erasures in a code word. RS codes are normally used concatenated with convolutional codes, in order to make the use of the possible soft decision information of the demodulator.

Recently, RS codes have found many applications due to their error correcting capabilities and optimum structure. For example they are used in space, spread spectrum and mobile communications [2]-[10]. They have also widespread applications in magnetic and optical recording systems [11]-[13]. Every particular application has its own distinct requirements such as the error correction capability of the code and the code word length. Moreover, the encoder and decoder structures are different from throughput and complexity points of view.

For encoding RS codes, the general non-binary BCH encoding algorithms are used. The RS encoding algorithms are simple in structure and can be used to design RS encoders. Recently, Berlekamp has developed new concepts and techniques for implementation of RS encoders [14] which reduces the necessary hardware. Using this concept an RS(255, 223) encoder has been realized on a single Very-Large-Scale-Integration (VLSI) chip [15].

The existing algorithms for decoding RS codes have complicated structures. To design an RS decoder with specified requirements, a long design time is needed and the outcome is a complex decoder with a large number of Integrated-Circuits (IC). This results in a large, high weight, high power consumption and unreliable RS decoder, which is a disadvantage specially in space communications. To solve this problem, VLSI implementation of RS decoders can be considered. VLSI systems have the potential advantages of significant savings in size, weight, and consumption while at the same time providing higher reliability over systems implemented in discrete logic circuits.

VLSI design is time consuming and very expensive. To decrease the design time, cellular structures for RS decoders should be introduced. It may also be efficient to develop a universal decoder on a VLSI chip, to save the design cost of many different RS decoders. By a universal decoder we mean a decoder that can be used to decode any primitive RS code† up to the limits of the storage registers associated with the chip. Within these limits it should correct any number of errors and erasures, depending on the code, and for any block length and symbol alphabet.

1.2. Existing Reed-Solomon Decoders

One of the decoding algorithms for Reed-Solomon codes is the algebraic decoding algorithm [16]. In this algorithm, the syndrome is computed from the received vector. This step can be considered as a transformation of the received vector into the frequency domain. Then the error locations and error values are computed in time domain and finally the error correction is

† In this thesis we only consider primitive RS codes.

performed. This algorithm is performed partly in time and partly in frequency domain and hence it is called hybrid decoder [25].

Several papers and books in the area of error control coding discuss the structure of hardware RS decoders based on algebraic decoding algorithm [16]-[20]. All of these designs use linear feedback shift registers. These structures are very complex, noncellular, and hence not suitable for VLSI implementation. Moreover, they can only be used for a specific RS code and can not be easily generalized for designing a universal RS decoder.

Reed-Solomon decoders can also be implemented using general or special purpose processors [9],[22],[43],[50]. The first special purpose computer suitable for RS decoding, called GF1 processor, was presented by Berlekamp in 1979 [21]. Cohen has studied software RS decoders using general purpose microprocessors and GF1 processor [22]. He has shown that for software decoders the algebraic decoding is faster than other decoding algorithms for high rate codes. Then, by using the algebraic decoding algorithm, he has discussed implementation of RS decoders based on general purpose, bit-slice and special purpose microprocessors (GF1). He has shown that using GF1 processor and algebraic decoding one can have a worst case throughput of 5 Mb/s for RS(255, 239) code, while, in the case of general purpose microprocessors such as Intel 8086 and bit-slice microprocessors the achievable throughputs are 15 and 200 Kb/s, respectively. The proposed special purpose microprocessor was useful in designing high rate RS decoders, but for low rate decoders the speed decreases rapidly. Because large numbers of chips are required, decoders based on this processor are large, high weight, high consumption and unreliable. Moreover, for each RS code a separate software should be developed and for development and test of the software, special development tools are required.

To solve the above problems, VLSI implementation of RS decoders have been considered. Recently, two pipeline VLSI Reed-Solomon decoders based on algebraic decoding algorithm have been introduced [24],[59]. These structures are suitable for the design of individual RS decoders with fixed error correction capability and block length. They can be built on VLSI chips since they have modular and cellular structures. These structures use many kinds of cells for different steps of the algebraic decoding algorithm. The number of cells for each step of the algorithm depends on the error correction capability of the code. Because of these reasons, they can not be used in the design of a universal decoder.

Reed-Solomon codes can also be decoded using the transform decoding algorithm [25]. This algorithm is performed completely in the frequency domain, i. e., even the error values are evaluated in the frequency domain. At the end of this algorithm an inverse Fourier transform is needed to convert the corrected data into time domain information.

Several digital signal processing techniques can be used in error control coding [25]-[28]. In particular, we can apply many available Fast-Fourier-Transform (FFT) algorithms in digital signal processing to error control coding. These FFT algorithms reduce the decoding time of software decoders. As an example, fast syndrome calculation can be done using the Chinese Remainder Theorem and Winograd's algorithm [26]-[28]. But the problem with FFT techniques is that most of them can be used for certain block length and error correcting capabilities and therefore are not useful for universal RS decoders.

Recently, some attempts have been made to modify transform decoding algorithm and present structures suitable for hardware decoders using VLSI technology. Two VLSI architectures have been introduced both using transform decoding algorithm. The first design [29], uses Linear-Feedback-

Shift-Register (LFSR) synthesis circuit for finding the error locator polynomial. In [29], two kinds of VLSI chips are designed as the basic building blocks for RS decoders. These basic blocks are not the same for different finite fields. An RS(255, 223) decoder can be made using 40 VLSI chips and 100 Small-Scale-Integration (SSI), Medium-Scale-Integration (MSI) and memory chips. This amount is approximately 10% of chip count compared to their counterparts implemented using SSI and MSI chips. In [30], a pipeline structure of a transform RS decoder similar to a systolic array is developed. This structure is very modular and cellular and hence it is more suitable for VLSI design compared to the structure of [29]. This structure is very similar to the pipeline structure based on algebraic decoding. This systolic structure is also only good for designing individual RS decoders with fixed block length and error correction capability. However, it is not suitable for design of universal RS decoders.

In [31], a universal transform decoder is proposed. This hardware decoder can decode a wide range of RS codes; the symbol fields range from 4 to 8 bits, the code lengths range from 15 to 255 symbols, and the rates are programmable downward to a minimum of one-half. As an example the (255,127) RS code can be decoded by this decoder with throughput of 4 Mb/s. This decoding speed is much higher than the speed of the software universal decoder based on GF1 processor [22]. The difficulty with this decoder is that it is very complex and the design time of this decoder is very high. The most difficult part of the design is the control unit of the system which is complex because of the capability of the decoder for decoding wide range of codes. Moreover, this universal decoder is non-systematic, hence it loses the entire information block when the errors in the channel are higher than its error correction capability.

In 1984 Blahut [32] introduced fully time-domain decoding algorithms. These algorithms are equivalents of algebraic and transform decoding algorithms in time domain. In these algorithms no Fourier transform (syndrome computation) or inverse Fourier transform are needed and the algorithms work directly on the raw input data. This is part of the reason why Blahut felt that these algorithms are good candidates for universal decoders. In this thesis, we will introduce a few structures based on these algorithms.

1.3. Existing Multipliers and Inverters in Galois Fields

Reed-Solomon codes utilize the finite field of 2^m elements. This finite field (Galois field) is denoted by GF(2^m) and the size of each element of the field is m bits. The encoding and decoding of an RS code require algebraic operations in the chosen field GF(2^m). The operations of addition, subtraction, multiplication and division in GF(2^m) are quite different from the usual binary arithmetic operations. Addition and subtraction in GF(2^m) are bit-wise exclusive-or operations. Thus, it is easier than the usual binary addition and subtraction. Although arithmetic in GF(2^m) does not involve carries, multiplication and division are more complex and difficult than binary integer multiplication and division.

Several methods have been proposed to realize multiplication and division in finite fields. Some of these methods use look-up tables [16],[22],[33],[45] which are used in software decoding. Inversion can also be done in software decoding by the use of look-up table techniques and division is implemented by an inversion followed by a multiplication. Multiplication and inversion in finite fields can be also done using hardware circuitry. Most of the circuits proposed for multipliers have serial-in/serial-out structures [14],[16],[34],[35],[46]. These kind of multipliers are mainly used in data encryption.

For hardware Reed-Solomon decoders, parallel-in/parallel-out multipliers are required. Many such multiplier structures are available [36]-[41]. Some of these structures are cellular, but complex [36]-[39] and some others are noncellular [40].

Galois field elements can be represented with respect to a standard basis or a normal basis. For each of these bases, there are many polynomials which can generate Galois fields and the complexity of the multipliers depend on the choice of basis and polynomial [42],[44]. In this thesis, the multiplier structures are studied, compared, and new structures are introduced. Universal multipliers are also studied since they are used in the design of universal decoders. The universal multiplier is capable of multiplying two elements of Galois fields in GF(2^m) for m =4,5,6,7,8.

Inverter structures are very complex [41], and hence many efforts [30],[47] have been made to avoid inversion in the decoding algorithms. But the existing algorithms still suffer from at least one inversion circuitry. Therefore new inverter structures should be developed specially for universal inverters.

1.4. Thesis Outline

As discussed before, to design an RS decoder with specified requirements, a long design time is needed and the outcome is a complex decoder. This results in a large, high weight, high power consumption and unreliable RS decoder. VLSI implementation of RS decoders can be considered to solve these problems. VLSI design is time consuming and very expensive. To decrease the design time, cellular structures for RS decoders should be introduced which uses the same cell many times. To share the expense and the development time a versatile decoder on a VLSI chip can be considered. A versatile RS decoder is able to correct any number of errors and erasures for a wide range of RS codes.

The main objective of this research is to search for structures of versatile Reed-Solomon decoders suitable for VLSI design. The versatile decoder should have a cellular structure consisting of simple cells.

In Chapter 2, after a brief review of the characteristics of Reed-Solomon codes, RS decoding algorithms are presented. These algorithms are classified into two main categories: i) syndrome-based algorithms which can be further divided into algebraic decoding and transform decoding algorithms, ii) the time-domain decoding algorithms which are time-domain equivalent of syndrome-based algorithms. These two categories of algorithms are compared from the structural point of view and it is shown that time-domain algorithms are the best candidates for designing versatile hardware decoders.

RS decoders require computations in Galois fields such as parallel-in/parallel-out multipliers and multiplicative inverters. The complexity of an RS decoder is also dependent on the complexity of arithmetic functions in Galois fields. In Chapter 3, the arithmetics in these finite fields are discussed. Two main representation of the field elements, standard and normal basis, are explained. A new normal basis multiplier is introduced which has less complexity compared to Massey-Omura multiplier. The least complex multipliers in both bases are determined and it is shown that the standard basis multiplier has less complexity compared to normal basis multipliers. In normal basis, a low complexity structure is available for the inverter, but in standard basis there is no such an inverter. A universal multiplier capable of multiplying two elements of GF(2^m) for $m=4,5,6,7,8$ is also introduced in standard basis. Note that there is no low complexity universal inverter available in non of the two bases.

As discussed in Chapter 2, time-domain algorithms are suitable for the design of versatile hardware decoders. In Chapter 4, time-domain decoding algorithm based on transform decoder is considered and a versatile decoder structure for correcting errors and erasures is introduced. This decoder can decode RS codes with variable error correction capability defined over a fixed Galois field. It is shown that hardware versatile decoders based on syndrome-based algorithms require more circuitry and is more complex compared to the structure introduced in this chapter.

In Chapter 5, the time-domain decoding algorithm based on transform decoder is restructured in order to make it suitable for introducing cellular decoder structures. A cellular decoder is introduced to decode RS codes defined in a fixed Galois field having various error correction capabilities. The introduced decoder can be designed using n identical cells where n is the code length. The simple design of this cell and the control circuitry make the introduced decoder structure attractive for VLSI implementation.

Since universal inverters are fairly complex, it is desired to reduce the number of inversions used in RS decoders. In Chapter 6, a technique is introduced to reduce the number of inversions in the time-domain decoding algorithm based on algebraic decoder. The modified algorithm is used in the universal RS decoder to decrease the complexity of the universal decoder. This universal decoder can decode any RS code in GF(2^m), for m =4,5,6,7,8. In this chapter, a structure is also introduced to decode RS codes generated by any generator polynomial.

For completeness of the thesis, software RS decoders are considered in Chapter 7. Because of the structure of the time domain algorithms, the software for versatile decoders can be easily written using these algorithms. However, this point is not important since even a badly structured algorithm can be written in software with a good structure. In this chapter, the decoding algorithms are compared and it is shown that time-domain algorithms are slower than syndrome-based algorithms for software RS decoders. A method is also introduced for fast software decoding of high rate RS codes.

Chapter 8 concludes the thesis with a summary of results and suggestions for further research.

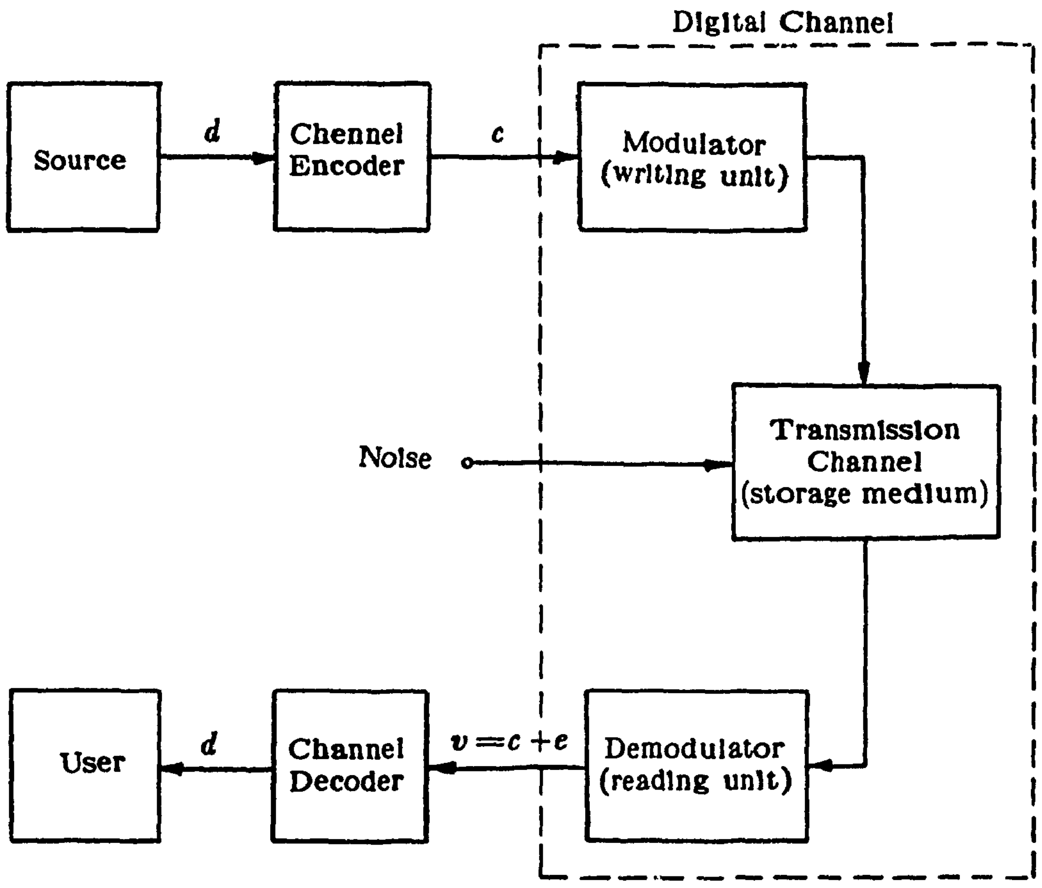


Figure 1.1: Block Diagram of a Data Transmission System

(or a Data Storage System)

CHAPTER TWO

REED-SOLOMON CODES AND DECODING ALGORITHMS

In this chapter, first the structure of Reed-Solomon codes and their properties are described. Then syndrome-based RS decoding algorithms and their time-domain equivalents are explained. Finally the decoding algorithms are compared from structural point of view.

2.1. Reed-Solomon Codes

2.1.1. Description

Reed-Solomon codes are a special subclass of generalized BCH codes. The BCH codes form a large class of powerful cyclic codes. Binary BCH codes were discovered by Hocquenghem in 1959 [51] and independently by Bose and Chaudhuri in 1960 [52]. Generalization of the binary BCH codes to codes in p^m symbols (where p is a prime) was obtained by Gorenstein and Zierler in 1961 [53]. The RS codes were introduced by Reed and Solomon in 1960 [1] independently of the works by Hocquenghem, Bose, and Chaudhuri.

For any choice of positive integers s and t, there exists a q-ary BCH code of length $n = q^s - 1$ ($t \leq n$), which is capable of correcting any combination of t or fewer symbol errors (q is a prime power). Let α be a primitive element in the field GF(q^s). The generator polynomial of a t-error-correcting BCH code is the polynomial of lowest degree with coefficients from GF(q) for which $\alpha, \alpha^2, \cdots, \alpha^{2t}$ are the roots. Let $\phi_i(x)$ denote the minimal polynomial of

α^i, and note that the degree of this polynomial is equal to or smaller than s. The generator polynomial of the q-ary BCH code is the Least-Common-Multiple (LCM) of $\phi_1(x), \phi_2(x), \cdots, \phi_{2t}(x)$. The degree of the generator polynomial is at most $2st$, and the number of parity-check digits of the code is no more than $2st$.

An important subclass of q-ary BCH codes is called Reed-Solomon codes which have capability of correcting both burst and random errors. RS codes can be formed from q-ary BCH codes when $s=1$. The practical codes in the class of RS codes are constructed over alphabet sizes which are powers of 2 ($q=2^m$) and the most commonly used codes have symbol sizes of $m=4, 5, 6, 7, 8$†.

An (n,k) primitive RS code with symbols from GF(2^m) has code word length of $n=2^m-1$ where m is the element size of the field. The ratio, k/n, is called the rate of the RS code. This code has minimum distance of $n-k+1$ and the number of parity-check symbols is $n-k$. The generator polynomial $\hat{g}(x)$ of the code is

$$\hat{g}(x)=\prod_{i=0}^{n-k-1}(x-\alpha^{h+i}), \tag{2.1}$$

where h is an integer constant. By choosing different values for the constant h, n different RS codes can be formed. The constant h is usually, but not always, taken to be 1. In that case, the generator polynomial $g(x)$ is

$$\begin{aligned} g(x) &= \prod_{i=0}^{n-k-1}(x-\alpha^{1+i}) \\ &= g_0+g_1x+\cdots+g_{n-k}x^{n-k}, \end{aligned} \tag{2.2}$$

and the code is called a narrow sense RS code.

† Henceforth we restrict our attention to GF(2^m).

2.1.2. Code Word Generation

Let $\underline{d}=(d_0, d_1, \cdots, d_{k-1})$ be the message vector to be encoded, then the message polynomial can be defined as, $d(x)=d_0+d_1x+\ldots+d_{k-1}x^{k-1}$. Any code word polynomial $c(x)$ can be formed as,

$$c(x)=d(x)g(x)=c_0+c_1x+\cdots+c_{n-1}x^{n-1}, \tag{2.3}$$

and the corresponding code word is $\underline{c}=(c_0, c_1, \cdots, c_{n-1})$. This code is non-systematic because k message symbols are not explicitly present in the code word. Hence, in the decoding process one extra step is needed to extract the information from the corrected code word.

The generation of RS codes by Eq. (2.3) is the time-domain generation of these codes. We can also generate the RS codes in the frequency domain. To explain the encoding process in frequency domain, let consider a valid code word $\underline{c}$ defined in the time domain and its corresponding frequency-domain vector $\underline{C}=(C_0, \cdots, C_{n-1})$ where

$$C_j=\sum_{i=0}^{n-1}\alpha^{ij}c_i, \qquad j=0, \cdots, n-1. \tag{2.4}$$

From Eqs. (2.3),(2.4), the components of $\underline{C}$ are

$$C_j=c(\alpha^j)=d(\alpha^j)g(\alpha^j), \quad j=0,\cdots,n-1. \tag{2.5}$$

Considering the fact that $g(\alpha^j)=0$ for $j=1,2,\cdots,n-k$ we have

$$C_j=0 \ , \qquad j=1,2,\cdots,n-k. \tag{2.6}$$

Therefore the RS code with minimum distance $n-k+1$ in GF(2^m) is the set of all vectors $\underline{c}$ whose frequency-domain components satisfy Eq. (2.6).

To encode in frequency domain first C_j for $j=1, \cdots, n-k$ are set to zero and the remaining k components of $\underline{C}$ to the k information symbols. An inverse Fourier transform shown in Eq. (2.7) gives the code word $\underline{c}$ in time domain.

$$c_i = \frac{1}{n} \sum_{j=0}^{n-1} \alpha^{-ij} C_j, \qquad i=0, \cdots, n-1. \tag{2.7}$$

It is also possible to generate the code words in systematic form. In systematic generation, the rightmost k symbols of each code word are the unaltered message symbols and the leftmost $n-k$ symbols are parity-check symbols. To generate the code words in systematic form, following three steps should be performed [18]:

Step 1: Premultiply the message polynomial $d(x)$ by x^{n-k}.

Step 2: Obtain the remainder $p(x)=p_0+p_1x+ \cdots +p_{n-k-1}x^{n-k-1}$ (the parity-check polynomial) from dividing $x^{n-k} d(x)$ by the generator polynomial $g(x)$.

Step 3: Combine $p(x)$ and $x^{n-k} d(x)$ to obtain the code word polynomial $p(x)+x^{n-k} d(x)$. This means that the code word vector is,

$$\underline{c} = (c_0, c_1, \cdots, c_{n-1}) = (p_0, p_1, \cdots, p_{n-k-1}, d_0, d_1, \cdots, d_{k-1}). \tag{2.8}$$

For hardware implementation of the systematic RS encoder, a feedback-shift-register division circuit can be used [16]. Recently, Berlekamp [14] has shown that for any RS code there is a specific constant h in Eq. (2.1) such that the generator polynomial becomes reciprocal. Using this property, he has given the structure of a systematic encoder which has a very low complexity. This is due to the fact that for a reciprocal generator polynomial just half of the coefficients need to be stored. Using this concept an RS(255, 223) encoder has been realized on a single VLSI chip [15].

2.1.3. Properties

For channels which are disturbed by burst errors, non-binary BCH codes are among the good choices. There is a distinct advantage to character oriented codes (like non-binary BCH codes) in terms of burst error correction capability compared to binary codes. Reed-Solomon codes are the most important subclass of non-binary BCH codes. They have a very large burst error correcting capability and an optimum structure.

An (n,k) linear code has minimum distance $d \leq n-k+1$ [16]. Codes with $d=n-k+1$ are called maximum distance separable (MDS) codes. Reed-Solomon codes are the most important subclass of MDS codes and have error correction capability of $t = \lfloor (n-k)/2 \rfloor$†.

Block codes such as RS codes, can use one bit of soft-decision information per symbol. This information, called an erasure indicator, is an indication that the received symbol is incorrect. In fact an erasure is an error with known location. An RS(n,k) code is able to correct up to $n-k$ erased symbols. Any pattern of ν symbol errors and ρ symbol erasures can be corrected provided that $2\nu+\rho \leq n-k$. To correct each error, the decoder must find both its location and its value, to correct each erasure, only the value must be found.

2.2. Decoding Reed-Solomon Codes Based on Syndrome

Let $c(x)$ be a code word polynomial of RS(n,k) code generated by the generator polynomial $g(x)$ and let $e(x)$ be an error polynomial

$$e(x)=e_0+e_1x+\cdots+e_{n-1}x^{n-1}. \tag{2.9}$$

† $\lfloor x \rfloor$ denotes the integer part of x.

The received polynomial at the input of decoder is

$$v(x)=c(x)+e(x)=v_0+v_1x+\cdots+v_{n-1}x^{n-1}, \tag{2.10}$$

where the polynomial coefficients are components of the received vector $\underline{v}$. We can evaluate this polynomial at roots of $g(x)$, which are $\alpha, \alpha^2, \cdots, \alpha^{2t}$. Since $c(\alpha^j)=0$ for $j=1,2,\cdots,2t$,

$$v(\alpha^j)=e(\alpha^j)=\sum_{i=0}^{n-1} e_i\alpha^{ij}, \qquad j=1,2,\cdots,2t. \tag{2.11}$$

This final set of $2t$ equations involves only the components of the error pattern, not those of the code word. Therefore they are the syndromes for evaluating the error patterns [25] and,

$$S_j=v(\alpha^j)=\sum_{i=0}^{n-1} v_i\alpha^{ij}, \qquad j=1,2,\cdots 2t. \tag{2.12}$$

After evaluation of syndromes, the error pattern e_i, $i=0,\cdots,n-1$, can be determined.

Now, suppose that ν errors actually occur, $0<\nu\leq t$, and that they occur in unknown locations $i_1, i_2, \cdots, i_\nu$. The error polynomial can be written as

$$e(x)=e_{i_1}x^{i_1}+e_{i_2}x^{i_2}+\cdots+e_{i_\nu}x^{i_\nu}, \tag{2.13}$$

where e_{i_l} is the value of the lth error. We do not know $i_1, \cdots, i_\nu$, nor do we know $e_{i_1}, \cdots, e_{i_\nu}$. In fact we do not even know the value of ν. These must be computed in order to correct the errors.

The first step in decoding is the evaluation of syndromes given by Eq. (2.12). Let us evaluate the received polynomial at α to obtain the syndrome S_1:

$$S_1=v(\alpha)=c(\alpha)+e(\alpha)=e(\alpha)=e_{i_1}\alpha^{i_1}+e_{i_2}\alpha^{i_2}+\cdots+e_{i_\nu}\alpha^{i_\nu}. \tag{2.14}$$

To change the notation [25], we define the error values $Y_l = e_{i_l}$ for $l=1, \cdots, \nu$, and the error location numbers $X_l = \alpha^{i_l}$ for $l=1, \cdots, \nu$, where i_l is the actual location of the lth error and X_l is the field element associated with this location. With this notation, the first syndrome is given by

$$S_1 = Y_1 X_1 + Y_2 X_2 + \cdots + Y_\nu X_\nu. \tag{2.15}$$

Similarly, we can evaluate the received polynomial at each of the powers of α used to define $g(x)$. Then we have the following set of $2t$ simultaneous equations in the ν unknown error location numbers $X_1, \cdots, X_\nu$ and the ν unknown error values $Y_1, \cdots, Y_\nu$:

$$S_j = Y_1 X_1^j + Y_2 X_2^j + \cdots Y_\nu X_\nu^j \qquad j=1,2, \cdots, 2t. \tag{2.16}$$

This set of equations must have at least one solution because of the way in which the syndromes are defined. It can be shown that the solution is unique [25].

Now we can summarize the decoding process in two following steps:

1) Evaluation of syndromes using the received vector Eq. (2.12).

2) Solving the system of $2t$ nonlinear equations given in Eq. (2.16) to find the error locations and error values.

The first step is in fact a Fourier transform evaluation, which finds $2t$ frequency-domain components of the received vector $\underline{v}$ according to Eq. (2.12).

In the second step, the direct solution of a system of nonlinear equations is too difficult, except for small $2t$ [9],[16],[58],[70]. A better approach requires some intermediate steps. The error-locator polynomial is introduced as,

$$\Lambda(x) = \Lambda_\nu x^\nu + \Lambda_{\nu-1} x^{\nu-1} + \cdots + \Lambda_1 x + 1. \tag{2.17}$$

This polynomial is defined with roots as the inverse error location numbers X_l^{-1} for $l=1, \cdots, \nu$. The error location numbers X_l indicate errors at location i_l for $l=1, \cdots, \nu$. That is,

$$\Lambda(x)=\prod_{l=1}^{\nu}(1-xX_l) \quad , \quad X_l=\alpha^{i_l} \tag{2.18}$$

The methods for finding the error locator polynomial from the syndromes, are explained later in this chapter. After obtaining the coefficients of the error-locator polynomial, the error locations and error values can be found as shown in the following subsections.

2.2.1. Algebraic Decoding

One way to decode RS codes is the algebraic decoding. In this algorithm, first the syndromes are evaluated using Eq. (2.12) Then, based on the syndromes the error-locator polynomial $\Lambda(x)$ is found. This polynomial has roots at the inverse error location numbers X_l^{-1} for $l=1, \cdots, \nu$.

The algebraic decoding continues with using the Chien search to evaluate the roots of $\Lambda(x)$ in order to find the error locations. Chien search method simply consists of the computation of $\Lambda(\alpha^j)$ for $j=0, \cdots, n-1$ and checking for zero. Use of this trial and error method is feasible, since the number of elements in a Galois field is finite [17].

After evaluation of roots of $\Lambda(x)$ using Chien search, the error location numbers $X_l=\alpha^{i_l}$ can be substituted in Eq. (2.16). Now, this system of $2t$ equations becomes linear and can be solved for the error values Y_j for $j=1, \cdots, \nu$ [53]. To find Y_j, one matrix inversion is required for each of the errors. This method is not efficient because of ν matrix inversions involved and, therefore, other methods should be used for finding the error values.

A more efficient way to find the error values is called the Forney algorithm [56]. This algorithm is derived starting with the error-locator polynomial $\Lambda(x)$. Define the syndrome polynomial

$$S(x)=\sum_{j=1}^{2t} S_j x^j = \sum_{j=1}^{2t}\sum_{i=1}^{\nu} Y_i X_i^j x^j, \tag{2.19}$$

and define the error-evaluator polynomial $\Omega(x)$ in terms of these known polynomials by the key equation,

$$\Omega(x)=[1+S(x)]\Lambda(x) \quad (mod \quad x^{2t+1}). \tag{2.20}$$

The error-evaluator polynomial can be related to the error-locations and error values as [56],

$$\Omega(X_l^{-1})=Y_l \prod_{i\neq l}(1-X_i X_l^{-1}), \tag{2.21}$$

and the error values are given by [25],

$$Y_l=-X_l \frac{\Omega(X_l^{-1})}{\Lambda'(X_l^{-1})} \quad , \quad l=1,\ldots,\nu, \tag{2.22}$$

where $\Lambda'(x)$ is the formal derivative of $\Lambda(x)$. The above equation defines the Forney algorithm which is a considerable improvement over matrix inversion but requires division. An important difference between matrix inversion and the Forney algorithm is that in the former, the Chien search must be completed before error evaluation can start. But in Forney algorithm, as soon as one X_l is found, the corresponding Y_l can be evaluated using Eq. (2.22).

After evaluation of the error values, the received vector can be corrected by subtracting the error values from components of the received vector.

In the algebraic decoding algorithm, we notice that first the received vector is transformed to frequency domain by evaluation of syndromes, and then, based on the syndromes, the error locations and error values are found in time domain. Because of the mixture of frequency and time domain techniques, this algorithm is sometimes called hybrid decoding algorithm [25].

2.2.2. Transform Decoding

To explain the transform decoding algorithm [25],[57], let's assume that the code word $\underline{c}$ is transmitted and the channel makes errors described by the vector $\underline{e}$. The syndromes of this noisy code word $\underline{v}=\underline{c}+\underline{e}$ are defined by the set of $2t$ equations given in Eq. (2.12). Obviously using the definition of Fourier transform, the syndromes are computed as $2t$ components of a Fourier transform. The received noisy code word has a Fourier transform given by $V_j=C_j+E_j$ for $j=0, \cdots, n-1$, and the syndromes are the $2t$ components of this spectrum from 1 to $2t$. But by construction of a Reed-Solomon code,

$$C_j=0 \quad j=1, \cdots, 2t; \tag{2.23}$$

hence,

$$S_j=V_j=E_j \quad j=1, \cdots, 2t. \tag{2.24}$$

The block of syndromes gives us a window through which we can look at $2t$ of the n components of the Fourier transform of error pattern. The decoder must find the entire transform of the error pattern given a segment of length $2t$ of that transform and the additional information that at most t components of the time-domain error pattern are nonzero. After evaluation of the frequency-domain error vector $\underline{E}$, the decoding can be completed by an inverse Fourier transform to find $\underline{e}$ and subtracting from $\underline{v}$ [25].

To find the rest of the components of $\underline{E}$, consider the vector $\underline{\Lambda}$ of length n whose components are coefficients of the polynomial $\Lambda(x)$. The vector $\underline{\Lambda}$ has an inverse transform $\underline{\lambda}=(\lambda_0, \cdots, \lambda_{n-1})$ where,

$$\lambda_i=\sum_{j=0}^{n-1}\Lambda_j\,\alpha^{-ij}=\Lambda(\alpha^{-i}). \tag{2.25}$$

Therefore, using Eq. (2.18),

$$\lambda_i = \prod_{l=1}^{\nu} (1-\alpha^{-i}\alpha^{i_l}), \tag{2.26}$$

which is zero if $i = i_l$, where i_l for $l=1, \cdots, \nu$ index the error locations; and λ_i is nonzero otherwise. Hence, $\lambda_i = 0$ if and only if $e_i \neq 0$. That is, in the time domain, $\lambda_i e_i = 0$; therefore, the convolution in the frequency domain is zero,

$$\underline{\Lambda} * \underline{E} = \sum_{j=0}^{n-1} \Lambda_j E_{i-j} = 0 \quad , \quad i = 0, \cdots, n-1. \tag{2.27}$$

Now we assume that the coefficients of the error locator polynomial is known, and hence this convolution can be considered as a set of n equations in k unknown components of $\underline{E}$. The methods for evaluation of the error-locator polynomial $\Lambda(x)$ will be explained in Section 2.3.

The remaining components of $\underline{E}$ can be obtained by iterative extension [25], that is, they can be sequentially computed from $\underline{\Lambda}$ using the convolution of Eq. (2.27) written in the form

$$E_i = -\sum_{j=1}^{n-1} \Lambda_j E_{i-j}, \qquad i = 0, \cdots, n-1. \tag{2.28}$$

and known frequency domain error pattern Eq. (2.24). In this way all components of the vector $\underline{E}$ are computed. An inverse transform on $\underline{E}$ results in the time-domain error vector $\underline{e}$ and error correction can be completed by $\underline{c} = \underline{v} - \underline{e}$.

In the transform decoding algorithm, first the received vector is transformed to frequency domain by evaluation of syndromes. Then the frequency-domain error vector, is evaluated, and, finally, an inverse Fourier transform is performed to find the time-domain error vector. Therefore, all the effort for finding the error values is performed in the frequency domain. That is the reason for calling this algorithm the frequency-domain decoding algorithm [25].

2.3. Evaluation of Error-Locator Polynomial

In this section, we explain the procedures for evaluation of the coefficients of the error locator polynomial $\Lambda(x)$ given by Eq. (2.17). This polynomial has ν roots at X_l^{-1} for $l=1, \cdots \nu$. The error location numbers X_l, indicate errors at location i_l for $l=1, \cdots, \nu$.

To find the coefficients of $\Lambda(x)$, multiply both sides of Eq. (2.17) by $Y_l X_l^{j+\nu}$ and set $x=X_l^{-1}$ [25]. Then the left side is zero and we have

$$Y_l(X_l^{j+\nu}+\Lambda_1 X_l^{j+\nu-1}+\cdots+\Lambda_\nu X_l^j)=0. \tag{2.29}$$

Such an equation holds for each l and each j. Sum up these equations from $l=1$ to $l=\nu$. This gives for each j,

$$\sum_{l=1}^{\nu} Y_l X_l^{j+\nu}+\Lambda_1 \sum_{l=1}^{\nu} Y_l X_l^{j+\nu-1}+\cdots+\Lambda_\nu \sum_{l=1}^{\nu} Y_l X_l^{j}=0. \tag{2.30}$$

The individual sums are recognized as syndromes, and thus the equation becomes

$$\Lambda_1 S_{j+\nu-1}+\Lambda_2 S_{j+\nu-2}+\cdots+\Lambda_\nu S_j=-S_{j+\nu}, \quad j=1, \cdots, \nu. \tag{2.31}$$

This is the set of linear equations relating the syndromes to the coefficients of $\Lambda(x)$ [18],[25]. These equations can be written in a matrix form as,

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_\nu \\ S_2 & S_3 & \cdots & S_{\nu+1} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ S_\nu & S_{\nu+1} & \cdots & S_{2\nu-1} \end{bmatrix} \begin{bmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \cdot \\ \cdot \\ \cdot \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \cdot \\ \cdot \\ \cdot \\ -S_{2\nu} \end{bmatrix}. \tag{2.32}$$

Note that Eq. (2.32) is a special case of Eq. (2.27). This system of equations can be solved by inverting the matrix provided that the matrix is nonsingular. However, first the correct value of ν should be found as follows. As a trial value, set ν to the error correction capability of the code t and compute the determinant of the matrix in Eq. (2.32). If it is nonzero it can be shown that this is the correct value of ν [25]. Otherwise, if the determinant is zero, reduce the trial value of ν by 1 and repeat. Continue in this way until a nonzero determinant is obtained. Now, the actual number of errors is known. Then, evaluate the coefficients of $\Lambda(x)$ using the value of ν and Eq. (2.32).

This method is very inefficient specially for the codes with high error correction capability (high t). To solve this problem three methods will be discussed in the following subsections.

2.3.1. Berlekamp-Massey Algorithm

Berlekamp-Massey algorithm [16],[19] relies on the fact that the matrix equation of Eq. (2.32) is not arbitrary in its form; rather, the matrix is highly structured. This structure is used to advantage to obtain the polynomial $\Lambda(x)$ by a method that is conceptually more complicated but computationally much simpler.

Suppose that the polynomial $\Lambda(x)$ is known. Then the first row of the above matrix equation defines $S_{\nu+1}$ in terms of $S_1, \cdots, S_\nu$. The second row defines $S_{\nu+2}$ in terms of $S_2, \cdots, S_{\nu+1}$, and so forth. This sequential process is summarized by the equation

$$S_j = -\sum_{i=1}^{\nu} \Lambda_i S_{j-i} \qquad j = \nu+1, \cdots, 2\nu. \tag{2.33}$$

For fixed $\Lambda(x)$, this is the equation of an autoregressive filter. It may be implemented as a linear-feedback shift register with taps given by the coefficients of $\Lambda(x)$.

Now, the problem is the design of the linear-feedback shift register that will generate the known sequence of syndromes. Many such shift registers exist, however, we wish to find the shortest linear-feedback shift register with this property. The shortest shift register gives the least-weight error pattern that will explain the received data, that is, $\Lambda(x)$ of the smallest degree. The polynomial of the smallest degree will have degree ν, and it is unique, since the $\nu \times \nu$ matrix of the original problem is invertible.

The Berlekamp-Massey algorithm is explained using the following set of recursive equations computing $\Lambda^{(2t)}(x)$:

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S_{r-j}, \tag{2.34}$$

$$L_r = \delta(r - L_{r-1}) + (1-\delta) L_{r-1}, \tag{2.35}$$

$$\begin{bmatrix} \Lambda^{(r)}(x) \\ B^{(r)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1}\delta & (1-\delta)x \end{bmatrix} \begin{bmatrix} \Lambda^{(r-1)}(x) \\ B^{(r-1)}(x) \end{bmatrix}, \tag{2.36}$$

for $r = 1, \cdots, 2t$. The initial conditions are $\Lambda^{(0)}(x) = 1$, $B^{(0)}(x) = 1, L_0 = 0$, and $\delta = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r-1$, and $\delta = 0$ otherwise. Then, $\Lambda^{(2t)}(x)$ is the smallest-degree polynomial with the properties that $\Lambda_0^{(2t)} = 1$ and,

$$S_r + \sum_{j=1}^{n-1} \Lambda_j^{(2t)} S_{r-j} = 0, \quad r = L_{2t+1}, \cdots, 2t. \tag{2.37}$$

It is also possible to find the error-evaluator polynomial $\Omega(x)$ of Eq. (2.20). To find $\Omega(x)$ we need to combine Eq. (2.20) to the above recursion to get,

$$\begin{bmatrix} \Omega^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1}\delta & (1-\delta)x \end{bmatrix} \begin{bmatrix} \Omega^{(r-1)}(x) \\ A^{(r-1)}(x) \end{bmatrix}. \tag{2.38}$$

The initial conditions are $\Omega^{(0)}(x) = 1$, and $A^{(0)} = 0$. After $2t$ iterations, $\Omega^{(2t)}(x)$ is the error evaluator polynomial. Note that in many references [16],[25], the algorithm is called "Berlekamp's algorithm" when both error polynomials are calculated. In this thesis we will continue to use this convention.

The Berlekamp-Massey algorithm is modified in [48] for speeding up the decoder. This method provides a saving in decoding time, if the low-weight errors are dominant errors.

2.3.2. Euclidean Algorithm

In 1975, a new method for evaluation of the error locator as well as error evaluator polynomial was introduced [54]. This method uses the Euclidean algorithm, a method for finding the Greatest-Common-Divisor (GCD) of two polynomials.

The Euclidean algorithm is explained using the following set of recursive equations:

$$Q^{(r)}(x)=\left\lfloor \frac{R^{(r)}(x)}{T^{(r)}(x)} \right\rfloor, \tag{2.39}$$

$$A^{(r+1)}(x)=\begin{bmatrix} 0 & 1 \\ 1 & Q^{(r)}(x) \end{bmatrix} A^{(r)}(x), \tag{2.40}$$

$$\begin{bmatrix} R^{(r+1)}(x) \\ T^{(r+1)}(x) \end{bmatrix}=\begin{bmatrix} 0 & 1 \\ 1 & Q^{(r)}(x) \end{bmatrix}=\begin{bmatrix} R^{(r)}(x) \\ T^{(r)}(x) \end{bmatrix}. \tag{2.41}$$

In Eq. (2.39), $\lfloor\ \rfloor$ means the quotient of the division. The initial conditions are $R^{(0)}(x)=x^{2t}$, $T^{(0)}(x)=\sum_{j=1}^{2t} S_j x^{j-1}$ and $A^{(0)}(x)=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The algorithm stops when degree of $T^{(r)}(x)$ is less than t. Then

$$\hat{\Omega}(x)=\Delta^{-1}T^{(r')}(x) \tag{2.42a}$$

$$\Lambda(x)=\Delta^{-1}A_{22}^{(r')}(x), \tag{2.42b}$$

where $\Delta=A_{22}^{(r')}(0)$, and A_{22} is the element of the matrix A, in the second column and the second row. Note that considering Eqs. (2.39)-(2.41) and $A_{22}^{(0)}(x)=1$, the value of $A_{22}^{(r')}(0)$ is nonzero in all the iterations and hence Δ is nonzero in Eq. (2.42).

In Eq. (2.42), the error locator polynomial $\Lambda(x)$ and the error evaluator polynomial $\hat{\Omega}(x)$ are unique solutions of Eq. (2.43), where their degrees are less than $t+1$, and t, respectively, and $\Lambda_0=1$.

$$\hat{\Omega}(x)=\sum_{j=1}^{2t} S_j x^{j-1}\Lambda(x) \quad (mod \ \ x^{2t}). \tag{2.43}$$

This algorithm is modified in [30]. The modified algorithm has a better structure for hardware implementation compared to the original algorithm. This modified algorithm is very similar to the Berlekamp-Massey algorithm.

2.3.3. Continued Fractions Algorithm

The continued fractions approximation technique which is applicable on integer numbers can be applied to find the error-locator and error-evaluator polynomials [55]. It can be shown that polynomials $\Lambda(x)$, $\Omega(x)$ and the syndromes $S_1, S_2, \cdots, S_{2t}$ are related by

$$\hat{S}(x)=S_1x^{-1}+S_2x^{-2}+\cdots+S_{2t}x^{-2t}+\cdots$$

$$=\frac{\sum_{i=0}^{t-1}\Omega_i x^i}{x^t+\sum_{k=1}^{t}\Lambda_k x^{t-k}}=\frac{\Omega(x)}{\Lambda(x)}, \tag{2.44}$$

where $\hat{S}(x)$ is a power series [55].

Let ν be the degree of $\Lambda(x)$, if only the first 2ν coefficients of $\hat{S}(x)$ are known, then $\hat{S}(x)$ and hence $\Omega(x)$ and $\Lambda(x)$ can be obtained by continued fractions operating on $S_1x^{-1}+\cdots+S_{2\nu}x^{-2\nu}$. In continued fractions method initially set

$$R^{(0)}=S_1x^{-1}+S_2x^{-2}+\cdots+S_{2\nu}x^{-2\nu} \tag{2.45}$$

and then divide 1 by $R^{(0)}$ to find $N^{(1)}$ and the remainder $R^{(1)}$ where $\frac{1}{R^{(0)}}=N^{(1)}+R^{(1)}$. Then continue in this way by dividing 1 by

$$R^{(r)}=a_{i_r}x^{-i_r}+a_{i_r+1}x^{-i_r-1}+\cdots \tag{2.46}$$

and evaluating

$$N^{(r+1)} = a_{i_r}^{-1} x^{i_r} + \cdots + b_0 \tag{2.47}$$

$$R^{(r+1)} = b_{-1} x^{-1} + \cdots \ . \tag{2.48}$$

The algorithm stops when $R^{(r)}$ is zero, $r = r'$. The values of $\Omega(x)$ and $\Lambda(x)$ are then evaluated from

$$\hat{S}(x) = \cfrac{1}{N^{(1)} + \cfrac{1}{N^{(2)} + \cfrac{1}{N^{(3)} + \cdots}}} = \frac{\Omega(x)}{\Lambda(x)}. \tag{2.49}$$

2.3.4. Comparison

Implementation of the Berlekamp-Massey algorithm is straightforward, and can be designed with shift registers [25]. The reason for having simple implementation is the structure of the algorithm which does not have any multiplication or division of two polynomials.

The hardware implementation of the Euclidean algorithm is not difficult using shift registers. But it is more difficult compared to implementation of the Berlekamp-Massey algorithm, since this algorithm has polynomial multiplication and polynomial division. The modified Euclidean algorithm [30], is very similar to the Berlekamp-Massey algorithm and can be used for hardware implementation.

Obviously the hardware implementation of continued fractions algorithm is very difficult. This is due to the polynomial inversions in this algorithm. Moreover, after evaluation of $N^{(r)}$ for $r = 1, 2, \cdots, r'$ and stopping the polynomial inversions, Eq. (2.49) should be considered for calculating $\Lambda(x)$ and $\Gamma(x)$. Evaluation of these two polynomials from Eq. (2.49) needs some polynomial multiplications and additions with a difficult control circuitry.

In this thesis, we will only consider the Berlekamp-Massey algorithm. The main reason is the simple structure of the algorithm.

2.4. Decoding Reed-Solomon Codes in Time Domain

As it was shown, in transform decoding, the algorithm starts with a Fourier transform and ends with an inverse Fourier transform. There is also a Fourier transform at the beginning of the algebraic decoding algorithm. However, instead of transforming the received word into frequency domain, it is possible to find the equivalent of the algorithm in the time domain. Using the time-domain algorithm, the Fourier transforms simply vanish.

As discussed above, the Berlekamp-Massey algorithm has a simple structure compared to other two structures, therefore, this algorithm is used in this section. Moreover, this algorithm does not have any polynomial division, and hence, transforming the algorithm in time domain is easier and results in a simple structure.

2.4.1. Algorithm Based on Transform Decoding

The transform decoding algorithm has two main steps, in addition to the Fourier transform at the beginning and the inverse transform at the end of the algorithm. These two steps are the Berlekamp-Massey algorithm to find the error locator polynomial $\Lambda(x)$ and the recursive extension to evaluate the components of the frequency-domain error pattern [25].

To find the equivalent of Berlekamp-Massey algorithm in the time domain, let $\underline{\lambda}$ and $\underline{b}$ denote the inverse Fourier transforms of the vectors $\underline{\Lambda}$ and $\underline{B}$, respectively. To express the Berlekamp-Massey equations in the time domain, simply replace the frequency-domain variables Λ_j and B_j with the time-domain variables λ_i and b_i, replace the delay operator x with α^{-i}, and replace product terms with convolution terms. The Berlekamp-Massey algorithm then transforms into a recursive procedure in the time domain, as described in the following theorem [25].

Theorem 2.1: Let $\underline{v}$ be the received noisy RS code word. Let the following set of recursive equations be used to compute $\lambda_i^{(2t)}$ for $i=0, \cdots, n-1$:

$$\Delta_r = \sum_{i=0}^{n-1} \alpha^{ir} [\lambda_i^{(r-1)} v_i], \tag{2.50}$$

$$L_r = \delta(r - L_{r-1}) + (1-\delta) L_{r-1}, \tag{2.51}$$

$$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} \\ \Delta_r^{-1}\delta & (1-\delta)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \end{bmatrix}, \tag{2.52}$$

$r=1, \cdots, 2t$; where the initial conditions are $\lambda_i^{(0)}=1, b_i^{(0)}=1$ for all i, and $\delta=1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r-1$, and otherwise $\delta=0$. Then, $\lambda_i^{(2t)}=0$ if and only if $e_i \neq 0$.

Proof: Take the Fourier transform of all vector quantities. Then the recursive equations (2.34)-(2.36) are obtained, with Eq. (2.34) having $\underline{V}$ in place of $\underline{S}$. But, $\Lambda_j^{(r-1)}=0$ for $j>2t$, and $V_j=S_j$ for $j=1, \cdots, 2t$. Therefore, the proof is complete. Q. E. D.

For RS codes we must also compute the error values. In the frequency domain, the error values can be computed using the following recursion:

$$E_j = -\sum_{i=1}^{n-1} \Lambda_i E_{j-i}, \qquad j=0,2t+1,2t+2, \cdots, n-1. \tag{2.53}$$

Once the error locator polynomial is known, this recursion can be used to extend the $2t$ known components of $\underline{E}$ to all components of $\underline{E}$, by iterating $n-2t$ times.

It is n[illegible] possible to find the Fourier transform of this equation without some restructuring. The time domain equivalent of the above equation is given in the following theorem [25].

Theorem 2.2: Let $\underline{v}=\underline{c}+\underline{e}$ be the received noisy RS code word. Given the time-domain error locator vector $\underline{\lambda}$, the following set of recursive equations

$$\Delta_r = \sum_{i=0}^{n-1} \alpha^{ir} v_i^{(r-1)} \lambda_i \tag{2.54}$$

$$v_i^{(r)} = v_i^{(r-1)} - \Delta_r \alpha^{-ri} \tag{2.55}$$

for $r=2t+1, \cdots, n$, results in

$$v_i^{(n)} = e_i \qquad i=0, \cdots, n-1. \tag{2.56}$$

Proof: The equation of the recursion in the frequency domain, Eq. (2.25), is rewritten by breaking it into two steps, starting with the received frequency-domain vector $\underline{V}$ and changing it to $\underline{E}$ one component at a time:

$$\Delta_r = V_r + \left(\sum_{j=1}^{n-1} \Lambda_j V_{r-j} \right) \tag{2.57}$$

$$V_j^{(r)} = \begin{cases} V_j^{(r-1)} & j \neq r \\ V_j^{(r-1)} - \Delta_r & j=r. \end{cases} \tag{2.58}$$

Since $V_j^{(2t)}=E_j$, for $j=1, \cdots, 2t$, and $\Lambda_j=0$, for $j>t$, the above equations are equivalent to Eq. (2.53). This equivalence proves the theorem. Q. E. D.

Now, the decoding algorithm has only two steps. In the first $2t$ iterations the error-locator vector $\underline{\lambda}$ is found. In the next $n-2t$ iterations, the error evaluator vector $\underline{e}=\underline{v}^{(n)}$ is calculated, and then error correction is performed.

2.4.2. Algorithm Based on Algebraic Decoding

It is possible to map the algebraic decoding algorithm in the time domain as well [32]. By this approach the last $n-2t$ iterations of the time-domain transform decoder will be avoided at the expense of increasing the complexity of the first $2t$ iterations.

Consider the Berlekamp's algorithm given in the previous section. This algorithm can be put in the form of time-domain equations as explained for the Berlekamp-Massey algorithm. To do this, first the Berlekamp's algorithm is expanded to compute the formal derivative of the error locator polynomial $\Lambda'(x)$ as well [32]. Note that this polynomial can be computed from $\Lambda(x)$ after the first $2t$ iterations are complete. To include $\Lambda'(x)$ as iterates, we must also introduce the temporary polynomial $B'(x)$. The iterations for the polynomial $\Lambda'(x)$ then becomes

$$\begin{bmatrix} \Lambda'^{(r)}(x) \\ B'^{(r)}(x) \end{bmatrix} = \begin{bmatrix} -\Delta_r & 1 & -\Delta_r x \\ (1-\delta) & \Delta_r^{-1}\delta & (1-\delta)x \end{bmatrix} \begin{bmatrix} B^{(r-1)}(x) \\ \Lambda'^{(r-1)}(x) \\ B'^{(r-1)}(x) \end{bmatrix}, \tag{2.59}$$

where Δ_r, L_r, and δ are as defined previously and the initial conditions are $\Lambda'^{(0)}(x) = B'^{(0)}(x) = 0$. After evaluation of the polynomials $\Lambda(x)$, $\Omega(x)$ and $\Lambda'(x)$ the Forney algorithm is used to find the error values.

To express the Berlekamp algorithm in the time domain, an approach similar to the one used in the case of the Berlekamp-Massey algorithm can be used [32]. The Berlekamp algorithm in time domain is expressed with the following set of recursive equations.

$$\Delta_r = \sum_{i=0}^{n-1} \alpha^{ir} [\lambda_i^{(r-1)} v_i], \tag{2.60}$$

$$L_r = \delta(r - L_{r-1}) + (1-\delta)L_{r-1}, \tag{2.61}$$

$$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \\ \lambda_i'^{(r)} \\ b_i'^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} & 0 & 0 \\ \Delta_r^{-1}\delta & (1-\delta)\alpha^{-i} & 0 & 0 \\ 0 & -\Delta_r & 1 & -\Delta_r \alpha^{-i} \\ 0 & (1-\delta) & \Delta_r^{-1}\delta & (1-\delta)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \\ \lambda_i'^{(r-1)} \\ b_i'^{(r-1)} \end{bmatrix}, \tag{2.62}$$

$$\begin{bmatrix} \omega_i^{(r)} \\ a_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} \\ \Delta_r^{-1}\delta & (1-\delta)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \omega_i^{(r-1)} \\ a_i^{(r-1)} \end{bmatrix}, \tag{2.63}$$

for $i=0, \cdots, n-1$ and for $r=1, 2, \cdots, 2t$. The initial conditions are $\lambda_i^{(0)}=b_i^{(0)}=\omega_i^{(0)}=1$ for all i; $\lambda_i'^{(0)}=b_i'^{(0)}=a_i^{(0)}=0$ for all i; $L_0=0$, and $\delta=1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r-1$, and $\delta=0$ otherwise. Then, $\lambda_i^{(2t)}=0$ if and only if the ith symbol of the received vector $\underline{y}$ is in error, and we have $\underline{\lambda}'=\underline{\lambda}'^{(2t)}$ and $\underline{\omega}=\underline{\omega}^{(2t)}$.

The reason for computing these vectors is the Forney algorithm, which evaluates the errata values,

$$e_i=-\alpha^i \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})}, \quad i=0, \cdots, n-1, \tag{2.64}$$

whenever α^{-i} is a root of $\Lambda(x)$. By the definition of the inverse Fourier transform we have $\omega_i=\Omega(\alpha^{-i})$, and $\lambda_i'=\Lambda'(\alpha^{-i})$. Therefore, Eq. (2.64) can be written as,

$$e_i=-\alpha^i \frac{\omega_i}{\lambda_i'}, \quad i=0, \cdots, n-1. \tag{2.65}$$

2.5. Decoding Errors and Erasures

An RS(n,k) code is capable of correcting up to $n-k$ erased symbols. Any pattern of ν symbol errors and ρ symbol erasures can be corrected provided that $2\nu+\rho \leq n-k$ [25]. To correct an error, the decoder must find both its location and its value, to correct an erasure, only the value must be found.

In this section, the decoding algorithms are modified to decode erasures as well [25],[32]. A summary of all the algorithms are also given.

2.5.1. Decoding Based on Syndrome

To modify syndrome-based algorithms to correct erasures as well, an erasure-locator polynomial $\Gamma(x)$ is formed [25],

$$\Gamma(x)=\prod_{l=1}^{\rho}(1-x\,\alpha^{i_l}), \tag{2.66}$$

where i_l, $l=1,2,\cdots,\rho$ are locations of ρ erasures. This polynomial can be combined with the error-locator polynomial $\Lambda(x)$ to form the errata-locator polynomial $\hat{\Lambda}(x)$,

$$\hat{\Lambda}(x)=\Gamma(x)\Lambda(x). \tag{2.67}$$

Roots of this polynomial indicate the locations of the errors and erasures.

To form the errata-locator polynomial, first the erasure-locator polynomial is formed based on Eq. (2.66) and then the Berlekamp's algorithm is performed for only $n-k-\rho$ iterations. In this step, the polynomials $\Lambda(x)$ and $B(x)$ are initialized by the erasure locator polynomial instead of 1 [25]. At the end, the errata-locator polynomial is obtained. We will now drop the notation $\hat{\Lambda}(x)$, replacing it with $\Lambda(x)$, which we will call the errata-locator polynomial.

Based on the above discussion, the complete algebraic decoding algorithm for errors and erasures is shown in Fig. 2.1 and summarized in 6 following steps.

Step 1: Calculate the syndromes using Eq. (2.12).

Step 2: Evaluate the erasure locator polynomial using Eq. (2.66), where i_l, $l=1,2,\cdots,\rho$ are locations of ρ erasures. This polynomial initializes the Berlekamp's algorithm explained in step 3.

Step 3: Perform the Berlekamp-Massey algorithm to obtain the errata locator polynomial. $\Lambda(x)$. Berlekamp's algorithm for decoding errors and erasures is explained using the iterative equations (2.34),(2.36),(2.68) to compute $\Lambda(x)$, for $r=\rho+1,\cdots,n-k$.

$$L_r=\delta(r-L_{r-1}-\rho)+(1-\delta)L_{r-1} \tag{2.68}$$

The initial conditions are $\Lambda^{(\rho)}(x)=B^{(\rho)}(x)=\Gamma(x)$, $L_\rho=0$, and $\delta=1$ if both $\Delta_r\neq 0$ and $2L_{r-1}\leq r-1+\rho$, and $\delta=0$ otherwise. Then $\Lambda^{(n-k)}(x)$ is the errata locator polynomial.

Step 4: Perform the Chien search to find the roots of $\Lambda(x)$. The roots of this polynomial, indicate the error locations in the received word. If $X_j^{-1}=\alpha^{-i}$ is one of the roots, then the received symbol v_i is in error or it is an erased symbol.

Step 5: Find the errata value polynomial using Eq. (2.20) and the errata values from Eq. (2.22).

Step 6: Correct the received vector $\underline{v}=(v_0, v_1, \dots, v_{n-1})$. This vector can be corrected from the knowledge of the errata locations and the errata values. The corrected code word $\underline{c}$ is found by subtracting the errata values from the received vector.

The transform decoding algorithm can also be modified to correct both errors and erasures [25]. This algorithm has six steps as shown in Fig. 2.2, which is explained as follows:

Steps 1, 2, 3: Compute syndromes, the erasure locator polynomial and the errata locator polynomial using the first three steps of the algebraic decoding algorithm given in the previous subsection.

Step 4: Calculate the remaining components of $\underline{E}=(E_0, \cdots, E_{n-1})$ by recursive extension using Eq. (2.28).

Step 5: Compute the inverse transform of E_j, $j=0, \cdots, n-1$ over $GF(2^m)$ to obtain the error vector $\underline{e}$.

Step 6: Finally, the estimate of the original code vector is obtained by subtracting the error vector $\underline{e}$ from the received vector $\underline{v}$.

2.5.2. Decoding in Time Domain

To correct errors and erasures in the time domain, the erasure locator vector $\underline{\gamma}=(\gamma_0, \cdots, \gamma_{n-1})$ can be introduced. This vector is the equivalent of the erasure locator polynomial in time domain [25].

$$\gamma_i = \prod_{l=1}^{\rho} (1-\alpha^{j_l} \alpha^{-i}), \qquad i=0,\ 1,\ \cdots,\ n-1. \tag{2.69}$$

In this equation, j_l indicates that the received noisy vector $\underline{v}$ has erasures at locations j_l for $l = 1, 2, \cdots, \rho$.

The time-domain decoding based on the transform decoding algorithm for correcting errors and erasures is shown in Fig. 2.3 and can be summarized as follows:

Step 1: In the first step of the algorithm, the erasure locator vector $\underline{\gamma}$ should be evaluated using Eq. (2.69). Evaluation of this vector can be put in the form of ρ iterations using the following recursive equation to compute $\gamma_i^{(\rho)}$:

$$\gamma_i^{(r)} = \gamma_i^{(r-1)}(1-\alpha^{j_r} \alpha^{-i}), \tag{2.70}$$

for $i=0, 1, \cdots, n-1$ and $r=1, 2, \cdots, \rho$. The initial condition is $\gamma_i^{(0)}=\alpha^0=1$ and finally $\gamma_i=\gamma_i^{(\rho)}$.

Step 2: In this step of the decoding, the time-domain errata locator vector $\underline{\lambda}=(\lambda_0, \cdots, \lambda_{n-1})$ is calculated using the time-domain Berlekamp-Massey algorithm. This step is expressed in terms of the recursive equations (2.50),(2.52),(2.68) for computing λ_i, for $i=0, \cdots, n-1$ and $r=\rho+1, \rho+2, \cdots, n-k$. The initial conditions are $\lambda_i^{(\rho)}=b_i^{(\rho)}=\gamma_i$ for all i, $L_\rho=0$, and $\delta=1$ if both $\Delta_r \neq 0$ and $2L_{r-1}-\rho \leq r-1$, and $\delta=0$ otherwise. Then $\lambda_i^{(n-k)}=0$ if and only if the ith symbol of the received vector $\underline{v}$ is in error and hence the components of the errata locator vector is $\lambda_i=\lambda_i^{(n-k)}$ for all i.

Step 3: Recursive equations (2.54), (2.55) for $r=n-k+1, \cdots, n$ can be used in time domain to find the errata values $e_i=v_i^{(n)}$ for $i=0, \cdots, n-1$. Starting with $\lambda_i=\lambda_i^{(n-k)}$ for $i=0, \cdots, n-1$, the last iteration results in the errata value vector.

Step 4: Finally, the estimate of the errata value vector $\underline{e}$ is obtained by

$$\begin{aligned} e_i &= v_i^{(n)}, && \text{if} \quad \lambda_i = 0 \\ e_i &= 0 && \text{if} \quad \lambda_i \neq 0 \end{aligned} \tag{2.71}$$

and subtracted from the received vector $\underline{v}$ to form the corrected vector $\underline{c}$.

The time-domain algorithm based on the algebraic decoder can not be put in the form of recursive equations of (2.60)-(2.63), since the errata evaluator vector, $\underline{\omega}$, can not be initialized with the erasure vector [16].

2.6. Comparison of Decoding Algorithms

Here we compare the time-domain algorithms with the syndrome-based ones. This comparison is only from the structural point of view. In the following chapters these algorithms will be compared in detail.

There are following distinct differences in the structure of these two categories of algorithms.

1- The time-domain algorithms have fewer steps. Therefore, fewer modules are required in the design of time-domain decoders.

2- Time-domain algorithms are dealing with vectors which have n components while different length vectors and different degree polynomials are used in various steps of the syndrome-based algorithms.

3- The operations in time-domain algorithms are mainly Galois field operations, while the syndrome-based algorithms have other operations such as shift to left or right operations.

4- By changing the error correction capability of the code, the operations in the time-domain algorithms remain the same, while in the syndrome-based algorithms, each step is dependent on the error correction capability.

Considering all the above factors, time-domain algorithms are the best candidates for designing versatile hardware decoders. In the following chapters, these algorithms are restructured and all the steps of the decoding are put in one single step. For this single step cellular and noncellular structures are introduced. It is also shown that the introduced structures are very simple to design and can be used in the design of universal decoders.

The syndrome-based algorithms are used when speed is an important factor, since they are faster than time-domain algorithms. These algorithms are used in software decoders and non-versatile hardware decoders.

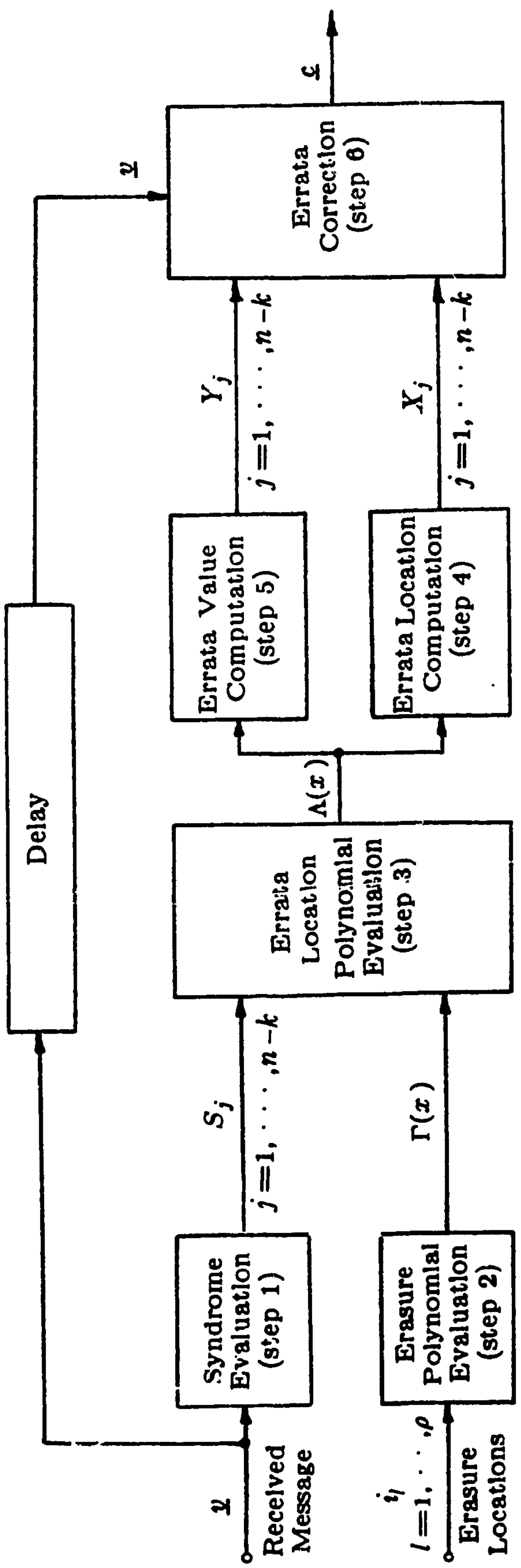

$\underline{v}$
Received Message
i_l
$l=1,\cdots,\rho$
Erasure Locations
Syndrome Evaluation (step 1)
Erasure Polynomial Evaluation (step 2)
S_j
$j=1,\cdots,n-k$
$\Gamma(x)$
Delay
Errata Location Polynomial Evaluation (step 3)
$\Lambda(x)$
Errata Value Computation (step 5)
Errata Location Computation (step 4)
Y_j
$j=1,\cdots,n-k$
X_j
$j=1,\cdots,n-k$
$\underline{v}$
Errata Correction (step 6)
$\underline{c}$

Figure 2.1: The Algebraic Decoder

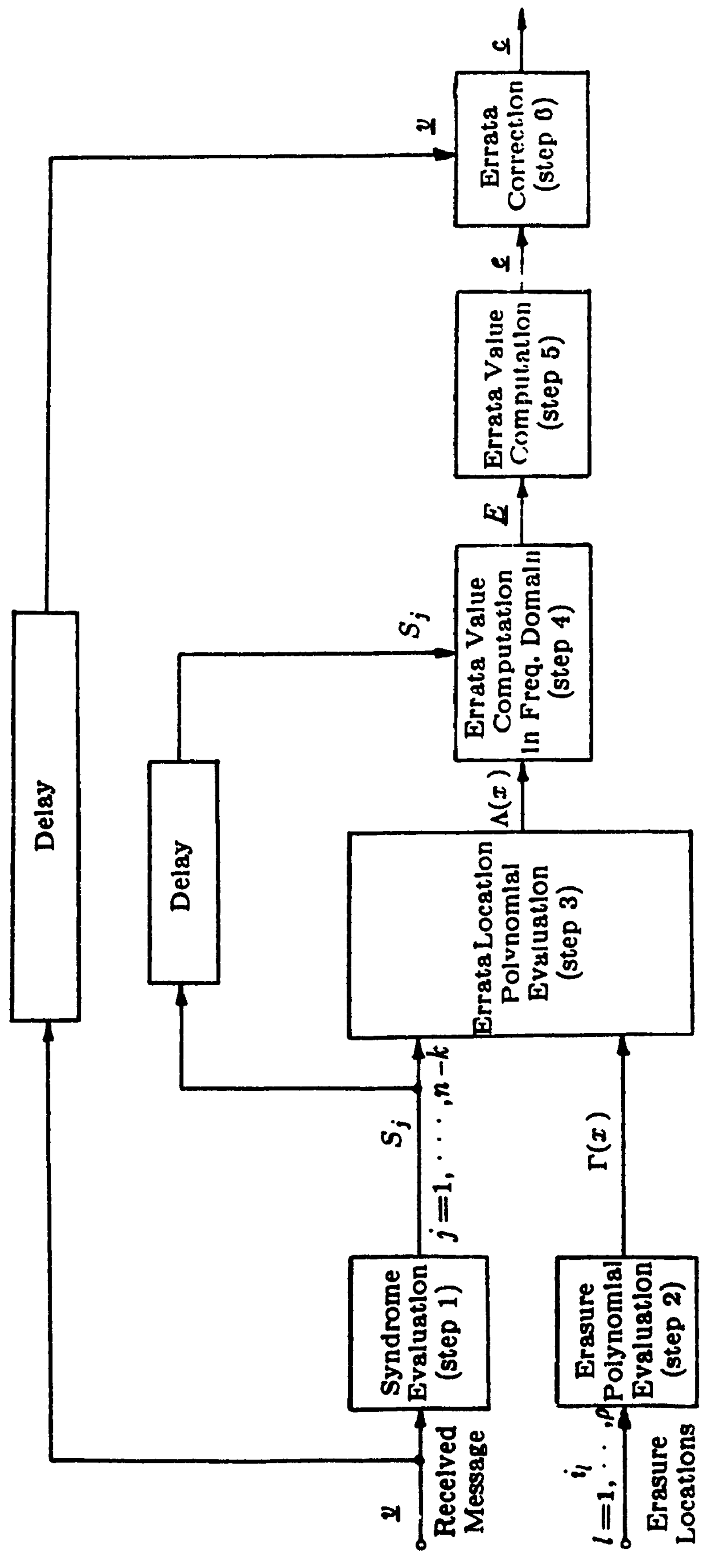

$\underline{v}$
Received Message
$l=1,\cdots,\rho$
i_l
Erasure Locations
Syndrome Evaluation (step 1)
Erasure Polynomial Evaluation (step 2)
S_j
$j=1,\cdots,n-k$
$\Gamma(x)$
Delay
Delay
Errata Location Polynomial Evaluation (step 3)
$\Lambda(x)$
S_j
Errata Value Computation In Freq. Domain (step 4)
$\underline{E}$
Errata Value Computation (step 5)
$\underline{e}$
$\underline{v}$
Errata Correction (step 6)
$\underline{c}$

Figure 2.2: The Transform Decoder

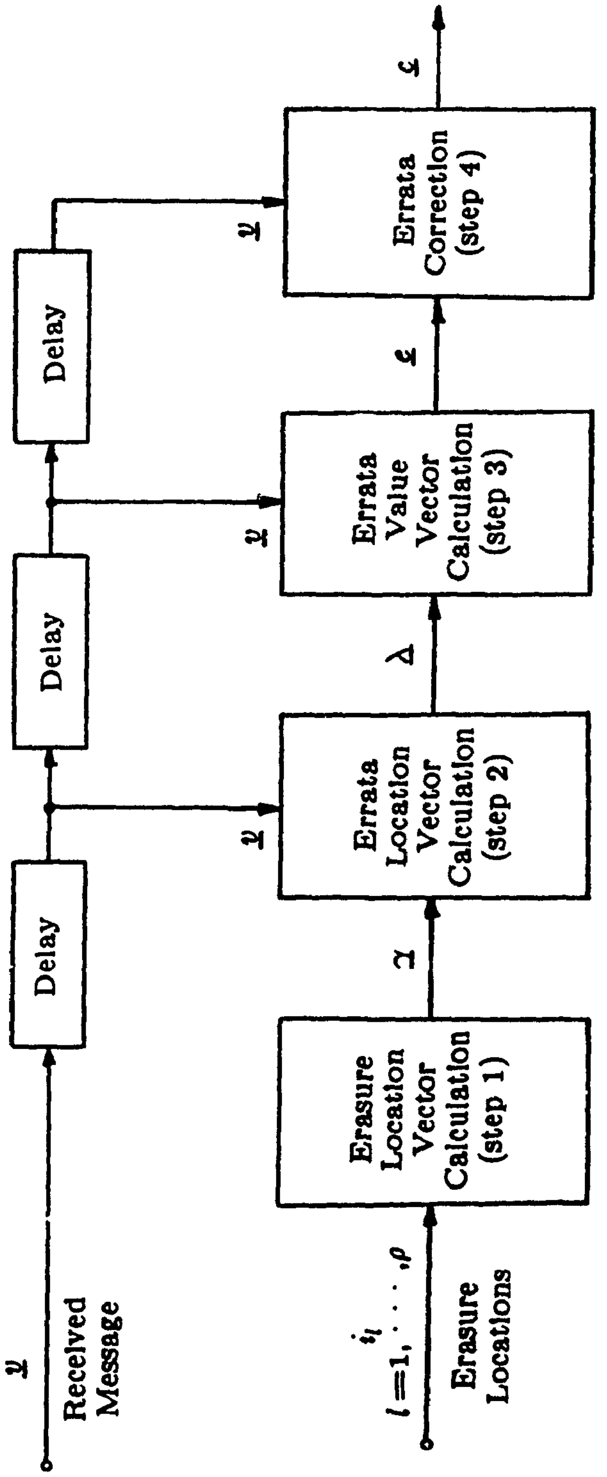


Figure 2.3: The Time-Domain Decoding Based on Transform Decoder

CHAPTER THREE

COMPUTATIONS IN GALOIS FIELDS

Reed-Solomon codes are defined in GF(2^m), and decoding of an RS code requires algebraic operations in this finite field. Therefore, before discussing decoder structures, it is necessary to study the arithmetic in Galois fields.

In this chapter, after an overview of the Galois fields and different bases for defining their elements, the multiplier and inverter structures are introduced both in standard and normal bases. It is shown that in standard basis it is possible to design a universal multiplier. By universal multiplier we mean a structure that can multiply two elements of GF(2^m) for different values of m.

As an illustrative example the least complex Massey-Omura multiplier in GF(2^5) and its LCA and VLSI designs are given.

3.1. Galois Fields

The set of integers modulo-2 (i.e. 0, 1) forms a finite or Galois field of order 2, and is denoted by GF(2). In binary arithmetic we use modulo-2 addition and multiplication. This arithmetic is actually equivalent to ordinary arithmetic, except that we consider results modulo-2, i.e., $1+1=0$. Note that since $1+1=0$, $1=-1$ and, hence, in binary arithmetic, subtraction is the same as addition. Using the binary field GF(2), one can construct extension fields with 2^m elements. There exists a Galois field of order 2^m, for every integer $m>0$ [20].

Let $I(x)$ be an irreducible polynomial of degree m over GF(2), i.e., a polynomial which has no roots in this field, given as

$$I(x)=x^m+I_{m-1}x^{m-1}+\cdots+I_0. \tag{3.1}$$

A root α of $I(x)$ exists and can be found in the extension field GF(2^m). The irreducible polynomial $I(x)$ is called the generator polynomial of GF(2^m).

The elements of GF(2^m) can be represented in powers of the primitive element α as,

$$GF(2^m)=\{0, 1, \alpha, \cdots, \alpha^i, \cdots \alpha^{2^m-2}\}. \tag{3.2}$$

The multiplication of two elements in GF(2^m) is defined as,

$$\begin{gathered} 0.\alpha^j=\alpha^j.0=0, \\ 1.\alpha^j=\alpha^j.1=\alpha^j, \\ \alpha^i.\alpha^j=\alpha^j\alpha^i=\alpha^{[(i+j)\ modulo-n]} \end{gathered} \tag{3.3}$$

where $n=2^m-1$.

It can be shown that all the elements in GF(2^m) given by The elements of GF(2^m) can be represented in polynomial form as,

$$\alpha^i=a_{i0}\alpha^0+\cdots+a_{i,m-2}\alpha^{m-2}+a_{i,m-1}\alpha^{m-1}. \tag{3.4}$$

The GF(2^m) can also be regarded as a vector space of dimension m over GF(2) where

$$S=\{1, \alpha, \alpha^2, \cdots, \alpha^{m-1}\} \tag{3.5}$$

forms a basis of the vector space. The definition of the field elements based on Eqs. (3.4), (3.5) is called standard basis definition. In general, there always exists a GF(2^m) defined in standard basis [20]. The irreducible polynomials for generating these fields are given in [20]. Using this basis each element α^i can be represented in vector form as,

$$\alpha^i=(a_{i0}, a_{i1}, \cdots, a_{i,m-1}). \tag{3.6}$$

The addition of two elements in GF(2^m) is defined as,

$$\alpha^i + \alpha^j = (a_{i0} + a_{j0},\ a_{i1} + a_{j1},\ \cdots,\ a_{i,m-1} + a_{j,m-1}), \tag{3.7}$$

where $a_{i,l} + a_{j,l}$ is carried out modulo-2.

One can find a field element α such that

$$N = \{\alpha,\ \alpha^2,\ \alpha^4,\ \cdots,\ \alpha^{2^{(m-1)}}\}, \tag{3.8}$$

is a basis for GF(2^m). This basis is called normal basis and every element α^i in GF(2^m) can be uniquely expressed in normal basis [70] as

$$\alpha^i = b_{i0}\alpha + b_{i1}\alpha^2 + b_{i2}\alpha^4 + \cdots + b_{i,m-1}\alpha^{2^{(m-1)}}, \tag{3.9}$$

or in vector form,

$$\alpha^i = (b_{i0},\ b_{i1},\ \cdots,\ b_{i,m-1}) \tag{3.10}$$

where the coefficients are in GF(2).

With regard to the properties of Galois fields, Peterson and Weldon [20] list a set of irreducible polynomials $I(x)$ of degree $m \leq 34$ over GF(2) for which the roots $\{\alpha, \alpha^2, \alpha^4, \cdots, \alpha^{2^{(m-1)}}\}$ are linearly independent. These linear independent roots clearly form a normal basis for GF(2^m) and there always exists such a basis for all positive integers, m [48].

The multiplication and addition operations in normal basis are defined in the same way as in standard basis. The multiplication is based on Eq. (3.3) and addition based on,

$$\alpha^i + \alpha^j = (b_{i0} + b_{j0},\ b_{i1} + b_{j1},\ \cdots,\ b_{i,m-1} + b_{j,m-1}). \tag{3.11}$$

Three useful properties of the finite field GF(2^m) are stated here. These properties are:

(1) Squaring in GF(2^m) is a linear operation. That is, given any two elements β and γ in GF(2^m),

$$(\beta+\gamma)^2=\beta^2+\gamma^2. \tag{3.12}$$

(ii) For any element β of GF(2^m),

$$\beta^{2^m}=\beta. \tag{3.13}$$

(iii) If α is a root of any irreducible polynomial $I(x)$ of degree m over GF(2), the powers, $\alpha, \alpha^2, \alpha^4, \cdots, \alpha^{2^{(m-1)}}$, are all in GF($2^m$) and constitute a complete set of roots of $I(x)$.

In the design of Reed-Solomon decoders, each element of the Galois field is represented by m binary digits (bits) similar to the vector representation of the element. Each bit of the element is the corresponding component of the vector representation which can be 0 or 1. Converting binary representation to decimal, the field elements have values of 0, 1, $\cdots$, n where $n=2^m-1$.

The addition or subtraction of two elements $\beta=(b_0, b_1, \cdots, b_{m-1})$ and $\gamma=(c_0, c_1, \cdots, c_{m-1})$ are performed as,

$$\beta+\gamma=\beta-\gamma=(b_0+c_0, \cdots, b_{m-1}+c_{m-1}) \tag{3.14}$$

where the component by component addition is performed in modulo-2. This operation can be performed easily using digital circuitry. For modulo-2 addition or subtraction of two m-bit elements, m bit wide exclusive or (XOR) gates are needed. Multiplication and division in Galois fields are not as simple as addition and subtraction. The division of an element β by γ is defined as $\beta/\gamma=\beta.\gamma^{-1}$ ($\gamma\neq0$). Therefore, first the multiplicative inverse of γ should be found and then multiplied by β to form the division of β/γ.

One way to do multiplication is using look-up tables, [16],[22],[33]. These methods are normally used in software decoding. In software decoders, inversion can also be done using look-up tables.

In the following sections the multipliers and multiplicative inverters are discussed for standard as well as normal basis for different generator polynomials. Our objective in these sections is to find parallel-in/parallel-out multipliers and inverters with low complexity and high speed. We are interested in universal multipliers and inverters as well as non-universal ones. By universal we mean structures which can operate in GF(2^m) for different values of m namely for m=4,5,6,7,8.

3.2. Arithmetic in Standard Basis

In this section first the structure of universal and non-universal multipliers are introduced. The best polynomials in GF(2^m) for m=4,5,6,7,8 are given and the complexity and speed of the multipliers are explained. At the end of the section the standard basis inverter is discussed.

3.2.1. Standard Basis Multiplier

Let $\beta=\sum_{i=0}^{m-1} b_i \alpha^i=(b_0, \cdots, b_{m-1})$ and $\gamma=\sum_{i=0}^{m-1} c_i \alpha^i=(c_0, \cdots, c_{m-1})$ be two elements of GF(2^m) in standard basis. The product can be written as,

$$\begin{aligned}\delta=\beta.\gamma&=(b_0+b_1\alpha+\cdots+b_{m-1}\alpha^{m-1}).\gamma\\&=b_0\gamma+b_1\gamma.\alpha+\cdots+b_{m-1}\gamma.\alpha^{m-1}.\end{aligned} \tag{3.15}$$

Eq. (3.15) can be implemented as shown in Fig. 3.1 [41]. This structure needs $m-1$, blocks of α-multiplier, m^2 AND gates and $m(m-1)$ XOR gates. In this structure only the α-multiplier circuitry depends on the generator polynomial of the field. The structure of Fig. 3.1 is modular and simple.

To complete the structure of the multiplier, the α-multiplier block should be designed. The multiplication by α can be explained by

$$\alpha.\gamma=c_0\alpha+c_1\alpha^2+\cdots+c_{m-1}\alpha^m . \tag{3.16a}$$

This equation can be reduced to

$$\alpha.\gamma = I_0 c_{m-1} + (c_0 + I_1 c_{m-1})\alpha + \cdots + (c_{m-2} + I_{m-1} c_{m-1})\alpha^{m-1}. \qquad (3.16b)$$

In the reduction to Eq. (3.16b), we have considered the fact that the element α is the root of the generator polynomial of the field, i. e.,

$$\alpha^m = I_{m-1}\alpha^{m-1} + \cdots + I_1\alpha + I_0.$$

In [18], it is shown that more than one irreducible polynomial can generate GF(2^m) for m =4,5,6,7,8. Therefore for each of the values of m, the polynomial which introduces the least complex α-multiplier, should be chosen. The maximum number of gates needed to design an α-multiplier in GF(2^m) is $m-1$ XOR gates and the propagation delay is always equivalent to delay of one gate.

Obviously the polynomial which will introduce the least complex α-multiplier is the one which has the least number of nonzero coefficients. As an illustrative example in Fig. 3.2., the least complex design of α-multiplier is given for GF(2^5). It is shown that there are two polynomials which introduce the least complex α-multiplier. Each of these α-multipliers need only one XOR gate.

In Table 3.1, the best irreducible polynomials in each GF(2^m) for m =4,5,6,7,8 are given. These generator polynomials are shown by the powers of their nonzero terms. For example, let m =4 then the notation (4, 3, 0) means the polynomial $x^4 + x^3 + 1$. As shown there are 2,2,3,4,17 polynomials available for m =4,5,6,7,8, respectively. The best polynomials for different values of m, generate multipliers with the complexities as shown in Table 3.2. The propagation delay of these multipliers are equal to delay of $m+1$ gates.

Table 3.1: The best Generator Polynomials of GF(2^m)

m	nonzero terms
4	(4, 1, 0) (4, 3, 0) *
5	(5, 2, 0) (5, 3, 0) *
6	(6, 1, 0) (6, 3, 0) * (6, 5, 0)
7	(7, 1, 0) (7, 3, 0) * (7, 4, 0) (7, 6, 0)
8	(8, 4, 3, 1, 0) * (8, 4, 3, 2, 0) * (8, 5, 3, 1, 0) * (8, 5, 3, 2, 0) * (8, 5, 4, 3, 0) * (8, 6, 3, 2, 0) * (8, 6, 5, 1, 0) (8, 6, 5, 2, 0) (8, 6, 5, 3, 0) * (8, 6, 5, 4, 0) (8, 7, 2, 1, 0) (8, 7, 3, 1, 0) * (8, 7, 3, 2, 0) * (8, 7, 5, 1, 0) (8, 7, 5, 3, 0) * (8, 7, 5, 4, 0) (8, 7, 6, 1, 0)

Table 3.2: Number of Gates of α-multiplier and Multiplier

Designs in GF(2^m)

m	α-Multiplier	Multiplier
4	1	31
5	1	49
6	1	71
7	1	97
8	3	141

3.2.2. Universal Standard Basis Multiplier

The structure of Fig. 3.1 can be used to design the universal multiplier. To have a universal multiplier in GF(2^m) for m =4,5,6,7,8, a universal α-

multiplier should be designed.

The complexity of the universal α-multiplier depends on the polynomials which generate the GF(2^m) for $m=4,5,6,7,8$. To have a low complexity α-multiplier, the generator polynomials should be chosen from Table 3.1. If, for different values of m, we can find polynomials with common nonzero coefficients, the complexity will be very low. For $m=4,5,6,7$, it is possible to find such polynomials having the general form of $I_m(x)=x^{m-1}+x^3+1$ (indicated by "*" in Table 3.1). But none of the polynomials in the GF(2^8) has this general form. In GF(2^8) any of the polynomials indicated by "*" in Table 3.1, can be used and we have considered the first polynomial, i.e., $x^8+x^4+x^3+x+1$. Using these polynomials, the definition of α-multiplier can be found based on Eq. (3.16). These definitions of α-multiplier can be reduced to

$$\alpha.\gamma=c_{m-1}+(c_0+c_7)\alpha+c_1\alpha^2+(c_2+c_{m-1})\alpha^3+(c_3+c_7)\alpha^4+c_4\alpha^5+c_5\alpha^6+c_6\alpha^7 \tag{3.17a}$$

or in vector form,

$$\alpha.\gamma=\alpha.(c_0,c_1,\cdots,c_{m-1})=(c_{m-1},c_0+c_7,c_1,c_2+c_{m-1},c_3+c_7,c_4,c_5,c_6). \tag{3.17b}$$

Eq. (3.17) defines the universal α-multiplier as long as the value of c_{m-1} is known. In Fig. 3.3, this α-multiplier is shown which is configurable to different α-multipliers based on the variables gf_m, $m=4,5,6,7,8$. At any time, only one of these variables has high level. This determines working field of the α-multiplier. As shown in Fig. 3.3., there is a need for 3 XOR and 5 PASS gates to design the universal α-multiplier. A PASS gate is a transistor which controls the transfer of information. The propagation delay of this α-multiplier is equivalent to two gates.

The design of the universal multiplier is the same as the GF(2^m) multiplier of Fig. 3.1 where $m=8$. There are two differences between GF(2^8) and universal multipliers. In universal multiplier we should use the universal α-

multiplier of Fig. 3.3. Moreover, in the universal multiplier a block should be added at the output of the multiplier to force the $8-m$ most significant digits of the product to zero. This block is shown in Fig. 3.4, which needs 4 AND gates and 3 OR gates. For example if gf_4 is high, only 4 least significant digits can be nonzero, i.e., $d_7=d_6=d_5=d_4=0$.

The universal multiplier needs 183 gates and has a propagation delay equivalent to 17 gates for any value of $m=4,5,6,7,8$.

3.2.3. Standard Basis Inverter

In standard basis there is no structure available for the inverter which has low complexity and which can find the inverse of the elements with a very high speed. Structures available are either high complexity [40] or low speed [16]. Therefore, table look-up method is normally used for the inverter design. In this method, the preprocessed inverse values of the elements of Galois field are stored in a Read-Only-Memory (ROM) chip. This ROM can find the inverse values by applying the element in binary form to the address lines of the chip.

Obviously this approach for finding the inverse, can not be used when a large number of inverters are needed in a specific design. Fortunately, in the design of RS decoders only one inverter is needed and hence it is possible to use the table look-up method.

To design a universal inverter using this method, five partitions for the ROM chip should be considered for $m=4,5,6,7,8$. These partitions have the inverse values for different values of m.

3.3. Arithmetic in Normal Basis

For constructing a GF(2^m) in normal basis, there is more than one irreducible polynomial for $m=4,5,6,7,8$ [20].

Table 3.3: Nonzero Terms of Generator Polynomials for Generation of GF(2^m) in Normal Basis

m	nonzero terms
4	(4, 3, 0)
	(4, 3, 2, 1, 0)
5	(5, 4, 2, 1, 0)
	(5, 4, 3, 1, 0)
	(5, 4, 3, 2, 0)
6	(6, 5, 0)
	(6, 5, 2, 1, 0)
	(6, 5, 4, 1, 0)
	(6, 5, 4, 2, 0)
7	(7, 6, 0)
	(7, 6, 2, 1, 0)
	(7, 6, 4, 1, 0)
	(7, 6, 4, 2, 0)
	(7, 6, 5, 2, 0)
	(7, 6, 5, 3, 2, 1, 0)
	(7, 6, 5, 4, 2, 1, 0)
8	(8, 7, 2, 1, 0)
	(8, 7, 3, 1, 0)
	(8, 7, 3, 2, 0)
	(8, 7, 5, 1, 0)
	(8, 7, 5, 3, 0)
	(8, 7, 5, 4, 0)
	(8, 7, 6, 1, 0)
	(8, 7, 4, 3, 2, 1, 0)
	(8, 7, 5, 5, 3, 2, 0)
	(8, 7, 6, 3, 2, 1, 0)
	(8, 7, 6, 4, 2, 1, 0)
	(8, 7, 6, 4, 3, 2, 0)
	(8, 7, 6, 5, 2, 1, 0)
	(8, 7, 6, 5, 4, 1, 0)
	(8, 7, 6, 5, 4, 2, 0)
	(8, 7, 6, 5, 4, 3, 0)

Suppose that $\{\alpha^{2^{(m-1)}}, \ldots \alpha^4, \alpha^2, \alpha\}$ is a normal basis for GF(2^m). Using the properties of the Galois fields, the square of an element $\beta = b_{m-1}\alpha^{2^{m-1}} + \cdots + b_0\alpha$ is

$$\beta^2 = b_{m-2}\alpha^{2^{(m-1)}} + \ldots + b_0\alpha^2 + b_{m-1}\alpha. \qquad (3.18)$$

Thus, if β is represented in vector form, $\beta = (b_{m-1}, b_{m-2}, \ldots, b_2, b_1, b_0)$, then the square of β is, $\beta^2 = (b_{m-2}, b_{m-3}, \ldots, b_1, b_0, b_{m-1})$. Hence in normal basis representation β^2 is a cyclic shift of β. This property can be used to define normal basis inverter and multipliers [39].

3.3.1. The Massey-Omura Multiplier

In this subsection the Massey-Omura multiplier which is based on normal basis representation is explained [39].

Let $\beta=(b_{m-1}, \ldots, b_1, b_0)$ and $\gamma=(c_{m-1}, \ldots, c_1, c_0)$ be two elements of GF(2^m) in a normal basis representation. Then the last term d_{m-1} of the product,

$$\delta=\beta.\gamma=(d_{m-1}, \ldots, d_1, d_0) \tag{3.19}$$

is some binary function of the components of β and γ, i.e.,

$$d_{m-1}=f\ (b_0, b_1, , \ldots, b_{m-1}; c_0, c_1, \ldots, c_{m-1}). \tag{3.20}$$

Since squaring means a cyclic shift of an element in a normal basis representation, one has $\delta^2=\beta^2.\gamma^2$ or equivalently,

$$\begin{aligned}&(d_{m-2}, \ldots, d_1, d_0, d_{m-1})=\\&\quad=(b_{m-2}, \ldots, b_1, b_0, b_{m-1}).(c_{m-2}, \ldots, c_1, c_0, c_{m-1}).\end{aligned} \tag{3.21}$$

Hence the last component d_{m-2} of δ^2 is obtained by the same function f in Eq. (3.20). That is, $d_{m-2}=f\ (b_{m-1}, b_0, b_1, \ldots, b_{m-2}; c_{m-1}, c_0, c_1, \ldots, c_{m-2})$. By squaring δ repeatedly, it is evident that

$$\begin{aligned}d_{m-1}&=f\ (b_0, b_1, \ldots, b_{m-1}; c_0, c_1, \ldots, c_{m-1})\\d_{m-2}&=f\ (b_{m-1}, b_0, b_1, \ldots, b_{m-2}; c_{m-1}, c_0, c_1, \ldots, c_{m-2})\\&\cdot\\&\cdot\\&\cdot\\d_0\ &=f\ (b_1, b_2, \ldots, b_{m-1}, b_0; c_1, c_2, \ldots, c_{m-1}, c_0)\end{aligned} \tag{3.22}$$

The equations in Eq. (3.22) define the Massey-Omura multiplier. In the normal basis representation, this multiplier has the property that the same logic function f which is used for finding the last component of d_{m-1} of the product δ can be used to find the remaining components $d_{m-2}, d_{m-3}, \ldots, d_0$ of the product. This feature can be used in the design of serial and parallel type multipliers [38],[39]. Here we only consider the parallel Massey-Omura multipliers, since this kind of multiplier is required in the structures introduced in following chapters.

For parallel type multipliers, the above mentioned feature permits the use of m identical logic functions, f, for calculating simultaneously all components of the product. In this case, the inputs to the m logic functions f are connected directly to the components of β and γ. The only difference in the connections to the components of β or γ to a function f is that they are cyclically shifted versions of one another. The main problem in the design of these multipliers is availability of more than one irreducible polynomial which generate a GF(2^m) for $m \geq 4$ and represent a normal basis. Therefore, there are more than one design for the GF(2^m) multiplier, since the structure of the parallel Massey-Omura multiplier depends on the generator polynomial of the field.

As an example, consider the design of the parallel multiplier in GF(2^5). The structure of this multiplier based on the product function f is given in Fig. 3.5. To complete the design in this figure we should focus on the product function f.

3.3.2. The Least Complex Product Function

The design of f in GF(2^5) begins with the selection of the irreducible polynomial of degree $m=5$ over GF(2) [38],[42]. There are three irreducible polynomials (Table 3.3) which generate GF(2^5) in a normal basis representation. These polynomials are

$$\begin{aligned}I_1(x)&=x^5+x^4+x^2+x+1,\\ I_2(x)&=x^5+x^4+x^3+x+1,\\ I_3(x)&=x^5+x^4+x^3+x^2+1.\end{aligned}\tag{3.23}$$

These three polynomials have linearly independent roots, namely, α, α^2, α^4, α^8, and α^{16}.

Any two elements of β and γ in GF(2^5) can be expressed as

$$\begin{aligned}\beta&=(b_0\alpha^1+b_1\alpha^2+b_2\alpha^4+b_3\alpha^8+b_4\alpha^{16}),\\ \gamma&=(c_0\alpha^1+c_1\alpha^2+c_2\alpha^4+c_3\alpha^8+c_4\alpha^{16}).\end{aligned}\tag{3.24}$$

The product of β and γ is $\delta=\beta.\gamma$ or equivalently,

$$\begin{aligned}&d_0\alpha^1+d_1\alpha^2+d_2\alpha^4+d_3\alpha^8+d_4\alpha^{16}=\\ &\quad=(b_0\alpha^1+b_1\alpha^2+b_2\alpha^4+b_3\alpha^8+b_4\alpha^{16}).\\ &\qquad.(c_0\alpha^1+c_1\alpha^2+c_2\alpha^4+c_3\alpha^8+c_4\alpha^{16})\\ &\quad=b_0c_0\alpha^2+b_0c_1\alpha^3+b_0c_2\alpha^5+b_0c_3\alpha^9+b_0c_4\alpha^{17}+\\ &\qquad b_1c_0\alpha^3+b_1c_1\alpha^4+b_1c_2\alpha^6+b_1c_3\alpha^{10}+b_1c_4\alpha^{18}+\\ &\qquad b_2c_0\alpha^5+b_2c_1\alpha^6+b_2c_2\alpha^8+b_2c_3\alpha^{12}+b_2c_4\alpha^{20}+\\ &\qquad b_3c_0\alpha^9+b_3c_1\alpha^{10}+b_3c_2\alpha^{12}+b_3c_3\alpha^{16}+b_3c_4\alpha^{24}+\\ &\qquad b_4c_0\alpha^{17}+b_4c_1\alpha^{18}+b_4c_2\alpha^{20}+b_4c_3\alpha^{24}+b_4c_4\alpha^{32},\end{aligned}\tag{3.25}$$

where $\alpha^{32}=\alpha$. Using Eq. (3.25) and the fact that α is a root of the generator polynomial, one obtains three product functions f_1, f_2, and f_3 corresponding to three generating polynomials $I_1(x)$, $I_2(x)$, and $I_3(x)$, respectively. These product functions are

$$f_1(b_0, b_1, b_2, b_3, b_4; c_0, c_1, c_2, c_3, c_4)=$$
$$=b_0c_2 + b_2c_0 + b_0c_4 + b_4c_0 + b_1c_2 + b_2c_1 +$$
$$b_1c_3 + b_3c_1 + b_3c_3, \tag{3.26}$$

$$f_2(b_0, b_1, b_2, b_3, b_4; c_0, c_1, c_2, c_3, c_4)=$$
$$=b_0c_1 + b_1c_0 + b_0c_3 + b_3c_0 + b_1c_3 + b_3c_1 +$$
$$b_2c_3 + b_3c_2 + b_2c_4 + b_4c_2 + b_3c_3, \tag{3.27}$$

$$f_3(b_0, b_1, b_2, b_3, b_4; c_0, c_1, c_2, c_3, c_4)=$$
$$=b_0c_2 + b_2c_0 + b_0c_3 + b_3c_0 + b_1c_2 + b_2c_1 +$$
$$b_1c_4 + b_4c_1 + b_2c_3 + b_3c_2 + b_2c_4 + b_4c_2 +$$
$$b_3c_3 + b_3c_4 + b_4c_3. \tag{3.28}$$

These three product functions offer three different logic circuitries. The logic circuitries for functions f_1, f_2 and f_3 need 9, 11 & 15 AND gates and also 8, 10 & 14 XOR gates, respectively. All of these three product functions have propagation delays equivalent to delay of five gates. In hardware design of the multiplier, the dominant factor for choosing the product function is the number of the gates needed for the design. Based on the gate numbers, f_1 introduces the least complex design and the corresponding polynomial $I_1(x)$ should be used to design the GF(2^5) multiplier. The function f_1 is equivalent to digit d_4 of the product and the other digits can be found using Eq. (3.22). The logic design of the product function f_1 is given in Fig. 3.6.

In [44], the minimum number of terms of the product function f is given for different values of m as shown in Table 3.4. Based on the number of terms of f, the complexity and propagation delay of the GF(2^m) multiplier can be evaluated which are also given in Table 3.4.

Table 3.4: Complexity and Propagation Delay of

Massey-Omura Multiplier

m	Minimum terms of f	Gate # of Multiplier	Delay
4	7	52	4
5	9	85	5
6	11	132	5
7	19	259	6
8	21	328	6

3.3.3. LCA and VLSI Designs of the Multiplier

In this section, the implementation of the GF(2^5) Massey-Omura multiplier is discussed [42]. LCA and VLSI designs are considered and compared. For LCA design, a new LCA system produced by Xilinx's [71] is considered. In VLSI design, Complementary-Metal-Oxide-Semiconductor (CMOS) 3 μm technology is used.

A 1200-gate LCA with an array of 64 Configurable-Logic-Blocks (CLB) and 58 Input/Output-Blocks (IOB) is used. Each CLB has a combinational logic section, a storage element, and an internal routing and control section. Each CLB has also four general purpose inputs, a clock input and two outputs. Each IOB provides an interface between the external package pin of the device and the internal logic. The creation of a logic network from these elements is defined by a configuration program stored in an internal memory (RAM). Therefore, there is no custom factory fabrication process. The configuration program which is developed on an IBM PC/AT is either loaded automatically from an external memory or programmed by a processor when the device is powered up, or upon command while the system is operating. This LCA chip is fabricated with 1.2 μm CMOS technology. Power consumption is very low and clock speed can be as high as 35 MHz.

To design the GF(2^5) multiplier using LCA, first the product function f_1 should be configured. In order to minimize the resources required by this design, it is necessary to partition the design of the product function f_1. Note that this partitioning is required to minimize the design, since each CLB of an LCA can generate a function of at most four variables. To prevent unnecessary use of blocks, we must try to have partitions with four input variables. The partitions of the expression for the function f_1 are $\{b_1, b_3, c_1, c_3\}$, $\{b_2, b_4, c_0, c_1\}$, and $\{b_0, b_1, c_2, c_4\}$ and, hence, f_1 is equivalent to,

$$\begin{aligned} f_1(b_0, b_1, b_2, b_3, b_4; c_0, c_1, c_2, c_3, c_4) &= \\ &= (b_3c_3 + b_3c_1 + b_1c_3) + (b_4c_0 + b_2c_0 + b_2c_1) + \\ &\quad (b_0c_4 + b_0c_2 + b_1c_2). \end{aligned} \tag{3.29}$$

The function f_1 in the above equation is equivalent to the digit d_4 of the product in Eq. (3.25). Other digits of the product can be found using Eq. (3.22) and the same kind of partitioning.

From the partitioned equation (3.29), it is clear that each digit of the product will require four logic blocks (CLBs). Three CLBs for three partitions and one for finding the resultant digit. Therefore to evaluate five digits of the product $(d_4, d_3, d_2, d_1, d_0)$, 20 out of 64 available CLBs are required. We also need 15 IOBs for 10 inputs and 5 outputs of the multiplier. The propagation delay of this LCA multiplier is 270 ns, and therefore, the maximum operating frequency can be 3.7 MHz. This propagation delay has been measured by simulation. The reason for the high propagation delay is the limited number of "direct" interconnect lines between the CLBs. In this implementation it is necessary to use indirect interconnections passing through pass transistors which yields to large propagation delay. The configuration of the LCA chip for the GF(2^5) multiplier is given in Fig. 3.7.

Using VLSI technology we can design multipliers which are more reliable, consume less power, and operate at higher speeds. However, this requires a longer development time and higher engineering cost. Normally, the VLSI chips can be afforded for large production volumes.

To develop the VLSI layout, first the layout of the product function f_1 of Fig. 3.6 is done to form the digit d_4 of the product in Eq. (3.25). To develop the other four bits of the product, this layout can be copied four times and proper interconnections according to Fig. 3.5 can be done. The VLSI layout of the GF(2^5) multiplier is shown in Fig. 3.8.

This VLSI chip takes a chip area of 3800 μm by 800 μm and the propagation delay is equal to 32 ns, and therefore, the maximum operating frequency can be about 30 MHz. This shows that VLSI design can be up to 9 times faster than LCA design. The propagation delay of 32 ns which also includes interconnection capacitance effects, has been measured by a simulation package.

3.3.4. Another Normal Basis Multiplier

Let $\beta=\sum_{i=0}^{m-1} b_i \alpha^{2^i}=(b_0, \cdots, b_{m-1})$ and $\gamma=\sum_{i=0}^{m-1} c_i \alpha^{2^i}=(c_0, \cdots, c_{m-1})$ be two elements of GF(2^m) in normal basis. The product in polynomial representation can be written as,

$$\begin{aligned}\delta=\beta.\gamma&=(b_0\alpha+b_1\alpha^2+\cdots+b_{m-1}\alpha^{2^{m-1}}).\gamma\\&=b_0\gamma.\alpha+b_1\gamma.\alpha^2+\cdots+b_{m-1}\gamma.\alpha^{2^{m-1}}.\end{aligned} \tag{3.30}$$

Eq. (3.30) can be implemented as shown in Fig. 3.9. This structure needs m, blocks of α^{2^i}-multiplier for $i=0,1,\ldots,m-1$. Moreover, m^2 AND gates and m^2 XOR gates are also used in the multiplier structure. In this structure, only the α^{2^i}-multiplier circuitry depends on the generator polynomial of the field.

To complete the structure of the multiplier, the α^{2^i}-multiplier blocks should be designed. These blocks can be designed using the Massey-Omura multiplier. As an example, we consider GF(2^5) and use the best generator polynomial to design the α^{2^i}-multipliers.

The α^{2^i}-multipliers are special case of the Massey-Omura multipliers since α^{2^i} for $i=0,1,\cdots,m-1$ are constant values. These elements can be represented in binary form as shown in Table 3.5. Using the binary form of these constants, the least complex product function f of Eq. (3.26) can be simplified as shown in Table 3.5. To form this table, it is assumed that α^{2^i} is multiplied by $\gamma=(c_4,c_3,c_2,c_1,c_0)$. In Table 3.5, complexity of each product function f and the corresponding α^{2^i}-multiplier is also given. The propagation delay of these α^{2^i}-multipliers is equivalent to delay of one gate for $i=0,1,2,3$ and has no delay for $i=4$. The complexity and propagation delay of the normal basis GF(2^m) multiplier of Fig. 3.9 are given in Table 3.6 for $m=4,5,6,7,8$.

Table 3.5: Complexity and Propagation Delay of α^{2^i}-Multipliers in Normal Basis

	Vector Form	Product Function (f)	# of Gates (f)	# of Gates (α^{2^i}-Mult.)	Delay (α^{2^i}-Mult.)
α	(00001)	c_2+c_4	1	5	1
α^2	(00010)	c_2+c_3	1	5	1
α^4	(00100)	c_0+c_1	1	5	1
α^8	(01000)	c_1+c_3	1	5	1
α^{16}	(10000)	c_0	0	0	0

Table 3.6: Complexity and Propagation Delay of

Normal-Basis Multiplier

m	Gate # of Multiplier	Delay
4	44	4
5	70	5
6	102	5
7	182	6
8	232	6

3.3.5. Normal Basis Inverter

For any element β in GF(2^m), $\beta^{2^m}=\beta$. Hence the inverse of β is $\beta^{-1}=\beta^{2^m-2}$. Let 2^m-2 be decomposed as $2+2^2+2^3+\cdots+2^{m-1}$ [38], then β^{-1} can be expressed as

$$\beta^{-1}=(\beta^2).(\beta^{2^2}).(\beta^{2^3}).\cdots.(\beta^{2^{m-1}}). \tag{3.31}$$

Therefore, the inverter in GF(2^m) can be realized by $m-2$ multipliers based on the above equation. Note that the square of the element β is just a change of interconnection to the multiplier circuit. As an illustrative example the inverter structure for GF(2^5) is given in Fig. 3.10. The GF(2^m) inverter has a propagation delay equivalent to 2 multipliers for $m=4,5$ and 3 multipliers for $m=6,7,8$.

3.4. Comparison of Normal and Standard Bases

In this chapter, a standard (Fig. 3.1) and a normal basis (Fig. 3.9) multiplier in GF(2^m) were introduced. The complexity of these multipliers and the Massey-Omura multiplier is given in Table 3.7 for comparison. This table shows that the num' ' gates for the standard basis multiplier is less than hat of the other . ' ɛrs. The Massey-Omura multiplier has the most complex structure, howe' is more modular than the other structures.

Table 3.7: Comparison of Complexity of

GF(2^m) Multipliers

m	Fig. 3.1	Fig. 3.5	Fig. 3.9
4	31	52	44
5	49	85	70
6	71	132	102
7	97	259	182
8	141	328	232

The propagation delay of the multipliers is compared in Table 3.8 which shows higher delay for the standard basis multiplier. The delays of two normal basis multipliers are the same. Note that it is possible to design these multipliers for lower propagation delays, but the complexity will increase.

In normal basis an inverter can be designed which has complexity of $m-2$ multipliers. The propagation delay of this GF(2^m) inverter is equivalent to delay of 2 multipliers for $m=4,5,6$ and 3 multipliers for $m=7,8$. The inverter in standard basis is very complex, therefore table look-up should be used.

Table 3.8: Comparison of Propagation Delay of

GF(2^m) Multipliers

m	Fig. 3.1	Fig. 3.5	Fig. 3.9
4	5	4	4
5	6	5	5
6	7	5	5
7	8	6	6
8	9	6	6

In this chapter, a universal multiplier was also introduced which can be configured for $m=4,5,6,7,8$. The universal multiplier needs 183 gates and has a delay of 17 gates. In normal basis, there is no universal multiplier available.

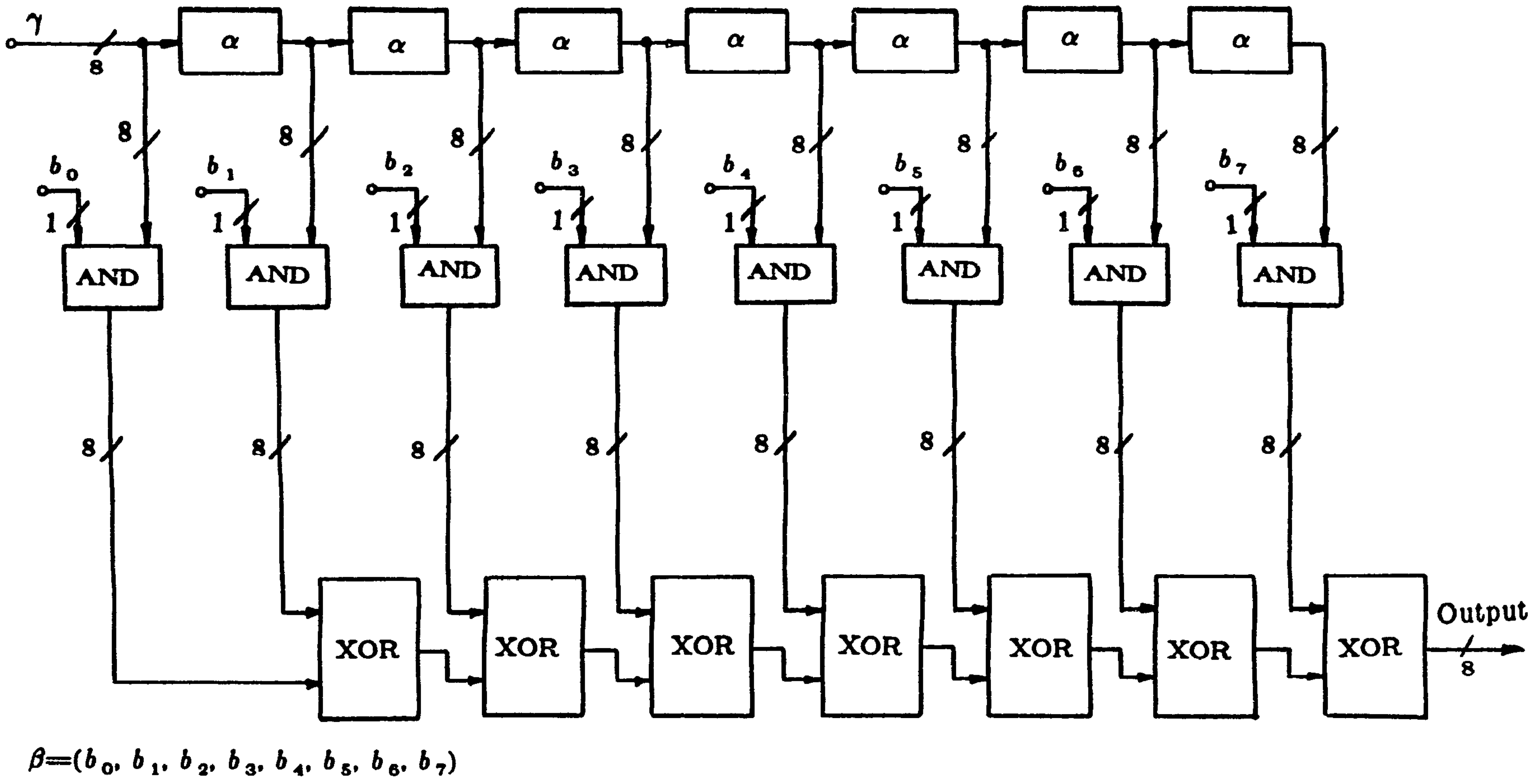
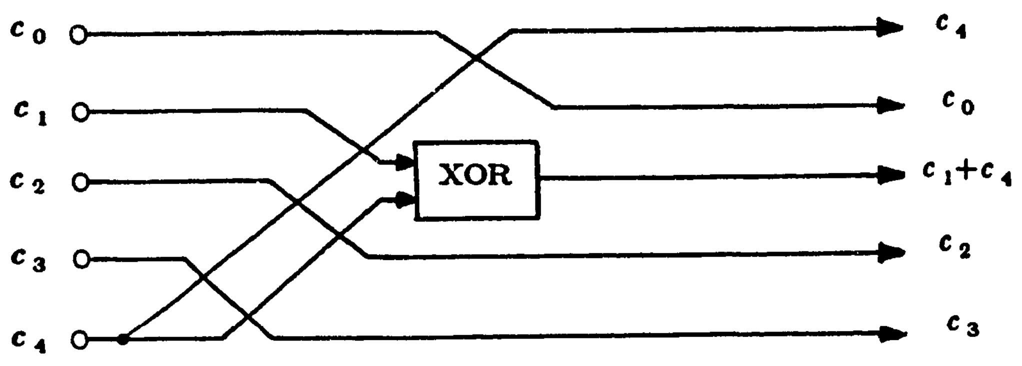
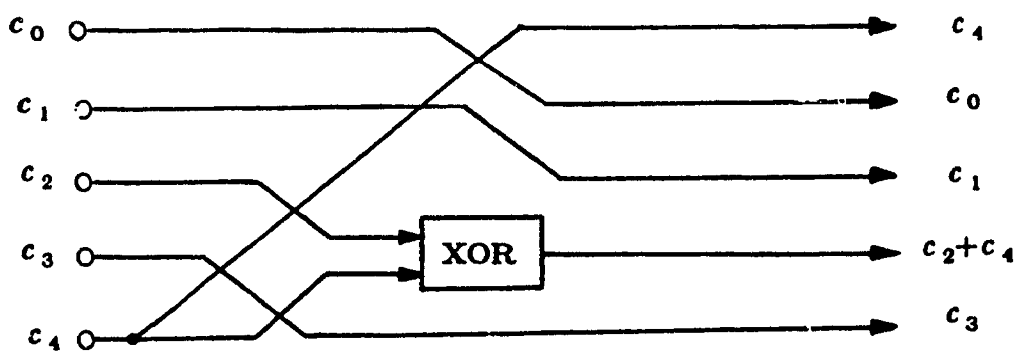

γ
8
α
8
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
1
AND
XOR
Output
$\beta = (b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$

Figure 3.1: Parallel-in/Parallel-out GF(2^8) Multiplier

in Standard Basis



(a)



(b)

Figure 3.2: GF(2^5) α-Multipliers in Standard Basis Generated by a) x^5+x^2+1 and b) x^5+x^3+1

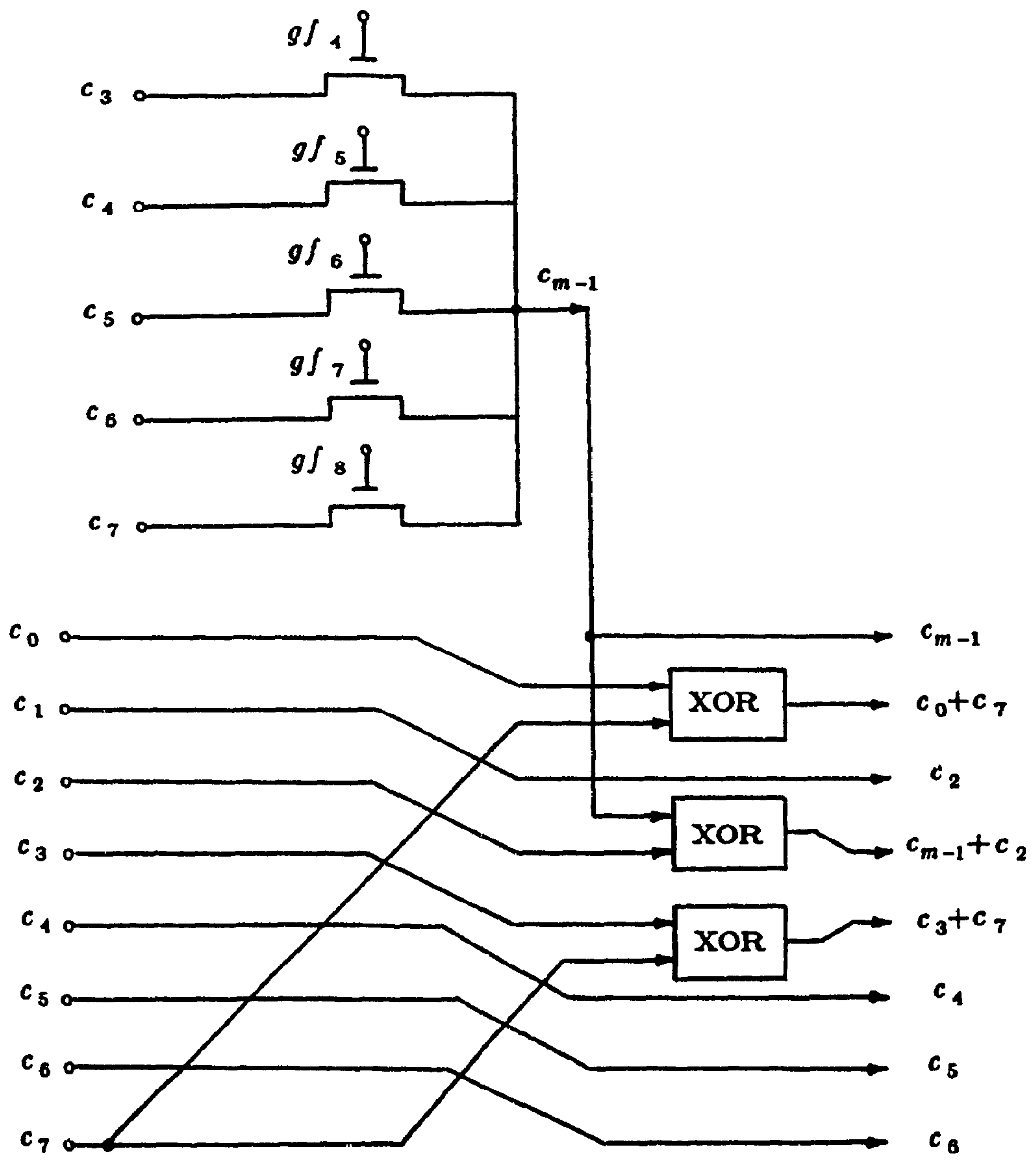


Figure 3.3: Universal α-Multiplier in GF(2^m)

for m =4,5,6,7,8

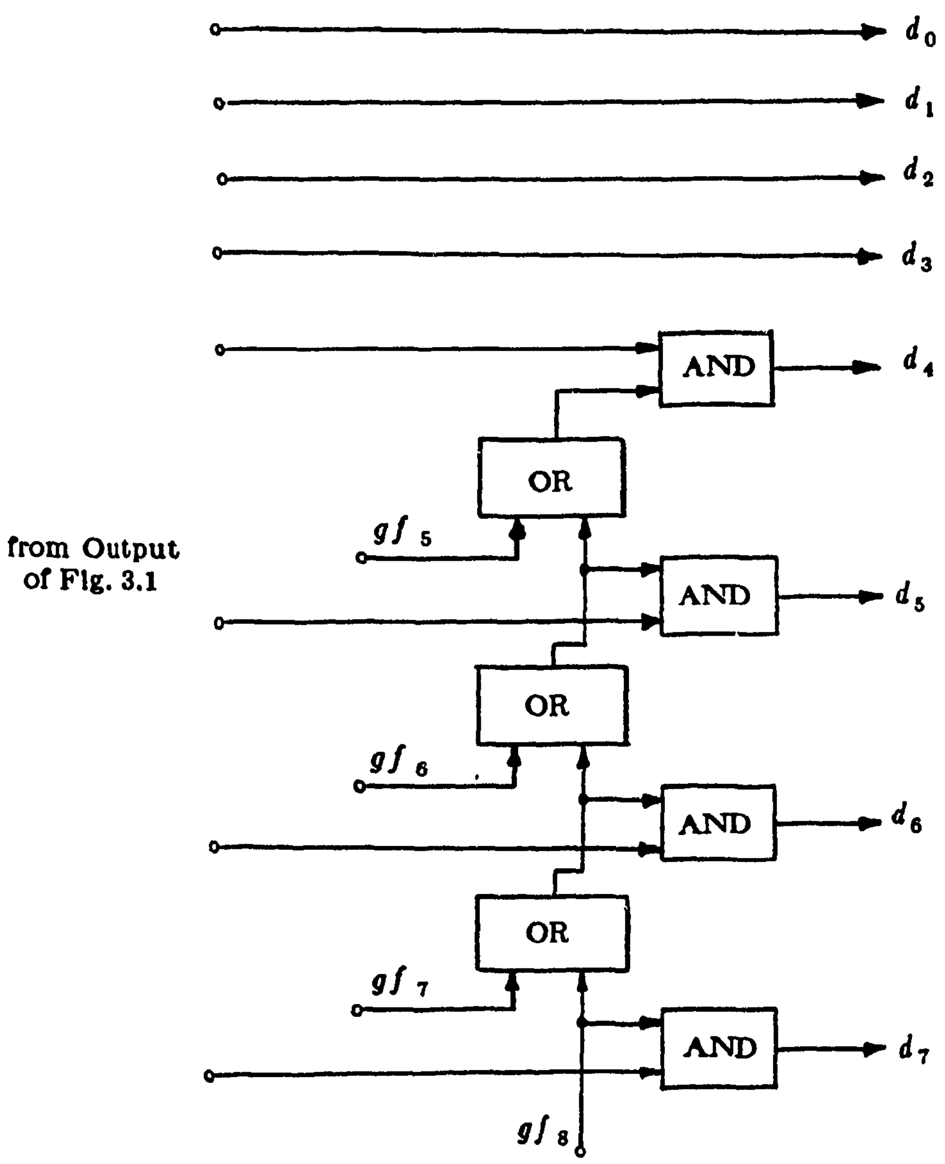


Figure 3.4: Block for Forcing the 8–m Most Significant Digits of the Product to Zero

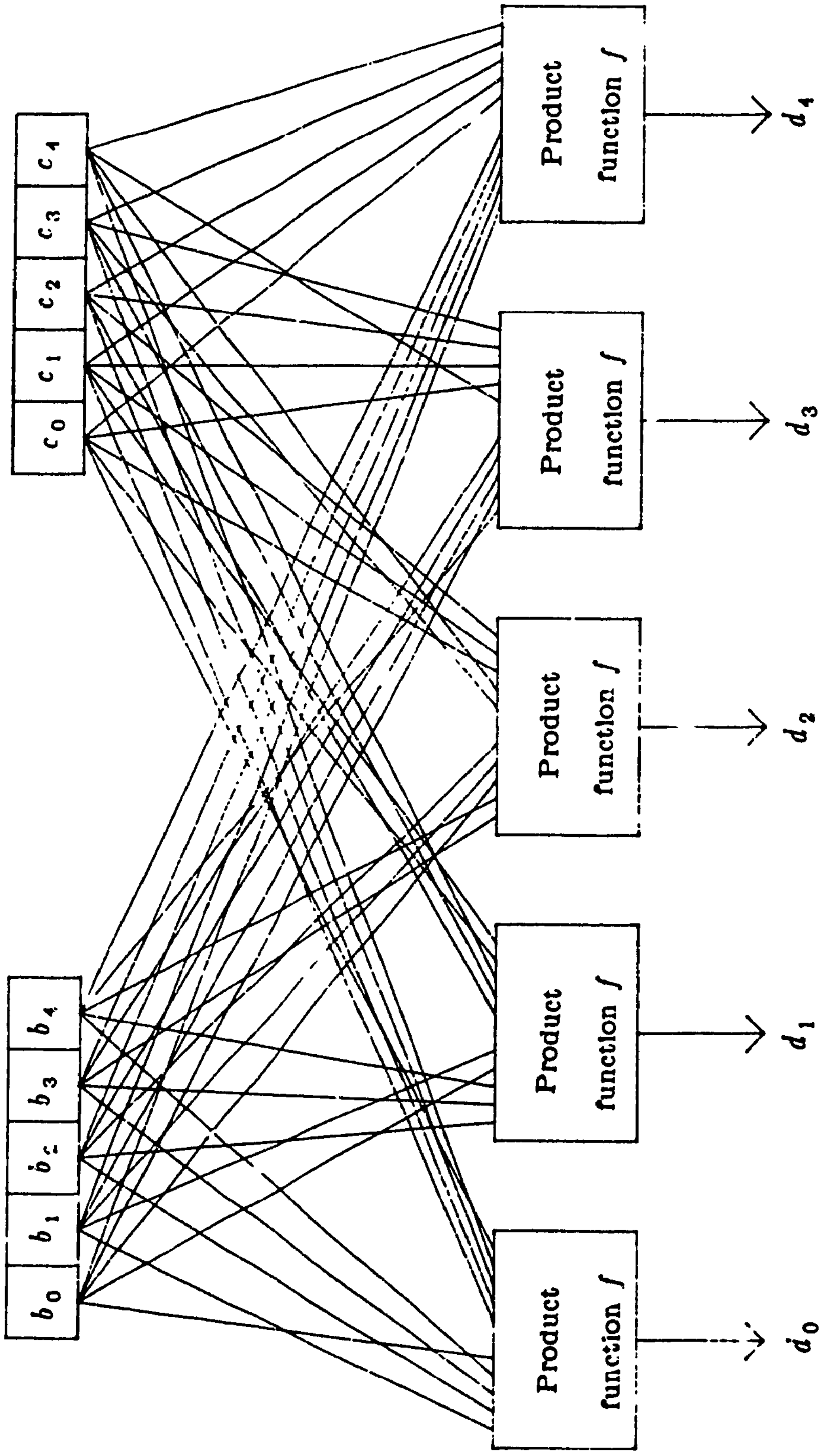


Figure 3.5: The Parallel-Type Massey-Omura Multiplier

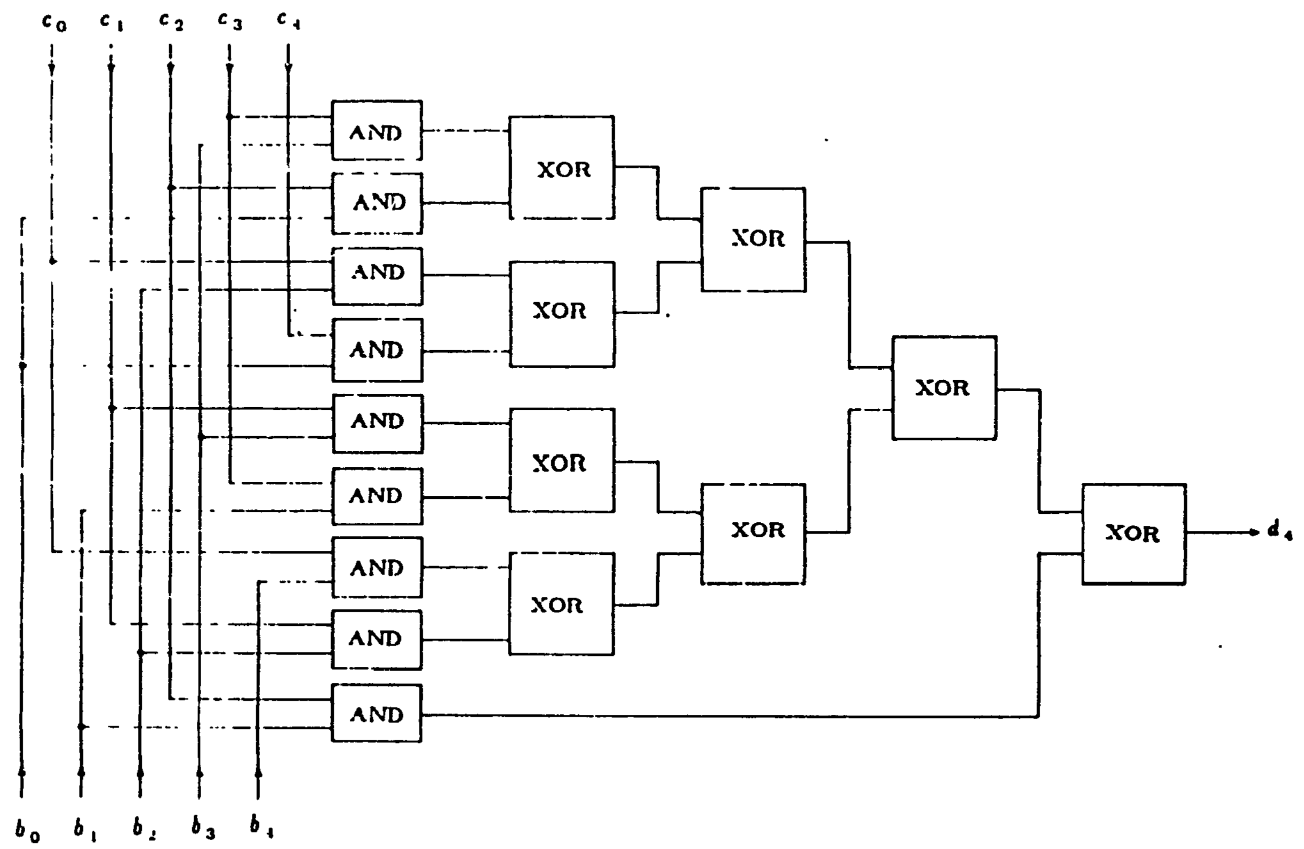


Figure 3.6: The Logic Design of the Least Complex Product Function for Massey-Omura Multiplier over GF(2^5)

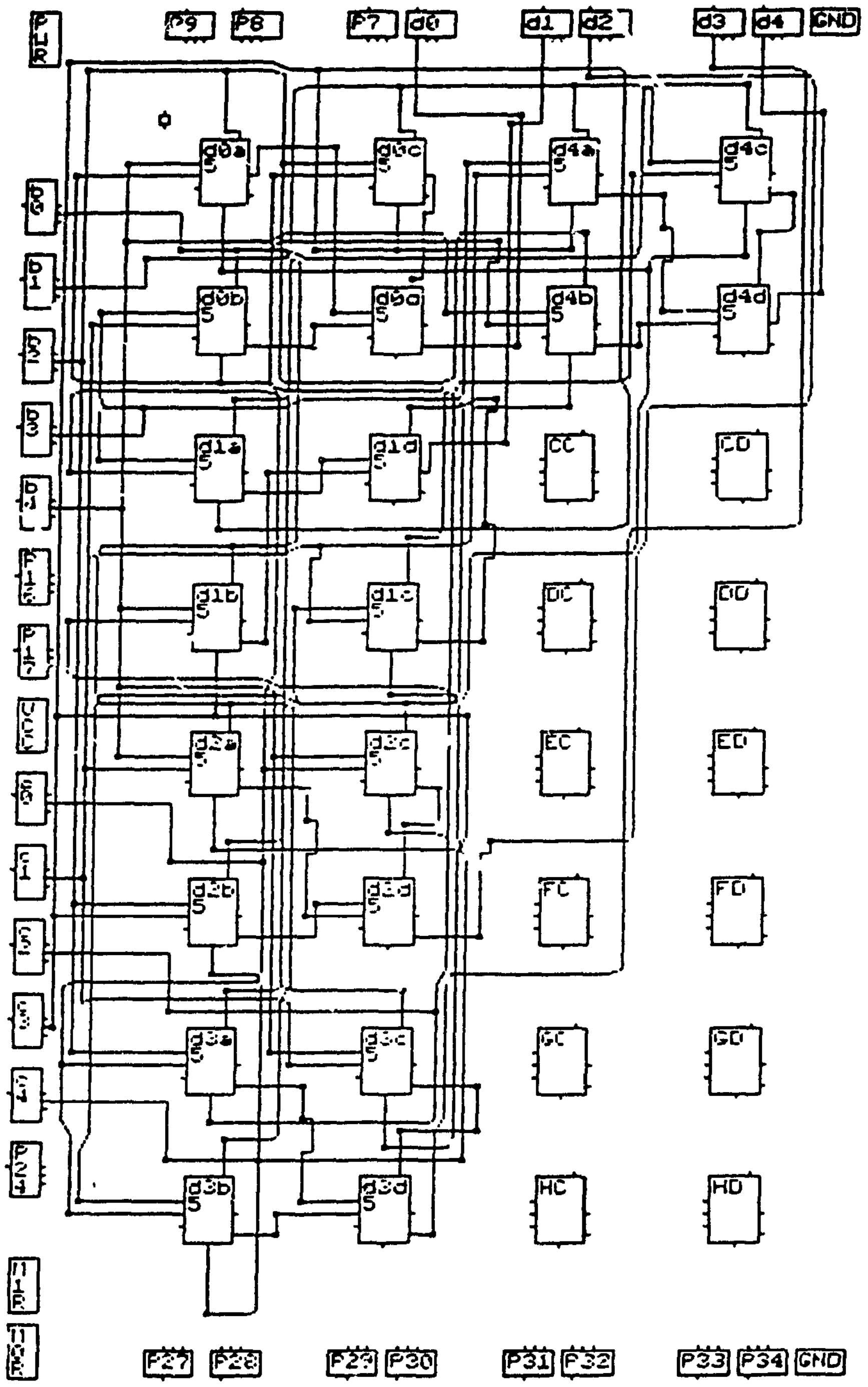


Figure 3.7: LCA Configuration of the Parallel-Type Massey-Omura Multiplier over GF(2^5)

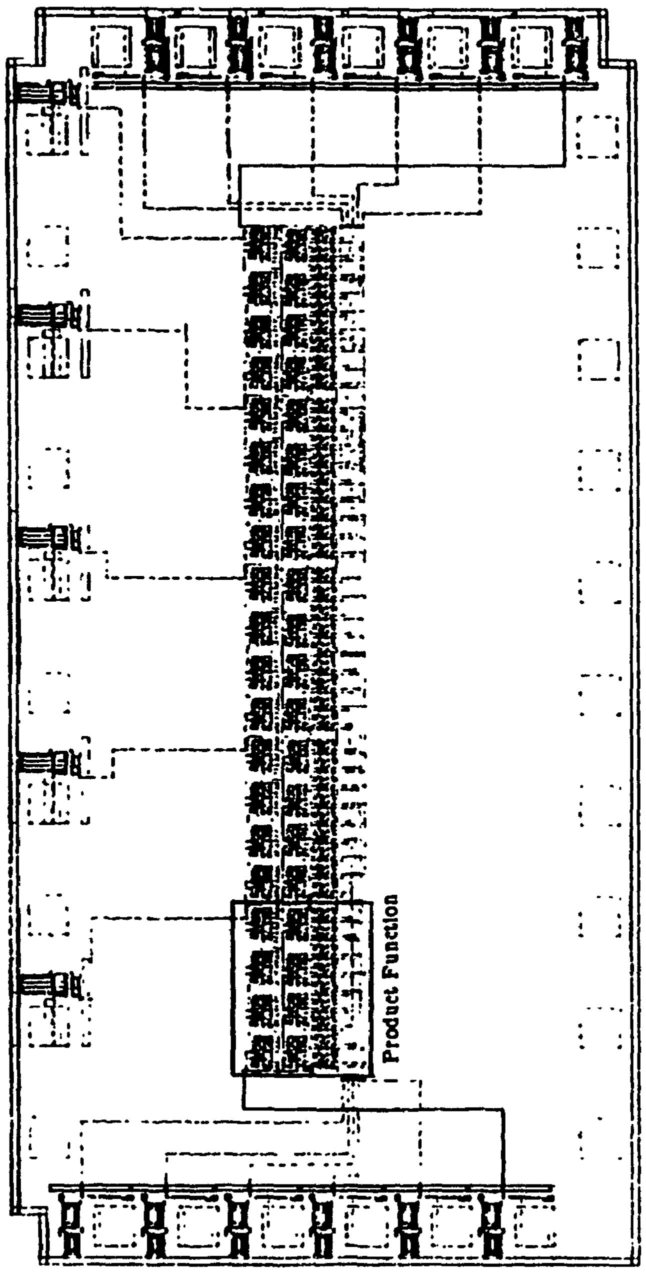


Figure 3.8: VLSI Layout of the Parallel-Type Massey-Omura Multiplier over $GF(2^5)$

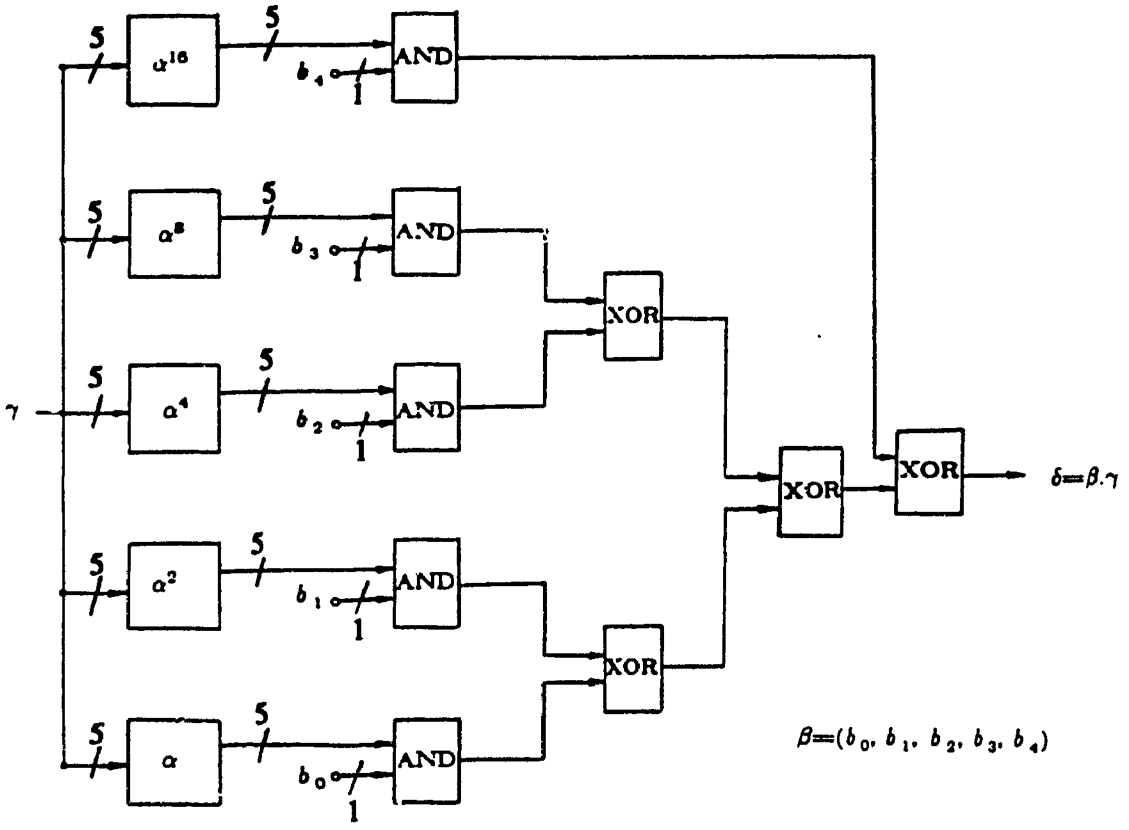


Figure 3.9: Parallel-in/Parallel-out GF(2^5) Multiplier

in Normal Basis

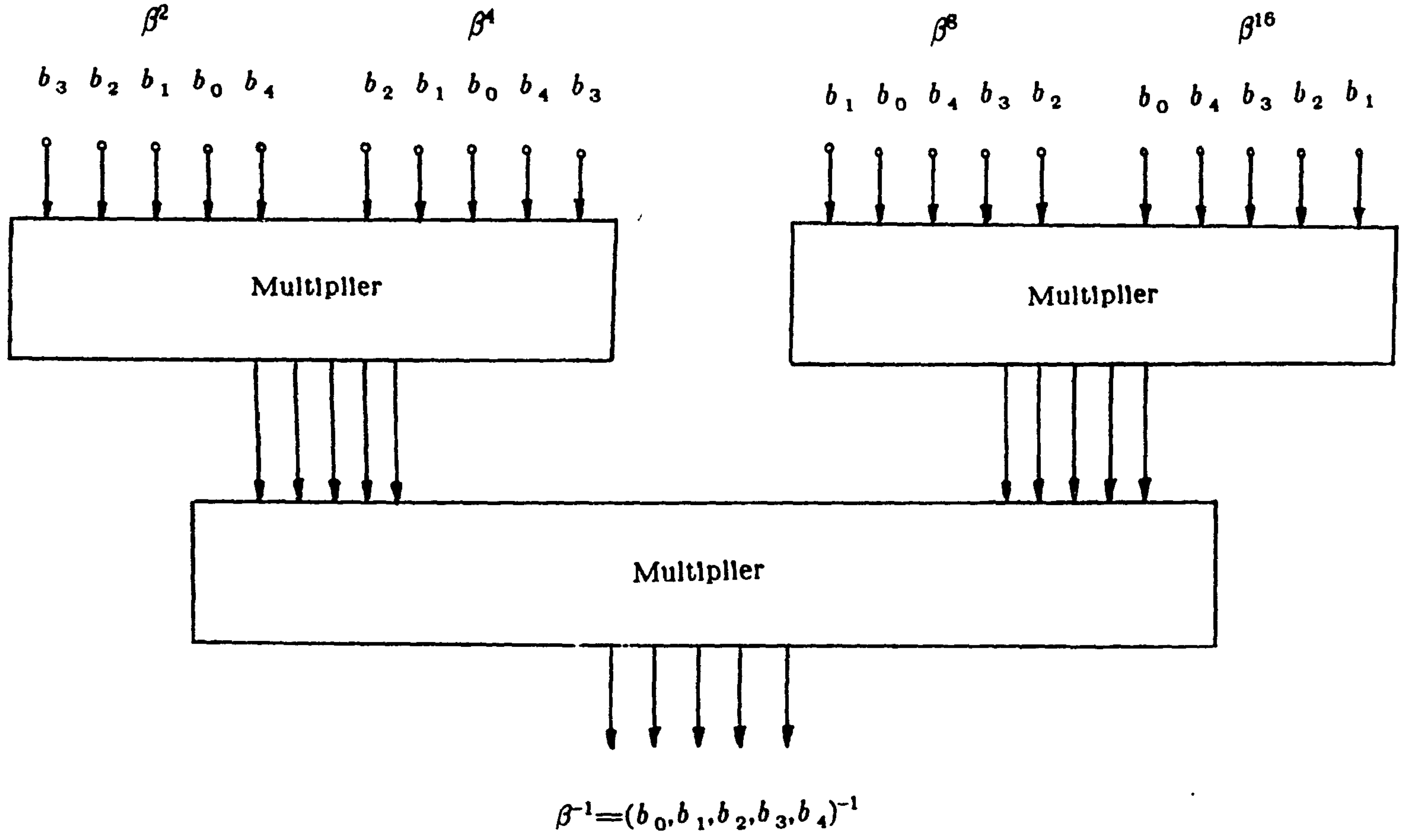

β^2
b_3 b_2 b_1 b_0 b_4
β^4
b_2 b_1 b_0 b_4 b_3
β^8
b_1 b_0 b_4 b_3 b_2
β^{16}
b_0 b_4 b_3 b_2 b_1
Multiplier
Multiplier
Multiplier
$\beta^{-1}=(b_0,b_1,b_2,b_3,b_4)^{-1}$

Figure 3.10: The Normal Basis Inverter in GF(2^5)

CHAPTER FOUR

A VERSATILE TIME DOMAIN REED-SOLOMON DECODER

In this chapter, the time-domain algorithm based on transform decoder [32] is restructured and a versatile Reed-Solomon (RS) decoder structure is developed [62]. This decoder can be programmed to decode any Reed-Solomon code defined in GF(2^m) with a fixed symbol size m. The reason for limiting ourselves to a specified GF(2^m) is that the universal multiplier in Galois fields has higher delay and complexity compared to the multipliers for one specified value of m. The versatile decoder can correct both errors and erasures for any RS code. It is shown that the introduced versatile decoder has a very simple structure and can be used to design high-speed, single-chip VLSI decoders.

The input/output interface and decoding unit of the decoder is given and the complexity and throughput is discussed. It is also shown that versatile decoders are best designed using time-domain decoding algorithms.

As an illustrative example, a Gate-Array-Based programmable RS decoder is implemented on a single chip [63]. This decoder chip can decode any RS code defined in GF (2^5) with any code word length and any number of information symbols. The fabrication of the decoder chip is done using low power 1.5 micron, 2-layer metal, high-speed CMOS technology.

4.1. Decoding Algorithm

The time-domain decoding algorithm based on transform decoding which was explained in Chapter 2 [32], may introduce a large number of extra errors, when the number of errors in the channel is larger than the error correction capability of the code. In this section, a block is added at the end of the time-domain decoding algorithm, to overcome this problem. Some other modifications are also given in order to simplify the structure of the decoder.

Consider the t error correcting time-domain Reed-Solomon decoder. When the number of errors introduced in the channel, ν, is larger than t, two cases can be distinguished.

In the first case, the pattern of the received vector $\underline{v}$ is such that the decoder assumes a code word has been received which contains $\mu \leq t$ errors $(\mu \neq \nu)$. Therefore, the decoder decodes the received vector by introducing at most μ errors. Hence the decoder output has at most $\mu+\nu$ errors.

In the second case, the received pattern to the decoder is such that the decoder assumes a code word has been received which contains $\mu > t$ errors. In this case the decoder tries to correct the received vector, but since this is out of the capability of the code, the decoder fails and can introduce up to n errors in the received vector. This happens even when the number of errors introduced in the channel is just above the error correction capability of the code, i.e., $\nu = t+1$. This case can be detected easily and the decoder can output the received vector as it is without introducing any extra error.

To detect the second case, one way is to count the zero components of the vector $\underline{\lambda}$. If this count is equal to the value of L (Chapter 2), the decoder performs the correction, and if not, the output is the received vector without correction, $\underline{c} = \underline{v}$ [16],[25]. For implementation of this method, a counter is required for counting the zero components of $\underline{\lambda}$.

To avoid this counter, consider the second step of the time-domain algorithm. In this step when the ith component of the errata locator vector $\underline{\lambda}$ is zero ($\lambda_i=0$), then there is an error in location i. Moreover in step 3 of the time-domain algorithm, when the ith component of the errata value vector $\underline{e}$ is zero ($e_i=0$) then the received vector $\underline{v}$ is not corrected in location i. In other words for each index of i, if the product $\lambda_i e_i \neq 0$ then we detect the second case explained above when $\nu > t$. As shown in the next sections, this comparison requires only one flip-flop and has less complexity compared to the previous method.

It is shown in [69] that malfunction may happen in the decoders where the Euclidean algorithm is used for calculation of the error location polynomial. This means that the decoder may have an output which is not a code word at all. In the time-domain algorithm based on transform decoder, it appears to be no malfunction, since we have used the Berlekamp-Massey algorithm.

In Fig. 4.1, other modifications are also provided to decrease the complexity or increase the speed of the decoder. As shown in this figure, after updating r, the block $s_i \leftarrow \alpha^i s_i$ is introduced [32]. This helps the equation for Δ_r not to have the factor in α, as does the equation for updating s_i. Since there are n iterations, the final s_i is multiplied by α^{ni}, but α^{ni} equals one since α has order n and, hence, the final result is not affected. This modification eliminates the calculation of α^{ir} in the algorithm.

In the algorithm of Fig. 4.1, the value of L is initialized to zero, λ_i and b_i, $i=0, \cdots, n-1$, are initialized to $\alpha^0=1$, and $\underline{s}=\underline{v}$.

After initialization, the first iteration starts and the components of $\underline{s}$ are updated to $s_i \alpha^i$. In the first ρ iterations, the components of the erasure locator vector $\underline{\gamma}$ is evaluated as,

$$\Delta=\alpha^{j_r}$$
$$\delta=0$$
$$\begin{bmatrix}\lambda_i\\ b_i\end{bmatrix}\leftarrow\begin{bmatrix}1 & -\Delta\alpha^{-i}\\ \Delta^{-1}\delta & (1-\delta)\alpha^{-i}\end{bmatrix}\begin{bmatrix}\lambda_i\\ b_i\end{bmatrix},$$
$$b_i=\lambda_i \tag{4.1}$$

where $\underline{\tau}=\underline{\lambda}$ at the end of ρth iteration. Eq. (4.1) is the approach to find the value of erasure locator vector with iterative equations similar to the Berlekamp-Massey iterations [32]. In the above equation, j_r for $r=1,2,\cdots,\rho$ are erasure locations. To use Berlekamp-Massey iterations to perform Eq. (4.1), initialization of $\Delta=\alpha^{j_r}$, $\delta=0$ is done in each iteration.

In iterations $r=\rho+1,\rho+2,\cdots,n-k$, the errata locator vector $\underline{\lambda}$ is evaluated using following set of recursive equations [32].

$$\Delta=\sum_{i=0}^{n-1}\lambda_i s_i$$
$$L\leftarrow\delta(r-L)+(1-\delta)L,$$
$$\begin{bmatrix}\lambda_i\\ b_i\end{bmatrix}\leftarrow\begin{bmatrix}1 & -\Delta\alpha^{-i}\\ \Delta^{-1}\delta & (1-\delta)\alpha^{-i}\end{bmatrix}\begin{bmatrix}\lambda_i\\ b_i\end{bmatrix}, \tag{4.2}$$

for $i=0,\cdots,n-1$, where $\delta=1$, if both $\Delta\neq0$ and $2L-\rho\leq r-1$, and $\delta=0$ otherwise. After the $(n-k)$th iteration, the estimate of the errata value vector $\underline{s}$ is calculated according to,

$$\Delta=\sum_{i=0}^{n-1}\lambda_i s_i,$$
$$s_i\leftarrow s_i-\Delta, \tag{4.3}$$

and finally, if $\lambda_i s_i=0$ for $i=0,\cdots,n-1$, the received vector $\underline{v}$ is corrected to

$$\underline{c}=\underline{v}-\underline{s}. \tag{4.4}$$

Note that in this algorithm, the vector $\underline{s}$, at the last iteration, is equivalent to the errata value vector $\underline{e}$.

4.2. Decoder Architecture

In this section, the architecture of the versatile time-domain decoder is presented [62]. The Reed-Solomon decoder has two units, namely, the Input/Output (I/O) Unit and the Decoding Unit. These two blocks are explained in this section assuming that the decoder receives the symbol v_{n-1} first and the symbol v_0 last. The output of the decoder is also in the same order, i. e., $d_{k-1}=c_{n-1}$ is transmitted first and $d_0=c_{n-k}$ last.

The structure discussed in this section can easily be modified to decode shortened, singly extended, decreased redundancy, and nonprimitive RS codes [32],[61],[65].

4.2.1. Input and Output Interface

The I/O Unit, shown in Fig. 4.2, accepts the received data ($\underline{v}$) at a symbol rate defined by Clk-In, and stores it in the buffer. The Decoding Unit corrects the erasures and errors of vector $\underline{v}$ up to its capability and transfers the corrected vector $\underline{c}$ to the I/O Unit symbol by symbol. The I/O Unit outputs the decoded data $\underline{c}$ at a symbol rate defined by Clk-Out.

The inputs and the outputs of the decoder are continuous, hence they can not be stopped or delayed. On the other hand, the decoder needs some time to correct the errors after receiving an entire input block. To let the decoder correct the erasures and errors without losing any data at the input or output of the decoder, three sets of n-stage parallel-in/parallel-out shift registers are used where $n=2^m-1$.

The I/O unit, upon receiving a code word, stores it symbol by symbol in one of these buffers and then the phase control connects this buffer to the decoder unit. When the decoder is correcting the errors another buffer receives the next code word and stores it. In the third phase, the buffer which holds the corrected code word outputs the data. The phase control is responsible for the operation of these buffers in each phase.

Note that we have assumed a systematic RS(n ,k) code and the information symbols of v_i and c_i are in locations with indexes $i = n-k, n-k+1, \cdots, n-1$ while the parity symbols are in locations $i=0, 1, \cdots, n-k-1$. Therefore after correction of the errors, the I/O Unit only needs to output the information symbols which are located at k locations at the right side of the output buffer. This is done automatically, since the output buffer is shifted out with a lower frequency clock compared to the Input-Buffer.

There is another input, ER , to the I/O unit which gives the erasure information about any received symbol. This input exhibits a low to high transition to denote that an erasure has occurred in the synchronously received symbol. If the received symbol v_i is an erasure then we need to store α^i and use it later. To generate α^i , a multiplier and the register R_{ER} are used. The register R_{ER} is initialized to $\alpha^0=1$ and the clock for this register is Clk-In. The erasure information is stored in n-stage parallel-in parallel-out shift registers. Obviously two sets of these n-stage shift registers should be used, one for the input and one for the work buffer. The phase control circuitry changes the role of these buffers consecutively. An up-counter finds the number of erasures, ρ, to be used in the Decoding Unit.

4.2.2. Decoding Unit

The Decoding Unit structure is shown in Fig. 4.3. In the Decoding Unit, there are three sets of n-stage parallel-in/parallel-out shift registers for storing s_i, λ_i, and b_i, for $i=0, 1, \cdots, n-1$. The contents of these registers are shifted to the Arithmetic Unit with clk_i which is the internal clock of the system. The Arithmetic Unit processes the contents of these registers and feeds them back circularly. To distinguish between the different iterations of the decoding process, the internal clock is divided by n to form clk_r which is fed to the iteration counter. Output of this counter indicates the iteration number r. There are $n+1$ iterations, and in each iteration, r determines the commands given to the Arithmetic Unit by the control circuitry.

In this design, three n-stage shift registers are shifted to the right with clk_i clock. This shifting is performed till the end of decoding process. The switches are connected based on the value of the iteration counter, r. The connection of these switches is shown on the contacts by specifying value of r.

In iteration $r=0$, the content of the work buffer, v_i, is multiplied by α^i and loaded in to the shift register s_i. At the same time, $v_i \alpha^i$ is accumulated in Δ_i register to form Δ for the first iteration. In this iteration, shift registers λ_i and b_i are loaded with the initial value of $\alpha^0=1$. In order to keep the data of the work buffer, values of v_i are also fed back to the work buffer with clk_i. In this iteration, L is initialized to zero.

In iterations 1 to ρ, the registers λ_i and b_i are initialized by the erasure locator vector. In iterations $\rho+1$ to $n-k$, the value of δ and updated value of L are calculated based on the value of iteration counter r, discrepancy Δ and previous value of L. In these iterations, λ_i, b_i and Δ are calculated simultaneously. In iterations $n-k+1$ up to $n-1$, value of s_i is added to Δ and simultaneously the new value of Δ is calculated. In these iterations λ_i is also shifted circularly to maintain the contents.

In the last, i.e., the n th, iteration, s_i is added to the previous value of Δ to be used for correcting v_i. For proper correction, the values of $\lambda_i s_i$ are compared with zero, and if $\lambda_i s_i=0$, then the corresponding component of $\underline{v}$ is corrected. If $\lambda_i s_i \neq 0$, then the component v_i and remaining components are directed to the work buffer without correction. This is done by an m-bit OR gate and a JK flip-flop.

In this design, two similar circuitries are used to calculate values of α^i and α^{-i}. The registers of these two circuitries are initialized at the beginning of each iteration to α^0 with Clk_i clock.

4.3. Complexity and Throughput

In this section, the complexity and throughput of the versatile decoder is discussed for different Galois field sizes.

In the decoding unit, seven multipliers and one inverter in GF(2^m) is used, where two of the multipliers have a constant multiplicand, and, therefore, have lower complexity. Since, the standard basis inversion in GF(2^m) requires a large number of gates or ROM as discussed in Chapter 3, we use normal basis arithmetic. Note that this choice is justified when there are not many multipliers in the design. As discussed in Chapter 3, the Massey-Omura multiplier is not used, since it has higher complexity compared to the other normal basis multiplier.

As shown in Fig. 4.3, the decoding unit uses , three n-stage parallel-in/parallel-out shift registers ($n=2^m-1$) where each register consists of m-bit symbols. Table 4.1 shows the number of gates needed for different parts of the decoding unit in GF(2^m), for $m=4,5,6,7,8$.

As shown in Fig. 4.2, the I/O unit uses 5, n-stage parallel-in/parallel-out shift registers ($n=2^m-1$) which contain m-bit symbols. These registers are used as buffers for storing input, output and erasure information. Table 4.2, gives the number of gates needed for the I/O unit for different Galois fields.

Table 4.1: Number of Gates Needed for the Decoding Unit of the RS Decoder in GF(2^m)

m	Buffers	Arithmetic & Control	Total
4	1100	700	1800
5	2800	1000	3800
6	6200	1400	7600
7	16100	1900	18000
8	36700	2500	39200

Table 4.2: Number of Gates Needed for the I/O Unit of the RS Decoder in GF(2^m)

m	Buffers	Arithmetic & Control	Total
4	2200	600	2800
5	5600	700	6300
6	12400	800	13200
7	32200	900	33100
8	73400	1000	74400

Considering Tables 4.1 and 4.2, we note that the number of gates for the buffers is much more than the number of gates for the control and arithmetic section of both I/O unit and the decoding unit.

All the buffers in the decoder are very easy to design, since they have modular structure and are similar to each other. The irregular parts of the design are the arithmetic and control sections. However, these sections are very small compared to the buffer sections, and, therefore, their design time is very short. For hardware design, this property will decrease the design and implementation cost by a large factor.

The decoding time of the decoder is determined by the longest delay path. This path has a delay τ equivalent to delay of 19 gates. The decoding algorithm needs $n=2^m-1$ iterations and one extra iteration for initialization. In each iteration the shift registers are shifted $n=2^m-1$ times to the right, so one iteration period is $T=n\tau$ and the decoding time is $n(n+1)\tau$. The decoding time can be used for evaluating the maximum input bit rate of the decoder, $m/(2^m)\tau$. Considering an internal clock of 20 MHz (τ=50 ns), the maximum bit rates at the input of the decoder for GF(2^m) are given in Table 4.3.

Table 4.3: Maximum Bit-Rate of the RS Decoder

m	Bit-Rate [Mb/s]
4	4.9
5	3.1
6	1.8
7	1.1
8	0.6

4.4. VLSI Gate-Array-Based Design

As shown in the previous section the buffer sections of the decoder use most of the gates needed for the decoder design. Since the buffer structure is cellular and regular, the design and layout time for VLSI implementation of these sections becomes very low. The control and arithmetic sections of the decoder are not cellular. However, they are very simple and require low number of gates, and, hence are very easy to be designed and implemented.

These features make this decoder structure suitable for VLSI design. One other reason for the suitability for VLSI design is the versatility of the RS decoder structure. In fact, because of high cost of fabrication, general purpose chips such as this decoder should be considered for VLSI design.

As an illustrative example, a Gate-Array-Based programmable RS decoder is implemented on a single chip [63]. The decoder chip operates in a 5-bit symbol Galois field, GF (2^5). For the fabrication of the decoder chip, the low-power, 1.5 micron, 2-layer metal high-speed CMOS technology is used. The decoder chip can decode any RS code defined in GF(2^5). The power dissipation of the chip is 0.6 watts in worst case and the design needs 20500 gates on a 68-pin package. One of the reasons for using more gates in this design compared to tables 4.1 and 4.2 is the restrictions available in the gate array design [60]. The other reason is the addition of some more features to the versatile RS decoder, such as microprocessor interface circuits, compared to figures 4.2 and 4.3.

The Gate-Array-Based RS decoder chip can be used in two modes: stand-alone and microprocessor-peripheral. In stand-alone mode, it operates as a stand-alone device, as shown in figures 4.2 and 4.3, directly in the data symbol stream. In this mode, it does not require a processor or any external buffer. In microprocessor-peripheral mode, it can be directly interfaced to any standard

4.4. VLSI Gate-Array-Based Design

As shown in the previous section the buffer sections of the decoder use most of the gates needed for the decoder design. Since the buffer structure is cellular and regular, the design and layout time for VLSI implementation of these sections becomes very low. The control and arithmetic sections of the decoder are not cellular. However, they are very simple and require low number of gates, and, hence are very easy to be designed and implemented.

These features make this decoder structure suitable for VLSI design. One other reason for the suitability for VLSI design is the versatility of the RS decoder structure. In fact. because of high cost of fabrication, general purpose chips such as this decoder should be considered for VLSI design.

As an illustrative example, a Gate-Array-Based programmable RS decoder is implemented on a single chip [63]. The decoder chip operates in a 5-bit symbol Galois field, GF (2^5). For the fabrication of the decoder chip, the low-power, 1.5 micron, 2-layer metal high-speed CMOS technology is used. The decoder chip can decode any RS code defined in GF(2^5). The power dissipation of the chip is 0.6 watts in worst case and the design needs 20500 gates on a 68-pin package. One of the reasons for using more gates in this design compared to tables 4.1 and 4.2 is the restrictions available in the gate array design [60]. The other reason is the addition of some more features to the versatile RS decoder, such as microprocessor interface circuits, compared to figures 4.2 and 4.3.

The Gate-Array-Based RS decoder chip can be used in two modes: stand-alone and microprocessor-peripheral. In stand-alone mode, it operates as a stand-alone device, as shown in figures 4.2 and 4.3, directly in the data symbol stream. In this mode, it does not require a processor or any external buffer. In microprocessor-peripheral mode, it can be directly interfaced to any standard microprocessor bus. This configuration is useful for applications in magnetic recording systems, data communications as well as digital-signal-processor based modems. In this mode the decoder chip operates as a standard peripheral device of a microprocessor or a digital signal processor.

4.5. Versatile Decoders Based on Other Algorithms

To introduce versatile RS decoders using syndrome-based decoding algorithms, many modules should be designed. For example, let's consider the module for the Berlekamp-Massey algorithm to evaluate the error locator polynomial $\Lambda(x)$. We have chosen this module since, as will be shown, the design is similar to the design of the decoding unit in Fig. 4.2. This module is the same for both syndrome-based decoding algorithms.

As we know the Berlekamp-Massey algorithm can be designed using shift registers as shown in Fig. 4.4. [16],[25]. In this figure, registers are provided for three polynomials $S(x)$, $\Lambda(x)$, and $B(x)$, and the length of each register is large enough to hold the largest degree of its polynomial. Shorter polynomials are stored simply by filling out the register with zeros. The $S(x)$ and $B(x)$ registers are each one stage longer than needed to store the largest polynomial. This, and the number of clocks applied to each shift register during an iteration, are set up so that the polynomials will process one position during each iteration. This supplies the multiplication of $B(x)$ by x and also offsets the index of S_j by r to provide S_{j-r}, which appears in the expression for Δ. To see this, refer to Fig. 4.4; the syndrome register is shown as it is initialized. During each iteration, it will be shifted to the right by one symbol so that it will be timed properly for multiplication with Λ. During one iteration, the Λ register is cycled twice, first to compute Δ, then to be updated.

This structure can compute the error locator polynomial for the case when only errors are available. To change the design to correct both errors and erasures, the $B(x)$, $\Lambda(x)$, and $S(x)$ shift registers should have $n-k+2$, $n-k+1$, and $n-k+1$ stages, respectively. For a versatile decoder with variable error correction capability, value of k should be variable $(1 \leq k < n)$. This means that the shift registers should have $n+1$, n, and n stages.

Now, we see the similarity of the Berlekamp-Massey module with the decoder of Fig. 4.3. They both have 3 sets of about n stage shift registers and hence, we can design the module of Fig. 4.4 in the same manner as Fig. 4.3.

Obviously the Berlekamp-Massey module has less arithmetic compared to the structure of Fig. 4.3. In Fig. 4.4., we need only three multipliers and one inverter but in Fig. 4.3 there is a need for seven multipliers and an inverter.

The main problem with the design of Berlekamp-Massey module for a versatile decoder is the structure of the shift registers. For different values of k, different number of these shift registers, $n-k$, should be shifted circularly. There are two ways to solve this problem:

1- To design a complex control circuitry to fill in zeros in the remaining registers which are not used in the decoding of that particular code.

2- To design a complex structure for the shift registers, such that by choosing values of k, the data is shifted circularly in $n-k$ registers only.

Both of these methods increase the design time and also the complexity of the versatile Berlekamp-Massey module.

We see that just the module for the Berlekamp-Massey algorithm in Syndrome-Based algorithms, is more complicated than the decoding unit of Fig. 4.3 which is the whole versatile decoder in the time-domain. Moreover, to design a versatile RS decoder using syndrome based algorithms, other modules should also be designed. Considering the above discussion, we note that the versatile time-domain decoder of Fig. 4.3 is less complex than the versatile decoders using syndrome-based decoding algorithms.

Now, let's consider two time-domain decoding algorithms and compare them. As previously mentioned, the time-domain algorithm based on the transform decoder needs 3 sets of n stage shift registers. The number of these shift registers should be doubled in the time-domain algorithm based on algebraic decoder to store values of $\lambda, \lambda', \underline{\omega}, \underline{b}, \underline{b}', \underline{a}$ vectors. The arithmetic and control units of the time-domain algorithm based on algebraic decoder is almost twice complex. Therefore, we can say that the time-domain algorithm based on algebraic decoder has twice the complexity of the other time-domain algorithm.

The time-domain algorithm based on the transform decoder should shift the registers n times circularly in each iteration. The number of iterations is n, and therefore this decoder can decode with n^2 clocks. The time-domain algorithm based on the algebraic decoder, has the same number of shifts in each iteration, n. However, this algorithm only needs $2t$ iterations to decode a t error correcting RS code. Therefore this decoder can decode with $2tn$ clocks which is much less than that of the other time-domain decoding algorithm, specially for high rate codes.

4.6. Summary

In this chapter, a versatile Reed-Solomon decoder, capable of decoding any RS(n,k) code in a specific Galois field was presented. The time-domain decoding algorithm based on transform decoder was used to introduce the decoder. The structure of the versatile decoder is such that it can be programmed to correct errors and erasures for different RS codes. As an illustrative example, a Gate-Array-Based design of the decoder in GF(2^5) was also given.

To compare the syndrome-based algorithms and time-domain ones, implementation of the Berlekamp-Massey algorithm was considered. The

Berlekamp-Massey algorithm is one of the steps of the syndrome-based algorithms which evaluates the error locator polynomial. It was shown that the hardware required for the implementation of this step alone, is more than that of the total structure of the introduced versatile decoder based on time-domain algorithms.

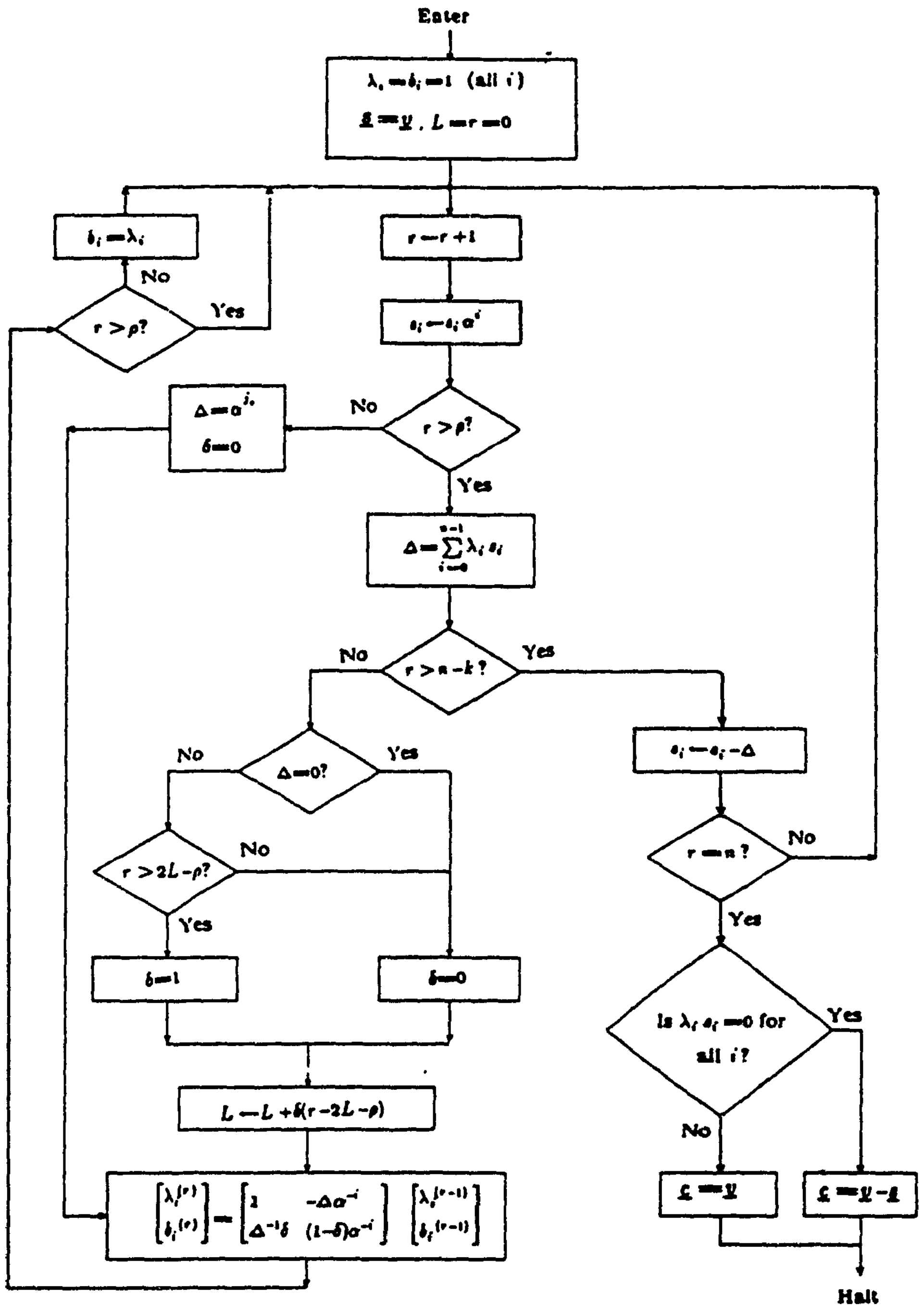

Enter
$\lambda_i = b_i = 1$ (all i)
$\underline{s} = \underline{y}$, $L = r = 0$
$b_i = \lambda_i$
No
$r > \rho$?
Yes
$r \leftarrow r+1$
$s_i \leftarrow s_i \alpha^i$
$\Delta = \alpha^{i_0}$
$\delta = 0$
No
$r > \rho$?
Yes
$\Delta = \sum_{i=0}^{n-1} \lambda_i s_i$
No
$r > n-k$?
Yes
No
$\Delta = 0$?
Yes
$s_i \leftarrow s_i - \Delta$
$r > 2L-\rho$?
No
Yes
$r = n$?
No
Yes
$\delta = 1$
$\delta = 0$
Is $\lambda_i s_i = 0$ for all i?
Yes
No
$L \leftarrow L + \delta(r-2L-\rho)$
$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta\alpha^{-i} \\ \Delta^{-1}\delta & (1-\delta)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \end{bmatrix}$
$\underline{c} = \underline{y}$
$\underline{c} = \underline{y} - \underline{s}$
Halt

Figure 4.1: Time-Domain Decoding Algorithm Based on Transform Decoder

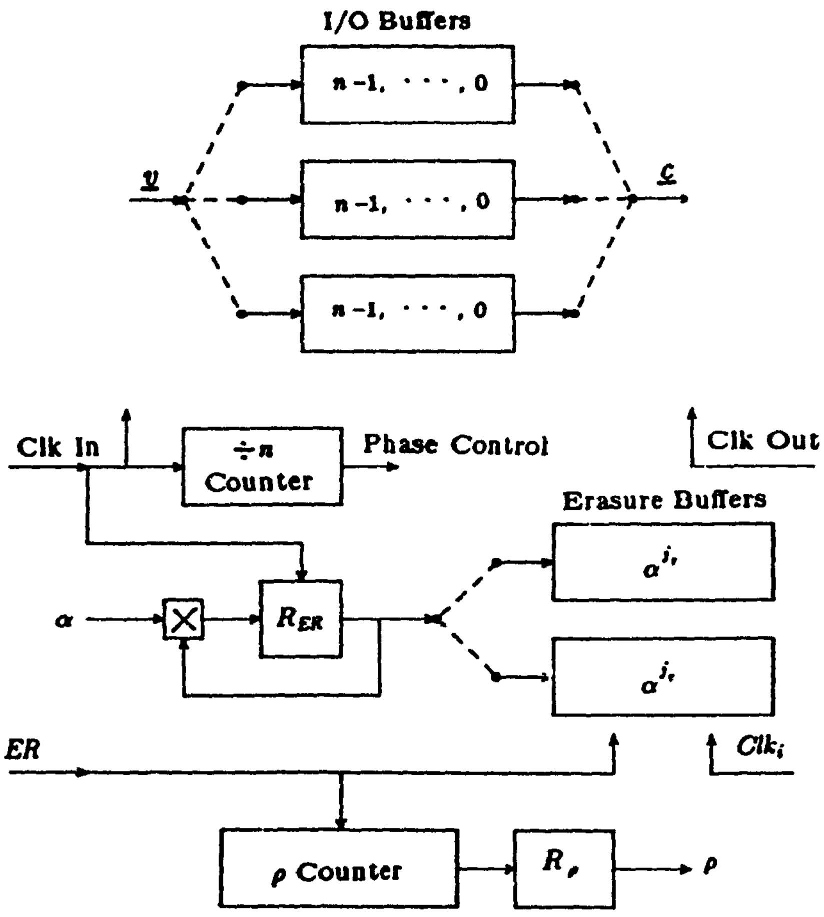


Figure 4.2: Input/Output Unit of the Versatile Decoder

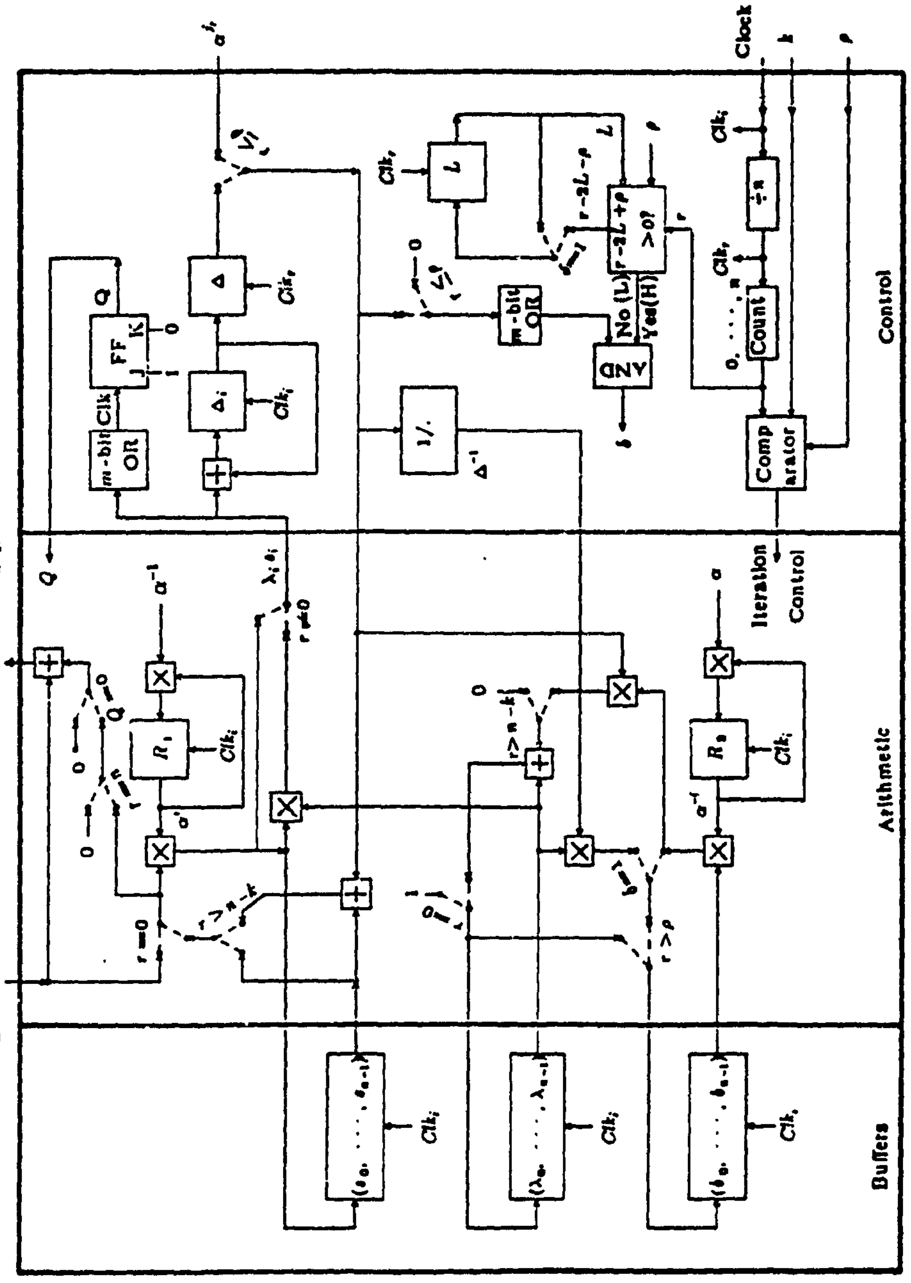


Figure 4.3: Decoding Unit of the Versatile Decoder

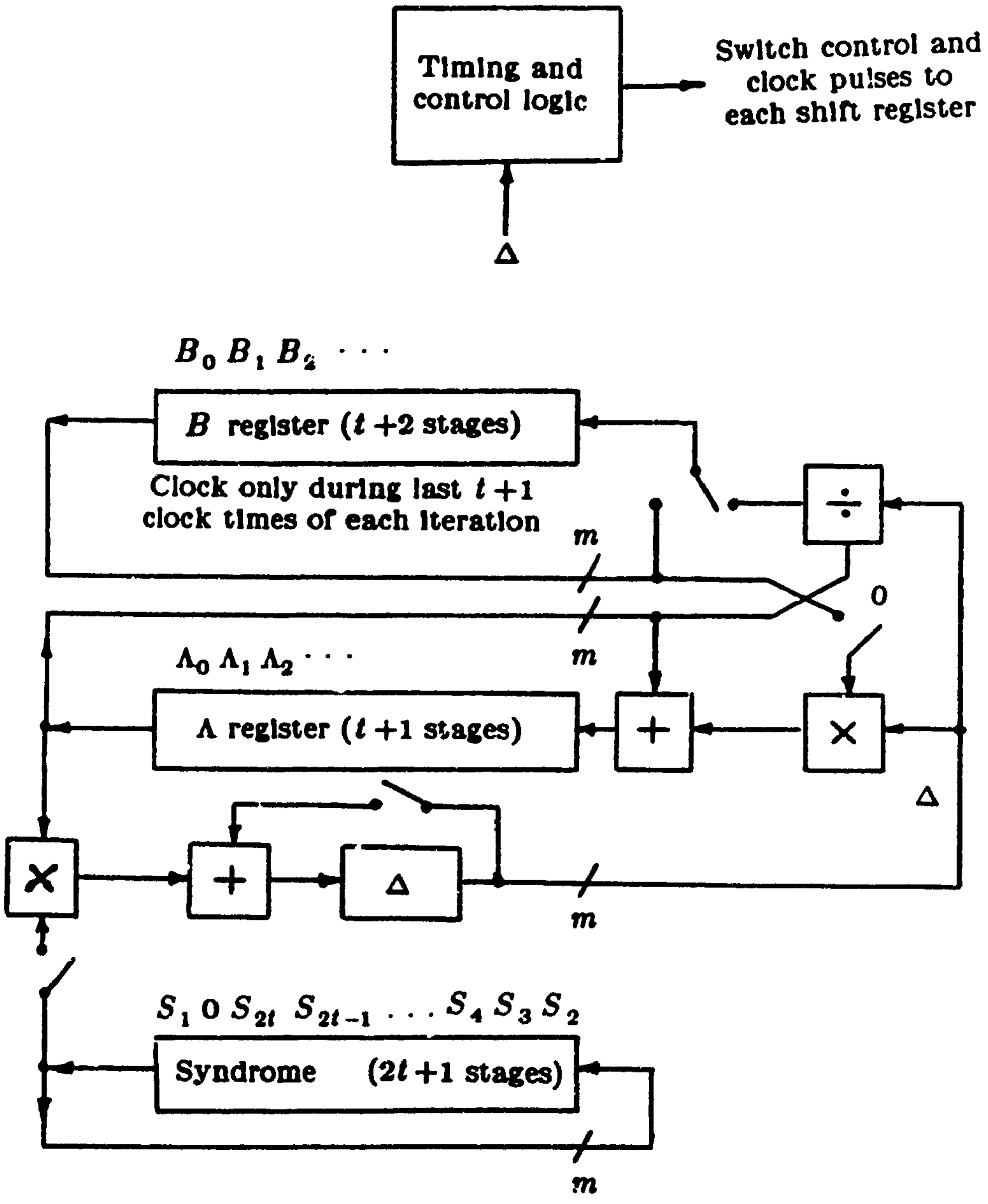

Timing and control logic
Switch control and clock pulses to each shift register
Δ
$B_0\, B_1\, B_2 \cdots$
B register ($t+2$ stages)
Clock only during last $t+1$ clock times of each iteration
m
$\div$
0
m
$\Lambda_0\, \Lambda_1\, \Lambda_2 \cdots$
m
Λ register ($t+1$ stages)
+
×
Δ
×
+
Δ
m
$S_1\, 0\, S_{2t}\, S_{2t-1} \ldots S_4\, S_3\, S_2$
Syndrome ($2t+1$ stages)
m

Figure 4.4: Berlekamp-Massey Module for Evaluation

of the Error-Locator Polynomial [25]

CHAPTER FIVE

A CELLULAR STRUCTURE FOR A VERSATILE REED-SOLOMON DECODER

The time-domain decoding algorithm based on the transform decoder is restructured to be suitable for cellular implementation. This cellular decoder can be programmed to decode any (n,k) RS code defined in the Galois field, GF(2^m), with a fixed block length n and a fixed symbol size m. The versatile decoder can correct both errors and erasures for any message length k. The introduced decoder is cellular and has a very simple structure and, hence, it is suitable for VLSI designs. The complexity and throughput of the decoder are explained. It is shown that the introduced structure has a very high decoding speed.

5.1. Decoding Algorithm

To design a cellular structure for the Reed-Solomon decoder, some modifications and restructurings should be introduced in the time-domain decoding algorithm. These modifications mainly reduce the complexity of the decoder. As shown in Fig. 4.1, there are three steps in the time-domain decoding algorithm for finding the errata locator and value vectors. The operations in these three steps are different, but, there are some similarities and all together n iterations are needed to perform the decoding. Our main objective is to combine these three steps and obtain an algorithm which has n iterations with the same operations in each iteration.

First step which has ρ iterations, is the time-domain erasure locator vector calculation. In this step, there is only one vector $\underline{\gamma}$, which is updated for ρ iterations.

In the second step, the Berlekamp-Massey algorithm is performed to find the errata locator vector $\underline{\lambda}$. In this step vectors $\underline{\lambda}$ and $\underline{b}$ are initialized with the result of the first step, $\underline{\gamma}$, and updated in each iteration and after $n-k-\rho$ iterations $\underline{\lambda}=\underline{\lambda}^{(n-k)}$ is the errata locator vector.

In the third step which has k iterations, the errata value vector is calculated. In this step, the vector $\underline{\lambda}$ is fixed and has the final value of the second step of the algorithm. But this time the value of another vector $\underline{s}$ is initialized with the received noisy vector $\underline{v}$ and updated in each iteration to correct $\underline{v}$ in the n th iteration.

To combine these three steps, two control variables are added to the algorithm to differentiate between three steps of the time-domain decoding algorithm. These variables, σ, and β, are equal to one in the first ρ iterations. In iterations $\rho+1, \cdots, n-k$, σ is zero, and β is one. In the last k iterations, both σ, and β are zero.

The restructured time-domain decoding algorithm, having the same operations in all n iterations, is as follows. Let $\underline{v}$ be the received noisy Reed-Solomon code word with erasures at locations j_r, $r=1, \cdots, \rho$. The following set of recursive equations can be used to compute $e_i^{(n)}$ for $i=0, 1, \cdots, n-1$:

$$\Delta=\beta\alpha^{j_r}+(1-\beta)\sum_{i=0}^{n-1}\lambda_i^{(r-1)}s_i^{(r-1)}, \tag{5.1}$$

$$L \leftarrow \delta(r-L-\rho)+(1-\delta)L\,, \tag{5.2}$$

$$\begin{bmatrix}\lambda_i^{(r)}\\ b_i^{(r)}\end{bmatrix}=\begin{bmatrix}1 & \sigma\Delta\alpha^{-i}\\ \delta\Delta^{-1} & (1-\delta)\alpha^{-i}\end{bmatrix}\begin{bmatrix}\lambda_i^{(r-1)}\\ b_i^{(r-1)}\end{bmatrix}, \tag{5.3}$$

$$e_i^{(r)}=s_i^{(r-1)}+(1-\sigma)\Delta, \tag{5.4}$$

$$s_i^{(r)}=\alpha^i\, e_i^{(r)}, \tag{5.5}$$

$$b_i^{(r)}\leftarrow\beta\lambda_i^{(r)}+(1-\beta)b_i^{(r)} \tag{5.6}$$

for $i=0, \cdots, n-1$ and for $r=1, 2, \cdots, n$. The initial conditions are $s_i^{(0)}=\alpha^i v_i$, $\lambda_i^{(0)}=b_i^{(0)}=\alpha^0=1$ and $L=0$. If both $\Delta\neq 0$ and $2L-\rho>r$, then $\delta=1$, and $\delta=0$ otherwise. Finally, after the nth iteration, the errata value vector has components equal to $e_i^{(n)}$, and the received vector $\underline{v}$ is corrected using the errata value vector $\underline{e}$, with components,

$$\begin{aligned} e_i &= e_i^{(n)}, && \text{if } \lambda_i^{(n)}=0,\\ e_i &= 0, && \text{if } \lambda_i^{(n)}\neq 0. \end{aligned} \tag{5.7}$$

The flow diagram of the restructured time-domain decoding algorithm is shown in Fig. 5.1. In this algorithm, first the initialization is performed. The iterations start by incrementing the iteration counter r and calculating the discrepancy Δ. Then, the control variables β, and σ are found based on the iteration counter r and the value of Δ is updated. After evaluation of δ and L. values of $\lambda_i^{(r)}$, $b_i^{(r)}$, $e_i^{(r)}$ and $s_i^{(r)}$ are calculated for $i=0, 1, \cdots, n-1$. Finally, after n iterations, the errata-value vector $\underline{e}$ is found. The components of this vector are equal to $e_i^{(n-1)}$ if $\lambda_i^{(n)}$ is zero and are zero, otherwise.

In the next section, this restructured algorithm is used to introduce a cellular structure for Reed-Solomon decoders. The part of the algorithm inside the dashed box forms the arithmetic and control unit of the decoder. Note that the part for evaluation of Δ and the variables λ_i, b_i, e_i, and s_i are not implicitly dependent on the value of iteration counter, r. This point will help in introducing a simple structure for the cells of the Reed-Solomon decoder.

5.2. The Cellular Structure

In this section, a cellular structure for Reed-Solomon decoders is introduced. This structure is based on the restructured time-domain decoding algorithm shown in Fig. 5.1. The introduced decoder can decode any (n,k) RS code with fixed code word length $n = 2^m - 1$ and programmable message length k. The structure of the decoder is shown in Fig. 5.2. This decoder consists of three sets of n identical cells, a Decoding-Control, and an Exponentiation cell.

The Input/output (I/O) cells receive the input to the RS decoder symbol by symbol with a symbol rate of Clk-In and provide each received component v_i to the ith Decoding Cell. The Decoding Cells evaluate the errata value vector and apply the ith component of this vector e_i to the ith I/O cell for all i. Then the ith I/O cell corrects the received symbol v_i by adding the errata symbol e_i. After correction the message part of the corrected code vector is sent to the output of the decoder with symbol rate of Clk-Out.

As the received vector is being stored in the I/O cells, the erasure information ER is passed through the Exponentiation cell and stored in the Erasure cells. After receiving the whole block of the received vector, the number of erasures ρ and values of α^{j_r}, $r = 1, \cdots, \rho$ are available to the Decoding control cell.

The Decoding cells are responsible for all the functions outside the dashed box of Fig. 5.1. These cells calculate the components of the errata value vector in n iterations using Clk-In. The Decoding Control circuitry controls the function of the Decoding cells in each iteration based on the erasure information, the discrepancy Δ_0 and the number of information symbols k.

5.2.1. Input/Output Cell

Three sets of registers are considered in the I/O cells as shown in Fig. 5.3. The in_i registers receive the input block and at the same time the out_i registers transmit the decoded block to the output of the decoder. The third set of the registers dec_i keeps the previously received block to be decoded by the Decoding cells.

To explain the function of the I/O cells let T be the time interval for receiving one block of data (n symbols), and also assume that the decoder receives the symbol v_{n-1} first and the symbol v_0 last. The output of the decoder is also in the same order which means that $d_{k-1}=c_{n-1}$ is transmitted first and $d_0=c_{n-k}$ last.

Fig. 5.4 shows the I/O cycle flow for receiving, decoding and transmitting the data. Let assume at time $t=0$ the decoder receives the first symbol of the first block of data. At time interval $0 \leq t < T$ the I/O cells receives the data symbol by symbol with a symbol rate of Clk-In and shifts them to the right I/O cells. At the end of the first time interval the in_i registers have stored the first block of data.

In the second time interval by receiving the first symbol of the second block of data ($t=T$), the previously received block is stored in the dec_i registers and the decoding process of the first block of data starts. In this interval ($T \leq t < 2T$), the input registers receive the second block of data and at the same time the Decoding cells evaluate the errata values e_i. Then, the first data block which was stored in dec_i registers is corrected and stored in the out_i registers by receiving the first symbol of the third data block ($t=2T$).

In Fig. 5.3, η is a control variable which is equal to 1 after receiving the last symbol of each data block and before receiving the first symbol of the next data block and is equal to zero otherwise. This variable is formed in the Decoding Control cell which will be explained later.

During the time interval $2T \leq t < 3T$ three functions are performed in parallel. The third data block is received and stored in in_i registers. The second data block is decoded and the first data block is transmitted to the output of the decoder. Note that we have assumed a systematic RS(n,k) code and the information symbols of v_i and c_i are in locations with indices $i = n-1, n-2, \cdots, n-k$ while the parity symbols are in locations $i = n-k-1, n-k-2, \cdots, 0$. Therefore, after correction of the errors, the I/O Unit only needs to output the information symbols which are stored in out_i registers located at k rightmost I/O cells. This is done automatically, since the output buffer is shifted out with a lower frequency clock (Clk-Out) compared to the input clock, Clk-In.

The function of the I/O unit after the third time interval is similar and is shown in Fig. 5.4.

5.2.2. Erasure and Exponentiation Cells

There is an input, ER, to the decoder which gives the erasure information about any received symbol. If the received symbol v_i is an erasure then we need to store α^i. To generate α^i, a multiplier and the m-bit register R is used as shown in Fig. 5.5. The register R is initialized to α^{-1} before receiving the first symbol of the data block (η=1). The clock of this register is Clk-In therefore after receiving the symbol v_i the output of this m-bit register has value of α^i for $i = 0, 1, \cdots, n-1$.

The erasure information is stored in n m-bit registers ein_i which are located in the Erasure cells of Fig. 5.6. The clock of these registers is the ER signal, and therefore only the values of α^i for which the symbol v_i is erased are stored in these registers, α^{j_r}, $r=1, \cdots, \rho$. Obviously, another set of m-bit registers, as the ones in the I/O cells are required. These registers, $eout_i$, are used for storing the values of α^{j_i} for the previously received data block. In each time interval T, the ein_i registers receive the values of α^{j_r} and store them by shifting to the right. Then after receiving the complete data block, contents of the ein_i registers are transferred to the $eout_i$ registers. In the next time interval T, the $eout_i$ registers are shifted to the left by Clk-In to be used by the Decoding Control cell. During the same time interval, the ein_i registers store the erasure information of the next data block.

In Fig. 5.5, the erasure counter is responsible for finding the number of the erasures in each data block and storing them in the m-bit register R_ρ. The output of the R_ρ register is available to the Decoding Control cell.

5.2.3. Decoding-Control Cell

The Decoding-Control cell is shown in Fig. 5.7. The algorithm of this unit is based on the dashed box in Fig. 5.1.

The decoder has a Reset input which becomes high to enable the RS decoder. This input is only applied to the iteration counter and is not propagated to any other cell of the decoder. The iteration counter which is a divide by n counter is activated by Clk-In. The output of this counter r indicates the iteration number of the decoding process. A comparator evaluates the iteration controls η, σ, and β based on the iteration number r, the message length k, and the number of erasures in each data block, ρ. Value of η is 1 in the iteration $r=0$ and is zero otherwise. Values of σ and β are as shown in Fig. 5.1.

There are two other signals for controlling the decoding process of the Decoding cells which are the m-bit discrepancy Δ and 1-bit variable δ. In iterations $r=1,2, \cdots, \rho$, the erasure information α^{j_r} are directed to the Δ output. In the rest of iterations value of Δ is equal to the received value of the discrepancy Δ_0 which is calculated in the Decoding cells. The control variable δ is evaluated as shown in Fig. 5.1. This value is found based on the iteration number r, the number of erasures ρ, the discrepancy Δ_0 and the value of the temporary variable L. The value of L is stored in an m-bit register which is initially set to zero and updated in each iteration based on Eq. (5.2).

5.2.4. Decoding Cell

The structure of the Decoding Cell is based on the operations available outside the dashed box in Fig. 5.1. The detailed structure of the Decoding cell is given in Fig. 5.8. This cell has three sets of m-bit registers for storing values of s_i, λ_i, and b_i after each iteration. The m-bit symbol v_i is the input to the cell and the m-bit symbol e_i is the component of the errata value vector which is calculated at the end of the decoding process.

In each iteration, the ith cell evaluates the partial value of the discrepancy Δ_i and propagates it to the next cell. This partial discrepancy is calculated as,

$$\Delta_i = \Delta_{i+1} + \lambda_i s_i , \tag{5.8}$$

based on the partial discrepancy coming from the previous cell Δ_{i+1}. As shown in Fig. 5.2, the partial discrepancy Δ_n is fixed to zero and this forces the output of the left Decoding cell in Fig. 5.2 to have the value of the discrepancy Δ_0 which is an input to the Decoding-Control unit.

In each iteration the Decoding-Control cell updates the discrepancy Δ_0 and feeds it back to the cells. The Decoding-Control cell also calculates the control variables β, σ, and δ. The 1-bit control variables η, β, σ, and δ together with the m-bit discrepancy Δ are propagated to all the cells which control the operation of the cells in each iteration. Note that values of α^i and α^{-i} are two constants which are applied to the i th cell.

To explain the iterations of the algorithm, assume that the received code word is stored in registers in_i in the I/O cells. Initialization of the algorithm shown in Fig. 5.1 is performed when the iteration counter is 0 which means $\eta=1$. Therefore, v_i is multiplied by α^i and is ready at the input of s_i register. At this step, input of registers λ_i and b_i are set to $\alpha^0=1$. By applying the first clock to three sets of registers the initialization is performed and the first iteration starts. Then, the discrepancy is calculated as explained above and the Decoding-Control cell sends the updated values of the control variables to all the cells. After receiving the controls, each cell calculates values of s_i, λ_i, and b_i. With the application of the next clock, the calculated values are stored in the registers and the second iteration starts.

After applying n clocks, the n th iteration starts and the control variable η becomes 1 again. At the end of this iteration, the errata values are found and sent to the I/O cells to correct the received data block which is available in the dec_i registers of the I/O cells. By applying the next clock, the corrected code word is loaded in registers out_i of the I/O cells and at the same time the registers of the Decoding cells are initialized to decode the next received data block.

5.3. Complexity and Throughput

5.3.1. Complexity

As shown in the previous section, main building blocks of the introduced cellular structure are the Galois field multiplier and m-bit register. To discuss the complexity, let's define the RS-Decoder cell as the collection of one Erasure (Fig. 5.6), one I/O (Fig. 5.3) and one Decoding cell (Fig. 5.8). The Decoder-

Control cell is the combination of the Decoding-Control cell (Fig. 5.7) and the Exponentiation cell (Fig. 5.5). Therefore, the cellular structure consists of $n = 2^m - 1$ RS-Decoder cells and one Decoder-Control cell.

Each RS-Decoder cell consists of five Galois field multipliers and eight m-bit registers. There are also four m-bit exclusive or gates (XOR), four m-bit switches (multiplexer), four m-bit simple switches and two 1-bit switches. Moreover, for fan-out problems we need to pass all the control signals through a delay gate, therefore $2m+4$ extra gates are needed for this purpose. By simple switch we mean that one of the inputs of the switch is constant.

The m-bit registers are very simple and each of them is a D Flip-Flop with one D input, one clock input and one Q output. These registers do not need clear or set controls. The XOR gates, switches and delay gates of an RS-Decoder cell are altogether equivalent to five m-bit registers from the gate count point of view. Therefore, an RS-Decoder cell requires equivalent of thirteen m-bit registers and five m-bit Galois field multipliers. The complexity of the decoder in terms of number of multipliers and number of m-bit registers is given in Table 5.1.

Table 5.1: Complexity of the Cellular Decoder

m	m-bit Register	m-bit Multiplier	Total # of Gates
4	208	80	7500
5	416	160	20500
6	832	320	53000
7	1664	640	132000
8	3328	1280	340500

In Table 5.1, the total number of gates needed for designing the cellular decoder, is also given. For the calculation of the total number of gates, standard basis multiplier and inverter are considered. The reason for this choice is having many multipliers and only one inverter in the design of the decoder, and the multiplier in standard basis has the least complexity compared to other multipliers. For the inverter, a combinational logic circuitry or a table look-up ROM can be used.

5.3.2. Throughput

To discuss the throughput of the introduced decoder, the maximum propagation delay path for each iteration should be found. This path has three following major components:

1- The delay for evaluation of Δ_0 after $\lambda_i s_i$ is ready in all of the Decoding cells. This delay is equivalent to delay of $2^m - 1$ gates.

2- The delay for propagating all the control signals from the Decoding-Control cell to all the Decoding cells. This delay is also equivalent to delay of $2^m - 1$ gates, considering extra gates included in each Decoding cell for fan-out problems.

3- The delay in Decoding-Control cell and the Decoding cells which is equivalent to the delay of three multipliers and 10 gates. Note that we have assumed the multiplier and inversion circuitry in Galois field have the same delay. Considering delay of $m+1$ gates for one multiplier, the total delay for this component is $3m+13$ gates.

Adding up the above three components, the propagation delay for each iteration is equal to the delay of $3m+11+2^{m+1}$ gates. And the maximum bit rate at the input of the decoder is $\frac{m}{(3m+11+2^{m+1})\tau}$ [bits/sec.] where τ [sec.] is the delay of one gate. Considering $2ns$ for delay of one gate the maximum bit rate at the input of the RS decoder is about 38, 25, 19, 12, 7 Mb/s for $m = 4, 5, 6, 7, 8$, respectively. To increase the input bit rate of the decoder some modifications in the structure of the decoder can be made to decrease the delay components of 1, and 2.

To decrease the delay component 1, instead of evaluating the partial values of Δ inside the Decoding cell of Fig. 5.8, we can output the values of $\lambda_i s_i$ from the cells and find Δ_0 by the use of an XOR summation circuitry. This summation circuitry has a delay of only m gates and, hence, the exponential delay of $2^m - 1$ is reduced to a linear delay of m.

The second delay component can also be reduced using the same kind of idea. In this case, instead of passing the control signals from one Decoding cell to another with one gate delay, we introduce a fan-out circuitry. That is, each bit of the control signals is applied to the cells in such a way that one control bit supplies 8 of the Decoding cells. Therefore, the delay of this circuitry is only $m-2$ gate delays.

The total delay for one iteration in this case is equal to delay of $5m+8$ gates and the maximum bit rate at the input of the decoder is $\frac{m}{(5m+8)\tau}$ [bits/sec.], where τ [sec.] is the delay of one gate. In this case, the maximum bit rate of the decoder, given in Table 5.2, is much higher. Note that this increase in the maximum bit rate decreases the number of gates of the decoder but the design of the decoder will be more difficult.

Table 5.2: Maximum Input Bit Rate of the Cellular Decoder

m	Maximum Bit Rate
4	71 Mb/s
5	75 Mb/s
6	78 Mb/s
7	81 Mb/s
8	83 Mb/s

5.3.3. Comparison with Other Structures

In Chapter 4, a versatile RS decoder based on time-domain decoding algorithm was presented. The versatile decoder of Chapter 4 was less modular and slower compared to the cellular structure we have introduced in this chapter. However, the cellular structure uses more chip area for VLSI design. Both these decoders can be used in a specified Galois field 2^m, i.e., for fixed m.

Recently, two universal RS decoders were proposed one based on algebraic decoding algorithm [22] and the other one based on transform decoding algorithm [31]. These decoders are not cellular and their designs are very difficult. The decoding speeds of these decoders are much less than that of the introduced cellular structure.

Three cellular RS decoders two, based on algebraic decoding [24], [59] and the third based on transform decoding [30], have also been introduced. The disadvantage of these structures is that they are not versatile and can only be used for decoding one specific RS code. Moreover, these structures use many different types of cells and this makes their design very difficult compared to that of the introduced cellular structure.

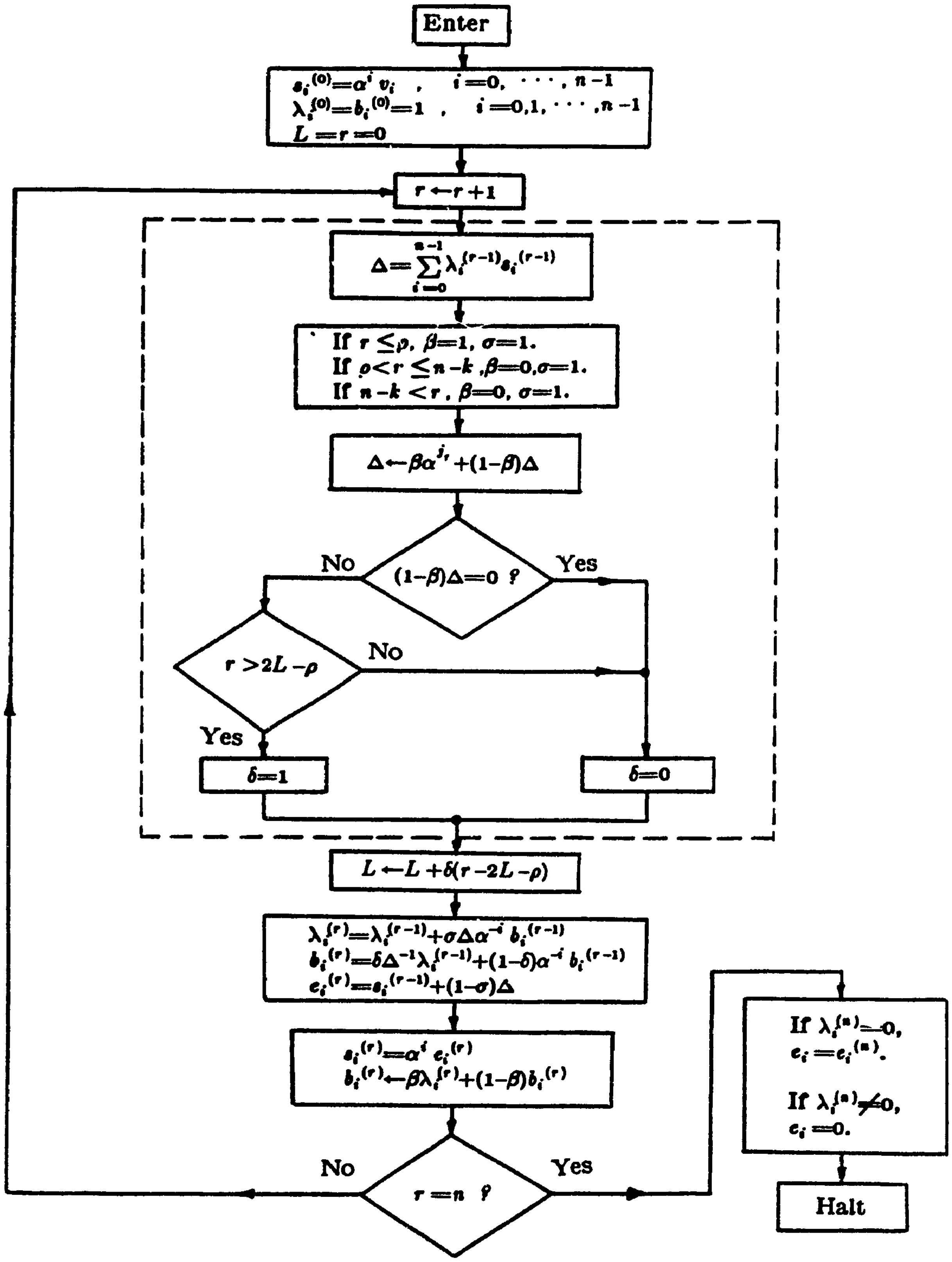

Enter
$s_i^{(0)}=\alpha^i v_i$, $i=0,\cdots,n-1$
$\lambda_i^{(0)}=b_i^{(0)}=1$, $i=0,1,\cdots,n-1$
$L=r=0$
$r \leftarrow r+1$
$\Delta=\sum_{i=0}^{n-1}\lambda_i^{(r-1)}s_i^{(r-1)}$
If $r\le\rho$, $\beta=1$, $\sigma=1$.
If $\rho<r\le n-k$, $\beta=0$, $\sigma=1$.
If $n-k<r$, $\beta=0$, $\sigma=1$.
$\Delta\leftarrow\beta\alpha^{j_r}+(1-\beta)\Delta$
$(1-\beta)\Delta=0$?
No
Yes
$r>2L-\rho$
No
Yes
$\delta=1$
$\delta=0$
$L\leftarrow L+\delta(r-2L-\rho)$
$\lambda_i^{(r)}=\lambda_i^{(r-1)}+\sigma\Delta\alpha^{-i}b_i^{(r-1)}$
$b_i^{(r)}=\delta\Delta^{-1}\lambda_i^{(r-1)}+(1-\delta)\alpha^{-i}b_i^{(r-1)}$
$e_i^{(r)}=s_i^{(r-1)}+(1-\sigma)\Delta$
$s_i^{(r)}=\alpha^i e_i^{(r)}$
$b_i^{(r)}\leftarrow\beta\lambda_i^{(r)}+(1-\beta)b_i^{(r)}$
$r=n$?
No
Yes
If $\lambda_i^{(n)}=0$, $e_i=e_i^{(n)}$.
If $\lambda_i^{(n)}\neq0$, $e_i=0$.
Halt

Figure 5.1: Restructured Time-Domain Decoding Algorithm

Based on Transform Decoder

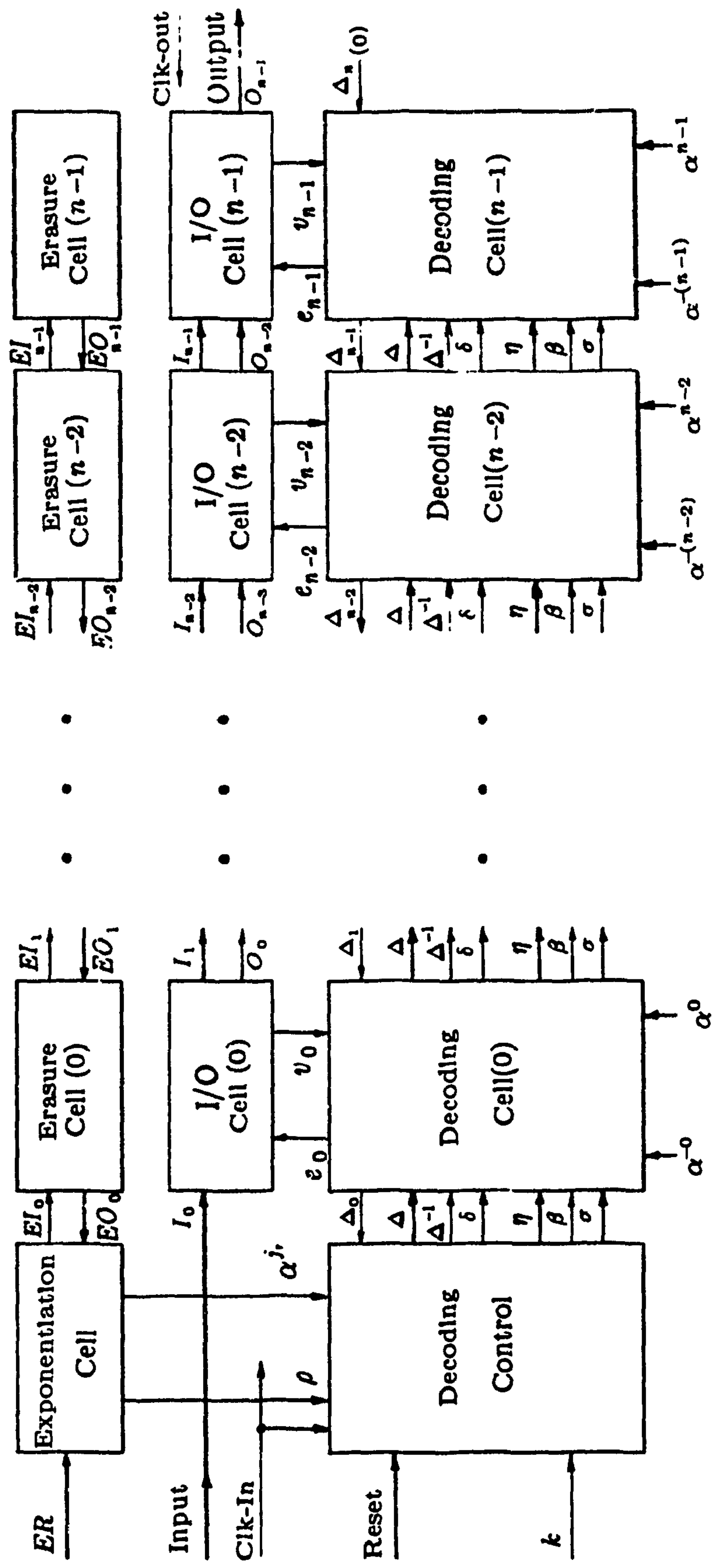


Figure 5.2: Structure of the Cellular RS Decoder

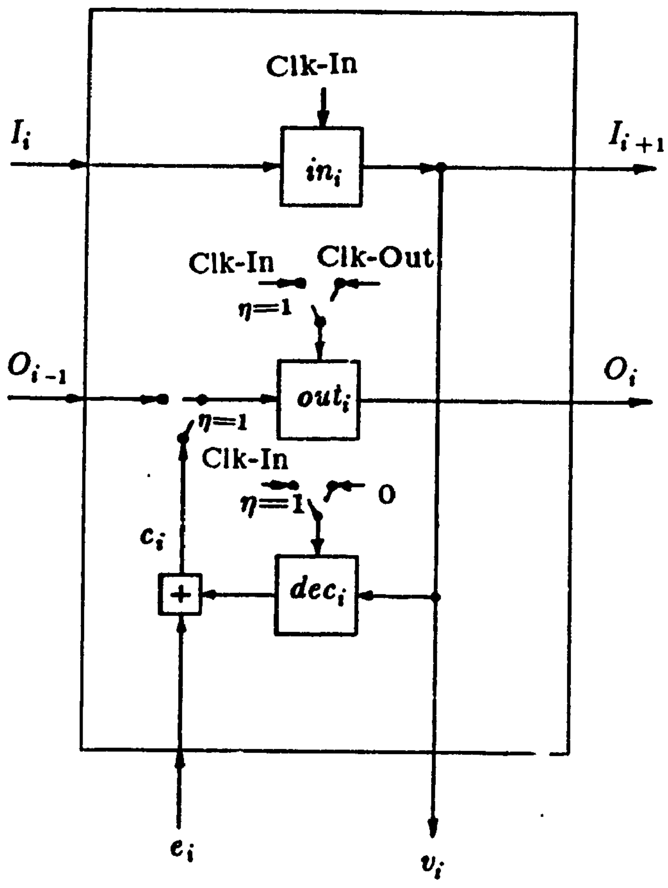


Figure 5.3: Input/Output Cell(i) of the Cellular Decoder

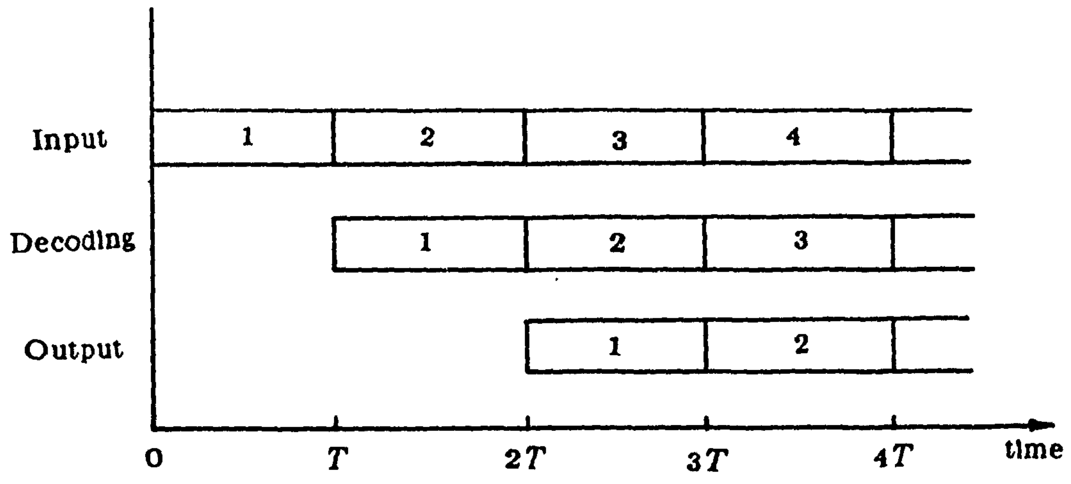


Figure 5.4: Input/Output Cycle Flow of the Cellular Decoder

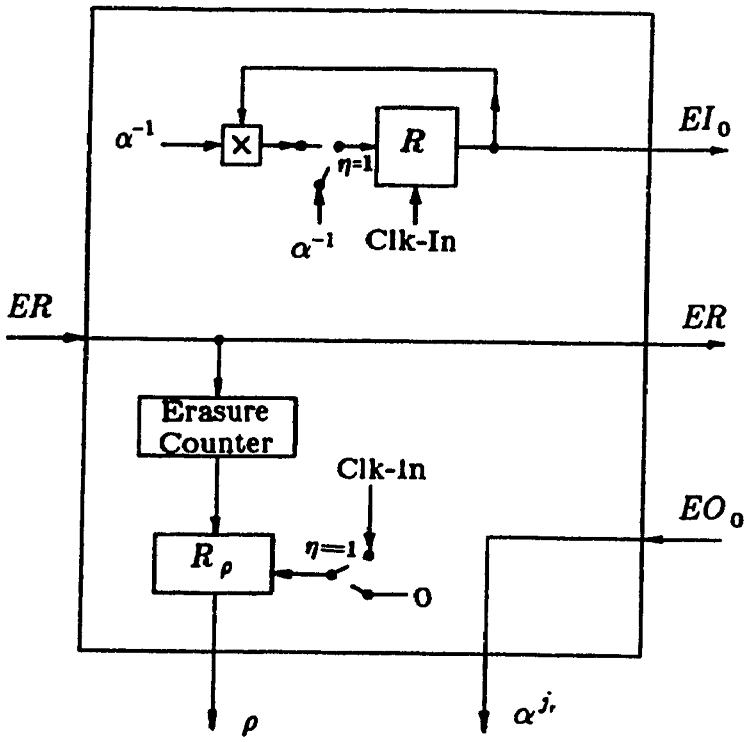


Figure 5.5: Exponentiation Cell of the Cellular Decoder

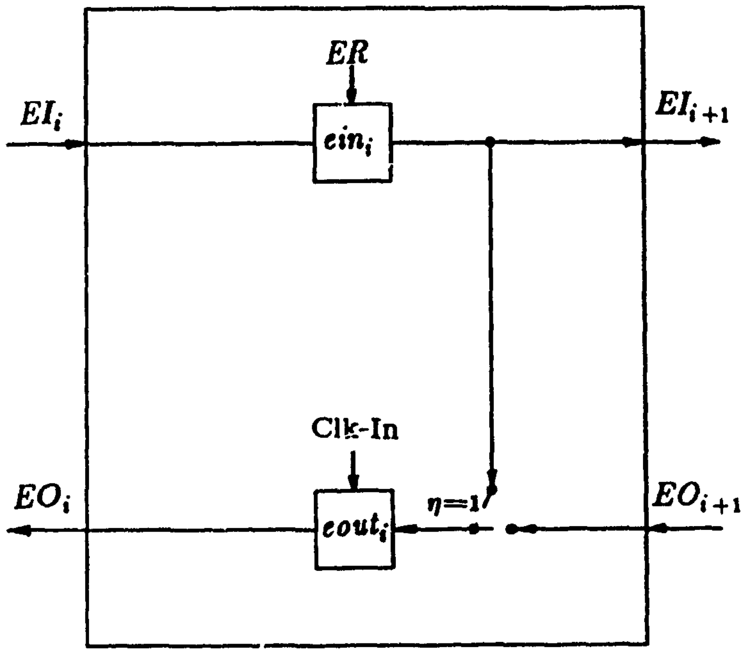


Figure 5.6: Erasure Cell(i) of the Cellular Decoder

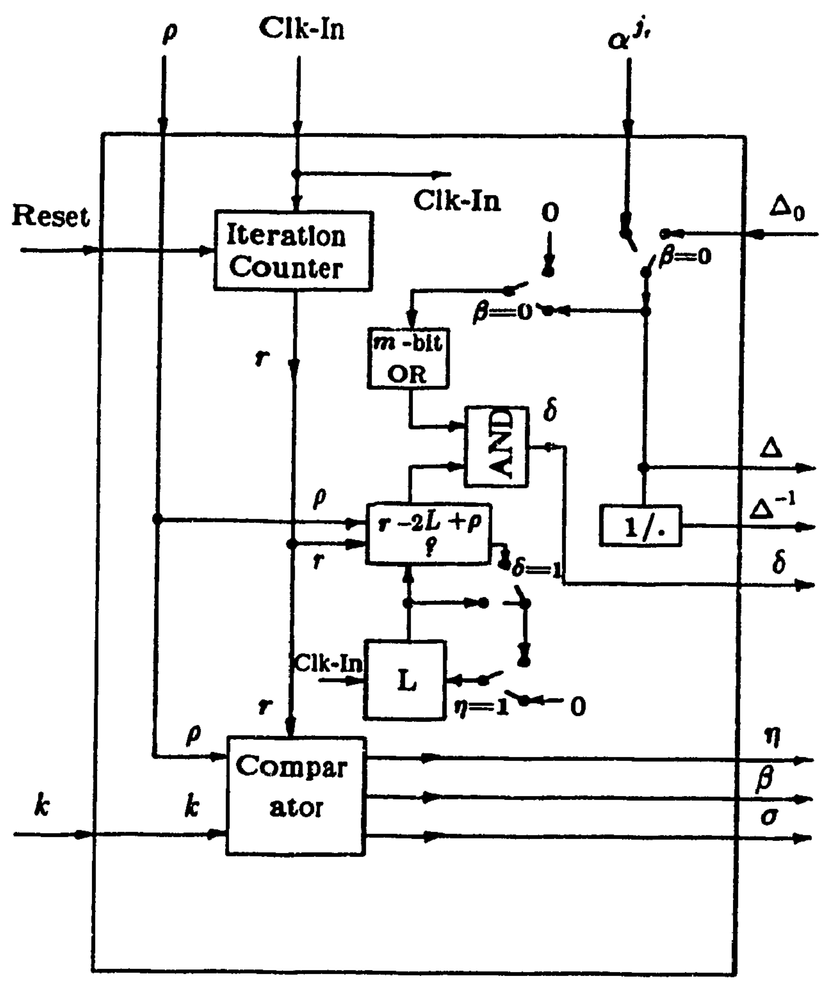


Figure 5.7: Decoding Control Cell of the Cellular Decoder

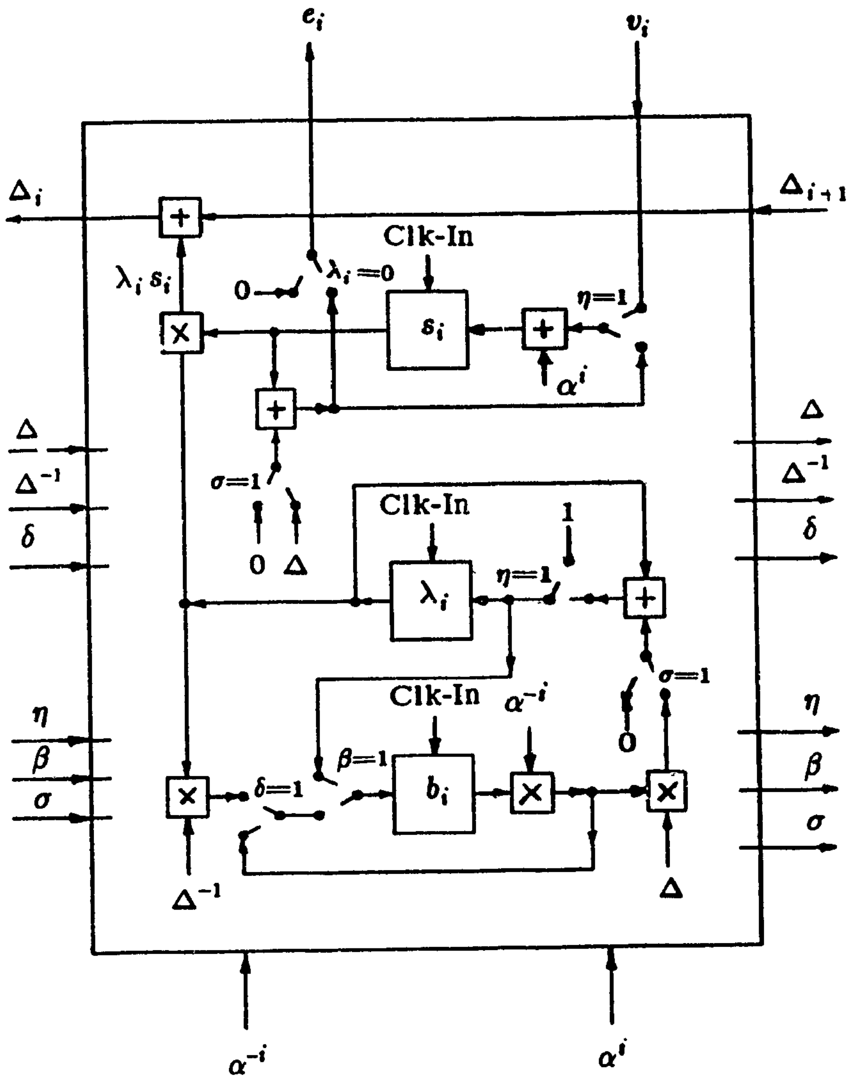


Figure 5.8: Decoding Cell(i) of the Cellular Decoder

CHAPTER SIX

A UNIVERSAL TIME-DOMAIN REED-SOLOMON DECODER

In this chapter, first the time-domain algorithm based on algebraic decoder is modified to reduce one out of two inverters, and then the universal decoder structure is given. This decoder can be programmed to decode any Reed-Solomon code defined in Galois field 2^m, $m=4,5,6,7,8$. The introduced universal decoder can only correct errors. The universal decoder has a simple structure. Complexity and throughput of this structure is discussed.

An RS code with block length n can be generated by $n=2^m-1$ different generator polynomials. To increase the versatility of the introduced universal decoder, a structure is given which is capable of decoding RS codes generated by any generator polynomial [66].

6.1. Decoding Algorithm

In the design of the universal decoder, we have to use the universal multiplier in standard basis, since there is no universal multiplier available in normal basis. Therefore, we should use a combinatorial logic circuitry or a ROM for the inverter. Obviously, using a universal multiplier, increases the propagation delay and the number of gates of the decoder compared to the structures of the previous chapters. So, the decoder of Chapter 4 will be very slow and the structure of Chapter 5 will be very complex for the design of the universal decoder.

To have a low complexity and high speed universal decoder, we propose a structure similar to that of Chapter 4, however, with the time-domain algorithm based on the algebraic decoder [32]. This algorithm is much faster than the algorithm used in Chapters 4, and 5. There is only one problem which is the need for two inverters. One of these inverters can be omitted by modifying the algorithm. The modification is introduced and justified in this section.

The time-domain decoding algorithm based on algebraic decoder for correcting errors has one main step which evaluates the error-locator vector $\underline{\lambda}$, error evaluator vector $\underline{\omega}$ and the vector $\underline{\lambda}'$. This algorithm is explained by the following set of recursive equations [32].

$$\Delta_r = \sum_{i=0}^{n-1} \alpha^{ir} [\lambda_i^{(r-1)} v_i], \tag{6.1}$$

$$L_r = \delta_r (r - L_{r-1}) + (1-\delta_r) L_{r-1}, \tag{6.2}$$

$$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \\ \lambda_i'^{(r)} \\ b_i'^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} & 0 & 0 \\ \Delta_r^{-1}\delta_r & (1-\delta_r)\alpha^{-i} & 0 & 0 \\ 0 & -\Delta_r & 1 & -\Delta_r \alpha^{-i} \\ 0 & (1-\delta_r) & \Delta_r^{-1}\delta_r & (1-\delta_r)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \\ \lambda_i'^{(r-1)} \\ b_i'^{(r-1)} \end{bmatrix}, \tag{6.3}$$

$$\begin{bmatrix} \omega_i^{(r)} \\ a_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} \\ \Delta_r^{-1}\delta_r & (1-\delta_r)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \omega_i^{(r-1)} \\ a_i^{(r-1)} \end{bmatrix}, \tag{6.4}$$

for $i=0, \cdots, n-1$, and $r=1, 2, \cdots, 2t$. The initial conditions are $\lambda_i^{(0)}=b_i^{(0)}=\omega_i^{(0)}=1$ for all i; $\lambda_i'^{(0)}=b_i'^{(0)}=a_i^{(0)}=0$ for all i; $L=0$, and $\delta_r=1$ if both $\Delta_r \neq 0$ and $2L \leq r-1$, and $\delta_r=0$ otherwise. Then, $\lambda_i^{(2t)}=0$ if and only if the ith symbol of the received vector $\underline{v}$ is in error. Also we have $\underline{\lambda}'=\underline{\lambda}'^{(2t)}$, and $\underline{\omega}=\underline{\omega}^{(2t)}$. If $\lambda_i=0$, the error values are evaluated as

$$e_i = -\alpha^i \frac{\omega_i}{\lambda_i'}, \tag{6.5}$$

and $e_i=0$ otherwise.

In this algorithm, one inverter is needed in Eq. (6.5) and another for the evaluation of Δ_r^{-1} in Eqs. (6.3), (6.4). The inverter for Δ_r^{-1} can be omitted and, hence, the structure for the universal decoder will need only one inverter for evaluating e_i in Eq. (6.5).

To omit the inverter of Δ_r^{-1}, we will show that the Berlekamp-Massey algorithm in time domain can be explained by the following recursive equations.

$$\hat{\Delta}_r=\sum_{i=0}^{n-1}\alpha^{ir}\,[\hat{\lambda}_i^{(r-1)}v_i], \tag{6.6}$$

$$\hat{L}_r=\hat{\delta}_r(r-\hat{L}_{r-1})+(1-\hat{\delta}_r)\hat{L}_{r-1}, \tag{6.7}$$

$$\begin{bmatrix}\hat{\lambda}_i^{(r)}\\ \hat{b}_i^{(r)}\\ \hat{\lambda}_i^{'(r)}\\ \hat{b}_i^{'(r)}\end{bmatrix}=\begin{bmatrix}\theta_{r-1} & -\hat{\Delta}_r\alpha^{-i} & 0 & 0\\ \hat{\delta}_r & (1-\hat{\delta}_r)\alpha^{-i} & 0 & 0\\ 0 & -\hat{\Delta}_r & \theta_{r-1} & -\hat{\Delta}_r\alpha^{-i}\\ 0 & (1-\hat{\delta}_r) & \hat{\delta}_r & (1-\hat{\delta}_r)\alpha^{-i}\end{bmatrix}\begin{bmatrix}\hat{\lambda}_i^{(r-1)}\\ \hat{b}_i^{(r-1)}\\ \hat{\lambda}_i^{'(r-1)}\\ \hat{b}_i^{'(r-1)}\end{bmatrix}, \tag{6.8}$$

$$\begin{bmatrix}\hat{\omega}_i^{(r)}\\ \hat{a}_i^{(r)}\end{bmatrix}=\begin{bmatrix}\theta_{r-1} & -\hat{\Delta}_r\alpha^{-i}\\ \hat{\delta}_r & (1-\hat{\delta}_r)\alpha^{-i}\end{bmatrix}\begin{bmatrix}\hat{\omega}_i^{(r-1)}\\ \hat{a}_i^{(r-1)}\end{bmatrix}, \tag{6.9}$$

$$\theta_r=\hat{\delta}_r\hat{\Delta}_r+(1-\hat{\delta}_r)\theta_{r-1} \tag{6.10}$$

for $i=0,\cdots,n-1$, and for $r=1,2,\cdots,2t$. The initial conditions are $\hat{\lambda}_i^{(0)}=\hat{b}_i^{(0)}=\hat{\omega}_i^{(0)}=1$ for all i; $\hat{\lambda}_i^{'(0)}=\hat{b}_i^{'(0)}=\hat{a}_i^{(0)}=0$ for all i; $\hat{L}_0=0$, $\theta_0=1$ and $\hat{\delta}_r=1$ if both $\hat{\Delta}_r\neq 0$ and $2\hat{L}_r\leq r-1$, and $\hat{\delta}_r=0$, otherwise. Then, $\hat{\lambda}_i^{(2t)}=0$ if and only if the ith symbol of the received vector $\underline{v}$ is in error. We have $\hat{\lambda}_i'=\hat{\lambda}_i^{'(2t)}$, and $\hat{\omega}_i=\hat{\omega}_i^{(2t)}$, for all i. If $\hat{\lambda}_i=0$, the error values are evaluated as

$$e_i=-\alpha^i\frac{\hat{\omega}_i}{\hat{\lambda}_i'}, \tag{6.11}$$

and $e_i = 0$, otherwise.

The new algorithm is shown in Fig. 6.1. In this algorithm, a new parameter θ_r is added compared to the original algorithm. This parameter is initialized in the beginning of the algorithm and updated in each iteration.

To prove that this algorithm is equivalent to the original one, we will prove the following:

1) $\hat{\lambda}_i = K\lambda_i$, for all i where K is a non-zero element of GF(2^m).

2) $e_i = \alpha^i \dfrac{\hat{\lambda}_i'}{\hat{\omega}_i} = \alpha^i \dfrac{\lambda_i'}{\omega_i}$ for all i.

To prove above two conditions we present the following theorem.

Theorem 6.1: The variables of the original algorithm and the modified one are related by,

$$\hat{L}_r = L_r \tag{6.12}$$

$$\hat{b}_i^{(r)} = \theta_r\, b_i^{(r)}, \tag{6.13}$$

$$\hat{\lambda}_i^{(r)} = \lambda_i^{(r)} \prod_{l=0}^{r-1} \theta_l, \tag{6.14}$$

$$\hat{b}_i'^{\,(r)} = \theta_r\, b_i'^{\,(r)}, \tag{6.15}$$

$$\hat{\lambda}_i'^{\,(r)} = \lambda_i'^{\,(r)} \prod_{l=0}^{r-1} \theta_l, \tag{6.16}$$

$$\hat{a}_i^{(r)} = \theta_r\, a_i^{(r)}, \tag{6.17}$$

$$\hat{\omega}_i^{(r)} = \omega_i^{(r)} \prod_{l=0}^{r-1} \theta_l, \tag{6.18}$$

where $r = 2t$. Note that after proving Eqs. (6.12), (6.13), (6.15), (6.17), they are used in the proof of the other three equations.

Proof: To prove this theorem we use induction [47]. The basic step of the induction is evident from the initialization of the vectors and $\prod_{l=0}^{0} \theta_l = \theta_0 = 1$. The assumptions of the induction are Eqs. (6.12)-(6.18) when $r = r'$. Therefore, assuming Eqs. (6.12)-(2.18) are true for $r = r'$, we should prove the same equations for $r = r' + 1$.

Considering Eqs. (6.1), (6.6), and Eq. (6.14) for $r = r'$, we have

$$\hat{\Delta}_{r'+1} = \Delta_{r'+1} \prod_{l=0}^{r'-1} \theta_l . \tag{6.19}$$

According to Eq.(6.10) and the definition of $\hat{\delta}_r$, it is obvious that θ_l is always nonzero and so is $\prod_{l=0}^{2t-1} \theta_l$. Hence, from Eq. (6.19), $\hat{\Delta}_{r'+1}$ and $\Delta_{r'+1}$ are both zero or both nonzero at iteration $r'+1$. This results in

$$\hat{\delta}_{r'+1} = \delta_{r'+1}, \tag{6.20}$$

and $\hat{L}_{r'+1} = L_{r'+1}$ and the proof of Eq. (6.12) is complete.

Now the proof of Eqs. (6.13)-(6.18) can be given by expanding the left-hand sides of these equations and showing that they are the same as the right-hand side. This can be done by using Eqs. (6.10), (6.19), (6.20), (6.3), (6.4), (6.8), (6.9), and assumptions of the induction.

Proof of Eq. (6.13):

$$\begin{aligned}
\hat{b}_i^{(r'+1)} &= \hat{\delta}_{r'+1} \hat{\lambda}_i^{(r')} + (1 - \hat{\delta}_{r'+1}) \alpha^{-i} \hat{b}_i^{(r')} \\
&= \hat{\delta}_{r'+1} \prod_{l=0}^{r'-1} \theta_l \lambda_i^{(r')} + (1 - \hat{\delta}_{r'+1}) \alpha^{-i} \theta_{r'} b_i^{(r')} \\
&= \hat{\delta}_{r'+1} \Delta_{r'+1}^{-1} \hat{\Delta}_{r'+1} \lambda_i^{(r')} + (1 - \hat{\delta}_{r'+1}) \alpha^{-i} \theta_{r'} b_i^{(r')} \\
&= \hat{\delta}_{r'+1} \Delta_{r'+1}^{-1} \theta_{r'+1} \lambda_i^{(r')} + (1 - \hat{\delta}_{r'+1}) \alpha^{-i} \theta_{r'+1} b_i^{(r')} \\
&= \theta_{r'+1} b_i^{(r'+1)}.
\end{aligned}$$

Proof of Eq. (6.14):

$$\begin{aligned}
\hat{\lambda}_i^{(r'+1)} &= \theta_{r'} \hat{\lambda}_i^{(r')} + \hat{\Delta}_{r'+1} \alpha^{-i} \hat{b}_i^{(r')} \\
&= \theta_{r'} \prod_{l=0}^{r'-1} \theta_l \lambda_i^{(r')} + \hat{\Delta}_{r'+1} \alpha^{-i} \theta_{r'} b_i^{(r')} \\
&= \prod_{l=0}^{r'} \theta_l \lambda_i^{(r')} + \Delta_{r'+1} \prod_{l=0}^{r'-1} \theta_l \ \alpha^{-i} \theta_{r'} b_i^{(r')} \\
&= \prod_{l=0}^{r'} \theta_l \lambda_i^{(r'+1)}.
\end{aligned}$$

Proof of Eq. (6.15):

$$
\begin{aligned}
\hat{b}_i'^{(r'+1)} &= (1-\hat{\delta}_{r'+1})\hat{b}_i^{(r')} + \hat{\delta}_{r'+1}\hat{\lambda}_i'^{(r')} + (1-\hat{\delta}_{r'+1})\alpha^{-i}\hat{b}_i'^{(r')} \\
&= (1-\hat{\delta}_{r'+1})\theta_{r'} b_i^{(r')} + \hat{\delta}_{r'+1}\prod_{l=0}^{r'-1}\theta_l \lambda_i'^{(r')} + (1-\hat{\delta}_{r'+1})\alpha^{-i}\theta_{r'} b_i'^{(r')} \\
&= (1-\hat{\delta}_{r'+1})\theta_{r'} b_i^{(r')} + \hat{\delta}_{r'+1}\Delta_{r'+1}^{-1}\hat{\Delta}_{r'+1}\lambda_i'^{(r')} + (1-\hat{\delta}_{r'+1})\alpha^{-i}\theta_{r'} b_i'^{(r')} \\
&= (1-\hat{\delta}_{r'+1})\theta_{r'+1} b_i^{(r')} + \hat{\delta}_{r'+1}\Delta_{r'+1}^{-1}\theta_{r'+1}\lambda_i'^{(r')} + (1-\hat{\delta}_{r'+1})\alpha^{-i}\theta_{r'+1} b_i'^{(r')} \\
&= \theta_{r'+1} b_i'^{(r'+1)}.
\end{aligned}
$$

Proof of Eq. (6.16):

$$
\begin{aligned}
\hat{\lambda}_i'^{(r'+1)} &= \hat{\Delta}_{r'+1}\hat{b}_i^{(r')} + \theta_{r'}\hat{\lambda}_i'^{(r')} + \hat{\Delta}_{r'+1}\alpha^{-i}\hat{b}_i'^{(r')} \\
&= \Delta_{r'+1}\prod_{l=0}^{r'-1}\theta_l\,\theta_{r'} b_i^{(r')} + \theta_{r'}\prod_{l=0}^{r'-1}\theta_l \lambda_i'^{(r')} + \Delta_{r'+1}\prod_{l=0}^{r'-1}\theta_l\,\alpha^{-i}\theta_{r'} b_i'^{(r')} \\
&= \prod_{l=0}^{r'}\theta_l\,[\Delta_{r'+1} b_i^{(r')} + \lambda_i'^{(r')} + \Delta_{r'+1}\alpha^{-i} b_i'^{(r')}] \\
&= \prod_{l=0}^{r'}\theta_l\,\lambda_i'^{(r'+1)}.
\end{aligned}
$$

The proofs of Eqs. (6.17), (6.18) are the same as the proof of Eqs. (6.13), (6.14), respectively. Note that according to Eq. (6.10), $\theta_{r'+1}=\theta_{r'}$ for $\hat{\delta}_{r'+1}=0$ and $\theta_{r'+1}=\hat{\Delta}_{r'}$, otherwise. This fact is used in expanding above equations. Q.E.D.

6.2. Decoder Architecture

The structure of the universal decoder is shown in Fig. 6.2 [32]. Figure 6.2 shows that we need six sets of shift registers to store values of the vectors $\hat{b}$, $\hat{\lambda}$, $\hat{b}'$, $\hat{\lambda}'$, $\hat{\omega}$, and $\hat{a}$. This decoder works the same way as the versatile decoder of Chapter 4. In iteration $r=0$, all of the vectors are initialized to their initial values. In each iteration the received vector $\underline{y}$ is entered to the arithmetic and control unit for calculating $\hat{\Delta}_r$ according to Eq. (6.6). All the shift registers are shifted right by the system clock and the components of the vectors are updated in the arithmetic and control unit based on Eqs. (6.8), (6.9) and are stored again in the shift registers. At the end of iteration $2t$, the received vector is corrected at components where $\hat{\lambda}_i = 0$ using Eq. (6.11), and $\underline{c} = \underline{v} - \underline{e}$.

The length of the shift register buffers, should be equal to 51,31,63,127,255 for GF(2^m), m =4,5,6,7,8, respectively. To have a variable length buffer, we use 255-stage buffer and a multiplexer to direct the output of the right register to the arithmetic and control unit. The structure of the buffer is shown in Fig. 6.3, where gf_m is a variable indicating the GF(2^m), for m =4,5,6,7,8. The variable gf_m is high only for one of the values of m =4,5,6,7,8 at a time, which chooses the right Galois field. For this design, 5 extra pass gates are needed.

To design the arithmetic unit of the universal decoder, 14 multipliers, 3 α-multipliers, and one inverter are needed. As shown in Chapter 3, universal multiplier and universal α-multiplier are available in standard basis. The universal multiplier needs 183 gates and has a delay equivalent to 17 gates. For the inverter, table look-up using a ROM can be used. Considering the complexity of the multiplier and the number of multipliers used, we need about 5000 gates to build the arithmetic and control unit. The buffers require about 75000 gates. So, the complexity of the decoder is about 80000 gates.

The decoding time of the decoder is determined by the longest delay path. This path has a delay of τ, equivalent to delay of about 50 gates. The decoding algorithm needs $2t$ iterations and one extra iteration for initialization. In each iteration the shift registers are shifted to the right $n = 2^m - 1$ times. So, one iteration period is $T = n\tau$ and the decoding time is $n(2t+1)\tau$. The decoding time can be used to evaluate the maximum bit rate of the code, which is $\frac{m}{(2t+1)\tau}$. Considering a clock of 10 MHz, the maximum bit rates at the input of the decoder for GF(2^m) are given in Table 6.1.

Table 6.1: Maximum Bit Rate of the Universal

Reed-Solomon Decoder [Mb/s]

m	n	Rate=1/2	Rate=3/4	
4	15	4.4	8.0	
5	31	3.0	5.5	
6	63	1.8	3.3	
7	127	1.1	2.1	
8	255	0.6	1.2	

6.3. Decoding with any Generator Polynomial

A Reed-Solomon (RS) code with block length n and number of information symbols k, can be generated by $n=2^m-1$ different generator polynomials,

$$\hat{g}(x)=\prod_{i=0}^{n-k-1}(x+\alpha^{h+i}), \tag{6.21}$$

where the constant h can be chosen to have values of 0, 1, 2, $\cdots$, $n-1$ and m is the element size of the field.

Berlekamp has shown [14] that for any RS code there is a specified constant h in Eq. (6.21) such that the generator polynomial becomes reciprocal. Using this property, he has given the structure of a systematic encoder which has a very low complexity. This is due to the fact that in a reciprocal generator polynomial only half of the coefficients need to be stored. On the other hand the decoder structures are simpler when the constant h in Eq. (6.21) is equal to one, i. e.,

$$g(x)=\prod_{i=0}^{n-k-1}(x+\alpha^{1+i}). \tag{6.22}$$

The relation between $\hat{g}(x)$ and $g(x)$ can be written as,

$$\hat{g}(x)=\alpha^{(n-k)(h-1)}g(\alpha^{-(h-1)}x). \tag{6.23}$$

In this section, generation of RS codes using different generator polynomials are compared, and a method is introduced to use a decoder which is based on $g(x)$, for decoding a code generated using $\hat{g}(x)$. The design of the extra hardware needed for this structure is also given [66].

6.3.1. RS Code Generation with Different Polynomials

In this section, generation of RS codes using two generator polynomials represented by Eqs. (6.21), (6.22) are compared. It is shown that generation of the code polynomial $c(x)$ by the generator polynomial $g(x)$ from the message polynomial $d(x)$, is equivalent to generation of $\hat{c}(x)$ by the generator polynomial $\hat{g}(x)$ from $\hat{d}(x)$ and then evaluating $c(x)$ from $\hat{c}(x)$, where

$$\hat{d}(x)=d(\alpha^{(h-1)}x) \quad or \quad d(x)=\hat{d}(\alpha^{-(h-1)}x), \tag{6.24}$$
$$c(x)=\hat{c}(\alpha^{-(h-1)}x) \quad or \quad \hat{c}(x)=c(\alpha^{(h-1)}x). \tag{6.25}$$

The equations in Eq. (6.24) are alternate representations of each other, this also applies to Eq. (6.25).

Considering Eq. (6.24) is true, we will prove that Eq. (6.25) is also true. Note that it is also possible to prove the inverse which means to prove Eq. (6.24) from Eq. (6.25). Let $d(x)$ be the message polynomial and $p(x)$ be its parity polynomial which is the remainder from dividing $x^{n-k}d(x)$ by the generator polynomial $g(x)$,

$$x^{n-k}d(x)=g(x)q(x)+p(x) \tag{6.26}$$

where $deg[p(x)]<deg[q(x)]$†. The code word polynomial for $d(x)$ can be formed as,

$$c(x)=d(x)+x^k p(x). \tag{6.27}$$

† deg [ ] denotes the degree of the polynomial.

The parity polynomial of $\hat{d}(x)$ which is $\hat{p}(x)$ is formed from dividing $x^{n-k}\,\hat{d}(x)$ by the generator polynomial $\hat{g}(x)$. The remainder of this division is $\hat{p}(x)$,

$$x^{n-k}\,\hat{d}(x)=\hat{g}(x)\hat{q}(x)+\hat{p}(x) \tag{6.28}$$

where $deg\,[\hat{p}(x)]<deg\,[\hat{q}(x)]$. The code word polynomial for $\hat{d}(x)$ can be formed as,

$$\hat{c}(x)=\hat{d}(x)+x^{k}\,\hat{p}(x). \tag{6.29}$$

To find the relation between $p(x)$ and $\hat{p}(x)$ and prove Eq. (6.25), the parity polynomial $\hat{p}(x)$ is evaluated by substituting Eqs. (6.23), (6.24) into Eq. (6.28)

$$x^{n-k}\,d(\alpha^{-(h-1)}x)=\alpha^{(n-k)(h-1)}g(\alpha^{-(h-1)}x)\hat{q}(x)+\hat{p}(x). \tag{6.30}$$

Changing the variable $x\rightarrow\alpha^{(h-1)}x$ and dividing both sides of Eq. (6.30) by $\alpha^{(n-k)(h-1)}$ yields,

$$x^{(n-k)}d(x)=g(x)\hat{q}(\alpha^{(h-1)}x)+\alpha^{-(n-k)(h-1)}\hat{p}(\alpha^{(h-1)}x). \tag{6.31}$$

Comparing Eq. (6.26) and Eq. (6.31) and noting that degrees of $p(x)$ and $\hat{p}(x)$ are less than that of $q(x)$ and $\hat{q}(x)$, respectively, we can conclude that,

$$p(x)=\alpha^{-(n-k)(h-1)}\hat{p}(\alpha^{h-1}x). \tag{6.32}$$

Now, $c(x)$ can be evaluated using Eqs. (6.24), (6.27), (6.32)

$$c(x)=\hat{d}(\alpha^{-(h-1)}x)+\alpha^{-(n-k)(h-1)}x^{k}\,\hat{p}(\alpha^{-(h-1)}x). \tag{6.33}$$

Change of variable, $x\rightarrow\alpha^{(h-1)}x$ in Eq. (6.29) yields,

$$\hat{c}(\alpha^{-(h-1)}x)=\hat{d}(\alpha^{-(h-1)}x)+\alpha^{k(h-1)}x^{k}\,\hat{p}(\alpha^{-(h-1)}x). \tag{6.34}$$

Since in a field of order n, the nth power of any element of the field is 1, so $\alpha^{-n(h-1)}=1$ and right hand sides of Eqs. (6.33), (6.34) are similar. Therefore, the relation between $c(x)$ and $\hat{c}(x)$, shown in Eq. (6.25), is proved.

The discussion in this section can be used to introduce a decoder structure, which can decode an RS code generated by $\hat{g}(x)$, using a decoder based on $g(x)$.

6.3.2. Decoder Structure

To introduce the decoder structure, Eqs. (6.24), (6.25) are written in the form as shown in Eqs. (6.35), (6.36), respectively.

$$\hat{d}_j = d_j \alpha^{j(h-1)} \qquad j=0,1,\cdots,k-1 \tag{6.35}$$
$$c_i = \hat{c}_i \alpha^{-i(h-1)} \qquad i=0,1,\cdots,n-1 \tag{6.36}$$

In these equations d_j, $\hat{d}_j$, c_i and $\hat{c}_i$ are coefficients of the corresponding polynomials in Eqs. (6.24), (6.25).

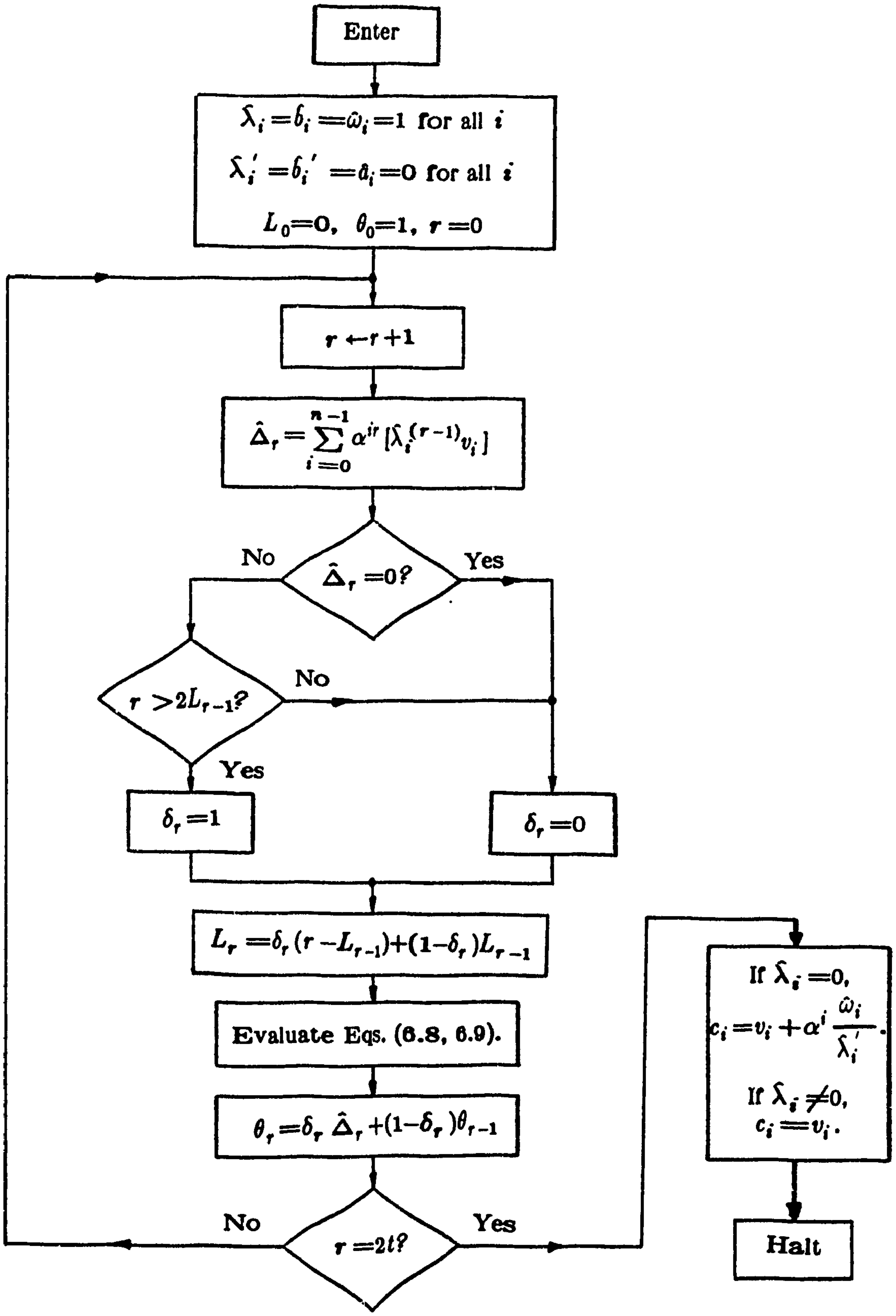
Now, let's consider an RS code word, $\underline{\hat{c}}$, generated by the generator polynomial $\hat{g}(x)$. In the channel the error vector $\underline{\hat{e}}$ is added to $\underline{\hat{c}}$ to form the received vector $\underline{\hat{v}}$ at the input of the decoder. To decode this code, the structure shown in Fig. 6.4 can be used. This structure is based on the discussion presented in previous subsection. In Fig. 6.4, each symbol of the received vector, $\hat{v}_i$, is multiplied by $\alpha^{-i(h-1)}$ to form v_i. The vector $\underline{v}$ has two components, the code word vector $\underline{c}$ and a new error vector $\underline{e}$. The i th symbol of these vectors are as shown in Eqs. (6.36), (6.37).

$$e_i = \hat{e}_i \alpha^{-i(h-1)} \qquad i=0,1,\cdots,n-1. \tag{6.37}$$

The new error vector $\underline{e}$ has the same zero components as the vector $\underline{\hat{e}}$, since the coefficient $\alpha^{-i(h-1)}$ in Eq. (6.37) is nonzero for all values of i and h. Therefore, the new vector $\underline{v}$ has the same error locations as $\underline{\hat{v}}$, and a decoder based on $g(x)$ can be used to correct the errors introduced in the channel. However after the error correction, the message vector $\underline{d}$, is based on generator polynomial $g(x)$ and should be transferred to $\underline{\hat{d}}$ as given in Eq. (6.35). This transfer of the message vector, completes the decoding.

In Fig. 6.4, the generation of the constants needed for multiplications is also shown. In this design, we have assumed that the decoder receives the symbol $\hat{v}_0$ first and the symbol $\hat{v}_{n-1}$ last. The output of the decoder is also in the same order which means that $\hat{d}_0$ is transmitted first and $\hat{d}_{k-1}$ last. To generate $\alpha^{-i(h-1)}$, the m-bit register R_{in} is initialized to $\alpha^0=1$ when the first symbol $\hat{v}_0$ is received. By receiving the next symbol $\hat{v}_1$, the register R_{in} is clocked and the new value in this register $\alpha^{-(h-1)}$ is multiplied by $\hat{v}_1$ to form v_1. This procedure carries on and by receiving each input symbol, the register R_{in} is clocked and proper v_i is generated. The constant $\alpha^{-(h-1)}$ which is an element of Galois field, is fixed based on the constant h of the generator polynomial $\hat{g}(x)$. The circuitry for generating the output of the decoder is similar to the input circuitry. In this circuitry, the register R_{out} is clocked when outputting a symbol from the decoder.

As shown in this section some extra hardware is used to change the structure of the $g(x)$ decoder to the $\hat{g}(x)$ decoder. Four parallel type Galois field multipliers, and two registers of size m are needed for this design. Note that for designing the $g(x)$ decoder there is no limitation on the choice of the decoding algorithm, and any algorithm such as algebraic, transform and time-domain decoding algorithms can be used.



Enter
$\hat{\lambda}_i = \hat{\delta}_i = \hat{\omega}_i = 1$ for all i
$\hat{\lambda}_i' = \hat{\delta}_i' = \hat{a}_i = 0$ for all i
$L_0 = 0, \ \theta_0 = 1, \ r = 0$
$r \leftarrow r+1$
$\hat{\Delta}_r = \sum_{i=0}^{n-1} \alpha^{ir} [\hat{\lambda}_i^{(r-1)} v_i]$
$\hat{\Delta}_r = 0?$
No
Yes
$r > 2L_{r-1}?$
No
Yes
$\delta_r = 1$
$\delta_r = 0$
$L_r = \delta_r (r - L_{r-1}) + (1-\delta_r) L_{r-1}$
Evaluate Eqs. (6.8, 6.9).
$\theta_r = \delta_r \hat{\Delta}_r + (1-\delta_r) \theta_{r-1}$
$r = 2t?$
No
Yes
If $\hat{\lambda}_i = 0$, $c_i = v_i + \alpha^i \frac{\hat{\omega}_i}{\hat{\lambda}_i'}$.
If $\hat{\lambda}_i \neq 0$, $c_i = v_i$.
Halt

Figure 6.1: Modified Time-Domain Decoding Algorithm

Based on Algebraic Decoder

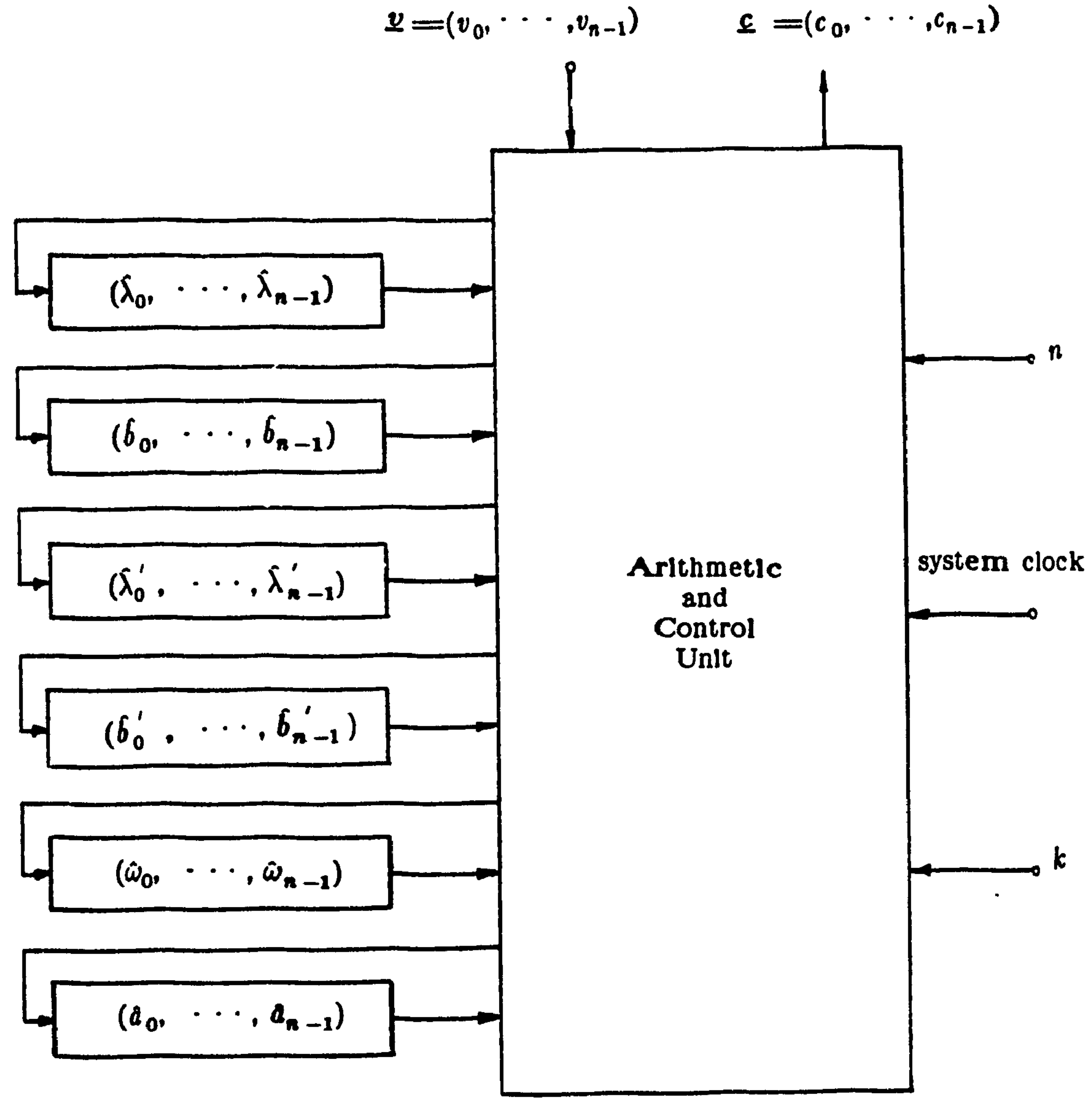

$\underline{v} = (v_0, \cdots, v_{n-1})$
$\underline{c} = (c_0, \cdots, c_{n-1})$
$(\hat{\lambda}_0, \cdots, \hat{\lambda}_{n-1})$
$(\hat{b}_0, \cdots, \hat{b}_{n-1})$
$(\hat{\lambda}'_0, \cdots, \hat{\lambda}'_{n-1})$
$(\hat{b}'_0, \cdots, \hat{b}'_{n-1})$
$(\hat{\omega}_0, \cdots, \hat{\omega}_{n-1})$
$(\hat{a}_0, \cdots, \hat{a}_{n-1})$
Arithmetic and Control Unit
n
system clock
k

Figure 6.2: Universal Decoder Structure [32]

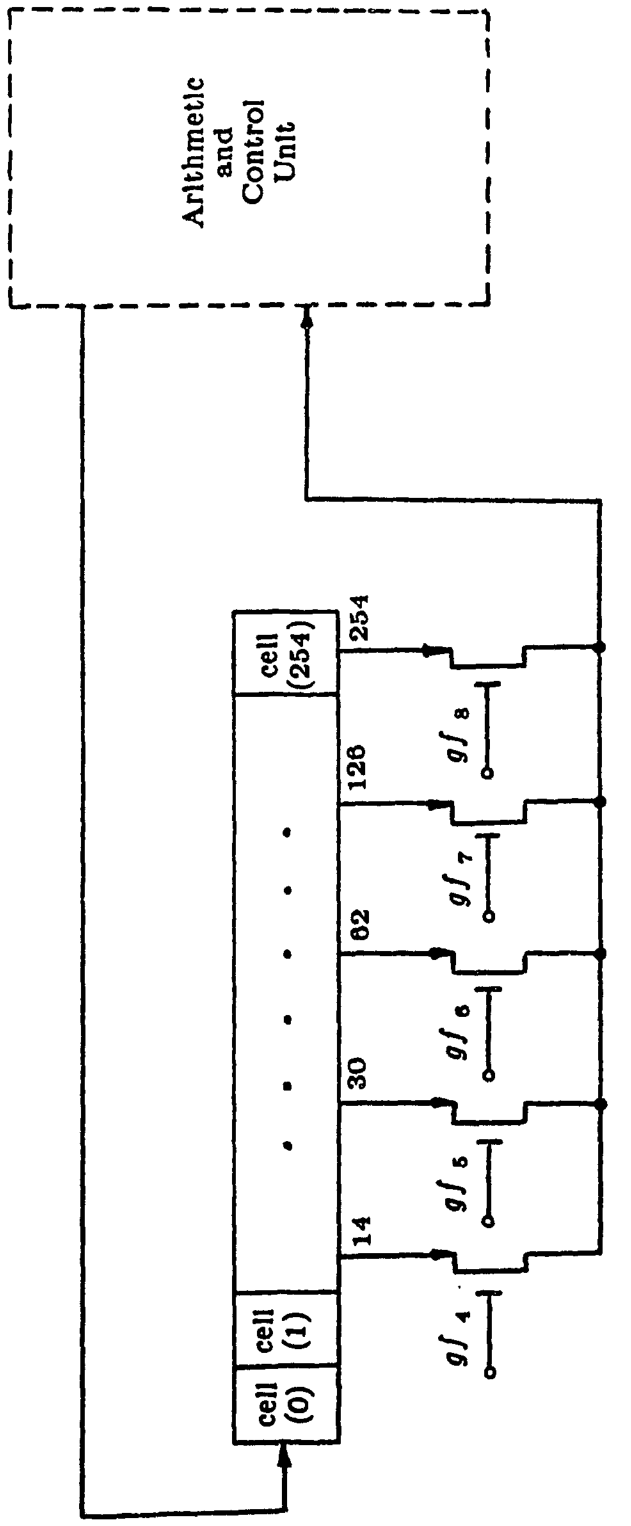


Figure 6.3: Buffer Structure of the Universal Decoder

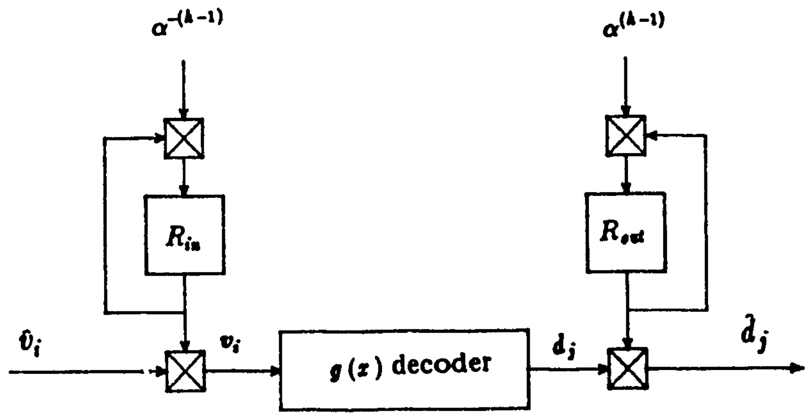


Figure 6.4: Structure of $\hat{g}(x)$ decoder

CHAPTER SEVEN

SOFTWARE REED-SOLOMON DECODERS

In this chapter, microprocessor based RS decoders are discussed. First the look-up table method is explained in detail, since in software decoders based on general purpose microprocessors, this is the best method for arithmetic in Galois fields [22]. Then, the syndrome-based algorithms and time-domain algorithms are compared for microprocessor-based decoders. It is shown that syndrome-based algorithms are best suited for software decoders.

In the last section, a method is introduced to speed up the Chien search step of the algebraic decoding algorithm for high rate codes [23]. To introduce this method software evaluation of the roots of polynomials defined in Galois fields are considered. It is shown that second and third degree polynomials can be solved using a small look-up table by applying a linear translation to the polynomials [67]. For finding the roots of higher degree polynomials, a new and fast algorithm is presented based on the translated polynomials.

7.1. Arithmetic in Galois Fields Using Look-Up Table

The look-up table method for Galois field arithmetic is used in software decoders based on general purpose processors. In look-up table method it is possible to increase the speed of the multiplication in Galois fields by increasing the memory space [22]. This approach is explained in this section.

Each element of Galois field (2^m) is represented by m binary digits (bits). Converting binary representation to decimal, the field elements have values of

0, 1, $\cdots$, n where $n=2^m-1$.

The addition or subtraction of two elements $\beta=(b_0, b_1, \cdots, b_{m-1})$ and $\gamma=(c_0, c_1, \cdots, c_{m-1})$ are performed as,

$$\beta+\gamma=\beta-\gamma=(b_0+c_0, \cdots, b_{m-1}+c_{m-1}), \tag{7.1}$$

where the component by component addition is performed in modulo-2. This operation can be performed easily by microprocessors.

Multiplication of two elements β and γ can be performed by look-up table method. This method can be explained using the following formula [22]:

$$\beta.\gamma=alog\ [\log\ [\ \beta\ ]+\log\ [\ \gamma\ ]\ ]. \tag{7.2}$$

The log [ ] table is set up so that

$$\begin{aligned}\log\ [\ i\ ]&=\log_\alpha i, \quad \text{for} \quad 1\leq i\leq 2^m-1\\ \log\ [\ 0\ ]&=2n-1,\end{aligned} \tag{7.3}$$

where α is a root of the generator polynomial of the field. The largest value in the log [ ] table (other than the special value log [0]) is $n-1$. Thus, the sum of two values from the log [ ] table cannot exceed $2(n-1)=2n-2$, unless one of the two addends is log [0].

The *alog* [ ] table is set up so that

$$\begin{aligned}&alog\ [\ i\ ]=\alpha^i, \quad \text{for} \quad 0\leq i\leq n-1,\\ &alog\ [\ i\ ]=\alpha^{i-n}, \quad \text{for} \quad n\leq i\leq 2(n-1),\\ &alog\ [\ i\ ]=0, \quad \text{for} \quad 2n-1\leq i\leq 3n-2, \quad \text{and} \quad i=4n-2.\end{aligned} \tag{7.4}$$

In this way, table space is traded off against running time and code space, i.e., the log [ ] table is made $m+1$ bits wide (instead of m), and the *alog* [ ] table is made $4n-1$ characters long (instead of $n+1$). As a result, no testing is necessary for doing Galois field multiplication by summing the logarithms of the multiplicands and taking the antilog of the sum. If one of the multiplicands is zero, the large value $(2n-1)$ stored at log [0] guarantees that the antilog of the sum is zero (since the zero entries are stored from location $2n-1$ onwards in the *alog* [ ]).

The multiplicative inversion can also be found using table look-up method. The inverse of an element β based on log/alog tables is

$$1/\beta = alog\ [\, n - \log [\, \beta \,]\,] \tag{7.5}$$

provided that $\beta \neq 0$. It is also possible to implement a new table which gives the inverse of an element, the entries in the inverse table are multiplicative inverses of the Galois field elements. The inverse values are preprocessed and stored in the memory as,

$$1/\beta = 1/\alpha^i = \alpha^{-i} = \alpha^{n-i} . \tag{7.6}$$

The size of inverse table is $m \times 2^m$ bits.

In software RS decoders, the table look-up method can be used easily. It is only necessary to preprocess two tables for multiplication (log and alog) and one table for inversion (inverse table). These tables should be stored in the permanent memory of the system. For multiplication of two elements, three memory references and for each inversion only one memory reference are needed.

The table look-up method which was explained in this section, can be implemented for both standard and normal basis representation of the elements of GF(2^m). The choice of the basis or the generator polynomial of the field does not affect the performance or complexity of table look-up multipliers or inverters.

In applications where the symbol size (m) is small, a direct table look-up can be used for multiplication [43]. In the direct method, multiplication of two elements requires a memory space of $m \times 2^{2m}$ bits. As an example the size of memory for GF(2^5) is 1K table of 5-bit symbols.

7.2. Comparison of Decoding Algorithms

To design software decoders, the following two main points should be discussed which are

1) The decoding algorithm and the software implementation.

2) ! Microprocessor type and hardware implementation.

In this section, we will only compare the decoding algorithms and discuss the software implementation. We will not discuss the second topic here and will refer the reader to [22], where the hardware implementation based on general purpose processors, bit slice processors, and special purpose processors are discussed in detail.

The choice of the algorithm affects the versatility of the software implementation. As we discussed earlier, time-domain algorithms have a very good structure for versatile decoders. Obviously, writing a software using these algorithms for versatile decoders is much easier than using the syndrome-based algorithms. However, this point is not important since even a badly structured algorithm can be written in software with a good structure. This can be done by writing different subroutines and macros, and structuring the program. In writing the software, because of the low cost of memory devices, we can use a large amount of ROM for storing the source code, and avoid branches in the program. Avoiding branches in the program speeds up the decoder by a large factor, at the expense of increased memory requirement.

The comparison of algebraic and transform decoding algorithms based on number of multiplications has been presented in [22], and [50]. In [50], the number of multiplications for the algebraic and transform decoding algorithms is given as, $n(4t-1)+5t^2+3t-1$, and $(n-1)[I(m)+2]+t(n+3)-1$, respectively. These values are for the worst case condition, and hence the error only case has been considered. Note that $I(m)$ is the number of irreducible polynomials of degree higher than 1 in GF(2) having a root in GF(2^m). The cross over of these two values, occurs at

$$t^* = (-3n + \sqrt{9n^2 + [20I(m)+60]n - 20[I(m)+2]})/10. \qquad (7.7)$$

If we consider, t-error correcting RS(255,k) code defined in GF(2^8), the cross over happens at $t^* = 11.43$ ($I(8) = 34$). For values of $t \leq 11$, the algebraic decoding is faster and for $t > 11$ the transform one.

Now, let's evaluate the number of multiplications for the time-domain decoding algorithms. To have the worst case we will consider the error only algorithms.

The time-domain algorithm based on transform decoder, shown in Fig. 4.1, has three steps, i.e., calculating the error locations, finding the error values, and performing the correction.

In the first step, $2t$ iterations are performed to evaluate the error locator vector $\underline{\lambda}$. In each iteration, we need to evaluate $\underline{s}$, Δ, $\underline{\lambda}$ and $\underline{b}$. In the worst case, these values can be calculated with $5n$ multiplications per iteration, and in total $10tn$ multiplications are needed.

The second step of the algorithm has $n-2t$ iterations, and in each iteration we should calculate $\underline{s}$ and Δ. Calculation of these values needs $2n$ multiplications in each iteration and in total $4tn$.

The error correction step has no multiplications. So, the total number of multiplications for the time domain decoding algorithm based on transform decoder is $14tn$.

The time domain algorithm based on algebraic decoder has two steps. In the first step, there are $2t$ iterations and the values of Δ, $\underline{\lambda}$, $\underline{b}$, $\underline{\lambda}'$, $\underline{b}'$, $\underline{\omega}$ and $\underline{a}$ should be evaluated. Evaluation of these values needs $12n$ multiplications in each iteration and $24tn$ in total. Considering the error correction step which has $2n$ multiplications, the total number of multiplications for the time domain decoding based on algebraic decoder is $24tn+2n$. The cross over point occurs at $t^*=-0.1$. That is, there is no cross over for positive t.

To compare the syndrome-based algorithms with the time domain ones, consider RS(255, k) code defined in GF(2^8). The number of multiplications needed for all the decoding algorithms versus t, is drawn in Fig. 7.1. Figure 7.1 shows that the syndrome-based decoders are much faster than time-domain ones. For high rate codes ($t \leq 11$), algebraic decoding and for low rate ones ($t \geq 12$) transform decoding are the fastest algorithms. Therefore, in software decoders, syndrome-based algorithms should be considered.

7.3. Software Evaluation of the Roots of Polynomials

In this section, a fast method is introduced to find the roots of polynomials. This approach can be used in software RS decoders for high-rate codes.

Let's consider a polynomial of degree t defined over GF(2^m),

$$P_t(x)=x^t+a_{t-1}x^{t-1}+\ldots+a_1x+a_0. \tag{7.8}$$

This polynomial has $\nu \leq t$ roots in GF(2^m).

The only general way for finding the roots of $P_t(x)$ is simply to evaluate this polynomial for all elements of the field and comparing the result with zero (Chien search). This procedure is used for finding the roots of polynomials since 1964 [17]. In Chien search, which is one of the steps of the algebraic decoding algorithm, $P_t(x)$ should be calculated and compared with zero 2^m times. In software implementation, this large number of calculations of the polynomial (large m), takes a long time.

Another way for finding the roots of $P_t(x)$ is to use look-up tables. In this approach, since coefficients of the polynomial in Eq. (7.8) contain all the information about the roots, we can use these coefficients as addresses and the roots of polynomial as entries of a look-up table. This approach is very fast but needs a large look-up table even for low degree polynomials. Therefore, in most cases, it is not practical to use this approach and it is only used for low degree polynomials in low order finite fields such as GF(2^4).

To reduce the size of look-up tables, the polynomials can be translated into other polynomials using a linear translation [16],[67]. In this section, roots of the polynomials are studied for different degrees based on the translated polynomials. At the end, a fast algorithm is presented based on the translated polynomials.

7.3.1. Linearly Translated Polynomials

For finding the roots of the polynomial $P_t(x)$ defined in GF(2^m), this polynomial can be translated into another polynomial [16], [58], [67], using the linear translation,

$$x = u_1 z + u_0 \ . \tag{7.9}$$

By substituting Eq. (7.9) into the polynomial $P_t(x)$ and simplifying the polynomial we can find a new polynomial $Q_t(z)$ with the same number of roots,

$$Q_t(z) = z^t + b_{t-1} z^{t-1} + \ . \ . \ . + b_1 z + b_0 . \tag{7.10}$$

The coefficients of $Q_t(z)$ are functions of coefficients of $P_t(x)$ and the translation coefficients u_1, and u_0. Therefore, it is possible to find u_1 and u_0 as functions of coefficients of $P_t(x)$ such that two of the coefficients of $Q_t(z)$ are constant values. Using this method, i.e., by determining u_1 and u_0 appropriately, $Q_t(z)$ will have $t-2$ coefficients where $P_t(x)$ has t coefficients. This decrease in the number of coefficients is advantageous in both general ways of finding the roots, i. e., Chien search and look-up table method.

In the following subsections, this approach is discussed in detail for different degrees of the polynomial $P_t(x)$. It is also compared to two general methods of finding the roots of polynomials.

Let's consider the second degree polynomial $P_2(x)$ defined in GF(2^m) as,

$$P_2(x)=x^2+a_1x+a_0. \tag{7.11}$$

The appropriate linear translation is,

$$x=a_1z \quad , \quad a_1\neq 0, \tag{7.12}$$

which yields,

$$Q_2(z)=z^2+z+b_0 \quad , \quad b_0=a_0/a_1^2. \tag{7.13}$$

The polynomial $Q_2(z)$ has just one coefficient, b_0, and hence for finding roots of $Q_2(z)$ a smaller look-up table is needed in comparison to $P_2(x)$. The size of the table for finding each root of $P_2(x)$ is 2^{2m} bytes, but for $Q_2(z)$ it is just 2^m bytes. So, using this method, the size of look-up table for finding the roots decreases by a factor of 2^m.

After finding the roots of $Q_2(z)$, the roots of $P_2(x)$ can be calculated using Eq. (7.12). When $a_1=0$, we have $P_2(x)=x^2+a_0$, and there is no need for the linear translation. In this case, root of $P_2(x)$ is equal to $\sqrt{a_0}$.

For finding the roots of $P_2(x)$ using this method, the following steps should be performed:

a) If $a_1=0$ go to step "e".

b) Calculate $b_0=a_0/a_1^2$.

c) Apply b_0 to the look-up table and find one of the roots z_1. The other root is $z_2=1+z_1$.

d) Calculate $x_i=a_1z_i$, $i=1,2$, and go to step "f".

e) Calculate $x_1=x_2=\sqrt{a_0}$ using a table of square root.

f) Stop.

Let's consider the third degree polynomial $P_3(x)$ defined in GF(2^m) as,

$$P_3(x)=x^3+a_2x^2+a_1x+a_0. \tag{7.14}$$

The appropriate linear translation is,

$$x=(a_2+\sqrt{a_1})z+\sqrt{a_1} \quad , \text{ if } \quad a_2\neq\sqrt{a_1}, \tag{7.15}$$

and

$$x=z+\sqrt{a_1} \quad , \text{ if } \quad a_2=\sqrt{a_1}, \tag{7.16}$$

which yields,

$$Q_3(z)=z^3+z^2+b_0 \quad , \text{ if } \quad a_2\neq\sqrt{a_1}, \tag{7.17}$$

where,

$$b_0=(a_2a_1+a_0)/(a_2+\sqrt{a_1})^3,$$

and,

$$Q_3(z)=z^3+(a_2a_1+a_0), \quad \text{if} \quad a_2=\sqrt{a_1}. \tag{7.18}$$

For the case of $a_2=\sqrt{a_1}$, roots of $Q_3(z)$ can be found easily using Eq. (7.18). When $a_2\neq\sqrt{a_1}$, since, $Q_3(z)$ has only one coefficient, each root of $Q_3(z)$ can be found using a look-up table with a size of 2^m bytes. In the case of using $P_3(x)$ and applying look-up table method, the size of table would be 2^{3m} bytes because of having three coefficients. So, using this method the size of look-up table is reduced by a factor of 2^{2m}.

For finding roots of $P_3(x)$ using this approach the following steps should be performed:

a) If $a_2=\sqrt{a_1}$ go to step "d".

b) Calculate $b_0=(a_2a_1+a_0)/(a_2+\sqrt{a_1})^3$.

c) Apply b_0 to the look-up table and find the roots z_1, z_2, and z_3. Calculate

$x_i = (a_2 + \sqrt{a_1})z_i + \sqrt{a_1}$, $i = 1, 2, 3$ and stop.

d) Solve $z^3 = a_2 a_1 + a_0$ to find z_1, z_2 and z_3. Calculate $x_i = z_i + \sqrt{a_1}$ for $i = 1,2,3$ and stop.

This method is much faster than Chien search, since in Chien search the polynomial should be calculated 2^m times and compared with zero, but in this method very few calculations are required.

For higher degree polynomials we can apply similar linear translation we used for the second and third degree polynomials. As a result of this translation, the number of coefficients of a t-degree polynomial decreases from t to $t-2$. So, in the look-up table approach for finding each root, the size of look-up table is decreased by a factor of 2^{2m}. For example this reduction factor in GF(2^8) is $2^{16} = 64k$.

Note that even by using this method the size of the look-up table for high degree polynomials becomes large for high order fields. In this case, the only way to find the roots is the Chien search. In the next subsection, an algorithm is introduced which is a mixture of Chien search and the look-up table method. It will be shown that using this algorithm [68], roots of high degree translated polynomials can be found much faster than conventional Chien search.

7.3.2. Segmented Search Algorithm

Let's consider a translated polynomial of degree t, having $t-2$ coefficients. In the segmented search algorithm, first the field elements are divided into two or more segments and the segment which contains most of the roots is identified. Then a Chien search is applied to that segment and the corresponding roots are found. For finding the rest of the roots, the degree of the polynomial is reduced and then Chien search is applied to the second segment which has more roots compared to rest of the segments. After finding new set of roots, the procedure is continued as before. In this algorithm, to decide where to apply the Chien search at each step, a look-up table is preprocessed and stored in the permanent memory of the system. Each entry of this look-up table indicates the segment of the field which contains most of the roots. Since coefficients of the translated polynomial have all the information about the roots, they are used as addresses to these entries.

To explain the look-up table in detail, let's consider the GF(2^m). Now we divide n into l equal segments. Number of segments, l, is an integer power of two, which is defined as,

$$l = 2^i . \tag{7.19}$$

So, the length of each segment is 2^{m-i} elements. For preprocessing the table, for each combination of $t-2$ coefficients, a Segment-Identifier (SI) value is assigned which indicates the segment with most of the roots. Each SI entry in the table is i bits long, where i is given in Eq. (7.19). Since each coefficient of $Q_t(z)$ is represented by m bits, the size of SI table is equal to $i.\ 2^{m(t-2)}$ bits.

Now, the segmented search algorithm based on preprocessed SI table can be explained in detail using the flowchart shown in Fig. 7.2. For explaining this algorithm, we assume that there are ν roots ($\nu \leq t$), and the translated polynomial is formed as explained. In the flowchart of Fig. 7.2, first $t-2$ coefficients of the polynomial $Q_t(z)$ are applied to the SI table and the segment which has most of the errors is identified. Then $Q_t(z)$ is calculated for all the elements of the segment and the roots in this segment is found (j roots). Then the degree of the polynomial is reduced to $t-j$ and a new polynomial $Q_{t-j}(z)$ is formed. If the degree of the new polynomial is zero ($t-j=0$), then there is no more roots left and the search is stopped. If the degree is not equal to zero, the algorithm shown in Fig. 7.2 is repeated for the new polynomial.

The execution times of the Chien search and the new algorithm can be compared using Fig. 7.2. For comparison, we consider the worst case only. In Chien search, in the worst case, the translated polynomial of degree t should be calculated and compared with zero, 2^m times.

In Tables 7.1 and 7.2, estimated number of Central-Processing-Unit (CPU) cycles and memory size, for Chien search (I) and the algorithm of previous sub-section (II) are given. As an illustrative example, Intel 8086 microprocessor and three Reed-Solomon codes defined in GF(2^8) are used. The codes are RS(255, 251), RS(255, 249), and RS(255, 247) which can correct two, three, and four errors, respectively. Note that the degree, t, of the error locator polynomial for each code is the same as error correcting capability.

Table 7.1: Worst Case Number of CPU cycles for calculating the error locations

Codes	t	I	II
RS(255,251)	2	13000	500
RS(255,249)	3	20000	600
RS(255,247)	4	35000	1500

Table 7.2: Required memory size (Kbytes) for calculating the error locations

Codes	t	I	II
RS(255,251)	2	5.5	1.
RS(255,249)	3	8.	1.5
RS(255,247)	4	14.	65

Tables 7.1 and 7.2 show that for these codes the method of previous subsection (II) is about 20 to 30 times faster than the Chien search (I). But in the case of 4-error correcting code, RS(255, 247), the memory size becomes very large.

Table 7.3 compares the number of CPU cycles and memory size for the following algorithms.

I) The Chien search.

II) Using translated polynomial and look-up table method.

III) Using translated polynomial and segmented search algorithm with $l=2$.

Table 7.3: Comparison of three algorithms for 4-error correcting RS(255, 247) code

RS(255,247)	I	II	III
# of CPU cycles	35000	1500	8000
memory size (Kbytes)	14	65	12

This table shows that for the 4-error correcting RS(255, 247) code, the best algorithm is the segmented search algorithm applied on translated polynomial (III). This algorithm, using almost the same memory space as the Chien search (I), finds the error locations almost four times faster.

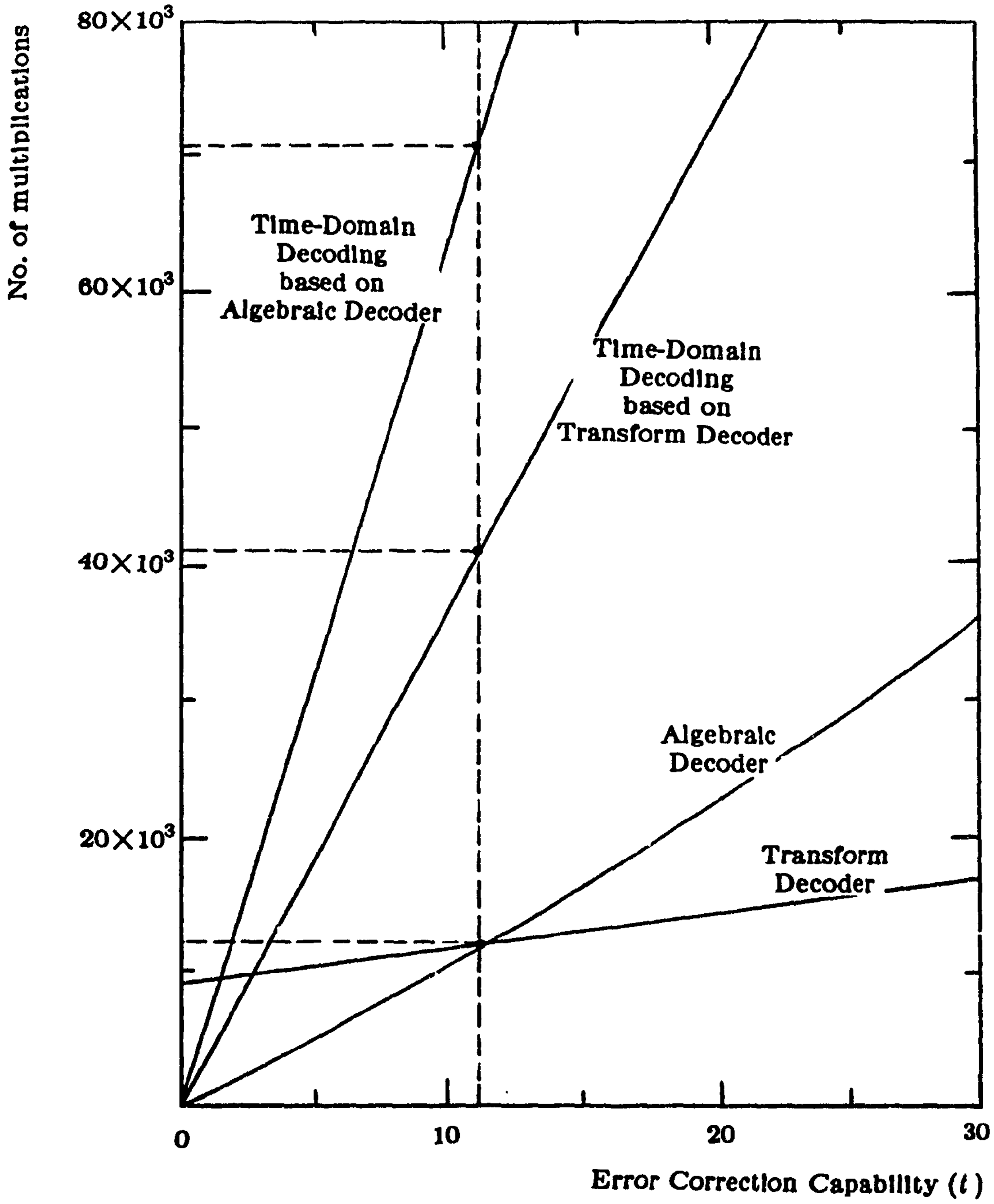


Figure 7.1: Comparison of Decoding Algorithms

for Software Decoders

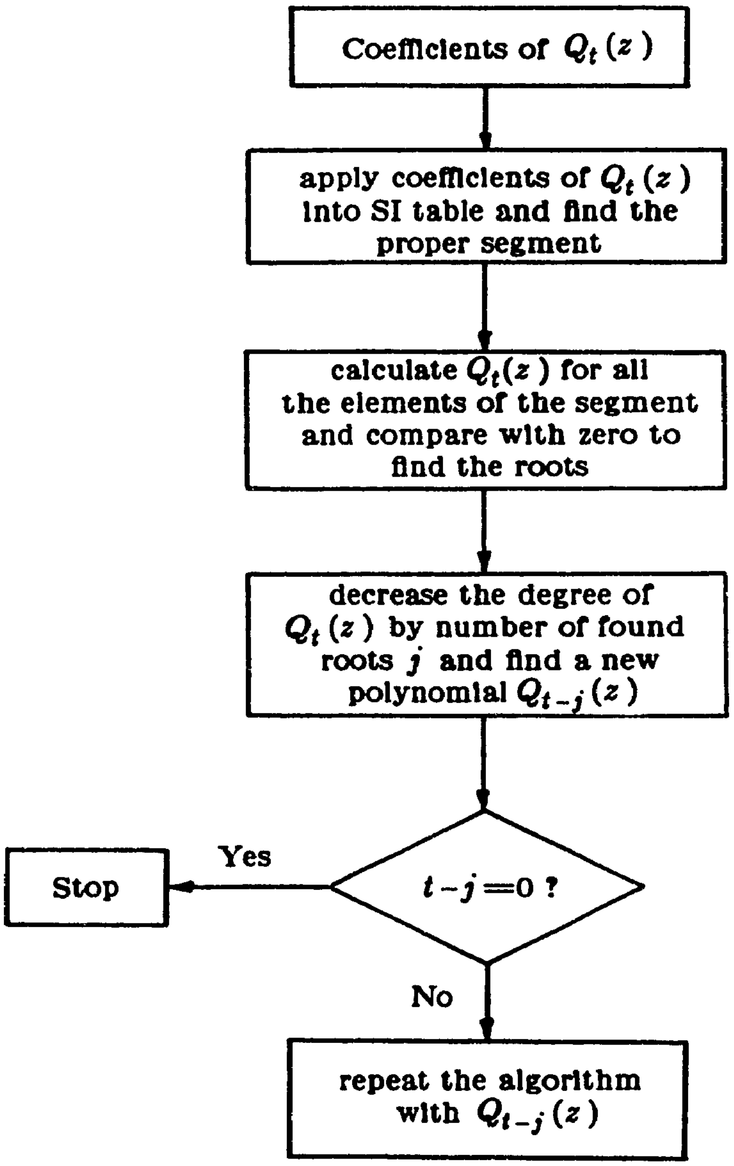

Coefficients of $Q_t(z)$
apply coefficients of $Q_t(z)$ into SI table and find the proper segment
calculate $Q_t(z)$ for all the elements of the segment and compare with zero to find the roots
decrease the degree of $Q_t(z)$ by number of found roots j and find a new polynomial $Q_{t-j}(z)$
$t-j=0$?
Yes
Stop
No
repeat the algorithm with $Q_{t-j}(z)$

Figure 7.2: Segmented Search Algorithm for Finding Roots of Polynomials

CHAPTER EIGHT

CONCLUSION AND FUTURE WORK

8.1. Conclusion

This thesis presents a contribution to the effort that has been made by a number of researchers to make simple and fast Reed-Solomon decoders. Our goal has been to introduce versatile Reed-Solomon decoder structures which can be used for decoding a large number of RS codes with different parameters.

In this thesis, after a survey of the existing decoder structures, Reed-Solomon decoding algorithms for correcting both errors and erasures were considered. It was shown that from the structural point of view, time-domain algorithms are good candidates for designing versatile Reed-Solomon decoders. The time domain decoding algorithms were modified and versatile decoders were introduced.

Reed-Solomon codes are defined over GF(2^m). In decoding these codes, algebraic operations are required. These operations were studied and least complex multiplier structures over Galois fields were introduced both in standard and normal basis. It was shown that the complexity of the standard basis multiplier is less than that of the one in normal basis. A new multiplier in normal basis was introduced which is less complex compared to the Massey-Omura multiplier. A universal multiplier over GF(2^m) was also introduced which could be configured for m =4,5,6,7,8. It was shown that the inverter structure in standard basis is very complex, and, therefore table look-up should be used.

In normal basis, the inverter can be designed using a few multipliers and this is the main advantage of normal basis over standard basis in the design of RS decoders.

A versatile Reed-Solomon decoder, capable of decoding any RS(n,k) code in a specific Galois field was presented. The time-domain decoding algorithm based on transform decoder was used to introduce the decoder. The structure of the versatile decoder is such that it can be programmed to correct errors and erasures for different RS codes. As an illustrative example, a Gate-Array-Based design of the decoder in GF(2^5) was also given. To compare the syndrome-based algorithms and time-domain ones, implementation of the Berlekamp-Massey algorithm is considered. The Berlekamp-Massey algorithm is one of the steps of the syndrome-based algorithms which evaluates the error locator polynomial. It was shown that the hardware required for the implementation of this step alone, is more than that of the total structure of the introduced versatile decoder based on time-domain algorithms.

A new cellular structure for a versatile Reed-Solomon decoder was presented. The time domain decoding algorithm was restructured in order to make it suitable for introducing the cellular structure. The structure of the cellular decoder is such that it can be programmed to correct errors and erasures for fixed block lengths n and fixed symbol size m for different RS codes in GF(2^m), by changing the length k. The structure of this decoder is very simple, cellular and hence easy to implement using VLSI. It was shown that this decoder is much simpler compared to the decoders based on algebraic and transform decoding algorithms. The maximum bit rate at the input of the decoder is very high and is equivalent to $\frac{m}{(5m+8)\tau}$ [bits/sec.] where τ [sec.] is the delay of one gate.

The time-domain algorithm based on algebraic decoder was modified to reduce one out of two inverters, and then a structure for the universal decoder was given. The universal decoder is capable of decoding any RS code defined over GF(2^m), m =4,5,6,7,8. Complexity and throughput of this universal decoder was discussed. To increase the versatility of the universal decoder a method was introduced for decoding an RS code generated by any generator polynomial. The design of the extra hardware required for applying this method was also given.

It was shown that the structure of the decoding algorithm is not an important factor in the choice of the algorithm for software decoders. Therefore, decoding algorithms were compared from speed point of view and it was shown that syndrome-based decoding algorithms are better choices. High rate RS codes were considered and a method was introduced which increases the speed of the software decoder for these codes.

8.2. Future Work

As explained in Chapter 2, the time-domain algorithms are time-domain equivalents of syndrome-based ones. In this thesis, for finding the time-domain algorithms, the Berlekamp-Massey algorithm was transformed to time-domain. The Berlekamp-Massey algorithm is used for the evaluation of the error locator polynomial in syndrome-based algorithms. However, there are other algorithms for finding this polynomial, such as the Euclidean and continued fractions algorithms. Therefore, one direction for future research would be the modification of these algorithms in order to transform them into time domain. After this transformation new time-domain algorithms can be used to design Reed-Solomon decoders. New structures can be developed and compared with the time-domain structures introduced in this thesis.

As explained in this thesis, normal basis multipliers are more complex than standard basis ones. So, standard basis should be used in structures, such as the cellular structure of Chapter 5, where a large number of multipliers are required. On the other hand, the standard basis has the problem of very complex inverters. Two approaches can be considered to solve this problem. One way is to modify the Reed-Solomon decoding algorithms to omit the only inversion available in the algorithms. The other approach is to introduce a new structure for the inversion circuitry with low complexity.

REFERENCES

[1] I. S. Reed, and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. Soc. Ind. Appl. Math.*, 8, pp. 300-304, June 1960.

[2] H. Kummer, "*Recommendation for Space Data System Standards: Telemetry Channel Coding: Issue-1,*" Consult. Comm. Space Data Syst., Sept. 1983.

[3] K. Y. Liu, and J. J. Lee, "Recent Results on the use of Concatenated Reed-Solomon/Viterbi Channel Coding and Data Compression for Space Communications," *IEEE Trans. Commun.*, vol. Com-32, pp. 518-23, May 1984.

[4] L. B. Milstein, S. Davidovici, and D. L. Schilling, "Coding and Modulation Techniques for Frequency-Hoped Spread-Spectrum Communications over a Pulse-Burst Jammed Rayleigh Fading Channel," *IEEE J. Sel. Areas in Commun.*, vol. SAC-3, pp. 644-51, Sept. 1985.

[5] M. B. Pursley, and W. E. Stark, "Performance of Reed-Solomon Coded Frequency-Hop Spread-Spectrum Communications in Partial-Band Interference," *IEEE Trans. Commun.*, vol. Com-33, pp. 767-74, Aug. 1985.

[6] S. M. Krone, and D. V. Sarwate, "Quadriphase Sequences for Spread-Spectrum Multiphase-Access Communication," *IEEE Trans. Inform. Theory,* vol. IT-30, pp. 520-9, May 1984.

[7] A. W. Lam, and D. V. Sarwate,"Time-Hopping and Frequency-Hopping Multiple-Access Packet Communications", *IEEE Trans. Commun.*, vol. COM-38, pp. 875-888, June 1990.

[8] B. Vyden, and M. J. Miller, "Error Correction Coding for Industrial Data Communication Channels," *Conf. Proc. Tencon'84*, pp. 47-49, 1984.

[9] Y. R. Shayan, T. Le-Ngoc, and V. K. Bhargava, "Design of (16,12) Reed-Solomon Codec for North American Advanced Train Control Systems", *IEEE Trans. Veh. Tech.*, vol. 39, pp. 400-409, Nov. 1990.

[10] B. C. Mortimer, M. J. Moore, and M. Sablatash, "The Design of a High-Performance Error-Correcting Coding Scheme for the Canadian Broadcast Telidon System Based on Reed-Solomon Codes," *IEEE Trans. Commun.* vol. COM-35, pp. 1113-1123, Nov. 1987.

[11] J. B. H. Peek, "Communications Aspects of Compact Disc Digital Audio System", *IEEE Commun. Mag.*, pp. 7-15, Feb. 1985.

[12] Y. Aoki, T. Ihashi, N. Sato, and S. Miyaoka, "A Magneto-Optic Recording System," *IEEE Trans. Magn.*, Mag-21, pp. 1624-8, Sept. 1985.

[13] S. W. Golomb, "Optical Disk Error Correction", *Byte,* 11, pp. 203-210, May 1986.

[14] E. R. Berlekamp, "Bit-Serial Reed-Solomon Encoders," *IEEE Trans. Inform. Theory,* vol. IT-28, pp. 869-874, Nov. 1982.

[15] I. S. Hsu, I. S. Reed, T. K. Truong, K. Wang, C. S. Yeh, and L. J. Deutsch, "The VLSI Implementation of a Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm," *IEEE Trans. Comput.*, vol. C-33, pp. 906-911, Oct. 1984.

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

[17] R. T. Chien. "Cyclic Decoding Procedures for Bose-Chaudhuri-

Hocquenghem Codes," *IEEE Trans. Inform. Theory,* vol. IT-10, pp. 357-363, Oct. 1964.

[18] W. W. Peterson,"Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes", *IEEE Trans. Inform. Theory.,* vol. IT-6, pp. 459-470, 1960.

[19] J. L. Massey, "Shift-Register Synthesis and BCH Decoding," *IEEE Trans. Inform. Theory,* vol. IT-15, pp. 122-127, Jan. 1969.

[20] W. W. Peterson, and E. J. Weldon, Jr., *Error Correcting Codes.* Cambridge, Mass.: MIT Press, 1972.

[21] E. R. Berlekamp, "*Galois Field Computer*", U.S. Patent no. 4,162,480, July 24, 1979.

[22] E. T. Cohen, "*On the Implementation of Reed-Solomon Decoders*", Ph.D. dissertation, University of California, Berkeley, May 1983.

[23] Y. R. Shayan, and T. Le-Ngoc, "On Software Evaluation of the Roots of Polynomials Defined in Galois Fields", *AEU Proc.*, to be published.

[24] H. M. Shao, and I. S. Reed, " On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays," *IEEE Trans. Comput.*, vol. 37, pp. 1273-1280, Oct. 1988.

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

[26] S. Winograd, "On Computing the Discrete Fourier Transform", *Proc. Natl. Acad. Sci.,* 73, pp. 1005-6., 1976.

[27] I. S. Reed, and T. K. Truong, "Fast Mersenne-Prime Transforms for Digital Filtering," *IEE Proc.,* 125, pp. 433-440, 1978.

[28] T. K. Truong, R. L. Miller, and I. S. Reed, "Fast Technique for Computing Syndromes of BCH and Reed-Solomon Codes", *Electronics Letters*, vol. 15, pp. 720-1, Oct. 1979.

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

[30] H. M. Shao, T. K. Truong, L. J. Deutsch, J. H. Yuen, and I. S. Reed, "A VLSI Design of a pipeline Reed-Solomon Decoder," *IEEE Trans. Comput.*, vol. C-34, pp. 393-403, May 1985.

[31] B. L. Johnson, "Design and Hardware Implementation of a Versatile Transform Decoder for Reed-Solomon Codes," *Impact of Processing Techniques on Communications (J. K. Skwirzynski, Editor)*, Proc. of the Nato Advanced Study Institute, pp. 447-464, 1985.

[32] R. E. Blahut, "A Universal Reed-Solomon Decoder," *IBM J. Res. Develop.*, vol. 28, pp. 150-158, Mar. 1984.

[33] J. H. Conway, "A Tabulation of Some Information Concerning Finite Fields", *Comput. Math. Res.*, Churchhouse and Herz, eds., North-Holland, Amsterdam, 1968.

[34] P. A. Scott, S. T. Tavares, and L. E. Peppard, "A Fast VLSI Multiplier for GF(2^m)", *J., Sel. Areas Commun.*, vol. SAC-4, pp. 62-63, Jan. 1986.

[35] T. Beth, and D. Gollmann, "Algorithm Engineering for Public Key Algorithms", *IEEE J. Sel. Areas Commun.*, vol. 7, pp. 458-466, May 1989.

[36] B. A. Laws, and C. K. Rushforth, "A Cellular-Array Multiplier for GF(2^m)", *IEEE Trans. Comput.*, vol. C-20, pp. 1573-1578, Dec. 1971.

[37] C. S. Yeh, I. S. Reed, and T. K. Truong, "Systolic multipliers for Finite Fields GF(2^m)", *IEEE Trans. Comput.*, vol. C-33, pp. 357-360, Apr. 1984.

[38] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed, "VLSI architectures for Computing Multiplications and Inverses in GF(2^m)", *IEEE Trans. Comput.*, vol. C-34, pp. 709-717, Aug. 1985.

[39] J. L. Massey, and J. K. Omura, USA Patent Application of *Computational Method and Apparatus for Finite Field Arithmetic*, Submitted in 1981.

[40] A. M. Patel, "On-the-fly Decoder for Multiple Byte Errors", *IBM J. Res. Develop.*, vol. 30, pp. 259-269, May 1986.

[41] M. Lagasse, "VLSI Design of a GF(2^m) Parallel Galois Field Multiplier", ELEC 465 Project Report, Univ. Victoria, Victoria, B. C., Apr. 1989.

[42] Y. R. Shayan, and T. Le-Ngoc, "The Least Complex Parallel Massey-Omura Multiplier and its LCA and VLSI Designs", *IEE Proc.*, vol. 136, pt. G, pp. 345-349, Dec. 1989.

[43] M. H. Kang, "Software Algorithms for the Reed-Solomon Code," M. S. Thesis, Dept. of Elect. and Comput. Eng., Univ. of Illinois at Urbana-Champaign, Sept. 1985.

[44] R.C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, "Optimal Normal Bases in GF(p^n)", *Discrete Applied Mathematics 22*, Elsevier Science Publishers B. V., North-Holland, 1988.

[45] T. C. Bartee, and D. I. Schneider, "Computation with Finite Fields", *Inform. Contr.*, vol. 6, pp. 79-98, Mar. 1963.

[46] I. S. Hsu, T. K. Truong, L. J. Deutsch, and I. S. Reed, "A Comparison of VLSI Architecture of Finite Field Multipliers Using Dual, Normal, or Standard Bases", *IEEE Trans. Comput.*, vol. 37, pp. 735-739, June 1988.

[47] H. O. Burton, "Inversionless Decoding of Binary BCH Codes", *IEEE Trans. Inform. Theory,* vol. IT-17, pp. 464-466, July 1971.

[48] C. L. Chen, "High-speed Decoding of BCH Codes", *IEEE Trans. Inform. Theory,* vol. IT-27, pp. 254-256, Mar. 1981.

[49] Y. R. Shayan, and T. Le-Ngoc, "A Cellular Time-Domain Reed-Solomon Decoder", *Conf. Proc. ICCS'90,* Singapore, Nov. 5-9, 1990.

[50] R. S. Brunton, "*A Comparative Analysis of Reed-Solomon Decoding Techniques*", Master Thesis, University of Waterloo, Waterloo, Ontario, 1981.

[51] A. Hocquenghem, "Codes corecteurs d'erreurs," *Chiffres,* vol. 2, pp. 147-156, 1959.

[52] R. C. Bose, and D. K. Ray-Chaudhuri, "on a Class of Error Correcting Binary Group Codes," *Inform. Control,* vol. 3, pp. 68-79, Mar. 1960.

[53] D. Gorenstein, and N. Zierler, "A Class of Cyclic Linear Error-Correcting Codes in p^m Symbols," *J. Soc. Ind. Appl. Math.*, 9, pp. 107-214, June 1961.

[54] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A Method for Solving Key Equations for Decoding Goppa Codes," *Inform. and Control,* vol. 27, Jan. 1975.

[55] I. S. Reed, R.A. Scholtz, T. K. Truong, and L. R. Welch, "The Fast Decoding of Reed-Solomon Codes Using Fermat Theoretic

Transforms and Continued Fractions," *IEEE Trans. Inform. Theory,* vol. IT-24, pp. 100-106, 1978.

[56] G. D. Forney, "On Decoding BCH Codes," *IEEE Trans. Inform. Theory,* vol. IT-11, pp. 549-557, Oct. 1965.

[57] W. C. Gore, *Transmitting Binary Symbols with Reed-Solomon Code,* Johns-Hopkins, EE report no. 73-5, Apr. 1973.

[58] F. Polkinghorn, Jr.: 'Decoding of Double and Triple Error Correcting Bose-Chaudhuri Codes", *IEEE Trans. Inform. Theory,* vol. IT-12, pp. 480-481, Oct. 1966.

[59] T. K. Truong, W. L. Eastman, I. S. Reed, and I. S. Hsu, "Simplified Procedure for Correcting both Errors and Erasures of Reed-Solomon Code Using Euclidean Algorithm", *IEE Proc.*, vol. 135, pt. E, pp. 318-324, Nov. 1988.

[60] LSI Logic Corp., *Channel-Free Array Design Manual,* LSI Logic Corp., CA., 1989.

[61] D. M. Mandelbaum,"Decoding of Erasures and Errors for Certain RS Codes by Decreased Redundancy", *IEEE Trans. Inform. Theory,* vol. IT-28, pp. 330-336, Mar. 1982.

[62] Y. R. Shayan, T. Le-Ngoc, and V. K. Bhargava, "A Versatile Time-Domain Reed-Solomon Decoder", *IEEE J. Sel. Areas in Commun.*, vol. 8, pp. 1535-1542, Oct. 1990.

[63] T. Le-Ngoc, M. Vo, B. Mallett, and V. K. Bhargava, "A Gate-Array-Based Programmable Reed-Solomon Codec: Structure, Implementation, Applications," *Proc. Milcom'90 Conf.*, Monterey, CA., Sept. 30 - Oct. 3, 1990.

[64] P. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-*

Correcting Codes. Amsterdam: North-Holland Publishing, 1978.

[65] J. K. Wolf, "Adding Two Information Symbols to Certain Non-binary BCH Codes and Some Applications," *Bell Syst. Tech. J. 48*, pp. 2405-2424, 1969.

[66] Y. R. Shayan and T. Le-Ngoc, "Decoding Reed-Solomon Codes Generated by any Generator Polynomial", *Electronics Letters*, Vol. 25, pp. 1223-1224, Aug. 1989.

[67] W. S. Burnside, and A. W. Panton, *The Theory of Equations*. New York: Dover, 1960.

[68] Y. R. Shayan, T. Le-Ngoc, and V. K. Bhargava, "A Binary-Decision Approach to Fast Chien Search for Software Decoding of BCH Codes", *IEE Proc.*, vol. 134, pt. F, pp. 629-632, Oct. 1987.

[69] D. V. Sarwate, and R. D. Morrison,"Decoder Malfunction in BCH Decoders", *Trans. Inform. Theory,* vol. IT-36, pp. 884-889, July 1990.

[70] A. R. Michelson, and A. H. Levesque, *Error-Control Techniques for Digital Communication*. New York: John Wiley and Sons, 1985.

[71] *The Programmable Gate Array Design Handbook*, Xilinx Inc., San Jose, Calif., 1986.