A VLSI SYNTHESIS OF A REED-SOLOMON PROCESSOR FOR DIGITAL COMMUNICATION SYSTEMS

PHILEMON JOHN CHOSE

A VLSI Synthesis of a Reed-Solomon Processor for Digital Communication Systems

By

©Philemon John Chose, B.Eng.

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Engineering

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

July, 1998

St. John's Newfoundland Canada

Abstract

The Reed-Solomon codes have been widely used in digital communication systems such as computer networks, satellites, VCRs, mobile communications and high-definition television (HDTV), in order to protect digital data against erasures, random and burst errors during transmission. Since the encoding and decoding algorithms for such codes are computationally intensive, special purpose hardware implementations are often required to meet the real time requirements.

One motivation for this thesis is to investigate and introduce reconfigurable Galois field arithmetic structures which exploit the symmetric properties of available architectures. Another is to design and implement an RS encoder/decoder ASIC which can support a wide family of RS codes.

An m-programmable Galois field multiplier which uses the standard basis representation of the elements is first introduced. It is then demonstrated that the exponentiator can be used to implement a fast inverter which outperforms the available inverters in $GF(2^m)$. Using these basic structures, an ASIC design and synthesis of a reconfigurable Reed-Solomon encoder/decoder processor which implements a large family of RS codes is proposed. The design is parameterized in terms of the block length n, Galois field symbol size m, and error correction capability t for the various RS codes. The design has been captured using the VHDL hardware description language and mapped onto CMOS standard cells available in the 0.8-μm BiCMOS design kits for Cadence and Synopsys tools. The experimental chip contains 218,206 logic gates and supports values of the Galois field symbol size $m = 3, 4, 5, 6, 7, 8$ and error correction capability $t = 1, 2, 3, ..., 16$. Thus, the block length n is variable from 7 to 255. Error correction t and Galois field symbol size

m are pin-selectable.

Since low design complexity and high throughput are desired in the VLSI chip, the algebraic decoding technique has been investigated instead of the time or transform domain. The encoder uses a self-reciprocal generator polynomial which structures the codewords in a systematic form. At the beginning of the decoding process, received words are initially stored in the first-in-first-out (FIFO) buffer as they enter the syndrome module. The Berlekemp-Massey algorithm is used to determine both the error locator and error evaluator polynomials. The Chien Search and Forney's algorithms operate sequentially to solve for the error locations and error values respectively. The error values are exclusive or-ed with the buffered messages in order to correct the errors, as the processed data leave the chip.

Acknowledgements

I would like to take this opportunity to thank my supervisors Dr. P. Gillard, Dr. R. Venkatesan and Dr. R. Donnelly whose constant encouragement and suggestions kept me on the right track.

The enthusiastic support from Dr. J.J. Sharp, Associate Dean of Graduate Studies and Research, and Dr. R. Seshadri, Dean of Engineering and Applied Science, is greatly appreciated. I would also like to thank Mr. Michael Rendell of the Computer Science Department and Mr. Brent Veitch of the Canadian Microelectronics Corporation for their help with the Synopsys and Cadence suite of software tools.

The financial support of the Faculty of Engineering and Applied Science of Memorial University of Newfoundland, and Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

I am especially indebted to my wife, Michele, for her love and emotional support during my graduate studies.

Contents

Abstract i

Acknowledgments iii

Table of Contents iv

List of Figures vi

List of Tables viii

List of Symbols and Acronyms ix

1 Introduction 1

1.1 Statement of the Problem 1

1.2 VLSI Architectures for Implementing Galois Field Arithmetic . . . 3

1.2.1 An Overview of Galois Field Arithmetic 3

1.2.2 Multipliers 6

1.2.3 Dividers and Inverters 10

1.2.4 Exponentiators 13

1.2.5 Summary 16

1.3 Scope of the Work 16

1.4 Organization of the Thesis 17

2 Theoretical Background on Reed-Solomon Codes **19**

2.1 General RS Code Definition 19

2.2 Encoding 21

2.3 Algebraic Decoding 23

2.3.1 Berlekamp-Massey Algorithm 27

2.3.2 Euclid's Algorithm 29

2.3.3 Chien Search 31

2.3.4 The Forney Algorithm 31

2.4 Time-Domain Decoding 34

2.4.1 Error Locator and Evaluator Polynomials 34

2.4.2 Error Evaluation 34

2.5 Error Correction 35

2.6 Algebraic vs. Time-Domain Decoding Algorithms 35

2.7 RS Encoder/Decoder Architectures 36

2.8 Summary 41

3 Proposed VLSI Arithmetic Architectures **42**

3.1 m-Programmable Galois Field Multiplier 43

3.2 m-Programmable Exponentiator/Inverter 58

3.3 Discussion and Summary 67

4 Synthesis of the Reed-Solomon Encoder/Decoder ASIC **68**

4.1 Design Flow, Functional Verification and Test 68

4.2 Chip Architecture 71

4.3 Modules 75
4.3.1 Encoder 75
4.3.2 Syndrome 78
4.3.3 Berlekamp 82
4.3.4 Error Magnitude Evaluation 86
4.3.4.1 The Chien Search 86
4.3.4.2 The Forney Algorithm 88
4.3.5 Error Correction and Verification 92
4.3.6 First-In-First-Out Buffer 94
4.3.7 Finite State Machine 96
4.4 Testing and Results 101
4.5 Discussion and Summary 108

5 Conclusion and Future Work **111**
5.1 Galois Field Arithmetic Architectures 111
5.2 VLSI Reed-Solomon Encoder/Decoder 112
5.3 Future Work 114

References **116**

List of Figures

2.1 A Digital Communication System 20

2.2 Algebraic Decoder 33

2.3 Time-Domain Decoder 35

3.1 A Parallel-In-Parallel-Out Multiplier for $GF(2^8)$ 45

3.2 Symbolic Architecture of the Programmable Multiplier 57

3.3 A General Exponentiation Architecture 61

3.4 Exponentiation/Inverse Architecture for $GF(2^8)$ 62

3.5 Symbolic Architecture of the Programmable Exponentiator/Inverter 64

4.1 Design Flow 69

4.2 Symbolic diagram of the RS Encoder/Decoder 73

4.3 RS Encoder 76

4.4 Symbolic diagram of the RS encoder 77

4.5 Systolic Array to compute Syndrome Polynomial 79

4.6 Symbolic diagram of the Syndrome Module 81

4.7 Symbolic diagram of the Berlekamp Module 85

4.8 The Chien Search Hardware 87

4.9 Error Evaluation Polynomial Circuit 89

4.10 Derivative Circuit 90
4.11 Block Diagram of the Error Magnitude Evaluation 91
4.12 Symbolic diagram of the Error Magnitude Evaluation 93
4.13 Symbolic diagram of FIFO 95
4.14 RS Encoder/Decoder Finite State Machine 98
4.15 Symbolic diagram of the Finite State Machine 100
4.16 Gate Level Simulations of the RS Encoder 103
4.17 Gate Level Simulations of the Syndrome Module 104
4.18 Gate Level Simulations of the overall RS ASIC (Encoding) 105
4.19 Gate Level Simulations of the overall RS ASIC (Decoding) 106
4.20 Gate Level Simulations of the RS ASIC (Decoding Failure) 107

List of Tables

1.1 Elements generated by $P(x) = 1 + x + x^3$ 6

3.1 Comparison of the programmable Unpipelined and Pipelined Multiplier 58

3.2 Features of the programmable Exponentiator/Inverter 63

3.3 Comparison of Exponentiator/Inverter with other Inverters for fixed m 66

4.1 RS Encoder/Decoder Characteristics 72

4.2 RS Encoder/Decoder I/O Pins 74

4.3 Encoder I/O Pins 78

4.4 Syndrome I/O Pins 82

4.5 Berlekamp Module I/O Pins 86

4.6 I/O Pins of the Error Magnitude Evaluation 92

4.7 I/O Pins of the FIFO Module 94

4.8 I/O Pins of the FSM Module 99

4.9 RS Encoder/Decoder Data Rates 109

4.10 RS Encoder/Decoder Modules and Equivalent Gate Count 110

5.1 Projected RS Encoder/Decoder Data Rates 115

List of Symbols and Acronyms

ASIC	Application specific integrated circuit
BCH	Bose-Chaudhuri-Hocquenghem
BiCMOS	Bipolar complementary metal oxide semiconductor
BJT	Bipolar junction transistor
$C(x)$	Codeword polynomial
CAD	Computer aided design
CMOS	Complementary metal oxide semiconductor
CMC	Canadian Microelectronics Corporation
CODEC	Coding and decoding
DDA	Division-and-accumulation
$E(x)$	Error polynomial
FEC	Forward error correction
FIFO	First-in first-out
FSM	Finite state machine
GF	Galois field
$G(x)$	Generator polynomial
HDTV	High-definition television
I/O	Input output
IC	Integrated circuit

k	Number of message symbols
m	Symbol size of a Galois field element
$M(x)$	Message polynomial
MUX	Multiplexer
n	Code length of an RS code
NASA	National Aeronautics and Space Administration
NMOS	$n-$ type metal oxide semiconductor
$p(x)$	Primitive polynomial
ROM	Read-only memory
RS	Reed-Solomon
RTL	Register transfer level
$S(x)$	Syndrome polynomial
t	Error correcting capability of an RS code
VHDL	Very high speed integrated circuit hardware description language
VLSI	Very large scale integration
$\Lambda(x)$	Error locator polynomial
$\Lambda'(x)$	First derivative of the error locator polynomial
$\sigma(x)$	Error locator polynomial
$\Omega(x)$	Error locator polynomial

Chapter 1

Introduction

1.1 Statement of the Problem

The Reed Solomon (RS) codes are widely used in digital communication systems to increase the reliability and efficiency of the communication channel. They work in finite or Galois field arithmetic and have the ability to efficiently protect digital data against erasures, random, and burst errors during transmission. The interest in the RS codes was primarily theoretical until the concept of concatenated coding which uses a convolutional code/RS code channel system, was formulated and first introduced in [1]. Their success is now reflected in modern day digital audio or compact discs, computer networks, deep space telecommunication systems, spread-spectrum systems, computer memories, VCRs and high-definition television (HDTV) applications [2][3][4].

Since the encoding and decoding algorithms for such codes are computationally intensive, special purpose hardware implementations are often required to meet the real-time processing requirements. The choice of a specific RS code depends on the characteristics of the communication channel. As such, RS encoders and decoders are traditionally designed with fixed values of the error correction capability t, block

length n and symbol size m. The reason for choosing fixed design parameters is that the exponentiator, multiplier, divider and inverter have different designs for different values of m [2][5][6]. Such an approach is evidently inflexible and hence inefficient because the system has to be redesigned if the channel characteristics ultimately change. Moreover, the design complexity increases with the error correction capability of the code, thus making it impractical to implement the system using off-the-shelf discrete integrated circuit components. However, rapid advances in VLSI technologies may offer attractive solutions because of higher reliability, better performance, smaller area, lighter weight and lower power consumption [7].

Hence, there is a direct need for fast finite field arithmetic circuits which operate in $GF(2^m)$, where m is variable. The ability to operate with different symbol sizes of m-bits has been a limiting factor in past efforts to implement universal and possibly reconfigurable RS hardware. If such arithmetic circuits could be developed, it would be possible to design highly efficient single chip encoders/decoders whose total design cost is amortized over a wide application base. Hence, the motivations for this thesis are:

(1) to investigate and introduce programmable Galois field arithmetic structures which exploit the symmetric properties of available architectures. It appears that very little work has been done in the literature to develop reconfigurable hardware which can operate in finite fields.

(2) to design and implement an RS encoder/decoder ASIC which can support a wide family of RS codes whose symbol size m and error correction capability t can be varied directly in hardware. Such a general-purpose ASIC would be suitable for a wide variety of digital communication systems which require different RS codes.

The choice of m and t depends on the application and is usually based on the overall correction performance and throughput of the code. The specific Galois field symbol size $m = 8$ has been standardized by the European Space Agency and the National Aeronautics and Space Administration for satellite communication [2]. The error correction circuits for advanced train control systems, mobile radio systems, magnetic recording systems, data communications and digital signal processing based modems use $m = 5$ [2][5].

1.2 VLSI Architectures for Implementing Galois Field Arithmetic

The following subsections present an overview of Galois field arithmetic and a literature review of the various arithmetic operations.

1.2.1 An Overview of Galois Field Arithmetic

Recently, Galois fields or finite fields have received great attention because of their widespread applications in error control coding using linear block codes. They have also been extensively used in digital signal processing, pseudo-random number generation, encryption and decryption protocols in cryptography. The design of efficient multiplier, inverter and exponentiation circuits for Galois field arithmetic is needed for these applications. These circuits should have low complexity, short computation delay and low latency when used in high-performance systems [2].

A finite field or a Galois field designated $GF(p)$, is a finite set of elements which has defined rules for arithmetic. These rules are not algebraically different from those used in arithmetic with ordinary numbers except that there is only a finite

set of elements involved. All finite fields have the following properties:

1. Multiplication and addition are the two operations defined for combining the elements.

2. The result of adding or multiplying two elements is always a third element contained in the field.

3. The field always contains the multiplicative identity element 1 and the additive identity element 0 such that $a+0=a$ and $a \cdot 1 = a$ for any element a.

4. Every element a has an additive inverse element $(-a)$ and a multiplicative inverse element a^{-1} such that $a+(-a)=0$ and $a \cdot a^{-1} = 1$. The existence of these elements permits subtraction and division to be performed.

5. The associative $[a+(b+c)=(a+b)+c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c]$, commutative $[a+b=b+a$ and $a \cdot b = b \cdot a]$, and distributive $[a.(b+c) = a \cdot b + a \cdot c]$ laws apply.

$GF(p^m)$ is an extension field of the ground field $GF(p)$, where m is a positive integer. For $p=2$, $GF(2^m)$ is an extension field of the ground field $GF(2)$ of two elements {0,1}. $GF(2^m)$ is a vector space of dimension m over $GF(2)$ and hence is represented using a basis of m linearly independent vectors. The finite field $GF(2^m)$ contains $2^m - 1$ non-zero elements. All finite fields contain a zero element and an element, called a generator or primitive element, such that every non-zero element in the field can be expressed as a power of this element.

In order to introduce the mathematical concepts of the trace and dual basis, the following definitions are necessary [8][9].

Definition 1: The trace of an element β which belongs to $GF(2^m)$ is defined as

$$Tr(\beta) = \sum_{k=0}^{m-1} (\beta)^{2^k} \tag{1.1}$$

Definition 2: A basis $\{\mu_j\}$ in $GF(2^m)$ is a set of m linearly independent elements

in $GF(2^m)$, where $0 \leq j \leq m-1$.

Definition 3: Two bases $\{\mu_j\}$ and $\{\lambda_k\}$ are the dual of one another if

$$\begin{aligned} Tr(\mu_j \lambda_k) &= 1, \quad \text{if } j = k \\ &= 0, \quad \text{if } j \neq k \end{aligned} \tag{1.2}$$

where $0 \leq j \leq m-1$ and $0 \leq k \leq m-1$.

The elements of $GF(2^m)$ are usually expressed as powers of the primitive element α, where α is defined as the root of the primitive polynomial $P(x) = x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} ... + f_1 x + 1$ where $f_i \epsilon \{0,1\}$. Each element z of $GF(2^m)$ can also be written in [8][9][10]

- the standard basis as $z = a_{m-1}\alpha^{m-1} + ... + a_2\alpha^2 + a_1\alpha^1 + a_0$.
- the normal basis as $z = a_{m-1}\alpha^{2^{m-1}} + ... + a_2\alpha^{2^2} + a_1\alpha^{2^1} + a_0\alpha$.
- the dual basis λ_k as $z = \sum_{k=0}^{m-1} z_k \lambda_k = \sum_{k=0}^{m-1} Tr(z\mu_k)\lambda_k$.

where

$a_i \epsilon GF(2)$ and $z_k = Tr(z\mu_k)$ is the k-th coefficient of the dual basis.

The standard basis is commonly used in implementing algebraic Reed-Solomon decoders in hardware. Since multiplication is the most dominant arithmetic operation, the standard basis multiplier is often preferred for its lowest design complexity compared to the normal and dual based multipliers [6][11]. It does not require basis conversion and thus can be more easily matched to any input/output system.

As an example, the power, polynomial and 3-tuple representations of the Galois field elements generated by the primitive polynomial $P(x) = 1 + x + x^3$ are tabulated in Table 1.1. The non-zero elements are generated using a 3-stage linear feedback shift register initialized to 001, with taps defined by the coefficients of the primitive polynomial $P(x)$.

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

Table 1.1: Elements generated by $P(x) = 1 + x + x^3$

Examples:

Addition: $\alpha^1 + \alpha^2 = \alpha^4$

Multiplication: $\alpha^2 \cdot \alpha^3 = \alpha^5$

Division: $\frac{\alpha^5}{\alpha^2} = \alpha^3$

Exponentiation: $(\alpha^6)^3 = \alpha^{18} = \alpha^4$

Inversion: $\frac{1}{\alpha^2} = \alpha^{-2} = \alpha^{7-2} = \alpha^5$

since $\alpha^7 = \alpha^0 = 1$

Addition and subtraction in finite fields are relatively straightforward, but multiplication, division, exponentiation and inversion are not. Using a symbol size m, addition and subtraction can be realized using m-bit exclusive-or gates. However, since the more complex operations are extensively used in RS encoding and decoding algorithms, the development of their hardware structures have received considerable attention.

1.2.2 Multipliers

For arbitrary elements $A(x) = \sum_{k=0}^{m-1} a_k x^k$, $B(x) = \sum_{k=0}^{m-1} b_k x^k$ in $GF(2^m)$, and the primitive polynomial $P(x) = \sum_{k=0}^{m-1} p_k x^k$, the product $C(x)$ of $A(x)$ and $B(x)$ is

given by

$$\begin{aligned} C(x) &= A(x)B(x) \ mod \ P(x) \\ &= [\sum_{k=0}^{m-1} A(x)b_k x^k] \ mod \ P(x) \\ &= (...(A(x)b_{m-1}x + A(x)b_{m-2})x + ...)x + A(x)b_0 \\ &= c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + ... + c_1 x + c_0 \end{aligned} \tag{1.3}$$

A direct implementation of multiplication by combinational logic was proposed by Bartee and Schneider [12]. A canonical basis is used to represent the elements of the field. Depending on the primitive element, this implementation requires as many as $(m^3 - m)$ two-input adders over $GF(2)$. This approach has a high circuit complexity and also lacks regularity suited for full custom VLSI designs.

A cellular array multiplier was originally conceived by Laws and Rushforth in 1971 [13]. The array requires approximately $2m$ gate delays, a considerable improvement over the traditional linear feedback register type multiplier which computes the desired product sequentially in m clock cycles. A simple parity check circuit is incorporated in the design.

In 1984, Yeh *et al* [14] presented systolic multipliers for performing multiplication of arbitrary elements in $GF(2^m)$ in $O(m)$ time and area suitable for VLSI implementation. In the design, the elements in the field are represented in the conventional manner. The throughput rate for the serial-in serial-out one-dimensional systolic array is m clock cycles and the parallel-in parallel-out two-dimensional systolic array, one clock cycle. Both designs have a latency of $2m$ clock cycles.

In 1985, Wang *et al* [10] developed a pipeline structure to implement the multiplication algorithm proposed by Massey and Omura for Galois fields based on the normal basis representation. By taking advantage of the squaring property of the normal basis representation, the same pipeline structure is reconfigured to compute the inverse elements in $GF(2^m)$. The throughput rate for the multiplier is one

product per clock cycle after an initial delay of m clock cycles. Since the design is dependent on the primitive polynomial used to generate the field elements, the number of XOR gates in the product function increases enormously for large m. Hence, the pipeline structure is only practical for small m.

In 1986, Scott *et al* [15] presented a bit-slice architecture of a serial-in serial-out multiplier well suited for VLSI implementation. The multiplier has a latency of m clock cycles and yields a computation time and implementation area of $O(m)$. It is shown that the architecture is attractive for use in data encryption systems where data are segmented into long blocks to achieve high security and maximum throughput.

A parallel-in parallel-out systolic array and a serial-in serial-out systolic array proposed for fast multiplication in the finite fields $GF(2^m)$ with the standard basis representation were presented by Wang and Lin in 1991 [11]. The architectures are regular, concurrent and have unidirectional data flow. A system with unidirectional data flow is highly desirable when designing high-speed VLSI systems. It is further shown that the proposed parallel implementation can more easily incorporate fault-tolerance compared to previously published designs. The serial-in serial-out array only requires one control signal instead of two as in [14]. If the input data pass in continuously, the parallel-in parallel-out array yields output results at a rate of one output per clock cycle after a latency of $3m$ cycles. It is worth noting that the minimum clock period is governed by the propagation delay of an AND gate in series with an XOR gate. All the operations of each basic cell are pipelined in such a manner that each cell performs a small fraction of the multiplication and passes the data to the neighbouring cells for further processing. Under the same

operating conditions, the serial-in serial-out array yields output results at a rate of one per m cycles after an initial delay of $3m$ cycles.

A bit-serial systolic divider circuit and multiplier over $GF(2^m)$ was presented by Hasan and Bhargava in 1992 [16]. The design is based on the Gauss-Jordan Elimination algorithm and completely eliminates global data communications and dependency of the time step duration on m. The division algorithm requires the formulation of the supporting elements and the corresponding coefficient matrix by using a one-dimensional systolic array. The resulting system of $2m-1$ simultaneous linear equations in $2m-1$ unknowns are solved using a two-dimensional systolic array. With minor modifications, the same structure is used to perform multiplication over $GF(2^m)$ in a computational time of $3m-1$ time steps. The proposed inverter/divider requires three processors and a control signal consisting of $2.5m^2+11.5m-6$ registers, $4m^2+12m-5$ AND gates, $1.5m^2+7.5m-2$ OR gates, and $0.5m^2+1.5m-1$ XOR gates. The structure has a computational time of $5m-1$ time steps and is independent of the irreducible polynomial.

A division and bit-serial multiplication algorithm were presented by Hasan and Bhargava in 1992 [17]. Using the coordinates of supporting elements, division over $GF(q^m)$ is performed by solving a system of m linear equations over $GF(q)$ when the field elements are represented by polynomials. It is further shown that division can be performed with a lower order of computational complexity by solving a Wiener-Hopf equation of degree m. The discrete-time Wiener-Hopf equation is defined as a system of m linear inhomogeneous equations, with m unknowns [17].

Structures for parallel multipliers derived from irreducible all-one and equally spaced polynomials were developed by Hasan *et al* in 1992 [18]. It is shown that the

three basis modules of an all-one polynomial based parallel multiplier of a small field can be used to construct all the corresponding equally spaced polynomials of larger fields. A normal basis parallel-type multiplier for finite fields $GF(2^m)$ generated by the irreducible all-one polynomials was recently presented by Hasan *et al* in 1993 [19]. It is a modified version of the Massey-Omura multiplier.

A systolic power-sum circuit designed to implement the function AB^2+C where A, B and C are elements of the field was presented by Wei in 1994 [20]. By adding one multiplexer and one demultiplexer, the power-sum circuit is configured to compute eight different types of computations viz $AB, AB+C, A^2, A^2+C, AB^2, AB^2+C, A^3$ and A^3+C. All these computations are needed in decoding multiple error correcting BCH and Reed-Solomon codes in cases where the coefficients of the error locator polynomial are solved algebraically.

A bit-serial multiplier which has the same hardware requirements as the traditional Berlekamp multiplier was recently presented by Fenn *et al* in 1995 [21]. In the design, the variable multiplier is represented over the dual basis and the constant multiplicand is represented over the polynomial basis. The reverse is true with a constant traditional Berlekamp multiplier. It is shown that constant multipliers based on the proposed approach can operate at a higher frequency than those based on the traditional Berlekamp multiplier.

1.2.3 Dividers and Inverters

Finding the inverse of an element over $GF(2^m)$ is computationally intensive in hardware and still remains an active area of research. Finite field inversion and division are critical in decoding Reed-Solomon and BCH codes. During the de-

coding process, the Berlekamp-Massey and Forney algorithms often employ these arithmetic operations. The derived algorithms for decoding double error-correcting Reed-Solomon codes require the same functions as well. Thus, the latency and throughput of the inverters and dividers may dictate the overall speed of the decoder.

The traditional method for computing the inverse of elements in $GF(2^m)$ uses read-only memory (ROM), Fermat's theorem or Euclid's algorithm. The size of the ROM is $m2^m$ bits. The coordinates of an element are used as the address of the location in the ROM where the corresponding inverse is stored. The value of m can range from 3 to infinity. These methods are inefficient for VLSI implementation if large values of m are required. In recent years, several algorithms and their corresponding VLSI architectures for computing the inverse elements have been presented in the literature. For an arbitrary element A in the finite field $GF(2^m)$, the inverse operation of an element A is denoted by $A^{-1} = A^{2^m-2}$. Rewriting the exponent $2^m - 2$ as $2^1 + 2^2 + 2^3 + ... + 2^{m-1}$, allows the inverse operation to be expressed as [10]

$$A^{-1} = (A^2) \cdot (A^{2^2}) \cdot (A^{2^3}) \cdots (A^{2^{m-1}}) \tag{1.4}$$

In 1985, Wang *et al* [10] invented a parallel-in serial-out circuit for solving Equation (1.4) based on the Massey-Omura multiplier. In their design, the normal basis representation of the elements in the form $(\alpha^{2^0}, \alpha^{2^1}, \alpha^{2^2}, ..., \alpha^{2^{m-1}})$ is used. The method is impractical for large values of m since the number of XOR gates in the product function correspondingly becomes large. Since squaring is a cyclic shift operation in the normal basis, the inverse function is found in m clock cycles.

In 1989, Feng [22] developed a serial-in parallel-out architecture based on the normal basis representation of the finite elements. The algorithm requires a computational complexity of $O(m \log_2 m)$. A throughput rate and latency of $m(q+p)$ clock cycles, where p is the number of ones in the binary expression of $m-1$ and q is the lower bound on $\log_2 m$, are needed to compute the inverse elements.

In 1993, Wang and Li [23] presented a serial-in serial-out systolic array architecture for performing the inverse element in $GF(2^m)$. In the analysis, the standard basis representation of the field elements is used. The design for $GF(2^m)$ mimics the systolic array based on the Gauss-Jordan elimination algorithm for solving a system of $2m-1$ linear equations over $GF(2)$ [24]. The proposed inversion circuitry has a latency of $7m-3$ clock cycles and a maximum throughput rate of $2m-1$ clock cycles. Without any modifications in hardware, the multiply-and-divide operation can easily be performed. The logic design of the architecture is independent of the primitive polynomials used to generate the field elements. All the operations of the serial-in serial-out systolic array are pipelined in such a manner that each cell completes a small fraction of the computations and passes the data to the neighbouring cells. The entire systolic array is made up of $\frac{(m^2+m)}{2}$ main array cells and m boundary cells, where m is the size of the Galois field.

A fast normal basis inversion circuitry was presented by Fenn *et al* in 1996 [25]. The hardware scheme uses two registers, a multiplier, a squarer and a generator device in $GF(2^m)$. It exploits the properties of Fermat's theorem in order to progressively generate the solution in approximately $\frac{m}{2}$ clock cycles. The inverter is shown to be more efficient for odd values of m and its features make it suitable for double error-correcting Reed-Solomon codes. The same design was recently reex-

amined and improved by Yen in 1997 [26]. It is demonstrated that the number of clock cycles per iteration can be further reduced to around $\frac{m}{3}$. Yen's algorithm clearly outperforms the algorithm by Fenn *et al* for large values of m. Another modification to the algorithm by Fenn *et al* was reported by Calvo and Torres in 1997 [27]. The generator and squarer devices have been totally eliminated from the original circuit.

In 1997, Hasan [28] presented an algorithm to perform sequential computation of division-and-accumulation (DDA) over $GF(2^m)$. The algorithm can also be used for the conventional rational numbers. It is shown that in the cases where n multiplications and n inversions are required in the DDA, the new algorithm only requires $3n+1$ multiplications and one inversion. Such a proposition is advantageous to fields where a division operation is at least three times more complex than a multiplication. The DDA structure is suitable for the systolic Reed-Solomon encoder [29] to efficiently compute the parity symbols during the encoding process.

1.2.4 Exponentiators

Exponentiation is extensively used in cryptosystems and error-correcting codes. The conventional approach for finding the exponent of an element in $GF(2^m)$ uses read-only memory or table lookup. The value of m can range from 3 to infinity, which would require storing 2^m elements of m-bit wide. This method is inefficient when m becomes too large. In recent years, several exponentiation algorithms and their corresponding VLSI architectures have been proposed.

For an arbitrary element β in the finite field $GF(2^m)$ and an integer $N(1 \leq N \leq 2^m - 1)$, the exponentiation function is defined as $\delta = \beta^N$. Clearly, δ is in $GF(2^m)$.

If N is represented in binary form as $n_0, n_1, n_2, ..., n_{m-1}$ such that $N = \sum_{i=0}^{m-1} n_i \cdot 2^i$, then $\delta = \beta^N$ can be expressed as follows [30][31]

$$\begin{aligned} \delta &= \beta^N = \beta^{\sum_{i=0}^{m-1} n_i.2^i} \\ &= (\beta)^{n_0} \cdot (\beta^2)^{n_1} \cdot (\beta^{2^2})^{n_2} \cdots (\beta^{2^{m-1}})^{n_{m-1}} \\ &= \prod_{i=0}^{m-1} (\beta^{2^i})^{n_i} \\ &= \prod_{i=0}^{m-1} E_i \end{aligned} \tag{1.5}$$

where

$$E_i = \beta^{2^i} \quad \text{if } n_i = 1 \tag{1.6}$$

$$E_i = 1 \quad \text{if } n_i = 0 \tag{1.7}$$

In 1988, Scott *et al* [32] proposed several sequential and parallel VLSI architectures for computing the product terms of the exponent in $GF(2^m)$. As described in the reference [32], the designs are targeted for applications that use Galois fields $GF(2^m)$ for large values of m. Both the standard and normal based exponentiations are considered. The sequential exponentiation unit requires $O(m^2)$ clock cycles assuming repeated use of a multiplier which possesses a throughput rate of one multiplication every m clock cycles. The fully parallel computation of the product terms yields one exponentiation per m clock cycles, assuming the use of $(m-1)$ multipliers whose combined minimum latency is $m+2m\log_2 m$ clock cycles. A multiplier latency of $2m$ clock cycles is assumed.

A VLSI design and implementation of an exponentiation circuit was also presented by Wang and Pei in 1990 [30]. The architecture can be used to generate pseudorandom number sequences in spread spectrum, cryptographic systems and digital signal processing applications such as noise generation. Elements in the finite field are represented in the normal basis. In this design, the exponentiation of

an element is found in m clock cycles. The architectural details and VLSI layout of the chip for $GF(2^4)$ are extensively illustrated.

In 1993, Arazi [33] presented two efficient exponentiation circuits which can be adopted for smartcard applications. They operate over the standard basis representation of elements in $GF(2^m)$. In one scheme, the algorithm is completed in $2m$ clock cycles instead of m. The shift registers can be implemented with dynamic instead of static registers, owing to the limited space in a smartcard-mounted chip. The second scheme is simpler and uses duplicates of the same cell to compute exponentiation in $6m^2$ clock cycles.

A parallel-in-parallel-out bit-level systolic array architecture with unidirectional dataflow for computing exponentiation was first presented by Wang in 1994 [31]. Using the systolic multiplier proposed by Wang and Li in the reference [11], two-level pipelining is employed to achieve a maximum throughput of one output every clock cycle after an initial delay of $2m^2+m$ cycles. Unidirectional dataflow is highly desirable in designing high-speed systems. The design can easily incorporate fault-tolerance.

An exponentiation algorithm based on a pattern matching and recognition technique was recently presented by Kovač and Rangathathan in 1996 [34]. Unlike the conventional methods which use repeated multiplications, the algorithm can perform the exponentiation operation on-the-fly. In the analysis, the nonzero elements of the Galois field $GF(2^m)$ are represented in the standard basis. The elements are divided into subsets, where each subset corresponds to a pattern. More details on the related theorems and proofs are given in the reference [34]. In an effort to obtain high speed and maximum throughput, a systolic architecture which uses a

multistage linear pipeline and parallelism is proposed by the authors. Once the pipe is filled, a new result is obtained every clock cycle following a latency of 2^m clock cycles. Thus, the architecture is recommended for applications that use $GF(2^m)$ for values of m less than or equal to eight. The hardware allows the programming of different primitive irreducible polynomials of degree m less than or equal to eight. The design issues related to the CMOS VLSI implementation of the chip which performs the exponentiation operation over Galois field $GF(2^4)$ are extensively enumerated. A maximum computational rate of 40 million exponentiations per second at a clock frequency of 40 MHz is possible.

1.2.5 Summary

An overview of Galois field arithmetic operations has been presented. The multiplication, inverse, division, and exponentiation operations in $GF(2^m)$ have been extensively described. The traditional method for evaluating these functions uses ROM, Fermat's theorem or Euclid's algorithm. However, these techniques are inefficient for VLSI implementation if large values of m are required. Thus, the latency and throughtput of the arithmetic units may dictate the overall speed of the global system. The development of more efficient algorithms and their corresponding VLSI architectures still remains an active area of research.

1.3 Scope of the Work

In this thesis I propose an m-programmable Galois field multiplier which uses the standard basis representation of the elements. A structure is also designed to implement both the exponent and inverse functions over $GF(2^m)$, where m is vari-

able. The ability to operate with different symbol sizes of m-bits wide has been a limiting factor in past attempts to implement universal and reconfigurable encoders/decoders [2][5][6].

By using the proposed arithmetic circuits, coupled with a multiplexing technique to select different RS code parameters m and t, an ASIC synthesis of a testable RS encoder/decoder which implements a wide family of RS codes in $GF(2^m)$ is developed. Unlike the chips which are customized for a specific m and t as reported in [35]-[51], it is reconfigurable and supports values of the Galois field symbol size $m = 3, 4, 5, 6, 7, 8$ and error correction capability t ranging from 1 to 16. This means the total cost of such a design is amortized over a wide application base. Since low design complexity and high throughput are desired in the experimental VLSI chip, the algebraic decoding technique is preferred over the time or transform domain methods.

Gate arrays, standard cells and full-custom are three potential VLSI technologies that could have been used to implement the RS encoder/decoder chip. However, a CMOS standard cell based design methodology, which uses hardware description language (HDL) logic synthesis, is found suitable because it allows easy mapping and optimization of the logic level design into integrated circuit (IC) layout using the state-of-the-art VLSI CAD tools. The design has been simulated at a frequency of 50 MHz and contains 218,206 logic gates.

1.4 Organization of the Thesis

The remaining chapters of the thesis are organized as follows:

In Chapter 2, the mathematical background and necessary theoretical details

are described for understanding Reed-Solomon codes.

Chapter 3 proposes an m-programmable Galois field multiplier which uses the standard basis representation of the elements. Using this multiplier, it is shown that the exponentiation and inverse operations can be both performed using the same reconfigurable hardware.

Chapter 4 discusses the design methodology, VLSI synthesis and operational features of a new programmable Reed-Solomon encoder/decoder processor.

Chapter 5 highlights the major conclusions of this research and recommendations for possible future work.

Chapter 2

Theoretical Background on Reed-Solomon Codes

In this chapter, the RS encoding and decoding algorithms are first explained. A survey on the existing RS encoder and decoder architectures usually designed for a fixed m is given.

2.1 General RS Code Definition

Discovered by I.S. Reed and G.S. Solomon in 1960, Reed-Solomon codes are an important subclass of nonbinary BCH codes. They are among the most versatile and powerful error control codes commonly used to correct both random and burst errors in digital communications and magnetic storage systems ranging from the digital audio disc to the *Voyager* spacecraft. A general block diagram of a digital communication system is shown in Figure 2.1.

The interest in RS codes was primarily theoretical until the concept of concatenated codes was formulated and first introduced by Forney in 1966 [1]. Concatenated coding has since been adopted by the U.S. National Aeronautics and Space Administration (NASA) for interplanetary space missions. It uses the con-

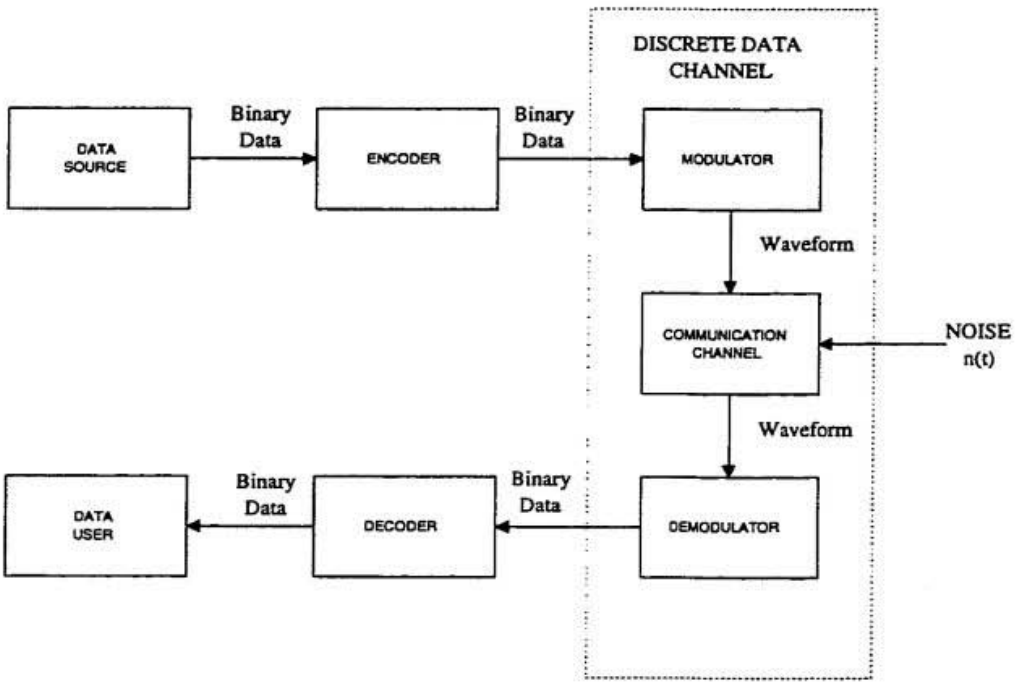

Figure 2.1: A Digital Communication System

volutional/RS channel encoding and decoding system.

For any positive integer $m \geq 3$ and error correcting capability $t \geq 1$, there exists a t-error correcting RS code from the Galois field $GF(2^m)$ with the following parameters [52]-[57]

Block Length $n = 2^m - 1$ symbols

Number of Parity Check $2t = n - k$ symbols

Minimum Distance $d_{min} = 2t + 1$

where k is the data message in symbols.

An (n, k, t) RS code has a generator polynomial $G(x)$ of degree $n - k$ often written as $G(x) = (x + \alpha)(x + \alpha^2)...(x + \alpha^{2t})$.

2.2 Encoding

The generator polynomial $G(x)$ of an RS code has the form

$$\begin{aligned} G(x) = \quad & \textstyle\sum_{i=b}^{2t+b-1}(x - \alpha^i) \\ & = \textstyle\sum_{i=0}^{2t} g_i x^i \\ & = g_0 + g_1 x + ... + g_{2t} x^{2t} \end{aligned} \tag{2.1}$$

where b is a nonnegative integer often chosen to be 1. The number of distinct coefficients of $G(x)$ can be reduced by almost half by carefully choosing $b = 2^{m-1} - t$ satisfying the relationship [8]

$$2b + 2t = 2^m \tag{2.2}$$

There are two ways to encode the message $M(x)$. In nonsystematic encoding, the codeword $C(x)$ is generated simply as

$$C(x) = M(x)G(x) \tag{2.3}$$

Thus the message $M(x)$ is not explicitly present in the codeword $C(x)$.

In systematic encoding, given a message polynomial $M(x)$ and generator polynomial $G(x)$, the codeword $C(x)$ is generated as follows:

(1) multiply the message $M(x)$ by x^{2t} to obtain $M(x)x^{2t}$

(2) divide $M(x)x^{2t}$ by $G(x)$ to obtain the remainder polynomial $\widehat{R}(x)$ and form the codeword $C(x)$

$$C(x) = x^{2t}M(x) + \widehat{R}(x) = Q(x)G(x) \tag{2.4}$$

where $Q(x)$ is the quotient and $\widehat{R}(x) = \widehat{r}_0 + \widehat{r}_1 x + \widehat{r}_2 x^2 + ... + \widehat{r}_{2t-1}x^{2t-1}$ is the remainder or parity polynomial.

Circuits for performing division by $G(x)$ or any arbitrary polynomial are available. The number of distinct multipliers $g_0, g_1, ..., g_{2t}$ can be reduced almost by half by choosing $b = 2^{m-1} - t$.

Maki and Owsley [58] presented the VLSI design and implementation of the parallel Berlekamp architecture which has the speed performance equivalent to the conventional, but at a hardware cost 8 times the serial Berlekamp architecture. The serial and parallel VLSI architectures by Berlekamp perform encoding in the dual or trace orthogonal basis representation of the field elements.

A transmitted codeword $C(x)$ may be corrupted in a noisy channel. The received polynomial $R(x)$ can be expressed as the sum of the transmitted codeword $C(x)$ and error polynomial $E(x)$ as

$$R(x) = C(x) + E(x) = r_{n-1}x^{n-1} + ... + r_1 x + r_0 \tag{2.5}$$

The following sections describe available techniques which can be used to find and correct the errors in the received polynomial $R(x)$.

2.3 Algebraic Decoding

The first task of an algebraic decoder is to determine the syndrome polynomial $S(x)$ based on $R(x)$. The coefficients of the syndrome polynomial are given by [54]

$$S_j = R(\alpha^j) = E(\alpha^j) = \sum_{i=0}^{n-1} r_i \alpha^{ij} \tag{2.6}$$

$1 \leq j \leq 2t$ for nonsymmetric coefficients of $G(x)$

or $2^{m-1} - t \leq j \leq 2^{m-1} + t - 1$ for symmetric coefficients of $G(x)$

After the evaluation of the syndromes, the error values $e_0, e_1, ..., e_{n-1}$ can be found. If v errors actually occur in $R(x)$, at the unknown locations $i_1, i_2, ..., i_v$, the error polynomial can be expressed as

$$E(x) = Y_1 x^{i_1} + Y_2 x^{i_2} + ... + Y_v x^{i_v} \tag{2.7}$$

where Y_l is the magnitude of the lth error at location i_l.

Prior to decoding, the values of v, $i_1, ..., i_v$ and $Y_1, ..., Y_v$ are initially unknown.

If X_l is the field element associated with the error location i_l, then the syndrome coefficients are given by

$$S_j = \sum_{i=1}^{v} Y_l X_l^i \tag{2.8}$$

for $j = 1, 2, ..., 2t$ or $j = 2^{m-1} - t, ..., 2^{m-1} + t - 1$

where Y_l is the error value and X_l is the error location of the lth error symbol.

An expansion of Equation (2.8) gives the following set of $2t$ simultaneous equations in v unknown error locations $X_1, ..., X_v$ and v unknown error magnitudes $Y_1, ..., Y_v$

$$S_1(x) = Y_1X_1 + Y_2X_2 + ... + Y_vX_v$$

$$S_2(x) = Y_1{X_1}^2 + Y_2{X_2}^2 + ... + Y_v{X_v}^2$$

$$S_3(x) = Y_1{X_1}^3 + Y_2{X_2}^3 + ... + Y_v{X_v}^3$$

.

.

.

$$S_{2t}(x) = Y_1{X_1}^{2t} + Y_2{X_2}^{2t} + ... + Y_v{X_v}^{2t}$$

The above set of equations must have at least one solution because of the way the syndromes are defined. This solution is unique. Thus, the decoder's task is to find the unknowns, given the syndromes. This is equivalent to a problem in solving a system of nonlinear equations.

Clearly, the direct solution of the system of nonlinear equations is too difficult for large values of v. Instead, intermediate variables can be computed using the syndrome coefficients S_j from which the error locations $X_1, ..., X_v$ can be determined. The error-locator polynomial is introduced as

$$\sum(x) = \Lambda(x) = \Lambda_v x^v + \Lambda_{v-1}x^{v-1} + ... + \Lambda_1 x + 1 \tag{2.9}$$

The polynomial is defined with roots at the error locations X_l^{-1} for $l = 1, 2, ..., v$. The error location numbers X_l indicate errors at locations i_l for $l = 1, 2, ..., v$. That is to say,

$$\Lambda(x) = \sum_{l=1}^{v}(1 - xX_l) = (1 - xX_1)(1 - xX_2)...(1 - xX_v) \qquad (2.10)$$

where $X_l = \alpha^{i_l}$.

To determine the coefficients of $\Lambda(x)$ from the syndromes, equate Equations (2.9) and (2.10) and multiply both sides by $Y_l X^{j+v}$ and set $x = X_l^{-1}$, i.e.,

$$Y_l X^{j+v}(\Lambda(x) = \sum_{l=1}^{v}(1 - xX_l) = \Lambda_v x^v + \Lambda_{v-1}x^{v-1} + ... + \Lambda_1 x + 1) \qquad (2.11)$$

Then the left side becomes zero, giving

$0 = Y_l X_l^{j+v}(1 + \Lambda_1 X_l^{-1} + \Lambda_2 X_l^{-2} + ... + \Lambda_{v-1} X_l^{-(v-1)} + \Lambda_v X_l^{-v})$

or

$Y_l(X_l^{j+v} + \Lambda_1 X_l^{j+v-1} + ... + \Lambda_v X_l^j) = 0$

Such an equation holds for each l and each j. Summing up these equations from $l = 1$ to $l = v$, for each j, gives,

$\sum_{l=1}^{v} Y_l(X_l^{j+v} + \Lambda_1 X_l^{j+v-1} + ... + \Lambda_v X_l^j) = 0$

or

$\sum_{l=1}^{v} Y_l X_l^{j+v} + \Lambda_1 \sum_{l=1}^{v} Y_l X_l^{j+v-1} + ... + \Lambda_v \sum_{l=1}^{v} Y_l X_l^j = 0$

The individual sums seem to be the syndromes and thus the equation becomes

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

where $j = 1, 2, ..., v$

This set of linear equations relates the syndromes to the coefficients of the error-location polynomial $\Lambda(x)$. It can also be expressed in matrix form as

$$A\Lambda = \begin{bmatrix} S_1 & S_2 & S_3 & \cdots & S_{v-1} & S_v \\ S_2 & S_3 & S_4 & \cdots & S_v & S_{v+1} \\ S_3 & S_4 & S_5 & \cdots & S_{v+1} & S_{v+2} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ S_v & S_{v+1} & S_{v+2} & \cdots & S_{2v-2} & S_{2v-1} \end{bmatrix} \begin{bmatrix} \Lambda_v \\ \Lambda_{v-1} \\ \Lambda_{v-1} \\ . \\ . \\ . \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{v+1} \\ -S_{v+2} \\ -S_{v+3} \\ . \\ . \\ . \\ -S_{2v} \end{bmatrix} \tag{2.12}$$

The above system of equations has a unique solution for Λ which can be obtained by inverting the matrix A, if A is nonsingular. The matrix A is nonsingular if $v \leq t$ [54].

Peterson's direct-solution algorithm solves for the error locator polynomial $\Lambda(x)$ in Equation (2.12) as follows [54]: as a trial value, v is set to the error correction capability of the code t and the determinant of the matrix computed. If the determinant is nonzero, it can be shown that this is the correct value of v. Otherwise, if it is zero, then the trial value of v is reduced by 1 and the process is repeated until a nonzero determinant is obtained. After the determinant has been obtained, the coefficients of $\Lambda(x)$ are determined using the value of v in Equation (2.12) by standard techniques of linear algebra.

Peterson's direct-solution algorithm is inefficient for codes with a large error correcting capability t. The number of computations necessary to invert a v by v matrix is directly proportional to v^3. In most applications, designers often prefer to use codes that correct a large number of errors. The following subsections detail two efficient decoding methods: the Berlekamp-Massey algorithm and Euclid's algorithm.

2.3.1 Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm relies on the fact that the matrix equation of Equation (2.12) is not arbitrary in its form, rather, the matrix is highly structured. This structure is used to obtain the vector Λ by a method that is conceptually more complicated but computationally much simpler [54][59][60].

If the vector Λ is known, then the first row of the above matrix equation defines S_{v+1} in terms of $S_1, ..., S_v$. The second row defines S_{v+2} in terms of $S_2, ..., S_{v+1}$ and so forth. This sequential process can be summarized by the recursive relation

$$S_j = -\sum_{i=1}^{v} \Lambda_i S_{j-i}, \qquad j = v+1, ..., 2v \tag{2.13}$$

For fixed Λ, this is equivalent to the equation of an autoregressive filter. It can be implemented as a linear-feedback shift register with taps given by the coefficients of Λ.

Using this argument, the problem has been reduced to the design of a linear-feedback shift register that will consequently generate the known sequences of syndromes. Many such shift registers exist, but it is desirable to find the smallest linear-feedback shift register with this property. This will give the least-weight error pattern with a polynomial $\Lambda(x)$ of smallest degree v. The polynomial of smallest degree v is unique, since the $v \times v$ matrix of the original problem is invertible.

Any procedure for designing the autoregressive filter is also a method for solving the matrix equation for the Λ vector. The procedure applies in any field and does not assume any special properties for the sequence $S_1, S_2, ..., S_{2t}$. To design the required shift register, the shift register length L and feedback connection polynomial $\Lambda(x)$ must be determined. $\Lambda(x)$ has the form

$$\Lambda(x) = \Lambda_v x^v + \Lambda_{v-1} x^{v-1} + ... + \Lambda_1 x + 1 \tag{2.14}$$

where $\deg \Lambda(x) \leq L$.

The Berlekamp-Massey Algorithm uses the initial conditions $\Lambda^{(0)}(x) = 1, B^{(0)} = 1$, and $L_0 = 0$, to compute $\Lambda^{(2t)}(x)$ as follows:

$$\Delta_r = \sum_{j=0}^{n-1} \Lambda_j^{(r-1)} S_{r-j} \tag{2.15}$$

$$L_r = \delta_r (r - L_{r-1}) + (1 - \delta_r) L_{r-1} \tag{2.16}$$

$$\begin{bmatrix} \Lambda^{(r)}(x) \\ B^{(r)}(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1} \delta_r & (1 - \delta_r) x \end{bmatrix} \begin{bmatrix} \Lambda^{(r-1)}(x) \\ B^{(r-1)}(x) \end{bmatrix} \tag{2.17}$$

for $r = 1, ..., 2t$

$$\delta_r = 1, \text{ if both } \Delta_r \neq 0 \text{ and } 2L_{r-1} \leq r - 1; \text{ and } \delta_r = 0, \text{ otherwise.} \tag{2.18}$$

At the end of the $2t$ iterations, the smallest-degree polynomial $\Lambda^{(2t)}(x)$ with $\Lambda_0^{(2t)} = 1$ satisfying the relation

$$S_r + \sum_{j=1}^{n-1} \Lambda_j^{(2t)} S_{r-j} = 0$$

where $r = L_{2t} + 1, ..., 2t$ will be obtained.

Then if we define the error evaluation polynomial $\Omega(x)$ by the relation

$$S(x)\Lambda(x) = \Omega(x) \bmod x^{2t} \tag{2.19}$$

then we can use $\Omega(x)$ to solve for the error magnitudes $Y_1, ..., Y_v$.

2.3.2 Euclid's Algorithm

Euclid's algorithm is a recursive procedure for calculating the greatest common divisor (GCD) of two polynomials [61]. In a slightly expanded version, the algorithm will always produce the polynomials $a(x)$ and $b(x)$ satisfying

$$GCD[s(x), t(x)] = a(x)s(x) + b(x)t(x) \tag{2.20}$$

Euclid's algorithm uses the initial conditions
$R^{(0)}(x) = x^{2t}$, $T^{(0)}(x) = \sum_{j=1}^{2t} S_j x^{j-1}$, and

$$A^{(0)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

to compute $\Lambda^{(2t)}(x)$ as follows:

$$Q^{(r)}(x) = \left\lfloor \frac{R^{(r)}(x)}{T^{(r)}(x)} \right\rfloor \tag{2.21}$$

$$A^{(r+1)}(x) = \begin{bmatrix} 1 & 0 \\ 0 & Q^{(r)}(x) \end{bmatrix} A^{(r)}(x) \tag{2.22}$$

$$\begin{bmatrix} R^{r+1}(x) \\ T^{(r+1)}(x) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & Q^{(r)}(x) \end{bmatrix} \begin{bmatrix} R^{r}(x) \\ T^{(r)}(x) \end{bmatrix} \tag{2.23}$$

The algorithm stops when the degree of $T^{(r)}$ is less than t.

At the end of the iteration, the error evaluator and error locator polynomials are found using

$$\Omega(x) = \Delta^{-1} T^{(r)}(x) \tag{2.24}$$

$$\Lambda(x) = \Delta^{-1} A_{22}^{(r)}(x) \tag{2.25}$$

respectively, where $\Delta = A_{22}^{(r)}(0)$ and A_{22} is the element of the matrix $A^{(r)}$ in the second row and second column.

This algorithm has been modified by Shao *et al* to avoid the computation of the inverse elements of the Galois field [36][62]. The modified Euclid's algorithm recursively finds the i-th remainder $R_i(x)$ and the quantities $\gamma_i(x)$ and $\lambda_i(x)$ that satisfy the relation

$$\gamma_i(x)A(x) + \lambda_i(x)S(x) = R_i(x)$$

and stops when the degree of the remainder polynomial $R_i(x)$ is less than t, where $A(x) = x^{2t}$ and $S(x) = \sum_{k=1}^{2t} S_k x^{2t-k}$.

Using the initial conditions $R_0(x) = A(x), Q_0(x) = S(x), \lambda_0(x) = 0, \mu_0(x) = 1, \gamma_0(x) = 1, \eta_0(x) = 0$, it computes $R_i(x), \lambda_i(x)$ and $\gamma_i(x)$ as follows:

$$\begin{aligned} R_i(x) &= [\sigma_{i-1}b_{i-1}R_{i-1}(x) + \widehat{\sigma}_{i-1}a_{i-1}Q_{i-1}(x)] \\ &\quad -x^{|l_{i-1}|}[\sigma_{i-1}a_{i-1}Q_{i-1}(x) + \widehat{\sigma}_{i-1}b_{i-1}R_{i-1}(x)] \end{aligned} \tag{2.26}$$

$$\begin{aligned} \lambda_i(x) &= [\sigma_{i-1}b_{i-1}\lambda_{i-1}(x) + \widehat{\sigma}_{i-1}a_{i-1}\mu_{i-1}(x)] \\ &\quad -x^{|l_{i-1}|}[\sigma_{i-1}a_{i-1}\mu_{i-1}(x) + \widehat{\sigma}_{i-1}b_{i-1}\lambda_{i-1}(x)] \end{aligned} \tag{2.27}$$

$$\begin{aligned} \gamma_i(x) &= [\sigma_{i-1}b_{i-1}\gamma_{i-1}(x) + \widehat{\sigma}_{i-1}a_{i-1}\eta_{i-1}(x)] \\ &\quad -x^{|l_{i-1}|}[\sigma_{i-1}a_{i-1}\mu_{i-1}(x) + \widehat{\sigma}_{i-1}b_{i-1}\gamma_{i-1}(x)] \end{aligned} \tag{2.28}$$

$$Q_i(x) = \sigma_{i-1}Q_{i-1}(x) + \widehat{\sigma}_{i-1}R_{i-1}(x) \tag{2.29}$$

$$\mu_i(x) = \sigma_{i-1}\mu_{i-1}(x) + \widehat{\sigma}_{i-1}\lambda_{i-1}(x) \tag{2.30}$$

$$\eta_i(x) = \sigma_{i-1}\eta_{i-1}(x) + \widehat{\sigma}_{i-1}\gamma_{i-1}(x) \tag{2.31}$$

where a_{i-1} and b_{i-1} are the leading coefficients of $R_{i-1}(x)$ and Q_{i-1} respectively, $l_{i-1} = \deg[R_{i-1}(x)] - [\deg(Q_{i-1}(x)]$, $\sigma_{i-1} = 1$ if $l_{i-1} \geq 0$ and $\sigma_{i-1} = 0$ if $l_{i-1} < 0$.

The iterations stop when $\deg[R_i(x)] < t$, after which the error locator polynomial $\Lambda(x) = \lambda_i(x)$ and error evaluator polynomial $\Omega(x) = R_i(x)$.

Once the error locator $\Lambda(x)$ and error evaluator $\Omega(x)$ polynomials have been determined using the above techniques, the error locations and error values or magnitudes can be found using the Chien search and the Forney algorithm. These methods are described in the following subsections.

2.3.3 Chien Search

Once the coefficients of the error locator polynomial $\Lambda_1, ..., \Lambda_v$ have been found, the roots of $\Lambda(x)$ can be computed using the Chien search. The Chien search is a systematic means of evaluating the error locator polynomial at all elements in a field $GF(2^m)$ [63]. The evaluation of each element is performed in

$$\Lambda(x) = \Lambda_v x^v + \Lambda_{v-1} x^{v-1} + ... + \Lambda_1 x + 1 \tag{2.32}$$

to check for $\Lambda(x) = 0$.

2.3.4 The Forney Algorithm

The Forney algorithm is an efficient method often used to compute the error magnitudes. The error evaluator polynomial $\Omega(x)$ is defined as [59]

$$\Omega(x) = S(x)\Lambda(x) \bmod x^{2t} \tag{2.33}$$

where $\Lambda(x) = \Lambda_v x^v + \Lambda_{v-1} x^{v-1} + ... + \Lambda_1 x + 1 = \prod_{l=1}^{v}(1 - xX_l)$

and

$S(x) = \sum_{j=1}^{2t} S_j x^j = \sum_{j=1}^{2t} \sum_{i=1}^{v} Y_i X_i^j x^j$

Equation (2.33) can now be expanded as

$$\Omega(x) = x \sum_{i=1}^{v} Y_i X_i \prod_{l \neq i} (1 - X_l x) \tag{2.34}$$

Instead of using matrix inversion to find the error magnitudes, the Forney algorithm calculates them as

$$Y_l = \frac{\Omega(X_l^{-1})}{\prod_{j \neq l}(1 - X_j X_l^{-1})} = -\frac{\Omega(X_l^{-1})}{X_l^{-1}\Lambda'(X_l^{-1})} = -\frac{X_l \Omega(X_l^{-1})}{\Lambda'(X_l^{-1})} \tag{2.35}$$

where the derivative of $\Lambda(x)$ is defined as

$$\Lambda'(x) = -\sum_{i=1}^{v} X_i \prod_{j \neq i} (1 - x X_j) \tag{2.36}$$

and hence

$$\Lambda'(X_l^{-1}) = -X_l \prod_{j \neq l} (1 - X_j X_l^{-1}) \tag{2.37}$$

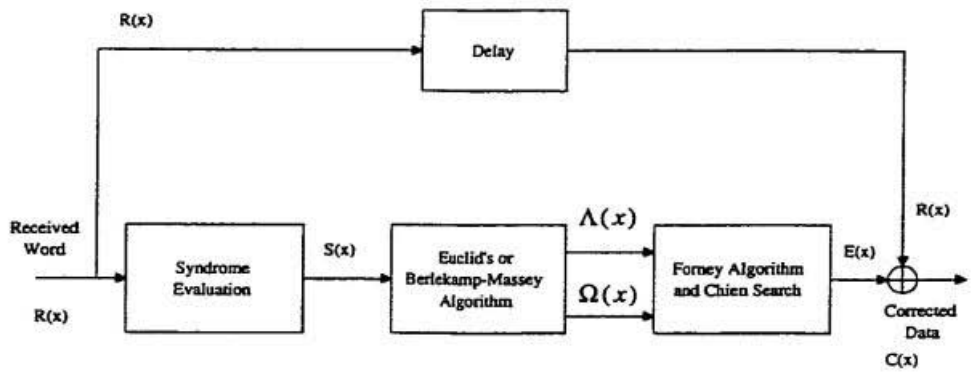

Figure 2.2: Algebraic Decoder

In summary, the algebraic decoding algorithm works as follows:

Step 1: Calculate the syndromes according to Equation (2.6).

Step 2: Perform the Berlekamp-Massey or Euclid's algorithm to obtain the error locator polynomial $\Lambda(x)$. Also find the error evaluator polynomial $\Omega(x)$.

Step 3: Perform the Chien Search to find the roots of $\Lambda(x)$.

Step 4: Find the error values $Y(x) = E(x)$ according to Equation (2.35).

Step 5: Correct the received word $C(x) = E(x) + R(x)$

The structure of the algebraic decoder is shown in Figure 2.2.

2.4 Time-Domain Decoding

2.4.1 Error Locator and Evaluator Polynomials

The time-domain decoding algorithm was first proposed by Blahut [64]. It is explained in detail in the references [5][6][54] and is only summarized in this subsection.

The time-domain algorithm uses the initial conditions $\lambda_i^{(0)} = b_i^{(0)} = \omega_i^{(0)} = 1$ and $\lambda_i^{'(0)} = b_i^{'(0)} = a_i^{(0)} = 0$ for all i, to compute the following set of recursive equations:

$$\Delta_r = \sum_{i=0}^{n-1} \alpha^{ir} [\lambda_i^{(r-1)} r_i] \tag{2.38}$$

$$L_r = \delta_r (r - L_{r-1}) + (1 - \delta_r) L_{r-1} \tag{2.39}$$

$$\begin{bmatrix} \lambda_i^{(r)} \\ b_i^{(r)} \\ \lambda_i^{'(r)} \\ b_i^{'(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} & 0 & 0 \\ \Delta_r^{-1} \delta_r & (1-\delta_r)\alpha^{-i} & 0 & 0 \\ 0 & -\Delta_r & 1 & -\Delta_r \alpha^{-i} \\ 0 & (1-\delta_r) & \Delta_r^{-1}\delta_r & (1-\delta_r)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \lambda_i^{(r-1)} \\ b_i^{(r-1)} \\ \lambda_i^{'(r-1)} \\ b_i^{'(r-1)} \end{bmatrix} \tag{2.40}$$

$$\begin{bmatrix} \omega_i^{(r)} \\ a_i^{(r)} \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r \alpha^{-i} \\ \Delta_r^{-1}\delta_r & (1-\delta_r)\alpha^{-i} \end{bmatrix} \begin{bmatrix} \omega_i^{(r-1)} \\ a_i^{(r-1)} \end{bmatrix} \tag{2.41}$$

for $i = 0, ..., n-1$, $r = 1, 2, ..., 2t$.

$L = 0$ and $\delta = 1$ if both $\Delta_r \neq 0$ and $2L \leq r - 1$, and $\delta = 0$ otherwise.

2.4.2 Error Evaluation

Using the error locator vector λ, the vector $\lambda' = \lambda^{'(2t)}$, the error evaluator vector $\omega = \omega^{(2t)}$, the error magnitudes are computed as

$e_i = -\frac{\alpha^i(w_i)}{\lambda_i'}$, if $\lambda_i = 0$

$e_i = 0$, if $\lambda_i \neq 0$

The structure of the time-domain decoder is shown in Figure 2.3.

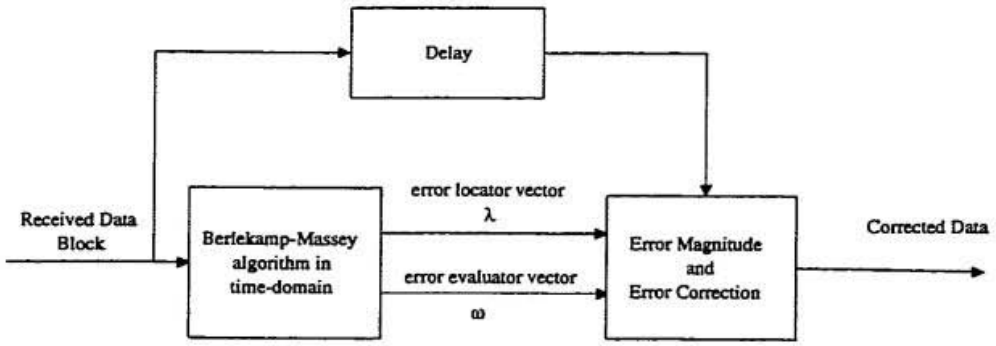

Figure 2.3: Time-Domain Decoder

2.5 Error Correction

Once $E(x)$ is known, the corrected codeword $C(x)$ can be obtained from $C(x) = R(x) + E(x)$.

2.6 Algebraic vs. Time-Domain Decoding Algorithms

Based on the above discussion, the fundamental differences between the algebraic and time-domain decoding algorithms are listed below:

(1) The time-domain algorithm has one major computational step. Unlike the algebraic decoding algorithm, it does not compute the syndromes or perform the Chien search to find the error locations.

(2) The time-domain algorithm deals with vectors which have n components while different length vectors and different degree polynomials are used in the various steps of the algebraic algorithms.

(3) By changing the error correction capability of the code t, the operations in the

time-domain algorithms essentially remain the same, while those in the algebraic algorithm are dependent on t.

(4) Although complex to design, the algebraic decoding technique is recommended for high speed applications. The major drawback of the time-domain algorithm is its high computation count. This is brought about by the fact that it has to operate on the complete data sequence of length n, while the algebraic algorithm needs to work only on the syndrome sequence of length $2t = (n - k)$ m-bit symbols.

2.7 RS Encoder/Decoder Architectures

In 1984, Blahut [64] originally presented two architectures for universal RS decoders based on the time-domain algorithms. The decoders work directly on the received data to generate the error sequence. They are attractive for VLSI design since one major computational step is required. Unlike the algebraic decoders, neither the syndrome evaluation nor the Chien search is required. Such a decoder can be used to decode any RS or BCH codeword up to the limits of the storage registers associated with the chip. Within these limits, it can correct any number of random errors and erasures depending on the received data. Shayan *et al* restructured the time-domain algorithm to implement a versatile time-domain [5] and a cellular decoder [6] which can operate in a Galois field $GF(2^m)$ with a fixed m.

Conceptual models for the logic structures of the RS encoder and decoder chips were presented in [65][66]. The encoder is constructed by cascading and interconnecting a group of VLSI chips. The decoder architecture is based on the repetitive and recursive properties of RS decoding procedures.

Truong *et al* [8][9] reported a single chip VLSI RS encoder implemented in

NMOS technology. The encoding algorithm is a bit-serial multiplication algorithm developed by Berlekamp for the encoding of RS codes using a dual basis over a Galois field. Compared to the conventional RS encoder for long codes, which often requires lookup tables to perform the multiplication of two field elements, Berlekamp's algorithm requires only shifting and exclusive-OR operations.

Shao *et al* [62] developed a pipeline structure of a transform decoder similar to a systolic array to decode RS codes. The error locator polynomial is computed by the modified Euclid's algorithm which avoids computing inverse elements. The modified Euclid's algorithm architecture is based on the pipeline architecture suggested by Brent and Kung [67] to compute the greatest common divisor of two polynomials.

A full-custom CMOS implementation of a RS encoder was proposed by Maki *et al* in 1986 [35]. In order to reduce the transistor count, domino logic was used. Its architecture is invariant in operational speed or silicon area to the field polynomial, generator polynomial or operation in the dual basis or normal field. With k encoder chips operating in parallel, a $k-1$ fault tolerant system can be constructed.

A pipelined RS decoder based on the transform decoding algorithm presented earlier by the authors is described in [36][37]. The transform decoding technique is replaced by a time domain algorithm to permit efficient pipeline processing with reduced circuitry. By using multiplexing, the proposed Euclid's algorithm maintains the throughput rate with little additional complexity.

In 1990, Tong [38] presented an 8-error correcting RS encoder-decoder. The encoder and decoder can independently process 40 Mbytes of data per second. The chip was designed using a standard ASIC methodology and fabricated in a 1-μm CMOS compact-array technology.

In 1991, Seroussi [29] presented a systolic architecture for a RS encoder. The architecture completely eliminates the global feedback signal found in the conventional encoder architectures which use the linear feedback shift register (LFSR). The encoding algorithm is based on the Cauchy representation of the generator matrix of the code. The architecture is suitable for very high speed applications, where global signals and the need for global synchronization may pose restrictions on the achievable switching speed of the encoder.

A full-custom CMOS VLSI implementation of a Reed-Solomon decoder for the Hubble Space Telescope and television applications was presented by Whitaker *et al* in 1991 [3][39]. The architecture is similar to others presented in the references [40][41]. It is implemented in a 1.6 μm double metal CMOS technology and operates at a data rate of 80 Mbits/s using a 10 MHz system/data clock. In these designs, Euclid's algorithm is used to determine both the error location and error magnitude polynomials.

In order to solve the problem of multiple notations and multiple algorithms often faced by designers, high level synthesis is used to study the different BCH and RS decoding algorithms [42]. Special VHDL packages are created to describe the various operations on Galois fields. A VHDL synthesis tool consequently allows efficient exploration of various architectures in order to select an optimum one.

Methods for reducing the computation count in the time domain algorithm for RS decoding were presented by Choomchuang and Arambepola in 1993 [43]. An architecture for an error correction circuit suitable for high-rate data decoding of RS codes was proposed in [44]. The operational steps for multiple-error decoding are reduced by a 4-stage pipeline and a superscalar processor of a Galois field. The

experimental chip achieves 16 Mbytes/s of data decoding sufficient for compressed video signals of high-definition as well as those of standard-definition TV's.

The use of high level synthesis techniques to realize a high-speed Reed-Solomon CODEC was reported by Cools *et al* in 1994 [45]. High level synthesis allows rapid design exploration over a large range of architectures. An error free transfer is guaranteed between all the levels of the design process. The design was captured using a combination of Mentor Graphics and a Cathedral-$\frac{2}{3}$ compiler. The architectural design phase concentrates on the composition of the data path and global cycle count; logic synthesis performs local optimizations in terms of hardware and timing; whereas the place-and-route tools compose the final layout.

A low circuit complexity architecture for a Reed-Solomon encoder suitable for satellites and pocket size wireless terminals was presented by Hasan and Bhargava in 1995 [46]. The encoder uses the triangular basis multiplication algorithm. Using pipeline and bit-serial operations the encoder is able to obtain code rates ranging from unity to a minimum value determined by the associated hardware circuitry.

In 1995, Chen *et al* [47] presented a three stage pipelined VLSI architecture of a Reed-Solomon decoder. The decoder has an erasure function and uses the modified Euclid's algorithm to solve the key equation. The block length is variable. The hardware complexity is shown to be only dependent on the number of parity check bytes. The modified Euclid's algorithm allows the error evaluator and error location vectors to be determined sequentially by using a smaller amount of hardware. The algorithm state machine and architecture were verified using Verilog hardware description language.

In 1995, Iwamura *et al* [48] proposed a class of systolic arrays to perform binary

RS decoding procedures including erasure correction. Such an RS decoder is suitable for VLSI implementation since the arrays consist of simple processing elements of the same type.

In 1997, Hsu and Wang [49] presented a pipelined VLSI architecture of a Reed-Solomon decoder which combines a modified-time domain Berlekamp-Massey algorithm with the remainder decoding concept. For a t-error correcting RS code with block length n, only $2t$ consecutive symbols, instead of n are required to determine the discrepancy value during the decoding process.

A VLSI architecture for an area efficient Reed-Solomon product-code encoder and decoder was published by Kwon and Shin in 1997 [4]. The architecture uses functional block sharing to implement the encoder, modified syndrome and erasure locator polynomial evaluations. The modified Euclid's algorithm is used to determine the error/erasure locator and error/erasure evaluator polynomials. The architecture is recommended for encoding/decoding audio and video signals over $GF(256)$.

Rapid prototyping was used to implement a Reed-Solomon decoder in [50]. Erasure correction is supported. The chip includes two 256-byte ROMs, a table look-up for the inverse of the elements in $GF(2^8)$ and one 512-byte RAM or buffer registers.

A Reed-Solomon decoder which operates in the $GF(2^8)$ was presented by Saodt in [51]. The ASIC is targeted for military anti-jamming applications in microwave links. It uses FIFO buffers that are external to the chip.

2.8 Summary

The various decoding algorithms for Reed-Solomon codes have been presented. A survey on the existing RS encoder and decoder architectures usually designed for a fixed m has been given. Universal RS decoder architectures based on the time-domain algorithms first appeared in 1984. Versatile time-domain and cellular decoders were subsequently derived from them. They require only one major computational step in locating the error patterns. Single chip RS decoders that implement the algebraic and transform decoding algorithms have also been reported. The Berlekamp-Massey or Euclid's algorithm is often used to find the error location and magnitude polynomials.

Chapter 3

Proposed VLSI Arithmetic Architectures

This chapter introduces and describes an approach which exploits the symmetric properties of available VLSI arithmetic architectures to perform multiplication, exponentiation and inverse operations in $GF(2^m)$. Traditionally, such operations are performed using hardware which has been designed to function over $GF(2^m)$ for a fixed value of m. The requirement to operate with different symbol sizes of m-bits seems to recur throughout the design of the RS encoder and decoder circuits. VLSI chips which have been reported in the literature always use a fixed block length n and a fixed symbol size m since the exponentiation, multiplication and division circuits in Galois fields have different designs for different values of m. One of the major contributions of this thesis has been to demonstrate that the parameters m and n can be variable without a significant increase in hardware.

The proposed approach defines a standard symbol of $\widehat{m}$-bits which readily allows any symbol from $GF(2^m)$ where $m \leq \widehat{m}$ to be represented as an $\widehat{m}$-bit symbol whose $(\widehat{m} - m)$ most significant bits have been set to zero. This principle facilitates all arithmetic functions in the Galois field with the symbol size $m \leq \widehat{m}$ to

be implemented as subsets of $m = \widehat{m}$ with a small penalty in hardware. To illustrate the concept, an m-programmable Galois field multiplier which uses the standard basis representation of the elements is first proposed, where $m \leq 8$. By using this multiplier, it is shown that the exponent and inverse functions can be implemented using the same hardware structure. The resulting circuits are systolic and have simple, regular communication and control structures. They also allow unidirectional data flow which is advantageous over systems with contraflowing data streams [68][69]. These circuits will be used in the design of an m and t-programmable RS encoder/decoder which is later described in Chapter 4. The choice of a fixed symbol size $m = 8$ is fairly common in a wide range of practical applications [2][3][4][8][35][39][45][47][65][66], but is made variable for values of $m = 3, 4, 5, 6, 7$ and 8 as an illustration in this thesis. The architecture can be easily extended to accommodate larger values of m.

3.1 m-Programmable Galois Field Multiplier

For arbitrary elements $A(x) = \sum_{k=0}^{m-1} a_k x^k$, $B(x) = \sum_{k=0}^{m-1} b_k x^k$ in $GF(2^m)$, and the primitive polynomial $P(x) = \sum_{k=0}^{m-1} p_k x^k$, the product $C(x)$ of $A(x)$ multiplied by $B(x)$ is given by

$$\begin{aligned} C(x) &= A(x)B(x) \ mod \ P(x) \\ &= [\textstyle\sum_{k=0}^{m-1} A(x)b_k x^k] \ mod \ P(x) \\ &= (...(A(x)b_{m-1}x + A(x)b_{m-2})x + ...)x + A(x)b_0 \\ &= c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + ... + c_1 x + c_0 \end{aligned} \tag{3.1}$$

As described in [15], the product $C(x)$ as defined in Equation (3.1) can be computed recursively as

$$\begin{aligned} T_0(x) &= 0 \\ T_i(x) &= [T_{i-1}(x)x] \; mod \; P(x) \; + \; A(x)b_{m-i}, \quad i = 1, 2, ..., m \\ C(x) &= T_m(x) \end{aligned} \tag{3.2}$$

where

$$T_i(x) = t_{i,m-1}x^{m-1} + t_{i,m-2}x^{m-2} + ... + t_{i,1}x + t_{i,0} \tag{3.3}$$

Denoting the most significant bit (MSB) of $T_i(x)$ as M_i, the recurrence relation can be rewritten as

$$T_i(x) = T_{i-1}(x)x + P(x)M_{i-1} + A(x)b_{m-i} \tag{3.4}$$

where $i = 1, 2, ..., m$.

The above computation can be implemented using a parallel-in-parallel-out two dimensional systolic array with $m \times m$ basic cells. Each cell at position (i, k) would perform the logic operation [11]

$$t_{i,k} = t_{i-1,k+1} \oplus (p_{m-k} \cdot M_{i-1}) \oplus (a_{m-k} \cdot b_{m-i}) \tag{3.5}$$

where $i = 1, 2, ..., m$ and $k = 1, 2, ..., m$.

In the case where $m = \widehat{m} = 8$, the systolic array with 8×8 basic cells is shown in Figure 3.1. As shown in the figure, the coefficients of $A(x)$ and $P(x)$ enter the array from the top whereas those of $B(x)$ enter from the left-hand side, such that the operation defined in Equation (3.5) is performed at the ith row. It consists of 279 logic gates as reported by the Synopsys synthesis tools.

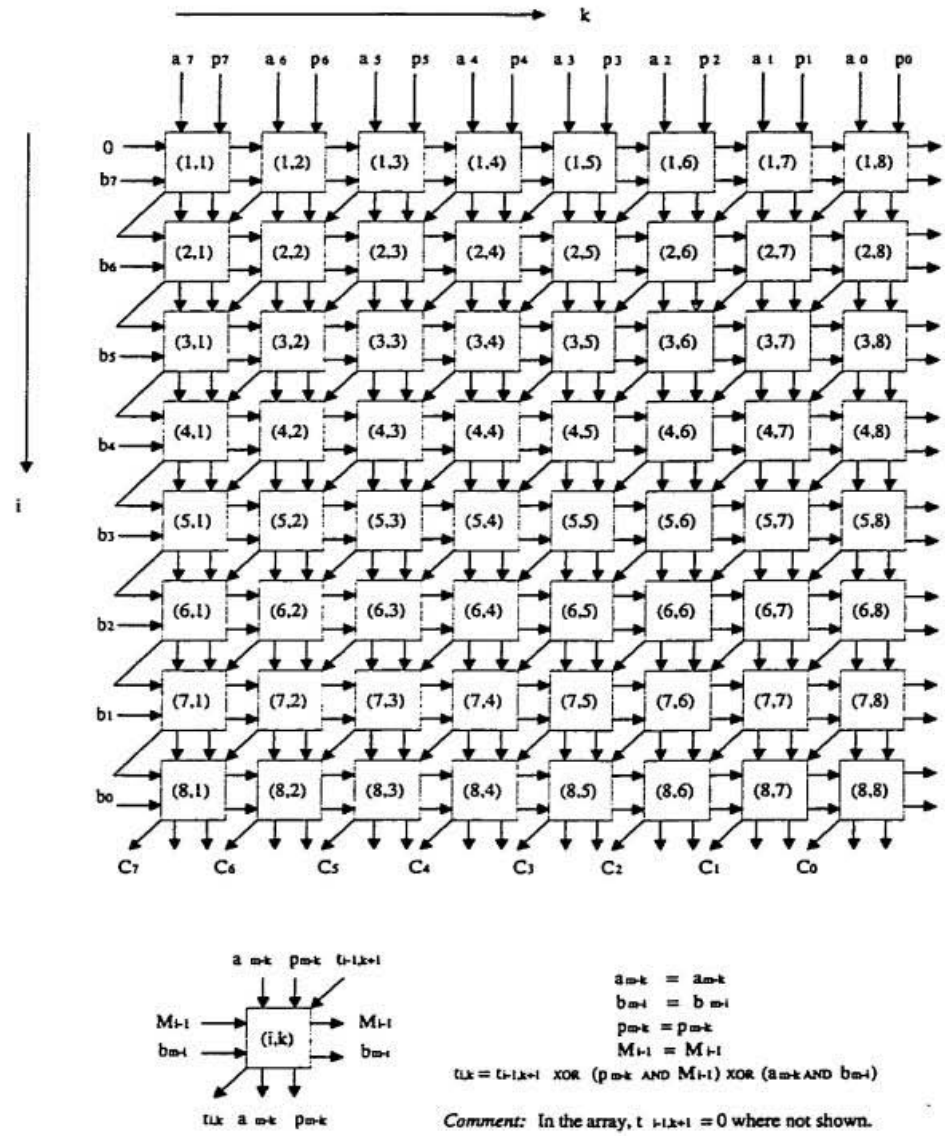

Figure 3.1: A Parallel-In-Parallel-Out Multiplier for $GF(2^8)$

The *Boolean Equations* $t_{i,k}$, as defined by Equation (3.5), for all the 64 cells of the $GF(2^8)$ multiplier are as follows:

m = 8: Primitive Polynomial $P(x) = x^8 + x^4 + x^3 + x^2 + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_7 \cdot a_7)$	$t_{2,1} = (b_6 \cdot a_7) \oplus t_{1,2}$	$t_{3,1} = (b_5 \cdot a_7) \oplus t_{2,2}$
$t_{1,2} = (b_7 \cdot a_6)$	$t_{2,2} = (b_6 \cdot a_6) \oplus t_{1,3}$	$t_{3,2} = (b_5 \cdot a_6) \oplus t_{2,3}$
$t_{1,3} = (b_7 \cdot a_5)$	$t_{2,3} = (b_6 \cdot a_5) \oplus t_{1,4}$	$t_{3,3} = (b_5 \cdot a_5) \oplus t_{2,4}$
$t_{1,4} = (b_7 \cdot a_4)$	$t_{2,4} = (b_6 \cdot a_4) \oplus M_1 \oplus t_{1,5}$	$t_{3,4} = (b_5 \cdot a_4) \oplus M_2 \oplus t_{2,5}$
$t_{1,5} = (b_7 \cdot a_3)$	$t_{2,5} = (b_6 \cdot a_3) \oplus M_1 \oplus t_{1,6}$	$t_{3,5} = (b_5 \cdot a_3) \oplus M_2 \oplus t_{2,6}$
$t_{1,6} = (b_7 \cdot a_2)$	$t_{2,6} = (b_6 \cdot a_2) \oplus M_1 \oplus t_{1,7}$	$t_{3,6} = (b_5 \cdot a_2) \oplus M_2 \oplus t_{2,7}$
$t_{1,7} = (b_7 \cdot a_1)$	$t_{2,7} = (b_6 \cdot a_1) \oplus t_{1,8}$	$t_{3,7} = (b_5 \cdot a_1) \oplus t_{2,8}$
$t_{1,8} = (b_7 \cdot a_0)$	$t_{2,8} = (b_6 \cdot a_0) \oplus M_1$	$t_{3,8} = (b_5 \cdot a_0) \oplus M_2$
Row 4: $M_3 = t_{3,1}$, $i = 4$	Row 5: $M_4 = t_{4,1}$, $i = 5$	Row 6: $M_5 = t_{5,1}$, $i = 6$
$t_{4,1} = (b_4 \cdot a_7) \oplus t_{3,2}$	$t_{5,1} = (b_3 \cdot a_7) \oplus t_{4,2}$	$t_{6,1} = (b_2 \cdot a_7) \oplus t_{5,2}$
$t_{4,2} = (b_4 \cdot a_6) \oplus t_{3,3}$	$t_{5,2} = (b_3 \cdot a_6) \oplus t_{4,3}$	$t_{6,2} = (b_2 \cdot a_6) \oplus t_{5,3}$
$t_{4,3} = (b_4 \cdot a_5) \oplus t_{3,4}$	$t_{5,3} = (b_3 \cdot a_5) \oplus t_{4,4}$	$t_{6,3} = (b_2 \cdot a_5) \oplus t_{5,4}$
$t_{4,4} = (b_4 \cdot a_4) \oplus M_3 \oplus t_{3,5}$	$t_{5,4} = (b_3 \cdot a_4) \oplus M_4 \oplus t_{4,5}$	$t_{6,4} = (b_2 \cdot a_4) \oplus M_5 \oplus t_{5,5}$
$t_{4,5} = (b_4 \cdot a_3) \oplus M_3 \oplus t_{3,6}$	$t_{5,5} = (b_3 \cdot a_3) \oplus M_4 \oplus t_{4,6}$	$t_{6,5} = (b_2 \cdot a_3) \oplus M_5 \oplus t_{5,6}$
$t_{4,6} = (b_4 \cdot a_2) \oplus M_3 \oplus t_{3,7}$	$t_{5,6} = (b_3 \cdot a_2) \oplus M_4 \oplus t_{4,7}$	$t_{6,6} = (b_2 \cdot a_2) \oplus M_5 \oplus t_{5,7}$
$t_{4,7} = (b_4 \cdot a_1) \oplus t_{3,8}$	$t_{5,7} = (b_3 \cdot a_1) \oplus t_{4,8}$	$t_{6,7} = (b_2 \cdot a_1) \oplus t_{5,8}$
$t_{4,8} = (b_4 \cdot a_0) \oplus M_3$	$t_{5,8} = (b_3 \cdot a_0) \oplus M_4$	$t_{6,8} = (b_2 \cdot a_0) \oplus M_5$
Row 7: $M_6 = t_{6,1}$, $i = 7$	Row 8: $M_7 = t_{7,1}$, $i = 8$	
$t_{7,1} = (b_1 \cdot a_7) \oplus t_{6,2}$	$t_{8,1} = (b_0 \cdot a_7) \oplus t_{7,2}$	
$t_{7,2} = (b_1 \cdot a_6) \oplus t_{6,3}$	$t_{8,2} = (b_0 \cdot a_6) \oplus t_{7,3}$	
$t_{7,3} = (b_1 \cdot a_5) \oplus t_{6,4}$	$t_{8,3} = (b_0 \cdot a_5) \oplus t_{7,4}$	
$t_{7,4} = (b_1 \cdot a_4) \oplus M_6 \oplus t_{6,5}$	$t_{8,4} = (b_0 \cdot a_4) \oplus M_7 \oplus t_{7,5}$	
$t_{7,5} = (b_1 \cdot a_3) \oplus M_6 \oplus t_{6,6}$	$t_{8,5} = (b_0 \cdot a_3) \oplus M_7 \oplus t_{7,6}$	
$t_{7,6} = (b_1 \cdot a_2) \oplus M_6 \oplus t_{6,7}$	$t_{8,6} = (b_0 \cdot a_2) \oplus M_7 \oplus t_{7,7}$	
$t_{7,7} = (b_1 \cdot a_1) \oplus t_{6,8}$	$t_{8,7} = (b_0 \cdot a_1) \oplus t_{7,8}$	
$t_{7,8} = (b_1 \cdot a_0) \oplus M_6$	$t_{8,8} = (b_0 \cdot a_0) \oplus M_7$	

The *Boolean Equations* for cases where $m < 8$ are as follows:

m = 3: Primitive Polynomial $P(x) = x^3 + x + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_2 \cdot a_2)$	$t_{2,1} = (b_1 \cdot a_2) \oplus t_{1,2}$	$t_{3,1} = (b_0 \cdot a_2) \oplus t_{2,2}$
$t_{1,2} = (b_2 \cdot a_1)$	$t_{2,2} = (b_1 \cdot a_1) \oplus M_1 \oplus t_{1,3}$	$t_{3,2} = (b_0 \cdot a_1) \oplus M_2 \oplus t_{2,3}$
$t_{1,3} = (b_2 \cdot a_0)$	$t_{2,3} = (b_1 \cdot a_0) \oplus M_1$	$t_{3,3} = (b_0 \cdot a_0) \oplus M_2$

m = 4: Primitive Polynomial $P(x) = x^4 + x + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_3 \cdot a_3)$	$t_{2,1} = (b_2 \cdot a_3) \oplus t_{1,2}$	$t_{3,1} = (b_1 \cdot a_3) \oplus t_{2,2}$
$t_{1,2} = (b_3 \cdot a_2)$	$t_{2,2} = (b_2 \cdot a_2) \oplus t_{1,3}$	$t_{3,2} = (b_1 \cdot a_2) \oplus t_{2,3}$
$t_{1,3} = (b_3 \cdot a_1)$	$t_{2,3} = (b_2 \cdot a_1) \oplus M_1 \oplus t_{1,4}$	$t_{3,3} = (b_1 \cdot a_1) \oplus M_2 \oplus t_{2,4}$
$t_{1,4} = (b_3 \cdot a_0)$	$t_{2,4} = (b_2 \cdot a_0) \oplus M_1$	$t_{3,4} = (b_1 \cdot a_0) \oplus M_2$

Row 4: $M_3 = t_{3,1}$, $i = 4$
$t_{4,1} = (b_0 \cdot a_3) \oplus t_{3,2}$
$t_{4,2} = (b_0 \cdot a_2) \oplus t_{3,3}$
$t_{4,3} = (b_0 \cdot a_1) \oplus M_3 \oplus t_{3,4}$
$t_{4,4} = (b_0 \cdot a_0) \oplus M_3$

m = 5: Primitive Polynomial $P(x) = x^5 + x^2 + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_4 \cdot a_4)$	$t_{2,1} = (b_3 \cdot a_4) \oplus t_{1,2}$	$t_{3,1} = (b_2 \cdot a_4) \oplus t_{2,2}$
$t_{1,2} = (b_4 \cdot a_3)$	$t_{2,2} = (b_3 \cdot a_3) \oplus t_{1,3}$	$t_{3,2} = (b_2 \cdot a_3) \oplus t_{2,3}$
$t_{1,3} = (b_4 \cdot a_2)$	$t_{2,3} = (b_3 \cdot a_2) \oplus M_1 \oplus t_{1,4}$	$t_{3,3} = (b_2 \cdot a_2) \oplus M_2 \oplus t_{2,4}$
$t_{1,4} = (b_4 \cdot a_1)$	$t_{2,4} = (b_3 \cdot a_1) \oplus t_{1,5}$	$t_{3,4} = (b_2 \cdot a_1) \oplus t_{2,5}$
$t_{1,5} = (b_4 \cdot a_0)$	$t_{2,5} = (b_3 \cdot a_0) \oplus M_1$	$t_{3,5} = (b_2 \cdot a_0) \oplus M_2$

m = 5: Primitive Polynomial $P(x) = x^5 + x^2 + 1$ *(continued)*

Row 4: $M_3 = t_{3,1}$, $i = 4$	*Row 5:* $M_4 = t_{4,1}$, $i = 5$
$t_{4,1} = (b_1 \cdot a_4) \oplus t_{3,2}$	$t_{5,1} = (b_0 \cdot a_4) \oplus t_{4,2}$
$t_{4,2} = (b_1 \cdot a_3) \oplus t_{3,3}$	$t_{5,2} = (b_0 \cdot a_3) \oplus t_{4,3}$
$t_{4,3} = (b_1 \cdot a_2) \oplus M_3 \oplus t_{3,4}$	$t_{5,3} = (b_0 \cdot a_2) \oplus M_4 \oplus t_{4,4}$
$t_{4,4} = (b_1 \cdot a_1) \oplus t_{3,5}$	$t_{5,4} = (b_0 \cdot a_1) \oplus t_{4,5}$
$t_{4,5} = (b_1 \cdot a_0) \oplus M_3$	$t_{5,5} = (b_0 \cdot a_0) \oplus M_4$

m = 6: Primitive Polynomial $P(x) = x^6 + x + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_5 \cdot a_5)$	$t_{2,1} = (b_4 \cdot a_5) \oplus t_{1,2}$	$t_{3,1} = (b_3 \cdot a_5) \oplus t_{2,2}$
$t_{1,2} = (b_5 \cdot a_4)$	$t_{2,2} = (b_4 \cdot a_4) \oplus t_{1,3}$	$t_{3,2} = (b_3 \cdot a_4) \oplus t_{2,3}$
$t_{1,3} = (b_5 \cdot a_3)$	$t_{2,3} = (b_4 \cdot a_3) \oplus t_{1,4}$	$t_{3,3} = (b_3 \cdot a_3) \oplus t_{2,4}$
$t_{1,4} = (b_5 \cdot a_2)$	$t_{2,4} = (b_4 \cdot a_2) \oplus t_{1,5}$	$t_{3,4} = (b_3 \cdot a_2) \oplus t_{2,5}$
$t_{1,5} = (b_5 \cdot a_1)$	$t_{2,5} = (b_4 \cdot a_1) \oplus M_1 \oplus t_{1,6}$	$t_{3,5} = (b_3 \cdot a_1) \oplus M_2 \oplus t_{2,6}$
$t_{1,6} = (b_5 \cdot a_0)$	$t_{2,6} = (b_4 \cdot a_0) \oplus M_1$	$t_{3,6} = (b_3 \cdot a_0) \oplus M_2$
Row 4: $M_3 = t_{3,1}$, $i = 4$	Row 5: $M_4 = t_{4,1}$, $i = 5$	Row 6: $M_5 = t_{5,1}$, $i = 6$
$t_{4,1} = (b_4 \cdot a_5) \oplus t_{3,2}$	$t_{5,1} = (b_1 \cdot a_5) \oplus t_{4,2}$	$t_{6,1} = (b_0 \cdot a_5) \oplus t_{5,2}$
$t_{4,2} = (b_4 \cdot a_4) \oplus t_{3,3}$	$t_{5,2} = (b_1 \cdot a_4) \oplus t_{4,3}$	$t_{6,2} = (b_0 \cdot a_4) \oplus t_{5,3}$
$t_{4,3} = (b_4 \cdot a_3) \oplus t_{3,4}$	$t_{5,3} = (b_1 \cdot a_3) \oplus t_{4,4}$	$t_{6,3} = (b_0 \cdot a_3) \oplus t_{5,4}$
$t_{4,4} = (b_4 \cdot a_2) \oplus t_{3,5}$	$t_{5,4} = (b_1 \cdot a_2) \oplus t_{4,5}$	$t_{6,4} = (b_0 \cdot a_2) \oplus t_{5,5}$
$t_{4,5} = (b_4 \cdot a_1) \oplus M_3 \oplus t_{3,6}$	$t_{5,5} = (b_1 \cdot a_1) \oplus M_4 \oplus t_{4,6}$	$t_{6,5} = (b_0 \cdot a_1) \oplus M_5 \oplus t_{5,6}$
$t_{4,6} = (b_4 \cdot a_0) \oplus M_3$	$t_{5,6} = (b_1 \cdot a_0) \oplus M_4$	$t_{6,6} = (b_0 \cdot a_0) \oplus M_5$

m = 7: Primitive Polynomial $P(x) = x^7 + x^3 + 1$

Row 1: $M_0 = 0$, $i = 1$	Row 2: $M_1 = t_{1,1}$, $i = 2$	Row 3: $M_2 = t_{2,1}$, $i = 3$
$t_{1,1} = (b_6 \cdot a_6)$	$t_{2,1} = (b_5 \cdot a_6) \oplus t_{1,2}$	$t_{3,1} = (b_4 \cdot a_6) \oplus t_{2,2}$
$t_{1,2} = (b_6 \cdot a_5)$	$t_{2,2} = (b_5 \cdot a_5) \oplus t_{1,3}$	$t_{3,2} = (b_4 \cdot a_5) \oplus t_{2,3}$
$t_{1,3} = (b_6 \cdot a_4)$	$t_{2,3} = (b_5 \cdot a_4) \oplus t_{1,4}$	$t_{3,3} = (b_4 \cdot a_4) \oplus t_{2,4}$
$t_{1,4} = (b_6 \cdot a_3)$	$t_{2,4} = (b_5 \cdot a_3) \oplus M_1 \oplus t_{1,5}$	$t_{3,4} = (b_4 \cdot a_3) \oplus M_2 \oplus t_{2,5}$
$t_{1,5} = (b_6 \cdot a_2)$	$t_{2,5} = (b_5 \cdot a_2) \oplus t_{1,6}$	$t_{3,5} = (b_4 \cdot a_2) \oplus t_{2,6}$
$t_{1,6} = (b_6 \cdot a_1)$	$t_{2,6} = (b_5 \cdot a_1) \oplus t_{1,7}$	$t_{3,6} = (b_4 \cdot a_1) \oplus t_{2,7}$
$t_{1,7} = (b_6 \cdot a_0)$	$t_{2,7} = (b_5 \cdot a_0) \oplus M_1$	$t_{3,7} = (b_4 \cdot a_0) \oplus M_2$
Row 4: $M_3 = t_{3,1}$, $i = 4$	**Row 5: $M_4 = t_{4,1}$, $i = 5$**	**Row 6: $M_5 = t_{5,1}$, $i = 6$**
$t_{4,1} = (b_3 \cdot a_6) \oplus t_{3,2}$	$t_{5,1} = (b_2 \cdot a_6) \oplus t_{4,2}$	$t_{6,1} = (b_1 \cdot a_6) \oplus t_{5,2}$
$t_{4,2} = (b_3 \cdot a_5) \oplus t_{3,3}$	$t_{5,2} = (b_2 \cdot a_5) \oplus t_{4,3}$	$t_{6,2} = (b_1 \cdot a_5) \oplus t_{5,3}$
$t_{4,3} = (b_3 \cdot a_4) \oplus t_{3,4}$	$t_{5,3} = (b_2 \cdot a_4) \oplus t_{4,4}$	$t_{6,3} = (b_1 \cdot a_4) \oplus t_{5,4}$
$t_{4,4} = (b_3 \cdot a_3) \oplus M_3 \oplus t_{3,5}$	$t_{5,4} = (b_2 \cdot a_3) \oplus M_4 \oplus t_{4,5}$	$t_{6,4} = (b_1 \cdot a_3) \oplus M_5 \oplus t_{5,5}$
$t_{4,5} = (b_3 \cdot a_2) \oplus t_{3,6}$	$t_{5,5} = (b_2 \cdot a_2) \oplus t_{4,6}$	$t_{6,5} = (b_1 \cdot a_2) \oplus t_{5,6}$
$t_{4,6} = (b_3 \cdot a_1) \oplus t_{3,7}$	$t_{5,6} = (b_2 \cdot a_1) \oplus t_{4,7}$	$t_{6,6} = (b_1 \cdot a_1) \oplus t_{5,7}$
$t_{4,7} = (b_3 \cdot a_0) \oplus M_3$	$t_{5,7} = (b_2 \cdot a_0) \oplus M_4$	$t_{6,7} = (b_1 \cdot a_0) \oplus M_5$

Row 7: $M_6 = t_{6,1}$, $i = 7$
$t_{7,1} = (b_0 \cdot a_6) \oplus t_{6,2}$
$t_{7,2} = (b_0 \cdot a_5) \oplus t_{6,3}$
$t_{7,3} = (b_0 \cdot a_4) \oplus t_{6,4}$
$t_{7,4} = (b_0 \cdot a_3) \oplus M_6 \oplus t_{6,5}$
$t_{7,5} = (b_0 \cdot a_2) \oplus t_{6,6}$
$t_{7,6} = (b_0 \cdot a_1) \oplus t_{6,7}$
$t_{7,7} = (b_0 \cdot a_0) \oplus M_6$

Careful examination of the Boolean Equations in all cases of $m = 3, 4, 5, 6, 7, 8$ clearly shows that a two-input AND gate and a two-input or three-input XOR gate are required to implement the function $t_{i,k}$ of each cell. It is thus possible to reuse a subset of the available 8×8 cells in Figure 3.1 to realize the logic function of the $m \times m$ cells for which $3 \leq m < 8$. Due to the sequential nature of the multiplication algorithm and the fact that each symbol is represented as an eight-

bit symbol whose $8-m$ most significant bits have been set to zero, the logic function $t_{i,k}$ of the $m \times m$ cells for $m < 8$ has been realized using the cells which occupy a square with coordinates $(9-m, 9-m)$, $(9-m, 8)$, $(8,8)$ and $(8, 9-m)$. Where necessary, redundant terms have been added to the Boolean equations of some of the 8×8 cells in rows 2 to 8. A simple relationship has been devised whereby each row of the $GF(2^8)$ uses a local controller which sets or clears the redundant terms in order to correctly implement the desired function $t_{i,k}$ for $m \leq 8$ using the same hardware. Each controller has been modelled as a multiplexer. Emphasis here has been placed on hardware reusability.

It should be noted that in the circuit implementation, the control variables $M_j, M_{j5}, M_{j6}, M_{j7}$, and M_{j-temp} have been introduced to the Boolean equations, defined in Equation (3.5), for $m = 8$ as overrides to allow the programmability of the multiplier for different $m = 3, 4, 5, 6, 7, 8$. Implementations of the various overrriding local cell equations are detailed below in algorithmic form.

The control variable M_j replaces M_1 in row 2 in cells $(2,4)$, $(2,5)$, $(2,6)$ and $(2,8)$ modifying them as follows:

$$
\begin{aligned}
t_{2,4} &= (b_6 \cdot a_4) \oplus M_j \oplus t_{1,5} \\
t_{2,5} &= (b_6 \cdot a_3) \oplus M_j \oplus t_{1,6} \\
t_{2,6} &= (b_6 \cdot a_2) \oplus M_j \oplus t_{1,7} \\
t_{2,8} &= (b_6 \cdot a_0) \oplus M_j
\end{aligned}
$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = t_{1,1}$;

else if $m = 7$ it sets $t_{1,3} = t_{1,4} = t_{1,5} = t_{1,6} = t_{1,7} = t_{1,8} = M_j = 0$;

end if;

Accordingly, row 2 of the $GF(2^8)$ also correctly implements row 1 of the function $t_{i,k}$ of the $GF(2^7)$ multiplier whose Boolean equations are defined in Equation (3.5).

The control variables M_{j7} and M_j have been introduced to row 3 in cells $(3,4)$, $(3,5)$, $(3,6)$ and $(3,8)$ modifying them as follows:

$$\begin{aligned} t_{3,4} &= (b_5 \cdot a_4) \oplus M_{j7} \oplus t_{2,5} \\ t_{3,5} &= (b_5 \cdot a_3) \oplus M_j \oplus t_{2,6} \\ t_{3,6} &= (b_5 \cdot a_2) \oplus M_{j7} \oplus t_{2,7} \\ t_{3,8} &= (b_5 \cdot a_0) \oplus M_j \end{aligned}$$

The local controller then operates as follows:

if $m = 8$ it sets $M_{j7} = M_j = t_{2,1}$;

else if $m = 7$ it sets $M_j = t_{2,2}$, $M_{j7} = 0$;

else if $m = 6$ it sets $t_{2,4} = t_{2,5} = t_{2,6} = t_{2,7} = t_{2,8} = M_j = M_{j7} = 0$;

end if;

Accordingly, row 3 of the $GF(2^8)$ correctly implements rows 2 and 1 of the function $t_{i,k}$ of the $GF(2^7)$ and $GF(2^6)$ multipliers respectively.

Control variables M_{j7}, M_{j-temp}, M_{j7} and M_j have been introduced to row 4 in cells $(4,4)$, $(4,5)$, $(4,6)$, $(4,7)$ and $(4,8)$ as follows:

$$\begin{aligned} t_{4,4} &= (b_4 \cdot a_4) \oplus M_{j7} \oplus t_{3,5} \\ t_{4,5} &= (b_4 \cdot a_3) \oplus M_{j-temp} \oplus t_{3,6} \\ t_{4,6} &= (b_4 \cdot a_2) \oplus M_{j7} \oplus t_{3,7} \\ t_{4,7} &= (b_4 \cdot a_1) \oplus t_{3,8} \oplus M_{j6} \\ t_{4,8} &= (b_4 \cdot a_0) \oplus M_j \end{aligned}$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = M_{j7} = M_{j-temp} = t_{3,1}, M_{j6} = 0$;

else if $m = 7$ it sets $M_j = M_{j-temp} = t_{3,2}, M_{j6} = M_{j7} = 0$;

else if $m = 6$ it sets $M_j = M_{j6} = t_{3,3}, M_{j7} = M_{j-temp} = 0$;

else if $m = 5$ it sets $t_{3,5} = t_{3,6} = t_{3,7} = t_{3,8} = M_{j6} = M_{j7} = M_{j-temp} = M_j = 0$;

end if;

The above procedure permits implementation of the logic functions of row 1 of the $GF(2^5)$ multiplier, row 2 of the $GF(2^6)$ multiplier, row 3 of the $GF(2^7)$ and row 4 of the $GF(2^8)$ multiplier using the same hardware.

Control variables M_{j7}, M_{j-temp}, M_{j5}, M_{j6} and M_j have been introduced to row 5 in cells $(5,4)$, $(5,5)$, $(5,6)$, $(5,7)$ and $(5,8)$ as follows:

$$
\begin{aligned}
t_{5,4} &= (b_3 \cdot a_4) \oplus M_{j7} \oplus t_{4,5} \\
t_{5,5} &= (b_3 \cdot a_3) \oplus M_{j-temp} \oplus t_{4,6} \\
t_{5,6} &= (b_3 \cdot a_2) \oplus M_{j7} \oplus M_{j5} \oplus t_{4,7} \\
t_{5,7} &= (b_3 \cdot a_1) \oplus t_{4,8} \oplus M_{j6} \\
t_{5,8} &= (b_3 \cdot a_0) \oplus M_j
\end{aligned}
$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = M_{j7} = M_{j-temp} = t_{4,1}, M_{j5} = M_{j6} = 0$;

else if $m = 7$ it sets $M_j = M_{j-temp} = t_{4,2}, M_{j5} = M_{j6} = M_{j7} = 0$;

else if $m = 6$ it sets $M_j = M_{j6} = t_{4,3}, M_{j5} = M_{j7} = M_{j-temp} = 0$;

else if $m = 5$ it sets $M_j = M_{j5} = t_{4,4}, M_{j6} = M_{j7} = M_{j-temp} = 0$;

else if $m = 4$ it sets $M_{j7} = M_{j-temp} = M_{j5} = M_j = M_{j6} = t_{4,6} = t_{4,7} = t_{4,8} = 0$;

end if;

Control variables M_{j7}, M_{j-temp}, M_{j5}, M_{j6} and M_j have been introduced to row 6 in cells $(6,4)$, $(6,5)$, $(6,6)$, $(6,7)$ and $(6,8)$ as follows:

$$\begin{aligned} t_{6,4} &= (b_2 \cdot a_4) \oplus M_{j7} \oplus t_{5,5} \\ t_{6,5} &= (b_2 \cdot a_3) \oplus M_{j-temp} \oplus t_{5,6} \\ t_{6,6} &= (b_2 \cdot a_2) \oplus M_{j7} \oplus M_{j5} \oplus t_{5,7} \\ t_{6,7} &= (b_2 \cdot a_1) \oplus t_{5,8} \oplus M_{j6} \\ t_{6,8} &= (b_2 \cdot a_0) \oplus M_j \end{aligned}$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = M_{j7} = M_{j-temp} = t_{5,1}, M_{j5} = M_{j6} = 0$;

else if $m = 7$ it sets $M_j = M_{j-temp} = t_{5,2}, M_{j5} = M_{j6} = M_{j7} = 0$;

else if $m = 6$ it sets $M_j = M_{j6} = t_{5,3}, M_{j5} = M_{j7} = M_{j-temp} = 0$;

else if $m = 5$ it sets $M_j = M_{j5} = t_{5,4}, M_{j6} = M_{j7} = M_{j-temp} = 0$;

else if $m = 4$ it sets $M_{j7} = M_{j-temp} = 0, M_j = M_{j6} = t_{5,5}$;

else if $m = 3$ it sets $M_{j7} = M_{j6} = M_{j5} = M_j = t_{5,7} = t_{5,8} = 0$;

end if;

Control variables M_{j7}, M_{j-temp}, M_{j5}, M_{j6} and M_j have been introduced to row 7 in cells $(7,4)$, $(7,5)$, $(7,6)$, $(7,7)$ and $(7,8)$ as follows:

$$\begin{aligned} t_{7,4} &= (b_1 \cdot a_4) \oplus M_{j7} \oplus t_{6,5} \\ t_{7,5} &= (b_1 \cdot a_3) \oplus M_{j-temp} \oplus t_{6,6} \\ t_{7,6} &= (b_1 \cdot a_2) \oplus M_{j7} \oplus M_{j5} \oplus t_{6,7} \\ t_{7,7} &= (b_1 \cdot a_1) \oplus t_{6,8} \oplus M_{j6} \\ t_{7,8} &= (b_1 \cdot a_0) \oplus M_j \end{aligned}$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = M_{j7} = M_{j-temp} = t_{6,1}, M_{j5} = M_{j6} = 0$;

else if $m = 7$ it sets $M_j = M_{j-temp} = t_{6,2}, M_{j5} = M_{j6} = M_{j7} = 0$;

else if $m = 6$ it sets $M_j = M_{j6} = t_{6,3}, M_{j5} = M_{j7} = M_{j-temp} = 0$;

else if $m = 5$ it sets $M_j = M_{j5} = t_{6,4}, M_{j6} = M_{j7} = M_{j-temp} = 0$;

else if $m = 4$ it sets $M_{j7} = M_{j-temp} = 0, M_j = M_{j6} = t_{6,5}$;

else if $m = 3$ it sets $M_{j7} = M_{j5} = 0, M_j = M_{j6} = t_{6,6}$;

end if;

Finally, the control variables M_{j7}, M_{j-temp}, M_{j5}, M_{j6} and M_j have been introduced to row 8 in cells $(8,4)$, $(8,5)$, $(8,6)$, $(8,7)$ and $(8,8)$ as follows:

$$
\begin{aligned}
t_{8,4} &= (b_0 \cdot a_4) \oplus M_{j7} \oplus t_{7,5} \\
t_{8,5} &= (b_0 \cdot a_3) \oplus M_{j-temp} \oplus t_{7,6} \\
t_{8,6} &= (b_0 \cdot a_2) \oplus M_{j7} \oplus M_{j5} \oplus t_{7,7} \\
t_{8,7} &= (b_0 \cdot a_1) \oplus t_{7,8} \oplus M_{j6} \\
t_{8,8} &= (b_0 \cdot a_0) \oplus M_j
\end{aligned}
$$

The local controller then operates as follows:

if $m = 8$ it sets $M_j = M_{j7} = M_{j-temp} = t_{7,1}, M_{j5} = M_{j6} = 0$;

else if $m = 7$ it sets $M_j = M_{j-temp} = t_{7,2}, M_{j5} = M_{j6} = M_{j7} = 0$;

else if $m = 6$ it sets $M_j = M_{j6} = t_{7,3}, M_{j5} = M_{j7} = M_{j-temp} = 0$;

else if $m = 5$ it sets $M_j = M_{j5} = t_{7,4}, M_{j6} = M_{j7} = M_{j-temp} = 0$;

else if $m = 4$ it sets $M_{j7} = M_{j-temp} = 0, M_j = M_{j6} = t_{7,5}$;

else if $m = 3$ it sets $M_{j7} = M_{j5} = 0, M_j = M_{j6} = t_{7,6}$;

end if;

Another controller assigns the output, i.e. product, elements as follows:

if $m = 7$ it sets $t_{8,1} = 0$;

else if $m = 6$ it sets $t_{8,1} = t_{8,2} = 0$;

else if $m = 5$ it sets $t_{8,1} = t_{8,2} = t_{8,3} = 0$;

else if $m = 4$ it sets $t_{8,1} = t_{8,2} = t_{8,3} = t_{8,4} = 0$;

else if $m = 3$ it sets $t_{8,1} = t_{8,2} = t_{8,3} = t_{8,4} = t_{8,5} = 0$;

end if;

followed by

$C_0 = t_{8,8}, C_1 = t_{8,7}, C_2 = t_{8,6}, C_3 = t_{8,5}, C_4 = t_{8,4}, C_5 = t_{8,3}, C_6 = t_{8,2}, C_7 = t_{8,1}$

according to Figure 3.1.

Based on the above analysis, it can be seen that a two-input AND gate and a two-input or three-input XOR gate implements the function $t_{i,k}$. Registers and D-flipflops have been placed between adjacent rows in order to facilitate pipeline processing of data between neighbouring cells. The pipelined version of this m-programmable multiplier outputs the product C at a rate of one output per cycle after an initial delay of m cycles. The clock period is governed by the propagation delay of a signal through a multiplexer, a 2-input AND gate and a 2-input or 3-input XOR gate.

The resulting $GF(2^m)$ multiplier is systolic and has a simple, regular communication and control structure. It also allows unidirectional data flow which is advantageous over a system with contraflowing data streams. Most fault tolerance schemes which are suitable for linear arrays route information around faulty cells [68][69]. This can introduce significant transmission delays between cells. In unidirectional data flow arrays, latches are often inserted in all data streams which are

rerouted around a faulty cell but at the expense of increased system latency. This does not change the required data interactions, since the relative delays between all data paths are zeros. This technique is not suitable for arrays with contraflowing data streams because the relative delay between paths would be non-zero and hence data interactions may be corrupted.

The symbolic architecture of the multiplier is shown in Figure 3.2. A and B are the 8-bit elements to be multiplied; clk is the clock signal; m is the symbol size; *test_se*, *test_si*, *test_so* are the test ports; O is the 8-bit product of A and B.

A comparison of the unpipelined and pipelined multiplier is shown in Table 3.1. The number of gates with and without scan chain, number of detected faults and fault coverage are automatically generated by the Synopsys synthesis tools. The maximum clock frequency is estimated by interactively simulating the VHDL gate level netlist file, using repeated functional verification and timing analysis techniques. The pipelined version has a higher gate count because of the added registers between neighbouring cells. Both versions of the multiplier have a 100% fault coverage which ensures high quality and ease of testing after fabrication. The multiplexed scan chain improves the controllability and observability of the internal circuit nodes, thereby reducing the complexity of test generation.

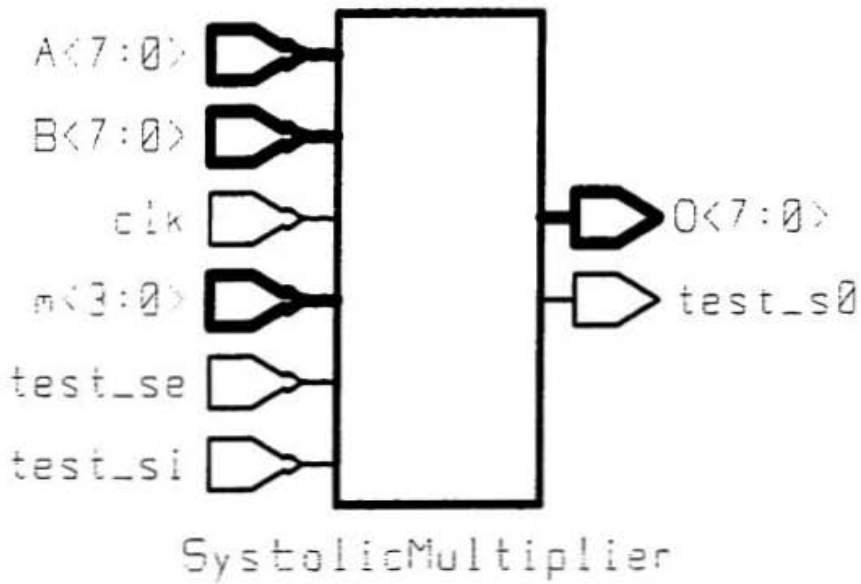

Figure 3.2: Symbolic Architecture of the Programmable Multiplier

Circuit Properties	Unpipelined	Pipelined
Latency	1	m
Throughput rate	1	1
Number of Gates	517	2,583
Number of Gates with Scan Chain	551	3,419
Number of Detected Faults	2050	8892
Maximum Clock Frequency (MHz)	60	200
Fault Coverage %	100	100

Table 3.1: Comparison of the programmable Unpipelined and Pipelined Multiplier

The design procedure can be summarized as follows:

1. For a selected $\widehat{m}$, derive all the Boolean Equations $t_{i,k}$ for all the $\widehat{m}^2$ cells. Also derive the Boolean Equations for $m < \widehat{m}$ using $t_{i,k} = t_{i-1,k+1} \oplus (p_{m-k} \cdot M_{i-1}) \oplus (a_{m-k}.b_{m-i})$ where $i = 1, 2, ..., m$ and $k = 1, 2, ..., m$.

2. Beginning with $m = \widehat{m} - 1$ and adding control variables to each cell where appropriate, restrict implementation of $t_{i,k}$ to a square with coordinates $(\widehat{m} + 1 - m, \widehat{m} + 1 - m)$, $(\widehat{m} + 1 - m, \widehat{m})$, $(\widehat{m}, \widehat{m})$ and $(\widehat{m}, \widehat{m} + 1 - m)$. Repeat the procedure for all $m = \widehat{m} - 2, \widehat{m} - 3$, ..., 4,3.

3. Add registers between neighbouring cells to obtain the pipelined version of the multiplier.

3.2 m-Programmable Exponentiator/Inverter

Definition 1: For an arbitrary element A in the finite field $GF(2^m)$, the inverse of an element A is denoted by $A^{-1} = A^{2^m-2}$ [10]. Rewriting the exponent $2^m - 2$ as $2^1 + 2^2 + 2^3 + ... + 2^{m-1}$, allows the inverse operation to be expressed as

$$A^{-1} = (A^2) \cdot (A^{2^2}) \cdot (A^{2^3}) \cdots (A^{2^{m-1}}) \tag{3.6}$$

Definition 2: For an arbitrary element A in the finite field $GF(2^m)$ and an integer $N(1 \leq N \leq 2^m - 1)$, the exponentiation function is defined as $\delta = A^N$. Clearly, δ is in $GF(2^m)$. If N is represented in binary form as $n_0, n_1, n_2, ..., n_{m-1}$ such that $N = \sum_{i=0}^{m-1} n_i \cdot 2^i$, then $\delta = A^N$ can be expressed as follows [30][31]

$$\begin{aligned} \delta &= A^N = A^{\sum_{i=0}^{m-1} n_i \cdot 2^i} \\ &= (A)^{n_0} \cdot (A^2)^{n_1} \cdot (A^{2^2})^{n_2} \cdots (A^{2^{m-1}})^{n_{m-1}} \\ &= \prod_{i=0}^{m-1} (A^{2^i})^{n_i} \\ &= \prod_{i=0}^{m-1} E_i \end{aligned} \tag{3.7}$$

where

$$E_i = A^{2^i} \quad \text{if } n_i = 1 \tag{3.8}$$

$$E_i = 1 \quad \text{if } n_i = 0 \tag{3.9}$$

Assuming the temporary result is $R_j = \prod_{i=0}^{j} E_i$, then the following recursion is derived.

$$\begin{aligned} R_0 &= 1 \cdot E_0, \\ R_1 &= R_0 \cdot E_1, \\ &\vdots \\ R_k &= R_{k-1} \cdot E_k, \\ &\vdots \\ R_{m-1} &= R_{m-2} \cdot E_{m-1} \\ &= A^N \end{aligned} \tag{3.10}$$

By using the definition for the exponentiation function, an alternate method can be derived to evaluate the inverse of an element in $GF(2^m)$. Equations (3.6) and (3.7) show that if $N = (n_0, n_1, n_2, ..., n_{m-1})$ such that $N = \sum_{i=0}^{m-1} n_i \cdot 2^i$ as in Definition 2, then the inverse function is a special case of exponentiation. They are equivalent, that is $A^N = A^{-1}$, if and only if the following conditions are satisfied

$$\begin{aligned} n_0 &= 0 \text{ and} \\ n_1 &= n_2 = ... = n_{m-1} = 1 \end{aligned} \tag{3.11}$$

These conditions are always valid as we can observe that Equation (3.6) can be restructured as Equation (3.7) in the form

$$A^{-1} = (A)^{n_0} \cdot (A^2)^{n_1} \cdot (A^{2^2})^{n_2} \cdots (A^{2^{m-1}})^{n_{m-1}} = A^N \tag{3.12}$$

when $N = (n_0, n_1, n_2, ..., n_{m-1}) = (0, 1, 1, ..., 1)$

Henceforth, similar to the exponentiation function, the inverse element can be computed as

$$\begin{aligned} A^{-1} &= A^{\sum_{i=0}^{m-1} n_i \cdot 2^i} \\ &= (A)^0 \cdot (A^2)^1 \cdot (A^2)^1 \cdots (A^{2^{m-1}})^1 \\ &= \prod_{i=0}^{m-1} (A^{2^i}) \\ &= \prod_{i=0}^{m-1} E_i \end{aligned} \tag{3.13}$$

where

$$\begin{aligned} E_i &= A^{2^i} \text{ if } i \neq 0 \\ E_0 &= 1 \text{ if } i = 0 \end{aligned} \tag{3.14}$$

Let the temporary result be $R_j = \prod_{i=0}^{m-1} E_i$, then the following recursion is also obtained,

$$\begin{aligned} R_0 &= 1, \\ R_1 &= R_0 \cdot E_1, \\ &\vdots \\ R_k &= R_{k-1} \cdot E_k, \\ &\vdots \\ R_{m-1} &= R_{m-2} \cdot E_{m-1} \\ &= A^{-1} \end{aligned} \tag{3.15}$$

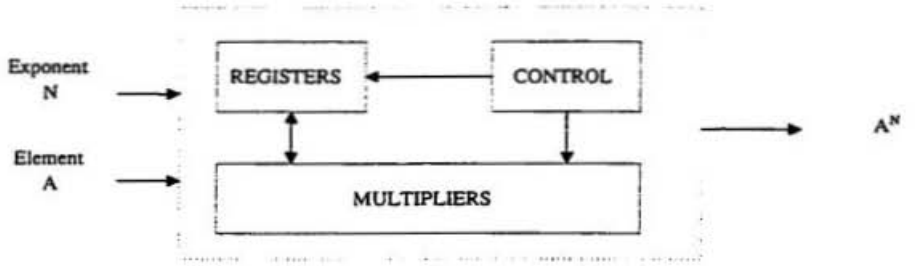

Figure 3.3: A General Exponentiation Architecture

From a hardware implementation point of view, the exponentiation architecture can be used to compute the inverse element as well. It can be implemented using registers to hold the data, control circuitry and repeated use of a single multiplier or use of multipliers in parallel. According to the above analysis, multiplication stands out as the most critical arithmetic operation. Thus, the ideal multiplier circuit structure must be modular, easily expandable and require a simple control scheme. A global system diagram comprising the three main components is depicted in Figure 3.3.

In the case where $m = 8$, one only needs to set $N = (n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7) = (0, 1, 1, 1, 1, 1, 1, 1)$ in order to evaluate inverse elements by using the same exponentiation hardware. A structure for computing exponentiation or inverse elements of the $GF(2^8)$ is shown in Figure 3.4. It is an extended version of the array described in [31].

The word-level systolic array consists of 14 multipliers (MUL_1 to MUL_{14}), 8 8-bit multiplexers (MUX_0 to MUX_7), 28 1-bit one-cycle delay elements (D_1) and one 8-bit one cycle delay element (D_8).

The multipliers on the left bank (MUL_1 to MUL_7) evaluate A^{2^i} for $i = 1, 2, ..., m-$

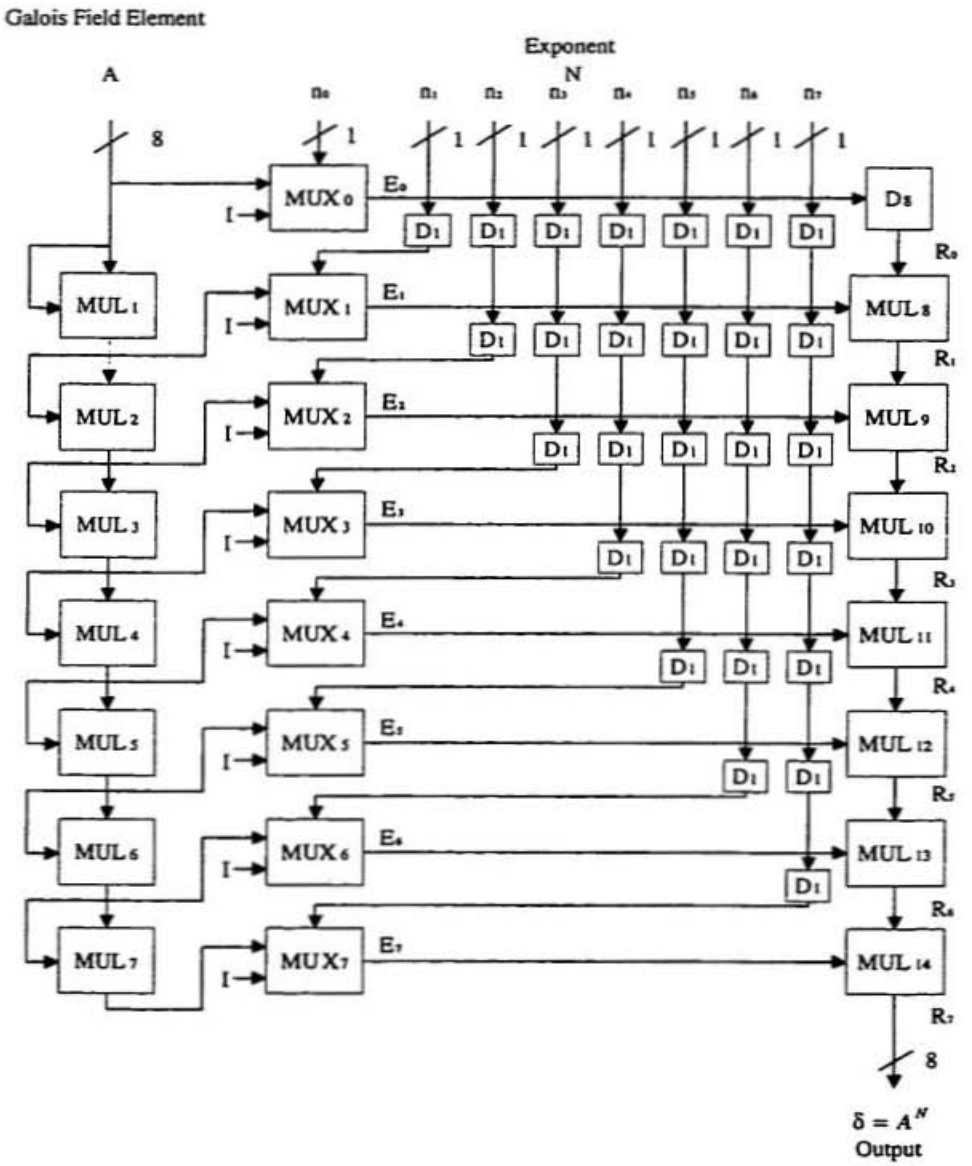

Figure 3.4: Exponentiation/Inverse Architecture for $GF(2^8)$

Latency	m
Throughput rate	1
Number of Gates	7,735
Number of Gates with Scan Chain	8,821
Number of Detected Faults	29,705
Fault Coverage %	99.8

Table 3.2: Features of the programmable Exponentiator/Inverter

1 while those on the right bank (MUL_8 to MUL_{14}) evaluate $R_i = R_{i-1} \cdot E_i$ for $i = 1$ to $m - 1$. The multiplexers (MUX_0 to MUX_7) select A^{2^i} if $n_i = 1$ or the 8-bit identity element $I =$ "00000001" if $n_i = 0$ as the output E_i.

Thus the output $\delta = A^N$ is available as R_7. The m-programmable $GF(2^m)$ multiplier has been used to implement the MUL_i such that the output is also accessible at various points R_{m-1} for $m = 3, 4, 5, 6, 7, 8$ as specified in Equations (3.10) and (3.15). The output points R_2 to R_7 are directly connected to an independent module which assigns them to $\delta = A^N$ based on the word size m.

The symbolic structure of the combined exponentiation/inverse architecture is shown in Figure 3.5. *CHIPmode* configures the chip to operate as an exponentiator or inverter; *ExponentIn* is the port for the exponent; *GF_ELEMENT* is the Galois field element A; *clkIn* is the clock signal; m is the symbol size; *test_se*, *test_si*, *test_so* are the test ports; *VALUE* is the 8-bit inverse or exponentiation of *GF_ELEMENT*.

Its circuit properties are shown in Table 3.2. The number of gates with and without scan chain, number of detected faults and fault coverage are automatically generated by the Synopsys synthesis tools.

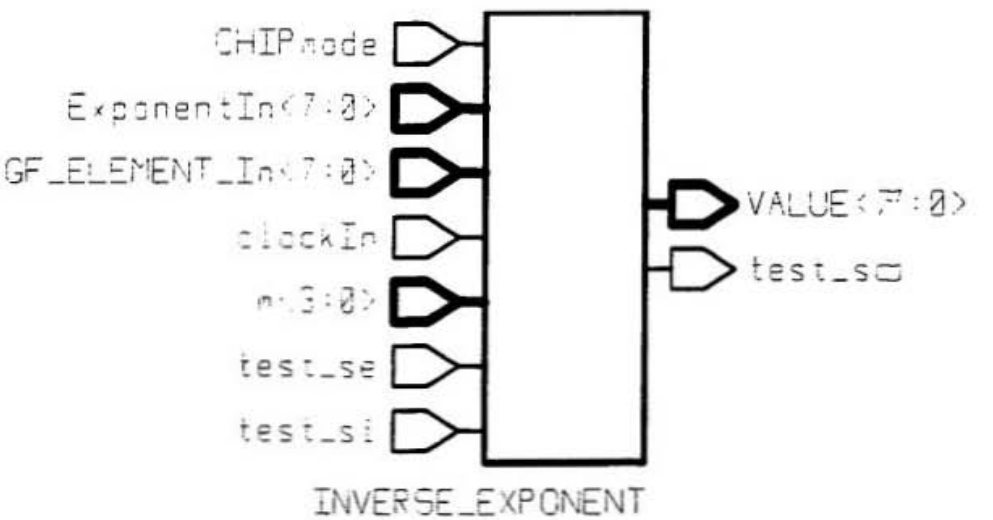

Figure 3.5: Symbolic Architecture of the Programmable Exponentiator/Inverter

A comparison of the programmable exponentiator/inverter *when operating as an inverter for fixed* m with other inversion circuits is illustrated in Table 3.3. The algorithm in [22] requires a computational complexity of $O(m \log_2 m)$. p is the number of ones in the binary expression of $m - 1$ and q is the lower bound on $\log_2 m$. If the input data pass in continuously, the VHDL gate-level simulations show that the parallel-in parallel-out inverter can produce results at a rate of one output per clock cycle after a latency of m cycles. The new inverter is flexible and clearly outperforms circuits proposed in [22][23].

Since the standard basis is commonly used in implementing algebraic RS decoders in hardware, the exponentiator/inverter is implicitly based on the normal basis, and therefore exponentiation can be easily implemented via cyclic shifts. However, additional circuitry would be required to convert between the normal and standard basis representation of the Galois field elements, thus making the design of the RS decoder more complex.

Circuit Properties	Proposed Inverter	Inverter in [23]	Inverter in [22]
Latency	m	$7m-3$	$m(p+q)$
Throughput	1	$2m-1$	$m(p+q)$
Computational Complexity	$O(1)$	$O(m)$	$O(m\log_2 m)$
Regularity	High	High	Moderate
Dependence on Primitive Polynomial	Yes	No	Yes
Basis	Standard	Standard	Normal
I/O format	Parallel-In Parallel-Out	Serial-In Serial-Out	Serial-In Serial-In

Table 3.3: Comparison of Exponentiator/Inverter with other Inverters for fixed m

3.3 Discussion and Summary

An m-programmable Galois field multiplier, which can operate in the $GF(2^m)$, where m is variable, has been presented. It uses a simple controller in some of the basic cells. The cases where $m = 3, 4, 5, 6, 7$ and 8 have been considered. Its pipelined version has a speedup factor of about four. Using this multiplier and the modified version of the word level systolic array for exponentiation discussed in [31], it has been discovered that both the inverse and exponentiation functions can be evaluated using the same hardware structure. The results show that the proposed method of performing inversion of Galois field elements is more efficient and faster than available circuits. These arithmetic circuits are systolic and have simple, regular communication and control structures. They also allow unidirectional data flow which is advantageous over systems with contraflowing data streams [68][69]. A very high fault coverage has been obtained by using a full scan test methodology which uses multiplexed flip-flops. This means they will be easy to test using automatic test equipment after fabrication [70]. All the gate-level simulations for the proposed architectures have been performed using the Synopsys VHDL System Simulator. These programmable arithmetic circuits are easily expandable, hence can be tailored for a wide range of applications requiring variable symbol size m.

Chapter 4

Synthesis of the Reed-Solomon Encoder/Decoder ASIC

This chapter presents the design methodology, circuit synthesis and functional verification of the major modules of the RS encoder/decoder ASIC.

4.1 Design Flow, Functional Verification and Test

The circuit synthesis of the RS encoder/decoder ASIC was realized using the 0.8-μm BiCMOS design kits for Synopsys and Cadence tools licensed by the Canadian Microelectronics Corporation (CMC).

The design flow made use of a 0.8-μm CMOS standard cell library, which did not include any bipolar junction transistors (BJTs), provided in the BiCMOS fabrication software. It supported a top down VLSI design methodology in which the functional abstraction of the digital IC could be initially specified using the VHDL hardware description language. The circuit models were described using a subset of the VHDL constructs called Register Transfer Levels (RTL). Once logic simulation was completed and verified, the RTL circuit models were then synthesized to obtain the gate level (structural) circuit models using the Synopsys suite

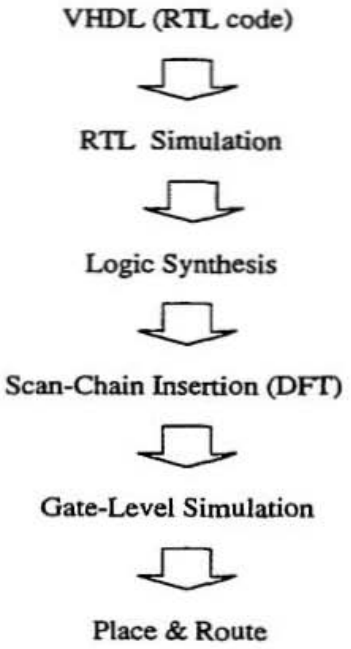

Figure 4.1: Design Flow

of tools. The design could then be imported into the Cadence environment as a Verilog gate level netlist file, automatically generated by the Synopsys tools. It could then be automatically placed and routed in order to create the IC masks required for the fabrication process. These steps were independent of each other and are summarized in Figure 4.1.

As shown in Figure 4.1 the modelling, verification and implementation processes were integrated. The integrated design flow reduced the amount of code that had to be maintained and the risk of inconsistencies between models. Thus, an error free transfer was ensured between all the levels in the design process. Rapid design exploration of the different architectural styles could easily be made. The Synopsys

tools focus on the composition of the datapath and global cycle count while the Cadence suite of tools concentrate on the creation of the integrated circuit layout.

Once the behavioral model of the RS encoder and decoder had been captured using the VHDL hardware description language, each block was then partitioned into smaller modules which were modelled separately using a subset of the VHDL constructs suitable for logic synthesis. The size of each synthesizable module varied from 45 to a maximum of 20,000 gates although a reasonable gate count (250 to 5,000 gates per module) was recommended in order to reduce the compile time [71]. Larger modules were characterized by sequential processes which had heavy dataflow dependencies. They required large CPU time, huge memory and logic synthesis run times of up to three days. The functional correctness of each VHDL RTL model was verified using an interactive UNIX based RS encoder/decoder simulator written in C [72][73].

Hierarchical compile is the simplest method for compiling a hierarchical design [74]. However, a bottom-up compile strategy whereby individual modules were compiled first followed by higher modules, was adopted. This way once a module had been compiled, it was assigned a *dont_touch* flag so that Synopsys did not need to compile or read it again. The bottom-up compile method worked well when the entire chip was synthesized into gates since the entire design was not required to be stored in memory. Unlike the hierarchical compile, this led to significant savings in CPU and swap space. The design rules were checked and an initial fault coverage reported on each module as it was developed. The report helped identify the block that had design rule violations or an unacceptable fault coverage so that testability problems could be fixed at an early stage. Testability analysis was then performed

on the top-level core design before scan insertion, because test design rule violations could be introduced due to interconnect between the hierarchical blocks. At the time, a partial or full scan test methodology which used the multiplexed flip-flops was the only design-for-testability (DFT) style supported in the 0.8-μm BiCMOS technology.

The architecture of the chip and its major modules are described in the following section.

4.2 Chip Architecture

This thesis implements an algebraic encoder/decoder chip using CMOS standard cells. The standard algebraic decoder for decoding RS codes is described in detail in Chapter 2. The complex arithmetic operations needed in the encoder and decoder generally require the use of the m-programmable multiplier and exponentiator/inverter proposed in Chapter 3. As previously reported, the first step in the decoding algorithm is to calculate the syndrome polynomial $S(x)$ which contains the information to correct correctable errors or detect uncorrectable errors. The Berlekamp-Massey or Euclid's algorithm can be used to determine the error-locator polynomial [3]. The Berlekamp-Massey algorithm was selected because its low design complexity makes it suitable for VLSI synthesis. Another module exists to determine the error magnitude polynomial using the relationship between the syndrome and error location polynomials, i.e., $\Omega(x) = S(x)\sigma(x) \bmod x^{2t}$ or $\Omega(x) = S(x)\Lambda(x) \bmod x^{2t}$. Once the location and magnitude of the errors have been determined using the Chien Search and the Forney algorithm respectively, the received messages can be corrected.

Sizes of the field	$GF(2^8)$,$GF(2^7)$,$GF(2^6)$ $GF(2^5)$,$GF(2^4)$,$GF(2^3)$
Primitive Polynomials, $P(x)$	$x^8+x^4+x^3+x^2+1$
	x^7+x^3+1
	x^6+x+1
	x^5+x^2+1
	x^4+x+1
	x^3+x+1
Generator Polynomial, $G(x)$	$(x+\alpha)(x+\alpha^2)...(x+\alpha^{2t})$
Error Correction Capability, t	t=1,2,3,...,16
Encoder Latency	n
Decoder Latency	$2n+2t+m+3$
Gate Count	218,206
Technology	0.8-μm BiCMOS

Table 4.1: RS Encoder/Decoder Characteristics

The symbolic architecture of the programmable RS encoder/decoder is shown in Figure 4.2. Values of the Galois field symbol size $m = 3, 4, 5, 6, 7, 8$ and error correction capability t ranging from 1 to 16 are supported. Thus the block length $n = 2^m - 1$ varies from 7 to 255 symbols. The device contains 218,206 gates, where a gate is defined as a 2-input NAND gate. Its characteristics are given in Table 4.1.

The size and description of each I/O signal are given in Table 4.2.

Implementation details of the main modules of the RS encoder/decoder are described in the following subsections. All the required arithmetic operations in Galois fields are accessible to global *entities* as *components* or *functions* defined in a VHDL package. Each module has been modelled in VHDL using component instantiation.

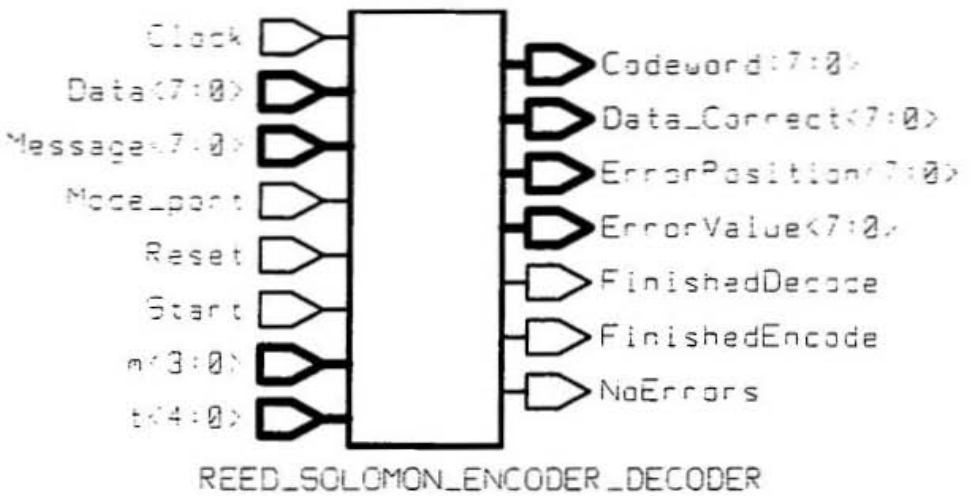

Figure 4.2: Symbolic diagram of the RS Encoder/Decoder

Signal	Bit Size	Description
Clock	1	clock signal
Data	8	input port for the received word *from channel*
Message	8	input port for the message symbols
Mode_port	1	selects the encoding or decoding mode
Reset	1	reset signal
Start	1	starts the encoding or decoding process
m	4	Galois field symbol size
t	5	error correction capability
Codeword	8	output port for the codeword polynomial *to channel*
Data_Correct	8	output port for the corrected data
ErrorPosition	8	output port for the error positions *from Chien Search*
ErrorValue	8	output port for the error value/magnitude *from Forney Algorithm*
FinishedDecode	1	goes high when the decoding process is complete
FinishedEncode	1	goes high when the encoding process is complete
NoErrors	1	goes high when there is a functional error

Table 4.2: RS Encoder/Decoder I/O Pins

4.3 Modules

4.3.1 Encoder

Let the message polynomial be $M(x) = c_{2t}x^{2t} + c_{2t+1}x^{2t+1} + ... + c_{n-1}x^{n-1}$ and the parity check polynomial be $\widehat{P}(x) = c_o + c_1x + ... + c_{2t-1}x^{2t-1}$. Then the encoded RS code polynomial, often called the *codeword*, can be expressed as

$$\begin{aligned} C(x) &= M(x) + \widehat{P}(x) \\ &= Q(x)G(x) \\ \Rightarrow M(x) &= Q(x)G(x) - \widehat{P}(x) \end{aligned} \tag{4.1}$$

The quantity $Q(x)G(x)$ means that a valid code polynomial $C(x)$ must also be a multiple of the generator polynomial $G(x)$. Hence, the encoder must find $\widehat{P}(x)$ from $M(x)$ and $G(x)$. This is achieved by the division algorithm. That is, dividing $M(x)$ by $G(x)$ gives the remainder polynomial $\widehat{R}(x)$ such that

$$M(x) = Q(x)G(x) + \widehat{R}(x) \tag{4.2}$$

where $Q(x)$ is the quotient.

In this thesis, the RS encoder uses a conventional architecture to perform the division of $M(x)$ by $G(x)$ to obtain the parity check polynomial $\widehat{P}(x) = -\widehat{R}(x)$ defined in Equation (4.1). Its structure is shown in Figure 4.3.

In the figure, $G(i)'s$ are the symmetrical coefficients of $G(x)$. Initially all the registers are cleared and both switches set to position A. The message symbols $c_{n-1}, ..., c_{2t}$ are fed into the division circuit and are also transmitted from the encoder symbol by symbol every clock cycle. Immediately after k clock cycles, both switches are set to position B to allow the parity check symbols to be serially shifted out of the encoder to form the complete $C(x)$. The shifting process takes $2t$ clock cycles.

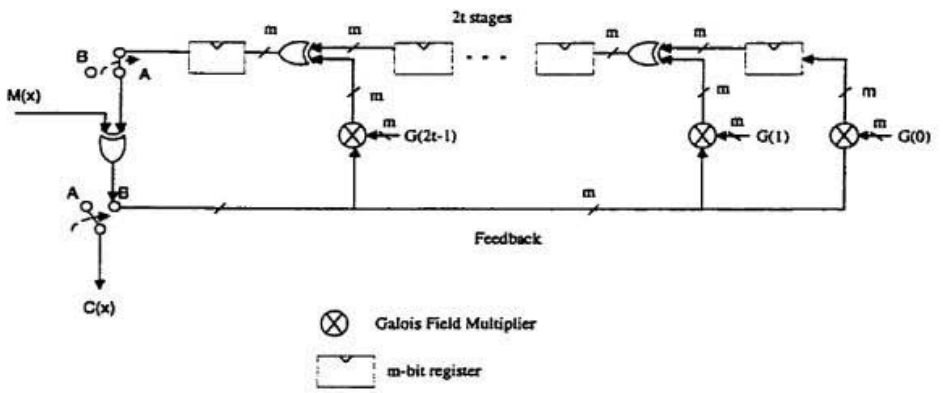

Figure 4.3: RS Encoder

The encoder module consists of 9,644 gates. It has been designed as a 32-stage 8-bit linear feedback shift register (LFSR) whose components include 16 m-programmable Galois field multipliers, a modulo-255 counter and 8-bit registers. m and t are variable for $3, 4, 5, 6, 7, 8$ and $1, 2, 3, ..., 16$ respectively. If m had been fixed as in [35][65], the multipliers for the coefficients $G(i)'s$ and the feedback connection could have been designed as constant multipliers consisting of a tree of XOR gates. This version of the encoder is also available for fixed $m = 8$ and fixed $t = 16$. The VHDL code for any constant multiplier in $GF(2^m)$ can be automatically generated using a C++ program written by the author.

Figure 4.4 shows a symbolic diagram for the RS encoder. The calculation of the various coefficients of $G(x)$ is generally tedious and hence was automated using a C program available in [72][73]. Their values are heavily dependent on t and m and are stored in the appropriate registers during chip initialization.

The bit size and description of each I/O signal are given in Table 4.3.

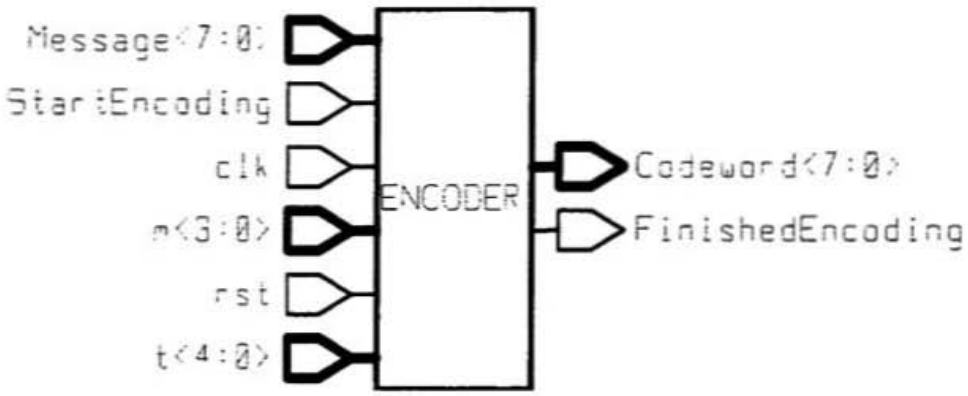

Figure 4.4: Symbolic diagram of the RS encoder

Signal	Bit Size	Description
Message	8	input port for the message symbols
StartEncoding	1	starts the encoding process
clk	1	clock signal
m	4	Galois field symbol size
rst	1	reset signal
t	5	error correction capability
Codeword	8	output port for the codeword polynomial *to channel*
FinishedEncoding	1	goes high when encoding is complete

Table 4.3: Encoder I/O Pins

4.3.2 Syndrome

As noted earlier, the $2t$ syndromes or syndrome polynomial coefficients are computed as

$$S_j = \sum_{i=0}^{n-1} r_i \alpha^{ij}, \quad 2^{m-1} - t \leq j \leq 2^{m-1} + t - 1 \tag{4.3}$$

where $r_i (0 \leq i \leq n-1)$ are the coefficients of the received polynomial $R(x)$.
By using Horner's rule, Equation (4.3) can be rewritten as

$$S_j = (...(((r_{n-1}\alpha^j + r_{n-2})\alpha^j + r_{n-3})\alpha^j + ...r_1)\alpha^j + r_0) \tag{4.4}$$

Figure 4.5 shows a block diagram for computing the syndrome values defined in Equation (4.4).

As shown in the figure, each cell implements the following register transfer relation:

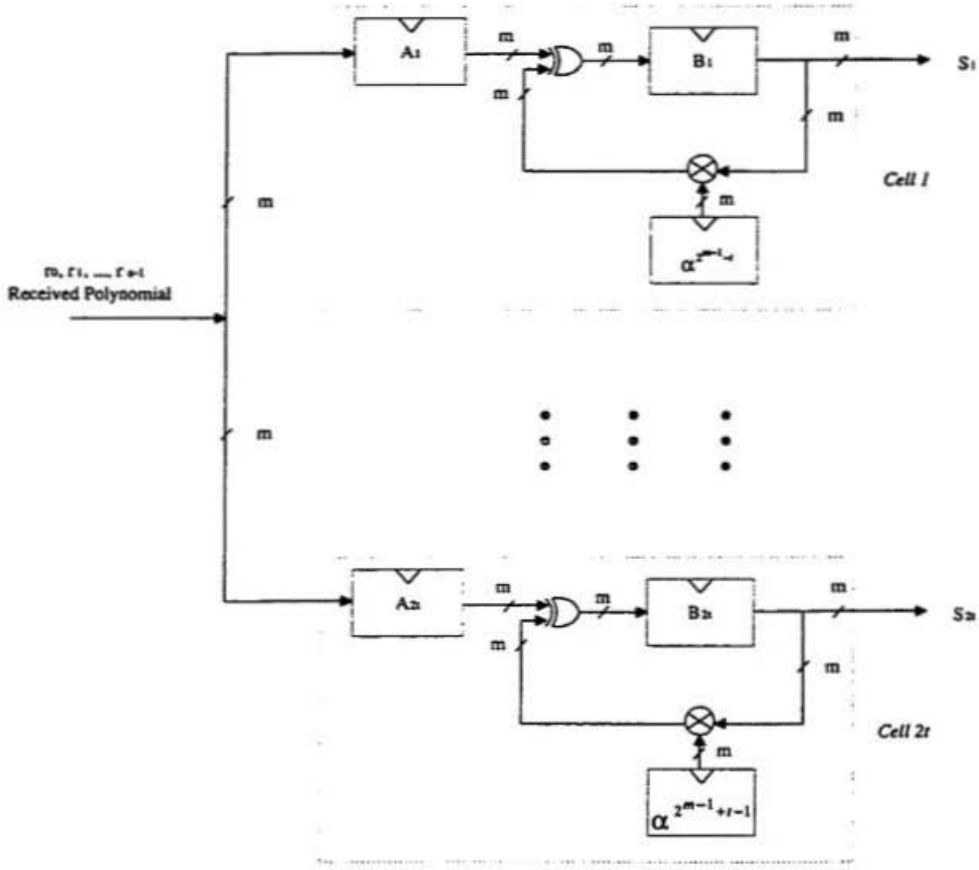

Figure 4.5: Systolic Array to compute Syndrome Polynomial

$$
\begin{aligned}
&cell\ 1: \quad B_1 \leftarrow A_1 \oplus B_1\alpha^{2^{m-1}-t} \\
&cell\ 2: \quad B_2 \leftarrow A_2 \oplus B_2\alpha^{2^{m-1}-t+1} \\
&cell\ 3: \quad B_3 \leftarrow A_3 \oplus B_3\alpha^{2^{m-1}-t+2} \\
&\quad . \\
&\quad . \\
&\quad . \\
&cell\ 2t: \quad B_{2t} \leftarrow A_{2t} \oplus B_{2t}\alpha^{2^{m-1}+t-1} \\
&\quad \text{or} \\
&cell\ k: \quad B_k \leftarrow A_k \oplus B_k\alpha^j
\end{aligned}
\tag{4.5}
$$

for

$$2^{m-1} - t \leq j \leq 2^{m-1} + t - 1; \quad 1 \leq k \leq 2t$$

where $\leftarrow$ implies the operation "is replaced by".

Based on the values of t and m, the corresponding α^j variables are stored in the cell registers during chip initialization. After the complete $R(x)$ has entered, the required syndromes S_j are contained in the registers B_j. After n clock cycles, the $2t$ syndromes are shifted out in parallel and fed into the Berlekamp-Massey module. A maximum of 32 syndrome cells are supported by the device.

The symbolic structure of the syndrome module is shown in Figure 4.6. It consists of 22,515 gates. The size and description of each I/O signal are given in Table 4.4.

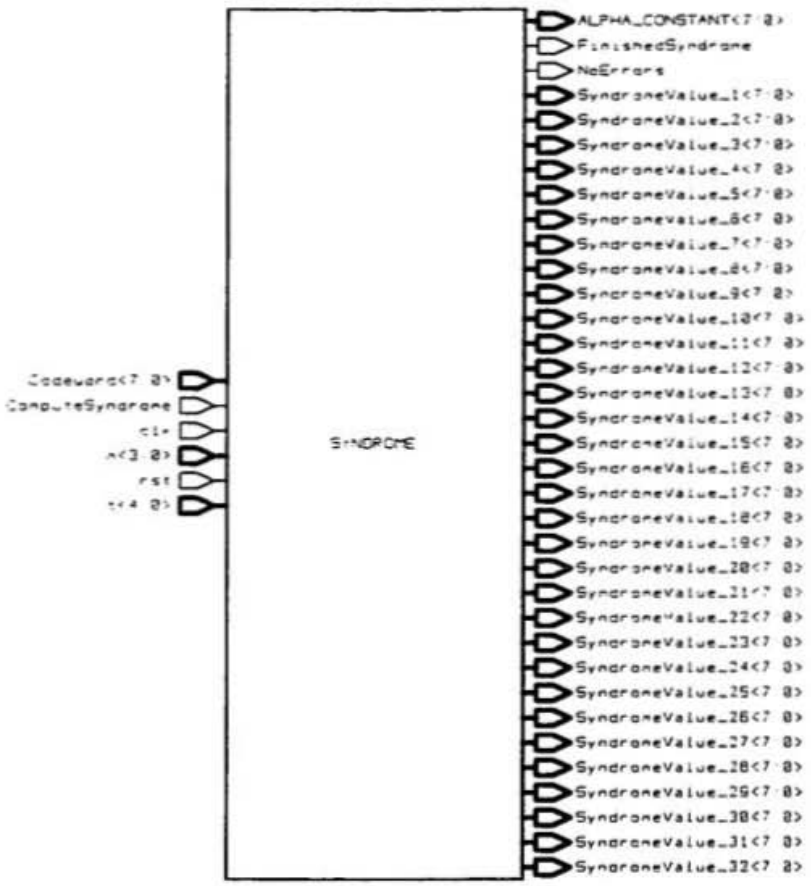

Figure 4.6: Symbolic diagram of the Syndrome Module

Signal	Bit Size	Description
Codeword	8	input port for the received word *from channel*
ComputeSyndrome	1	starts the decoding process
clk	1	clock signal
m	4	Galois field symbol size
rst	1	reset signal
t	5	error correction capability
ALPHA_CONSTANT	8	offset term required in the Forney algorithm
FinishedSyndrome	1	goes high after the syndromes have been calculated
NoErrors	1	goes high when all syndromes are zero
SyndromeValue_XX	256	32 syndrome values

Table 4.4: Syndrome I/O Pins

4.3.3 Berlekamp

The Berlekamp-Massey Module implements the following algorithm[53][54][57]. As shown below, minor modifications to the original Berlekamp-Massey algorithm are necessary to facilitate RTL logic synthesis. $\sigma(X)$, $\widehat{\sigma}(X)$ and $\beta(X)$ are 16-byte registers. L and γ can have maximum integer values of 16 and 32 respectively since $t_{max} = 16$.

The Massey-Berlekamp Algorithm

Step 1: If Reset = 1 then *Initialize all the flip-flops and registers*

$\gamma = 0$; $\sigma(X) = 0$; $L = 0$, $\beta(X) = 1$; $\widehat{\sigma}(X) = 0$; $\Delta = 0$;

else

Step 2: *For γ = 1 to 2*t clock cycles*

--compute the error discrepancy Δ and Δ^{-1}

$\Delta = \sum_{i=0}^{L} \sigma_i S_{\gamma-i}$; $\Delta^{-1} = \frac{1}{\Delta}$;

```
        if Δ > 0 then
```

$\hat{\sigma}(X) = \sigma(X) - \Delta X\beta(X)$;

```
                if 2L < γ then
```

$\beta(X) = \Delta^{-1}\sigma(X)$;

$\sigma(X) = \hat{\sigma}(X)$;

$L = \gamma - L$;

```
                else
```

$\beta(X) = X\beta(X)$;

$\sigma(X) = \hat{\sigma}(X)$;

```
                end if;
        else
```

$\beta(X) = X\beta(X)$;

```
        end if;
end for;
```

After the error locator polynomial $\sigma(X)$ has been determined, the error evaluator polynomial $\Omega(X)$ is found using the relationship $\Omega(X) = S(x)\sigma(X) \bmod x^{2t}$. The 16 coefficients of $\Omega(X)$ are determined in parallel after γ = 2*t cycles as shown below :

```
Step 3: Error Evaluation Polynomial
If γ = 2*t cycles then --Compute the 16 error evaluator polynomial
                       -- coefficients in parallel
```

$\Omega_1 = S_1$ **xor** σ_1;

$\Omega_2 = S_2$ **xor** $\sigma_1 S_2$ **xor** σ_2;

.

.

.

Ω_v = S_v xor $\sigma_1 S_{v-1}$ xor $\sigma_2 S_{v-2}$ xor ... xor σ_v;

.

.

.

Ω_{16} = S_{16} xor $\sigma_1 S_{15}$ xor $\sigma_2 S_{14}$ xor ... xor σ_{16};

end if;

The symbolic structure of the Berlekamp Module is shown in Figure 4.7. Its major components include 170 multipliers, an exponentiator/inverter, a 32-byte register, a 16-to-1 multiplexer, a 32 integer counter, a 16-byte shift register, 4 16-byte registers. It requires $2t+1$ clock cycles to determine $\sigma(X)$ and $\Omega(X)$. The total gate count is 107,015. If gate count had been a design issue, an aggressive design could drastically reduce the number of multipliers by almost one-third but at the expense of speed. All the 16 coefficients of $\Omega(X)$ could be determined sequentially using the last expression $\Omega_{16} = S_{16}$ xor $\sigma_1 S_{15}$ xor $\sigma_2 S_{14}$ xor ... xor σ_{16}. In this case, a minimum of $2t + 17$ clock cycles would be needed to compute $\sigma(X)$ and $\Omega(X)$.

The size and description of each I/O signal are given in Table 4.5.

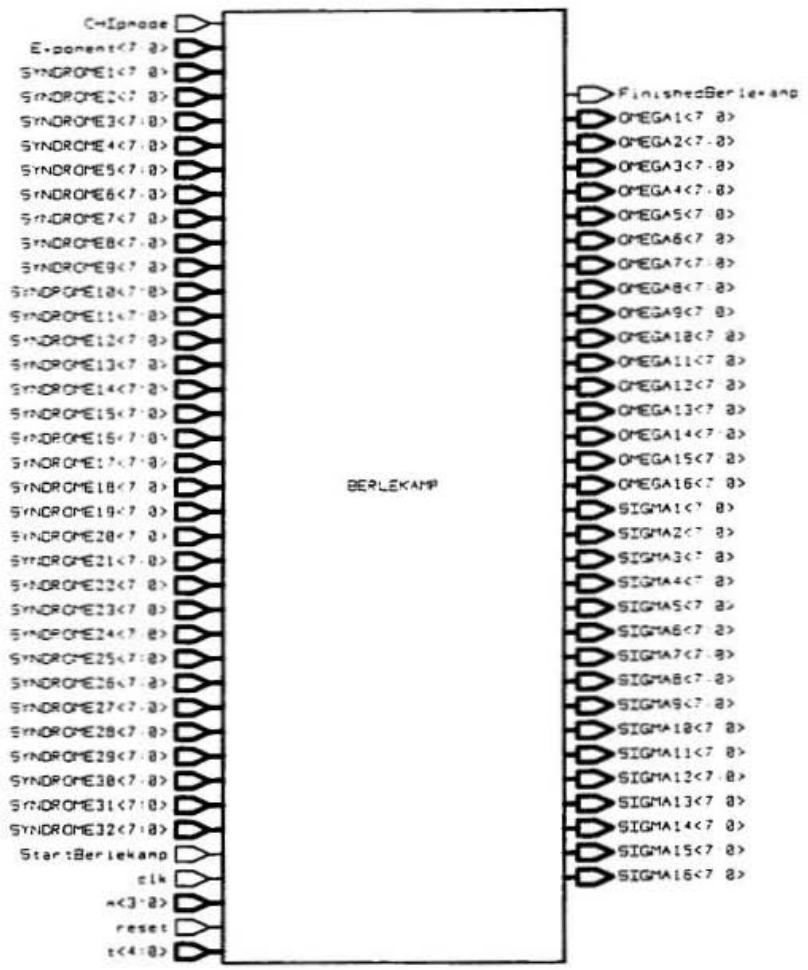

Figure 4.7: Symbolic diagram of the Berlekamp Module

Signal	Bit Size	Description
CHIPmode	1	configures the exponentiator/inverter to operate as an inverter
Exponent	8	set to $FE_{hexadecimal}$ for inversion
Syndrome_XX	256	32 syndrome values
StartBerlekamp	1	starts the evaluation of σ and Ω
clk	1	clock signal
m	4	Galois field symbol size
rst	1	reset signal
t	5	error correction capability
FinishedBerlekamp	1	goes high after σ and Ω have been found
OMEGA_XX	128	16 Ω coefficients
SIGMA_XX	128	16 σ coefficients

Table 4.5: Berlekamp Module I/O Pins

4.3.4 Error Magnitude Evaluation

4.3.4.1 The Chien Search

The Chien search evaluates the error locator polynomial

$$\Lambda(x) = \prod_{l=1}^{v}(1 - xX_l) = \alpha_v x^v + \alpha_{v-1}x^{v-1} + ... + \alpha_1 x + 1 \tag{4.6}$$

at all elements of the Galois field $GF(2^m)$, where $X_l = \alpha^{i_l}$.

The actual error locations are indicated by α^{i_1}, α^{i_2}, ..., α^{i_v}. Obviously $\Lambda(x)$ has roots α^{-i_1}, α^{-i_2}, ..., α^{-i_v}. Clearly an error occurs at position i if and only if $\Lambda(\alpha^{-i}) = 0$ or

$$\Lambda(\alpha^{-1}) = 1 + \sum_{l=1}^{v}\Lambda_l\alpha^{-i_l} = 0 \Rightarrow \sum_{l=1}^{v}\Lambda_l\alpha^{-i_l} = 1 \tag{4.7}$$

An implementation of Equation (4.7) is done using the Chien search circuit shown in Figure 4.8. It consists of an mt-input exclusive-or gate, t multipliers and t registers. In this design, a maximum of 16 multipliers and 16 registers can be

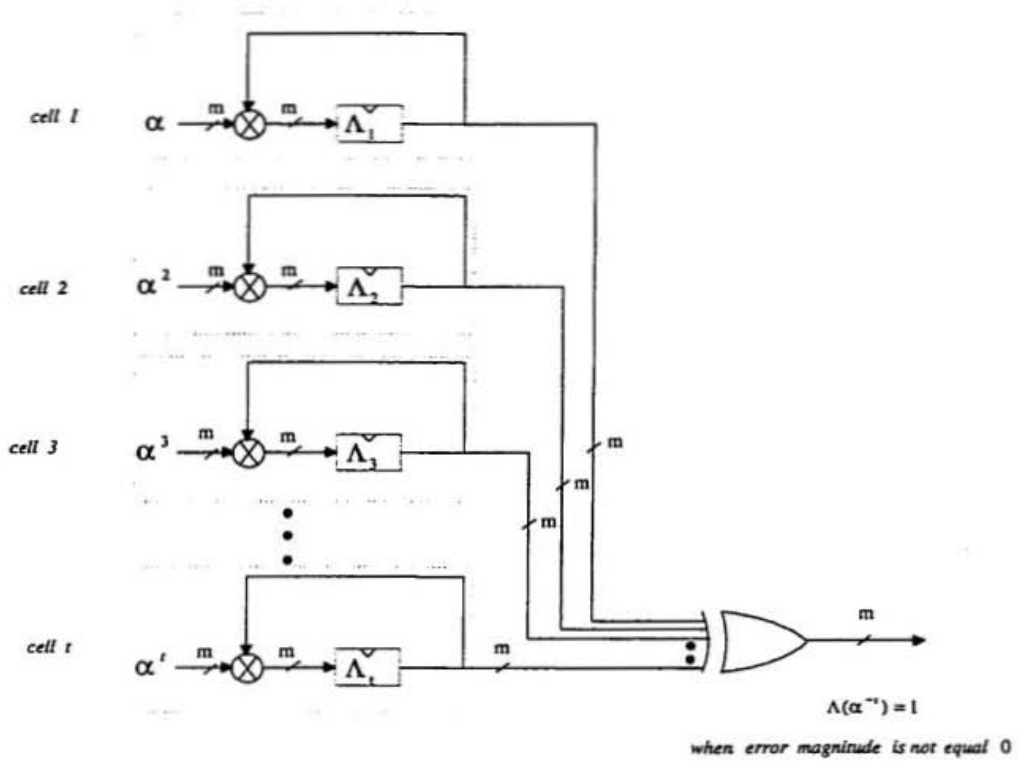

Figure 4.8: The Chien Search Hardware

used.

The Chien search operates as follows:

Each coefficient of $\Lambda(x)$ is repeatedly multiplied by α^i, where α is the primitive element in $GF(2^m)$. Each set of the products is then summed by the mt-input exclusive-or gate to obtain the $Output = \sum_{l=1}^{v} \Lambda_l \alpha^{-i_l}$

If α^i is a root of $\Lambda(x)$ then the $Output = 1$, and an error is indicated at the coordinate associated with $\alpha^{-i} = \alpha^{n-i}$. Otherwise, if the $Output = 0$, there is no error.

4.3.4.2 The Forney Algorithm

If α^{-i_m} is a zero of $\Lambda(x)$, then the error magnitude module computes the error value at location R_{n-i_m} using [54]

$$Y_{i_m} = -\frac{\Omega(\alpha^{-i_m})}{\Lambda'(\alpha^{-i_m})}\alpha^{i_m j} \tag{4.8}$$

where $j = 2 + t - 2^{m-1}$,

$\Lambda'(x)$ is the first derivative of $\Lambda(x)$ with respect to x and $\alpha^{i_m j}$ is the offset term.

The calculations indicated by Equation (4.8) are almost identical to those required in the Chien search, and the realization shown in Figure 4.8 can be modified to evaluate each of the polynomials $\Omega(\alpha^{-i_m})$ and $\Lambda'(\alpha^{-i_m})$. The structure of the circuit which evaluates the error evaluation polynomial $\Omega(x)$ at $x = \alpha^{-1}$ is shown in Figure 4.9. It consists of an mt-input exclusive-or gate, t m-bit registers and t multipliers. Unlike the Chien search circuit, this circuit does not have a one-detector at its output [53].

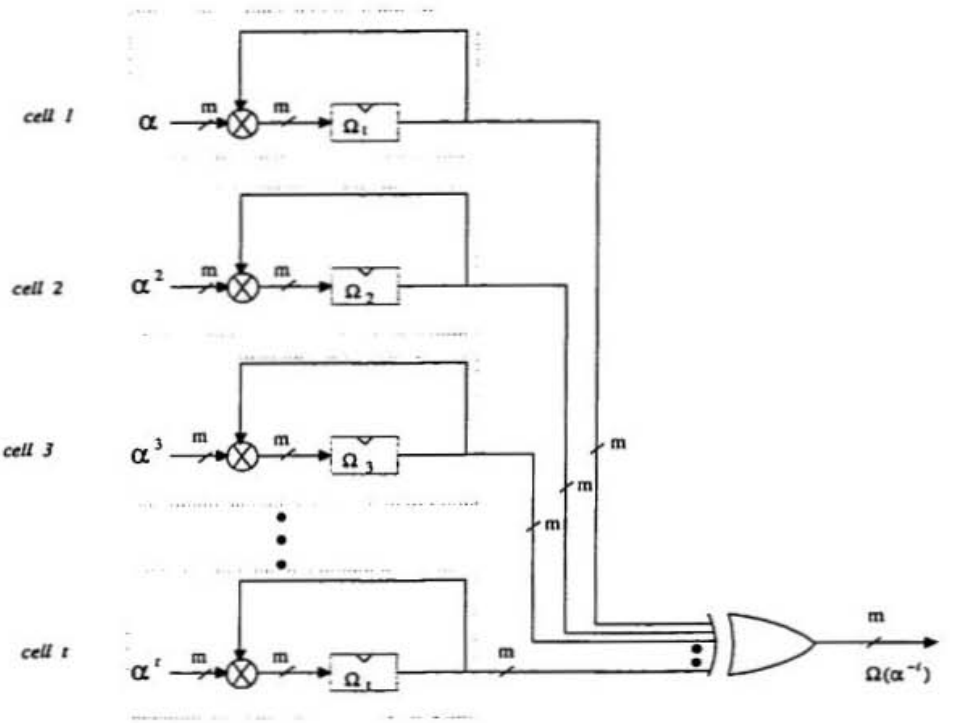

Figure 4.9: Error Evaluation Polynomial Circuit

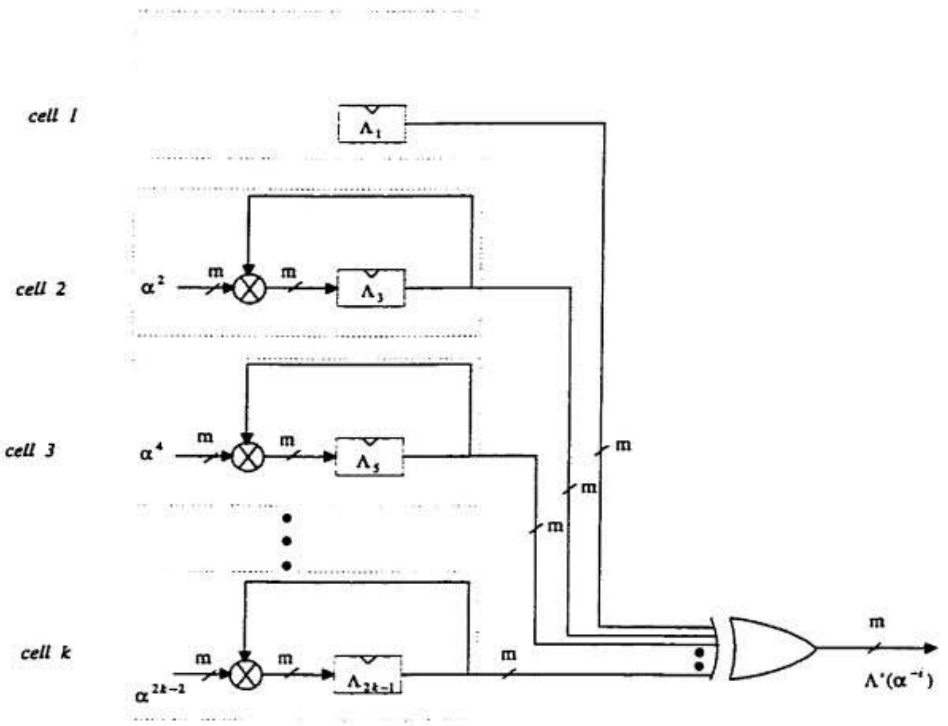

Figure 4.10: Derivative Circuit

The first derivative of $\Lambda(x)$, i.e., $\Lambda'(x)$ required in the Forney algorithm is given by

$$\Lambda'(x) = \Lambda_1 + \Lambda_3 x^2 + \Lambda_5 x^4 + \Lambda_7 x^6 + ... = \sum_{k=1}^{\lfloor \frac{t+1}{2} \rfloor} \Lambda_{2k-1} x^{2k-2} \tag{4.9}$$

where k is an integer.

An evaluation of $\Lambda'(\alpha^{-i})$ suggests the implementation shown in Figure 4.10.

The derivative circuit consists of an $m(2k-2)$ input exclusive-or gates, $2k-2$ multipliers, and $2k-2$ registers, where k is an integer. The offset term $\alpha^{i_m j}$ and the inverse of $\Lambda'(\alpha^{-i_m})$ as specified in the Forney algorithm are evaluated using the proposed exponentiator/inverter configured to operate as an exponentiator and

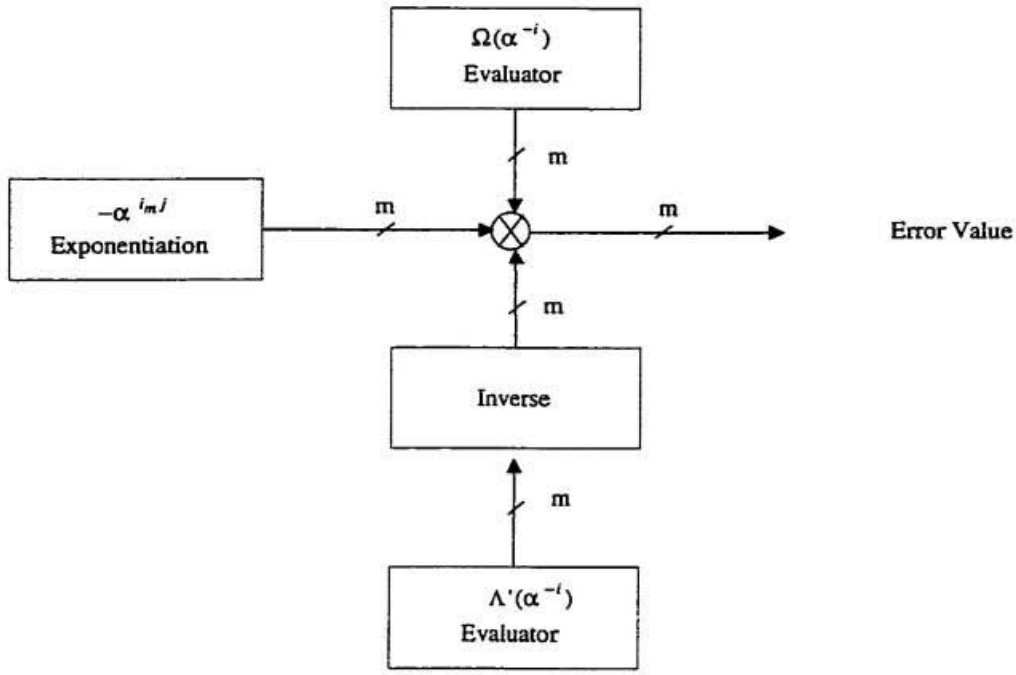

Figure 4.11: Block Diagram of the Error Magnitude Evaluation

inverter respectively. The block diagram of the Forney algorithm is shown in Figure 4.11.

As shown in Figures 4.8, 4.9 and 4.10 the error location polynomial $\Lambda(x)$, error evaluation polynomial $\Omega(x)$ and the first derivative of $\Lambda(x)$ are evaluated for the same field elements. The values of α^j, which are based on user defined t and m, are stored in the cell registers during chip initialization. All the polynomials $\Lambda(x)$, $\Lambda'(x)$ and $\Omega(x)$ are evaluated in parallel followed by one inversion, one exponentiation and two multiplications to obtain the error values. This process takes $n + m + 2$ clock cycles. The symbolic structure of their module is shown in Figure 4.12. It consists of 37,952 gates. The bit size and description of each I/O signal are given

Signal	Bit Size	Description
ALPHA_CONSTANT	8	offset term in Forney algorithm
Exponent	8	exponent
Mode_EXPONENT	1	configures the exponentiator/inverter to operate as an exponentiator
Mode_INVERSE	1	configures the exponentiator/inverter to operate as an inverter
OMEGA_XX	128	16 Ω coefficients
SIGMA_XX	128	16 σ coefficients
StartCalculateValue	1	starts the evaluation of error values
clk	1	clock signal
m	4	Galois field symbol size
reset	1	reset signal
t	5	error correction capability
Cycles	8	clock cycles for error value evaluation
ERROR_VALUE	8	error value/magnitude
ErrorPosition	8	error position
FinishedCalculateValue	1	goes high after the errors are found

Table 4.6: I/O Pins of the Error Magnitude Evaluation

in Table 4.6.

4.3.5 Error Correction and Verification

The error correction module consists of eight 2-input XOR gates and eight multiplexers which are equivalent to 45 gates. It performs the Galois field addition operation $C(x) = R(x) + E(x)$ which exclusive or's the error values $E(x)$ with the buffered messages $R(x)$ in order to correct the errors.

The error verification module computes the syndrome values of the corrected symbols to check if they are all zero as the processed data leave the chip through the *Data_Correct* output port. If the syndrome values are not all zero, an error flag is generated by the *NoErrors* signal indicating that the data contains an uncorrectable number of errors.

Figure 4.12: Symbolic diagram of the Error Magnitude Evaluation

Signal	Bit Size	Description
CLK	1	clock signal
DATA_IN	8	input port for the received word
RD	1	enables reading the symbols from the stack
RESET	1	reset signal
WR	1	enables writing the symbols to the stack
DATA_OUT	8	output port for the buffered symbols

Table 4.7: I/O Pins of the FIFO Module

4.3.6 First-In-First-Out Buffer

The first-in-first-out (FIFO) buffer is a 255-byte register stack which is used to temporarily store the received code polynomial as the decoder determines the location and magnitude of the erroneous symbols.

When a write (WR) request is generated by the finite state machine, the symbols are pushed onto the stack if it is not full. The FIFO stack does not write to a full stack, hence this condition is monitored.

When a read (RD) request is generated by the FSM, the symbols are read from the "bottom" of the stack at depth $2^m - 1$ where $m = 3, 4, 5, 6, 7, 8$. If the stack is empty, then no symbol is read.

The symbolic architecture of the FIFO is shown in Figure 4.13. It consists of 18,127 gates. The size and description of each I/O signal are given in Table 4.7.

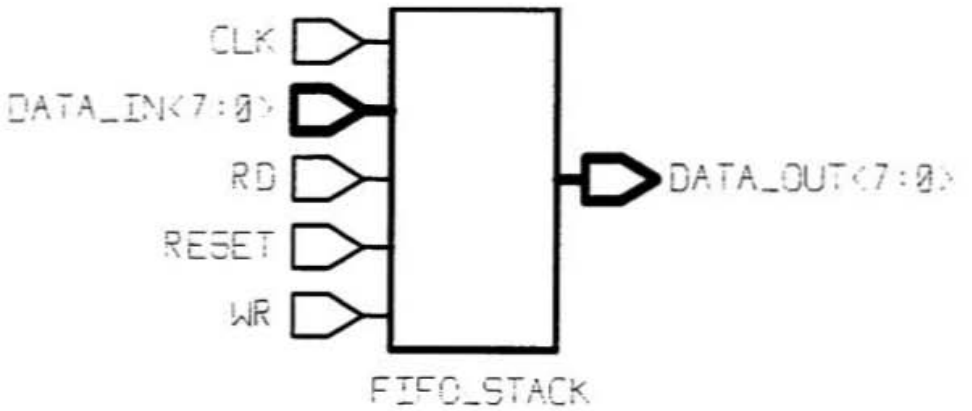

Figure 4.13: Symbolic diagram of FIFO

4.3.7 Finite State Machine

As shown in Figure 4.14, the finite state machine (FSM) for the RS encoder/decoder has ten (10) states which can change on the rising edge of the clock. These states are detailed below;

1. *Resetting:* In this state all the counters, flip-flops and registers in the RS modules are initialized to all zeros.

2. *SetMode:* In this state the chip can be configured to operate as an encoder or decoder.

3. *Encoding_State1:* In this state, the encoder calculates the codewords based on the error correction capability, symbol size and message symbols. The parity check symbols are also shifted out of the encoder.

4. *Encoding_State2:* The FSM module monitors whether the encoding process is complete before it advances to the *Resetting* state.

5. *Syndrome_State1:* In this state, the Syndrome module calculates the syndrome values based on the error correction capability, symbol size and received word. At the same time, the received word symbols are stored in the FIFO.

6. *Syndrome_State2:* The FSM module monitors if the syndrome evaluation is complete before it advances to the *Berlekamp_State1* state.

7. *Berlekamp_State1:* In this state, the Berlekamp module calculates the error location and error evaluation polynomials based on the error correction capability, symbol size and syndrome values.

8. *Berlekamp_State2:* The FSM module monitors if the error location and error evaluation polynomials have been calculated before it advances to the *ErrorValueCorrect_State1* state.

9. *ErrorValueCorrect_State1:* In this state, the Error Magnitude module finds the error locations and error values/magnitudes and corrects erroneous symbols. An uncorrectable error condition is also tested and reported.

10. *ErrorValueCorrect_State2:* The FSM module monitors if the error correction has been completed before going back to the *Resetting* state.

The finite state machine is a Moore machine, i.e., a sequential state machine whose outputs depend only on the current state, independent of the inputs. In other words, the functionality can be expressed as:

Next State (N) = *function* [current state (P), Input (I)]
Outputs (O) = *function* [current state (P)]

The FSM has been completely described by a single VHDL *process* with a synchronous reset signal. It uses a one-hot encoding style which requires the use of one positive edge triggered flip-flop per state, the current state being determined by the flip-flop that is on.

The symbolic architecture of the FSM is shown in Figure 4.15. It consists of 313 gates. The size and description of each I/O signal are given in Table 4.8.

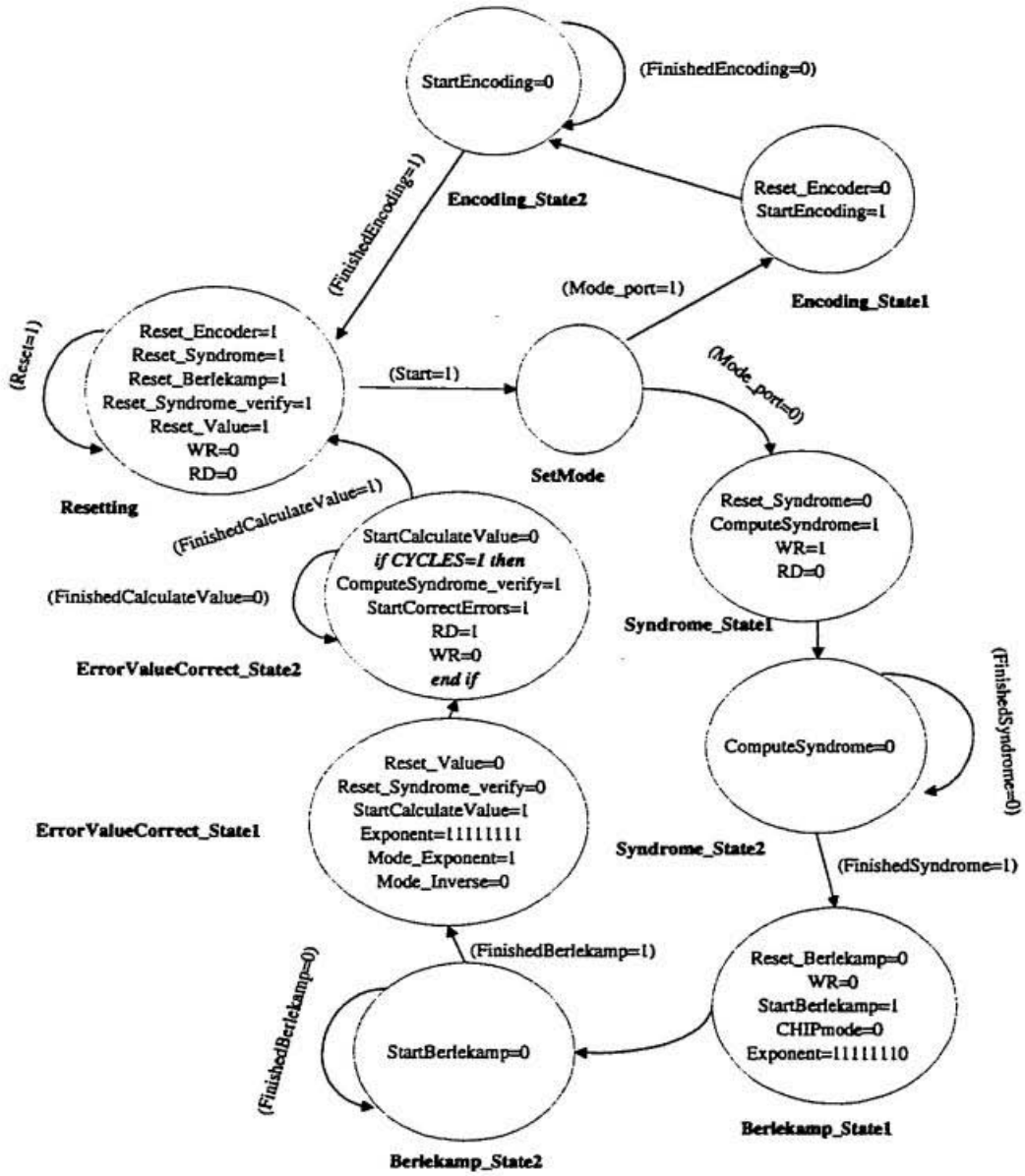

Figure 4.14: RS Encoder/Decoder Finite State Machine

Signal	Bit Size	Description
CYCLES	8	the clock cycles for error magnitude evaluation
Clock	1	clock signal
FinishedBerlekamp	1	notifies the FSM after σ, Ω are found
FinishedCalculateValue	1	goes high after the errors are found
FinishedEncoding	1	notifies the FSM when encoding is done
FinishedSyndrome	1	signals the FSM after syndromes are found
Mode_port	1	selects the encoding or decoding mode
Reset	1	resets the RS encoder/decoder
Start	1	starts the encoding or decoding process
m	4	Galois field symbol size
t	5	error correction capability
CHIPmode	1	configures the exponentiator/inverter to operate as an inverter
ComputeSyndrome	1	starts the decoding process
ComputeSyndrome_verify	1	tests the uncorrectable error condition
Exponent	8	set to $FE_{hexadecimal}$ during inversion
Mode_EXPONENT	1	configures the exponentiator/inverter to operate as an exponentiator
Mode_INVERSE	1	configures the exponentiator/inverter to operate as an inverter
RD	1	enables reading symbols from the stack
Reset_Berlekamp	1	resets the Berlekamp module
Reset_Encoder	1	resets the encoder
Reset_Syndrome	1	resets the Syndrome module
Reset_Syndrome_verify	1	resets the syndrome module which determines uncorrectable errors
Reset_Value	1	resets the Error Magnitude module
StartBerlekamp	1	starts the evaluation of σ and Ω
StartCalculateValue	1	starts the evaluation of error values
StartCorrectErrors	1	starts the error correction
StartEncoding	1	starts the encoding process
WR	1	enables writing symbols to the stack
m_out	4	Galois field symbol size *to modules*
t_out	5	error correction capability *to modules*

Table 4.8: I/O Pins of the FSM Module

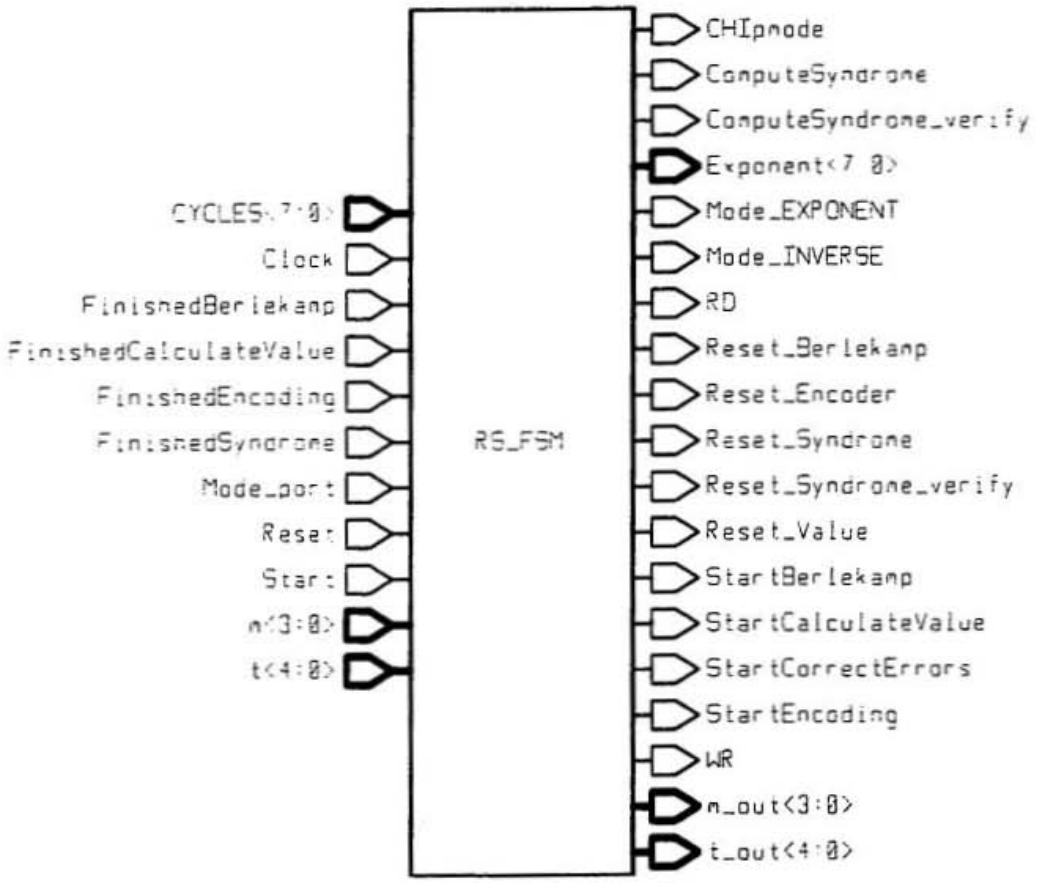

Figure 4.15: Symbolic diagram of the Finite State Machine

4.4 Testing and Results

As previously mentioned, once the behavioral model of the RS encoder and decoder has been captured using the VHDL hardware description language, each block is then partitioned into smaller modules which are modelled separately using a subset of the VHDL constructs suitable for logic synthesis. The size of each synthesizable module varies from 45 to a maximum of 20,000 gates. Larger modules are characterized by sequential processes which have heavy dataflow dependencies. The functional correctness of each VHDL RTL model has been verified using an interactive UNIX based RS encoder/decoder simulator written in C [72][73].

This section presents partial sample simulations showing the encoding and decoding stages of the ASIC at a frequency of 50 MHz, which is equivalent to a clock period of 20 *nanoseconds*. Test cases where error-correction succeeds and where it fails are considered for selected values of $m = 8$ and $t = 3$. The user can reconfigure the ASIC on-the-fly using any combination of $m = 3, 4, 5, 6, 7, 8$ and $t = 1, 2, 3, ..., 16$ depending on the application. To simplify the examples, it is assumed that a message of $k = n - 2t = 255 - 6 = 249$ zero symbols is input to the ASIC for encoding. Each symbol is $m = 8$ bits so that the generated codeword contains $n = 2^m - 1 = 255$ symbols.

In the first instance, it is assumed that an error value of 00000010_{binary} occurs at position 1 during the transmission of the codeword via the communication channel. In the second scenario, it is assumed that 4 errors occur in the received data at positions 1, 2, 3 and 4. In the timing diagrams, the time scale is in *nanoseconds* and all the signal values are represented as hexadecimal numbers.

Figure 4.16 illustrates the states of the input/output signals of the RS encoder

module described in section 4.3.1, during the encoding stage of the ASIC. As indicated by the *FinishedEncoding* signal, the encoding process takes 255 clock cycles, for the chosen variables $m = 8$ and $t = 3$.

Figure 4.17 illustrates the input/output signals of the syndrome module described in section 4.3.2, during the first the step of the decoding stage. As indicated by the *FinishedSyndrome* signal, syndrome evaluation takes 255 clock cycles. The meaningful 6 syndromes are indicated, all other syndrome values are zero. These can easily be verified using the equations described in section 4.3.2.

Figure 4.18 illustrates the simulation cycles of the overall RS ASIC during encoding. As shown in the figure, the encoding process takes 255 clock cycles for the chosen parameters. The relevant signals are indicated.

Figure 4.19 illustrates the simulation cycles of the overall RS ASIC during the decoding process. As shown in the figure, the decoding process takes 553 clock cycles for the chosen parameters. The relevant signals are indicated. The error position is correctly denoted at position 1 of the received data.

Figure 4.20 illustrates the simulation cycles of the overall RS ASIC during the decoding process, where error correction fails. Since there are 4 errors in the received data at positions 1, 2, 3 and 4, the decoder fails to correct the errors because the actual number of errors $v = 4$ is greater than the selected $t = 3$. As shown in the figure, the error positions and error values cannot be determined by the ASIC.

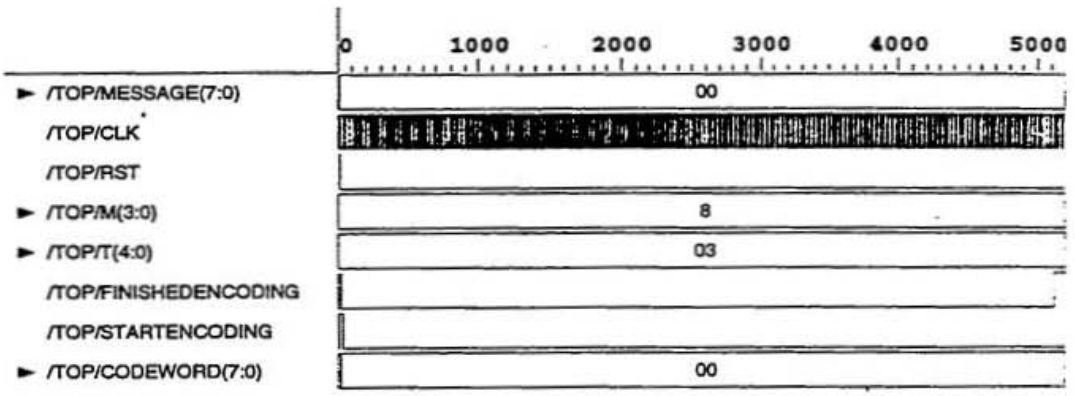

Figure 4.16: Gate Level Simulations of the RS Encoder

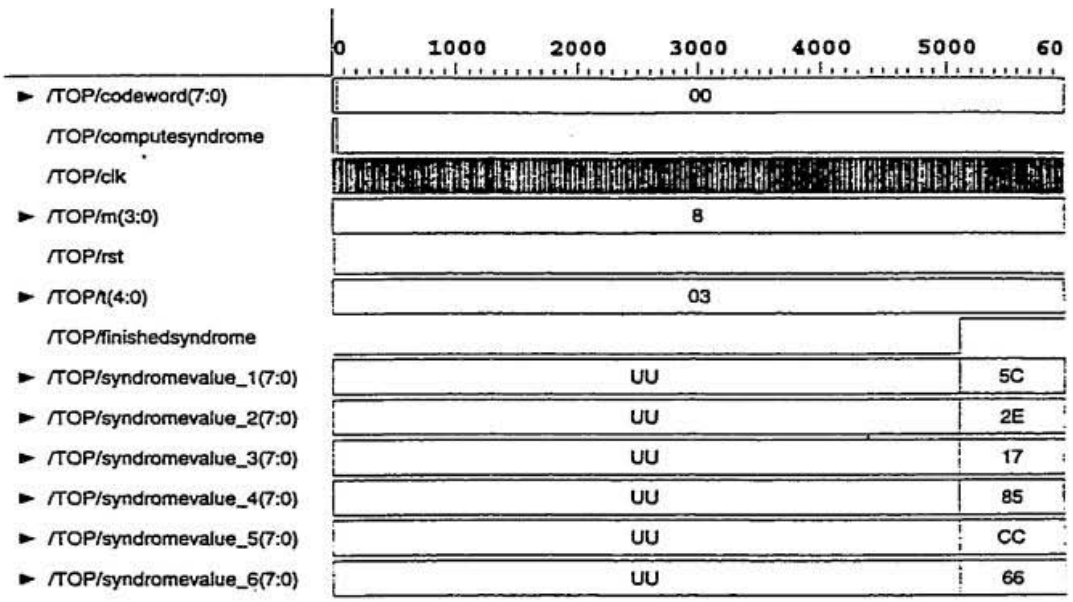

Figure 4.17: Gate Level Simulations of the Syndrome Module

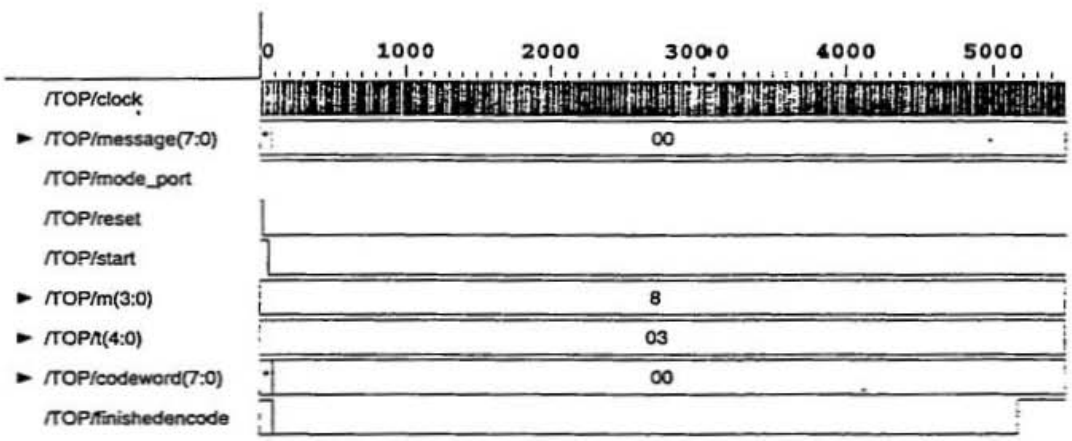

Figure 4.18: Gate Level Simulations of the overall RS ASIC (Encoding)

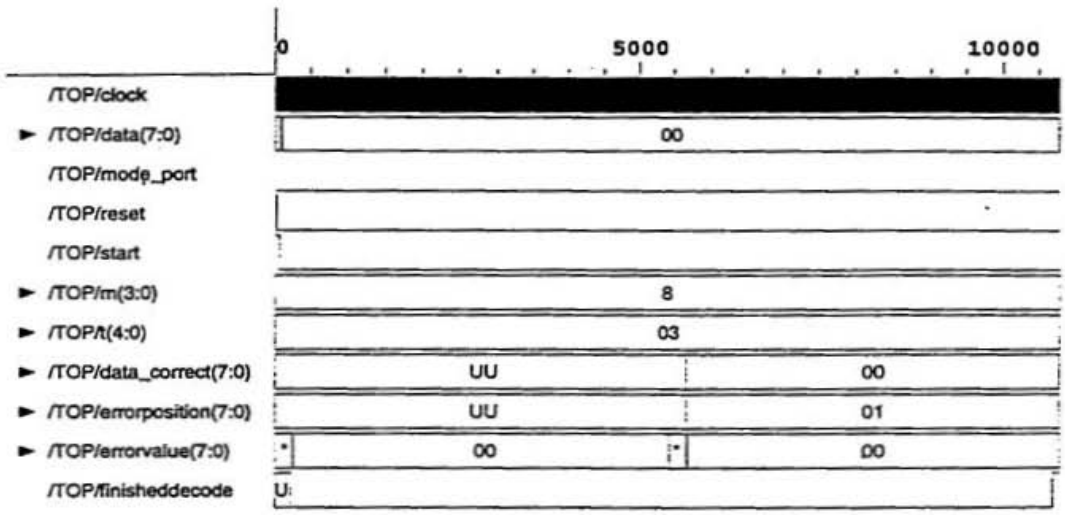

Figure 4.19: Gate Level Simulations of the overall RS ASIC (Decoding)

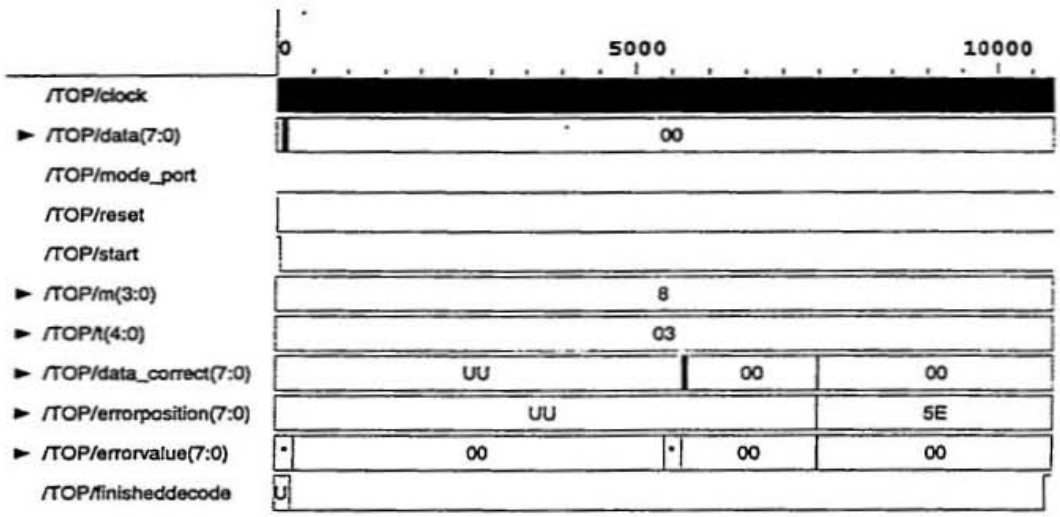

Figure 4.20: Gate Level Simulations of the RS ASIC (Decoding Failure)

4.5 Discussion and Summary

By using the proposed arithmetic circuits and a multiplexing technique which selects the different values of m and t for various RS codes, a new programmable RS encoder/decoder is designed and implemented. Values of the Galois field symbol size $m = 3, 4, 5, 6, 7, 8$ and error correction capability $t = 1, 2, 3, ..., 16$ are supported in the illustration. The chip contains 218,206 gates, where a gate is equivalent to a 2-input NAND gate. An inverse ROM is completely eliminated for performing Galois field element inversion as suggested in the various decoder implementations presented in the literature [2][36][38][39][45][47][49][50][51]. They have been customized for a specific m and t. The reason for avoiding a ROM is that six different ROMs would have been required because the inverse elements in Galois fields are different for variable symbol size m. The same argument applies for the exponentiation operation. Therefore, a total of twelve different ROMs would have been required for the inverse and exponentiation operations. The constant multipliers have also been replaced with the general m-programmable multipliers throughout the encoder, syndrome, Chien search and error value evaluation circuits.

The design is parameterised directly in VHDL in terms of the symbol size m and the error correction capability t. The syndrome values are calculated in n clock cycles, the error locator and error evaluator polynomials in $2t+1$ clock cycles, and the error value calculations and error corrections in $n+m+2$ clock cycles. The overall clock cycles for the encoder and the decoder are n and $2n+2t+m+3$ respectively. Thus, the encoder can generate codewords at a sustained rate of $\frac{m}{\tau} \times 10^3$ Mbits/sec whereas the decoder can process incoming data at a maximum rate of $\frac{nm}{(2n+2t+m+3)(\tau)} \times 10^3$ Mbits/sec, where τ is the clock period in *nanoseconds*, and

m	t_{max}	Encoder (Mbits/sec)	Decoder (Mbits/sec)
3	3	150	40
4	7	200	59
5	15	250	78
6	16	300	113
7	16	350	150
8	16	400	184

Table 4.9: RS Encoder/Decoder Data Rates

$n = 2^m - 1$ is the block length. The VHDL gate-level simulations were performed at 50 MHz. The estimated data rates when $t = t_{max}$ and $\tau = 20$ ns are shown in Table 4.9.

Clearly, higher data rates can be expected at higher frequencies or by using the more aggressive technologies such as the 0.35-μm CMOS. It also appears that the decoder datapath could be constructed with three linear pipeline stages in order to further increase the decoding throughput rate [75][76]. Pipeline registers would be required between the syndrome, Berlekamp and error magnitude modules. Thus, using the 0.8-μm CMOS standard cells, the pipelined version of the chip should be able to process data at three times the estimated rates in Table 4.9.

The gate counts for the various modules are shown in Table 4.10.

MODULE	GATE COUNT
Encoder	9,644
Syndrome	22,515
Berlekamp	107,015
Error Magnitude Evaluation	37,952
Error Correction	45
Error Verification	22,515
First-In-First-Out Buffer	18,127
Finite State Machine	254
Glue Components	138
TOTAL NUMBER OF GATES	218,206

Table 4.10: RS Encoder/Decoder Modules and Equivalent Gate Count

Chapter 5

Conclusion and Future Work

Forward error correction (FEC) is a common technique used to improve the reliability and efficiency of communication channels. The RS codes are widely used in modern day digital communications systems to correct erasures, random and burst errors during data transmission. As a contribution to the field, this thesis introduced

(1) new parameterizable Galois field arithmetic VLSI structures.

(2) an algebraic encoder/decoder ASIC which implements a wide family of RS codes. The design is parameterized in terms of the RS code variables m, n and t. Hence, it can be configured to operate in various communication channels which require different RS codes.

5.1 Galois Field Arithmetic Architectures

An overview of Galois field arithmetic operations and their corresponding VLSI implementations was presented in Chapter 1. Only the most complex operations namely exponentiation, inversion and multiplication were considered. Chapter 3 introduced new m-programmable arithmetic structures which exploited the sym-

metric properties of available architectures. These could be configured for the symbol size $m = 3, 4, 5, 6, 7$ and 8. The standard representation for the elements was used. It appeared that little work had been done in the literature to develop such structures. For this purpose, exponentiation, inversion and multiplication circuits were investigated in detail. It was also demonstrated that inversion was a form of exponentiation in Galois fields. An m-programmable array which evaluated both operations was designed and simulated. It had a low design complexity, low latency, high throughput rate and a very high fault coverage compared to other structures. The proposed exponentiator/inverter outperformed the inverters presented in [22][23] when it was configured to compute field element inversion. All the proposed architectures were implemented in standard cells using a VHDL based design entry. Thus, they could be used in applications that required a variable symbol size m.

5.2 VLSI Reed-Solomon Encoder/Decoder

The different RS decoding algorithms were described in Chapter 2. A survey of the existing encoder and decoder structures was also presented. A multiplexing technique and the proposed arithmetic circuits were used throughout the design and implementation of the new programmable RS encoder/decoder in CMOS standard cells. The chip supported a wide family of RS codes whose symbol size m and error correction capability t could be parameterized to meet different user requirements. Unlike the decoders customized for a fixed m and t as presented in the literature [2][36][38][39][45][47][49][50][51], it was found to be flexible since the symbol size m, block length n and error correction capability t were all variable.

Constant multipliers and inverse ROMs were also completely avoided to allow ease of reconfigurability. In the thesis, example values of the Galois field symbol size $m = 3, 4, 5, 6, 7, 8$ and error correction capability $t = 1, 2, 3, ..., 16$ were supported. The main advantage of such an ASIC is that its total design cost is amortized over a wide application base.

The algebraic encoding/decoding technique was used. The encoder used the self-reciprocal generator polynomial which structured the codewords in a systematic form. The first step in the decoding algorithm calculated the syndrome polynomial *S(x)*. The Berlekamp-Massey algorithm determined the error-locator polynomial. Its low design complexity made it suitable for VLSI synthesis. The error magnitude polynomial was calculated using the expression $\Omega(x) = S(x)\sigma(x) \bmod x^{2t}$. Once the location and magnitude of the errors had been determined using the Chien Search and the Forney algorithm respectively, the received messages were corrected and verified as they left the chip.

It was found that the overall clock cycles for the encoder and the decoder were n and $2n + 2t + m + 3$ respectively. Hence, the encoder could generate codewords at a sustained data rate of $\frac{m}{\tau} \times 10^3$ Mbits/sec whereas the decoder could process incoming data at a maximum data rate of $\frac{nm}{(2n+2t+m+3)(\tau)} \times 10^3$ Mbits/sec, where τ was the clock period in *nanoseconds*, and $n = 2^m - 1$ was the block length. All the VHDL gate-level simulations were performed at a frequency of 50 MHz. The equivalent gate count was 218,206 gates.

This thesis fully demonstrated that the parameters m, n and t can indeed be variable in RS encoder/decoder design by using the same hardware.

5.3 Future Work

As indicated in the thesis, emphasis was placed on the algebraic decoding technique alone. Other algorithms could be investigated to see if their designs could be parameterized in terms of m, n and t as well. Due to limitations in the design kit, it was not possible to investigate design issues such as power dissipation and backannotation. Backannotation would have allowed the original gate-level netlist to be annotated with extracted parasitics from the layout so that a more accurate VHDL simulation could be performed. These simulations would confirm the timing and help estimate power dissipation as well. One direction for future research is to investigate the effects of parasitics and power dissipation on increasing values of the RS code parameters m, n and t when advanced design kits are released by CMC. The only major changes required in the current ASIC are the increase of the sizes of the bus signals and redesign of the multiplier and exponentiator/inverter to accommodate larger values of $m > 8$. Such an exercise requires a small fraction of the effort and cost of the original design if a maximum of $m = 64$ was required, for instance.

It can be inferred from Chapter 5 that the design complexity increases with the block length, error correction capability and symbol size of the code. One could further investigate how the overall gate count varies with these parameters as a measure of design complexity. A relationship between the clock cycles and these design variables has already been found.

The sequential nature of the decoding algorithm suggests that the datapath may be constructed with three linear pipeline stages in order to further increase the decoding throughput rate [75][76]. A substantial portion of the decoder is always

m	t_{max}	Encoder (Gbits/sec)	Decoder (Gbits/sec)
3	3	1.80	0.48
4	7	2.40	0.71
5	15	3.00	0.94
6	16	3.60	1.36
7	16	4.20	1.80
8	16	4.80	2.21

Table 5.1: Projected RS Encoder/Decoder Data Rates

idling during the decoding process. Pipeline registers would be required between the main modules. Higher data rates in the Gbits/sec region could be expected if the pipeline version was implemented using the more aggressive technologies. The technology roadmap projects a 0.18-μm CMOS technology to be available in 1999 and 0.1-μm in 2001 [77]. It is projected that the encoder and decoder could have maximum throughput rates of $\frac{m}{\tau} \times 3$ Gbits/sec and $\frac{nm}{(2n+2t+m+3)(\tau)} \times 3$ Gbits/sec respectively. These are shown in Table 5.1 for $\tau = 5$ ns. To meet the specifications for a *k*current data rate* Gbits/sec channel, it also seems that k chips could be configured to operate in parallel, where $k = 1, 2, 3, ...$ is an integer. One could investigate the design issues and limitations involved. Work in this direction is also recommended.

References

[1] G.D. Forney, *Concatenated Codes*, Cambridge: M.I.T. Press, 1966.

[2] S. Wicker and V.K. Bhargava, *Reed-Solomon Codes and Their Applications*, New York: The Institute of Electrical and Electronics Engineers, 1994.

[3] S. Whitaker, J. Canaris, and K. Cameron, "Reed Solomon VLSI Codec for Advanced Television," *IEEE Trans. Circuits and Systems for Video Tech.*, vol. 1, no. 2, pp. 230-236, June 1991.

[4] S. Kwon and H. Shin, "An Area-Efficient VLSI Architecture of a Reed-Solomon Decoder/Encoder for Digital VCRs," *IEEE Trans. Consumer Electronics*, vol. 43, no. 4, pp. 1019-1027, Nov. 1997.

[5] Y.R. Shayan, T. Le-Ngoc, and V.K. Bhargava, "A Versatile Time-Domain Reed-Solomon Decoder," *IEEE J. Selected Areas in Comm.*, vol. 8, pp. 1,535-1,542, Oct. 1990.

[6] Y.R. Shayan and T. Le-Ngoc, "A Cellular Structure for a Versatile Reed-Solomon Decoder," *IEEE Trans. Comput.*, vol. 46, no. 1, pp. 80-85, Jan. 1997.

[7] W. Wolf, *Modern VLSI Design: A Systems Approach*, New Jersey: Prentice Hall, 1994.

[8] T.K. Truong, L.J. Deutsch, I.S. Reed, I.S. Hsu, K. Wang, and C.S. Yeh, "The VLSI Design of a Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm," *Third Caltech Conf. on VLSI*, pp. 303-329, 1983.

[9] I.S. Hsu, I.S. Reed, T.K. Truong, K. Wang, C.S. Yeh, and L.J. Deutsch, "The VLSI Implementation of a Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm," *IEEE Trans. Comput.*, vol. C-33, no. 10, pp. 906-911, Oct. 1984.

[10] C.C. Wang, T.K Truong, H.M. Shao, L.J. Deutsch, J.K. Omura, and I.S. Reed, "VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$," *IEEE Trans. Comput.*, vol. C-34, no. 8, pp. 709-716, Aug. 1985.

[11] C.L. Wang and J.L. Lin, "Systolic Array Implementation of Multipliers for Finite Fields $GF(2^m)$," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 796-800, July 1991.

[12] T.C. Bartee and D.I. Schneider, "Computation with Finite Fields," *Information and Computers*, vol. 6, pp. 79-88, March 1963.

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

[14] C.S. Yeh, I.S Reed, and T.K. Truong, "Systolic Multipliers for Finite Fields $GF(2^m)$," *IEEE Trans. Comput.*, vol. C-33, no. 4, pp. 357-360, Apr. 1984.

[15] P.A. Scott, S.E. Tavares, and L.E. Peppard, "A Fast VLSI Multiplier for $GF(2^m)$," *IEEE J. Select. Areas Commun.*, vol. SAC-4, no.1 pp. 62-66 Jan. 1986.

[16] M.A. Hasan and V.K. Bhargava, "Bit-Serial Systolic Divider and Multiplier for Finite Fields $GF(2^m)$," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 972-980, Aug. 1992.

[17] M.A. Hasan and V.K. Bhargava, "Division and Bit-Serial Multiplication over $GF(2^m)$," *IEE Proc.*, part E, vol. 139, no. 3, pp. 230-236, May 1992.

[18] M.A. Hasan, M.Z. Wang and V.K. Bhargava, "Modular Construction of Low Complexity Parallel Multipliers for a Class of Finite Fields $GF(2^m)$," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 962-971, Aug. 1992.

[19] M.A. Hasan, M.Z. Wang and V.K. Bhargava, "A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields," *IEEE Trans. Comput.*, vol. 42, no. 10, pp. 1278-1280, Oct. 1993.

[20] W. Wei, "A Systolic Power-Sum Circuit for $GF(2^m)$," *IEEE Trans. Comput.*, vol. 43, no. 2, pp. 226-229, Feb. 1994.

[21] S.T.J. Fenn, M. Benaissa, and D. Taylor, "Bit-serial Berlekamp-like Multipliers for $GF(2^m)$," *Electronics Letters*, vol. 31, no. 22, pp. 1893-1894, Oct. 1995.

[22] G.L. Feng, "A VLSI Architecture for Fast Inversion in $GF(2^m)$," *IEEE Trans. Comput.*, vol. 38, no. 10, pp. 1383-1386, Oct. 1989.

[23] C.L. Wang and J.L. Lin, "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields $GF(2^m)$," *IEEE Trans. Comput.*, vol. 42, no. 9, pp. 1141-1146, Sept. 1993.

[24] G.I. Davida, "Inverse of Elements of a Galois Field," *Electronics Letters*, vol. 8, pp. 518-520, Oct. 1972.

[25] S.T.J. Fenn, M. Benaissa, and D. Taylor, "Fast Normal Basis Inversion," *Electronics Letters*, vol. 32, no. 17, pp. 1566-1567, Aug. 1996.

[26] S.M. Yen, "Improved Normal Basis Inversion in $GF(2^m)$," *Electronics Letters*, vol. 33, no. 3, pp. 196-197, Jan. 1997.

[27] I.J. Calvo and M. Torres, "Complexity of the Inversion in $GF(2^m)$," *Electronics Letters*, vol. 33, no. 3, pp. 194-195, Jan. 1997.

[28] M.A. Hasan, "Division-and-Accumulation over $GF(2^m)$," *IEEE Trans. Comput.*, vol. 46, no. 6, pp. 705-708, June 1997.

[29] G. Seroussi, "A Systolic Reed-Solomon Encoder," *IEEE Trans. Inform. Theory*, vol. 37, no. 4, pp. 1217-1220,, July 1991.

[30] C.C. Wang and D. Pei, "A VLSI Design for Computing Exponentiations in $GF(2^m)$ and Its Application to Generate Pseudorandom Number Sequences," *IEEE Trans. Comput.*, vol. 39, no. 2, pp. 258-262, Feb. 1990.

[31] C.L. Wang, "Bit-Level Systolic Array for Fast Exponentiation in $GF(2^m)$," *IEEE Trans. Comput.*, vol. 43, no. 7, pp. 838-841, July 1994.

[32] P.A. Scott, S.J. Simmons, S.E. Tavares, and L.E. Peppard, "Architectures for Exponentiation in $GF(2^m)$," *IEEE J. Select. Areas Commun.*, vol. 6, no.3, pp. 578-586, Apr. 1988.

[33] B. Arazi, "Architectures for Exponentiation Over $GF(2^m)$," *IEEE Trans. Comput.*, vol. 42, no. 4, pp. 494-497, Apr. 1993.

[34] M. Kovač and N. Ranganathan, "ACE: A VLSI Chip for Galois Field $GF(2^m)$ Based Exponentiation," *IEEE Trans. Circuits Syst. II: Analog and Digital Signal Processing*, vol. 43, no.4, pp. 289-297, Apr. 1996.

[35] G. Maki, P. Owsley, K. Cameron, and J. Shovic, "A VLSI Reed Solomon Encoder: An Engineering Approach," *IEEE Trans. Custom Integrated Circuits Conf. Rec.*, pp. 177-181, May 1986.

[36] H.M. Shao and I.S. Reed, "On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays," *IEEE Trans. Comput.*, vol. 37, no. 10, pp. 1273-1280, Oct. 1988.

[37] H.M. Shao, T.K. Truong, I.S. Hsu, and L. J. Deutsch, "A Single Chip VLSI Reed-Solomon Decoder," *International Conf. Acoustic, Speech and Signal Processing*, pp. 2151-2154, 1986.

[38] P. Tong, "A 40-MHz Encoder/Decoder Chip Generated by a Reed-Solomon Code Compiler," *IEEE Trans. Custom Integrated Circuits Conf. Rec.*, pp. 13.5.1-13.5.4, May 1990.

[39] S. Whitaker, K. Cameron, G. Maki, J. Canaris, and P. Owsley, "VLSI Reed Solomon Processor for the Hubble Space Telescope," *VLSI Signal Processing IV*, IEEE Press, Chapter 35, 1991.

[40] K. Winters, P. Owsley, and G. Maki, "A VLSI Error Correction Decoder for Satellite Communication," *Proc. Int'l. Conf. on Systems Engineering*, pp. 37-44, Sept. 1984.

[41] G. Maki, P. Owsley, K. Cameron, and J. Venbrux, "VLSI Reed Solomon Decoder Design," *IEEE Military Communications Conf. Rec.*, pp. 46.5.1-46.5.6, Oct. 1986.

[42] F. Mendez, "VHDL and Cyclic Corrector Codes," *IEEE European Design Automation Conf. DAC*, pp. 526-531, 1994.

[43] S. Choomchuay and B. Arambepola, "Time Domain Algorithms and Architectures for Reed-Solomon Decoding," *IEE Proc.*, part I, vol. 40, no. 3, pp. 189-196, June 1993.

[44] T. Iwaki, T. Tanaka, T. Okuda, and T. Sasada, "Architecture of a High Speed Reed-Solomon Decoder," *IEEE Trans. Consumer Electronics*, vol. 40, no. 1, pp. 75-81, Feb. 1994.

[45] K. Cools, D. Devisch, K. Van Nieuwenhove, S. Vernalde, Bolsens, K. Chansik, O. Younguk and R. Lee, "ASIC Synthesis of a Flexible 80 Mbit/s Reed-Solomon Codec," *IEEE European Design Automation Conf. DAC*, pp. 658-663, 1994.

[46] M.A. Hasan and V.K. Bhargava, "Architecture for a Low Complexity Rate-Adaptive Reed-Solomon Encoder," *IEEE Trans. Comput.*, vol. 44, no. 7, pp. 938-942, July 1995.

[47] 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 Telecommuncations Conf.*, pp. 1455-1459, 1995.

[48] K. Iwamura, Y. Dohi, and H. Imai, "A Design of a Reed-Solomon Decoder with Systolic-Array Structure," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 118-122, Jan. 1995.

[49] J.M. Hsu and C.L. Wang, "An Area-Efficient Pipelined VLSI Architecture for Decoding of Reed-Solomon Codes Based on a Time-Domain Algorithm," *IEEE Trans. Circuits and Systems for Video Tech.*, vol. 7, no. 6, pp. 864-871, Dec. 1997.

[50] J.L. Politano and D. Deprey, "A 30 Mbits/s (255,223) Reed-Solomon Decoder," *EUROCODE Int'l Symp. on Coding Theory and Applications*, pp. 385-392, Nov. 1990.

[51] Ph. Sadot, "VLSI Implemetation of Error Correcting Codes," *Electrical Comm.*, vol. 65, no. 2, pp. 161-167, Jan. 1992.

[52] S.W. Wei and C.H. Wei, "High-Speed Decoder of Reed-Solomon Codes," *IEEE Trans. Commun.*, vol. 41, no. 11, pp. 1588-1593, Nov. 1993.

[53] G.C. Clark and J.B. Cain, *Error Control Coding For Digital Communications*, New York: Plenum, 1981.

[54] R.E Blahut, *Theory and Practice of Error Control Codes, Reading*, Mass: Addison Wesley, 1984.

[55] A. Michelson and A. Levesque, *Error Control Techniques for Digital Communication*, New York: Wiley, 1985.

[56] T.R Rao and E. Fujiwara, *Error-Control Coding For Computer Systems*, New Jersey: Prentice Hall, 1989.

[57] J. Anderson and S. Mohan, *Source and Channel Coding An Algorithmic Approach*, Boston: Kluwer Academic Publishers, 1991.

[58] G. Maki and P. Owsley, "Parallel Berlekamp vs. Conventional VLSI Architectures," *Government Microcircuit Applications Conf. Rec.*, pp. 5-9. Nov. 1986.

[59] S. Wicker, *Error Control Systems for Digital Communication and Storage*, New Jersey: Prentice Hall, 1995.

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

[61] Y. Sugiyama, S. Kasahara, and T. Namekawa, "A Method for Solving the Key Equation for Decoding Goppa Codes," *IEEE Trans. Contr.*, vol. 27, pp. 87-89, Jan. 1975.

[62] H.M. Shao, T.K. Truong, L.J. Deutsch, J. Yuen and I.S. Reed, "A VLSI Design of a Pipeline Reed-Solomon Decoder," *IEEE Trans. Comput.*, vol. C-34, no. 5, pp. 393-403, May 1985.

[63] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications*, New Jersey: Prentice Hall, 1983.

[64] R.E. Blahut, "A Universal Reed-Solomon Decoder," *IBM J. Research and Development*, vol. 28, no. 2, pp. 150-159, Mar. 1984.

[65] K.Y. Liu, "Architecture for VLSI Design of Reed-Solomon Encoders," *IEEE Trans. Comput.*, vol. C-31, no. 2, pp. 170-175, Feb. 1982.

[66] K.Y. Liu, "Architecture for VLSI Design of Reed-Solomon Decoders," *IEEE Trans. Comput.*, vol. C-33, no. 2, pp. 178-189, Feb. 1984.

[67] R.P. Brent and H.T. Kung, "Systolic VLSI Arrays for Polynomial GCD Computation," *IEEE Trans. Comput.*, vol. C-33, no. 8, pp. 731-736, Aug. 1984.

[68] H. Kung and M. Lam, "Fault tolerance and two-level Pipelining in VLSI Systolic Arrays," *Proc. MIT Conf. Advanced Res. VLSI*, pp. 74-83, Jan. 1984.

[69] J. McCanny, R. Evans, and J. McWhirter, "Use of Unidirectional Data Flow in bit-level Systolic Array Chips," *Electronics Letters*, vol. 22, pp. 540-541, May 1986.

[70] F. Wang, *Digital Circuit Testing: A Guide to DFT and Other Techniques*, San Diego: Academic Press, 1991.

[71] Synopsys: *Guidelines and Practices for Successful Logic Synthesis: Online Manual*, 1996.

[72] Z. Young, "A Reed-Solomon Code Simulator and Periodicity Algorithm," *M.Eng. thesis*, Memorial University of Newfoundland, St. John's, Newfoundland, 1994.

[73] Y. Ye, "A General Purpose Reed-Solomon CODEC Simulator and New Periodicity Algorithm," *M.Eng. thesis*, Memorial University of Newfoundland, St. John's,Newfoundland, 1995.

[74] Synopsys: *Design Compiler Family Reference Manual: Online Manual*, 1996.

[75] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, New York: McGraw-Hill, 1993.

[76] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, San Francisco: Morgan Kaufmann Publishers, 1996.

[77] L. Geppert, *IEEE Spectrum*, New York: The Institute of Electrical and Electronics Engineers, vol. 35, no. 1, pp. 23-28, Jan. 1998.