NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

**An Assessment of
VLSI and Embedded Software Implementations
for
Reed-Solomon Decoders**

by

Ted S. Fill

A Thesis submitted in conformity with the requirements
for the degree of Master of Applied Science,
Edward S. Rogers Sr. Department of Electrical and Computer Engineering,
University of Toronto

© Copyright by Ted S. Fill 2001

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-63016-1

Canada

**An Assessment of
VLSI and Embedded Software Implementations
for
Reed-Solomon Decoders**

Ted Stanley Fill

Master of Applied Science, 2001

Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

Abstract

Reed-Solomon decoders are used extensively in numerous applications ranging from cellular telephones to deep-space communications. This thesis examined Reed-Solomon time-domain and frequency-domain decoder implementations in both software and hardware. Thus far, there have been no clear, definitive statements in the published literature about the relative merits and limitations of each type of decoder implementation. In response, a detailed comparison is presented through tangible results from dedicated hardware and software implementations. The focus was on designing area-efficient, low-power and low-complexity decoders suitable for today's moderate rate applications. Two decoder chips were designed in a $0.18\mu\text{m}$ CMOS process and they targeted a decoding rate of 160 Mbps. The time-domain decoder was fabricated and had a core area of 1.50 mm^2 and an overall silicon die area of 3.54 mm^2 .

Acknowledgments

At times it seemed unreachable and unending, but this thesis is finally complete. It would not have been possible without the help and generosity of many people.

Sincere thanks to my advisor Professor Glenn Gulak for his ideas, support and encouragement throughout this thesis. Thank-you Glenn for your guidance and advice.

Financial assistance from NSERC as well as fabrication support from CMC were greatly appreciated.

Special mention goes out to Kostas Pagiamtzis for all his help with the memory cores and suggestions for the design flow. Thanks Kos. I would also like to thank Vincent Gaudet for his much appreciated help and with the chip testing . May the West be strong.

Thanks to all my friends in PT392 and EECG including: Ahmad, Ajay, Amy, Andy, Dave, Derek, Elias, Guy, Leslie, Mark, Marcus, Nirmal, Paul, Peter, Roman, Shahriar, Sirish, Tor, Tooraj, Warren, William, and Yadi.

Peace to the Westside UofA boyz from Edmonton: Brad, Dan, Erik, Jax, Kelly, Matt, Michael, Rob, and Vinesh. City of Champions forever.

Most of all, I am extremely grateful to my family for their love, understanding and support. It's finally done! Thanks Gerty for all your help and all that you have done for me in Toronto. To my sisters Carolin and Teresa: thanks so much for your continued support which helped me get through this. I would especially like to express my utmost sincere appreciation, thanks and love to my Mom and Dad for their never-ending encouragement. Dreams are possible with loving parents like you.

Contents

CHAPTER 1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Thesis Overview	3
CHAPTER 2 Reed-Solomon Decoding	5
2.1 Block Codes	5
2.1.1 Forward Error Correction	7
2.1.2 Reed-Solomon Codes	8
2.1.3 Applications	11
2.2 Reed-Solomon Decoding Algorithms	13
2.2.1 Berlekamp-Massey Algorithm	15
2.2.2 Modified Euclidean Algorithm	16
2.2.3 Other Decoding Algorithms	17
2.3 Previous Reed-Solomon Decoder Implementations	19
2.4 Summary	29
CHAPTER 3 Reed-Solomon Decoder Design	31
3.1 Implementation Overview	31
3.2 Syndrome Calculation	34
3.3 Erasure Handling	35
3.4 Polynomial Generation	36
3.4.1 Erasure Locator Polynomial	36
3.4.2 Forney Syndrome Polynomial	37
3.5 Modified Euclidean Algorithm	38
3.6 Time-Domain Decoder Output	39
3.7 Frequency-Domain Decoder Output	42
3.7.1 Remaining Error Transform	42
3.7.2 Inverse Error Transform	43
3.8 Summary	44
CHAPTER 4 Reed-Solomon Hardware Implementation	47
4.1 Reed-Solomon Time-Domain Decoder	47
4.1.1 VLSI Architecture	48
4.1.2 Implementation Results	56
4.1.3 ASIC Fabrication Testing	59
4.2 Reed-Solomon Frequency-Domain Decoder	59

4.2.1 VLSI Architecture.....	60
4.2.2 Implementation Results	62
4.2.3 Testing.....	64
4.3 Comparative Analysis of Time and Frequency Domain Implementations	64
4.4 Summary.....	70
CHAPTER 5 Reed-Solomon Software Implementation	73
5.1 System Specifications	73
5.2 Time-Domain Decoding Results.....	74
5.3 Frequency-Domain Decoding Results.....	77
5.4 Comparative Analysis and Summary.....	79
CHAPTER 6 Conclusions	83
6.1 Summary.....	83
6.2 Conclusions	84
6.3 Contributions of this Thesis.....	85
6.4 Future Research	85
6.4.1 Reed-Solomon Decoding Algorithms.....	85
6.4.2 ASIC Design Methodology and Flow	86
6.4.3 Galois Field Architecture Comparisons.....	86
References	87
APPENDIX A Galois Field Primer	93
APPENDIX B Reed-Solomon MATLAB Code	97
APPENDIX C Software Profiling Results	117

List of Tables

2-1 RS Code Modifiers.....	9
2-2 Applications and their Corresponding RS Code Specifications	13
2-3 RS Decoder Implementation Literature Summary	27
3-1 Decoder Elements as a Function of Code Parameters	44
4-1 Non-Optimized Hardware Requirements for (255,239) RS Decoders	48
4-2 RS Decoder I/O Pin Description	57
4-3 Reed-Solomon TDD VLSI Results	58
4-4 Reed-Solomon FDD VLSI Results	63
4-5 Reed-Solomon Decoder Comparison.....	65
4-6 RS Decoder Thesis Summary for Table 2-3	69
5-1 TDD Software Implementation Results.....	74
5-2 Compiler Optimization Methods	75
5-3 TDD Software Static Memory Usage Results.....	77
5-4 FDD Software Implementation Results.....	78
5-5 FDD Software Static Memory Usage Results.....	79
5-6 Summary of Software Implementation of RS Decoders.....	81
A-1 GF(2⁴) Based on the Primitive Polynomial $x^4 + x^3 + 1$	95
C-1 Profiler Process Name Explanations.....	117
C-2 Sample TDD gprof Profiling Session	118
C-3 Sample FDD gprof Profiling Session	120

List of Figures

1-1	Comparing the Performance of Two Platforms and Two Domains	3
2-1	Model for an additive noisy communication channel [Wick1995]	6
2-2	Reed-Solomon Decoder Outline and Timeline	29
3-1	Structural Overview of the Time Domain Decoder	32
3-2	Structural Overview of the Frequency Domain Decoder	33
3-3	Generic Syndrome Calculation Unit	35
3-4	Generic α^k Generation Unit	36
3-5	Erasur e Locator Polynomial Generation Unit [HsTr1988].....	36
3-6	Forney Syndrome Polynomial Generation Unit [HsTr1988]	37
3-7	One Recursive MEA Cell [Shao1988].....	40
3-8	Polynomial Evaluation Circuit for $\sigma(x)$ and $\sigma'(x)$	41
3-9	Remaining Error Transform Block	42
3-10	Inverse Error Transform Block	43
4-1	Memory Block Partition.....	50
4-2	Memory Interface Circuit	51
4-3	Syndrome Calculation Unit	51
4-4	α^k Generation Block	52
4-5	Modified Erasure Locator Polynomial Generation Block	53
4-6	Modified Forney Syndrome Polynomial Generation Block.....	53
4-7	Modified Chien Search Block	54
4-8	TDD Die Photo	59
4-9	Error Transform Block.....	60
4-10	Inverse Transform Block	61
4-11	FDD Pre-Fabrication Layout.....	64

List of Symbols

c	Reed-Solomon Codeword.
d	Minimum distance of an error correction code.
e_i	i^{th} coefficient of the inverse transform of the error pattern (for the FDD).
$\hat{e}_i$	i^{th} errata pattern magnitude (for the TDD).
k	Number of information symbols in a Reed-Solomon Code.
m	Size in bits of the Galois Field.
n	Codeword length, in symbols, of a Reed-Solomon Code.
r	Received Reed-Solomon input codeword.
r_i	i^{th} received input symbol.
t	Error correcting capability of an Reed-Solomon code.
v	Number of erasures.
C_i	i^{th} output codeword symbol.
E_i	i^{th} error transform coefficient.
$S(x)$	Syndrome polynomial.
S_k	k^{th} coefficient of the syndrome polynomial.
$T(x)$	Forney syndrome polynomial
T_k	k^{th} coefficient of the Forney syndrome polynomial.
α	Primitive element in a Galois Field.
$\sigma(x)$	Errata locator polynomial.
$\omega(x)$	Errata magnitude polynomial.
$\Lambda(x)$	Erasure locator polynomial.

List of Acronyms

ADSL	Asymmetrical Digital Subscriber Line
ARQ	Automatic Repeat Request
ASIC	Application Specific Integrated Circuit
BER	Bit Error Rate
BM	Berlekamp-Massey
CCSDS	Consultative Committee for Space Data Systems
CD	Compact Disc
CDMA	Code Division Multiple Access
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
DFT	Discrete Fourier Transform
DG	Dependence Graph
DOCSIS	Data-Over-Cable Service Interface Specification
DVB	Digital Video Broadcasting
DVD	Digital Versatile Disc
DSM	Deep Submicron
FD	Frequency-Domain
FDD	Frequency-Domain Decoder
FEC	Forward Error Correction
FPGA	Field-Programmable Gate Array
GCD	Greatest Common Divisor

GF	Galois Field
GFFT	Galois Field Fourier Transform
HDD	Hard-Decision Decoder
HDTV	High Definition Television
HDL	Hardware Description Language
LFSR	Linear Feedback Shift Register
LM	Layer Metal
Mbps	Mega-bits per second
MDS	Maximum-Distance Separable
OS	Operating System
QAM	Quadrature Amplitude Modulation
ROM	Read-Only-Memory
RS	Reed-Solomon
SDD	Soft-Decision Decoder
SNR	Signal-to-Noise Ratio
SOC	System-On-a-Chip
SPDM	Single-Poly Double Metal
TD	Time-Domain
TDD	Time-Domain Decoder
VDSL	Very-High-Speed Digital Subscriber Line
VIS	Virtual Instruction Set
VLSI	Very Large Scale Integration
WB	Welsh-Berlekamp

Introduction

1.1 Motivation

Globalization, the Internet, and a technological revolution have coalesced the world; thus accentuating the importance of telecommunications in society. The need to establish and sustain reliable methods of sending information has become imperative. Noisy communication channels corrupt transmitted data streams such that a receiver may interpret the information incorrectly. This situation is mitigated through the use of powerful error correction codes.

Error correction codes dramatically improve the probability of receiving error-free information by encoding the message data with redundancy and then decoding the data at the receiver. Reed-Solomon (RS) codes and decoders are extremely powerful error correction tools that greatly enhance transmission quality. These inconspicuous techniques have proliferated in the marketplace and are used in a diverse assortment of applications ranging from the compact disc™ (CD) player to the Hubble space telescope. One of the first implementations of RS codes was in the Voyager spacecraft for deep-space communications [Wick1994].

In recent years there has been a shift from large-scale, high-speed uses to small, moderate data rate applications such as the ubiquitous cell phone. Wireless and cellular technology have progressed at a tumultuous rate, and have inundated the marketplace with a preponderance of products. Driving this euphoric development is the insatiable demand for

lighter and smaller devices with greater capabilities. There is a need therefore, to focus on area and power considerations rather than solely on speed.

Another area of interest to RS codes is the home networking concept, which is still in its infancy. Greater accessibility to the Internet and an increase in the number of electronic devices in the average home have made home networking a practical technological application. Consider the following:

- Forecasters predict that nearly 30 million North American households will own two or more computers by the end of 2002 [3Com2000].
- Home networks for communications and entertainment will find their way into over six million U.S. households by 2003 [Dabi1995].
- A precipitous fall in PC prices in the past five years has made computers available for \$999 or less [Rusn1997].

Therefore, it seems highly probable that home networking has a future as a viable industry. Currently, there are at least three competing technologies in this area. These are phone line or asymmetrical digital subscriber line (ADSL), wireless, and powerline communications. The versatility of RS decoders make them amenable to a diverse number of applications, including the aforementioned technologies. This thesis concentrates on two specific areas that have identical RS code parameters: powerline and ADSL communications. However, the results are not strictly limited to these two concepts. Other applications that employ RS codes can certainly benefit from the findings and concepts elaborated on in this thesis.

What has become increasingly clear however, is that Reed-Solomon decoders have been subject to over-design for their given applications. A Herculean emphasis has been placed on achieving aggressive decoding speeds; with less attention on optimizing area and power. Moreover, there has yet to be a presentation in the literature of a concise documented comparison of the various VLSI decoder implementations. Thus, a qualitative and quantitative comparison of RS decoders is needed. The design emphasis here will focus on reducing area and power consumption rather than on achieving high speed alone. Consequently, this thesis establishes and discusses this comparison.

1.2 Objectives

A global and integrated economy has escalated the importance of time-to-market for products in a voraciously competitive marketplace. Therefore, choosing the most efficient and cost-effective VLSI architecture becomes imperative as organizations strive to remain

competitive. This thesis closely examines and compares the various Reed-Solomon decoding architectures and implementation approaches.

An RS decoder can either be implemented in hardware as an application specific integrated circuit (ASIC), or in embedded software that resides in memory as part of a system-on-a-chip (SOC) implementation. Furthermore, another division can be made between these two implementations. An RS decoder can be designed in either the frequency-domain or the time-domain. Both approaches have advantages and inherent trade-offs. Consequently, a four quadrant qualitative comparison of RS decoders in a hardware/embedded software platform and the frequency/time domain is the primary objective of this thesis. An illustration of this concept is shown below in Fig. 1-1.

Fig. 1-1: Comparing the Performance of Two Platforms and Two Domains

In the hardware implementations, the time-domain decoder (TDD) and the frequency-domain decoder (FDD) will be compared in terms of the decoding speed, silicon area and power consumption. Conversely, the software implementation will examine executable size, memory usage and execution time. In short, the central objective of this thesis is to juxtapose the implementations of the four RS decoding approaches and then provide a definitive comparative statement.

1.3 Thesis Overview

The organization of this thesis is structured as follows. Chapter 2 begins by introducing RS codes and subsequently compares the various known decoding methodologies. Then the merits and limitations of previous RS decoder implementations are discussed. Following this is Chapter 3, which elaborates on the underlying theory, mathematics and structure used to construct RS decoders. Chapter 4 presents the time and frequency domain hardware

implementations of the RS decoder. All of the VLSI test results are documented in this chapter, and an analytical comparison between the two approaches is presented. Next is Chapter 5, where the embedded software implementation of the RS decoder is discussed. Results from both domains are analyzed and compared. Chapter 6 then presents the conclusions of this thesis, which are also accompanied by suggestions for possible future work. Following the main chapters are the cited references for the thesis. Appendix A contains a succinct primer of the area of Galois fields, upon which Reed-Solomon codes are based on. Then, Appendix B includes the MATLAB code describing the functional behaviour of both the time and frequency domain RS decoders. Finally, Appendix C contains a sample of the software profiling session for both decoders.

Reed-Solomon Decoding

This chapter begins by defining the basics of Reed-Solomon codes. Then, the usefulness of the codes is corroborated by illustrating the vast array of applications that use RS codes. Actual decoding methods and previous decoder implementations are discussed. The history of RS decoders dates back several decades, so there has been a great deal of research in this area. However, it will be shown that although the literature is extensive it does not provide any concrete definitive statements on RS decoder implementation comparisons. This thesis attempts to clarify this ambiguity.

2.1 Block Codes

Before the actual Reed-Solomon codes are described in full detail, it is necessary to explain the basic communication concepts pertaining to RS codes. The intention of any digital communication system is to deliver a message from the transmitter to the receiver. Reed-Solomon codes are a set of extremely powerful error correcting block codes that serve to improve the quality of this transmission. A block error control code starts with a stream of binary message data and subsequently breaks it up into distinct blocks of fixed length. Each message block u , consists of k information bits, which correspond to a total of 2^k distinct messages. At this stage, the codes introduce a certain amount of redundancy into the message by using an encoder mapping. The blocks are mapped into a binary n -tuple c , referred to as the codeword, with $n > k$. Since there are 2^k messages, there must be a set of 2^k codewords, which is called the block code. Each of the encoded blocks are denoted as symbols. The next property of interest to block codes, and hence RS codes, is linearity.

A block code of length n with 2^k codewords is called a linear (n,k) code if its 2^k codewords form a k -dimensional vector subspace of all the n -tuples over the Galois Field $GF(2)$. The reader interested in a brief description of Galois fields (GF) is referred to the GF Primer in Appendix A. It follows then, that the dimension of a linear code is the dimension of the corresponding vector space. A more detailed description of linear block codes and their properties can be found in [Lin1983].

The purpose of block codes is to provide an effective means of correcting received data that differs from the original transmitted data. This is important because data on a transmission channel is continually corrupted. Codeword corruption occurs when additive noise is introduced into the communications channel. The model for this process is illustrated below in Fig. 2-1.

Fig. 2-1: Communication Channel Model with Additive Noise [Wick1995]

At the receiver, a decoder determines whether the received word is a codeword or not. If the decoder does not recognize the codeword then it assumes that the channel has caused one or more symbol errors in the transmitted codeword. This step is called error detection. Depending on the specifics of the decoder, the next step performs some operation to deal with the corrupted codeword.

The decoder must successfully detect that an error has been received. After detection, there are three fundamental techniques that the receiver can use to handle transmission errors and they are listed below.

- Request the retransmission of the codeword.
- Denote the received data as incorrect and pass it to the next stage.
- Attempt to correct the errors in the received data.

The first method is referred to as automatic repeat request (ARQ) protocols. They place a premium on receiving the correct data regardless of the circumstances. Hence, ARQ is used

where an extremely low bit-error rate (BER) is demanded. It can be extremely slow if there are numerous errors at the receiver. The next option is used in situations that require a high throughput rate. When the decoder receives and detects the data error, it will flag the data as being incorrect. This data is passed to the next stage, but it is marked as being an error. A high BER must be tolerated in this case since there is no attempt at correcting the error. The last method is known as forward error correction (FEC). FEC systems determine the validity of the received data and can correct it based on the arithmetic or algebraic structure of the code. RS codes are a type of FEC method.

2.1.1 Forward Error Correction

There are several characteristics of FEC codes that need to be discussed before the concept of RS decoders can be described. First, if the decoder accepts the received word as being valid, but it is a codeword different from the one that was initially transmitted then this is known as an undetectable error pattern. This occurs when the channel inundates the data with an inordinate amount of errors; thereby changing the original codeword into a completely different but valid codeword. Recognizing the word as being a valid codeword, the decoder assumes it to be correct and does not attempt to correct it. If there are a total of M codewords then there will be $(M - 1)$ such undetectable error patterns.

Second, one of the inherent limitations of FEC is that it is possible for the decoder to commit a decoding error. This occurs when the decoder recognizes that the received word is in error, but it incorrectly selects a codeword other than the one which was transmitted. Unfortunately, if this occurs it is impossible for the decoder to indicate that it has failed to correct the word. This phenomenon typically occurs when the number of errors in the codeword exceeds the error correcting capability of the decoder (i.e., it exceeds the distance properties of the code).

To characterize the error correction capabilities of FEC block codes, a few concepts must first be defined. The number of nonzero positions in a codeword or error pattern is called its *weight*. In addition, the *Hamming distance* between two symbols of a word is the number of coordinates or bit positions in which the two blocks differ. The shortest Hamming distance between all distinct pairs of codewords in a block code C , is the minimum distance of the code. This is significant because a code with a minimum distance $d_{\min}$ can detect all error patterns of weight less than or equal to $(d_{\min} - 1)$. Although it is important to detect errors, the main function of the FEC decoder is to correct received error patterns. Therefore, a code with a minimum distance $d_{\min}$ can correct all error patterns of weight less than or equal to $\lfloor (d_{\min} - 1)/2 \rfloor$. This equation represents the upper bound on the weight for which one can

correct all error patterns. It is possible, but highly improbable, that one more error than given by the upper error bound can be corrected in certain received blocks.

Finally, FEC decoders are susceptible to committing a decoder failure, which are completely detectable unlike decoder errors. Let a t -error correcting decoder receive a word $\mathbf{r}$. The decoder then tries to select a codeword $\mathbf{c}$ that minimizes the distance between the two. A decoder failure occurs when no such $\mathbf{c}$ exists that satisfies the following:

$$\text{minimum distance between } (\mathbf{c}, \mathbf{r}) \leq t .$$

Nevertheless, detecting the failure for RS codes is not trivial especially in hardware implementations of the decoder. It involves testing a polynomial generated by the decoding algorithm to ensure that there are no repeated roots.

Using the preceding discussion as a basis, it now becomes possible to fully elaborate on Reed-Solomon codes, which is one of the fundamental topics of this thesis.

2.1.2 Reed-Solomon Codes

Reed-Solomon codes are special types of FEC block codes. They are based on the same fundamentals as other block codes. However, RS codes are an extremely powerful set of error correcting codes that are based on symbols rather than on bits. The general structure of an RS code can be described as follows. Each code is composed of n symbols with a certain number of message symbols k and redundant symbols $(n - k)$. The code is referred to as an (n, k) Reed-Solomon code of length n , and dimension k over a Galois field $\text{GF}(q)$, where q is the power of a prime number $q = p^m$. It has a minimum distance of d , where $d = n - k + 1$ and $n = q - 1$, and an error correction capability of $t = (n - k)/2$ errors.

Codewords are generated by a set of polynomials with degree $k - 1$ and coefficients from $\text{GF}(q)$. All of the RS codes that are relevant to digital communications are based on the binary extension field $\text{GF}(2^m)$, where each symbol is an m -bit word. Conveniently, if $m = 8$ then each symbol is an 8-bit word or byte, which is suitable for applications of digital communications. RS codes are also defined to be maximum-distance separable (MDS) codes, which means they satisfy with equality an axiom known as the Singleton bound $d_{\min} \leq n - k + 1$. Thus, their minimum distance is always one more than the number of redundant symbols [Cost1998].

RS codes are suited for communication channels that are susceptible to burst errors. To elaborate, the RS decoder operates on symbols rather than on individual bits of the data stream. Therefore, the decoder corrects the entire received m -bit symbol regardless if there

is one bit error or m -bit errors caused by a burst noise error event. Conversely, if there are a smattering of bit errors throughout the code word, then the decoder's resources are not being put to optimal usage. In perspective, the bit-error correcting capability can thus range from m bits (bit errors are dispersed) to m^2 bits (the bit errors are contiguous).

Furthermore, the use of *erasures* enhances the error correcting capability of an RS code. Erasures provide the decoder with more information about the errors in the codeword. An erasure location is a symbol location in the codeword, which the decoder recognizes as being incorrect. However, the decoder does not know which bits are in error or what the correct symbol is. This differs from an error in that the decoder does not know where or what magnitude the error is when it receives a corrupted codeword. Since erasures provide additional information, they increase the error correcting capability of a code. If there are v erased coordinates then it is possible to correct $t = (\lfloor d_{\min} - v - 1 \rfloor) / 2$ errors in the unerased coordinates of the received word. Hence, the decoder can correct e errors and v erasures as long as $(2e + v) < d'_{\min}$. A decoder is thus able to correct twice as many erasures as errors because of the additional location information.

The applicability of RS codes is greatly augmented through the use of linear code modifiers. There are several implementations where external constraints determine the allowed length of the error control code. If the original code construction is not suitable for a particular application, then the code may be altered without changing its basic functionality. These slight codeword deviations are puncturing, extending, shortening, lengthening, expurgating and augmenting. The effects on the original (n, k) code are summarized in Table 2-1.

Table 2-1: RS Code Modifiers

Modification	New Form of (n, k) Code	Explanation
Puncturing	$(n - x, k)$	x parity coordinate(s) deleted
Extending	$(n + x, k)$	Add x redundant coordinate(s)
Shortening	$(n - x, k - x)$	x message coordinate(s) deleted from the encoding process
Lengthening	$(n + x, k + x)$	Add x message coordinate(s)
Expurgating	$(n, k - x)$	x code word(s) deleted
Augmenting	$(n, k + x)$	x new code word(s) added

To illustrate, digital video broadcasting (DVB) is based on a (255,239) RS code with a primitive polynomial $m(x) = x^8 + x^4 + x^3 + x^2 + 1$ [Sohi2000]. However, in order to form the actual code that is used by DVB, the original RS code is shortened by $x = 51$ symbols resulting in a (204,188) code. The use of these modifiers is quite pervasive and they are found in a myriad of applications. This will become apparent in Section 2.1.3, which discusses the areas that use RS codes.

Time-Domain and Frequency-Domain Interpretations in GFs

As previously stated, Reed-Solomon codes can either be decoded using a time-domain (TD) or a frequency-domain (FD) approach. Therefore a succinct discussion now follows, which relates the FD to the TD in the context of GF arithmetic. This serves to facilitate the understanding of the two RS decoding approaches.

If $\mathbf{c}$ is a vector over $\text{GF}(q)$ then it is related to its transform $\mathbf{C}$ by the following equations:

$$C_j = \sum_{i=0}^{n-1} \alpha^{ij} c_i, j = 0, 1, \dots, n-1 \quad (2-1)$$

$$c_i = \frac{1}{n} \sum_{j=0}^{n-1} \alpha^{-ij} V_j, i = 0, 1, \dots, n-1 \quad (2-2)$$

where the character α is a primitive element in $\text{GF}(2^m)$ (see Appendix A for a detailed description). The preceding transform is a generalization of the discrete Fourier transform (DFT) to finite fields. Hence, the same properties of the DFT are true for the Galois field Fourier transform (GFFT) in Equation 2-1. A few example properties are linearity, shifting and convolution.

Codewords are generally transmitted as a sequence of symbols indexed in time. Therefore, the GFFT is a convenient method for translating TD characteristics into FD characteristics. The FD equivalent or spectrum of the TD polynomial $c(x) = c_0 + c_1x + \dots + c_{n-1}x^{n-1}$ is the GFFT of the vector $\mathbf{c} = (c_0, c_1, \dots, c_{n-1})$. From this relation, two more theorems can be defined. First, α^j is considered a zero of the polynomial $c(x)$ if and only if the j^{th} frequency component of the spectrum of $c(x)$ equals zero [Wick1995]. Conversely, α^{-j} is a zero of the polynomial $C(x)$ if and only if the i^{th} time component c_i of the inverse transform $\mathbf{c}$ of $\mathbf{C}$ equals zero. These principles can be applied to RS codes in order to achieve FD decoding. A full explanation of the implementation of the FD and TD decoders is presented in Chapter 3.

2.1.3 Applications

RS decoders can be found in a wide array of applications. Although the majority of RS decoder applications have been in the last decade or so, implementations which used the code became apparent shortly after their discovery in the early 1960's.

One of the first areas to use RS codes as a means of error correction was deep-space communication in the early 1970s [Wick1994]. The Mariner Mars orbiter mission in 1971 used a (6,4) RS code over $GF(2^6)$. However, it could be argued that this was not really a true implementation of the code, but rather only a very special case. The code had a redundancy of two so it could only correct one error. Hence, it was more aptly called a generalized Hamming code. It was not until 1977 that the Voyager mission made full use of the powerful error correcting abilities of RS codes. The spacecraft was fitted with a concatenated (255,233) RS code with a primitive polynomial $m(x) = x^8 + x^4 + x^3 + x^2 + 1$. In a concatenated system, an inner and outer decoder are used in conjunction with one another. The inner decoder is set adjacent to the noisy communication channel and it occasionally makes errors that tend to be bursty and hard-quantized. These are exactly the types of errors that an outer RS decoder is proficient at correcting.

The most recent standard [Wick1994] on RS codes for deep-space applications was set out in 1984 by the Consultative Committee for Space Data Systems (CCSDS). It has been adopted for use in several planetary and deep-space missions. The standard is a (255,223) RS code over $GF(2^8)$ and is represented by the polynomial $m(x) = x^8 + x^7 + x^2 + x + 1$. Therefore, the search for more efficient RS decoding implementations is reinforced by the continued exploration of the depths of the universe.

Probably the most widely known application of RS codes is the CD player. What is interesting here is that it makes use of RS code modifiers. It uses two shortened RS codes with 8-bit symbols taken from $GF(2^8)$. The two codes are cross-interleaved together with the first code being a (32,28) code and the second a (28,24) code.

In addition, the digital versatile disc (DVD) standard is based on a (255,239) RS code with symbols from $GF(2^8)$ and primitive polynomial $m(x) = x^8 + x^4 + x^3 + x^2 + 1$. It makes use of the code modifiers in an innovative row-column configuration. The column RS-code is shortened by 47 symbols to form a (208,192) code. Next, the row RS-code is shortened by 67 symbols and punctured by 6 symbols resulting in a (182,172) code.

DVB has three standards that use identical RS code parameters, which were first described in Section 2.1.2. These are satellite (DVB-S), cable (DVB-C) and terrestrial (DVB-T) transmission. The most widely used of the three protocols is DVB-S [Sohi2000].

The two applications that are targeted in this thesis are G.lite (ADSL) and powerline communications. Both standards use a (255,239) RS code, have an error correcting capability of $t = 8$, and are based on the primitive polynomial $m(x) = x^8 + x^4 + x^3 + x^2 + 1$. A maximum decoding speed of 20 MBps is required for these applications. This speed comfortably accommodates G.lite (1 MBps) [ITU1999], very-high-speed digital subscriber line (VDSL) (13 MBps) [Coop2000], and powerline communications (13.75 MBps). The reasons for choosing these standards are three-fold. First, these code parameters are one of the most widely used RS codes to date. If the research focus of this thesis is targeted to these RS codes, then the results will be applicable to a greater number of applications. Second, the RS code parameters are amenable to VLSI design. The codeword symbols are 8-bits or one byte and the primitive polynomial facilitates the design of GF arithmetic circuits. Finally, these standards are both applicable to the concept of home networking, which is rapidly increasing in popularity.

However, the Data-Over-Cable Service Interface Specification (DOCSIS) standard could be a possible target of this thesis as well. DOCSIS, popularly known as a cable modem, has a variable error correction capability in the standard, which ranges from $t = 1$ to $t = 10$ errors. If $t = 8$, then the DOCSIS RS specifications would be identical to those of powerline communications and G.lite. Other applications where RS codes are used include computer memory, wideband code division multiple access (WCDMA), and magnetic disk storage. Table 2-2 provides a summary of some of the more relevant applications that use RS codes. The numerous applications serve to reinforce that there is a need for conducting research into the area of RS codes and RS decoders. Nevertheless, the difficulty in choosing the appropriate RS code lies with the development of an efficient, but yet simple decoding algorithm.

Table 2-2: Applications and their Corresponding RS Code Specifications

Application	Primitive Polynomial	RS Code Specification (<i>n</i>, <i>k</i>)	Error Correcting Ability <i>t</i>
CD [Wick1994] [Mits1999]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	Shortened & Cross-Interleaved Dual Code (32,28) & (28,24)	4
DVD [Sohi2000]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	Column – Shortened (208,192) Row – Shortened & Punctured (182,172)	8
DVB [Sohi2000]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	Shortened (204,188)	8
DOCSIS [RFI1999]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	Adjustable ^a <i>k</i> (255,235) to (255,253)	1 to 10
CCSDS [Wick1994]	$m(x) = x^8 + x^7 + x^2 + x + 1$	(255,223)	16
G.lite (ADSL) [ITU1999]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	(255,239)	8
Powerline Communications	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	(255,239)	8
VDSL [ETSI2001]	$m(x) = x^8 + x^4 + x^3 + x^2 + 1$	Varies (144,128) or (240,224). Other (<i>n</i> , <i>k</i>) are possible but $n \leq 255$.	0 to 8

a. Ranges depend on the desired size of the error correcting capability of the codes *t*, which can range from $t = 1$ to $t = 10$.

2.2 Reed-Solomon Decoding Algorithms

The most challenging aspect of RS codes is finding efficient techniques for decoding the received symbols. There are several problematic issues that arise with decoding RS codes. For instance, how to minimize the occurrence of undetectable errors and how to reduce the number of decoder failures, which were both explained in Section 2.1.

Research in this area has led to the development of a few relatively reliable decoding algorithms. This section succinctly discusses these algorithms. Hence, the goal here is to

provide the reader with a basic understanding of the mathematical steps each algorithm requires, rather than a theoretical in-depth explanation. The reader who is interested in the latter is directed to [Wick1994] and [Wick1995].

Before the ensuing discussion, some concepts are required to be defined for some of the algorithms. First, let the error vector be $e = (e_0, e_1, \dots, e_{n-1})$, which has a polynomial representation

$$e(x) = e_0 + e_1x + \dots + e_{n-1}x^{n-1}. \quad (2-3)$$

If the original encoded codeword is c and it has a corresponding polynomial $c(x)$, then the received polynomial at the decoder is

$$v(x) = c(x) + e(x) = v_0 + v_1x + \dots + v_{n-1}x^{n-1}. \quad (2-4)$$

This polynomial can be evaluated at $\alpha, \alpha^2, \dots, \alpha^{2t}$, which are the roots of the generator polynomial $g(x)$. Therefore, a set of $2t$ equations must be solved in order to obtain what are known as the syndromes $S_k, k = 1, 2, \dots, 2t$ where

$$S_k \equiv v(\alpha^k) = \sum_{i=0}^{n-1} v_i \alpha^{ik}. \quad (2-5)$$

The syndrome polynomial can then be defined as

$$S(x) = \sum_{i=1}^{2t} S_i x^i, \quad (2-6)$$

where S_i are the syndromes from Equation 2-5. Furthermore, let there be v errors, at v unknown error location numbers $X_1, X_2, \dots, X_v$ with v unknown error values $Y_1, Y_2, \dots, Y_v$. An error evaluator polynomial is related to the error locations and error values as follows:

$$\Omega(X_l^{-1}) = Y_l \prod_{i \neq l} (1 - X_i X_l^{-1}). \quad (2-7)$$

Now define the error locator polynomial $\Lambda(x)$, to be

$$\Lambda(x) = 1 + \Lambda_1 x + \dots + \Lambda_v x^v. \quad (2-8)$$

This polynomial has as its roots the inverse error location numbers X_l^{-1} , for $l = 1, 2, \dots, v$. So

$$\Lambda(x) = \prod_{l=1}^v (1 - x X_l) \quad X_l = \alpha^{i_l}. \quad (2-9)$$

All of the preceding polynomials can be related by means of a key equation defined as

$$\Omega(x) = [1 + S(x)]\Lambda(x) \bmod x^{2t+1} \quad (2-10)$$

Therefore, it is the function of the decoding algorithm to solve the key equation for $\Lambda(x)$. In FD decoding, the same key equation is used, but there is a different interpretation of the results. Hence, its subsequent treatment is quite different. This will be fully elaborated on in Chapter 3.

2.2.1 Berlekamp-Massey Algorithm

The Berlekamp-Massey (BM) algorithm is an efficient method used for correcting a large number of errors. Its complexity increases linearly, which allows for the construction of efficient decoders.

A linear feedback shift register (LFSR) can be used to find $\Lambda(x)$, where the coefficients of $\Lambda(x)$ are taken to be the taps of the LFSR. The algorithm uses five parameters: a length k LFSR $\Lambda^{(k)}(x)$, the correction polynomial $T(x)$, the discrepancy $\Delta^{(k)}$, the length L of the shift register, and an indexing variable k . The decoding process [Wick1995] is as follows:

- (i) Compute the syndromes $S_1, S_2, \dots, S_{2t}$ for the received word.
- (ii) BM algorithm initialization is: $k = 0$, $\Lambda^{(0)}(x) = 1$, $L = 0$, and $T(x) = x$.
- (iii) Set $k = k + 1$. Then calculate the discrepancy $\Delta^{(k)}$ by subtracting the k^{th} output of the LFSR defined by $\Lambda^{(k-1)}(x)$ from the k^{th} syndrome as follows:

$$\Delta^{(k)} = S_k - \sum_{i=1}^L \Lambda_i^{(k-1)} S_{k-i}$$

- (iv) If $\Delta^{(k)} = 0$, then go to (viii).
- (v) Else set $\Lambda^{(k)}(x) = \Lambda^{(k-1)}(x) - \Delta^{(k)}T(x)$, then go to (vi).
- (vi) If $2L \geq k$, then go to (viii).
- (vii) Else set $L = k - L$ and $T(x) = \Lambda^{(k-1)}(x) / \Delta^{(k)}$, then go to (viii).
- (viii) Set $T(x) = x * T(x)$.
- (ix) If $k > 2t$, then go to (iii).
- (x) Else if roots are NOT distinct then go to (xii).

- (xi) Else (as part **if** statement in step (ix)) roots are distinct so determine the roots of $\Lambda(x) = \Lambda^{(2t)}(x)$ by finding the error magnitudes which are defined as

$$e_{i_k} = \frac{-X_k \Omega(X_k^{-1})}{\Lambda'(X_k^{-1})}.$$

Correct the corresponding locations in the received word by adding the corresponding error magnitude and STOP.

- (xii) Declare a decoder failure and STOP.

This concludes the explanation of the BM decoding algorithm.

2.2.2 Modified Euclidean Algorithm

The Modified Euclidean Algorithm (MEA) is based on the Euclidean algorithm, which iteratively finds the greatest common divisor (GCD) of two elements in a Euclidean domain. Mathematically, the original Euclidean algorithm proceeds as follows:

- (i) Define two Euclidean elements (a, b) and the initial conditions: $r_{-1} = a$, $r_0 = b$, $s_{-1} = 1$, $s_0 = 0$, $t_{-1} = 0$ and $t_0 = 1$.
- (ii) Recursively compute s_i , t_i , and r_i (while $r_i < r_{i-1}$) as follows:

$$\begin{aligned} r_i &= r_{i-2} - q_i r_{i-1}, \\ s_i &= s_{i-2} - q_i s_{i-1}, \\ t_i &= t_{i-2} - q_i t_{i-1}. \end{aligned}$$
- (iii) If the remainder $r_n = 0$, then r_{n-1} is the GCD of a and b . STOP.
- (iv) Else repeat (ii).

At any given time in the algorithm, the recursion relations in (ii) guarantee that the relation $s_i a + t_i b = r_i$ is true. This relation corresponds to the key equation stated in Equation 2-10.

However, the original Euclidean algorithm is not suitable for RS VLSI design because each iteration requires a division to be computed [Shao1985]. Finite-field inverses are used in the division calculation. Unfortunately, they are area intensive operations. The original algorithm can be modified though to eliminate the computation of inverses during the iterations, which in turn reduces the hardware complexity. This modification makes the MEA comparable to the BM algorithm in terms of VLSI implementation feasibility [Wick1995]. The benefit of the MEA though is its low complexity and simplicity with which it is understood and applied. Furthermore because of its structure, the MEA finds both the error locator and the error evaluator polynomials.

Consequently for practical reasons, the MEA rather than the original algorithm is used for decoding the key equation. All implementations that are based on Euclid's GCD algorithm use the modified version. Thus, a thorough presentation of the MEA will be provided in Section 3.5 of Chapter 3, and the original version will not be discussed further in this thesis. The reader who would like a more detailed explanation of the Euclidean algorithm in terms of RS decoding is referred to [Wick1995].

The MEA was chosen to be the decoding algorithm for this thesis. Hence, a comprehensive discussion of the MEA is deferred to the next chapter in Section 3.5. A greater emphasis will be placed on the MEA as opposed to the BM algorithm. The reasons for choosing the MEA are as follows. First, it can be argued that both algorithms are comparable in terms of efficiency. However, the MEA is conceptually easier to understand and structurally less complex. This translates into less design time, which is crucial in meeting today's intense time-to-market demands. Second, there were more architectural enhancements suggested for the MEA than for the BM algorithm such as those found in [Shao1985], [HsTr1988], [Kwon1997] and [Oh1997]. By examining certain key designs, it was ascertained that the MEA could be modified so that it was applicable to the targets of this thesis. Various blocks of the MEA were found to be structured in such a way as to minimize area and reduce power consumption. In addition, finding an implementation that was optimal in terms of silicon area was integral to this thesis. GF multiplication and division are two arithmetic operations that are area intensive. Therefore, it became imperative to use the MEA implementation, which reduced the number of GF operations used in the decoder's iterative procedure.

Finally, after reviewing previous implementations, it became apparent that the MEA was used more often [Shao1985], [ShTr1985], [HsTr1988], [Shao1988], [Truo1988], [Whit1991], [Chen1995], [Jeon1995], [Kwon1997], [Oh1997], [Jenn1998], [Wilh1999] and [Huan1999] than the BM algorithm [Reed1991], [Hsu1996], [Ragh1997], [Fitz1998], [Jeng1999] and [Chan2001]. Research that uses the MEA and offers a definitive qualitative and quantitative comparison would thus be of greater value to the field of RS decoders. In short, it is believed that the MEA provides the most optimal decoder architecture that focuses on the goals of this thesis.

2.2.3 Other Decoding Algorithms

The MEA and the BM algorithm are the two decoding approaches found most frequently in applications using RS decoders. They are straightforward to understand and their structures are suitable for VLSI design. However, research is being conducted into more

abstract, but efficient methods of RS decoding. These are the areas of soft-decision decoding and remainder-based decoding. A brief discussion of these methods is included for thoroughness.

Remainder-based decoding does not require the computation of the syndromes from Equation 2-5. Instead the algorithm uses a remainder polynomial $r(x)$, which is obtained from the division of the received polynomial $v(x)$ by the generator polynomial $g(x)$, as

$$r(x) = r_0 + r_1x + \dots + r_{n-k-1}x^{n-k-1}$$

$$r(x) = v(x) \bmod g(x).$$

The algorithm was developed by Welch and Berlekamp and is aptly named the Welsh-Berlekamp (WB) algorithm. In terms of hardware realization, there is no circuitry required for syndrome calculation. The WB algorithm involves the use of four polynomials instead of the usual two found in either the MEA or the BM algorithm [Wick1994]. Therefore, additional registers, multipliers and adders are needed for the decoder block. Nevertheless, the algorithm can be implemented using systolic arrays and is similar in complexity to both the BM algorithm and the MEA [Dabi1995]. This thesis did not consider implementing a WB algorithm because of the lack of existing practical applications that use this approach. The reader is referred to [Berl1986], [Wick1994] and [Dabi1995] for a more engaged description of the algorithm and the associated architecture.

Each of the algorithms discussed thus far can be classified as being a hard-decision decoder (HDD). The detection circuit requires that its inputs be from the same symbol alphabet as the channel input [Wick1994]. However in certain instances, the received signal may not offer a clear choice as to which of the possible symbols has been transmitted. The receiver then must guess as to which of the symbols the received value most closely resembles. On the other hand, a soft-decision decoder (SDD) accepts an actual vector of noisy channel output samples and estimates the vector of channel input symbols that was transmitted. Unlike the HDD, the SDD does not force a decision which is likely to be incorrect. Therefore, SDDs can provide a coding gain of about 2dB more than that provided by HDDs.

The decoder can be supplied with soft-decision data quite readily, but the difficulty is obtaining an optimal SDD that is not prohibitively complex. Current algorithms are non-algebraic and run in time that scales exponentially with the length of the code. However, in [Koet2000], a new algebraic soft-decision decoding technique is developed. Currently, the research focus in this area is in devising an efficient decoding algorithm that can be practically implemented in VLSI applications.

2.3 Previous Reed-Solomon Decoder Implementations

The foundation for RS decoder VLSI research can be found in [Shao1985] where for the first time, an efficient hardware implementation of an RS FDD was discussed. Before this approach, the decoder design was extremely complicated and it occupied an prohibitively large silicon area. To its credit, [Shao1985] developed a regular pipelined architecture that was suitable for VLSI design. Nevertheless, there was no mention of fabricated silicon. Thus, no predefined benchmark was provided for other researchers in this area to compare their work against. In addition, the technique did not incorporate erasure correction into the decoder. The ability of an RS decoder to support erasures dramatically improves the correction capability of the decoder.

Subsequent to [Shao1985], the same authors reported an actual silicon implementation in [ShTr1985]. Several modifications were made such as including erasures, and performing time-domain based decoding. In addition, the TDD was stated as being superior because it was simpler, more regular, had less area and operated equally as fast as the FDD in [Shao1985]. However, there were no specifics given about the actual area which was saved or the speed of operation of either decoder. The statement about the TDD having a more regular structure was incorrect as well. One of the apparent advantages, if any, of the FDD over the TDD is that its structure is simple, repetitive, and therefore less complex than the TDD.

Following this groundbreaking work, a comparison between the two decoding methods is made in [HsTr1988]. Hardware architectures are illustrated for both methods and the design of each approach is clearly shown. All stages of the decoder are compared and contrasted using both methodologies. The FDD is lauded as being superior to the TDD because the FDD is stated as being considerably less complex. Moreover, the only apparent trade-off is that the FDD occupies a larger silicon area, but for most of the codes in use today the extra area is insignificant. However, stating that an FDD is less complex than a TDD gives no substantiated information on the performance of the decoder. No numbers are provided for how much larger an area the FDD occupies or how fast each of the decoders operate. In fact, there is no mention of any results from silicon. So although [HsTr1988] established some preliminary distinctions between the two decoding approaches, no definitive results were published.

Finally [Shao1988] elucidated the comparison of chip area that each decoder required, through the use of a comparison table. Various decoder stages from each implementation were compared based on the area that a polynomial multiplication circuit used. It showed

for the first time that the FDD occupied prohibitively more area than its time domain counterpart. In fact, the TDD was extolled for having lower area, lower power consumption, higher reliability and higher yield [Shao1988], all of which lead to lower costs. Ambiguity nevertheless obfuscates the definitiveness of the results. For example, multiplier area was stated as being the unit of comparison for area. However, there was no mention of the exact area or speed of the multiplier. This approach failed to show a meaningful analogy because it lacked any tangible results from a fabricated chip. In short, an ameliorated view of the TDD and FDD was brought to light, but [Shao1988] failed to provide unambiguous evidence to support their claims. The TDD implementation was reiterated in [Truo1988] with no new information.

Nevertheless, the publications previously discussed were the inflection point for an exponential growth in RS decoder research. There were extensive research contributions to this area over the past decade and they are referenced in the ensuing discussion. However, all of them have failed to produce a definitive statement that is quantitatively supported.

[Reed1991] designed a VLSI RS decoder using an inverse-free Berlekamp-Massey algorithm. The stated benefits of this algorithm were that it was less complex than the Euclidean algorithm. However, again there was no accompanying VLSI support for this statement. Although the algorithm managed to eliminate an area intensive inverse calculation, it used a significantly large number of multipliers. No hardware results were provided.

VLSI results were reported in [Whit1991] for an RS time-domain codec. This implementation used the MEA and was based on a (167,147) shortened RS code. The chip contained 200 000 transistors, had an area of $68.88mm^2$, operated at a data rate of 80 Mbps, consumed 500mW of power and was fabricated in a $1.6\mu m$ CMOS process. Erasures were supported and it could correct up to and including $t = 10$ errors. This chip was strictly targeted to advanced television and several limitations existed. First, the design was done using a full-custom process. Thus, it had a high degree of complexity and the design time was significant. Today's intense time-to-market demands however require designs to be expeditious. Second, large memory buffers were used extensively, which contributed to the high silicon area. Next, the reasoning as to why the TDD approach was chosen was that it could be implemented as a small array of identical cells. This statement could be said about the FDD as well. Finally, this design could not be used as a pure decoder benchmark since it incorporated the encoder design as well.

[Choo1992] discussed a VLSI architecture and offered a comparison between the TDD and the FDD. No hardware was produced and the designs did not include any erasure handling

capability. The discussion is strictly based on the algebra behind the decoding and can use either the MEA or the BM algorithm for decoding. In addition, the proposed algorithm used more multipliers than several known implementations. There were several key comparisons stated in this paper, and they are discussed shortly. Keep in mind however, that these comparisons were assumptions based on the theory of the algorithm. No tangible results were given because there was no accompanying silicon. First, the TDD algorithm was said to require twice as many multiplications as the FDD algorithm. Thus, the TDD would not be suitable for high-speed applications (> 200 Mbps), which require numerous multipliers. However, for medium speed applications, the TDD would be superior because the architecture would then be more dependent upon other criteria. These include regularity, control complexity and flexibility. In addition, the time domain algorithm was lauded because it could be implemented as a regular array of identical cells. However, this was known to be true for the FDD as well. Finally, the TDD is deemed to be better suited for decoding truncated RS codes. In short, numerous comparative statements were made about the two approaches to RS decoding, but they were all based exclusively on the mathematics of the algorithm. A hardware implementation would have provided a truly conclusive comparison.

[Chi1993] designed an RS decoder which eliminated the Chien search block in order to achieve a higher speed. The Chien search block was replaced by a redesigned block of hardware that was faster. However, this new block occupied a larger silicon area as compared to the Chien search block. In addition, the decoding algorithm was modified to accommodate this new approach. Unfortunately, it was slower than both the MEA and the BM algorithm. Overall, the decoder was stated as being faster, but this was at the expense of an increase in area. No hardware was actually produced in this implementation either.

The design in [Chen1995] used a modified Euclidean algorithm (MEA) for the decoder. This design was developed and tested with Verilog, which is a hardware description language (HDL). However, no specific numbers from either software or hardware were mentioned. In addition, the circuit used an area intensive read-only-memory (ROM) to store all the required inverses.

[Jeon1995] presented what is probably the best comparison to date of the TDD and FDD. However, the goal here was to accomplish this juxtaposition without fabricating any hardware. It used a variation of Euclid's GCD algorithm for solving the key equation. Comparisons were made using a dependency graph (DG) and an entirely mathematical approach. These comparisons were based on derived dependency structures in terms of total

computation (DG size) and critical path delay. It presented a good overview on RS decoders, which could be used as a preliminary estimate before doing the actual hardware design. It showed that the TDD is superior to the FDD in terms of delay and area. This was done strictly in C-programming software and no HDLs were used. However, there were some conspicuous limitations. Most importantly, the approach did not provide a method for reporting power dissipation for either decoder. This is often the single most significant factor in determining the feasibility and efficiency of a VLSI design. In addition, adding to the complexity and ambiguity of the estimates were the numerous factors that had to be considered for the study. These included choosing appropriate mathematical strategies, types of multipliers and dividers, and finite-field polynomials. Not all of these factors were fully specified in the discussion.

An algebraic comparison of a Euclidean based algorithm and a Berlekamp-Massey algorithm was presented in [Saka1995]. Although no silicon was produced in this publication, a prominent conclusion was stated. The results provided a proof of the equivalence between the Berlekamp-Massey algorithm and the Euclidean algorithm in the sense that both methods yielded distinct but similar parallel architectures with the same optimal complexity.

In [Hsu1996], an RS decoder based on the Berlekamp-Massey algorithm was designed. No hardware was fabricated but it was assumed, based on several coarse estimations, that the design would have 406 000 transistors for a $t = 8$ error correcting code. The area was significant because of several multipliers that were used in repeated cells in the decoding algorithm. No other tangible specifications were mentioned.

[Ragh1997] developed a low power RS decoder design that was targeted towards portable wireless receivers. However, it had double the area of traditional RS decoder implementations and achieved a speed that was clearly in excess of what the target applications required. The chip was still in the stages of testing and was not fabricated at the time of publication. It used the Berlekamp-Massey decoding algorithm and it was synthesized using a $2\mu\text{m}$ library. Estimated numbers from the design showed a bit rate of 343Mbps with 13945 gates. However, in order to achieve the lower power of the design the preceding bit rate had to be reduced.

[Kwon1997] designed a combined RS decoder/encoder for digital VCRs using a modified Euclidean algorithm. This approach combined the encoder with the decoder in order to save hardware. Therefore, [Kwon1997] cannot be used as a benchmark for comparing other decoders. No silicon was fabricated and the results were based on approximations deduced

solely from the algorithm. This proposed method included erasure handling, used 35 000 gates and targeted a speed of 18MHz. An important feature was its superior implementation of the Chien search for saving area at the expense of speed. The design combined two large computational blocks that are usually implemented separately. Although this lowered the speed of the search block, it optimized the area occupied by this traditionally large block. The reduction in speed should not be an issue for meeting the design parameters of the target applications in this thesis. Thus, this approach was used for the VLSI implementation of the Chien search block in the TDD designed in this thesis.

[Oh1997] designed a similar RS decoder structure to that in [Truo1988]. The design implemented a (207,187) RS code on an field-programmable gate array (FPGA) that had an equivalent gate count of 50 000 and a decoding speed of 10 Mbps. Target applications for this decoder were DVDs and high definition television (HDTV). A comparison between the proposed architecture and the one developed in [Truo1988] was made. There were some interesting improvements that were applicable to this thesis. First, the overall decoder complexity was reduced by about 30% to that found in [Truo1988]. Complexity in this case, referred to the implementation area. The reduction was achieved by changing the parallel structure of the polynomial expansion block to a serial architecture to save area at the expense of speed. However, the Chien search and the MEA that were used had a greater degree of complexity than the previous approach. Nevertheless, the idea developed in the polynomial expansion block can be incorporated into this thesis. The decoding speed was too slow for the targeted applications of this thesis though. It was also difficult to compare this decoder with other hardware designs because it was implemented in an FPGA not as an ASIC.

[Fitz1998] developed an alternative to the traditional RS decoder algorithms called the Fitzpatrick algorithm, and compared it to the Berlekamp-Massey algorithm. This comparison was strictly theoretical and offered no significant information about a hardware implementation. The Fitzpatrick algorithm was viewed as having a lower degree of complexity, but it used $2t^2$ multipliers for the decoding process. Designs with significantly fewer multipliers have been realized, including the ones in this thesis.

[Jenn1998] offered an area/power comparison of an RS decoder based on a Euclidean algorithm and the Fitzpatrick algorithm. The latter algorithm was co-devised by one of the authors in [Jenn1998]. However, the algorithm was not amenable to VLSI design. It was stated that to just find the error locator polynomial alone required at most $2t^2$ multiplications and $2t$ divisions. This would amount to a prohibitively large area. In fact,

[Jenn1998] showed that the area was almost double that found in a Euclidean based algorithm. The large increase in area was accounted for by the large amount of code required for the Fitzpatrick algorithm. Therefore, a higher degree of complexity can be said to be associated with that algorithm. A meager improvement in power dissipation was realized, but with significant trade-offs in area and control complexity. The implementation was only simulated in software and no fabrication of silicon was attempted. In addition, the comparisons were made with RS codes of code length $n = 24$ and $n = 32$. This coding scheme is not applicable to many of today's implementations. The design did not incorporate erasure handling either.

A VLSI chip with published results was discussed in [Chan1999] and [ChSu1999], which used the Euclidean algorithm for the decoder. The chip had a total of 31 000 gates, operated at 40 MHz in the worst case and was fabricated in a $0.6\mu\text{m}$ process. There were significant limitations with this implementation though. A maximum of $t = 6$ errors could be corrected and the highest RS code it could accommodate was (200,188). It used area intensive ROMs as inverse lookup tables. However, it boasted a 16% improvement in hardware complexity over a previous approach [Kwon1997]. The results do not refer to actual silicon area because the references to which this design is compared against do not fabricate any hardware. Finally, the chip's design was too focused on speed with little emphasis placed on area optimization. The applications to which it is targeted, simply do not require decoding speeds in excess of 40 MHz.

A highly efficient, but extremely complex design in terms of control complexity, timing and layout was presented in [Wilh1999]. A time-domain Euclidean algorithm was used for the decoder. The chip was not fabricated, but the results of the proposed design were estimated from a $0.5\mu\text{m}$ technology. This design had a speed of 620 Mbps, an error correcting capability of $t = 8$, and an area of 1.6mm^2 using 43 000 transistors and additional memory. It was estimated that the design could be scaled to operate at 1280 Mbps, which corresponded to an area of 3.0mm^2 with 120 000 transistors plus memory. The design was full-custom however, with a high degree of control complexity. Therefore, time-to-market could be an issue. The decoding speeds were also exceedingly high for most practical applications today.

[Huan1999] designed an RS decoder targeted at ADSL applications. No hardware was produced and all performance parameters were estimated. This design operated at a bit-rate of 48 Mbps, had a gate count of 43 987, and had an error correcting capability of $t = 8$. However, the design did not incorporate the ability to handle erasures. Thus, there would be an increase in the silicon area and degree of complexity if this design was to be used in a

practical ADSL application. The design was a TDD that was based on the Euclidean algorithm. There was no reasoning given as to why the design was chosen to be a TDD instead of an FDD though.

[Jeng1999] presented an RS decoder that used an inverse-free Berlekamp-Massey algorithm. The design was implemented in software using C++, but no hardware was fabricated. A (255,239) RS code was used to illustrate the benefits of the algorithm. However, it was shown that the proposed structure used a total of 113 multipliers. This was quite a large number for the given code parameters. In addition, a finite-field divider was still required by the Chien search block. The decoding algorithm did not use division, but it is possible to use the modified Euclidean algorithm without performing a division as well.

An RS decoder for DVDs, which used a Berlekamp-Massey decoding algorithm was presented in [Chan1998], [ChSh1999] and [Chan2001]. A chip was fabricated in a 0.6 μ m CMOS process for this implementation. It had a total area of 4.22x3.64mm², a core area of 2.90x2.88mm², a gate count of ~26 000, an operating speed of 33MHz, and a power dissipation of 102mW. The contribution from this implementation was significant. A serial architecture which had only three finite-field multipliers was used to implement the algorithm efficiently. However, the finite-field multipliers used in this design were not as optimal as other implementations. In this design, the multiplier was constructed from 73 XOR gates and 64 AND gates. However in [Solj1996], a multiplier for GF(2⁸) was formulated which essentially used only 48 AND gates and 62 XOR gates. RS decoders use numerous multipliers so the design in [Solj1996] was more suitable for the area sensitive design in this thesis. In addition, [Chan1998], [ChSh1999] and [Chan2001] used an area intensive look-up table for the inverse calculations required by the Chien search. However, [Solj1996] presented an area optimized inverter. When used in conjunction with the multiplier, it offered a considerable savings in area over a look-up table. In short, this design offered an efficient decoding algorithm design, but the finite-field mathematical structures were not the most optimal in terms of area efficiency.

To complete this discussion, a few examples of commercial RS decoder implementations will be presented. Radyne ComStream developed an RS Codec [Rady1999] that was DVB compatible. It was based on a (204,188) code, with an error correcting capability of $t = 8$. Furthermore, it supported data rates ranging from 4.8 kbps to 8.5 Mbps. This DVB compatible chip met all DVB specifications.

Texas Instruments developed a software approach [Texa2000] of an RS decoder using C code. It used the BM algorithm and was based on a (204,188) RS code with $t = 8$. The entire

decoding process took 2180 machine cycles to complete. This software was targeted to run on the Texas Instruments' C6400 digital signal processor.

e-MDT Inc. markets a programmable RS codec [eMDT2000]. Its code length was variable between $n = 85$ and $n = 255$. It was implemented in a $0.8\mu\text{m}$ technology, operated at a speed of 80 Mbps, and had an error correcting capability of $t = 8$. In addition, the encoder and decoder operated independently for full duplex operation. It was encased in a 68 pin package and used one clock cycle per byte of processing. The device is suitable for magnetic recording systems and other high-performance storage media applications. It may be adapted to a wide range of wireless applications as well.

LSI Logic has developed a DVB Quadrature Amplitude Modulation (QAM) Modulator [LSIL2000] which used an RS decoder. The modulator can accommodate a (204,188) RS code and it can correct up to $t = 8$ errors. These parameters satisfy the DVB standard specification.

Finally, Advanced Hardware Architectures (AHA), developed a 100 Mbps RS error correction device. It supports several programmable parameters including block size, error threshold, number of check bytes, and order of output. In addition, it can be programmed to correct from 1 to 10 error bytes or 20 erasure bytes per block. These block lengths are programmable from 3 to 255 bytes. It was encased in a 44 pin package and its target applications include HDTV, ADSL, high performance modems, and global positioning systems. A single-phase clock synchronized all chip functions.

Table 2-3 quantitatively summarizes the results from the literature to date. It illustrates the conspicuous lack of implementation detail in published RS decoder research. In addition, none of the publications mention or describe any details about indicating a decoder failure. All of the designs appear to be focused on the decoding algorithm and there was no mention of how to indicate or detect a decoder failure. As can be seen from the ensuing table, there are a smattering of test results which do not provide definitive answers in terms of comparing the TDD and the FDD in hardware. Much of the published literature failed to discuss several important design parameters. This can be seen by the numerous blank entries found in the table.

Table 2-3: RS Decoder Implementation Literature Summary

Reference	RS Code Parameters ^a & Domain	Transistors	Power (mW)	Speed	Line Width (μm)	Area (mm^2)	Norm ^b Area (mm^2)
[Shao1985]	(15,9) FDD EA	–	–	–	–	–	–
[ShTr1985]	Erasures EA (31,15) Both	–88 k	–	–	–	–	–
[Shao1988]	Erasures EA (255,223) Both	–	–	–	–	–	–
[Truo1988]	Erasures EA (255,223) Both	–	–	–	–	–	–
[HsTr1988]	Erasures EA (255,223) Both	–	–	–	–	–	–
[Reed1991]	(n,k) TDD BM	C-code	–	–	–	–	–
[Whit1991]	Erasures EA (167,147) TDD $t = 10$	200 k Full Custom	500	80 Mbps	1.6 2 LM	68.88	0.872
[Choo1992]	(n,k) TDD N/A	C-code	–	–	–	–	–
[Chi1993]	(n,k) TDD N/A	–	–	–	–	–	–
[Chen1995]	Erasures $96 \leq n \leq 255$ TDD EA $1 \leq t \leq 10$	C-code & Verilog (Functional)	–	–	–	–	–
[Jeon1995]	TDD & FDD EA $n = 255$ $1 \leq n - k \leq 30$ e.g., $k = 231$	C-code e.g. = 150 848	–	40 MHz clock e.g.	–	–	–
[Hsu1996]	$n = 255$ TDD EA $5 \leq t \leq 10$	226k to 546k	–	–	–	–	–
[Kwon1997]	Erasures TDD EA (14,9), (85,77), (149,138)	35k gates including encoder	–	36 MHz clock	–	–	–
[Oh1997]	Erasure TDD (207,187) $t = 10$ EA	FLEX10K 50 FPGA 50k gates	–	80 Mbps	–	–	–

Table 2-3: RS Decoder Implementation Literature Summary

Reference	RS Code Parameters ^a & Domain	Transistors	Power (mW)	Speed	Line Width (μm)	Area (mm^2)	Norm ^b Area (mm^2)
[Ragh1997]	(63,57) TDD $t = 3$ BM	48 570, 13 945 <i>gates</i>	–	343 Mbps	2 2 LM	–	–
[Chan1998]	(208,192) TDD $t = 8$ BM	21910 <i>gates</i>	–	16.67 MHz clock	0.6 SPDM	12.35	1.11
[Fitz1998]	(n,k) TDD BM	algorithm	–	–	–	–	–
[Jenn1998]	(24,20) & (32,26) TDD $t = 3$ EA	–	280 (nJ)	111 kHz clock	–	–	–
[Chan1999] [ChSu1999]	(200,188),(60,40) (120,108),(40,28) $t = 6$ EA	~31k <i>gates</i>	–	40 MHz clock	0.6 2LM	–	–
[Huan1999]	(255, k) TDD $t = 8$ EA	43 987 <i>gates</i>	–	48 Mbps	–	–	–
[Jeng1999]	Erasures (255,239) TDD $t = 8$ BM	C++ Code	–	–	–	–	–
[Wilh1999]	Erasures (167,147) TDD $t = 10$ EA	73 000	–	256 Mbps	0.5 ^c	1.7	0.612
[ChSh1999] [Chan2001]	(208,192) & (182,172) TDD $t = 8$ BM	~26 000 <i>gates</i> 99 pins	102 @ 3V	33 MHz clock	0.6 SPDM	15.36	1.38

a. Algorithm used in decoder BM - Berlekamp-Massey or EA - Euclidean Based Algorithm or N/A

b. Area normalized to a 0.18 μm CMOS technology and was calculated as follows. Divide the line width by 0.18 and square the quotient. Then divide the area by this result to find the normalized area in 0.18 μm . Note however that the number of layers of metal were not factored into this calculation.

c. The number of metal layers was not provided in the publication.

2.4 Summary

To put the history of RS decoding in perspective, a succinct outline of the progression of RS decoder research is presented in Fig. 2-2. It shows a timeline of when RS codes originated and then the subsequent research contributions towards finding the most efficient decoding approach. Furthermore, it also shows when each of the major decoding approaches was first discovered.

Fig. 2-2: Reed-Solomon Decoder Outline and Timeline
 (FDD Indicated by Dashed Oval - - -)

Chapter 2 has shown that attempts thus far have failed to give palpable evidence of the superiority of one RS decoder implementation over the other. This has served as the motivational precedence for this thesis. To recapitulate the essential points of this chapter:

- Reed-Solomon codes were described as a specific type of forward error correcting block code.
- The parameters and properties that define an RS code were thoroughly presented.
- Applications that use RS codes, encompassing the target applications of this thesis, were presented. The widespread use of the RS decoder motivates the research behind this thesis.
- The various RS decoding algorithms were explained, including the rationale behind choosing to implement the MEA in this thesis.
- A literature review of previous implementation proposals was presented. It illustrated the inherent lack of tangible evidence to support the VLSI implementation of either the TDD or the FDD.

Consequently, a TDD and an FDD will be designed and fabricated in this thesis to finally provide an authoritative statement on each of the RS decoding approaches. The next chapter describes the mathematics and structures associated with the RS TDD and FDD.

Reed-Solomon Decoder Design

The general structure of RS codes and the algorithms used in decoding them were described in Chapter 2. This chapter goes into further detail and explains the mathematics and methodology involved in designing an RS decoder. The design of an RS decoder can essentially be divided into several distinct functions or blocks. It has a pipelined structure, meaning that the received codeword moves through the various stages until it is decoded and ready to be output. In addition, included in the description will be an explanation of erasure handling. This adds some complexity to the decoder, but at the same time it provides a greater error correcting capability. Nevertheless, erasures can be omitted from the decoder design if the specific application does not require it. Before elaborating on each block though, a brief overview of the general structure of both the TDD and the FDD will be given.

3.1 Implementation Overview

TDD and FDD designs are developed in [Shao1985], [Shao1988], and [HsTr1988], which serve as the basis for this discussion. To begin, the RS code is constructed from $GF(2^m)$ with 2^m elements. Each codeword has a length of $n = 2^m - 1$ and a minimum distance d . The number of message symbols k , that are encoded can be expressed as $k = n - (d - 1)$. This forms an (n, k) RS code with k m -bit message symbols and $(d - 1)$ parity symbols. Throughout the rest of the discussion, the following five vectors will be used:

$\mathbf{c} = (c_0, c_1, \dots, c_{n-1})$, codeword vector

$\mathbf{r} = (r_0, r_1, \dots, r_{n-1})$, received vector

$\mathbf{e} = (e_0, e_1, \dots, e_{n-1})$, error vector

$$\mathbf{u} = (u_0, u_1, \dots, u_{n-1}), \text{ erasure vector}$$

$$\mathbf{w} = (w_0, w_1, \dots, w_{n-1}), \text{ errata vector.}$$

The preceding vectors are related by $\mathbf{w} = \mathbf{e} + \mathbf{u}$ and $\mathbf{r} = \mathbf{c} + \mathbf{u} + \mathbf{e}$. If t errors and v erasures occur in a received vector $\mathbf{r}$, then the decoder can correct them provided that $v + 2t \leq d - 1$. Now a general overview of the TDD will be discussed.

A structural illustration of the TDD is shown in Fig. 3-1. It can be decomposed into two main functional sections: the algorithm processing stage (I) and the output stage (II). First, stage I is where the data codeword $\mathbf{r}$, and the erasure locations $\mathbf{u}$, are received and then decoded using the Modified Euclidean Algorithm. The codeword $\mathbf{r}$ is received one m -bit symbol at a time, and then sent to two different blocks: the syndrome computation unit and the memory or delay unit. Received symbols are stored in memory because they need to be combined at the output stage with the decoded symbols to produce the correct RS codeword. At the same time, the erasure location bit stream is serially input into the power calculation unit. A '1' indicates the presence of an erasure at the current codeword symbol position. If this is the case, then the power calculation unit will convert it to a power of α , where α is primitive element of the GF over which the RS code is defined.

Fig. 3-1: Structural Overview of the Time Domain Decoder

The subsequent section has the syndromes and α powers as its inputs. In this section, three blocks calculate parameters that the MEA requires for its decoding. Each block uses the information from the α power calculation unit. First, the polynomial expansion unit uses the syndromes and α powers to find the Forney syndromes. These syndromes are used to form a polynomial which initializes the MEA to be able to correct both errors and erasures. Second, the α powers are expanded into an erasure locator polynomial in the power expansion block. If an input symbol r_i is labeled an erasure, then α^{-i} will be a root of this erasure locator

polynomial. This polynomial is then used to initialize the MEA. The last block computes $\lfloor (d + v - 3)/2 \rfloor$, which is used as a stop indicator for the MEA.

The following section contains the MEA, which is the fundamental core of the RS decoder. If the degree of the Forney polynomial is less than the degree of the erasure locator polynomial, then there are no errors and the MEA may be skipped. However if there are errors, then a specific decoding algorithm iterates to solve the key equation (Equation (2-10)). Two results are produced from this block: the errata locator and the errata magnitude polynomials.

These polynomials are subsequently passed to the Chien search unit in stage II. Here an exhaustive search is performed to find the roots of the errata locator polynomial by evaluating it for all α^i , where $i = 0, 1, \dots, n - 1$. If a root is found, then the corresponding symbol r_i is corrupted. The errata magnitudes are found by exhaustively evaluating the errata magnitude polynomial and the derivative of the errata locator polynomial for all α^i in the particular Galois field. Finally, these errata values are GF added (XORed) with the original symbols stored in memory. This sum forms the corrected codeword that the decoder outputs.

The FDD, which is illustrated in Fig. 3-2, is quite similar to the TDD except for a few subtle differences. First, an extra delay or memory element is required to store the syndromes from

Fig. 3-2: Structural Overview of the Frequency Domain Decoder

the first section of the decoder. These syndromes are required in stage II. The second difference is the output of the MEA. Only the errata locator polynomial is required from the MEA to initialize the transform error pattern block. The delayed syndromes are input into

this block as well. Nevertheless, the most conspicuous difference from the original TDD is the entirely new output block that handles and interprets the decoding results from the MEA.

The first section of the output stage calculates the m -bit transforms of the errata pattern. Delayed syndromes from the first section in stage I, form the first $n - k$ error transforms. To calculate the remaining k transforms, a recursive equation is used in conjunction with the syndromes and the errata locator polynomial. Each of the m -bit transforms are then sent to the next output block: the inverse transform unit.

The inverse of all n errata transforms must be calculated before being added with the stored input symbols. This inverse transform is taken over $GF(2^m)$. Finally, the symbols can be GF added (XORed) to the stored input symbols and then sent to the decoder output. In short, the FDD and TDD are almost identical except for the key aforementioned differences.

Now that the general structure of both the TDD and FDD have been discussed, each of the blocks will now be elaborated on in full detail. Unless stated otherwise, the explanation of each block pertains to both the TDD and the FDD.

3.2 Syndrome Calculation

The decoder receives m -bit symbols from the input RS codeword r_i , where $(0 \leq i \leq n - 1)$. Both the TDD and FDD store the input codeword symbols in a block of memory for use later on in the final section of their respected output stages. These symbols are also used to calculate the syndrome polynomial given as

$$S(x) = \sum_{k=1}^{d-1} S_k x^{k-1}. \quad (3-1)$$

The S_k values are the syndrome coefficients and are calculated in Equation (3-2), where α is a primitive element in $GF(2^m)$.

$$S_k = \sum_{i=0}^{n-1} r_i \alpha^{ik}, \quad 1 \leq k \leq d-1 \quad (3-2)$$

If all of the coefficients of the syndrome polynomial are equal to zero, or $\alpha^{-\infty}$ in terms of GF representation, then there are no errors or erasures in the input codeword. Hence, the rest of the decoding can be omitted to save time. If the decoding is being implemented in hardware, then this omission also reduces power consumption significantly.

By using what is known as Horner's equation, Equation (3-2) can be rewritten to express the syndrome coefficients as

$$S_k = (\dots(((r_{n-1}\alpha^k + r_{n-2})\alpha^k + r_{n-3})\alpha^k + \dots)\alpha^k + r_0) \quad (3-3)$$

This modification facilitates the design of the syndrome calculation block shown in Fig. 3-3, where the input symbols are shifted in, beginning with r_{n-1} . It is constructed using $d - 1$

Fig. 3-3: Generic Syndrome Calculation Unit

cells, where each cell consists of an m -bit GF adder, constant GF multiplier and register. At any given instant then, each cell k , has an input symbol of r_i and performs the recursive function of $S_k \leftarrow r_i + S_k \alpha^{d-k}$. The syndromes are ready to be output to the polynomial generation block when all n codeword symbols have passed through the unit. However, the FDD also stores the calculated syndromes in delay registers, and these syndromes are then later used in the error transform calculation block found in its output stage.

3.3 Erasure Handling

Erasure handling was included because it increases the error correcting capability of the decoder, and thus the design's usefulness. This block assumes that the erasure location information is received in the form of a binary sequence that is synchronous to the received codeword. The output consists of powers of α , which represent a primitive element in $GF(2^m)$. In addition, this block also keeps track of how many erasures have been detected.

The general structure of the α^k generation unit is shown in Fig. 3-4. It consists of an m -bit constant field multiplier, an m -bit register, a control unit and a $((d - 1) \text{ by } m\text{-bit})$ register block. The register block stores the m -bit powers of α when there is an erasure flag. After the generation is complete, the erasure count is stored and then output to be used later as part

Fig. 3-4: Generic α^k Generation Unit

of the MEA. The powers of α are then output to the next section, which is the polynomial generation unit.

3.4 Polynomial Generation

The polynomial generation unit receives as its inputs: the syndromes, the α^k powers and the erasure count. At the same time, two different internal blocks are used to generate two polynomials that will be used for the MEA. These are the erasure locator polynomial and the Forney syndrome polynomial.

3.4.1 Erasure Locator Polynomial

The erasure locator polynomial $\Lambda(x)$ can be defined as follows. For each received symbol r_i that is labeled an erasure, α^i should be a root of $\Lambda(x)$, such that

$$\Lambda(x) = \prod_{\alpha^i \in \Lambda} (x - \alpha^i). \quad (3-4)$$

This equation can be represented by the structural diagram depicted in Fig. 3-5. It consists

Fig. 3-5: Erasure Locator Polynomial Generation Unit [HsTr1988]

of $(d-1)$ cells which each contain: an m -bit register, GF multiplier and GF adder. The GF adder of the first cell can be omitted because one of the summands is zero. Each of the registers are cleared when erasure powers are received from the next new input codeword. When the nonzero outputs of the α^k generation block are shifted in, the switches are closed to store the new values in the registers. Otherwise, these switches are left open. The

generated polynomial $\Lambda(x)$, is then passed to the MEA as one of the initialization parameters. In addition, it is also used in the algebra that forms the Forney syndrome polynomial.

3.4.2 Forney Syndrome Polynomial

Another polynomial required for the MEA initialization is the Forney syndrome polynomial $T(x)$. To calculate $T(x)$, the syndrome polynomial $S(x)$ is multiplied by the erasure locator polynomial $\Lambda(x)$, such that $T(x) = S(x)\Lambda(x) \bmod x^{d-1}$. This can be expressed as

$$T(x) = \sum_{k=1}^{d-1} T_k x^{k-1} \quad (3-5)$$

However for the sake of expediency, the calculation of $T(x)$ can be found without using $\Lambda(x)$ directly in the calculation. Instead, as was illustrated in Fig. 3-1 and Fig. 3-2, the erasure α powers can be directly input into this block. This concept is shown in Fig. 3-6. The block

Fig. 3-6: Forney Syndrome Polynomial Generation Unit [HsTr1988]

consists of $(d-1)$ cells, which each contain an m -bit register, GF multiplier and GF adder. However, the first cell does not contain a GF adder because one of the summands is zero and cell $(d-1)$ does not contain a GF multiplier. Registers are initialized with the coefficients S_k from the syndrome polynomial $S(x)$. The switches remain open until a nonzero power of α is shifted in, at which point all of the switches close for that particular symbol.

The other block in this section is the degree calculator. It uses the number of erasures v , to calculate $\lfloor (d + v - 3)/2 \rfloor$. This result is passed on to the MEA block and used as a stop indicator for the algorithm.

All of the necessary information that is required by the MEA is now ready to be input to the MEA block, which is the final section in stage I of the decoder. At the end of stage I however, the structure of the FDD and TDD diverges.

3.5 Modified Euclidean Algorithm

The Modified Euclidean Algorithm is used as the fundamental decoding algorithm for both the TDD and the FDD in this thesis. Its purpose is to calculate the errata locator polynomial $\sigma(x)$, and the errata evaluator polynomial $\omega(x)$. However, the structure of the FDD's output stage only requires the errata locator polynomial to be found, but both polynomials are needed in the TDD. Nevertheless, the same methodology is followed in both decoders, and the algorithm is presented next.

Consider the following equation

$$R_i(x) = T(x)\lambda_i(x) + M(x) \quad (3-6)$$

where $T(x)$ is the Forney syndrome polynomial and $M(x)$ is defined to be $M(x) = x^{d-1}$. The MEA recursively calculates the i^{th} remainder $R_i(x)$ and also $\lambda_i(x)$. To begin the MEA, the following initializations are performed:

$$\begin{aligned} \mu_0(x) &= \Lambda(x) & R_0(x) &= x^{d-1} \\ \lambda_0(x) &= 0 & Q_0(x) &= T(x) \end{aligned}$$

Next, the following iterations are computed:

$$\begin{aligned} R_i(x) &= [\sigma_{i-1}b_{i-1}R_{i-1}(x) + \bar{\sigma}_{i-1}a_{i-1}Q_{i-1}(x)] \\ &\quad - x^{l_{i-1}}[\sigma_{i-1}a_{i-1}Q_{i-1}(x) + \bar{\sigma}_{i-1}b_{i-1}R_{i-1}(x)] \end{aligned} \quad (3-7)$$

$$\begin{aligned} \lambda_i(x) &= [\sigma_{i-1}b_{i-1}\lambda_{i-1}(x) + \bar{\sigma}_{i-1}a_{i-1}\mu_{i-1}(x)] \\ &\quad - x^{l_{i-1}}[\sigma_{i-1}a_{i-1}\mu_{i-1}(x) + \bar{\sigma}_{i-1}b_{i-1}\lambda_{i-1}(x)] \end{aligned} \quad (3-8)$$

$$Q_i(x) = [\sigma_{i-1}Q_{i-1}(x) + \bar{\sigma}_{i-1}R_{i-1}(x)] \quad (3-9)$$

$$\mu_i(x) = [\sigma_{i-1}\mu_{i-1}(x) + \bar{\sigma}_{i-1}\lambda_{i-1}(x)] \quad (3-10)$$

The leading coefficients of $R_{i-1}(x)$ and $Q_{i-1}(x)$ are designated as a_{i-1} and b_{i-1} , respectively. For brevity, the degree of a polynomial will be denoted as $\deg$. The rest of the required equations are given as follows

$$l_{i-1} = \deg(R_{i-1}(x)) - \deg(Q_{i-1}(x)) \quad (3-11)$$

$$\sigma_{i-1} = 1 \quad \text{if } l_{i-1} \geq 0 \quad (3-12)$$

$$\sigma_{i-1} = 0 \quad \text{if } l_{i-1} < 0. \quad (3-13)$$

The recursive algorithm stops when $\deg(\lambda_i(x)) > \deg(R_i(x))$. A maximum of $2t$ recursions are required to obtain the errata polynomials. Now, let the errata locator polynomial $\sigma(x) = \lambda_i(x)$ and the errata evaluator polynomial $\omega(x) = R_i(x)$.

Both of the errata polynomials carry a common scale factor compared to those computed by the original Euclidean algorithm. In the TDD, the factor can be ignored and the two results are passed directly to the output stage. The reasoning for this will become apparent shortly. However, the FDD only uses $\sigma(x)$ in its output stage, so the scale factor must be eliminated. This is performed by normalizing the polynomial by dividing it by its leading coefficient as follows:

$$\sigma(x) = \lambda(x) = \lambda_0 x^{t+v} + \lambda_1 x^{t+v-1} + \dots + \lambda_{t+v}$$

$$\sigma(x) = x^{t+v} + \sigma_1 x^{t+v-1} + \dots + \sigma_{t+v}$$

$$\text{where } \sigma_k = \lambda_k / \lambda_0, \quad 1 \leq k \leq t+v, \text{ and } \lambda_0 \neq 0$$

The implementation of this algorithm is realized by cascading $d - 1$ recursion cells [Shao1985]. However, it was shown in [Shao1988] that the area of the MEA block could be reduced by only using two MEA cells for a (255,239) RS code. The liability in doing so is an increase in control logic complexity. To illustrate, Fig. 3-7 shows the modified structure in [Shao1988]. The original recursion cell from [Shao1985] is shown within the dotted square.

Each polynomial required in Fig. 3-7, is serially shifted in by its coefficients for decreasing powers of x . This concludes the decoding process in stage I, and the data can now be processed by the output stage.

3.6 Time-Domain Decoder Output

Both the errata evaluator and errata locator polynomials are used in the TDD. Therefore, there is no need to normalize the polynomials to eliminate the scale factor. The roots of $\sigma(x)$ are the inverse locations of the t errors and v erasures. To calculate these roots, a method known as the Chien search [HsTr1988] is used.

In the Chien search, the roots are found by evaluating the errata locator polynomial $\sigma(x)$, with α^{-i} , where $i = 0, 1, \dots, n - 1$. If $\sigma(\alpha^{-i}) = 0$, then r_i is a corrupted symbol and needs to be corrected. The magnitude of the errata pattern of the corrupted symbol is subsequently

Fig. 3-7: One Recursive MEA Cell [Shao1988]

calculated by evaluating $\omega(x)$ and $\sigma'(x)$ for α^{-i} , where $i = 0, 1, \dots, n-1$. Therefore, the errata pattern magnitude is

$$\hat{e}_i = \frac{\omega(\alpha^{-i})}{\sigma'(\alpha^{-i})} \quad 0 \leq i \leq n-1. \quad (3-14)$$

This division cancels the scale factor that occurs in both $\omega(x)$ and $\sigma(x)$. Therefore, this eliminates the need to use an additional area intensive GF division operation at the output of the MEA.

A structure that can evaluate a polynomial can be explained as follows. Consider the polynomial for $x = \alpha^{-j}$, $j = 1, 2, \dots, n$:

$$A(x) = \sum_{i=0}^{d-2} a_i x^i \quad (3-15)$$

Therefore,

$$A(x)|_{x=\alpha^{-i}} = \sum_{j=0}^{d-2} a_j(\alpha^{-i})^j. \quad (3-16)$$

So each value of $a_j(\alpha^{-i})^j$ can be calculated by the recursive multiplication of a fixed constant α^i , as j varies from 1 to n . To accomplish this, a GF summation of $d - 1$ terms is performed. A practical realization of this arithmetic is shown in Fig. 3-8.

Fig. 3-8: Polynomial Evaluation Circuit for $\sigma(x)$ and $\sigma'(x)$

A unique property of the derivative of a polynomial enables the simultaneous evaluation of $\sigma'(x)$ and $\sigma(x)$. Only the odd power terms of $\sigma(x)$ are needed in order to calculate $\sigma'(x)$. There are $d - 1$ cells, each of which contain an m -bit register and a constant GF multiplier. For $\sigma(x)$, all products generated by the cells are GF added (XORed), but for $\sigma'(x)$, only the odd numbered products are used. A similar structure can be used for $\omega(x)$, where all the products are summed to produce $\omega(\alpha^i)$ since the derivative is not required. Finally, if the symbol is not flagged as being an error then the errata magnitude value output by the Chien search is zero ($\alpha^{-\infty}$).

The final step in the TDD is to correct the input codeword that is stored in memory and output the decoded codeword. This is accomplished by adding each of the stored input symbols with each calculated errata magnitude value to yield the decoded codeword

$$C_i = r_i + \hat{e}_i \quad 0 \leq i \leq n - 1, \quad (3-17)$$

thereby completing the decoding process of the TDD. The output stage for the FDD will now be discussed.

3.7 Frequency-Domain Decoder Output

The FDD requires the errata locator polynomial $\sigma(x)$, to be input from the MEA and not the errata magnitude polynomial. However, $\sigma(x)$ must first be normalized in order to be in the correct format for the remaining error transform block. The normalized version of $\sigma(x)$ along with the stored syndromes from stage I are used to calculate all of the errata transforms.

3.7.1 Remaining Error Transform

The first $d - 1$ terms of the error transform E_k , are already known and they are the syndromes from stage I. To calculate the remaining transforms a recursive equation is used. The remaining $d \leq k \leq n$ elements of E_k are found by

$$E_{d-1+j} = \sum_{k=1}^{v+t} \sigma_k E_{d-1+j-k}, \quad j \geq 1 \quad (3-18)$$

For practical implementations, this equation can be realized with the configuration shown in Fig. 3-9. The syndromes are serially input and form the first $d - 1$ elements of E_k . There are

Fig. 3-9: Remaining Error Transform Block

$d - 1$ cells each containing two m -bit registers, an m -bit GF multiplier and an m -bit GF adder. However, in cell $d - 1$, the adder is omitted because one of the summands is zero. The values shifted out of register R_1 are the error transforms after the $d - 1$ syndromes are shifted through. Nonzero coefficients from the normalized errata transform polynomial $\sigma(x)$, initialize the second register in each cell. Once all n error transforms have been shifted out, the next step in the FDD is to calculate the inverse error transforms that will be used to correct the input codeword.

3.7.2 Inverse Error Transform

The inverse transforms e_k , of the error transform pattern E_k , from the previous block are

$$e_k = \sum_{i=0}^{n-1} E_i \alpha^{-ik}, \quad 0 \leq k \leq n-1 \quad (3-19)$$

Equation (3-19) is found to be quite similar to Equation (3-2), which was used to calculate the coefficients of the syndrome polynomial. Hence, a similar implementation structure shown in Fig. 3-10 is used to calculate the inverse error transform. It consists of n identical cells that each contain two m -bit registers, a GF adder and a constant GF multiplier. The error transform symbols are serially shifted into this block as soon as they are calculated in the error transform block.

Fig. 3-10: Inverse Error Transform Block

The final step in the FDD is identical to the procedure used in the TDD. Each of the stored input codeword symbols are added to each inverse transform symbol e_k , in Equation (3-17). As in the TDD, if an input symbol is not corrupted then its corresponding e_k will be all zero. The summation results in the decoded codeword symbols, which are then output from the decoder. Thus, it can be seen that the FDD structure is heavily dependent upon the size n of the RS codeword. This contrasts with the TDD format, which is contingent on t , the error correcting ability of the RS code. Therefore it can be claimed, but not completely confirmed that the TDD is more practical for larger RS codes since the size of the FDD's area becomes prohibitive for a reasonable VLSI implementation. The results from the actual hardware should fully support this preliminary statement about the area of the decoders.

3.8 Summary

In this chapter, the theory behind the design of the TDD and FDD was thoroughly discussed. Specifically, equations and block diagrams were provided to explain RS decoder design. Table 3-1 summarizes the results of this chapter by showing the decoder elements as a function of code parameters (n, k) for both decoders. It gives an approximation of the number of m -bit registers (m from $GF(2^m)$), GF adders and GF multipliers that are required for each block. This is provided as a preliminary benchmark for the results of this thesis.

Table 3-1: Decoder Elements as a Function of Code Parameters^a

Decoder Block ^b	TDD Requirements	FDD Requirements
Syndrome Calculation	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k)$ GF adders • $(n-k)$ constant GF mult. 	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k)$ GF adders • $(n-k)$ constant GF mult.
Erasur e Handling	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k)$ GF adders • 1 constant GF mult. 	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k)$ GF adders • 1 constant GF mult.
Erasur e Locator Polynomial	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k-1)$ GF adders • $(n-k)$ GF mult. 	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k-1)$ GF adders • $(n-k)$ GF mult.
Forney Syndrome Polynomial	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k-1)$ GF adders • $(n-k-1)$ GF mult. 	<ul style="list-style-type: none"> • $(n-k)$ registers • $(n-k-1)$ GF adders • $(n-k-1)$ GF mult.
MEA	<ul style="list-style-type: none"> • 7 x $(n-k)$ registers • 3 x $(n-k)$ GF adders • 4 x $(n-k)$ GF mult. 	<ul style="list-style-type: none"> • 7 x $(n-k)$ registers • 3 x $(n-k)$ GF adders • 4 x $(n-k)$ GF mult.
TDD Output	<ul style="list-style-type: none"> • 2 x $(n-k)$ registers • $(4n-4k-2)$ GF adders • 2 x $(n-k)$ GF mult. 	0
Remaining Error Transform	0	<ul style="list-style-type: none"> • 2 x $(n-k)$ registers • $(n-k-1)$ GF adders • $(n-k)$ GF mult.
Inverse Error Transform	0	<ul style="list-style-type: none"> • 2 x (n) registers • (n) GF adders • (n) constant GF mult.
Total	<ul style="list-style-type: none"> • $(13n-13k)$ registers • $(11n-11k-4)$ GF adders • $(n-k+1)$ constant GF mult. • 8 x $(n-k) - 1$ GF mult. 	<ul style="list-style-type: none"> • $(15n-13k)$ registers • $(9n-8k-3)$ GF adders • $(2n-k+1)$ constant GF mult. • 7 x $(n-k) - 1$ GF mult.

a. where n = codeword symbols, k = message symbols, and mult = multiplier(s)

b. all registers, adders, and multipliers have a length of m -bits

In short:

- A basic implementation overview of both the TDD and FDD was shown, which divided the decoding process into distinct stages and blocks.
- The mathematical theory governing the designs of an RS TDD and FDD were presented, including a general explanation of how to practically realize each section.
- It was shown that both decoders are essentially identical up to the Modified Euclidean Algorithm. The main difference is the structure of the output stages of each decoder.

The following chapter presents the VLSI design and performance results of each of the RS decoder implementations.

Hardware Implementation

One of the fundamental objectives of this thesis was to implement both the FDD and TDD in VLSI hardware in order to ascertain a definitive statement about the two RS decoder implementations. Chapter 3 elaborated on the algebra and theory that are used to construct RS decoders. Both the TDD and the FDD were discussed in terms of the mathematics and structure involved in each approach.

In this chapter the hardware design, VLSI methodology, and final test results for each decoder will be presented. A silicon implementation of the TDD was produced and the FDD was designed up to the point of, but not including actual fabrication. Chapter 4 is divided into three sections. First, the TDD VLSI approach is explained, which is then followed by the FDD hardware presentation. Finally, the last section offers a comparative analysis of the two decoding approaches.

4.1 Reed-Solomon Time-Domain Decoder

The design of the RS TDD followed the structure outlined in the previous chapter. However, this thesis attempted to achieve the most optimal RS decoder in terms of area and power consumption while maintaining a sufficient speed to meet the target application's timing requirements. Therefore, some modifications were made to improve the efficiency of the design.

4.1.1 VLSI Architecture

The structure of the RS decoder chip was chosen to be comparable to existing products available today such as those in [Core2000], [Xili2000] and [4i2i2000]. The RS design in [Core2000] was only preliminary and no real specifications were provided as to its performance. However, it required two external synchronous double-port RAM blocks with a clock input, an active-low write-enable input, an address input and an active-low chip-enable input. Another HDL RS core implementation was found in [4i2i2000]. This design included the encoder, optional erasure decoding, parameterizable (n,k) RS code size, and other optional features. An estimate of the number of gates for a maximum $t = 8$ RS encoder/decoder was given as 47 000 gates, of which 35 000 were used for the decoder. For a (255,239) RS code, the decoding speed was stated as being 4500 clocks per codeword. No specific rates were provided though. Finally, [Xili2000] markets an FPGA realization of an RS decoder. It can implement a variety of codes including symbol sizes ranging from three to twelve bits. In addition, it can support erasures and different primitive polynomials. For a (204,188) RS code for DVB, the following specifications were provided for the XC4000/Spartan FPGA: no erasure support, 8-bit symbols, 702 configurable logic blocks (CLBs) with 82 unused CLBs, and a maximum clock frequency of 47MHz.

The designs in this thesis target a (255,239) RS code that incorporates erasure handling. These decoders have an error correcting capability of $t = m = 8$ errors. Overall the decoder can correct e errors and v erasures provided that $v + 2e \leq d - 1$, where $d - 1 = 16$. Although the structure of the TDD resembles that presented in the previous chapter, it is architecturally different.

As a preliminary estimate, the results from Table 3-1 in Chapter 3 are used to compare the hardware requirements of a TDD and an FDD before any optimizations are performed. Based on the target (255,239) RS code, the following components are required:

Table 4-1: Non-Optimized Hardware Requirements for (255,239) RS Decoders

Hardware	TDD	FDD
Registers (8-bit)	208	718
GF Adders (8-bit)	172	386
Constant GF Multiplier (8-bit)	17	272
GF Multiplier (8-bit)	127	111

These hardware requirements are for a general case RS decoder and do not take into consideration architectural optimizations, control logic or required memory blocks. In particular, the TDD has added area and complexity to that shown in the table with its memory requirement. At this early stage however, it can be seen that the TDD appears to be superior to the FDD in terms of the silicon area requirement. Yet at this point, it is difficult to quantitatively estimate the relative area benefits of one implementation over the other one. Modifications were made in order to achieve an optimal area for each design.

The design was partitioned into the following four pipelined stages:

- (i) Syndrome calculation block, α^k generation block and input symbol memory block.
- (ii) Forney syndrome polynomial calculation block, degree calculation block, erasure locator polynomial generation block, and MEA block.
- (iii) Chien search unit.
- (iv) Decoding status determination (Success/Failure?) and GF addition of the decoded symbols with the delayed input symbols.

This design used two clocks: *clock* and *symbol_clock*. The *clock* was chosen to be four times the speed of *symbol_clock* and both clocks were completely in phase. Moreover, *symbol_clock* was responsible for driving flip-flops that had the codeword symbols as inputs and was targeted to operate at 20MHz. The *clock* was used everywhere else in order to be able to achieve the speeds of the target applications. For instance, *symbol_clock* clocked the received symbols into the decoder and the decoded symbols out of the decoder. However, *clock* was used in areas such as controller blocks. The length of time required for each pipeline stage was 255 *symbol_clock* cycles. This was the amount of time required to shift in one RS codeword.

The reasons for choosing the preceding clock scheme were as follows. First, an attempt was made to have the decoder be consistent with current implementations, such as those found in [Core2000]. This would increase the applicability of the results from this thesis. Second, the optimized area design of the modified Chien search block from [Kwon1997] was used instead of the traditional approach in [Shao1988]. A significant savings in area was realized by using the modified design, but the trade-off was that a faster clock was needed to meet the timing requirements. This approach was taken because area was an important design parameter that this thesis focused on optimizing. Finally, this clocking scheme was suitable for the available memory core modules. The division of the pipeline stages and implementation of the memory cores were facilitated by using these clock rates. This will be

further clarified by an explanation of the partitioning and design of the memory module in the ensuing section.

One of the blocks that has a detrimental effect on the optimization of all RS decoder designs is the memory or delay required for storing the input codewords. Traditionally this was done through the use of register blocks. However, this aspect was accomplished by using a 256x32bit RAM core module cell. The structure of the RAM block design is shown in Fig. 4-1.

Fig. 4-1: Memory Block Partition

All of the received symbols were 8-bit words, but unfortunately the memory core only had a 32-bit read/write capability. Therefore, the circuit was partitioned as illustrated in Fig. 4-1.

To make the most efficient use of the block, it was divided into four sections with only one byte being unused in each section. Since the decoder was implemented as a four stage pipeline circuit, the read operation was offset from the write by four memory blocks. For instance, a codeword C_1 starts being written into memory starting at address \$80. Each of these writes are 32-bits long, so the first block of symbols is stored in addresses \$80-\$83. At the same time, the output block starts reading symbols from address \$c0 of the memory in order to add them with the decoded error symbols. Similarly, each of the reads are 32-bits long, so the first block of symbols are retrieved from addresses \$c0-\$c3. After the last block of symbols from C_1 are finished being stored at addresses \$bc-\$be, a new cycle begins. A new input codeword C_2 is written into the memory cell starting at address \$c0 and another is simultaneously read from starting address \$00. It therefore takes four pipeline cycles from the time a codeword is input into memory to the time it is retrieved from memory.

This structure allowed for a substantial savings in power. Data was written to and read from memory in 32-bit word sizes, but the decoder used 8-bit symbols. Therefore, the RAM only needed to be powered up on every fourth cycle. An alternative could be to use memory that supported byte writes. However, this thesis had to make due with the memory cores that were available.

The 8-bit data symbols needed to be accumulated, so it was necessary to build an interface circuit at the input and output of each memory block. This circuit is illustrated in Fig. 4-2.

Fig. 4-2: Memory Interface Circuit

Each of the thin line arrows represents an 8-bit data bus. The operation at the input and output were similar, so for expediency, only the input interface will be explained. Symbols would be stored in registers R_1 to R_3 . No register is used for the fourth symbol. This is because at the time when the fourth symbol is input, the write port becomes enabled and all four symbols are concatenated then stored in the RAM. This was done to minimize the number of clock cycles required for reading and writing. A similar operation is performed when reading from the RAM.

In order to achieve the minimum decoding speed required for the target applications, the syndrome block was essentially kept the same as in other designs. The syndrome block contains sixteen cells and is shown in Fig. 4-3. To save area, hardwired constant multipliers

Fig. 4-3: Syndrome Calculation Unit

were implemented. A significant savings in silicon area was realized. In fact, in some cases a 97% reduction in area over the variable input multiplier was realized. This was largely dependent upon the 8-bit value of the constant being multiplied.

The multiplier architecture from [Solj1996], which essentially used 48 AND gates and 62 XOR gates, was used throughout this design. An additional 18 XOR gates for the *entire* decoder are required to use these multipliers. There needs to be 9 XOR gates at the input and 9 XOR gates at the output of the decoder. This is for changing the input data symbols from the standard notation to a composite notation [Solj1996] for use inside the decoder. Once the output symbols are found, they are then transformed back into standard notation. The functionality of the decoder is not changed in any way. This block used the *symbol_clock* to clock in the received input symbols. Therefore, this section took a total of 255 *symbol_clock* cycles to compete.

The erasure detection circuitry has also been modified from other approaches. In [Sohi2000], no multiplier was used for determining the powers of α . However, this circuitry would be too slow for the target applications of this thesis and no real savings in area is gained from this approach. By modifying the circuit in [HsTr1988], an efficient α power generator can be created. Thus, the generator used in this design is illustrated in Fig. 4-4. One 8-bit register

Fig. 4-4: α^k Generation Block

and one GF constant multiplier are used in the design. In addition, there are sixteen registers to hold the possible α powers that correspond to erasures. Since the maximum erasure correcting capability is sixteen, there is no need to have more registers. The control unit keeps count of how many erasures have been received. If too many erasures are input then the control unit declares a decoding error.

The α^k generation block operates in parallel with the syndrome calculation unit. This is a requirement because the powers of α indicate that the current codeword symbol position is an erasure location. So it is imperative that the α power in register R in Fig. 4-4, matches the number of the codeword symbol position.

The next section of the decoder contains the polynomial generation blocks. A considerable savings in area was realized using the modified designs in [Oh1997]. The process of finding the Forney syndromes and the erasure locator polynomial can be done serially instead of in parallel. This reduced the GF multiplier and GF adder count in each block from sixteen to one. The trade-off was a lower latency but this block was clocked by *clock*, so it had ample time to complete. Timing was therefore, not an issue.

To further expand on this concept, the modified erasure locator polynomial generator is shown in Fig. 4-5. The extra register S_{17} is only used if there are sixteen erasures.

Fig. 4-5: Modified Erasure Locator Polynomial Generation Block

Successive shifting operations replace the excessive parallel multiplications required in the original approach from [HsTr1988]. The presence of the multiplexor (MUX) was needed in order to skip the addition on the sixteenth shift during the current erasure power of α^k . This was necessary for the successful functional operation of the block.

Similarly, the same approach was be used to construct the Forney syndrome polynomial generation block. The modified design is provided in Fig. 4-6. Again, only one multiplier and

Fig. 4-6: Modified Forney Syndrome Polynomial Generation Block

one adder are used in conjunction with register shifting. The syndromes are used to initialize the registers and the first Forney syndrome remains the same as the first input syndrome, S_1 . Therefore, S_1 must be stored during the shifting operations and then reloaded after each set of sixteen shifts. The number of sets of sixteen shifts depends on how many erasure powers of α^k are present for the current codeword. In the worst case of sixteen erasures there would need to be 16 *shifts* x 16 *erasures* = 256 *shifts* performed. This block is

clocked with *clock* and the number of cycles that it takes to complete vary with the number of erasures present in the input codeword.

The next architectural improvement was with the MEA. [Shao1988] suggested that for a (255,239) RS decoder, two MEA cells can be used repetitively instead of using sixteen cascaded cells. The savings in area was offset by the increased complexity of the control module for this block. However, it was found that only one MEA cell was needed to meet the timing requirements of the target applications if *clock* was used. The outputs of the MEA cell are delayed and then fed back as inputs. Therefore by using a single MEA cell, the decoder area is dramatically reduced and the overall complexity is simplified as well. The completion time is highly irregular and varies according to the codeword and the number of errors and erasures that are present. However, there was always sufficient time for the decoding to complete because of the timing requirements of the preceding and succeeding pipeline stages.

As previously mentioned, the Chien search unit was also modified from the traditional approach in [Shao1988]. By using the suggested architecture in [Oh1997], the area of this block was reduced significantly. The new structure is illustrated in Fig. 4-7. Hardware blocks

Fig. 4-7: Modified Chien Search Block

are shared to compute $\sigma(\alpha^{-i})$, $\sigma_{\text{odd}}(\alpha^{-i})$ and $\omega(\alpha^{-i})$. The results can be summarized as follows:

- (i) Reduction from 4 XOR tree summation blocks to 2.
- (ii) Reduction from 33 GF multipliers to 17.
- (iii) Completion time was doubled.

Timing was not an issue since *clock* was used. The main objective here was to reduce the area for the RS decoder. During even *clock* cycles, $\sigma(\alpha^{-i})$ and $\sigma_{\text{odd}}(\alpha^{-i})$ are computed, while

during odd cycles, $\omega(\alpha^{-i})$ is computed. In addition, a memory cell was used to store the error symbol results from the Chien search block. The memory module architecture and design were similar to the ones used for storing the input codewords (Fig. 4-1 and Fig. 4-2). The non-zero error pattern symbols that are generated from the Chien search are counted, and then used to determine if a decoder failure has occurred.

The Chien search block involved calculating the errata pattern magnitudes, which were stated previously in Equation 3-14 as being

$$\hat{e}_i = \frac{\omega(\alpha^{-i})}{\sigma'(\alpha^{-i})} \quad 0 \leq i \leq n-1.$$

This step involved a GF division. A finite-field division can be partitioned into two separate operations: an inversion and a multiplication. Instead of using an area intensive 255x8bit look-up table for inverses, a much more optimal approach from [Solj1996] was used. The inverter used 58 AND and 71 XOR gates. Furthermore, this divider implementation also reduced the complexity involved with a GF division. Using a look-up table would require additional complicated control circuitry to retrieve the correct inverse symbol. Designs can be facilitated by using the simple, but yet highly efficient architecture in [Solj1996].

Power consumption in the RS decoder was attempted to be reduced as well. This was accomplished by using the error and erasure information generated by the decoder. For instance, if the sixteen syndromes are all calculated to be zero then there are no errors, and the entire decoding process can be skipped. Thus, registers involved in decoding are not enabled, which reduces the power consumption.

In addition, memory cores were used where possible to reduce the number of register shifting operations. Moreover, the memory blocks were disabled when not reading or writing to them. In fact, the memory blocks were only powered on one out of every four clock cycles. The data bus was 32-bits wide, but the data of the decoder was 8-bits wide. This means that data needs to be accumulated for a write and partitioned for a read.

The following is a description of a write operation that serves to illustrate the power conservation operation of the memory. To begin, the term “clock” in the following explanation will refer to either the *clock* or *symbol_clock*. The write operation is the same regardless of which clock is connected to the memory’s clock ports. Three bytes of data are stored in three temporary 8-bit registers over a duration of three “clock” cycles. Then upon the fourth “clock” cycle, the last byte of data is concatenated to the first three bytes of data. The memory is enabled and the 32-bits of data are written in at this time. A similar procedure was used for

reading data from the memory as well. In that case, data is shifted out in 32-bit lengths and then partitioned and stored in temporary registers.

The memory approach eliminates the need of having 255 8-bit registers, which are switching at the same time on every “clock” cycle. In addition, a further reduction in power consumption was achieved by using the 32-bit read/write size of the memory data bus. This allowed the RAM to be powered down for three out of every four “clock” cycles. The address decoding and word lines did not have to be powered up continually. This is important because these two blocks could substantially increase the power requirement if they were required to be powered up continuously. Thus, a considerable savings in power was realized by powering down the memories when they were not in use.

An improved RS TDD design in terms of area optimization and power reduction was achieved by implementing the preceding architectural modifications. The next section presents the VLSI results from the fabrication of this decoder design.

4.1.2 Implementation Results

The functionality of the RS TDD was verified through the use of MATLAB code, which is provided in Appendix B. Once the algorithm operated correctly, the design was implemented using Verilog. The testing procedure was based on a bottom-up approach. Each lower level module was designed, coded and then tested with separate testbenches. Then, this methodology was repeated when these blocks were combined to form a new level of hierarchy. To increase the likelihood of a successful fabrication, testing occurred at several stages of the design. Once the HDL coding of the entire decoder was complete, it was verified to operate correctly through the use of a global testbench. Then after synthesis in Synopsys, the gate-level netlist was tested with the same testbench and verified. The last test confirmed that the post-layout, gate-level netlist functioned correctly. This was the TDD design that was fabricated. For static timing analysis, Pearl was used to verify that there were no setup or hold violations for the decoder.

The RS TDD was implemented in a six-metal layer 0.18 μ m CMOS process technology. A standard-cell automated place-and-route was used to generate the layout in order to reduce design time. The pins on the chip are listed and described in Table 4-2. Both the TDD and the FDD used the same pin configuration. Similarly, for both decoders, there were several pins which were not used. The reasoning for this is explained in Section 4.1.3.

Table 4-2: RS Decoder I/O Pin Description

Pin	Description
rxed_symbol[7:0]	The 8-bit symbols of the (255,239) RS codeword that are being received as inputs to the decoder.
new_word	Indicates for one <i>clock</i> cycle that a new codeword has arrived to the decoder.
resetb	Global active-low asynchronous reset
clock	The global clock (4 times as fast as <i>symbol_clock</i>).
symbol_clock	The symbol clock (clocks symbols in/out of decoder).
erasure	Indicates that the current <i>rxed_symbol[7:0]</i> is an erasure location.
tie_high	Global tie-high signal.
test_mode	Global tie-low signal. (Originally intended to force the memory outputs to a known value for scan chain purposes.)
test_se	Scan-chain enable. (Not used)
test_si_slow	Scan in for <i>symbol_clock</i> registers. (Not used)
test_si_fast	Scan in for <i>clock</i> registers. (Not used)
test_out_fast	Scan out for the <i>clock</i> registers. (Not used)
test_out_slow	Scan out for the <i>symbol_clock</i> registers. (Not used)
data_valid	Determines whether or not the <i>decoded_symbol[7:0]</i> output is valid or not. It becomes and stays high shortly after a global reset of the decoder.
decoder_failure	Denotes that the current <i>decoded_symbol[7:0]</i> output is incorrect. There are more errors/erasures than the decoder is capable of handling.
decoded_symbol[7:0]	The 8-bit corrected (255,239) RS codeword symbols that are being output from the decoder.

The VLSI specifications shown in Table 4-3 were the results obtained from testing the final post-layout, gate-level netlist.

Table 4-3: Reed-Solomon TDD VLSI Results

Parameter	TDD Chip Characteristics
Process	MOSIS 0.18 μ m CMOS 6 layer-metal Virtual Silicon Library Cells [Virt2001]
Core Area	1.44mm x 1.04mm = 1.50mm ²
Total Area	2.09mm x 1.69mm = 3.54mm ²
Supply Voltage	1.8V
Memory Blocks	Two 32x256bit 6 Transistor embedded SRAM cores
Standard Cells	8482
Registers	2206
Transistors ^a	305,538 (154,974 without SRAM)
Pin Count	44 (14 pwr/gnd, 30 I/O)
Packaging	44 CQFP
Speed	22.0 MHz symbol rate 176 Mbps bit rate
Tested Speed (20MHz Tester) ^b	20 MHz symbol rate 160 Mbps bit rate
Power Consumption (1.8V)	58 mW (tested) 56 mW (estimated)

a. The first number represents all the transistors used in the entire design, including the memory cores and I/O pads.

b. Faster testers could probably have resulted in a higher bit-rate.

The worst-case operating speed of 176 Mbps (22.0 MHz) meets the timing requirements of the target applications in this thesis. A die photo of the chip is provided in Fig. 4-8.

Fig. 4-8: TDD Die Photo

4.1.3 ASIC Fabrication Testing

The ASIC designs in this thesis did not incorporate scan-chains because of the presence of tri-state outputs on the memory cores. However, it was later discovered that the Synopsys command called *insert_dft*, could successfully generate scan-chains for designs with tri-state outputs. Unfortunately, this command was found after both the TDD and the FDD had been fully designed and created. Therefore, the *insert_dft* instruction could be used in future synthesis procedures to ascertain whether the fabrication had any associated manufacturing faults. Thus the only testing that was performed was functional testing. A 20MHz tester was used to verify the functionality of the decoder. A discussion of the RS FDD design and acquired results is next.

4.2 Reed-Solomon Frequency-Domain Decoder

The design of the RS FDD followed the structure outlined in the previous chapter. In terms of architecture it is relatively similar to the TDD just presented. Furthermore, the same optimizations were targeted, namely, area and power consumption. The operating speed was required to simply meet or exceed the timing dictated by the target applications.

Architectural modifications were made to improve the efficiency of the design over previous approaches.

4.2.1 VLSI Architecture

Structurally, the first stage of the decoder was similar to that presented in Section 4.1 for the TDD. All of the architectures before the output stage of the decoder were reused from the TDD design. There were some minor differences nonetheless. For instance, an extra block of registers was added to store the sixteen 8-bit syndromes that are used to initialize the error pattern transform block. Since a four-stage pipelined design was used again, there needed to be four of these register blocks to allow for the appropriate amount of delay.

In addition, the MEA was only required to output the errata locator polynomial and not the errata magnitude polynomial. However, this change necessitated the need to normalize the output, which required the use of a GF division. The choice of divider was the same as that used in the Chien search block for the TDD in Section 4.1.

The quintessential difference between an FDD and a TDD is the structure of the output stage. In an FDD, the output block consists of an error transform block, an inverse transform block and a GF adder. The architecture of the error transform block is illustrated in Fig. 4-9.

Fig. 4-9: Error Transform Block

Clocking is performed by *symbol_clock* and the circuit requires a total of 255 cycles to complete. Error transform symbols are shifted out of the block to the inverse transform block immediately after the error symbols are calculated. This eliminated any need for storing the symbols, which would have increased the area requirement of the design. The first sixteen error patterns are the delayed syndromes, S_1 to S_{16} , from the first section of the decoder.

These syndromes are used in the error transform block for calculating the remaining 239 transforms. Hence, there are a total of 255 error transform symbols that are calculated in this block. All of the data buses and registers are 8-bits in width. The “ R_i ” data registers in Fig. 4-9 are the recursive accumulation registers, and register R_1 outputs the error transform patterns S_{17} to $S_{255(0)}$. In addition, “ σ_i ” registers contain the normalized coefficients of the errata locator polynomial from the MEA. Each σ_i register value remains the same for the duration of the current codeword. Following this unit is the inverse error transform block.

After finding the error pattern transforms, the inverse transform of the 255 error symbols must be found. The architecture for this operation is shown in Fig. 4-10. There are a total of

Fig. 4-10: Inverse Transform Block

510 8-bit registers, 255 8-bit GF adders and 255 8-bit GF constant multipliers. A significant reduction in area was gained by using hardwired constant multipliers adapted from [Solj1996].

The block is clocked by *symbol_clock* and required a total of 255 cycles to complete. It operated in parallel with the error transform block in order to meet timing. After receiving the last error pattern transform ($S_{255(0)}$) for a given codeword, the registers R_1 to R_{255} contain the decoded error symbols. Upon the next active edge of *symbol_clock*, the first error pattern transform (S_1) of the succeeding codeword is input and all the values in R_1 to R_{255} are shifted into registers e_1 to e_{255} simultaneously. These “e” registers are cascaded together and the decoded error symbols are shifted out of this block one symbol at a time (all from

e_{255}). These symbols are then GF added with the stored input symbols to produce the decoded output symbols.

An exhaustive attempt was made in order to somehow replace the 510 registers with a memory core cell. However, it was ascertained that this section can only be practically implemented with registers. Using memory prohibitively increased the complexity of the design. In addition, it was not at all feasible to use memory for this block because of the need to perform the required operations in parallel. A serial implementation would have been ridiculously slow, and thus the FDD would not have met the timing requirements of the design.

This RS FDD design was optimized as much as possible, such that a meaningful comparison with its TDD counterpart could be ascertained. The next section presents the VLSI implementation results from the FDD decoder design.

4.2.2 Implementation Results

The behavioural functionality of the FDD was verified through the use of MATLAB code, which can be found in Appendix B. Furthermore, the Verilog coding and testing methodology used for the TDD was again repeated for the FDD design.

Testing occurred at several stages of the design process to ensure a smooth design flow and successful functionality. This involved testing in the HDL coding stage, the Synopsys synthesis stage, and finally the post-layout fabrication-ready stage. In the end, the gate-level netlist from the final FDD layout was successfully verified to be functioning correctly. Since the FDD used a memory core, it too was not able to support scan-chains because of the presence of tri-state outputs.

The FDD RS decoder was implemented in a six-metal layer 0.18 μ m CMOS process technology. A standard-cell automated place-and-route was used to generate the layout in order to reduce design time. The pins on the chip are the same as those for the TDD, which were listed and explained in Table 4-2.

All the final results from post-layout simulations are summarized in Table 4-4. The power dissipation was provided by Synopsys, the timing information was obtained from Pearl and the silicon area from Cadence. This decoder chip was not fabricated.

Table 4-4: Reed-Solomon FDD VLSI Results

Parameter	FDD Chip Characteristics
Process	TSMC 0.18 μ m CMOS 6 layer-metal Virtual Silicon Library Cells [Virt2001]
Core Area	1.84mm x 1.64mm = 3.02mm ²
Total Area	2.11mm x 2.31mm = 4.87mm ²
Memory Block	One 32x256bit 6 Transistor embedded SRAM core
Standard Cells	22621
Registers	6387
Transistors^a	415,284 (340,002 without SRAM)
Supply Voltage	1.8V
Pin Count	44 (14 pwr/gnd, 30 I/O)
Packaging	44 CQFP
Speed	20.4 MHz symbol rate 163 Mbps bit rate (Fast clock = 81.6 MHz)
Power Consumption (est.)	58 mW @ 1.8V

a. The first number represents all the transistors used in the entire design, including the memory cores and I/O pads.

A checkplot of the FDD is illustrated in Fig. 4-11.

Fig. 4-11: FDD Pre-Fabrication Layout

4.2.3 Testing

The FDD contained the same memory core module as the TDD. Therefore, it was not possible to insert test scan-chains into the design. However, this was not as significant as for the TDD because the FDD was not fabricated. In any case, the same testing procedure that was used for the TDD was also used for completely verifying the functionality of the FDD. The following section discusses the results from the hardware implementation of both decoders.

4.3 Comparative Analysis of the TDD & FDD Implementations

The quintessential aspect of this thesis was to offer a complete comparison of the TDD and FDD hardware implementations. All of the results from each decoder were previously shown in Table 4-3 and Table 4-4. A synopsis of the results is presented in Table 4-5. A discussion of the main results now follows.

Table 4-5: Reed-Solomon Decoder Comparison

Parameter	TDD Chip Characteristics	FDD Chip Characteristics
Process	MOSIS 0.18 μ m CMOS 6 layer-metal Virtual Silicon Library Cells [Virt2001]	0.18 μ m CMOS 6 layer-metal Virtual Silicon Library Cells [Virt2001]
Core Area	1.44mm x 1.04mm = 1.50mm ²	1.84mm x 1.64mm = 3.02mm ²
Total Area	2.09mm x 1.69mm = 3.54mm ²	2.11mm x 2.31mm = 4.87mm ²
Memory Block	Two 32x256bit 6 Transistor embedded SRAM cores	One 32x256bit 6 Transistor embedded SRAM core
Registers	2206	6387
Transistors^a	305,538 (154,974 without SRAM)	415,284 (340,002 without SRAM)
Pin Count	44 (14 pwr/gnd, 30 I/O)	44 (14 pwr/gnd, 30 I/O)
Packaging	44 CQFP	44 CQFP
Speed	22.0 MHz symbol rate 176 Mbps bit rate	20.4 MHz symbol rate 163 Mbps bit rate
Tested Speed (20MHz Tester)	22MHz symbol rate 160 Mbps	Not Fabricated
Power Consumption (1.8V)	56 mW (estimated) 58 mW (tested)	58 mW (estimated)

a. The first number represents all the transistors used in the design, including the memory cores and I/O pads.

Throughout the design of both the FDD and the TDD, several issues became apparent. First, in previously reported implementations the FDD was extolled as: (i) being simple to implement [Shao1985], and (ii) having a low complexity as compared to the TDD [HsTr1988]. It was shown that the most salient difference between the two designs was in the output stage. All sections before the output stage are thus essentially the same in terms of complexity. However, the most challenging component of the design of RS decoders was

found to be the actual decoding algorithm, which in this case is the MEA. Both decoders required the MEA block, so each one is equally as complex up to that point of the output stage. The complexity of the output stages of the decoders is comparable as well. Architectures used for the Chien search block in the TDD facilitate the design of the output block. The output stage of the FDD is trivial to implement, but this simplicity is offset by the high area requirement. Nevertheless, both the TDD and the FDD have a regular and well structured output stage. The control logic was ascertained to be comparable as well. Therefore, each of the decoders can be said to have the same degree of design complexity. This is because both approaches contained the MEA, which was found to be the most difficult RS decoder block to design efficiently.

In addition, it was discovered that the structure of the TDD facilitated the detection of decoding failures. The structure of the Chien search block allowed the TDD to implement more criteria for determining if the decoded output codeword sequence was incorrect. This could be done in real-time. So when the first symbol was output, the *decoder_failure* pin indicated if the current upcoming sequence of codeword symbols was successfully decoded. However, the output structure of the FDD complicated this procedure. Unfortunately, an efficient and reliable method for indicating a decoding failure for the FDD could not be found in the published literature. Hence, the reliability of the decoding failure output signal is significantly higher in the TDD.

Finally, the VLSI results provided in Table 4-5 indicate that the performance of the decoders were for the most part quite comparable. Both decoders were designed using the same 0.18 μ m CMOS standard-cell library components and they had the same operating voltage of 1.8V. A low overall power consumption was achieved with both designs, but nonetheless the TDD was still lower in this aspect of the design. The slightly lower power requirement of the TDD is believed to be the result of using memory blocks instead of register blocks where possible.

Both decoders used a memory block for storing the input codeword symbols. The TDD used an additional memory block to store the results from the Chien search algorithm. This was done in order to check if there had been a decoder failure. However, this was not required in the FDD since the inverse transform block immediately required the data from the remaining error transform block. Using a memory block for the inverse transform block was not feasible either.

Each decoder was designed to operate at approximately the same speed so that a meaningful comparison could be made. However, the TDD still proved to be slightly faster than the FDD.

The TDD and FDD had worst case symbol rates of 22.0 MHz and 20.4 MHz, respectively. Each symbol was 8-bits, which translated into a TDD bit-rate of 176 Mbps and an FDD bit-rate of 163 Mbps. The additional latency in the FDD can be contributed to the long computation times required by its output stage.

Nevertheless, the most conspicuous difference between the decoders is the significantly higher area required by the FDD implementation. A thorough undertaking to optimize the FDD in terms of area was attempted. However, even the most minimal area FDD design was precipitously worse than its TDD counterpart. In fact, the core area of the FDD was more than double that of the TDD. This was due to the hardware requirements of the output stages. In the non-optimized design, the FDD needed $2n$ more registers, $2n-3k-1$ less GF adders, $n-k$ less GF multipliers, and n more constant multipliers than the TDD. The extra registers and constant multipliers in the FDD increase the area substantially. Thus, there is a significant area advantage in terms of implementing a TDD over an FDD as was foreshadowed by the data in Table 3-1 and Table 4-1. A way to further optimize the output stage of the FDD beyond [Shao1985] and [HsTr1988] could not be found, such that it would more suitably be for VLSI design.

An RS encoder was not considered for this thesis because the encoding of RS codes is relatively simple. The fundamental arithmetic operation is a polynomial modulo operation which can be efficiently implemented by a LFSR [Jeon1995]. In addition, the designs in this thesis targeted a (255,239) RS code because of the numerous applications that used this code. However, it is believed that the results from this thesis, in terms of the TDD versus the FDD, could apply to RS codes with smaller n and k values. Different architectures would nevertheless have to be used for some of the blocks, including the multipliers, if the bit size of the symbols were different than the current 8-bits. Nonetheless, judging by the applications listed in Section 2.1 this should not be an issue. The majority of the standards for RS codes that have been released in the past few years all seem to be based on 8-bit codeword symbols. Most of the optimizations in the literature also target this codeword size. A use for RS codes larger than (255,239) could not be ascertained from examining the code structure of any practical applications of RS codes.

CMOS process technologies are continually improving and new design issues would have to be considered for a 0.13 μ m CMOS process. Time sharing certain blocks of hardware could further reduce the die area, and still meet the specifications of the target applications if the designs were to be implemented in a 0.13 μ m CMOS process. In addition, the design in this thesis could have been improved through the use of resource sharing in Synopsys. This

technique provided a 5.4% reduction in area in the same TDD design used in this thesis. Hence, possible further reductions in area and power could have been achieved if this optimization was used at the beginning of the design process.

This design is for the most part operating at its maximum bit rate. Pearl was used to obtain the critical path timing information for both designs. The critical path for the TDD had a total delay of 28.71 ns. It occurred in the recursive MEA block shown in Fig. 3-7 from Chapter 3. Conversely, the critical path for the FDD had a total delay of 31.29 ns. It occurred in the polynomial expansion unit illustrated in Fig. 4-6. There was a control signal called *GO*, which had to go to several registers. This signal was responsible for causing the critical path delay in the FDD.

Therefore, it might be possible through the synthesis process to decrease the speed of the decoder to try and reduce the die area slightly. Nevertheless, the majority of the design emphasis was on the minimization of die area and power reduction. However, if future applications require more speed, then the design would need to have more parallelism. This would require different architectures for some of the blocks that minimized the area of the design at the expense of speed.

In short, in terms of VLSI hardware the TDD definitely has a more efficient and cost effective architecture than the FDD. Each has a comparable degree of design complexity. However, the TDD was superior in terms of area, speed, power consumption and decoding failure reliability.

Referring back to Chapter 2, Table 2-3 presented an RS decoder implementation literature summary. Previous implementations of RS decoders were discussed in terms of several performance parameters including area, power and speed. The results from this thesis can be added to the list and are summarized in Table 4-6. Some of the designs in Table 2-3 do present an implementation with better normalized area. However, some publications used full-custom designs, they did not incorporate erasures and they did not accommodate as high of an RS (n,k) code as in the synthesizable designs of this thesis. In addition, the transistor count is lower in a few designs, but these publications do not include memory blocks in the transistor count and they do not accommodate as high of an RS (n,k) code as the designs in this thesis.

Table 4-6: RS Decoder Thesis Summary for Table 2-3

Parameter	TDD from this Thesis	Previous Best MEA [Wilh1999]	Previous Best BM Algorithm [Chan2001]
RS Code Parameters	Erasures ($t = 8$) (255,239)	Erasures ($t = 10$) (167,147)	(208,192) & (182,172) $t = 8$
Power	58 mW @1.8V	N/A	102mW @ 3V
Speed	160 Mbps 20 MHz symbol_clock	256 Mbps	33 MHz clock 264 Mbps
Transistors	154,974	73,000	~ 104,000
Design	Synthesized from Standard-cells	Full-custom	Full-custom
Line Width	0.18 μ m 6LM	0.5 μ m LM N/A	0.6 μ m SPDM
Core Area^a	1.50mm ²	0.612mm ²	1.38mm ²

a. Normalized to 0.18 μ m CMOS technology.

The results in Table 4-6 can be explained as follows. Previous best RS decoder implementations from [Wilh1999] and [Chan2001] show impressive results in several categories. There are several reasons as to why these two designs achieved these results. First, both designs were full-custom as compared with the TDD design in this thesis, which was synthesized from standard cells. Although the area and power can be reduced by using a full-custom layout, this process is extremely complex and time intensive. Second, both designs were based on previous work by the authors and the implementations took several years to be developed into their final results. The TDD in this thesis did not have the luxury of this extended design period.

Thus far, ambiguity about whether to implement the MEA or the BM algorithm has been an issue for VLSI designers of RS decoders. The ensuing discussion attempts to clarify this opaqueness. Until the work of [Chan2001], most practical VLSI decoder implementations used the MEA approach because there had been no efficient implementation of the BM algorithm. Previous BM approaches used significantly more multipliers than the MEA. For example, implementations in [Liu1984] and [Oh1996] required approximately $2t \sim 3t$

multipliers for the BM algorithm decoding, where t is the number of correctable errors. However, [Chan2001] developed an iterative time sharing algorithm which essentially made the VLSI implementation of the BM algorithm comparable to the MEA. The design reduced the number of finite field multipliers from $2t \sim 3t$ to three multipliers in the decoding algorithm, which is comparable to the four used in the standard MEA.

In addition, although the BM implementation in [Chan2001] is attractive in terms of its performance, it does have a drawback because there is a patent on the design. Conversely, the patent on the MEA has expired and can now be freely implemented by all designers.

Furthermore, two independent perspectives corroborate the use of either algorithm. An algorithmic comparison was presented in [Saka1995]. The conclusion was that both methods yield distinct but similar parallel architectures with the same optimal complexity. Then in [Chi1993], the two algorithms were compared in terms of machine cycles. It was ascertained that both algorithms required the exact same amount of cycles to complete for the syndrome calculation, key equation, Chien search, error value evaluation and overall decoding.

Finally, the memory requirements can be said to be equivalent for both algorithms. Each one requires the storage of the input codewords, which will then be used in their respective output stages to form the decoded codewords. Therefore, these two algorithms can be said to be equivalent, and the choice to implement one versus the other becomes a matter of personal preference for the RS decoder designer.

4.4 Summary

The contributions of this chapter are as follows.

- An optimized Reed-Solomon TDD and FDD were constructed using a combination of area-efficient and low power VLSI architectures.
- Two RS decoders were designed and built in a $0.18\mu\text{m}$ CMOS standard-cell technology. The TDD was fabricated.
- An FDD and a TDD were analyzed and compared in terms of their respected VLSI hardware implementations with the following results:
 - The decoding algorithm, in this case the MEA, is the largest contributor to design complexity. Each decoder is essentially identical up to and including the MEA. Therefore, the decoders are comparable in design and implementation complexity.

- The TDD was proven to be superior to the FDD because the TDD chip had a smaller area, faster decoding speed, lower power consumption and a more reliable decoder failure indicator.
- The prohibitive area of the FDD was its most prominent limitation to practical applications.
- The BM algorithm and the MEA are essentially identical, and the choice as to which one should be implemented for an RS decoder is arbitrary.

The next chapter presents the results from the C code implementation of both RS decoders for use in embedded software applications.

Software Implementation

The previous chapter discussed the hardware implementations of the TDD and the FDD. Chapter 5 now presents the results obtained from implementing each decoder in software using C code. The purpose of this section is to present a straightforward software comparison of the two decoders for the sake of thoroughness. This approach was intended to determine which decoder would be more suitable in a system containing an embedded processor core that is part of a larger SOC implementation.

There were several important parameters from the software implementation that needed to be obtained for comparing the two decoders. The executable size of the program and static memory allocation were of special interest to this thesis. Processor cores are small and the memory area dominates the overall die area in SOC implementations. For the sake of design expediency, no dynamic memory allocation was used in this implementation of RS decoders. Hence there was a slight loss in efficiency, but the reduction in design complexity offset this loss. In addition, the various computing times required by each decoder block were compared. This information would determine the specific blocks that had the highest computational requirements. Finally, several compiler optimization options were used to obtain the most efficient RS decoder software implementation.

5.1 System Specifications

The functionality of the (255,239) RS codes for both the TDD and FDD were verified through exhaustive simulations. Several hundred thousand test codewords and various error and erasure combinations were used in these simulations for both decoders.

All of the tests were performed on the same computer system in order to obtain consistent results. Simulations were run on a system [Ultr1997] with the following specifications:

- System Model: Sun Ultra 80.
- CPU: four Sun UltraSPARC-II 450 MHz CPUs.
- CPU Architecture: 64-bit SPARC-V9 reduced instruction set computer (RISC) architecture with Virtual Instruction Set (VIS). 9 stage pipeline. On-chip instruction cache and data cache of 16kB each. Software data prefetch. 4 integer execution units. 3 floating-point execution units.
- CPU Performance (est): 19.6 (SPECint95) and 27.1 (SPECfp95) @ 450MHz.
- Memory: 4.0 GB main memory and 7.2 GB virtual memory.
- Operating System (OS): SunOS v5.8 with sun4u architecture, Solaris 8 SPARC.
- C Compiler: UNIX gcc Compiler version 2.95.2.

5.2 Time-Domain Decoding Results

The initial verification of the Reed-Solomon decoding algorithms was performed using MATLAB. Therefore, it was relatively straight-forward to translate this software into a C code based implementation. The TDD design followed the algorithmic design presented in Chapter 3. Results from the TDD software are presented in Table 5-1. For each of the test

Table 5-1: TDD Software Implementation Results

		Optimization Method									
		1	2	3	4	5	6	7	8	9	10
Reed-Solomon Decoder Block	Syndrome (μ s)	353	130	148	131	97	119	128	110	103	128
	Erasure Handling (μ s)	13	5	5	3	4	2	6	2	6	6
	Euclidean Algorithm (μ s)	117	25	28	49	17	29	43	29	27	4
	Chien Search (μ s)	1105	162	123	484	155	154	485	179	186	453
	GF_mul (μ s)	1935	958	874	313	702	970	319	682	1324	324

Table 5-1: TDD Software Implementation Results

		Optimization Method									
		1	2	3	4	5	6	7	8	9	10
	internal_mcount (μ s)	2547	1552	1968	343	1271	1634	327	1240	1842	359
	Total (ms)	6.724	2.923	3.201	1.363	2.278	2.961	1.332	2.301	3.579	1.333
	exe Size (kb)	18.00	12.96	12.64	14.09	19.78	18.42	19.00	20.57	20.08	23.41

runs, there were 10 000 vectors simulated that each had six errors and four erasures. This large number of test vectors was required in order to get an accurate time reading. A UNIX profiling program called *gprof* was used to gather the timing information. The program provided the length of time that was required for each function or block. Then this time was divided by 10 000 to get an approximation for the decoding time for one codeword. The numbers stated in the table are normalized for the decoding of **one** codeword. A sample of the *gprof* profiling session of a code is provided in Appendix C.

There were ten optimization strategies used for both the TDD and the FDD software decoders. For a detailed explanation of each of the strategies, the reader is urged to reference the man pages of the C compiler listed in Section 5.1. The optimizations are summarized below in Table 5-2.

Table 5-2: Compiler Optimization Methods

Optimization Method in Table 5-1 & Table 5-4	Optimization Explanation
1	No compiler optimization performed.
2	Optimization Level 1: The compiler tries to reduce code size and execution time.
3	Optimization Level 2: Nearly all supported optimizations that do not involve a space-speed trade-off are performed.
4	Optimization Level 3: Optimize even more. This turns on everything the previous level does, along with also turning on inline functions.

Table 5-2: Compiler Optimization Methods

Optimization Method in Table 5-1 & Table 5-4	Optimization Explanation
5	Same as (2) except loop unrolling is performed. This is only done for loops whose number of iterations can be determined at compile or run time.
6	Same as (3) except loop unrolling is performed. This is only done for loops whose number of iterations can be determined at compile or run time.
7	Same as (4) except loop unrolling is performed. This is only done for loops whose number of iterations can be determined at compile or run time.
8	Same as (2) except loop unrolling is performed. This is done for ALL loops.
9	Same as (3) except loop unrolling is performed. This is done for ALL loops.
10	Same as (4) except loop unrolling is performed. This is done for ALL loops.

Timing information that is provided for the GF_mul function in Table 5-1 is for the total number of multiplication function calls performed in the decoder for a codeword. The total decoding time included internal memory count time, which was significant in most cases, and a few other miscellaneous blocks which did not contribute a readable time to the overall decoding time. For a simulation of 10 000 vectors, the total time that was provided for these blocks was zero. In addition, the executable size included the encoder and other blocks that were required for implementing GF functions. However, these blocks did not contribute a significant amount of overhead (< 1%).

Static memory allocation for each decoder block module was determined and the results are provided in Table 5-3. The table lists the total number of bytes and the percentage of the total memory block required to store the variables for each block. Using an embedded SRAM core block, 1719 bytes would require a memory size of 512 x 32 bits which would occupy a

total area of 0.319 mm^2 . However, due to the available block sizes, there would be some extra memory space that would not be used.

Table 5-3: TDD Software Static Memory Usage Results

Reed-Solomon Decoder Block	Memory (bytes)	Embedded SRAM Core (mm^2)
Syndrome Calculation Block	408	23.7%
Erase Handling	152	8.8%
GF Multiplier Function	16	0.9%
Modified Euclidean Algorithm	620	36.1%
GF Inverse Function	16	0.9%
Chien Search and Output	507	29.5%
Total	1719	0.319

The decoder block with the largest memory requirement was the modified Euclidean algorithm block, which occupied approximately 36% of the total static memory allocation. However, the output stage that included the Chien search block also required a significant amount of memory. The above results may vary after optimization and program execution. They are simply provided as a rough estimate for memory usage for the decoder. It nevertheless still serves to offer a comparison of the two decoders. All of the results from the RS TDD will be fully discussed following the presentation of the FDD software implementation results in the next section.

5.3 Frequency-Domain Decoding Results

The code used for the FDD was similar to that used for the TDD. Analogous to the hardware approach, the quintessential difference between the two decoders was in the output stage following the MEA block. Again, the FDD was first designed in MATLAB and then translated into C for an embedded software application. The FDD design followed the algorithmic design presented in Chapter 3. Results from the FDD software are presented in

Table 5-4. To be consistent with the TDD, each of the test runs simulated 10 000 vectors,

Table 5-4: FDD Software Implementation Results

		Optimization Method									
		1	2	3	4	5	6	7	8	9	10
Reed-Solomon Decoder Block	Syndrome (μ s)	403	115	124	165	114	125	126	107	105	107
	Erasure Handling (μ s)	17	6	9	2	3	4	5	6	9	5
	Euclidean Algorithm (μ s)	114	38	36	42	35	26	32	35	43	45
	Error Transform (μ s)	484	154	156	151	130	126	122	110	129	128
	Inverse Transform (μ s)	5859	2056	2171	2407	1947	2369	2257	1966	2181	2231
	GF_mul (μ s)	6299	6181	6520	6132	42	6180	5989	31	6604	6253
	internal_mcount (μ s)	12562	6082	6376	5978	67	6043	6031	81	6559	6124
	Total (ms)	32.24	15.16	15.88	15.45	2.350	15.29	15.09	2.354	16.19	15.29
	exe Size (kb)	18.22	13.23	12.96	13.88	19.77	18.35	19.00	20.81	20.65	23.08

which each had six errors and four erasures. This large number of test vectors was required in order to get an accurate time reading. Timing was calculated using the same profiling program *gprof*, and the same method as for the TDD. Hence, the numbers stated in the preceding table are normalized for the decoding of **one** codeword. In addition, compiler optimization strategies were used and their explanation was provided in Table 5-2. A sample of a FDD *gprof* profiling session is provided in Appendix C.

Static memory allocation for each FDD decoder block module was determined and the results are provided in Table 5-5. The table lists the total number of bytes and the

Table 5-5: FDD Software Static Memory Usage Results

Reed-Solomon Decoder Block	Memory (bytes)	Embedded SRAM Core (mm²)
Syndrome Calculation Block	408	22.1%
Erase Handling	152	8.3%
GF Multiplier Function	16	0.9%
Modified Euclidean Algorithm	620	33.7%
GF Inverse Function	16	0.9%
Error Transform Pattern	355	19.3%
Inverse Error Transforms	275	14.9%
Total	1842	0.319

percentage of the total memory block required to store the variables for each block. Using an embedded SRAM core, 1842 bytes would require a 512 x 32 bit memory with a total area of 0.319 mm². However, due to the available block sizes, there would be some extra memory space that would not be used. The decoder block with the single largest memory requirement was the modified Euclidean algorithm block, which occupied approximately 33.7% of the total memory area. However, if the two blocks of the output stage are combined, then they required the most memory allocation. The output stage occupied approximately 34% of the allocated memory. All of the results from the RS FDD will be fully discussed and a comparison to the TDD will be made in the following section.

5.4 Comparative Analysis and Summary

The purpose of implementing both decoders in software was to determine which approach would be superior for an embedded software SOC application. This section compares the results of the software parameters from the TDD and FDD.

To begin, the TDD required less static memory allocation than the FDD. Both decoders had the same memory requirements up to and including the MEA. The only difference was the output stage. Looking at the results in Table 5-3 and Table 5-5, it is shown that the FDD requires 123 bytes more memory than the TDD. This difference represents 6.7% of the 1842

total bytes required by the FDD. The variance is entirely due to the large memory required for the error transform output stage of the FDD. In fact, this was the same issue that plagued the FDD hardware implementation. Hence, it can be said that the TDD is more efficient in terms of memory usage.

Executable size of the entire Reed-Solomon program was compared. Again, the TDD was vastly superior to the FDD. To elaborate, each decoder was able to achieve a minimum executable size of 12.96kbytes through the use of various compiler optimization options. However, the decoding times for the TDD and the FDD were 2.923ms and 15.88ms, respectively. For the same area, the FDD was over five times as slow as the TDD. The source of the rather large latency in the FDD was the inverse transform block. It becomes unquestionably clear then, that the TDD is the decoder of choice for trying to achieve a more compact and thus cost-effective embedded software implementation.

Finally, the decoding times of each of the decoders were compared. Once again the FDD was proved to be inferior to the TDD. For most of the results, the FDD performed dramatically worse than the TDD. However by unrolling loops in the code, the compiler was able to devise a relatively fast implementation of the FDD. Nonetheless, the fastest FDD at 2.35ms was still twice as slow when compared to the fastest TDD at 1.332ms. These times correspond to decoding speeds of 868kpbs for the FDD and 1.53Mbps for the TDD on the given CPU. The TDD even had a smaller executable size than the FDD at these speeds. Table 5-1 and Table 5-4 show that the executable sizes for the faster TDD and FDD are 19.00kbytes and 19.77kbytes, respectively. In short, the TDD is undoubtedly more efficient for SOC applications in terms of minimizing executable size.

The most significant problem with the FDD approach, that caused such a precipitous degradation in performance, was the inverse transform block. It was computationally intensive and required significant memory allocation. As an analogy, the inverse transform block can be considered as being the Achilles heel of frequency-domain RS decoding. There have been essentially no proposals in the literature to ameliorate this issue. It seems then that the algorithmic structure of the output block makes it difficult to design a practical implementation of the FDD.

In terms of design complexity, the two decoders were found to be almost equivalent. The design of the decoders is essentially the same up to their respective output stages. Nevertheless, the most complicated aspect of decoding is the MEA decoding algorithm, which is present in both implementations. The output stage of the FDD is slightly more

intuitive to understand, but its extremely poor performance certainly overshadows this simplicity. To conclude, Table 5-5 summarizes the software results from this chapter.

Table 5-6: Summary of Software Implementation of RS Decoders

Software Parameter	TDD Results	FDD Results	Relative Performance
Static Memory Allocation	1719 bytes	1842 bytes	TDD requires 6.7% less memory
Decoding Speed	1.53 Mbps with 19 kbyte exe. size	868 kbps with 19.77 kbyte exe. size	TDD 1.76x Speed-up with 3.9% less exe.
Executable Size	12.96 kbytes (2.92ms decode time)	12.96 kbytes (15.88ms decode time)	Same min. exe. size TDD 5.44x Speed-up
SRAM Area	0.319mm ²	0.319mm ²	TDD used 123 fewer bytes than the FDD

The contributions of this chapter are as follows:

- A Reed-Solomon TDD and FDD were designed in software using C code.
- An FDD and a TDD were analyzed and compared in terms of their respected software implementations with the following results:
 - The MEA decoding algorithm is the largest contributor to design complexity. Each decoder is essentially identical up to and including the MEA. However, the FDD's output stage is somewhat more intuitive to understand, which makes its design complexity modestly simpler than the TDD in software.
 - The TDD was proven to be superior to the FDD in terms of fastest decoding speed, minimum executable size, and lowest memory requirement.
 - The prohibitive component of the FDD that makes it inferior to the TDD is the inverse transform block in the output stage.
 - Time-domain decoding is the clear implementation choice for embedded software applications. The results from this chapter prove that the FDD is inferior to the TDD in terms of software design.
- Optimizing compiler technology can make a significant difference in decoding speed and executable size.
- The CPU core, being the compiler target, is just as important. The numbers obtained in Table 5-6 are only applicable to the SUN CPU specified in this thesis. Typically, an

ARM7 or ARM9 CPU would be used and the exact same numbers cannot be expected for either of these implementations.

The concluding chapter is presented next.

Conclusions

6.1 Summary

This thesis examined Reed-Solomon time-domain and frequency-domain decoder implementations in software and hardware. The focus was on designing an area-efficient, low-power and low-complexity RS decoder.

Chapter 2 discussed the basics of error correction codes, including Reed-Solomon codes. The prolific use of RS decoders was confirmed by presenting some of the applications that use RS codes. In addition, decoding algorithms were presented and the reasoning of why the MEA was chosen for this thesis was defended. Finally, an extensive review of the literature pertaining to RS codes, decoders, and mathematics was presented. It showed that there are no comparisons of VLSI implementations of RS TDDs and FDDs that provide a definitive encompassing statement about the merits of either decoder in VLSI. This served as the primary motivation for the work presented in this thesis.

In chapter 3, the theory and arithmetic behind the design of an RS decoder was presented. All of the required equations for decoding in the time-domain or the frequency-domain were explained. In addition, each decoder was partitioned into its constituent blocks. Each block was then fully explained and block diagrams were provided to illustrate how to realize various equations in hardware. It was shown that both decoders were for the most part quite similar in architecture. However, the quintessential difference between the two was in the structure of the output stage.

Chapter 4 presented the hardware implementation of the RS decoders. The TDD was presented first and its architectural modifications were discussed. Various architectures were combined in order to achieve the most optimal design in terms of area and power consumption. It was shown that the design complexity was reduced significantly from previous approaches. The results from testing the final TDD ASIC design were also presented. Subsequently, the FDD design was discussed and its results were illustrated. Most important, the VLSI results from the two decoder implementations were compared and contrasted. This showed that the decoders are similar in terms of design complexity. The TDD was slightly better in terms of power consumption and decoding speed. Nonetheless, the FDD was found to be drastically inferior to the TDD in terms of area usage. The FDD had double the area compared to the TDD. Actual VLSI results corroborated the definitive statements made in this chapter.

In chapter 5, the software implementation of the two RS decoders was documented. The purpose of comparing the two decoders in software was to ascertain which approach would be better suited for use in a SOC realization. Hence, the areas that were examined were the executable size, static memory allocation and decoding speed. In all areas, the TDD performed significantly better than the FDD. The only advantage to using an FDD was that its output stage was more intuitive to understand. Nonetheless, this was not the most complex block in the decoder. In both cases, the block with the highest design complexity was the MEA. Finally, the degradation in performance of the FDD was attributed to the inefficiency of the inverse transform block.

6.2 Conclusions

Reed-Solomon decoders were shown to exist in a multitude of practical applications, thus establishing the need for performing further research in this area. Thus far, only ambiguous and uncorroborated statements about RS decoder implementations have been made. Virtually no remarks have been made on deep submicron (DSM) implementations synthesized from HDL. Based on the results from this thesis, the following definitive statements serve to clarify this ambiguity.

Time-domain RS decoding is a better approach for VLSI implementations than frequency-domain decoding. The FDD has double the silicon area for a slower decoding speed than the TDD. Power consumption is shown to be quite similar, with the TDD requiring slightly less power than the FDD. Moreover, the two decoders are comparable in terms of design complexity. Each decoder is essentially the same up to and including the decoding algorithm, which was the most complicated block of the decoder. Hence based on the VLSI results, it is

said with unequivocal confidence that the TDD is superior to the FDD for hardware applications.

In terms of software, the TDD is superior to the FDD. The TDD required less memory, had faster decoding speeds and had smaller executable sizes than the FDD. However, the only merit of the FDD was that its output stage was more intuitive to understand. It was ascertained that the detrimental factor of the FDD was the inverse transform block of the output stage. This block clearly degraded the FDD's performance such that it made the TDD the only feasible choice for software-based applications.

6.3 Contributions of this Thesis

The contributions of this thesis were:

- The presentation of a review of RS decoder literature, which up to this point has been ambiguous about which decoder approach (TDD or FDD) is superior.
- A low-power, low-complexity and area optimized Reed-Solomon decoder architecture for both a time-domain and a frequency-domain implementation.
- A 0.18 μ m ASIC implementation of the optimized RS TDD synthesized from a Verilog HDL description.
- Definitive statements comparing the TDD and FDD in both hardware and software.

6.4 Future Research

The search for more efficient algorithms, faster design times, and better implementations is a continual process. Whether it be from market demands or personal ambition, improvements over previous approaches will always be attempted. The field of Reed-Solomon decoder VLSI design is no exception. Hence, there are still several unanswered questions that remain and are thus open for future research.

6.4.1 Reed-Solomon Decoding Algorithms

Although hard-decision decoders dominate VLSI RS error correction applications, there are theoretically more efficient and powerful decoders. These are the algebraic soft-decision decoders [Koet2000] that are able to provide a lower BER at a given SNR. However, their complexity makes practical VLSI implementation prohibitively difficult at this time. Research into deriving algorithms that are suitable for VLSI design is currently under way.

6.4.2 ASIC Design Methodology and Flow

ASIC design methodology and flow becomes an integral issue with increasing time-to-market demands. Moore's Law states that approximately every eighteen to twenty-four months the number of transistors per unit of die area doubles, so new design flows must be developed and utilized quickly. The learning curve is precipitous and this considerably increases the design time. Moreover, bugs in the design flow cause a greater effort to be placed on debugging the vendor tools rather than on the design itself.

In an academic setting, it would be beneficial to have ASIC designers work in parallel while proceeding through the new design flow. This way, these researchers could collaborate and discuss design flow issues when they occur. Design time could be reduced and the focus of academic ASIC design could be placed back on the design and not on constructing a design flow.

6.4.3 Galois Field Architecture Comparisons

When designing an ASIC at the RTL design stage, it can be quite difficult to be able to predict design parameters such as proposed area, timing and power consumption. Therefore, it would be interesting if a chart of data could be constructed that provided preliminary information about these design parameters. A designer could simply count up how many registers, memory cells, and GF arithmetic structures there were in the design to get a lower bound estimate of the chip's performance.

Constructing this chart of data for GF arithmetic structures would be straightforward. Variable bit multipliers, adders and inverters would be synthesized from their respected HDL codes. Then each component would be placed, routed and verified. At that point, numbers for power consumption, timing and area could be ascertained. Finally, the designer could reference these numbers and estimate the ASIC's performance based on the number of components in the design.

References

- [AHA2000] Advanced Hardware Architectures (AHA) Inc., "http://www.aha.com/Publications/pb4013b_0600.pdf", 2000.
- [Berl1968] E. R. Berlekamp, *Algebraic Coding Theory*, New York: McGraw-Hill, 1968. (Revised edition, Laguna Hills: Aegean Park Press, 1984).
- [Berl1986] E. R. Berlekamp, and L. R. Welch, "Error Correction for Algebraic Block Codes," *U.S. Patent US4633470*, 1986.
- [Berl1987] E. R. Berlekamp, R. E. Peile, and S. P. Pope, "The Application of Error Control to Communication," *IEEE Comm. Magazine*, vol. 25, pp. 44-57, 1987.
- [Berl1989] E. R. Berlekamp, "Soft Decision Reed-Solomon Decoder," *U.S. Patent US4821268*, 1989.
- [Blah1979] R. E. Blahut, "Transform Techniques for Error Control Codes," *IBM Journal of Research and Development*, vol. 23, pp. 299-315, 1979.
- [Burt1971] H. O. Burton, "Inversionless Decoding of Binary BCH Codes," *IEEE Trans.*, vol. IT-71, pp. 464-466, 1971.
- [Chan1998] H. -C. Chang, and C. B. Shung, "A(208,192;8) Reed-Solomon Decoder for DVD Application," *IEEE Int. Conference on Communications*, vol. 2, pp. 957-960, June 1998.
- [ChSh1999] H. -C. Chang, and C. B. Shung, "New Serial Architecture for the Berlekamp-Massey Algorithm," *IEEE Trans. on Communications*, vol. 47, issue 4, pp. 481-483, April 1999.
- [Chan1999] H. Chang, and M. H. Sunwoo, "A Low-Complexity Reed-Solomon Architecture Using the Euclid's Algorithm," *IEEE Proc. of the International Symposium on Circuits and Systems, ISCAS '99.*, vol. 1, pp. 513-516, May 1999.
- [ChSu1999] H. Chang, and M. H. Sunwoo, "Design of an Area Efficient Reed-Solomon Decoder ASIC Chip", *IEEE Workshop on Signal Processing Systems, 1999. SiPS 99.*, pp. 578-585, Oct. 1999.
- [Chan2001] H. -C. Chang, C. B. Shung, and C. -Y. Lee, "A Reed-Solomon Product-Code (RS-PC) Decoder Chip for DVD Applications," *IEEE Journal of Solid-State Circuits*, vol. 36, issue 2, pp. 229-238, Feb. 2001.

- [Chen1995] H. -W. Chen, J. -C. Wu, G. -S. Huang, J. -C. Lee, and S. -S. Chang, "A New VLSI Architecture of Reed-Solomon Decoder with Erasure Function," *IEEE Global Telecommunications Conference, GLOBECOM '95.*, vol. 2, pp. 1455-1459, Nov. 1995.
- [Chi1993] D. T. Chi, "A New Fast Reed-Solomon Decoding Algorithm Without Chien Search," *IEEE Conference Record of Communications on the Move*, vol. 3, pp. 948-952, Oct. 1993.
- [Chie1964] R. T. Chien, "Cyclic Decoding Procedure for the Bose-Chaudhuri-Hocquenghem Codes," *IEEE Trans. on Information Theory*, vol. IT-10, pp. 357-363, Oct. 1964.
- [Choo1992] S. Choomchuay, and B. Arambepola, "An Algorithm and a VLSI Architecture for Reed-Solomon Decoding," *IEEE Proc. of the International Symposium on Circuits and Systems, ISCAS '92.*, vol. 5, pp. 2120-2123, May 1992.
- [Coop2000] I. R. Cooper, and M. A. Bramhall, "ATM Passive Optical Networks and Integrated VDSL," *IEEE Comm. Magazine*, vol. 38, issue 3, pp. 174-179, March 2000.
- [Core2000] CorePool, "http://www.corepool.com/products/fhg_rs_dec.htm", 2000.
- [Cost1998] D. J. Costello Jr., J. Hagenauer, H. Imai., and S.B. Wicker, "Applications of Error-Control Coding," *IEEE Trans. on Inform. Theory*, vol. 44 no. 6, Oct. 1998, pp. 2531-2560.
- [Dabi1995] D. Dabiri, and I. F. Blake, "Fast Parallel Algorithms for Decoding Reed-Solomon Codes Based on Remainder Polynomials," *IEEE Trans. on Information Theory*, vol. 41, no. 4, pp. 873-875, July 1995.
- [Dutt1999] A. Dutta-Roy, "Networks for Homes", *IEEE Spectrum*, vol. 26, issue 9, pp. 26-33, Dec. 1999.
- [eMDT2000] e-MDT Inc. , "<http://www.e-mdt.com/communication.htm>", 2000.
- [ETSI2001] ETSI standard, "Transmission and Multiplexing (TM); Access transmission systems on metallic access cables; Very high speed Digital Subscriber Line (VDSL); Part 2: Transceiver specification," ETSI TS 101 270-2 V1.1.1, 2001.
- [Fitz1998] P. Fitzpatrick, and S. M. Jennings, "Comparison of Two Algorithms for Decoding Alternant Codes," *Applicable Algebra in Eng., Commun., and Computing*, vol. 9, pp. 211-220, 1998.
- [Forn1965] G. D. Forney, "On Decoding BCH Codes," *IEEE Trans. on Information Theory*, vol. IT-11, pp. 549-557, Oct. 1965.
- [4i2i2000] 4i2i Communications Ltd., http://www.4i2i.com/reed_solomon_codes.htm, 2000.
- [Gore1961] D. Gorenstein, and N. Zierler, "A Class of Error Correcting Codes in p^m Symbols," *Journal of the Society of Industrial and Applied Mathematics*, vol 9., pp. 207-214, June 1961.

- [Guo1998] J. -H. Guo, and C. -L. Wang, "Hardware-efficient Systolic Architecture for Inversion and Division in $GF(2^m)$," *IEE Proc. on Computers and Digital Techniques*, vol. 145, no. 4, pp. 272-278, July 1998.
- [HsTr1988] I. S. Hsu, T. K. Truong, L. J. Deutsch, E. H. Satorious, and I. S. Reed, "A Comparison of VLSI Architectures for Time and Transform Domain Decoding of Reed-Solomon Codes," Jet Propulsion Laboratory, Pasadena, CA, TDA Progress Rep. 42-92, Oct-Dec. 1989, pp. 63-81, Jan 1988.
- [HsDe1988] 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 Basis," *IEEE Trans. on Computers*, vol. 37, no. 6, June 1988.
- [Hsu1996] J.- M. Hsu, and C.-L. Wang, "An Area-Efficient VLSI Architecture for Decoding of Reed-Solomon Codes," *IEEE Int. Conference on Acoustics, Speech and Signal Processing*, vol. 6, pp. 3291-3294, May 1996.
- [Huan1999] J. -C. Huang, C. -M. Wu, M. -D. Shieh and C. -H. Wu, "An Area-Efficient Versatile Reed-Solomon Decoder for ADSL," *IEEE Proc. of the 1999 International Symposium on Circuits and Systems, ISCAS '99.*, vol. 1, pp. 517-520, June 1999.
- [ITU1999] ITU Draft Recommendation G.992.2 G.lite Standard, http://www.itu.int/itudoc/itu-t/rec/g/g800up/g_992.2.html, 1999.
- [JeKu1999] J. H. Jeng, J. M. Kuo, and T. K. Truong, "A High Efficient Multiplier for the RS Decoder," *International Symposium on VLSI Technology, Systems, and Applications*, pp 116-118, June 1999.
- [Jeng1999] J. H. Jeng, and T. K. Truong, "On Decoding of Both Errors and Erasures of a Reed-Solomon Code Using an Inverse-Free Berlekamp-Massey Algorithm," *IEEE Trans. Commun.*, vol. 47, pp. 1488-1494, Oct. 1999.
- [Jenn1998] S. M. Jennings, and J. Kessels, "Comparison of the VLSI Cost/Performance Properties of Two Reed-Solomon Decoding Algorithms," *Integration the VLSI Journal*, vol. 25, pp. 103-110, June 1998.
- [Jeon1995] Y. Jeong, and W. Burleson, "High-Level Estimation of High-Performance Architectures for Reed-Solomon Decoding," *IEEE Int. Symposium on Circuits and Systems*, vol. 1, pp. 720-723, May 1995.
- [Koet2000] R. Koetter, and A. Vardy, "Algebraic Soft-Decision Decoding of Reed-Solomon Codes," Laboratoire I3S, C.N.R.S., France Paper, May 2000.
- [Kwon1997] S. Kwon, and H. Shin, "An Area-efficient VLSI Architecture of a Reed-Solomon Decoder/Encoder for Digital VCRs," *IEEE Trans. on Consumer Electronics*, vol 434, pp. 1019-1027, Nov. 1997.
- [Lin1983] S. Lin, and D. Costello, *Error Control Coding*, Prentice-Hall, 1983.
- [Liu1984] K. Y. Liu, "Architecture for VLSI Design of Reed-Solomon Decoders," *IEEE Trans. Computers*, vol. C-33, pp. 178-179, Feb. 1984.

- [LSIL2000] LSI Logic Corp., "http://www.lsilogic.com/techlib/techdocs/digital_tv/CableModem/777ds3.pdf", 2000.
- [Mass1969] J. L. Massey, "Shift Register Synthesis and BCH Decoding," *IEEE Trans. on Information Theory*, vol. IT-15, no. 1, pp. 122-127, Jan. 1969.
- [Mits1999] Mitsubishi M64403FP IC datasheet, <http://www.mitsubishichips.com/data/datasheets/assps/assppdf/ds/m64403e.pdf>, 1999.
- [Oh1996] Y. U. Oh, and D. Y. Kim, "Method and Apparatus for Computing Error Locator Polynomial for use in a Reed-Solomon Decoder," *U.S. Patent 4 663 470*, 1996.
- [Oh1997] K. Oh, and W. Sung, "An Efficient Reed-Solomon Decoder VLSI with Erasure Correction," *1997 IEEE Workshop On, Signal Processing Systems, 1997.SIPS 97 - Design and Implementation*, pp. 193-201, Nov. 1997.
- [Open2001] Opencores.org, <http://www.opencores.org/cores/reedsolomon/>, 2001.
- [Rady1999] Radyne ComStream, Reed-Solomon Codec, "<http://www.radynecomstream.com/pdf/reedsol.pdf>", 1999.
- [Ragh1997] A. Raghupathy, and K. J. Ray Liu, "Low Power / High Speed Design of a Reed-Solomon Decoder," *IEEE Proc. Int. Symposium on Circuits and Systems*, vol 3, pp. 2060-2063, June 1997.
- [Reed1960] I. S. Reed, and G. Solomon, "Polynomial Codes over Certain Finite Fields," *SIAM Journal of Applied Mathematics*, vol. 8, pp. 300-304, 1960.
- [Reed1991] I. S. Reed, and M. T. Shih, "VLSI Design of Inverse-free Berlekamp-Massey Algorithm," *IEE Proc. Computers and Digital Techniques*, vol. 138, issue 5, Sept. 1991, pp. 295-298.
- [RFI1999] RFI Standard, "Data-Over-Cable Service Interface Specifications," Radio Frequency Interface Specification, SP-RFI-I05-991105.
- [Rusn1997] J. G. Rusnak, "Anywhere in the Home", *IEEE Fourth Int. Workshop on Community Networking Proceedings*, pp. 19-24, Sept. 1997.
- [Saka1995] S. Sakata, and M. Kurihara, "A Fast Parallel Implementation of the Berlekamp-Massey Algorithm with a 1D Systolic Array Architecture," *AAECC-11 Proc., 11th International Symposium*, Paris, France, pp. 415-426, July 1995.
- [Shao1985] H. M. Shao, T. Troung, L. Deutsch, J. Yuen, and I. S. Reed, "A VLSI Design of a Pipeline Reed-Solomon Decoder," *IEEE Trans. on Computers*, vol. C-34, no. 5, pp. 393-403, May 1985.
- [ShTr1985] H. M. Shao, T. K. Truong, I. S. Hsu, L. J. Deutsch, and I. S. Reed, "A Single Chip VLSI Reed-Solomon Decoder," Jet Propulsion Laboratory, Pasadena, CA, TDA Progress Rep. 42-84, pp. 73-81, Oct-Dec. 1985.

- [Shao1988] H. M. Shao, and I. S. Reed, "On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays," *IEEE Trans. on Computers*, vol. 37, no. 10, pp. 1273-1280, Oct 1988.
- [Shao1989] H. M. Shao, T. K. Truong, I. S. Hsu, and L. J. Deutsch, "Architecture for Time or Transform Domain Decoding of Reed-Solomon Codes," *U.S. Patent US4868828*, 1989.
- [Sohi2000] N. Sohi, "A Multi-Standard Set-top Box Channel Decoder", MASC thesis, University of Toronto, 2000.
- [Solj1996] E. Soljanin, and R. Urbanke, "An Efficient Architecture for Implementation of a Multiplier and Inverter in $GF(2^8)$," *Bell-Labs Technical Memo*, BL011217-960308-08TM, 1996.
- [Song2000] L. Song, K. K. Parhi, I. Kuroda, and T. Nishitani, "Hardware/Software Codesign of Finite Field Datapath for Low-Energy Reed-Solomon Codecs," *IEEE Trans. on VLSI Systems*, vol. 8, no. 2, April 2000.
- [Suda1997] M. Sudan, "Decoding of Reed-Solomon Codes Beyond the Error Correction Bound," *J. Complexity*, vol. 12, pp. 180-193, 1997.
- [Sugi1975] Y. Sugiyama, Y. Kasahara, S. Hirasawa, and T. Namekawa, "A Method for Solving Key Equation for Goppa Codes," *Information and Control*, vol. 27, pp. 87-99, 1975.
- [3Com2000] 3Com Corp., "http://www.3com.com/technology/tech_net/white_papers/503052a.html", 2000.
- [Texa2000] Texas Instruments Corp., "<http://www.ti.com/sc/docs/psheets/abstract/apps/spra686.htm>", 2000.
- [Truo1988] 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. on Computers and Digital Techniques*, vol. 135, no.6 pp. 318-324, Nov. 1988.
- [Ultr1997] UltraSPARCI User's Manual, "<http://www.sun.com/microelectronics/manuals/index.html>", 1997.
- [Vard1991] A. Vardy, and Y. Be'ery, "Bit-level Soft-decision Decoding of Reed-Solomon Codes," *IEEE Trans. Comm.*, vol. 39, pp. 440-445, Mar. 1991.
- [Virt2001] Virtual Silicon Technology Inc., Standard Cell Products "http://www.virtual-silicon.com/prod/pd_sc.html", 2001.
- [Welc1986] L. R. Welch, and E. R. Berlekamp, "Error Correction for Algebraic Block Codes," *U.S. Patent US4633470*, 1986.
- [Whit1991] S. R. Whitaker, J. A. Canaris, and K. B. Cameron, "Reed-Solomon VLSI Codec For Advanced Television," *IEEE Trans. Circuits Syst.*, vol. 1, no. 2, pp. 230-236, June 1991.

- [Wick1994] S. Wicker, and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*, IEEE Press, 1994.
- [Wick1995] S. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, 1995.
- [Wilh1999] W. Wilhelm, "A New Scalable VLSI Architecture for Reed-Solomon Decoders," *IEEE J. of Solid-State Circuits*, vol. 34, no. 3, pp. 388-396, March 1999.
- [Xili2000] Xilinx Inc., "http://www.xilinx.com/ipcenter/reed_solomon/", 2000.

Galois Field Primer

Galois fields (GFs) constitute the fundamental mathematical basis for Reed-Solomon (RS) codes. However, the theory and principles behind this algebra is quite extensive, and most of it is beyond the scope of this thesis. Nevertheless, the purpose of describing GFs here is to provide the reader with a modicum of elementary information in order to understand the basic principles of RS codes. The reader who is interested in an in-depth treatment of the subject is referred to [Lin1983] and [Wick1995], which are the sources for most of the material that follows.

A GF is a specialization of various mathematical definitions that are much broader in scope. To begin with the most general concept, a *set* G is an arbitrary collection of elements that do not contain any predefined operations between its elements. A set's cardinality is defined to be the number of elements in the set, which may be finite or infinite. Now let a binary operation "*" be a rule, which when applied to two elements of G , generates a unique third element also in G . Hence, a set G where the binary operation "*" is defined is considered a *group* if the following conditions are satisfied:

- (i) **Associativity:** $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$.
- (ii) **Identity:** an identity element $e \in G$ exists, such that $a * e = e * a = a$ for all $a \in G$.
- (iii) **Inverse:** for any $a \in G$, there exists another unique element $a^{-1} \in G$, such that $a * a^{-1} = a^{-1} * a = e$.
- (iv) **Commutativity (only commutative or abelian group):** for all $a, b \in G$, $a * b = b * a$.

A further division generates *finite groups*, which are constructed using *modulo m* (mod m) addition in the following manner: $a + b = c \text{ modulo } m$. Now, the preceding concepts can be used to define yet another algebraic system called a *field*. A field F , is denoted by a set of elements on which two binary operations, addition “+” and multiplication “*” are defined. In addition, a field must satisfy all of the following properties:

- (i) F is a commutative group under addition +. The identity element e is denoted “0”.
- (ii) The set of nonzero elements in F is a commutative group under multiplication. The multiplicative unity element is labeled “1”.
- (iii) The binary operations are distributive: $a * (b + c) = (a * b) + (a * c)$.

The number of elements in a field is called its *order*. Fields with finite order q , where q is the power of a prime p , are known as Galois fields and are denoted $GF(q)$.

Of concern to digital data transmission and Reed-Solomon codes are the binary field $GF(2)$ and its extension field $GF(2^m)$, where m is a positive integer. $GF(2)$ is the simplest Galois field and consists of only two elements, $\{0,1\}$ under standard binary modulo-2 addition and multiplication.

Next, some basic properties of Galois fields and their elements will be examined. First, let β be an element in $GF(q)$ and let 1 be the multiplicative identity. Consider the following sequence of elements:

$$1, \beta, \beta^2, \beta^3, \beta^4, \beta^5, \beta^6, \dots$$

If β is in $GF(q)$, then this implies that all successive powers are in $GF(q)$ as well. Furthermore, since the field is finite, the sequence must repeat values starting with the identity 1. Hence, the order of β (denoted as $\text{ord}(\beta)$) is the smallest positive integer m such that $\beta^m = 1$, where m must be a divisor of $(q - 1)$. An element of order $(q - 1)$ in $GF(q)$ is denoted as a primitive element in $GF(q)$. Now, let α represent a primitive element in $GF(q)$, and again consider the following sequence:

$$1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{q-2}, \alpha^{q-1}, \alpha^q, \dots$$

So by the primitive element definition, α^{q-1} is the first positive power of α in the above sequence to repeat the value 1. Therefore, all nonzero elements in $GF(q)$ can be represented as $(q - 1)$ consecutive powers of a primitive element α . Multiplication in this nonprime order Galois field is thus accomplished by the modulo $(q - 1)$ addition of the exponents of α .

Polynomials over $GF(q)$ are of special interest since Reed-Solomon codes are simply sets of algebraic curves defined by polynomials with a limited range of degrees. To begin, a polynomial $f(x)$ is irreducible in $GF(q)$ if it cannot be factored into a product of lower degree polynomials in $GF(q)$. An irreducible polynomial $p(x) \in GF(p)$ of degree m is defined to be primitive only if the smallest positive integer n for which $p(x)$ divides $x^n - 1$ is $n = p^m - 1$. All the roots $\{\alpha_j\}$ of the preceding m th-degree polynomial have an order of $p^m - 1$. Consequently, the $(p^m - 1)$ consecutive powers of α form a multiplicative group having an order of $(p^m - 1)$. In order to perform the multiplication operation, the exponents of α are simply added modulo $(p^m - 1)$. Then, the powers of α of degree greater than or equal to m can be expressed as polynomials in α of degree $(m - 1)$ or less. Finally, these polynomials and the zero element form an additive group under polynomial addition.

The following example is provided to facilitate the understanding of GFs for RS code implementations. Consider the following 4th degree primitive polynomial $p(x) = x^4 + x^3 + 1$, with coefficients from $GF(2)$. The field is $GF(2^m) = GF(16)$ with $m = 4$. If α is a root of $p(x)$, then it follows that $\alpha^4 + \alpha^3 + 1 = 0$, or by rearranging $\alpha^4 = \alpha^3 + 1$. Now the exponential, polynomial and vector representations of this field can be generated and are shown below in Table A-1. The vector representation delineates the polynomial form by using m -bit vectors. GF arithmetic is amenable to this bit-vector representation.

Table A-1: $GF(2^4)$ Based on the Primitive Polynomial $x^4 + x^3 + 1$

Exponential Representation	Polynomial Representation	Vector Representation (α^3) (α^2) (α^1) (α^0)
α^0	1	0 0 0 1
α^1	α	0 0 1 0
α^2	α^2	0 1 0 0
α^3	α^3	1 0 0 0
α^4	$\alpha^4 = \alpha^3 + 1$	1 0 0 1
α^5	$(\alpha)\alpha^4 = \alpha(\alpha^3 + 1) = \alpha^3 + \alpha + 1$	1 0 1 1
α^6	$(\alpha)\alpha^5 = \alpha(\alpha^3 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha + 1$	1 1 1 1
α^7	$\alpha^2 + \alpha + 1$	0 1 1 1
α^8	$\alpha^3 + \alpha^2 + \alpha$	1 1 1 0

Table A-1: GF(2⁴) Based on the Primitive Polynomial $x^4 + x^3 + 1$

Exponential Representation	Polynomial Representation	Vector Representation (α^3) (α^2) (α^1) (α^0)
α^9	$\alpha^2 + 1$	0 1 0 1
α^{10}	$\alpha^3 + \alpha$	1 0 1 0
α^{11}	$\alpha^3 + \alpha^2 + 1$	1 1 0 1
α^{12}	$\alpha + 1$	0 0 1 1
α^{13}	$\alpha^2 + \alpha$	0 1 1 0
α^{14}	$\alpha^3 + \alpha^2$	1 1 0 0
$\alpha^{15} = \alpha^0$	$(\alpha)\alpha^{14} = \alpha(\alpha^3 + \alpha^2) = (\alpha^4 + \alpha^3) = \alpha^3 + 1 + \alpha^3 = 1$	0 0 0 1

Each representation has a particular use in GF arithmetic. Multiplication is facilitated by using the exponential form of the field. To calculate a product, the value of the exponents are simply added together modulo $(2^m - 1)$ and an example is shown below.

$$\alpha^{13} * \alpha^{12} = \alpha^{13+12} = \alpha^{25 \bmod 15} = \alpha^{10}$$

On the other hand, the polynomial/vector representation is more suited towards GF addition. Bits of the vector form are simply XORed together and the resultant vector is the sum. The following example serves to illustrate GF addition:

$$\alpha^{13} + \alpha^{10} = 0110 + 1010 = 1100 = \alpha^{14}$$

Two other operations that are of interest when considering RS codes and GFs are subtraction and division. Subtraction for GF(2^m) is identical to addition, where the vector forms of the two summands are XORed together. Division can be dichotomized into performing an inversion in GF(2^m) and then a multiplication. Thus, the inverse of an element α^b is simply α^{n-b} , where $n = 2^m - 1$.

A similar procedure is followed in order to generate Galois fields for various RS codes. For instance, ADSL and powerline communications are defined by the GF(2⁸) primitive polynomial of $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. However, now the vector symbol representation of field elements has a length of 8-bits (1 byte) since the degree $m = 8$.

Reed-Solomon MATLAB Code

B.1 Time-Domain Code

B.1.1 Syndrome Calculation

```
% TITLE:    Syndrome Calculation
% DESCRIPTION: Calculate Syndromes for an input received word
%             This program does not use time sharing, it calculates
%             2t syndromes where t is defined below.
%
% function synd = synCalc(r,n,t,tp)
%
% Parameters
% r: input received codeword as coefficients of  $x^{n-1}$ ,  $x^{n-2}$ , ...,  $x^0$ .
% n: block length defined as  $(2^m)-1$ 
% t: code's error correcting capability
% tp: vector tuple containing all elements of  $GF(2^m)$ 

function synd = synCalc(r,n,t,tp)

    % define alpha registers from  $a^1$  to  $a^{2*t}$ 
    alpha = 1:2*t;

    % initialize registers to zero ( $\alpha^{-\infty} = \text{zero}$ )
    A(1:2*t) = -Inf;
    B(1:2*t) = -Inf;

    % Calculate 2t syndromes
    %for timestep = 1:n,    % for non-script purposes (simulate manually)
    for timestep = n:-1:1,  % have to reverse rx word input order [0 ... 14]
        for i = 1:2*t,
```

```

        A(i) = gfadd(r(timestep), gfmul(alpha(i), A(i), tp), tp);
    end;
end;

synd = A; % The syndromes [S1 S2 S3 ... S2t]
% End program

```

B.1.2 α^k Generation

```

% TITLE:    Alpha^k Generation
% DESCRIPTION:  Converts the received erasure flags into powers of
%              alpha (alpha^k)
%
% function AK = Akgen(n,erasVec)
%
% Parameters
% n: block length defined as (2^m)-1
% erasVec: binary string where the presence of a '1' indicates an erasure
% indexA: the vector index of alphaK
% alphaK: vector containing powers of alpha indicating erasure flag locations

function AK = Akgen(n,erasVec)

    % erasVec is input MSD first. [2t-1, 2t-2. ... 2, 1, 0]
    alphaK=0;
    indexA=1;

    if sum(erasVec)==0 % If no erasures then output -Inf as a flag
        alphaK = -Inf;
    else
        for i=n:-1:1,
            if erasVec(i) == 1
                alphaK(indexA) = i-1;
                indexA = indexA + 1;
            end;
        end;
    end;

    AK = alphaK; % AK = -Inf if no erasures or AK = [2t-1, ..., 1, 0] for erasures
% End program

```

B.1.3 Polynomial Expansion (Forney Syndrome Calculation)

```

% TITLE:    Polynomial Expansion I (Syndromes and alpha^K gen. included)
% DESCRIPTION:  Converts the powers of alpha^K and the syndromes into a
%              polynomial (Forney Syndromes).
%
% function Fsyn = synPolyEx(synd,AK,numEras,t,tp)
%

```

% Parameters

% n: block length defined as $(2^m)-1$

% t: code's error correcting capability (where $t = \text{floor}[(d-1)/2]$)

% tp: vector tuple containing all elements of $GF(2^m)$

% AK: output of the α^K generation block

% synd: output of the syndrome calculation block

function Fsyn = synPolyEx(synd,AK,numEras,t,tp)

% synd is the syndrome = $[S_1 S_2 \dots S_{2t-1} S_{2t}]$. right justify in registers.
t2=2*t;

% load registers with 2t syndromes

F = synd;

if (AK ~= -Inf) **% if no erasures then output = syndrome polynomial**

for i=1:numEras, **% for each α^i value read in**

for k = 1:t2-1,

Ftmp = gfadd(F(t2), gfmul(F(t2-1), AK(i), tp), tp);

for m = t2:-1:2,

F(m) = F(m-1); **% Everything shifts one to the right for each gfmul/gfadd**

end;

F(1) = Ftmp; **% Put new value into S1**

end;

for m = t2:-1:2, **% After t2-1 shifts are over, shift everything again**

F(m) = F(m-1); **% so that T1 can be replaced by S1**

end;

F(1) = synd(1); **% T1 is always the same value as S1**

end;

end;

Fsyn = F; **% The Forney syndromes $[T_1, T_2, \dots, T_{2t}]$**

% End program

B.1.4 Polynomial Expansion II (Erasure Locator Calculation)

% TITLE: Polynomial Expansion II (Only α^K generation included)

% DESCRIPTION: Converts the powers of α^K into a polynomial. Uses modified

% structure with only one gfmul and one gfadd instead of 2t gfmul

% and 2t gfadd.

%

% function erasLoc = alphaPolyEx(AK,t,numEras,tp)

%

% Parameters

% n: block length defined as $(2^m)-1$

% t: code's error correcting capability (where $t = \text{floor}[(d-1)/2]$)

% tp: vector tuple containing all elements of $GF(2^m)$

% AK: output of the α^K generation block

% numEras: Number of erasures detected (if AK does not = -Inf)

% pro: product of registers and α^k

function erasLoc = alphaPolyEx(AK,t,numEras,tp)

% 2t elements

t2=2*t;

% Elements are left aligned

% $x^2, x^1, x^0, -\text{Inf}, -\text{Inf}, \dots, -\text{Inf}$

reg(1:t2+1)= -Inf;

reg(1) = 0;

if (AK ~= -Inf) % Skip if no erasures

for i=1:numEras,% for each α^i value read in

for k = 1:t2,

lsdcase = gfmul(reg(t2), AK(i), tp);

if ((i > (t2-2)) & (k == t2)) % Special case for implemented algorithm

Stmp = lsdcase;

else

Stmp = gfadd(reg(t2-1), lsdcase, tp);

end;

for m = t2:-1:2,

reg(m) = reg(m-1);

end;

reg(1) = Stmp;

end;

end;

end;

if numEras == t2 % Special case for 2t erasures

reg(t2+1) = 0; % t2+1 register must be set to 0 if there are 2t erasures

else

reg(t2+1) = -Inf; % No need for extra register, less than 2t erasures

end;

erasLoc = reg; % erasLoc = [$x^4, x^3, \dots, x^0$ -Inf, -Inf]

% Special Note: If there are more than 2t-2 erasures, then the special case must be used.

% When shifting on the 2t-1 or 2t case, the LSD is a special case. Instead of gfmul

% α^{erasure}

% location by R2t and then gfadd R2t-1, there is simply a gfmul. That value is then fed back

% into R1. Only for cases when 2t-1 or more erasures.

% End program

B.1.5 Modified Euclidean Algorithm

% TITLE: Modified Euclid's Algorithm (MEA)%

% DESCRIPTION: Performs the modified Euclid's Algorithm.

```

%
% function eu = MEA(Fsyn,t,tp,numEras,erasLoc)
%
% Parameters
% t: code's error correcting capability (where  $t = \text{floor}[(d-1)/2]$ )
% tp: vector tuple containing all elements of  $GF(2^m)$ 
% numEras: Number of erasures detected (if AK does not = -Inf)
% Fsyn: Forney syndrome corresponding to the received vector. Obtained from
%       synPolyEx function.
% erasLoc: Erasure locator polynomial. Obtained from alphaPolyEx function.

function eu = MEA(Fsyn,t,tp,numEras,erasLoc)

    % initializations
    maxShift=1; % extra space added to length of lambda and mu to accomodate shifting
    t2 = 2*t;
    sz = 2*t+1; % size of fields: 2t is max. degree, +1 for  $x^0$ 
    sz2 = 2*t+1+maxShift; % mu could be sz long if t2 eras, could be shifted by
                          % maxShift if  $dR \gg dQ$ 

    doneFlag = 0;

    % Power representation of field elements
    Q(1,sz) = -Inf;
    Q(1,1:t2) = Fsyn;

    R(1:sz) = -Inf;
    R(sz) = 0;
    lambda(1:sz2) = -Inf;

    mu(1:sz2) = -Inf;
    for j=t2+1:-1:1,
        if erasLoc(j) ~= -Inf
            firstNonZ=j;
            break;
        end;
    end;
    for j = 1:firstNonZ,
        mu(j) = erasLoc(firstNonZ-j+1);
    end;

    dR = deg(R,sz);
    dQ = deg(Q,sz);

    % -----
    % Main Cell Loop
    % Using only 1 cell for the improved architecture of MEA
    % input to each cell is R, Q, start, lambda, and mu
    while doneFlag == 0,

        maxDegP1 = max(dR,dQ) + 1;

```

```

% pre-flowchart setup:
l = dR - dQ;

if l >= 0,                                % dR and dQ are not equal
    sigFlag = 1;
else
    sigFlag = 0;
end;

% Align all inputs properly
if l ~= 0
    absI = abs(l);
    if sigFlag == 1                        % dR > dQ
        Qtmp(absI+1:sz) = Q(1:sz-absI);    % adjust Q, mu
        Qtmp(1:absI) = -Inf;
        muTmp(absI+1:sz2) = mu(1:sz2-absI);
        muTmp(1:absI) = -Inf;
        Rtmp = R;
        lambdaTmp = lambda;
    else % ie. if sigFlag == 0            % dQ > dR
        Rtmp(absI+1:sz) = R(1:sz-absI); % adjust R, lambda
        Rtmp(1:absI) = -Inf;
        lambdaTmp(absI+1:sz) = lambda(1:sz-absI);
        lambdaTmp(1:absI) = -Inf;
        Qtmp = Q;
        muTmp = mu;
    end;
else % dR = dQ
    Rtmp = R;                                % no alignment necessary; l(i) == 0
    lambdaTmp = lambda;
    Qtmp = Q;
    muTmp = mu;
end;

% -----
% flowchart implemented:
% Check to see whether algorithm is completed
% <step 1>
if ((dR <= floor((t2+1+numEras-3)/2)) | (dQ <= floor((t2+1+numEras-3)/2)))
    doneFlag = 1;

    if dR < dQ    %To swap, or not? <step 2>
        tmp = R;    % step <3>
        R = Q;
        Q = tmp;    % swap R, Q
        tmp = Rtmp;
        Rtmp = Qtmp;
        Qtmp = tmp;    % swap Rtmp, Qtmp
        tmp = lambda;
        lambda = mu;
    end;
end;

```



```

    mu = tmp;          % swap lambda, mu
    tmp = lambdaTmp;
    lambdaTmp = muTmp;
    muTmp = tmp;      % swap lambdaTmp, muTmp
    tmp = dR;
    dR = dQ;
    dQ = tmp; % swap current dR, dQ
end;

if Qtmp(maxDegP1) == -Inf      % Decision <4>
    dQ = dQ - 1;              % step <5>
    if (dQ <= floor((t2+1+numEras-3)/2))
        lambda = mu; % algorithm stops, output = mu(i)
        doneFlag = 1;
    end;                      % algorithm continues

else                          % ie. if Qtmp(maxDegP1) ~= -Inf
    dR = dR - 1;

    % We need to calculate R(i+1) and lambda(i+1)
    % calculate R(i+1), lambda(i+1)
    % assume that the "x^4" term has been handled by input alignment routine

    a = Rtmp(maxDegP1); % leading coefficient of R
    b = Qtmp(maxDegP1); % leading coefficient of Q

    nextR1 = gfmul(b,Rtmp,tp);
    nextR2 = gfmul(a,Qtmp,tp);
    R = gfadd(nextR1,nextR2,tp);

    nlambda1 = gfmul(b,lambdaTmp,tp);
    nlambda2 = gfmul(a,muTmp,tp);
    lambda = gfadd(nlambda1,nlambda2,tp);

    % pass Q and mu to next cell
    if (dR <= floor((t2+1+numEras-3)/2)) %decision <6a>
        doneFlag = 1;    % algorithm stops, output = lambda(i+1)
    end;

end;                          % if - Qtmp(maxDegP1) == -Inf check
end;                          % if (step <1>)
end;                          % ending main while loop

% -----
if dR <= floor((t2+1+numEras-3)/2) % dR and dQ should have totally trickled thru
    lambdaFin = lambda;% accept last lambda as final result
    RFin = R;
elseif dQ <= floor((t2+1+numEras-3)/2) %t
    lambdaFin = mu;% accept last mu as final result
    RFin = Q;
else

```

```

    'modEuc error - done all cells, but one of the degrees is still not < t'
end;%if

tauleadingCo = lambdaFin(1);
AleadingCo = RFin(1);
    if tauleadingCo == -Inf
        'modEuc error: lambdaFin is empty'
    elseif AleadingCo == -Inf
        'modEuc error: RFin is empty'
    end;

eutmp(1,:) = lambdaFin(1:sz);    %tau = sigma errata locator polynomial
eutmp(2,:) = RFin;              %A = omega errata evaluator polynomial
eu = eutmp;
% End program

```

B.1.6 Chien Search

```

% TITLE:      Chien search algorithm
% DESCRIPTION: Performs the Chien search for finding the roots of the errata
%              locator and evaluator polynomials o(x) and w(x) respectively.
%              Does an exhaustive search and evaluates o(alpha^i), o'(alpha^i)
%              and w(alpha^i) for 0 <= i <= n-1.
%
% function cs = Chien(eu,numEras,n,t,tp)
%
% Parameters
% euc: received errata locator and evaluator polynomials with coefficients of
%      x^0, x^1, ...x^2t-1.
% n: block length defined as (2^m)-1
% t: code's error correcting capability (where t = floor[(d-1)/2])
% tp: vector tuple containing all elements of GF(2^m)
% deg: degree + 1 of input polynomial

function cs = Chien(eu,numEras,n,t,tp)

```

```

    t2 = t*2;
    ohmflag = 0;

```

```

% Special case if maximum amount of erasures present
if numEras == t2
    choice = t2 + 1;
else
    choice = t2;
end;

```

```

% define alpha registers from a^0 to a^2*t because of case with 2t erasures
alpha = 0:t2;

```

```

% separate errata locator and errata evaluator polynomials and initialize registers
sig = eu(1,1:choice); % errata locator polynomial
omega = eu(2,1:choice); % errata evaluator polynomial

% initialize roots and op registers
op(1:choice) = -Inf;
ohm(1:choice) = -Inf;

% initialize sigma_prime multiplication factor
bfactor = n-1;
x = 1;

% Find the roots of the input polynomial
for symbol = n*2+2:-1:1,
    for i = 1:choice,
        out(i) = sig(i);
        tmp(i) = gfmul(alpha(i),sig(i),tp);
        sig(i) = omega(i);
        omega(i) = tmp(i);
    end;
    if (symbol <= (n*2))
        % Clear summations for the next summation
        evensum = -Inf;
        oddsum = -Inf;
        for ck = t:-1:1,
            oddsum = gfadd(oddsum, out(2*ck), tp);
            evensum = gfadd(evensum, out(2*ck-1), tp);
        end;
        if numEras == t2
            evensum = gfadd(evensum, out(choice), tp);
        end;
        sigorom = gfadd(oddsum, evensum, tp);

        if (mod(symbol,2) == 0) % check whether even or odd clock count for symbol.
                                % even clock, find sig and sigprime
            sigoutodd = oddsum;
            sigout = sigorom;
            if sigorom == -Inf % check to see whether root has been found (sigma = 0)
                ohmflag = 1;
                sigprime(x) = gfmul(sigoutodd, bfactor, tp);
            else
                sigprime(x) = -Inf;
            end;
            bfactor = gfmul(bfactor, (n-1), tp);
        else
            ohmout = sigorom;
            if ohmflag == 1
                ohm(x) = ohmout;
                ohmflag = 0;
            else

```

```

        ohm(x) = -Inf;
    end;
    x = x + 1;
end;
end;
end;

for i = n:-1:1,
    if sigprime(i) ~= -Inf
        cser(n-i+1) = gfddiv(ohm(i),sigprime(i),tp)';
    else
        cser(n-i+1) = -Inf;
    end;
end;

cs = cser;
% End program

```

B.1.7 Polynomial Degree Determination

```

% TITLE:    Polynomial Degree Determination
% DESCRIPTION: Determines the degree of the input polynomial

% Parameters
% poly: Input polynomial
% sz: length of the input polynomial vector

function degree = deg(poly,sz)

```

```

    i = sz;
    degrf = 0;
    while i > 0,
        if poly(i) ~= -Inf
            degrf = i-1;
            i = 1;
        end;
        i = i -1;
    end;
    degree = degrf;
% End program

```

B.1.8 Sample Testbench

% Tests only one error case with various erasure combinations.

```

% initialization of default parameters
m = 8;           % bits per message symbol
p = 2;           % base of Galois field GF(p^m)
n = 2^m-1;       % codeword length

```

```

k = 239;           % message length
t = floor((n - k)/2); % error correction capability of RS code
eflag = 0;         % error flag initially 0;
x = 1;             % error detector counter
t2 = t*2;

tp = gftuple([-1 : p^m-2]',m,p); % generate all elements in GF(2^m)
Pg = rspoly(n,k,tp);           % generator polynomial

NUM_ROW = 1; % Number of codewords that the program will correct
MSG = randint(NUM_ROW,k, [-1,n-1], 1008); % 4 errors 8 erasures in this case

c = rsencode(MSG,Pg,n,tp); % generation of RS codewords from message words
erasV = randbit(1,255,[1 1 1 1]/4, 901); % generation of 4 random errors
erasVec = randbit(1,255,[1 1 1 1 1 1 1 1]/8, 8); % generation of 8 random erasures

% change -1 in c to -Inf for our algorithm
for i=1:NUM_ROW,
    for j=1:n,
        if c(i,j) == -1
            c(i,j) = -Inf;
        elseif c(i,j) == n
            c(i,j) = 0;
        end;
    end;
end;

r = c + erasVec + erasV;

synd = synCalc(r,n,t,tp); % calculate the syndromes
numEras = sum(erasVec); % finds the number of erasures (if any)
AK = Akgen(n,erasVec); % finds the alpha powers from the erasure vector
erasLoc = alphaPolyEx(AK,t,numEras,tp); % calculate the errata locator polynomial
Fsyn = synPolyEx(synd,AK,numEras,t,tp); % calculate the Forney syndromes
eu = MEA(Fsyn,t,tp,numEras,erasLoc); % find omega and sigma polynomials
cs = Chien(eu,numEras,n,t,tp); % Chien search
rtmp = gfadd(cs,r,tp); % add the corrected symbols with the input symbols
isequal(rtmp,c) % is the decoded codeword = generated codeword?

```

B.2 Frequency - Domain Code

B.2.1 Syndrome Calculation

% See Time-Domain Code (Section B.1.1) for Description
function syn = synCalcBW(r,n,t,tp)

```
% load alpha registers
for i = 1:2*t,
    alpha(i)=i;
end

% initialize registers to zero (alpha^~Inf)
A(1:2*t) = -Inf;
B(1:2*t) = -Inf;

%update each register
for timestep = 1:n,
    for i = 1:2*t,
        A(i) = r(timestep);
        B(i) = gfadd(A(i), gfmul(alpha(i), B(i), tp), tp);
    end;
end

syn = B;
% End Program
```

B.2.2 α^k Generation

%alphaKgen: we're working in the power notation of GF elements
% if the 1st few elements of the input erasure flag sequence are 0, 1, 0, 1,..
% (ie. a '1' in positions that are erased) we should output alpha^1, alpha^3,..

function aK = alphaKgen(n,erasVec)

```
alphaK=0;
indexA=1;

if sum(erasVec)==0
    alphaK=0;
else
    for i=1:n
        if erasVec(i) == 1
            alphaK(indexA) = i-1;
            indexA = indexA + 1;
        end %if
    end %for
end %if
aK=alphaK;
% End Program
```

B.2.3 Erasure Locator Polynomial Calculation

% See Time-Domain Code (Section B.1.4) for description
function elp = alphaExp2(numEras,alph, t, tp)

```

t2=2*t;
numAlph=length(alph);
reg=zeros(1,t2+1);           %2t elements.
reg(1:t2)=-Inf;

if numEras>0
    for i=1:numAlph           % for each alpha^i value read in
        pro=gfmul(alph(i), reg, tp);
        for j=1:t2           % for each register
            reg(j)=gfadd(pro(j),reg(j+1),tp);
        end
        reg(t2+1)=pro(t2+1);
    end
end;
elp = reg;
% End Program

```

B.2.4 Forney Syndrome Polynomial Calculation

% See Time-Domain Code (Section B.1.3) for explanation
%
% numEras: number of erasures in codeword
% syn: syndrome calculated from the syndrome unit
% alph: powers of alpha correspondign to erasures
% t: error correcting capability
% tp: list of all elements in GF(256), generated from tp = gftuple([-1:n-1]', m)

function Fsyn = polyExp2(numEras,syn, alph, t, tp)

```

%syn is the syndrome = S1 S2 S3 S4 S5 S6. right justify in registers.
t2=2*t;
numAlph=length(alph);
S=syn;

if numEras>0
    for i=1:numAlph           % for each erasure (alpha^i) input
        prod=gfmul(alph(i), S(1:t2-1), tp);
        for j=t2:-1:2         % S1 remains the same throughout calculation
            S(j)=gfadd(prod(j-1),S(j),tp);
        end
    end
end
end
Fsyn=S;
% End Program

```

B.2.5 Modified Euclidean Algorithm

```

% See Time-Domain Code (Section B.1.5) for Description
%
% Fsyn: Forney syndrome corresponding to received vector
% t: error correcting capability
% tp: list of all elements in GF(256), generated from tp = gftuple([-1:n-1]', m)
% numEras: number of erasures
% erasLoc: output of alphaExp: erasure locator polynomial

function sig = modEuc(Fsyn,t,tp,numEras,erasLoc)

    % initializations
    maxShift=1; % extra space added to length of lambda and mu to accomodate shifting
    i = 1;
    t2 = 2*t;
    sz = 2*t+1; % size of fields: 2t is max. degree, +1 for x^0
    sz2 = 2*t+1+maxShift; % mu could be sz long if t2 eras, could be shifted by
                        % maxShift if dR>>dQ

    % we are using the power representation of field elements
    Q = zeros(sz);
    Q(:, :) = -Inf;
    Q(1,1:t2) = Fsyn;

    R = zeros(sz);
    R(:, :) = -Inf;
    R(1,sz) = 0;

    lambda = zeros(sz,sz2);
    lambda(:, :) = -Inf;

    mu = zeros(sz,sz2);
    mu(:, :) = -Inf;
    for j=1:t2+1
        if erasLoc(j) ~= -Inf
            firstNonZ=j;
            break;
        end;
    end;
    mu(1,1:(t2-firstNonZ+2)) = erasLoc(firstNonZ:t2+1);

    dR = zeros(1,sz);
    dR(1) = deg(R(i,:),sz);
    dQ = zeros(1,sz);
    dQ(1) = deg(Q(i,:),sz);

    %main cell loop (input to each cell is R, Q, start, lambda, and mu)
    for i = 1:t2,
        maxDegP1 = max(dR(i), dQ(i)) + 1;

```



```

% pre-flowchart setup:
l(i) = dR(i) - dQ(i);

if l(i) >= 0 ,
    sigFlag(i) = 1;
else
    sigFlag(i) = 0;
end

if l(i) ~= 0% align inputs properly
    absI = abs(l(i));
    if sigFlag(i) == 1
        Qtmp(absI+1:sz)=Q(i,1:sz-absI); % adjust Q, mu but R, lambda remain the same
        Qtmp(1:absI) = -Inf;
        muTmp(absI+1:sz2) = mu(i,1:sz2-absI);
        muTmp(1:absI) = -Inf;
        Rtmp = R(i,:);
        lambdaTmp = lambda(i,:);
    else % ie. if sigFlag(i) == 0
        Rtmp(absI+1:sz)=R(i,1:sz-absI); % adjust R, lambda but Q, mu remain the same
        Rtmp(1:absI) = -Inf;
        lambdaTmp(absI+1:sz) = lambda(i,1:sz-absI);
        lambdaTmp(1:absI) = -Inf;
        Qtmp = Q(i,:);
        muTmp = mu(i,:);
    end
else % no alignment necessary; l(i) == 0
    Rtmp = R(i,:);
    lambdaTmp = lambda(i,:);
    Qtmp = Q(i,:);
    muTmp = mu(i,:);
end

% flowchart implemented: step <1>:done?
if ((dR(i)<=floor((t2+1+numEras-3)/2)) | (dQ(i)<=floor((t2+1+numEras-3)/2)))
    dR(i+1) = dR(i);
    dQ(i+1) = dQ(i);

    if (dR(i) <= floor((t2+1+numEras-3)/2))
        lambda(i+1,:) = lambda(i,:); % pass lambda(i) to next cell
    elseif (dQ(i) <= floor((t2+1+numEras-3)/2))
        mu(i+1,:) = mu(i,:); % pass mu(i) to next cell
    end

else % regular case of the iterative algorithm
    if dR(i) < dQ(i) % step <2>: To swap, or not?
        tmp = R(i,:); % step <3>
        R(i,:) = Q(i,:);
        Q(i,:) = tmp; % swap R, Q
    end
end

```

```

    tmp = Rtmp;          % step <3>
    Rtmp = Qtmp;
    Qtmp = tmp;          % swap Rtmp, Qtmp

    tmp = lambda(i,:);
    lambda(i,:) = mu(i,:);
    mu(i,:) = tmp;       % swap lambda, mu

    tmp = lambdaTmp;
    lambdaTmp = muTmp;
    muTmp = tmp;        % swap lambdaTmp, muTmp

    tmp = dR(i);
    dR(i) = dQ(i);
    dQ(i) = tmp;        % swap current dR, dQ
end                      % end if <step 2>

if Qtmp(maxDegP1) == -Inf    % step <4>
    dR(i+1) = dR(i);
    dQ(i+1) = dQ(i) - 1;    % step <5>

    if (dQ(i+1) <= floor((t2+1+numEras-3)/2))
        Q(i+1,:) = Q(i,:);
        mu(i+1,:) = mu(i,:);    % algorithm stops, output = mu(i)
    else
        % algorithm continues
        R(i+1,:) = R(i,:);
        Q(i+1,:) = Q(i,:);
        lambda(i+1,:) = lambda(i,:);
        mu(i+1,:) = mu(i,:);
    end
    % if dQ(i+1) < t
else
    % ie. if Qtmp(maxDegP1) ~= -Inf
    dR(i+1) = dR(i) - 1;
    dQ(i+1) = dQ(i);

    % calculate R(i+1), lambda(i+1)
    a = Rtmp(maxDegP1);    % leading coefficient of R
    b = Qtmp(maxDegP1);    % leading coefficient of Q

    nextR1 = gfmul(b,Rtmp,tp);
    nextR2 = gfmul(a,Qtmp,tp);
    R(i+1,:) = gfadd(nextR1,nextR2,tp);

    nlambda1 = gfmul(b,lambdaTmp,tp);
    nlambda2 = gfmul(a,muTmp,tp);
    lambda(i+1,:) = gfadd(nlambda1,nlambda2,tp);
    Q(i+1,:) = Q(i,:);
    mu(i+1,:) = mu(i,:);

    % if (dR(i+1) <= floor((t2+1+numEras-3)/2)) algorithm stops, output = lambda(i+1)
end
    %if - Qtmp(maxDegP1) == -Inf check

```

```

        end    % if (step <1>)
    end        % main for

    if dR(sz) <= floor((t2+1+numEras-3)/2)    % dR and dQ should have totally trickled thru
        lambdaFin = lambda(sz,:);            % accept last lambda as final result
    elseif dQ(sz) <= floor((t2+1+numEras-3)/2)
        lambdaFin = mu(sz,:); % accept last mu as final result
    else
        'modEuc error - done all cells, but one of the degrees is still not < t'
    end

    degr = deg(lambdaFin,sz);
    leadingCo = lambdaFin(1);
    if leadingCo == -Inf
        'modEuc error: lambdaFin is empty'
    end

    sig = gfddiv(lambdaFin,leadingCo,tp)';
% End Program

```

B.2.6 Remaining Error Transform Calculation

```

% Need to Calculate all remaining error transforms.
% The Syndromes represent the first (n-k) error transforms.
%
% syn: syndrome from synCalc
% sig: sigma (monic error locator) from the Modified Euclidean Algorithm
% t: error correcting capability
% n: RS codeword length
% tp: tuple containing all elements of GF(2^m)

```

```

function E = remE(syn,sig,t,n,tp)

    E = syn(1:t2);    % first t2 coordinates of E are the syndromes
    t2 = 2*t;
    if t2 ~= length(syn)
        'remE error - syndrome is of incorrect length'
    end

    % find the first nonzero component
    R=sig(1:t2+1);
    for i=1:t2 + 1
        if ((R(i) == -Inf) & (R(i + 1) == 0))
            fnz = i + 2;    % index of first nonzero component
            break;
        end
        if ((R(i) == 0) & (i == 1))
            fnz = 2;
            break;
        end
    end

```

```

    end
end

%find the last nonzero component
lnz=t2+1;
for i=t2+1:-1:1
    if R(i)==-Inf
        lnz=lnz-1;
    else
        break;
    end
end
end

y=0;
sigm = R(fnz:lnz);          % coefficients of sigma poly to be used to calculate E trans
lenSigm = lnz + 1 - fnz;
% produce remaining coordinates of E
for j = 1:(n - t2)
    tmpE = E(t2+j-1:-1:t2+j-lenSigm);
    vect2add = gfmul(sigm, tmpE, tp);          % must add elements in vect2add
    acc = -Inf;
    for i=1:lenSigm
        acc = gfadd(acc,vect2add(i),tp);
    end
    E(t2 + j) = acc;
end
end
% End Program

```

B.2.7 Inverse Transform

```

% Calculate the inverse Fourier Transform of E to obtain the error pattern
% Almost identical to syndrome calculation except occurs over all n
%
% E: coefficients of input transform;  $E_k$ ,  $1 \leq k \leq n-1$ 
% m: bits/symbol
% n: block length defined as  $(2^m) - 1$ 
% t: error correcting capability
% tp: tuple containing all elements of  $GF(2^m)$ 

```

```
function e = invTrans(E,m,n,t,tp)
```

```

    % load alpha registers
    for i = 1:n,
        alpha(i)=n-i;
    end

```

```

A(1:n) = -Inf;
B(1:n) = -Inf;

```

```

% All processing elements
for timestep = 1:n
    for i = 1:n,
        A(i) = E(timestep);
        B(i) = gfadd(A(i), gfmul(alpha(i), B(i), tp), tp);
    end;
end
e = B;      % Output which will be GF added to the received codeword for final result
% End Program

```

B.2.8 Sample Testbench

% Tests only one error case with various erasure combinations.

% initialization of default parameters

```

m = 8;          % bits per message symbol
p = 2;          % base of Galois field GF(p^m)
n = 2^m-1;      % codeword length
k = 239;        % message length
t = floor((n - k)/2); % error correction capability of RS code
eflag = 0;      % error flag initially 0;
t2 = t^2;

```

```

tp = gftuple([-1 : p^m-2]',m,p); % generate all elements in GF(2^m)
Pg = rspoly(n,k,tp);             % generator polynomial

```

NUM_ROW = 1; % Number of codewords that the program will correct

MSG = randint(NUM_ROW,k, [-1,n-1], 2004); % 5 errors and 4 erasures

```

c = rsencode(MSG,Pg,n,tp); % generation of RS codewords from message words
erasV = randbit(1,255,[1 1 1 1 1]/5, 8); % generation of 5 random errors
erasVec = randbit(1,255,[1 1 1 1]/4, 12435); % generation of 4 random erasures

```

% change -1 in c to -Inf for our algorithm

```

for i=1:NUM_ROW,
    for j=1:n,
        if c(i,j) == -1
            c(i,j) = -Inf;
        elseif c(i,j) == n
            c(i,j) = 0;
        end;
    end;
end;

```

r = c + erasVec + erasV;

```

syn = synCalcBW(r,n,t,tp); % calculate the syndromes
numEras = sum(erasVec); % finds the number of erasures (if any)
alph=alphaKgen(n,erasVec); % finds the alpha powers from the erasure vector
erasLoc = alphaExp2(numEras,alph, t, tp); % calculate the errata locator polynomial

```

```
Fsyn=polyExp2(numEras, syn, alph, t, tp); % calculate the Forney syndromes
sig = modEuc2(Fsyn,t,tp,numEras,erasLoc); % find omega and sigma polynomials
E = remEnew2(syn,sig,t,n,tp); % calculate the remaining error transforms
e = invTrans(E,m,n,t,tp); % find the inverse transform of the error pattern
rtmp = gfadd(e,r,tp); % add the corrected symbols with the input symbols
isequal(rtmp,c) % is the decoded codeword = generated codeword?

% End Program
```

Software Profiling Results

C.1 Time-Domain Profiling Session

The following profiling session was obtained using the *gprof* program. First, some of the more relevant process names found in Table C-2 and Table C-3 are explained as follows:

Table C-1: Profiler Process Name Explanations

Process Name	Explanation
rs_errval	Evaluate the errata polynomials by performing the Chien search algorithm.
.umul	All multiplication functions.
internal_mcount	Initialize internal CPU memory for the given program.
syndrome	Syndrome calculation unit.
euclid	Modified Euclidean algorithm.
erasure_init	Initialize all polynomials and variables for the modified Euclidean decoding algorithm.
eras_dec_rs	Overall decoder program which calls the above functions.
inv_trans	FDD inverse transform calculation unit.
E_trans	FDD remaining error transform calculation unit.

granularity: each sample hit covers 4 byte(s) for 0.08% of 13.33 seconds

Table C-2: Sample TDD *gprof* Profiling Session

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Process Name
36.4	4.85	4.85	10000	0.48	0.48	rs_errval
27.9	8.57	3.72	–	–	–	internal_mcount
21.2	11.39	2.82	40.32M	0.00	0.00	.umul
9.6	12.67	1.28	10000	0.13	0.41	syndrome
2.4	12.99	0.32	10000	0.03	0.03	euclid
1.7	13.22	0.23	–	–	–	_mcount
0.5	13.29	0.07	10000	0.01	0.01	erasure_init
0.3	13.33	0.04	10000	0.00	0.94	eras_dec_rs
0.0	13.33	0.00	10001	0.00	0.00	_memcpy
0.0	13.33	0.00	10000	0.00	0.00	_memcmp
0.0	13.33	0.00	503	0.00	0.00	_randomjunk
0.0	13.33	0.00	503	0.00	0.00	random
0.0	13.33	0.00	254	0.00	0.00	.rem
0.0	13.33	0.00	30	0.00	0.00	_return_zero
0.0	13.33	0.00	15	0.00	0.00	_mutex_unlock
0.0	13.33	0.00	15	0.00	0.00	mutex_lock
0.0	13.33	0.00	4	0.00	0.00	_getopt
0.0	13.33	0.00	3	0.00	0.00	atexit
0.0	13.33	0.00	3	0.00	0.00	atoi
0.0	13.33	0.00	3	0.00	0.00	get_mem
0.0	13.33	0.00	3	0.00	0.00	strchr
0.0	13.33	0.00	3	0.00	0.00	strcmp

Table C-2: Sample TDD *gprof* Profiling Session

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Process Name
0.0	13.33	0.00	2	0.00	0.00	_fflush_u
0.0	13.33	0.00	2	0.00	0.00	_sbrk
0.0	13.33	0.00	2	0.00	0.00	_sbrk_unlocked
0.0	13.33	0.00	2	0.00	0.00	free_mem
0.0	13.33	0.00	1	0.00	0.00	_atexit_init
0.0	13.33	0.00	1	0.00	0.00	_cleanup
0.0	13.33	0.00	1	0.00	0.00	_exithandle
0.0	13.33	0.00	1	0.00	0.00	_fflush_u_iops
0.0	13.33	0.00	1	0.00	0.00	_lseek64
0.0	13.33	0.00	1	0.00	0.00	_malloc_unlocked
0.0	13.33	0.00	1	0.00	0.00	_morecore
0.0	13.33	0.00	1	0.00	0.00	_profil
0.0	13.33	0.00	1	0.00	0.00	cleanfree
0.0	13.33	0.00	1	0.00	0.00	encode_rs
0.0	13.33	0.00	1	0.00	0.00	exit
0.0	13.33	0.00	1	0.00	0.00	fflush
0.0	13.33	0.00	1	0.00	0.00	gen_poly
0.0	13.33	0.00	1	0.00	0.00	generate_gf
0.0	13.33	0.00	1	0.00	0.00	init_rs
0.0	13.33	0.00	1	0.00	9380.00	main
0.0	13.33	0.00	1	0.00	0.00	malloc
0.0	13.33	0.00	1	0.00	0.00	mem_init
0.0	13.33	0.00	1	0.00	0.00	realfree

C.2 Frequency-Domain Profiling Session

The following profiling session was obtained using the *gprof* program.

granularity: each sample hit covers 4 byte(s) for 0.01% of 152.90 seconds

Table C-3: Sample FDD *gprof* Profiling Session

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Process Name
40.9	62.53	62.53	688.02M	0.00	0.00	.umul
40.1	123.77	61.24	–	–	–	internal_mcount
14.6	146.08	22.31	10000	2.23	8.12	inv_trans
2.6	150.00	3.92	–	–	–	_mcount
0.8	151.28	1.28	10000	0.13	0.13	E_trans
0.7	152.35	1.07	10000	0.11	0.47	syndrome
0.3	152.80	0.45	10000	0.04	0.04	euclid
0.0	152.85	0.05	10000	0.01	0.01	erasure_init
0.0	152.89	0.04	10000	0.00	0.00	eras_dec_rs
0.0	152.90	0.01	10000	0.00	0.00	_memcmp
0.0	152.90	0.00	20030	0.00	0.00	_return_zero
0.0	152.90	0.00	20000	0.00	0.00	_memset
0.0	152.90	0.00	10015	0.00	0.00	_mutex_lock
0.0	152.90	0.00	10015	0.00	0.00	_memcpy
0.0	152.90	0.00	10000	0.00	0.00	_free_unlocked
0.0	152.90	0.00	10000	0.00	0.00	free
0.0	152.90	0.00	503	0.00	0.00	_randomjunk
0.0	152.90	0.00	503	0.00	0.00	random
0.0	152.90	0.00	254	0.00	0.00	.rem
0.0	152.90	0.00	3	0.00	0.00	atexit

Table C-3: Sample FDD *gprof* Profiling Session

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Process Name
0.0	152.90	0.00	3	0.00	0.00	get_mem
0.0	152.90	0.00	2	0.00	0.00	_fflush_u
0.0	152.90	0.00	2	0.00	0.00	_sbrk
0.0	152.90	0.00	2	0.00	0.00	_sbrk_unlocked
0.0	152.90	0.00	1	0.00	0.00	free_mem
0.0	152.90	0.00	1	0.00	0.00	_atexit_init
0.0	152.90	0.00	1	0.00	0.00	_cleanup
0.0	152.90	0.00	1	0.00	0.00	_exithandle
0.0	152.90	0.00	1	0.00	0.00	_fflush_u_iops
0.0	152.90	0.00	1	0.00	0.00	_getopt
0.0	152.90	0.00	1	0.00	0.00	_lseek64
0.0	152.90	0.00	1	0.00	0.00	_malloc _unlockd
0.0	152.90	0.00	1	0.00	0.00	_morecore
0.0	152.90	0.00	1	0.00	0.00	_profil
0.0	152.90	0.00	1	0.00	0.00	cleanfree
0.0	152.90	0.00	1	0.00	0.00	encode_rs
0.0	152.90	0.00	1	0.00	0.00	exit
0.0	152.90	0.00	1	0.00	0.00	fflush
0.0	152.90	0.00	1	0.00	0.00	gen_poly
0.0	152.90	0.00	1	0.00	0.00	generate_gf
0.0	152.90	0.00	1	0.00	0.00	init_rs
0.0	152.90	0.00	1	0.00	87740.00	main
0.0	152.90	0.00	1	0.00	0.00	malloc

Table C-3: Sample FDD *gprof* Profiling Session

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Process Name
0.0	152.90	0.00	1	0.00	0.00	mem_init
0.0	152.90	0.00	1	0.00	0.00	realfree