Logic Cell Array Designs for a (31,k) Reed-Solomon Codec

ACCEPTED
[FACU]LTY OF GRADUATE STUDIES
DEAN
4 FEB 93

by
Olaf Olgert Walter Dravnieks
B.Sc., University of Saskatchewan, 1983
B.Eng., University of Saskatchewan, 1984

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

Dr. Warren D. Little, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Vijay K. Bhargava, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Micaela Serra, Outside Member
(Department of Computer Science)

Dr. Fayez El Guibaly, Additional Member
(Department of Electrical and Computer Engineering)

Dr. Ruediger Vahldieck, Additional Member
(Department of Electrical and Computer Engineering)

Dr. Gholamali C. Shoja, External Examiner
(Department of Computer Science)

© Olaf Olgert Walter Dravnieks, 1992
University of Victoria

All rights reserved. Thesis may not be reproduced in whole or in part, by
photocopy or other means, without the permission of the author.

ABSTRACT

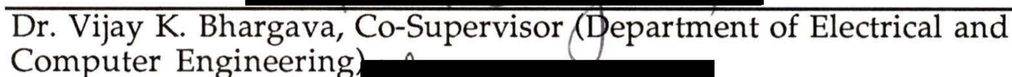
Supervisor: Dr. Warren D. Little

In this thesis, logic cell array designs for a (31, k) Reed-Solomon Codec are examined. The normal basis and a power basis representation for $GF(2^5)$ are considered with respect to the number of clock cycles and number of logical gates or configurable logic blocks required to implement Galois field addition, multiplication and inversion.

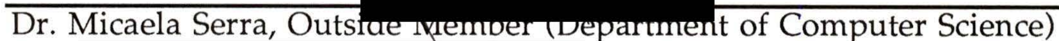
The decoder is partitioned into 5 logical modules. Pipelining options for the 5 logical modules are considered, in light of the resulting speed and hardware requirements. Two methods of passing information between the modules are considered: direct passing of information, and using external memory to store each modules results. A software implementation of the (31, k) codec in the C programming language, and a Xilinx XC3000 Series PGA design for a 5 stage external RAM based decoder are presented.

Examiners:

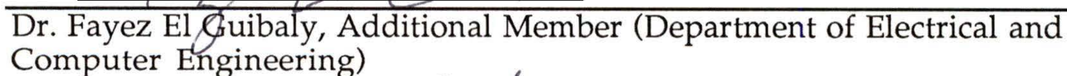
Dr. Warren D. Little, Supervisor (Department of Electrical and Computer Engineering)



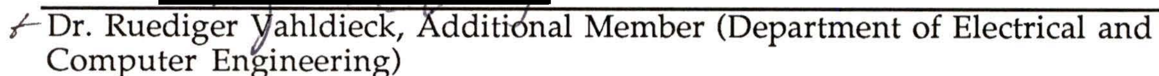
Dr. Vijay K. Bhargava, Co-Supervisor (Department of Electrical and Computer Engineering)



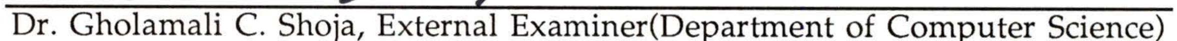
Dr. Micaela Serra, Outside Member (Department of Computer Science)



Dr. Fayez El Guibaly, Additional Member (Department of Electrical and Computer Engineering)



Dr. Ruediger Vahldieck, Additional Member (Department of Electrical and Computer Engineering)



Dr. Gholamali C. Shoja, External Examiner(Department of Computer Science)

TABLE OF CONTENTS

ABSTRACT ii
TABLE OF CONTENTS iii
LIST OF TABLES v
LIST OF FIGURES viii
ACKNOWLEDGEMENTS xii

Chapter1 Introduction 1

Chapter 2 Theory 8

2.1 Galois Fields 8
2.1.1 GF(2m) Basis and Arithmetic 10
2.1.2 Galois Field Addition 11
2.1.3 Galois Field Multiplication 12
2.1.4 Galois Field Inversion 14
2.2 Reed-Solomon Codes 15
2.3 Encoding of RS Codes 16
2.4 Time Domain Decoding of RS Codes 19
2.4.1 Syndrome 19
2.4.2 Error Location Polynomial $\partial(X)$ 20
2.4.3 Finding the Error Locations 22
2.4.4 Error Magnitude Correction 22

Chapter 3 Implementation Options 23

3.1 Implementation of Galois Field Arithmetic 25
3.1.1 Galois Field Adders 30
3.1.2 Galois Field Multipliers 32
3.1.3 Galois Field Inverters 34
3.1.4 Table Look-up of GF Elements 36
3.1.5 GF Arithmetic Summary 37
3.2 Pipelining and Array Passing Options 39
3.2.1 Direct Passing of Intermediate Values 41
3.2.3 RAM Storage of Intermediate Values 45
3.2.4 Changing the Number of Correctable Errors 46
3.3 Pipelining Options 46
3.3.1 Number of Operations per Module 46
3.3.2 Number of Operations per Passing Combination 49
3.3.3 Hardware Requirements of Passing Options 56
3.3.4 Summary of Passing Options 57

Chapter 4 (31,k) RS Codec Designs 62

4.1 Implementation in C 62
4.2 Xilinx Designs 63
4.3 Xilinx Design for the (31,k) RS Encoder 65
4.4 Xilinx Designs for the (31,k) RS Decoder 70

4.4.1 Xilinx Designs for the Syndrome Stage 71
4.4.2 Xilinx Designs for the Delta Stage 74
4.4.3 Xilinx Designs for the Location Stage 77
4.4.4 Xilinx Designs for the Z(X) Stage 79
4.4.5 Xilinx Design for the Correction Stage 80
4.5 Array Passing Logic and Handshaking 82
4.5.1 Direct Passing of Arrays 82
4.5.2 External RAM Array Passing 83
4.5.3 Switching Circuit 85
4.5.4 External Interfaces 88
4.5.5 Clocks 89
4.6 Testing 95
4.7 Conclusion 101

References 103

Appendix A C Implementation of (31,k) RS Codec 106

Appendix B Control Logic for (31,k) RS Codec 120

B.1 Encoder 120
B.2 Decoder 122
B.2.1 Syndrome Stage 122
B.2.2 Delta Stage 125
B.2.3 Location Stage 131
B.2.4 Z(X) Stage 131
B.2.5 Correction Stage 135

Appendix C Listing of Components 141

C.1 Encoder Components 141
C.2 Syndrome Components 142
C.3 Delta Components 143
C.4 Location Components 144
C.5 Z(X) Components 145
C.6 Correction Components 146

Appendix D Power - Vector Representation 147

LIST OF TABLES

Table 3.1.1 Maximum total steps, and breakdown of GF arithmetic steps for the syndrome stage. 27

Table 3.1.2 Maximum total steps, and breakdown of GF arithmetic steps for the delta stage 27

Table 3.1.3 Maximum total steps, and breakdown of GF arithmetic steps for the stage calculating the locations of the errors....28

Table 3.1.4 Maximum total steps, and breakdown of GF arithmetic steps for the stage calculating Z(X). 28

Table 3.1.5 Maximum total steps, and breakdown of GF arithmetic steps for the stage of the RS decoder calculating and correcting the errors. 29

Table 3.1.6 Percentage of total non-memory CLB use required by GF arithmetic units, per module....30

Table 3.1.7 AND and XOR gate requirements for normal basis and power representations of parallel GF arithmetic units. 37

Table 3.1.8 CLB requirements for normal basis and power representation of parallel GF arithmetic units. 37

Table 3.3.1 Number of direct passing steps as a percent of RAM passing steps per decoder module....49

Table 3.3.2 Possible pipelining combinations of five modules....50

Table 3.3.3 Intermediate array hardware requirements for pipelining combinations. 57

Table 3.3.4 Pipelining options giving approximately equal numbers of steps per stage....59

Table 4.3.1 Xilinx chip sizes. 63

Table 4.3.1 Number of steps vs correctable errors for modified LFSR (31,k) encoder algorithm and 5 stage pipelined (31,k) RS decoder. 68

Table 4.3.2 Example of mapping from i-te indicing to linear addressing....68

Table 4.5.1 Intermediate values passed between modules. 86

Table 4.5.2 RAM read and write signals....86

Table 4.5.3 Number of operations required to complete encoder and RAM passing decoder options., including symbol loading of RAM....91

Table 4.5.4 Ratio of encoder clock frequency to that of decoder....93

Table 4.6.1 Xilinx chips used for implementation of RAM passing (31,k) RS codec....95

Table 4.6.2 The speed of pipelining options at te = 15 as a percentage of the speed at te = 1....96

Table C.1.1 Encoder registers, their sizes and features....141

Table C.1.2 Encoder muxes, and their sizes....141

Table C.1.3 Other Encoder components, and their sizes....142

Table C.2.1 Syndrome registers, their sizes and features....142

Table C.2.2 Syndrome muxes, and their sizes....142

Table C.2.3 Other Syndrome components, and their sizes....142

Table C.3.1 Decoder registers, their sizes and features....143

Table C.3.2 Decoder muxes, and their sizes....143

Table C.3.3 Other Decoder components, and their sizes....144

Table C.4.1 Location registers, their sizes and features....144

Table C.4.2 Location muxes, and their sizes....144

Table C.4.3 Other Location components, and their sizes....144

Table C.5.1 Z(X) registers, their sizes and features....145

Table C.5.2 Z(X) muxes, and their sizes....145

Table C.5.3 Other Z(X) components, and their sizes....145

Table C.6.1 Correction registers, their sizes and features....146

Table C.6.2 Correction muxes, and their sizes....146

Table C.6.3 Other Correction components, and their sizes. 146

LIST OF FIGURES

Figure 1.1 Minimum digital communications system.................. 1

Figure 1.2 Digital communications system with error-control coding.. 2

Figure 2.3.1 LFSR coding of RS code words.................................. 18

Figure 3.1.1 Normal basis implementation of GF addition. 31

Figure 3.1.2 GF addition of power representation elements........... 31

Figure 3.1.3 Parallel Massey-Omura multiplier.............................. 32

Figure 3.1.4 Power representation implementation of GF multiplication.. 33

Figure 3.1.5 Normal basis implementation of GF inversion............. 35

Figure 3.2.1 Flow of passed intermediate results between modules... 40

Figure 3.2.2 Direct Data Passing Between Stages 41

Figure 3.2.3.1 Direct passing of intermediate arrays for the no pipelining option 42

Figure 3.2.3.2 Direct passing of intermediate arrays for the 2 stage pipelining option 43

Figure 3.2.3.3 Direct passing of intermediate arrays for the 3-stage pipelining option.. 43

Figure 3.2.3.4 Direct passing of intermediate arrays for the 4 stage pipelining option 44

Figure 3.2.3.5 Direct passing of intermediate arrays for the 5-stage pipelining option.. 44

Figure 3.2.4 RAM based array passing for pipelining options.......... 46

Figure 3.3.1 Number of operations vs correctable errors per decoder module for direct passing of arrays.............. 47

Figure 3.3.2 Number of operations vs correctable errors per decoder module for RAM passing of arrays................ 48

Figure 3.3.3 Number of steps needed to complete possible two stage pipelining option combinations, per number of correctable errors, for direct passing. 51

Figure 3.3.4 Number of steps needed to complete possible three stage pipelining option combinations, per number of correctable errors,for direct passing 52

Figure 3.3.5 Number of steps needed to complete possible four stage pipelining option combinations, per number of correctable errors, for direct passing 53

Figure 3.3.6 Number of steps needed to complete possible two stage pipelining option combinations, per number of correctable errors, for RAM passing 54

Figure 3.3.7 Number of steps needed to complete possible three stage pipelining option combinations, per number of correctable errors, for RAM passing 55

Figure 3.3.8 Number of steps needed to complete possible four stage pipelining option combinations, per number of correctable errors, for RAM passing 56

Figure 3.3.9 Number of operations vs correctable errors per pipelining option for direct passing 60

Figure 3.3.10 Number of operations vs correctable errors per pipelining option for RAM passing 61

Figure 4.2.1 Examples of bus hook-ups for XC 3000 series implementations 65

Figure 4.3.1 Block diagram of modified LFSR implementation of the RS encoder 66

Figure 4.3.3 Data flow block diagram of Xilinx implementation of the programmable (31,k) RS encoder. 70

Figure 4.4.1 Syndrome calculations using a LFSR. 71

Figure 4.4.2 Paired LFSR calculations of syndrome components 72

Figure 4.4.3 Data flow block diagram of module used to calculate syndrome for RAM passing. 73

Figure 4.4.4 Data flow block diagram of module used to calculate syndrome for direct passing. 73
Figure 4.4.5 Data flow block diagram of module used to determine ∂(X) for RAM passing............................... 75

Figure 4.4.6 Data flow block diagram of module used to determine ∂(X) for direct passing.............................. 76

Figure 4.4.7 Data flow block diagram of module used to find the error locations for RAM passing. 78

Figure 4.4.8 Data flow block diagram of module used to find the error locations for direct passing. 78

Figure 4.4.9 Data flow block diagram used to calculate Z(X) for RAM passing 79

Figure 4.4.10 Data flow block diagram used to calculate Z(X) for direct passing.. 80

Figure 4.4.11 Data flow block diagram of module used to correct errors and transmit the corrected information message for RAM passing... 81

Figure 4.4.11 Data flow block diagram of module used to correct errors and transmit the corrected information message for direct passing...................................... 82

Figure 4.5.1 Input buffer for direct passing of arrays...................... 83

Figure 4.5.2 Double buffered input buffer for direct passing of arrays. 83

Figure 4.5.3 Diagram of input of incoming symbols into RAM, for the two module RS decoder............................... 85

Figure 4.5.4 Decoder Module and RAM I/O pins. 87

Figure 4.5.5 Choosing proper RAM chip. 88

Figure.4.5.6 External pins for encoder and decoder. 89

Figure 4.5.7 Block diagram of encoder clock circuit, where the oscillator ratios are given in terms of the RS decoder clock frequency.. 94

Figure 4.6.1 Decoder speeds per pipelining option for RAM passing decoder..96

Figure 4.6.2 Decoder speeds per pipelining option for RAM passing decoder..97

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis advisers Professor Warren Little and Professor V.K. Bhargava for their advice and patience during the course of my research. I would also like to thank Professor Bhargava for his financial support during my thesis studies.

I would also like to thank my family and friends for their continuous encouragement and support.

1 Introduction

A minimum digital communications system consists of an information source, a digital modulator, a transmission channel, a digital demodulator and an information user as shown in Figure 1.1 [1].

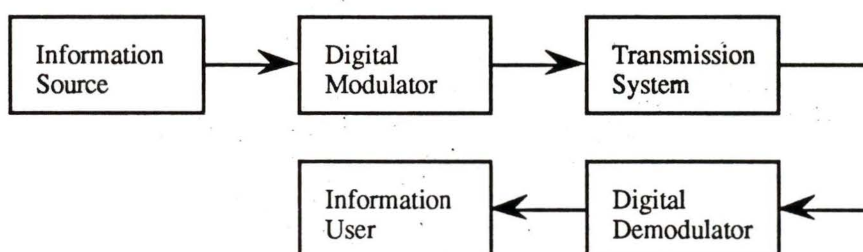


Figure 1.1: Minimum digital communications system.

The process of modulation, transmission and demodulation may introduce errors into the information reaching the user through a number of mechanisms [1]. The received signal at the input of the demodulator always contains broadband noise from thermal processes, and often also contains impulse noise from atmospheric interference, circuitry or deliberate jamming. Broadband noise tends to cause random errors, while impulse noise may cause random errors and/or burst errors. Symbol dispersion, intersymbol interference, and multi-path signal propagation may also make it difficult for the demodulator to correctly determine the message reaching it.

One possible solution to the problem of noise is to raise the signal-to-noise ratio by increasing the signal energy per bit. It is, however, often impossible to raise the signal energy per bit sufficiently, due to either legislated, physical or financial limitations [1]. A second

approach to the reduction of errors due to noise involves the addition of redundancy to the signal, in order to detect and correct errors before they reach the information user. The introduction of redundancy through error-control coding increases the signal bandwidth and the circuit complexity, but in some cases may be the most efficient means of reaching a desired level of accuracy. Error-control coding is added to a digital communications system by adding a digital encoder between the information source and the digital modulator, and adding a digital decoder between the digital demodulator and the information user, as shown in Figure 1.2 [1].

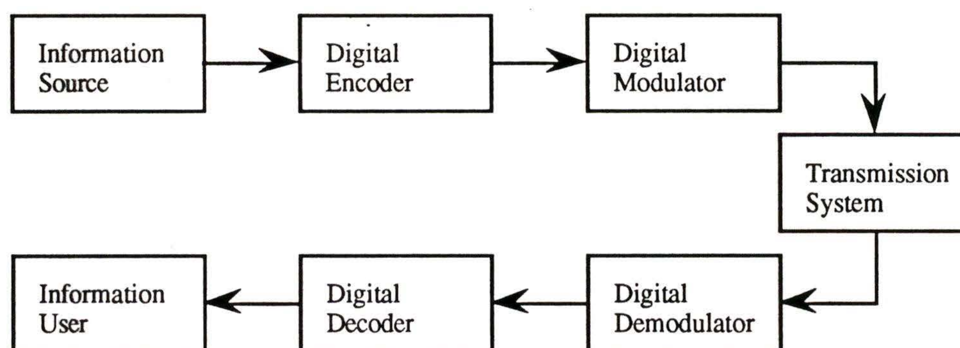


Figure 1.2: Digital communications system with error-control coding.

Error control codes used today may be divided into two general types, block codes and convolution codes [1,2]. The 2^k different information words in a block of k information bits may be encoded into 2^k different n-bit code words, where each n-bit code word has n- k parity check bits. A block code with k information bits and n- k parity check bits is called a (n,k) block code. The ratio of information bits, k, to coded bits n is called the code rate $R = k/n$. A convolution code encodes the incoming information symbols continuously rather

than waiting for a complete block of symbols. The convolutional encoder uses logical circuitry connected to an M-stage shift register to generate the encoded data stream. The number of shifts over which an information bit influences the encoder output is called the constraint length, and is equal to one more than the number of stages in the shift register. Block codes require the complete code word to be received before processing may begin, thus requiring buffers. Convolution codes continuously process incoming data, but become inpractical in cases where a large coding gain and low bit error rates are needed because of the complexity of the decoding algorithm for high constraint length codes [2].

Reed and Solomon [3] discovered in 1960 a class of cyclic error-correcting non-binary block codes called Reed-Solomon codes. The symbols of a Reed-Solomon code are Galois field $GF(2^m)$ elements, where m is a positive integer; each Galois field symbol may be represented by either a single symbol in an 2^m-ary modulation system, or by m bits in a binary modulation system. Since the symbols in a binary RS code are made up of m bits, a burst error of about m bits creates only one or two symbol errors. If the e errors are distributed randomly throughout the transmitted block, approximately e symbol errors would be expected; thus RS codes are more effective against burst errors than random errors. A RS code with symbols from $GF(2^m)$ has a block length of 2^m-1 symbols, and can be arranged to correct t_e errors, where $1 \leq t_e \leq 2^{m-1}-1$. A RS code of block length n symbols coded to correct for t_e symbol errors has $k = n - 2t_e$ information symbols, and is called an (n,k) RS code.

Encoding RS codes is easily done in the time domain with a linear feedback shift-register capable of Galois field multiplication and division, and is relatively simple to implement. The decoding of RS codes is more difficult. Two standard algorithms exist for the decoding of RS codes; a time domain decoding algorithm by Berlekamp [4] and Massey [5], and the frequency domain Fast Fourier Transform method [6]; mixed time and frequency domain algorithms also exist [7]. The designs presented here are based on the time domain decoding algorithm by Berlekamp and Massey.

Algorithms based on the Berlekamp-Massey algorithm correct for a specified number of errors, t_e, in the range $1 \leq t_e \leq 2^{m-1}-1$. It is possible to modify existing algorithms so that the user may specify any number of correctable errors in the possible range; an implementation of such a programmable RS codec allow the user to choose from a range of error correction without changing the transmission system.

In the implementation of a programmable RS codec, decisions must be made on which RS code to implement, on the algorithm to be used to decode the coded blocks, on the medium on which the RS codec is to be implemented, and upon the philosophy of the design. The RS code is chosen based upon the required accuracy of transmission, the expected rate and characteristics of errors, and the complexity of the implementation. Longer RS codes give improved performance with respect to correction [8], and in the case of user programmable codecs, a longer RS code gives a wider range of code rates. Longer codes, however, require more hardware and are thus more costly to implement.

One option to increase the decoding speed of RS codes is to use pipelining. A VLSI design for a pipeline RS transform decoder was proposed in 1985 [9] for a (15,9) RS code, using a modified Euclidean Algorithm [9]. The Euclidean Algorithm for computing the error-location polynomial completely avoids the computation of inverse field elements. In 1988 a pipeline RS time domain decoder using systolic arrays [10] was proposed. However, neither of these designs can be easily modified to correct a user specified number of correctable errors. In 1988 a microprocessor based programmable RS decoder was proposed [11]; however its bit rate is under 100 kilobits per second. A gate- array design for a programmable RS decoder was developed [12] concurrent with this work, based on a universal RS decoder algorithm by Blahut [13] whose speed is independent of the number of correctable errors.

The implementation media yielding the highest codec bit rates are standard cell and custom integrated circuits, and SSI and MSI circuits [14,15]. Standard cell and custom IC's give high density and low power consumption, but are expensive for low production volumes. Standard cell and custom IC's also have a relatively long and difficult design and production process [16]. The complete circuit must be designed, simulated and sent off for at least one month for production before any testing can be done; the simulation, production and testing cycle must be repeated if any errors are found by testing [14]. Implementation in SSI or MSI [17] components allows testing and design to occur concurrently, but lead to complex low density circuits with high power consumption and high volume cost. Digital Signal

Processing (DSP) implementation results in relatively simple circuits, but with a low codec bit rate. Programmable gate arrays (PGA's) [18] give a high density, medium bit rate circuit with low power consumption and concurrent testing and design. In particular, Xilinx Inc. [19] makes a family of PGA's, the XC3000 Series, whose five input logic functions are well suited to operating in $GF(2^5)$.

The architecture of Xilinx's Logic Cell Array (LCA) [15] technology consists of a matrix of configurable logic blocks (CLB's) inside an outer ring of I/O interface blocks (I/O pins). The functions and interconnections of the CLB's and I/O pins are stored in a file, either on a host computer or stored in external memory. The use of programmable gate arrays is based on the need for a quick turn-around time in the design and testing of a circuit. The design is down loaded into the Xilinx chip either directly from a computer terminal, or from PROM immediately before the use of the chip. A change in design only requires changing the file loaded into the PROM, in contrast to a period of weeks and considerable more cost required for changing a custom integrated circuit. If volume production of the PGA decoder is required, software exists [19] to convert between the files needed by Xilinx and the standard schematic capture format used in custom integrated circuit.

I introduce several new hardware design options for time-domain Reed-Solomon decoders here. The user can choose the number of correctable errors t_e for the programmable (31,k) RS encoder and decoder. The number of correctable errors may be different for each coded word; that is, the user may change the number of correctable

errors on the fly. Pipeline RS codec designs are introduced using the standard Berlekamp algorithm, using an efficient single step GF inverter. The designs are based on a Programmable Gate Array implementation, rather than a VLSI or software implementation. Designs are presented for either on-chip memory or external RAM storage of intermediate results, which allow the degree of pipelining to be easily modified. Finally, I exploit the power representation for Galois field elements is introduced here, allowing efficient single step GF operations.

A key consideration of the design presented here is to maintain a high level of modularity in the design process. The RS decoding algorithm may be separated into five sections, each section dependent on previous sections only for a few intermediate results. These sections can be implemented as independent modules, with the proper connecting logic. Each of these modules can be implemented through of a number of standard registers, Galois field and integer arithmetic units, and logical devices such as multiplexers and counters. The high level of modularity leads to a design slightly less efficient in terms of space when compared to a design based upon streamlining each component. The modular design is, however, easier to understand, debug, and modify. For example, components or groups of components may be downloaded onto the Xilinx chip for isolated testing. The modular design also allows the same design with minor alterations to be used for different size Reed-Solomon codecs.

2 Theory

2.1 Galois Fields

The theory needed for the coding and decoding of Reed-Solomon codes can be found in several standard texts on error-control coding [1,2,4,20,21] or finite field algebra [22]. A summary is given here. The bits in a Reed-Solomon code block are grouped into Galois field elements. In a (31, k) RS code block, for instance, the 155 bits of the block are treated as 31 elements from the Galois field $GF(2^5)$. A non-empty set S and a pair of operations $(\bullet, \oplus)$ are a field F if:

(i) S is a commutative group under $\oplus$.

(ii) The set of non-zero elements in S is a commutative group under $\bullet$.

(iii) The $\bullet$ operator is distributive over $\oplus$.

There exists for any prime number p a finite field of p elements, also called the Galois field GF(p). The p elements must include the zero element 0, and p-1 non-zero elements, including the multiplicative identity element 1. As the multiplicative group is closed under multiplication, the powers $\alpha, \alpha^2, \alpha^3, \ldots$ of any element α in GF(p) must also be in GF(p). Since the number of elements in the field GF(p) is finite, there must exist some smallest integer e such that $\alpha^e = 1$. This integer e is called the order of the field element α. The elements $1, \alpha, \alpha^2, \ldots, \alpha^{e-1}$ form a subgroup called a cyclic group.

If the order of α is p-1, that is , if $\alpha^{p-1} = 1$, then α is called a primitive element α of GF(p). For any GF(p) there exists a primitive element α. Since the cyclic group $1, \alpha, \alpha^2, \ldots, \alpha^{p-1}$ represent distinct elements, every non-zero element of GF(p) can be expressed as a power of a primitive element α.

Given a polynomial $f(X) = f_0 + f_1X + f_2X^2 + \ldots + f_nX^n$, if there exist two other polynomials s(X) and m(X) such that $s(X) \bullet m(X) = f(X)$, then it is said that s(X) and m(X) divide f(X) . If f(X) is not divisible by any polynomial of degree p, where $0 < p < n$, then f(X) is said to be irreducible. Monic irreducible polynomials play the same role in polynomial algebra as prime numbers do in integer theory. Given two polynomials f(X) and p(X) , the monic polynomial of greatest degree which divides both of them is called their greatest common divisor. If the greatest common divisor = 1 , the polynomials are said to be relatively prime.

For the polynomials mod m(X) to be a field, m(X) must be an irreducible polynomial. If m(X) is an irreducible polynomial with coefficients in GF(p), then the algebra of polynomials over GF(p) modulo m(X) is a field.

Starting with a field K, we may wish to construct a larger field L. One way of doing this is by creating m-tuples whose components belong to the field K, creating a vector space over K. That is, the field L is the space $L = K \times K \times \ldots \times K$. An example of this is the construction of the field of complex numbers from the field of real numbers.

If our ground field is GF(p), we can map the constructed field L to the field formed by taking polynomials modulo an irreducible polynomial m(X) of degree m, whose coefficients come from GF(p). This is called the extension field $GF(p^m)$.

If β is a root of m(X) in the extension field $GF(p^m)$, then β^n, where n = 1, p, ..., m-1, are all the roots of m(X), and have the

same order. If the irreducible polynomial m(X) has a primitive element α in $GF(p^m)$ as a root , it is called a primitive polynomial, and all its roots are primitive elements. We can construct the Galois field $GF(2^m)$ from the primitive polynomial

$p(X) = p_0 + p_1X + p_2X^2 + \ldots p_{m-1}X^{m-1}$,

with coefficients p_i from GF(2), as follows [2].

If α denotes a primitive element in $GF(2^m)$, then the elements of $GF(2^m)$ are $\{0, 1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2^m-2}\}$. If $\{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1}\}$ is a basis for $GF(2^m)$, then any element $\alpha^j \in GF(2^m)$ may be written as

$$\alpha^j = a_{j0}\,\alpha_0 + a_{j1}\,\alpha_1 + \ldots + a_{j,m-1}\,\alpha_{m-1}.$$

Since α is a root of p(X), $p(\alpha) = 0$, so

$$p(\alpha) = p_0 + p_1\alpha + p_2\alpha^2 + \ldots. + p_{m-1}\alpha^{m-1} = 0.$$

This gives a relationship between the elements $\alpha^j \in GF(2^m)$ which can be used iteratively to find their vector representation.

2.1.1 GF(2^m) Basis and Arithmetic

In this work only the extension fields $GF(2^m)$ are used. The complexity of implementation of $GF(2^m)$ addition, multiplication and inversion is dependent on the size m of the field and on the basis used to represent the $GF(2^m)$ elements. As discussed above, if α is a primitive element in $GF(2^m)$, then the non-zero elements $GF(2^m)$ may be constructed from the powers of α. That is, $GF(2^m)$ may be represented by

$$0,\ \alpha^0,\ \alpha^1, \ldots,\ \alpha^{2^n}$$

The elements α^i in $GF(2^m)$ may also be represented as m-tuples in some basis. A common choice of basis is the sequence [2] $(\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{m-1})$. It has been suggested [23] that implementation of

Galois field arithmetic is more easily done using the normal basis $(\alpha^1, \alpha^2, \alpha^4, \ldots, \alpha^{2(m-1)})$.

In addition to the standard choices of vector representations of the Galois field elements, the power representation (also called the anti-log) may also be used to represent the Galois field elements [1,2]. The power representation p of a field element $\beta \in GF(2^m)$, is defined by $\alpha^p = \beta$, where a is a primitive element of $GF(2^m)$.

In the power representation the integer 0 refers to α^0, and the additive zero element is represented by the integer $2^m - 1$. It is possible to preserve the representation of the additive zero element by the integer 0 if the integer representations are increased by one from the power representation; for example α^p would be represented by the integer (p + 1) in the power-add-1 representation.

2.1.2 Galois Field Addition

In vector representation, Galois field addition [2] is simply bit-wise integer addition modulo 2. For α, β, χ from $GF(2^5)$, where $\alpha = (a_0\ a_1\ a_2\ a_3\ a_4)$, $\beta = (b_0\ b_1\ b_2\ b_3\ b_4)$, $\chi = (c_0\ c_1\ c_2\ c_3\ c_4)$, if $\chi = \alpha \oplus \beta$ then for i = 0,1,2,3,4:

$$c_i = a_i \oplus b_i$$

In the power representation Galois field addition may be performed by translating from the power representation into a vector representation, adding bit-wise modulo 2, and translating back into the power representation. The mapping between the power and the vector representations may be done either through table look-up or by logical circuitry. In general, the mapping function is different for each of the m-bits $a'_k = f(a_0\ a_1\ a_2 \ldots a_{m-1})$, $k = 0,1,2,\ldots,m-1$. A bit a'_k will be

positive for half the $GF(2^m)$ elements, the subset Ps. In general, the mapping for the bit a'_k can be implemented by setting the bit a_k to 1 whenever one of the elements of Ps is seen. The equation for this is given by:

$$a_k^T = \sum_{i=1}^{2^{m-1}} \prod_{l=0}^{m-1} c_i^l a_l$$
.

The operator c_i^l either leaves a_k as it is, or inverts it. In the worst case, each bit is then calculated using 2^{m-1} XOR gates and $2^{m-1}(m-1)$ AND gates. However, fewer gates may be used by recombining the equations. See Appendix C for the translation logic between the power and normal representations used in the designs presented here.

2.1.3 Galois Field Multiplication

Choosing the normal basis for the vector representation allows the parallel Massey-Omura multiplier [24] to be used for the Galois field multiplication. In particular, the least complex Massey-Omura multiplier may be used in a Xilinx implementation [25].

For α ,β, χ from $GF(2^5)$, where $\alpha = (a_0\ a_1\ a_2\ a_3\ a_4)$, $\beta = (b_0\ b_1\ b_2\ b_3\ b_4)$, $\chi = (c_0\ c_1\ c_2\ c_3\ c_4)$, if $\chi = \alpha \bullet \beta$ then

$$c_0 = f(a_1, a_2, a_3, a_4, a_0 ; b_1, b_2, b_3, b_4, b_0)$$

$$c_1 = f(a_2, a_3, a_4, a_0, a_1 ; b_2, b_3, b_4, b_0, b_1)$$

$$c_2 = f(a_3, a_4, a_0, a_1, a_2 ; b_3, b_4, b_0, b_1, b_2)$$

$$c_3 = f(a_4, a_0, a_1, a_2, a_3 ; b_4, b_0, b_1, b_2, b_3)$$

$$c_4 = f(a_0, a_1, a_2, a_3, a_4 ; b_0, b_1, b_2, b_3, b_4)$$

where

$$f(a_0, a_1, a_2, a_3, a_4 ; b_0, b_1, b_2, b_3, b_4) = a_0 b_0 \oplus a_2 b_0 \oplus a_0 b_4 \oplus a_4 b_0 \oplus a_1 b_2 \oplus a_2 b_1 \oplus a_1 b_3 \oplus a_3 b_1 \oplus a_2 b_3 \oplus a_3 b_2$$

In the power representation Galois field multiplication over the field $GF(2^m)$ becomes integer addition modulo $2^m - 1$. Since the greatest possible result of integer addition of two numbers a, b less than $2^m - 1$ is $a + b \leq 2^{m+1} - 4 < 2^{m+1} - 2$, the modulo operation need only be carried out once. Implementation of the modulo $2^m - 1$ operation can than be carried out either by incrementing the sum of a and b by 1 if the sum is equal to or greater than $2^m - 1$, or by carrying out the multiplication with a normal integer addition (a + b) in parallel with an add-one addition (1+ a + b), with the result chosen from the add-one unit if the result of the normal addition is greater than $2^m - 2$. For $\alpha, \beta, \chi \in GF(2^5)$, where $\alpha = (a_0\, a_1\, a_2\, a_3\, a_4)$, $\beta = (b_0\, b_1\, b_2\, b_3\, b_4)$, $\chi = (c_0\, c_1\, c_2\, c_3\, c_4)$, the GF product $\alpha \bullet \beta = \chi$ is given by:

If $\alpha + \beta < 2^m - 1$, then:

$$c_0 = a_0 \oplus b_0$$

$$m_0 = a_0 \bullet b_0$$

$$c_i = m_{i-1} \oplus a_i \oplus b_i$$

$$m_i = a_i \bullet (b_i + m_{i-1}) + b_i \bullet m_{i-1}$$

for i = 1, 2, 3, 4, and where m_i is the ith carry bit.

If $\alpha + \beta \geq 2^m - 1$, then $\chi = \chi f$ and:

$$cf_0 = \sim(a_0 \oplus b_0)$$

$$mf_0 = \sim(a_0 \oplus b_0)$$

$$cf_i = mf_i \oplus a_i \oplus b_i$$

$$mf_i = a_i \bullet (b_i + m_{i-1}) + b_i \bullet m_{i-1}$$

for i = 1 ,2, 3, 4, and where mf_i is the ith carry bit.

2.1.4 Galois Field Inversion

The pipelined normal basis Galois field inverter may be constructed from the parallel Massey-Omura multiplier [24]. If α belongs to $GF(2^5)$ and $c = \alpha^{-1}$, then $c = \alpha \bullet \alpha^2 \bullet \alpha^4 \bullet \alpha^8 \bullet \alpha^{16}$. For α in a normal basis, α^2 is obtained by a cyclic shift operation. The GF inverse α^{-1} is given by the algorithm:

```
C = α;
B = α2 ;
k = 0;
while (k < 5) {
    D = C • B;
    B = B2;
    C = D;
    k = k+1;
}
α-1 = D;
```

In the power representation, inversion over the field $GF(2^5)$ only requires inversion of each bit of the field element. A special case must be made for the multiplication identity element α^0 represented by the integer 0; the inverter must return the integer 0 if the integer 0 is input. For $A, C \in GF(2^5)$, where $A = (a_0\ a_1\ a_2\ a_3\ a_4)$, $C = (c_0\ c_1\ c_2\ c_3\ c_4)$, the GF inversion $A^{-1} = C$ is given by:

If power(A) $\neq 0$

$c_i = \sim a_i$

else

$c_i = 0$

The logic required to check if the integer representation of A is zero is given by:

$$z = \sim c_0 \bullet \sim c_1 \bullet \sim c_2 \bullet \sim c_3 \bullet \sim c_4$$

and the inverted bits ci are given by:

$$c_i = \sim a_i \bullet \sim z$$

for i = 0, 1, 2, 3, 4.

2.2 Reed-Solomon Codes

The theory of Reed-Solomon codes may be found in standard works on error control coding [1,2,4,20,21]. A brief summary is given here. The symbols of a Reed-Solomon code are Galois field GF(q) elements. A t-error correcting RS code with elements from GF(q) has the following parameters:

block length	$n = q - 1$
parity check digits	$n - k = 2t$
minimum distance	$d_{min} = 2t + 1$

An RS code with symbols from $GF(2^m)$ has a block length of $2^m - 1$ symbols, and can be designed to correct t_e errors, where

$$1 \le t_e \le 2^{m-1} - 1.$$

A t_e error correcting RS code of block length $n = 2^m - 1$ has $2t_e$ parity check and $n - 2t_e$ information symbols per block. For two code vectors r and s, the distance between them is given by the number of digits in which they differ. The minimum distance is the smallest distance between any two code words.

A code can only detect an error if a non-code word results from the error. Since any number of errors is possible, we cannot guarantee finding the transmitted word. If the probability of error is small we can assume that the number of errors that occurred equals the distance between the received word and the closest code word.

2.3 Encoding of RS Codes

A description of the theory behind the encoding of Reed-Solomon codes can be found in the standard error-control coding texts [1] [2] [4] [20]. The RS code word can be represented by a code polynomial V(X), where

$$V(X) = V_0 + V_1X + V_2X^2 + \ldots + V_{n-1}X^{n-1}$$

The coefficients V_i are from GF(n+1) = GF(q), where q is either a prime number or a power of a prime number. As information is commonly sent in binary form, the most common RS codes are based on $GF(2^m)$. The information to be transmitted in the coded word can also be represented by an information polynomial U(X)

$$U(X) = U_0 + U_1X + U_2X^2 + \ldots + U_{k-1}X^{k-1}\text{ , where } k \le n.$$

Encoding is then the process of creating V(X) from U(X). This process may be represented by another polynomial g(X), called the generator polynomial. Since V(X) is of degree n-1 and U(X) of degree k-1, g(X) must be a polynomial of degree (n-1) - (k-1) = n - k. The code polynomial V(X) is then calculated by the GF product

$$V(X) = U(X)g(X) = (U_0 + U_1X + U_2X^2 + \ldots + U_{k-1}X^{k-1}) \bullet$$
$$(g_0 + g_1X + g_2X^2 + \ldots + g_{n-k}X^{n-k})$$

The generator polynomial g(X) of a t-error correcting RS code of length $n = 2^m - 1$ is given by

$$g(X) = (X + \alpha)(X + \alpha^2)(X + \alpha^3)\ldots(X + \alpha^{2t})$$

More generally, we could also take

$$g(x) = \prod_{j=b}^{b+2t-1}(x - \alpha^j) = \sum_{i=0}^{2t} g_i X^i$$

Creation of the code polynomial V(X) involves computing 2t check symbols given k information symbols, so that the resulting $k + 2t = N$ symbols are coefficients of a code polynomial. That is, the symbols make up a code word. The 2t parity check symbols and the k information digits making up the word can be sent in any order. One way of ordering the digits is called systematic form. Here the left-most n-k digits are parity check digits. A (3,2) code with a parity bit chosen so that the sum of the code bits is 0 mod 2 could send the information $u = (1\ 0)$ as $(1\ 0\ 1)$ or $(1\ 1\ 0)$ or $(0\ 1\ 1)$. Of these, the second representation is in systematic form.

In systematic form, the 2t parity check digits are the coefficients of the remainder polynomial

$$b(X) = b_0 + b_1X + b_2X^2 + \ldots + b_{2t-1}X^{2t-1}$$

resulting from the division of the product $X^{2t}\,U(X)$ by the generator polynomial g(X)

$$b(X) = X^{2t}U(X) \bmod g(X).$$

A RS encoder then basically performs polynomial division in a finite field. A linear feedback shift register, such as that shown in Figure 2.3.1, may be used to produce a RS code word from an information message. TjeCode for the RS encoder is given in Figure 2.3.2.

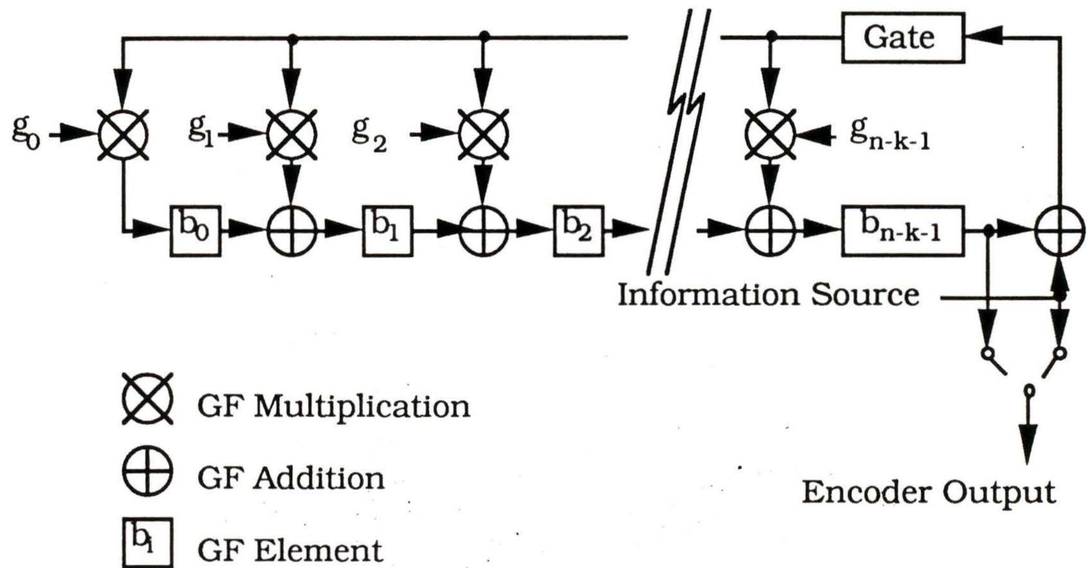


Figure 2.3.1: LFSR coding of RS code words.

```
set bi = 0 for i = 0 to i = 29;
i = 30;
while (i > 2te-1) do:
        gate = b2te-1 + Ui;
        k = 2te - 1;
        while (k > 0) do
                bk = gte,k  * gate + bk-1;
                k = k - 1;
        end while;
        i = i - 1;
end while;
b0 = gte,gate;
shift to the write for i = 0 to 2te-1;
```

Figure 2.3.2: Code for the RS Encoder.

2.4 Time Domain Decoding of RS Codes

The theory behind the decoding of Reed-Solomon codes can be found in the standard error-control coding textbooks [1,2,4,20,21]. Comparisons of RS decoding techniques have been made, particularly for software implementations [26,27].

Decoding an RS code involves four basic logical modules:

(1) computing the syndrome components S_i, $i = 1,2,\ldots,2t_e$.

(2) determining the error-location polynomial $\delta(X)$.

(3) finding the roots of the error-location polynomial $\delta(X)$.

(4) finding and correcting the error at each error location.

2.4.1 Syndrome

The symbols r_k in the received block of an RS code are the coefficients of the received vector

$$r(X) = r_0 + r_1X + r_2X^2 + \ldots + r_{n-1}X^{n-1}$$

where k is the position of the symbol r_k in the received block. The received vector r(X) is the sum

$$r(X) = t(X) + e(X)$$

where t(X) is the transmitted vector and e(X) is the vector of errors introduced during transmission. The $2t_e$ syndrome components are found by substituting α^i into the received vector r(X) for $i = 1,2,\ldots,2t_e$:

$$S_i = r(\alpha^i) = t(\alpha^i) + e(\alpha^i) = e(\alpha^i) \quad (2.4.1)$$

where α^i is a primitive element of $GF(2^m)$. The syndrome component is given by

$$S_0 = ((\ldots(r_{30} \bullet \alpha + r_{29}) \bullet \alpha + r_{28}) \ldots + r_{2te})$$

2.4.2 Error Location Polynomial $\delta(X)$

The decoder must find the locations of the the errors introduced by transmission; that is, the non-zero coefficients of e(X). Directly solving the system of equations (2.4.1) is difficult; an alternative approach is to introduce a polynomial, the error- location polynomial $\delta(X)$, whose roots are the locations of the transmission errors.

$$\delta(X) = (1+\beta_1 X)(1+\beta_2 X)\ldots(1+\beta_\mu X)$$
$$= \delta_0 + \delta_1 X + \delta_2 X^2 + \ldots + \delta_\mu X^\mu$$

β_i is the location of the ith error, and μ is the number of errors introduced during transmission. A number of methods exist for determining the error location-polynomial, $\delta(X)$; Berlekamp's iterative method is used in the programmable (31,k) Reed-Solomon decoder.

The algorithm used to calculate $\delta(X)$ is given in Figure 2.4.1.
d_i = the difference for the i-th line.
l_i = the order of the i-th line.
$\delta_p[0]$ = the error location polynomial for the p-th line.

```
dp = lp = lμ = 0;
δp[0] = δ[0] = α^0;
p = -1;
set δp[i] = 0 for i = 1 to te;
set δ[i] = 0 for i = 1 to te;

dμ = S1;
μ = 0;
while (μ < 2te) do:
        dμ-1 = dμ;
        lμ-1 = lμ;
        set δμ-1[i] = δ[i] for i = 0 to te;

        if (dμ ≠ 0) do:
                set shifted_δ[i] = 0 for i = 0 to te;
                shift_order = μ - p;
                d_fact = dμ * dp^-1;
```

```
            i = shift_order;
            while (i ≤ l_p + shift_order) do:
                  shifted_δ[i] = d_fact * δ_p[i - shift_order];
                  i = i + 1;
            end while;
            k = 0;
            while (k ≤ t_e) do:
                  δ[k] = (δ[k] + shifted_δ [k];
                  k = k + 1;
            end while;
            if (l_μ < (l_p + shift_order)) do:
                  l_μ = l_p + shift_order;
            end if;
            if (d_μ-1 ≠ 0 AND (p-l_p < μ - l_μ-1)) do:
                  p = μ;
                  d_p = d_μ-1;
                  l_p = l_μ-1;
            end if;
            i = 0;
            while (i ≤ t_e) do:
                  δ_p[i] = δ_μ-1[i];
                  i = i + 1;
            end while;
      end if;
      d_μ = S_μ+2;
      i = μ + 1;
      n = 1;
      while (n ≤ l_μ) do:
            d_μ = d_μ + S_i * δ[n];
            n = n + 1;
            i = i - 1;
      end while;
      μ = μ + 1;
end while;
```

Figure 2.4.1: Pseudocode for determining δ(X).

2.4.3 Finding the Error Locations

The roots β_i of the error-location polynomial must be found. This can be done either by the Chien search [1] or by substituting all the elements α^j in $GF(2^m)$ into the error locator polynomial and noting the elements which give $\delta(\alpha^j) = 0$.

2.4.4 Error Magnitude Computation

Finding the errors at the error locations requires solving the equations:

$$S_k = Y_1\beta_1 + Y_2\beta_2 + \ldots + Y_\mu\beta_\mu \quad \text{where} \quad k = 1, 2, \ldots, 2t$$

for the μ errors $Y_1, \ldots, Y_\mu$. An easy way of doing this is to first find the function $Z(X)$:

$$Z(X) = 1 + (S_1 + \delta_1)X + (S_2 + \delta_1 S_2 + \delta_2)X^2 + \ldots + (S_\mu + \delta_1 S_{\mu-1} + \delta_2 S_{\mu-2} + \ldots + \delta_\mu)X^\mu$$

The μ errors Y_i can then be found using

$$Y_i = \frac{Z(\beta_i^{-1})}{\prod_{\substack{j=1 \\ j \neq i}}^{\mu} (1 + \beta_i \beta_j^{-1})}$$

The transmitted vector $t(X)$ can then be found by adding the error vector $e(X)$ to the received vector $r(X)$:

$$t(X) = e(X) + r(X).$$

3 Implementation Options

Modifying the RS encoding and decoding algorithms so as to make them user programmable for the number of correctable errors is relatively simple, but creates a number of implementation problems. In the case of the RS encoder, the coefficients g_i of the generator polynomial

$$g(X) = g_0 + g_1(X) + g_2(X^2) + \ldots + g_{2te}X^{2te}$$

must be available for each case of the number of correctable errors t_e in the range $1 \leq t_e \leq 2^{m-1}-1$. For the RS decoder, the amount of storage space needed for intermediate value arrays increases as the number of correctable errors increases. The programmable (n,k) RS decoder must have enough storage space for the highest number of correctable errors $t_{e,max}$ for a given n; this means that at low numbers of correctable errors most of the storage space is unused. Buffers and control logic also must be added to allow the the number of correctable errors to be changed while the codec is in operation.

Design options also exist that apply to both the programmable RS codec and to the fixed error correcting RS codec. These decisions are made on the basis of how they will effect the trade-off between speed and hardware requirements. The decisions are not independent; decisions made on different aspects of the implementation affect each other. Implementation of the algorithm requires decisions to be made on the basis use to represent the Galois field, and on the design of the Galois field arithmetic units. The layout of internal registers and bus lines must also be considered. A number of pipeline options exist for the RS decoder; the amount of pipelining

to be used depends upon the speed requirements and space limitations. If a pipelined codec is needed, the external logic and memory needed to co-ordinate the stages can be implemented in a number of ways.

The design for a Logic Cell Array implementation of a RS decoder presented here is based on the five modules (stages) listed in Chapter 2 and shown in Chapter 4: the module calculating the syndrome (the syndrome stage), the module determining the error-location polynomial (the delta stage), the module finding the locations of the errors (the location stage), the module calculating Z(X) (the Z(X) stage), and the module correcting the errors (the correction stage). The algorithms used in these stages are based on the Berlekamp-Massey algorithm for the stages as shown in Chapter 2, modified for Logic Cell Array (LCA) implementation (see Appendix B).

The effect of a design option on the speed of the RS codec will be measured by the option's effect on the number of clock cycles (steps) required to complete the decoding of a RS code word using the LCA implementation algorithms. The number of steps varies with the number of correctable errors, and with the method used to pass intermediate value arrays between stages (see Section 3.2); directly passing the arrays between modules, and storing the arrays in external RAM are considered here.

3.1 Implementation of Galois Field Arithmetic

The RS encoding and decoding algorithms can in theory be written so as to be programmable over any Galois field; for example the user could choose between (31,k) and (63,k) RS codes over GF(2^5) and GF(2^6) respectively. However, the implementation of Galois field arithmetic is difficult to generalize, and the storage space in either software or hardware for the Galois field arithmetic operations increases with each additional field supported. The hardware implementation runs into further difficulties due to the dependency of bus and register widths on the field size. That is, a (31,k) RS code has bus and register widths of 5 bits, whereas a (63,k) RS code has bus and register widths of 6 bits.

Previous implementations of Galois field arithmetic have been done using vector representations; the normal basis in particular has been suggested as an efficient basis [23]. In this chapter it will be argued that the power representation allows a more efficient Xilinx XC3000 Series Programmable Gate Array (PGA) implementation for GF(2^5) arithmetic, and has some advantages over the normal basis implementations when basic logic gates are considered.

The advantages and disadvantages of a representation lies in the implementation it allows for Galois field arithmetic, and in the complexity of the hardware needed to implement the decoder in the representation. The power representation requires less configurable logic blocks (CLB's) than does the normal basis representation for Xilinx XC3000 Series implementation of arithmetic over GF(2^5). The power representation also has the advantage of allowing the Galois

field elements to be used as incremental counters; this simplifies the design of some parts of the decoder.

In the power representation as used in this work the integer 0 refers to α^0, and the additive zero element is represented by the integer 31. As discussed in Chapter 2, it is possible to preserve the representation of the additive zero element by the integer 0 if the integer representations are increased by one from the power representation; for example α^{27} would be represented by the integer 28 in the power-add-1 representation. It is slightly simpler to implement Galois field arithmetic in the power-add-1 representation than the true power representation; however, the true power representation allows the direct use of the element as a counter in loops and as a pointer in accessing arrays. Further discussion of the power representation in this work always refers to the true power representation.

It is possible to implement Galois Field multiplication and inversion operations as a pipelined operation taking two or more steps, or as a parallel operation carried out in one step. The advantage of the pipelined Galois field arithmetic units is the saving in hardware they offer. However, the number of decoding steps involving GF multiplication or inversion operations is high. In the case of the passed array (31,k) RS decoder design presented here, GF multiplication or inversion operations make up between 13% to 47% of the total number of steps in the five modules making up the RS decoder (Tables 3.1.1 to 3.1.5). The use of pipelined GF multiplication and/or inversion units would significantly increase the

number of decoder steps.

Correct. Errors	Total Steps	% GF Mult or GF Inv	GF Arith Steps	GF Add Steps	GF Mult Steps	GF Inv Steps
1	128	46	122	62	60	0
2	255	47	244	124	120	0
3	382	47	366	186	180	0
4	509	47	488	248	240	0
5	636	47	610	310	300	0
6	763	47	732	372	360	0
7	890	47	854	434	420	0
8	1017	47	976	496	480	0
9	1144	47	1098	558	540	0
10	1271	47	1220	620	600	0
11	1398	47	1342	682	660	0
12	1525	47	1464	744	720	0
13	1652	47	1586	806	780	0
14	1779	47	1708	868	840	0
15	1906	47	1830	930	900	0

Table 3.1.1: Maximum total steps, and breakdown of GF arithmetic steps for the syndrome stage.

Correct. Errors	Total Steps	% GF Mult or GF Inv	GF Arith Steps	GF Add Steps	GF Mult Steps	GF Inv Steps
1	56	14	14	6	6	2
2	129	14	37	18	15	4
3	222	15	70	36	28	6
4	335	15	113	60	45	8
5	468	16	166	90	66	10
6	621	16	229	126	91	12
7	794	16	302	168	120	14
8	987	17	385	216	153	16
9	1200	17	478	270	190	18
10	1433	17	581	330	231	20
11	1686	17	694	396	276	22
12	1959	17	817	468	325	24
13	2252	17	950	546	378	26
14	2565	18	1093	630	435	28
15	2898	18	1246	720	496	30

Table 3.1.2: Maximum total steps, and breakdown of GF arithmetic steps for the stage determining the error-location polynomial $\delta(X)$.

Correct. Errors	Total Steps	% GF Mult or GF Inv	GF Arith Steps	GF Add Steps	GF Mult Steps	GF Inv Steps
1	220	14	63	31	31	1
2	314	20	126	62	62	2
3	408	23	189	93	93	3
4	502	25	252	124	124	4
5	596	26	315	155	155	5
6	690	27	378	186	186	6
7	784	28	441	217	217	7
8	878	29	504	248	248	8
9	972	29	567	279	279	9
10	1066	30	630	310	310	10
11	1160	30	693	341	341	11
12	1254	30	756	372	372	12
13	1348	30	819	403	403	13
14	1442	31	882	434	434	14
15	1536	31	945	465	465	15

Table 3.1.3 : Maximum total steps, and breakdown of GF arithmetic steps for the stage calculating the locations of the errors.

Correct. Errors	Total Steps	% GF Mult or GF Inv	GF Arith Steps	GF Add Steps	GF Mult Steps	GF Inv Steps
1	15	13	4	2	2	0
2	31	16	10	5	5	0
3	51	17	18	9	9	0
4	75	18	28	14	14	0
5	103	19	40	20	20	0
6	135	20	54	27	27	0
7	171	20	70	35	35	0
8	211	20	88	44	44	0
9	255	21	108	54	54	0
10	303	21	130	65	65	0
11	355	21	154	77	77	0
12	411	21	180	90	90	0
13	471	22	208	104	104	0
14	535	22	238	119	119	0
15	603	22	270	135	135	0

Table 3.1.4 : Maximum total steps, and breakdown of GF arithmetic steps for the stage calculating Z(X).

Correct. Errors	Total Steps	% GF Mult or GF Inv	GF Arith Steps	GF Add Steps	GF Mult Steps	GF Inv Steps
1	18	11	4	2	1	1
2	56	28	24	10	6	10
3	118	35	60	24	15	27
4	204	39	112	44	28	52
5	314	41	180	70	45	85
6	448	42	264	102	66	126
7	606	43	364	140	91	175
8	788	44	480	184	120	232
9	994	45	612	234	153	297
10	1224	46	760	290	190	370
11	1478	46	924	352	231	451
12	1756	46	1104	420	276	540
13	2058	46	1300	494	325	637
14	2384	46	1512	574	378	742
15	2734	47	1740	660	435	855

Table 3.1.5 : Maximum total steps, and breakdown of GF arithmetic steps for the stage of the RS decoder calculating and correcting the errors.

Parallel Galois field arithmetic units increase the speed of the RS Codec at the cost of space. The number of CLB's used in the Xilinx XC3000 Series implementation by the parallel power representation GF arithmetic units varies from 10% to 42% of the total number of CLB's in each of the codec modules, excluding memory used to store arrays passed between modules (see Table 3.1.6). In the case of the non-pipelined RS decoder, only one GF adder, one GF multiplier and one GF inverter is required, making up 6% of the total decoder CLB usage. However, the routing may become so complex that more than one of each GF arithmetic unit may be desirable. In this work only the parallel Galois field arithmetic units will be examined.

Module	Hardware (CLB's)*	GF Arithmetic (CLB's)	% Total CLB's in GF arith. units
Encoder	245	33	13
Syndrome	81	33	40
Delta	338	37	10
Location	87	37	42
Z(X)	89	33	37
Error Correction	114	37	32
Non-pipelined	710	37	5

* - does not include CLB's used to store arrays passed between stages.

Table 3.1.6 Percentage of total non-memory CLB use required by GF arithmetic units, per module.

The comparison of vector and power representation implementations of Galois field arithmetic is dependent on the size of the Galois field, and on the medium in which the arithmetic unit is implemented. The hardware requirements for implementation of Galois field addition, multiplication, and inversion over GF(2^5) for the normal and power basis representations are discussed in terms of basic logic AND and OR gates [28], and in terms of Xilinx XC3000 Series CLB's. Table 3.1.7 and Table 3.1.8 summarize the hardware requirements of the GF arithmetic units in terms of basic logic gates and Xilinx XC3000 Series CLB's, respectively.

3.1.1 Galois Field Adders

In the vector representation, Galois field addition is simply bit-wise integer addition modulo 2, and the GF adder may be implemented as shown in Figure 3.1.1. In the power representation Galois field addition may be performed by translating from the power representation into a vector representation, adding bit-wise modulo 2, and translating back into the power representation, as shown in Figure 3.1.2.

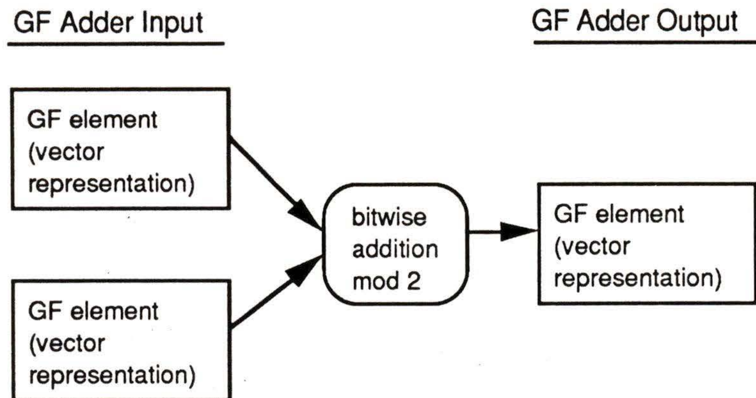


Figure 3.1.1: Normal basis implementation of GF addition.

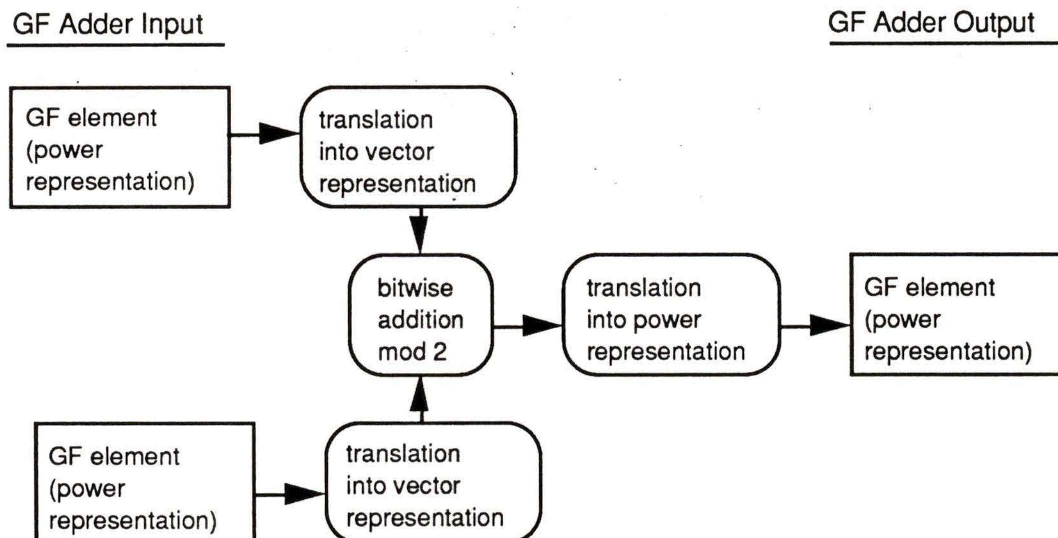


Figure 3.1.2: GF addition of power representation elements.

The normal basis representation implementation for addition over GF(2^5) requires 5 XOR gates. The power representation for addition over GF(2^5) requires 165 AND gates and 140 XOR gates. On the Xilinx XC3000 Series a vector representation adder requires one configurable logic block for every two bits of the field size, plus input registers, for a total of 8 CLB's for a GF(2^5) adder. The easiest implementation of power representation addition over GF(2^5) in the Xilinx XC3000 Series is to translate from power to vector

representation, add modulo 2, and translate back to the power representation. The five function logic of the Xilinx XC3000 Series allows the translation to be performed in the input and output registers; the power representation adder requires 18 CLB's in total.

3.1.2 Galois Field Multipliers

Choosing the normal basis for the vector representation allows the parallel Massey-Omura multiplier to be used for the Galois field multiplication. The parallel Massey-Omura multiplier may be implemented in the Xilinx XC3000 Series as shown in Figure 3.1.3.

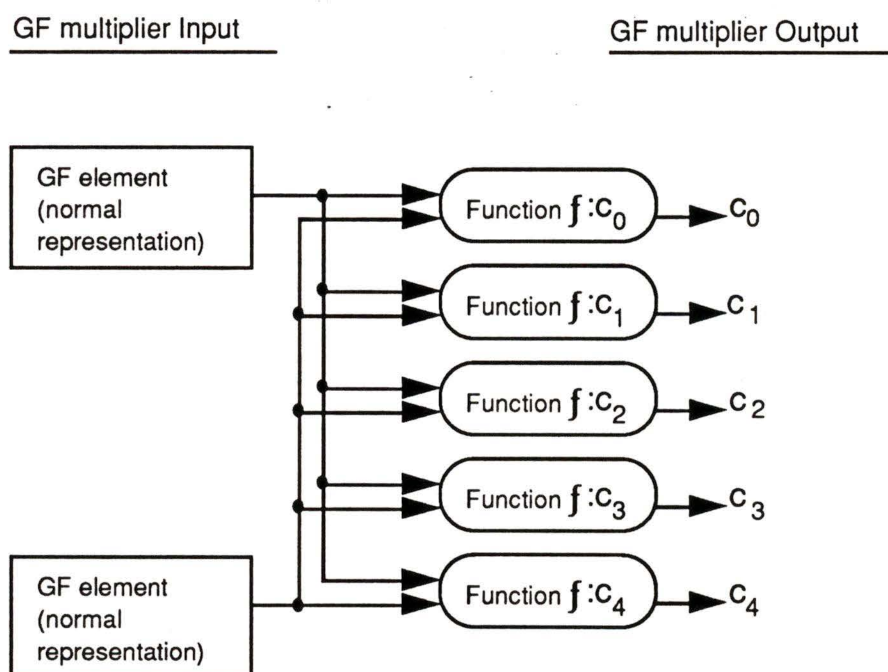


Figure 3.1.3: Parallel Massey-Omura multiplier.

In the power representation, Galois field multiplication over the field $GF(2^m)$ becomes integer addition modulo 2^m-1. Since the greatest possible result of integer addition of two numbers a, $b < 2^m - 1$ is $a + b \leq 2^{m+1} - 4 < 2^{m+1} - 2$, the modulo operation need only be

carried out once.

The the hardware required to implement the Galois field multiplier by doing the sum c = a + b and incrementing c if necessary is about 40% less than hardware required to implement the Galois field multiplier by doing the sums c = a + b and c = 1 + a + b in parallel . However, the former method requires two steps, whereas the latter method requires only one step, so only the latter method is used in this design. The Galois field multiplier is made slightly more complex by a need to be able to recognize the integer 31 as representing the $GF(2^5)$ zero element. An extra circuit must be introduced that sets the GF multiplier output to 31 if either multiplicand equals 31. However, in the case of the (31,k) RS Codec, this extra circuit is not required. The GF multiplier may be implemented as shown in Figure 3.1.4.

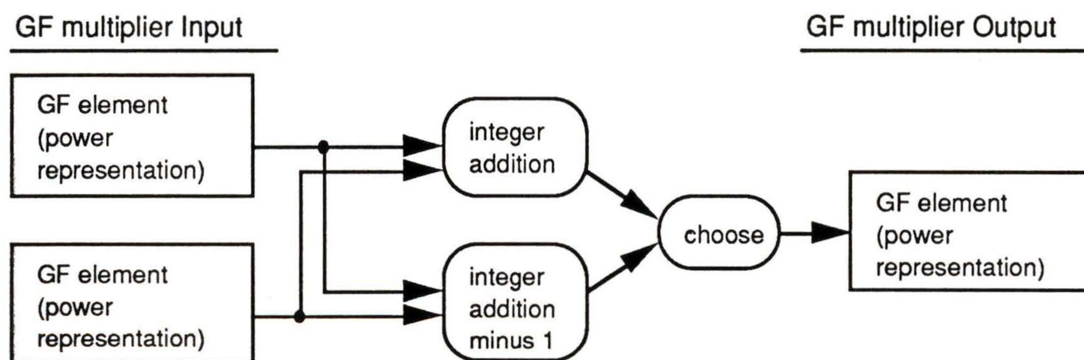


Figure 3.1.4: Power representation implementation of GF multiplication.

The least complex normal basis implementation for multiplication over $GF(2^5)$ requires 45 AND gates and 40 XOR gates [25]. A simple implementation of the integer addition modulo 2^m-1 used for the power basis implementation for multiplication over $GF(2^5)$ uses a five

bit ripple counter; this can be implemented using 23 AND gates and 35 XOR gates. At high clock rates, the delay caused by the ripple adder may at high clock rates cause timing problems for large GF sizes; the delay may be avoided by using a look ahead adder. A three stage look-ahead adder may be implemented using 35 AND gates and 47 XOR gates. On the Xilinx XC3000 Series LCA's the least-complex parallel Massey-Omura multiplier for GF(2^5) requires 20 CLB's, plus 5 CLB's for the input registers. For the power representation GF(2^5) multiplier 10 CLB's plus 5 CLB's for the input registers are needed.

3.1.3 Galois Field Inverters

The parallel normal basis Galois field inverter may be constructed from the parallel Massey-Omura multiplier and the pipeline normal basis Galois field inverter. Since squaring an element in a normal basis is equivalent to a cyclic shift of the element [24], the parallel normal basis Galois field inverter may be constructed as shown in Figure 3.1.5.

The four parallel Massey-Omura multipliers required by the parallel normal basis Galois field inverters make it too large to be practical in the (31,k) RS decoder; as well, the inverter would have to be run at a slower clock speed to allow the signal to propagate through all the levels of logic in one clock cycle. Alternatively, the pipelined normal basis Galois field inverter [24] could also be used, at the cost of extra steps each time the inverter is used.

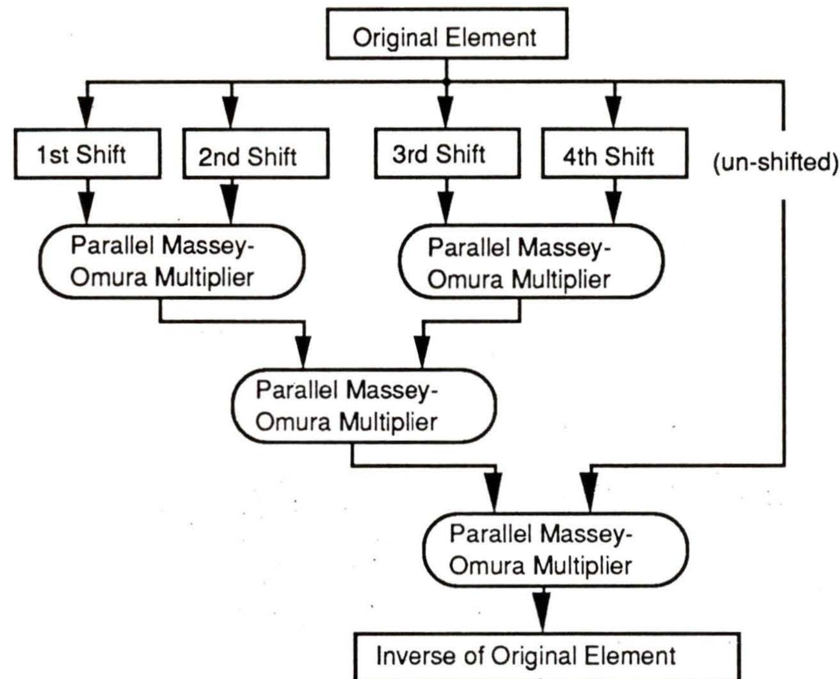


Figure 3.1.5: Normal basis implementation of $GF(2^5)$ inversion.

In the case of small field sizes, such as the field $GF(2^5)$ used in the (31, k) RS codec, the most hardware efficient means of inverting a normal basis field element in one step is to directly translate from the element α^i to its inverse $(\alpha^i)^{-1} = \alpha^j$. The logic for this is similar to the logic used in the power representation implementation of the GF adder.

The normal basis implementation of the parallel $GF(2^5)$ inverter requires 180 AND gates and 160 XOR gates, while the direct translation implementation for the normal basis inverter requires 55 AND gates and 45 XOR gates. The power representation implementation of the $GF(2^5)$ inverter requires 5 XOR gates.

In the Xilinx XC3000 Series implementation of the $GF(2^5)$ inverter, the simplest procedure for inverting the normal basis representation is to map each element to its inverse. The mapping over $GF(2^5)$

requires 8 CLB's. The power representation implementation of the $GF(2^5)$ inverter requires 4 CLB's.

3.1.4 Table Look-up of GF Elements

It is also possible to use table look-up to implement the GF arithmetic operations requiring the most hardware; this is the GF adder in the case of the power representation, and the GF multiplier and inverter in the case of the normal basis representation. For GF addition in the power representation, the vector representations are stored in ROM. The GF adder sends the power representation to the ROM as an address, and reads the vector representation as the data. The vector representations are then added modulo-2, and the result of the modulo-2 addition is sent as an address to a second ROM. The output of this second ROM is the result of the GF addition.

The GF multiplication in the normal basis representation would proceed similarly. The normal basis representations of the GF elements to be multiplied are sent as addresses to a ROM, which returns their anti-logs (power representation). The anti-logs are added as integers modulo-31, and the result of the addition modulo-31 is sent as an address to a second ROM. The output of the second ROM is the result of the GF multiplication.

GF inversion is implemented in the normal basis representation by sending the element to be inverted as an address to the ROM, which sends back the inverse as its data.

The advantage of using table look-up is its simplicity. The disadvantages of table look-up are the need for extra ROM chips, the

slow speed of ROM, and the need for an extra step for each access of external ROM.

3.1.5 GF Arithmetic Summary

The Galois field arithmetic operations needed by the RS decoding algorithm are addition, multiplication and inversion. GF addition and GF multiplication are needed in every logical module of the RS codec. The number of basic logic gates and CLB's needed for the most space efficient parallel implementations of GF addition, multiplication and inversion are given in Table 3.1.7 and Table 3.1.8, respectively.

Representation	GF Adder		GF Multiplier		GF Inverter	
Gates	AND	XOR	AND	XOR	AND	XOR
Normal	0	5	45	40	55	45
Power	165	140	23	35	9	11

Table 3.1.7: AND and XOR gate requirements for normal basis and power representations of parallel GF arithmetic units.

Representation	GF Adder	GF Mult	GF Inv
Normal	8	25	8
Power	18	15	4

Table 3.1.8: CLB requirements for normal basis and power representation of parallel GF arithmetic units.

It can be seen from Table 3.1.8 that when using the Xilinx XC3000 Series, the normal basis representation and the power representation both require 33 CLB's for implementation of GF addition and GF multiplication. It can also be seen from Table 3.1.4 that the power representation requires 4 less CLB's than does the normal basis representation for implementation of GF inversion. This means the power representation requires about 10% fewer CLB's to implement

the three GF arithmetic operations required by the RS decoding algorithm than does the normal basis representation. The three GF arithmetic operations are required for the non-pipelined RS decoder, and for the delta, location, and correction stages in the case of the pipelined RS decoders.

For a non-Xilinx implementation, the normal basis representation implementation of the three $GF(2^5)$ arithmetic operations requires 190 gates, about half the 383 gates required by the power representation implementation. The main use of gates in the power representation implementation of $GF(2^5)$ arithmetic is in the $GF(2^5)$ adder, which requires 305 of the 383 gates. Implementation of $GF(2^5)$ multiplication and $GF(2^5)$ inversion requires 78 gates in the power representation, and 180 gates in the normal basis representation. Thus if table look-up were to be used for one of the three $GF(2^5)$ arithmetic operations, the power representation gives an efficient implementation of the remaining $GF(2^5)$ arithmetic operations.

For larger field sized, $GF(2^m)$ where $m > 5$, the number of gates required for translation between power and normal basis representations rises sharply (see Chapter 2.1.2), and either table look-up or a multi-step algorithm would have to be used for power representation implementation of GF addition. The number of gates for power representation implementation of GF multiplication and GF inversion rises linearly. Thus, if it is desirable to limit the number of steps in each GF arithmetic operation to two or less, a power representation implementation of $GF(2^m)$ arithmetic, including a table look-up GF addition unit, should require less space than a normal basis

representation implementation. Note that this may not be true for a "pure" normal basis implementation, in which no translation between representations occurs.

3.2 Pipelining and Array Passing Options

Conventional designs for Reed-Solomon decoders use only one of the five decoder stages at a time. It is possible, however, to pipeline the RS decoder; that is, to have the RS decoder operate on more than one received word at a time. The coordination of the stages and the shared intermediate calculations becomes more complex and thus requires more hardware, as more stages of pipelining are used. Each of the five stages requires as its input either the received code word or an array of intermediate values calculated in previous stages as is shown in Figure 3.2.1.

The received code word and the syndrome are arrays of 31 $GF(2^5)$ elements (155 bits); the other arrays are 16 $GF(2^5)$ elements (80 bits) in size. If no pipelining is used, these values can be passed on by using the same memory for each stage. Because the RS decoding algorithm requires both the received coded word and each intermediate value array to be completed before being passed on, each level of pipelining introduced increases both the number of arrays that must be passed on, and the number of copies of each passed array that must be stored.

Two strategies for passing arrays between stages have been considered. The first strategy passes the data between the storage elements associated with each stage (direct passing), the second uses external RAM to store the intermediate values (RAM passing).

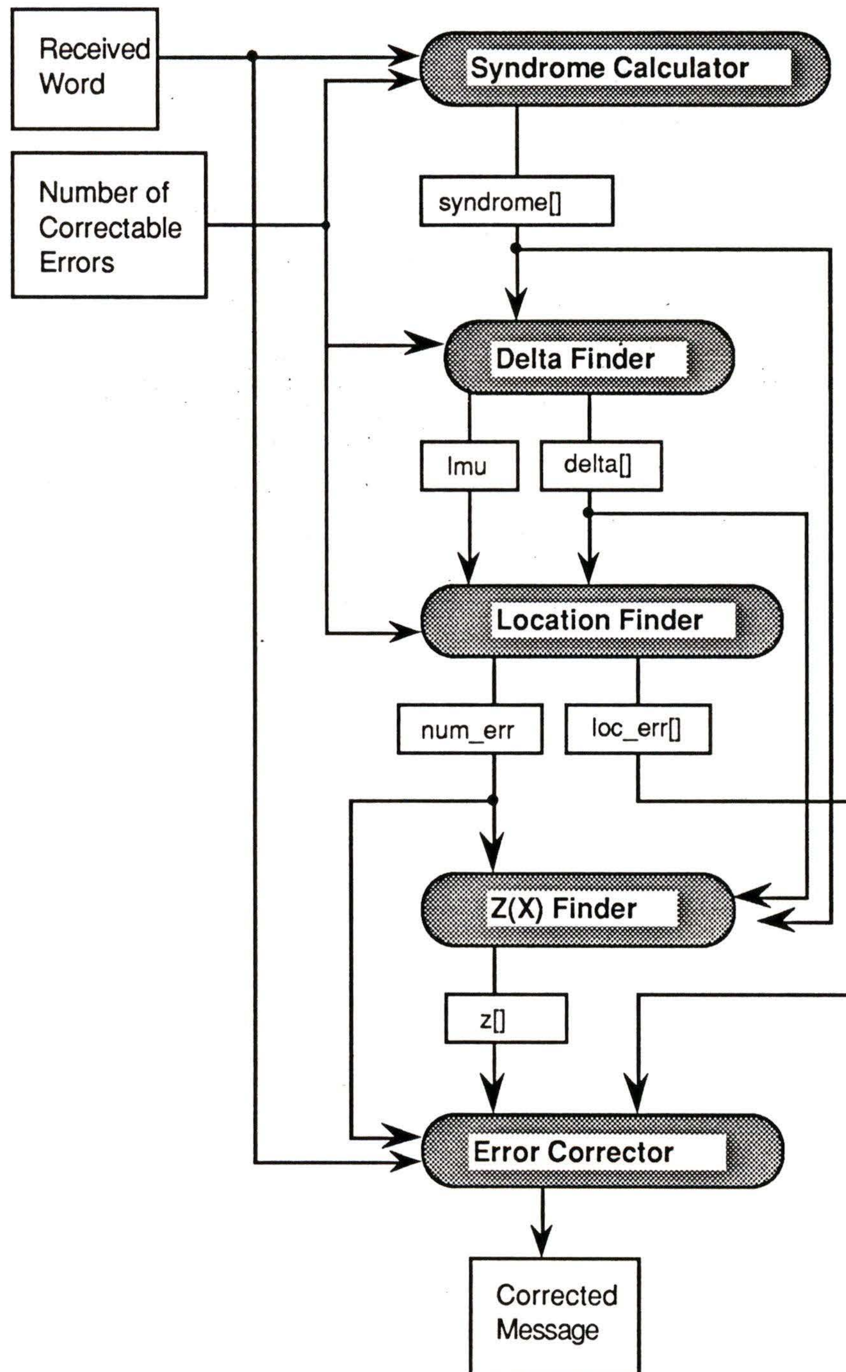


Figure 3.2.1 Flow of passed intermediate results between modules.

3.2.1 Direct Passing of Intermediate Values

Consider the case of five stage pipelining as an example. If some array A[X,t] is calculated in the first stage and used in the fourth stage, the copy of A[X,t] generated at time t = 0 will reach the fourth stage three stage shifts later. This means that when the array A[X,t =0] will be used in the fourth stage there will be an array A[X, t = 3] being calculated in the first stage and two arrays, A[X, t = 1] and A[X, t = 2] being stored for use in the fourth stage (Figure 3.2.2).

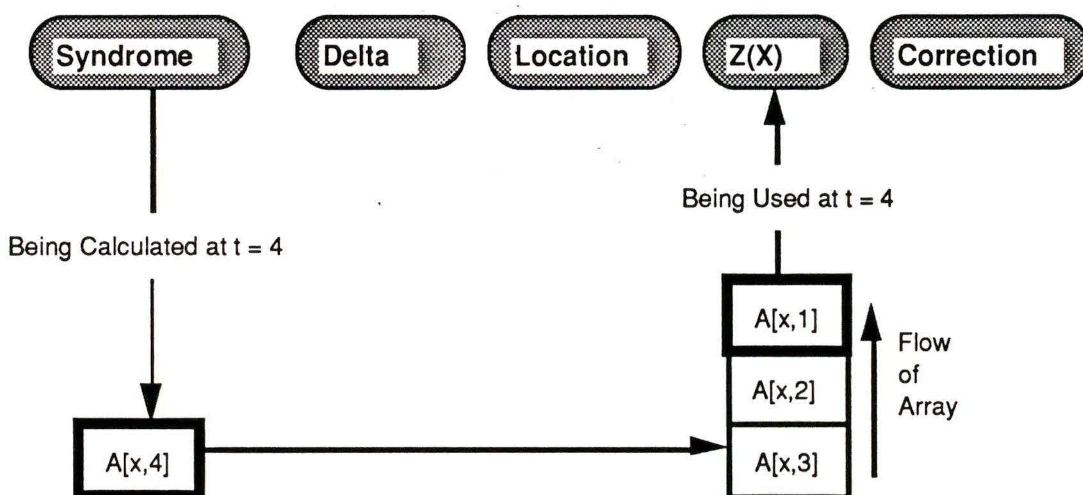


Figure 3.2.2: Direct Data Passing Between Stages

Since passing an array of 75 or 155 bits in one clock cycle requires too many input/output pins to be practical, the modules must shift the array in one stage before the array is to be used, to ensure the complete array will be available when required. In some cases, this requires the module to have two copies of the array, the copy being used A[X, t = 0] and the copy being shifted in A[X, t = 1]. Since only the arrays needed by a particular stage are passed to that stage, this strategy minimizes the amount of storage needed. Circular buffers

may be used for the arrays stored at a given stage. The direct passing strategy has the disadvantage of requiring relatively complex logic to control the passing process between stages, which reduces the modularity of the design and makes design alterations more difficult. The direct passing strategy requires either shift registers or PGA's with a very large number of input/output pins to store and pass the arrays between modules; either option requires a large number of CLB's in the Xilinx XC3000 Series, or a large number of chips if implemented in standard register chips. The flow of the arrays in the case of five pipelining options is shown in Figure 3.2.3.1 to Figure 3.2.3.5.

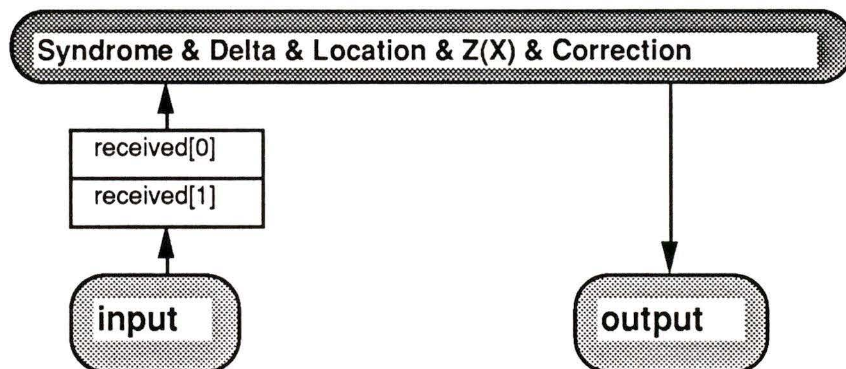


Figure 3.2.3.1: Direct passing of intermediate arrays for the no-pipelining option.

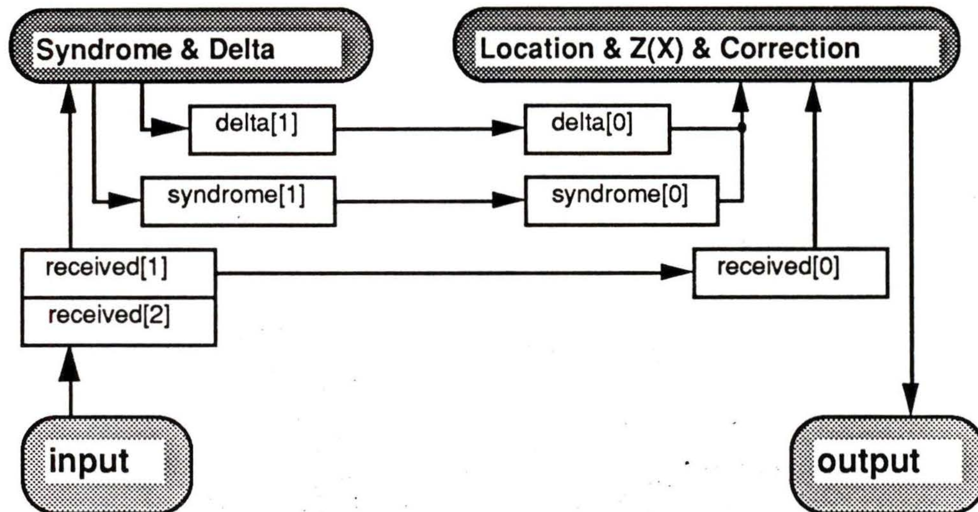


Figure 3.2.3.2: Direct passing of intermediate arrays for the two stage pipelining option.

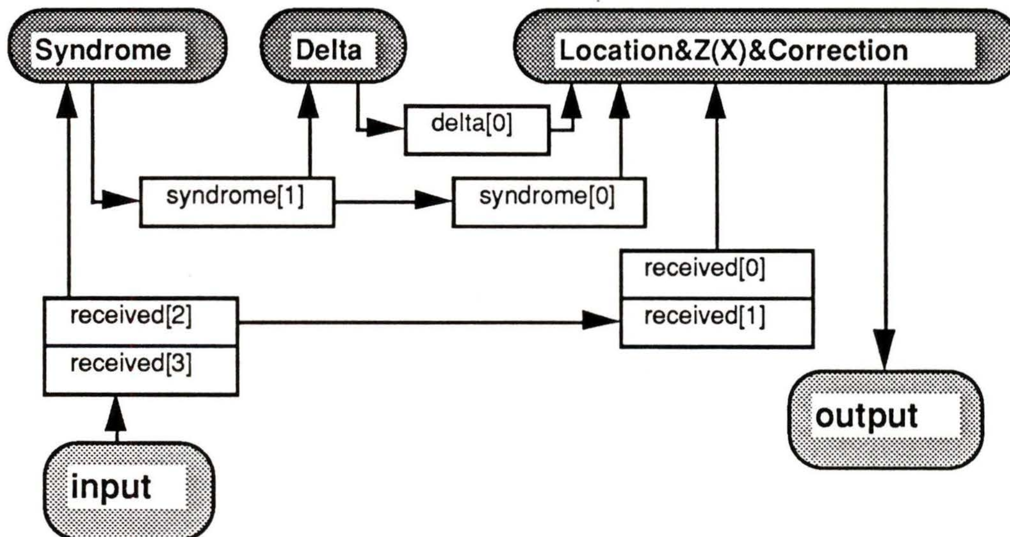


Figure 3.2.3.3: Direct passing of intermediate arrays for the three stage pipelining option.

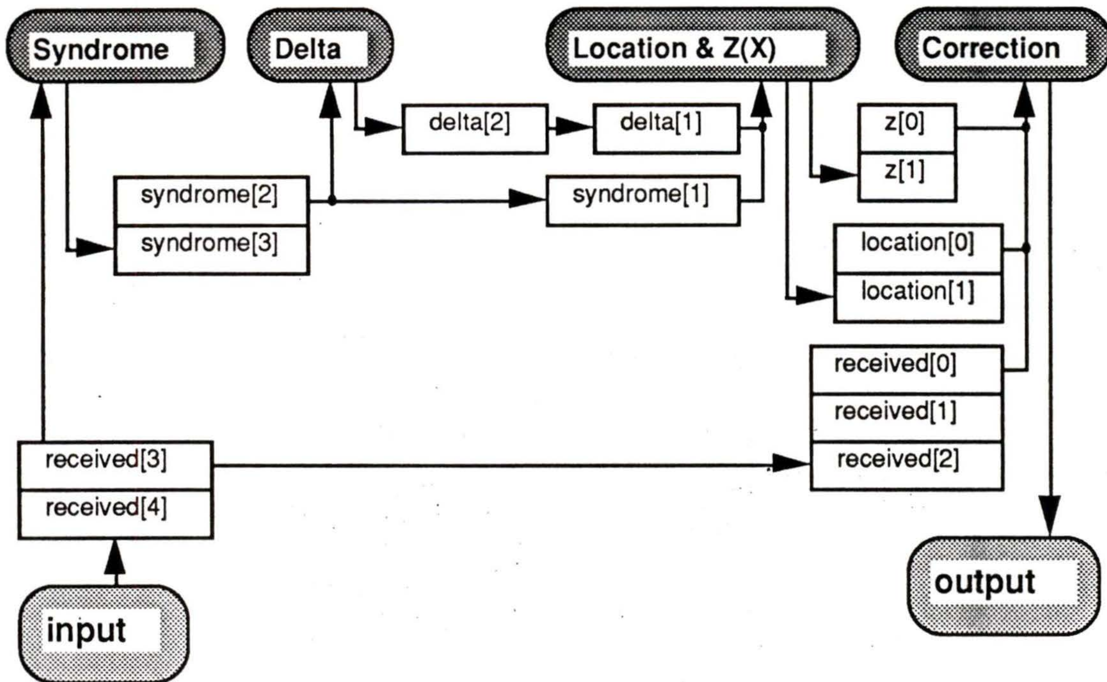


Figure 3.2.3.4: Direct passing of intermediate arrays for the four stage pipelining option.

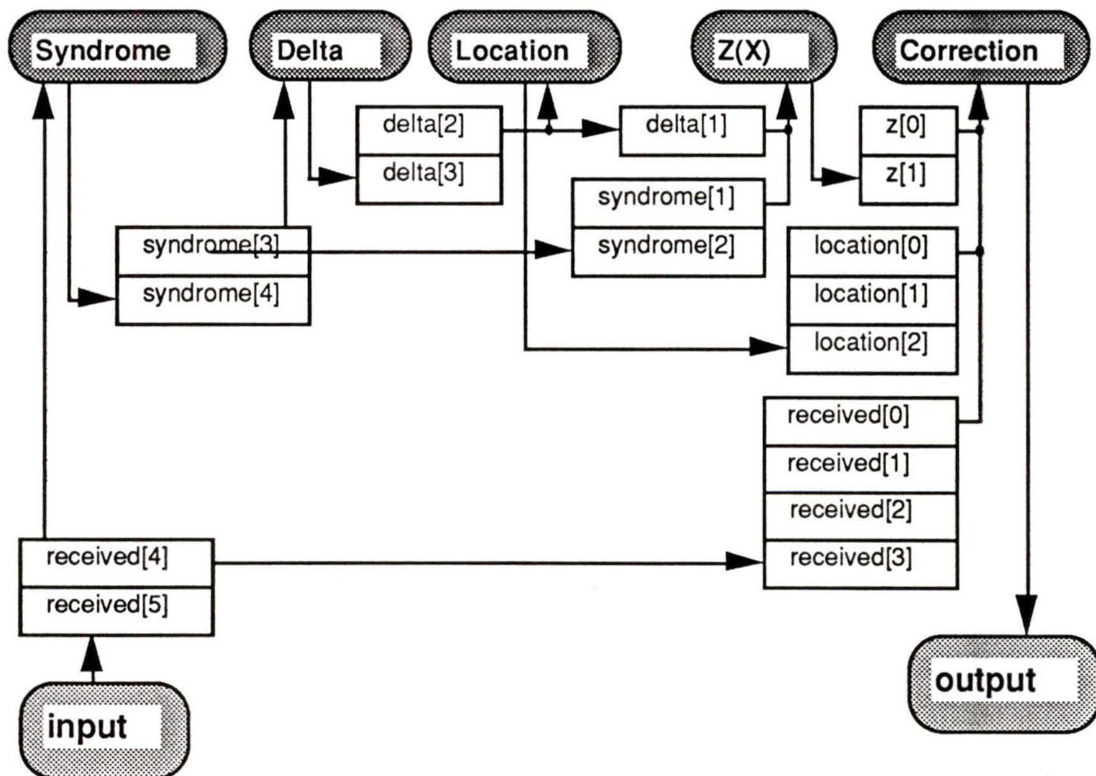


Figure 3.2.3.2: Direct passing of intermediate arrays for the five stage pipelining option.

3.2.3 RAM Storage of Intermediate Values

The second strategy is to pass a pointer to the data associated with each stage. Since one copy of each array used in the decoding algorithm must exist for each stage, this strategy requires more memory space than the first option. However, it maintains the modularity of the design and keeps the control logic simple. The arrays may also be stored in relatively cheap external RAM; because the sequence of steps in each module varies with the number of errors to be corrected, it is not possible to time RAM access among the modules because of memory contention. Instead, each module has its own RAM; this also allows the same memory design to be used with little change for Galois fields of size 2^8 and smaller. The modules are connected to the RAM through a switching network as shown in Figure 3.2.4; a counter is used to keep track of which module is connected to each RAM chip. Each of n RAM's would have a switching circuit address of 0 to n-1, and each decoder stage would choose a RAM based on a 0 to n-1 counter. As the counter cycled through 0 to n-1, each stage would cycle through the RAM addresses in sequence. The details of the switching circuit are discussed in Chapter 4.

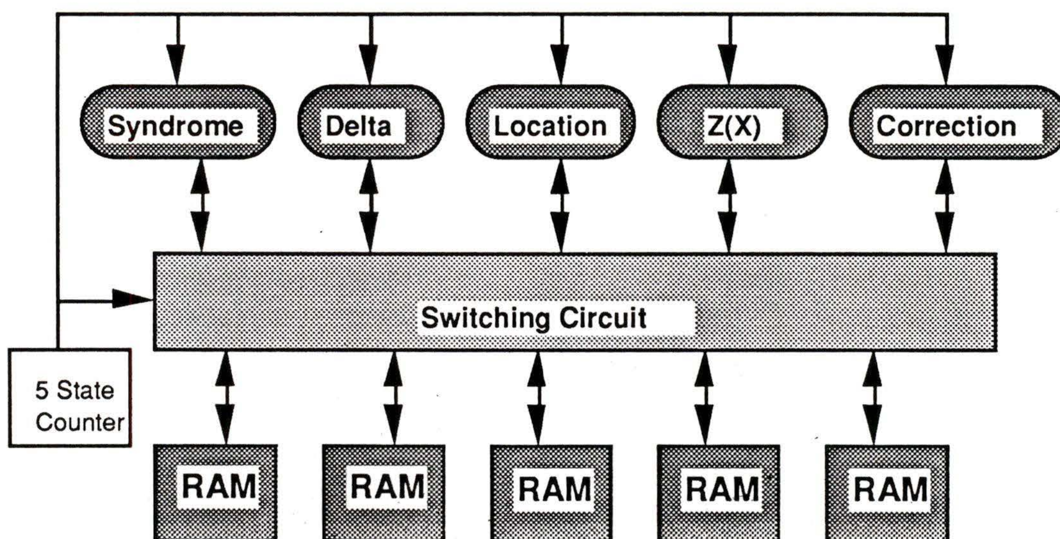


Figure 3.2.4: RAM based array passing for the five stage pipelining option.

3.2.4 Changing the Number of Correctable Errors

The use of external RAM to store variables allows the number of correctable errors t_e to be stored along with the arrays for each received word. Thus each received word can have a different number of correctable errors. Because the number of operations required to decode a received word varies with the number of correctable errors for a given pipelining option, the encoder will send data at the rate required by the slowest of the received words stored in the decoder. Henceforth we assume that the changes in the number of correctable errors are synchronized between the encoder and decoder. This design offers no solution for this synchronization, but allows for changes on the fly if the synchronization is achieved.

3.3 Pipelining Options

3.3.1 Number of Operations per Module

The independence of the five modules in the RS decoding

algorithm allows implementation of a pipelined RS decoder. The number of operations, the number of CLB's and the amount of memory for intermediate results varies greatly among modules. Therefore, a number of pipelining options must be considered in light of the trade-off between the decoder speed and the amount of hardware needed for implementation. The number of operations executed by each module, as a function of the number of correctable errors, is given in Figure 3.3.1 for direct passing of arrays, and in Figure 3.3.2 for RAM passing of arrays.

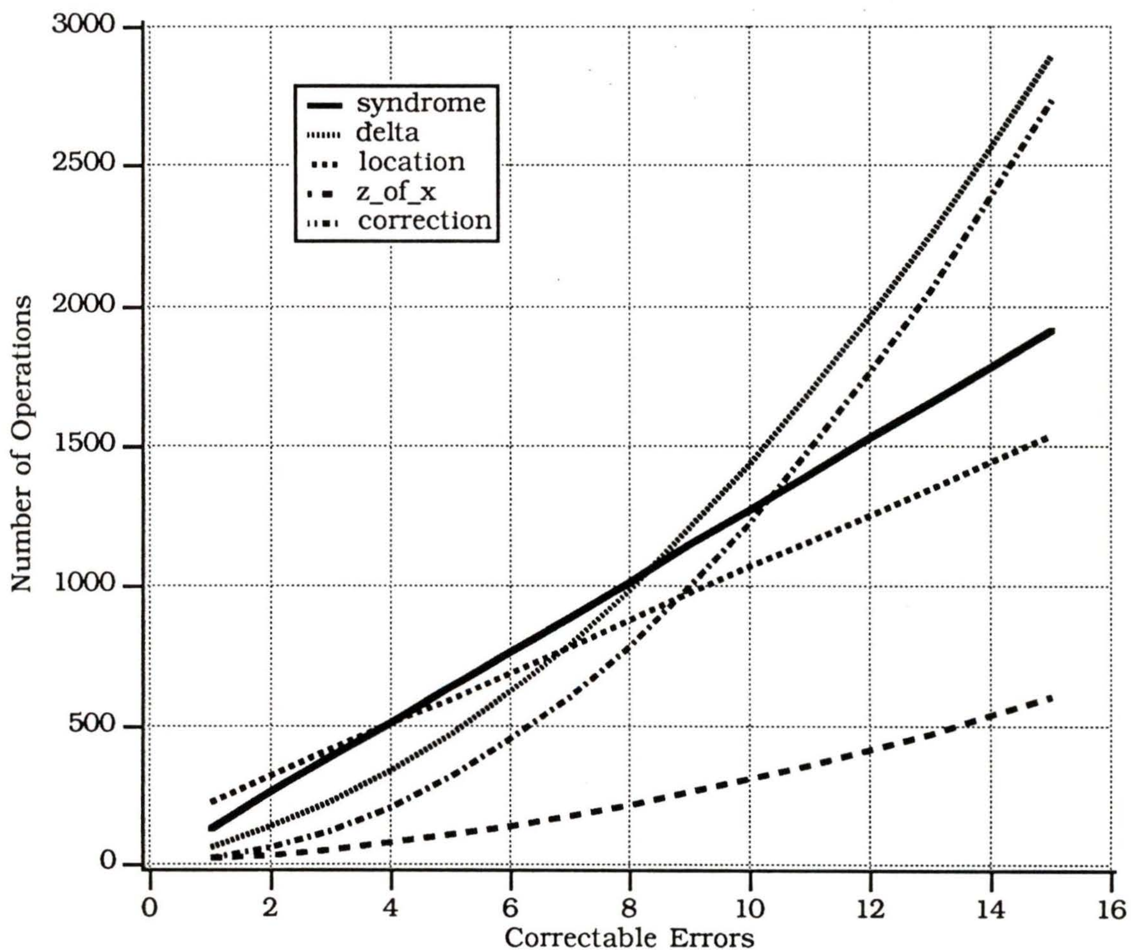


Figure 3.3.1: Number of operations vs. correctable errors per decoder module for direct passing of arrays.

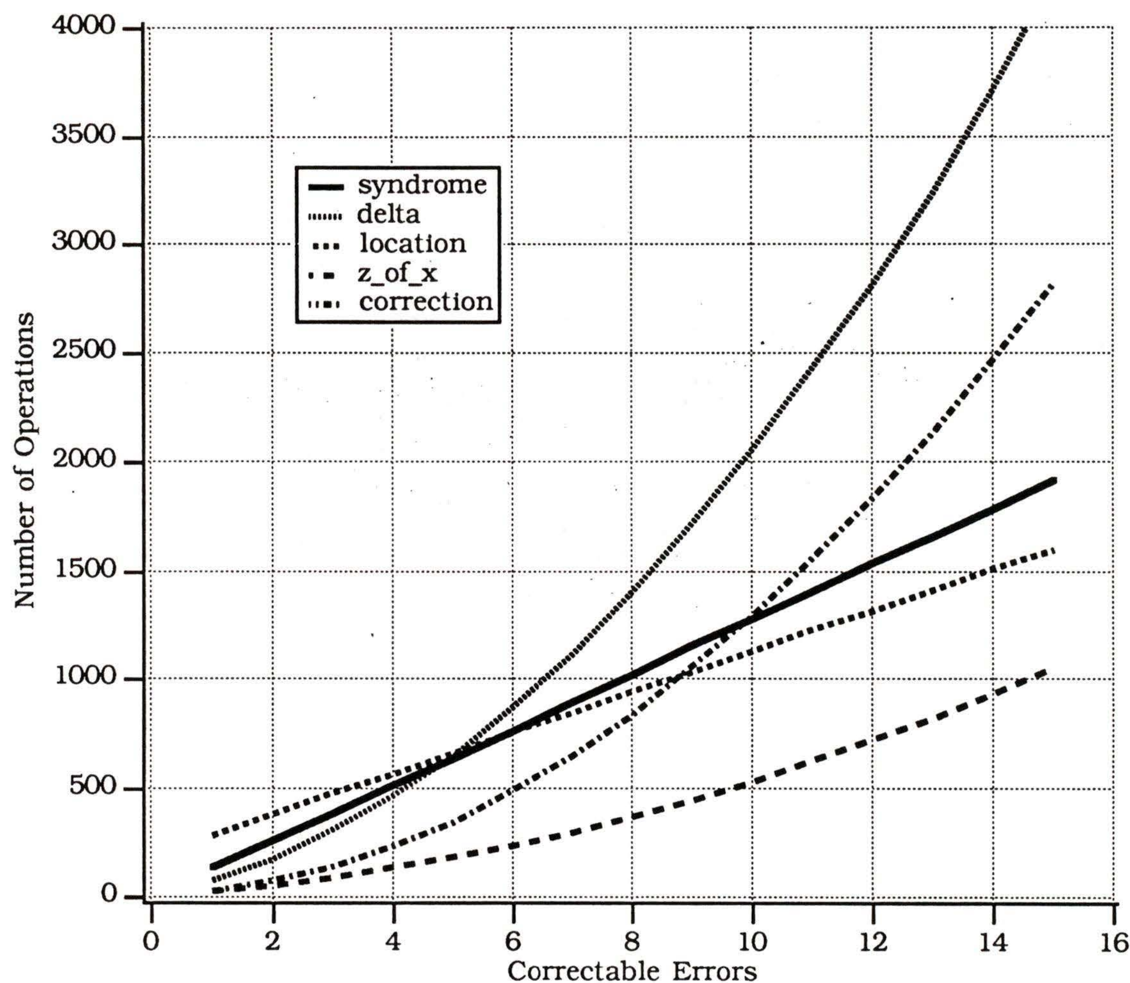


Figure 3.3.2: Number of operations vs correctable errors per decoder module for RAM passing of arrays.

The number of steps required by the RAM passing architecture, as a percent of the number of steps required by the direct passing architecture, per decoder module, is given in Table 3.3.1. From Table 3.3.1, it is seen that the RAM and direct passing modules speeds are similar for the syndrome, location and correction stages, but differ by 30% and 40% for the delta and Z(X) stages.

correctable errors	syndrome %RP/DP	delta %RP/DP	location %RP/DP	Z(X) %RP/DP	correction %RP/DP
1	97	72	77	57	72
2	98	73	83	57	81
3	99	72	86	57	86
4	99	72	88	57	89
5	99	71	90	57	90
6	99	71	91	57	92
7	99	70	92	57	93
8	99	70	93	57	94
9	99	70	93	57	94
10	99	69	94	57	95
11	99	69	94	57	95
12	99	69	95	57	96
13	99	69	95	57	96
14	99	69	95	57	96
15	99	69	96	57	96

Table 3.3.1: Number of direct passing steps as a percent of RAM passing steps per decoder module.

3.3.2 Number of Operations per Passing Combination

Not only does each module require a different number of operations to complete, but the number of operations to complete varies among modules either linearly or as the square of the number of correctable errors. Thus the total number of steps required by a pipelining option depends upon how the decoder stages are combined. There is one combination giving no pipelining or five stage pipelining of five modules, four combinations giving two stage or four stage pipelining of five modules, and six combinations giving three stage pipelining of five modules (see Table 3.3.2).

Pipelining Stages	Combinations of Five Modules : syndrome = s, delta = d, location = l, z(X) = z, correction = c
No Pipelining	sdlzc
Two	s-dlzec sd-lzc, sdl-zc sdlz-c
Three	s-d-lzc s-dl-zc s-dlz-c sd-l-zc sd-lz-c sdl-z-c
Four	sd-l-z-c s-dl-z-c s-d-lz-c s-d-l-zc
Five	s-d-l-z-c

Table 3.3.2: Possible pipelining combinations of five modules.

The maximum number of steps needed to complete each pipelining option, using the programmable (31,k) RS algorithm as modified for the Xilinx implementation, are given in Figures 3.3.3 to Figure 3.3.8. Note that since only one combination exists for the no pipelining and five stage pipelining options, the executable steps to completion for these are only given in Figure 3.3.9 and Figure 3.3.10 which give the results of the best combinations for each pipelining option.

Direct Passing of Arrays

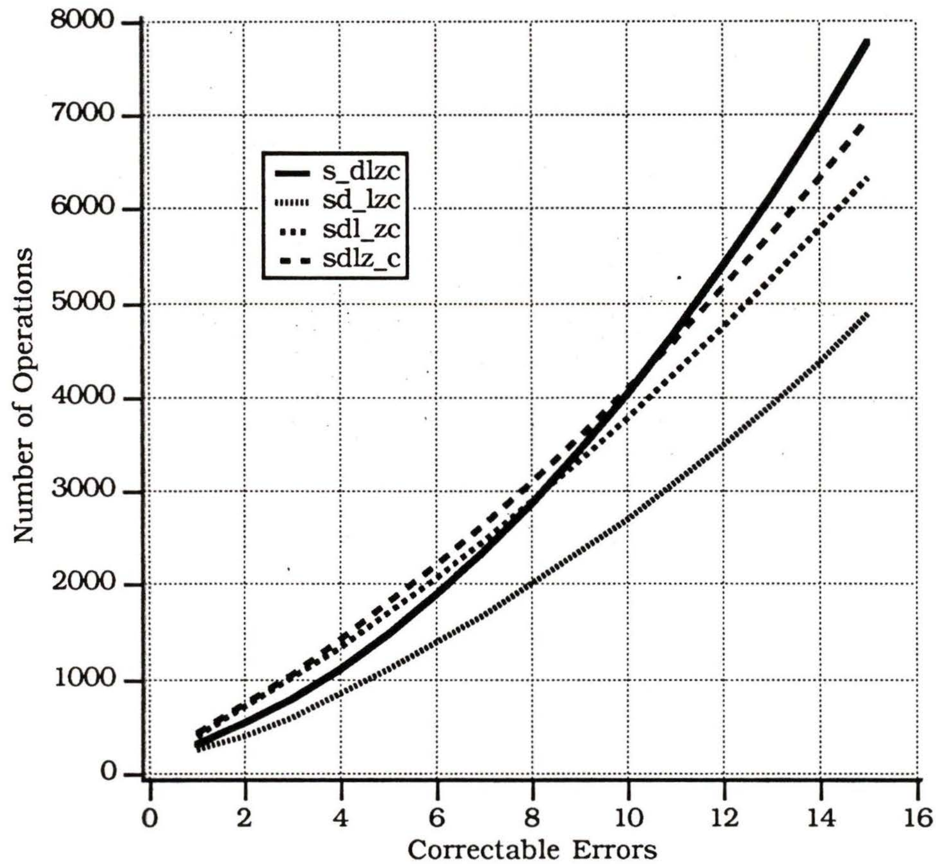


Figure 3.3.3: Number of steps needed to complete possible two stage pipelining option combinations, per number of correctable errors, for direct passing of arrays.

From Figure 3.3.3, it is seen that the best two stage pipelining option for direct passing of arrays is sd-lze, which is about 15% faster than the fastest of the remaining options

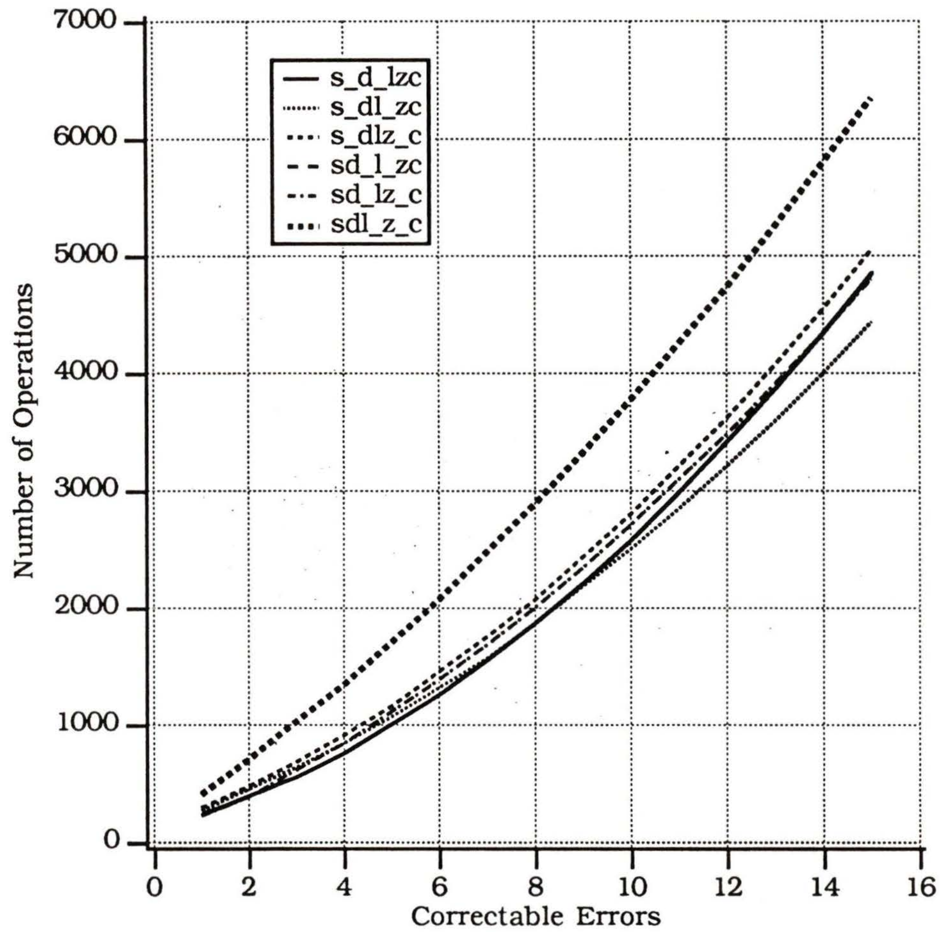


Figure 3.3.4: Number of steps needed to complete possible three stage pipelining option combinations, per number of correctable errors, for direct passing of arrays.

From Figure 3.3.4, it is seen that none of the direct passing three stage pipelining options is faster than the other options for all values of correctable errors. The option sd-l-zc is the fastest for high rate codes (t_e = 1,2), the option s-d-lzc is the fastest for the mid-range of correctable errors (t_e = 3 to 7), and the option s-dl-zc is the fastest for low rate codes (t_e = 8 to 15).

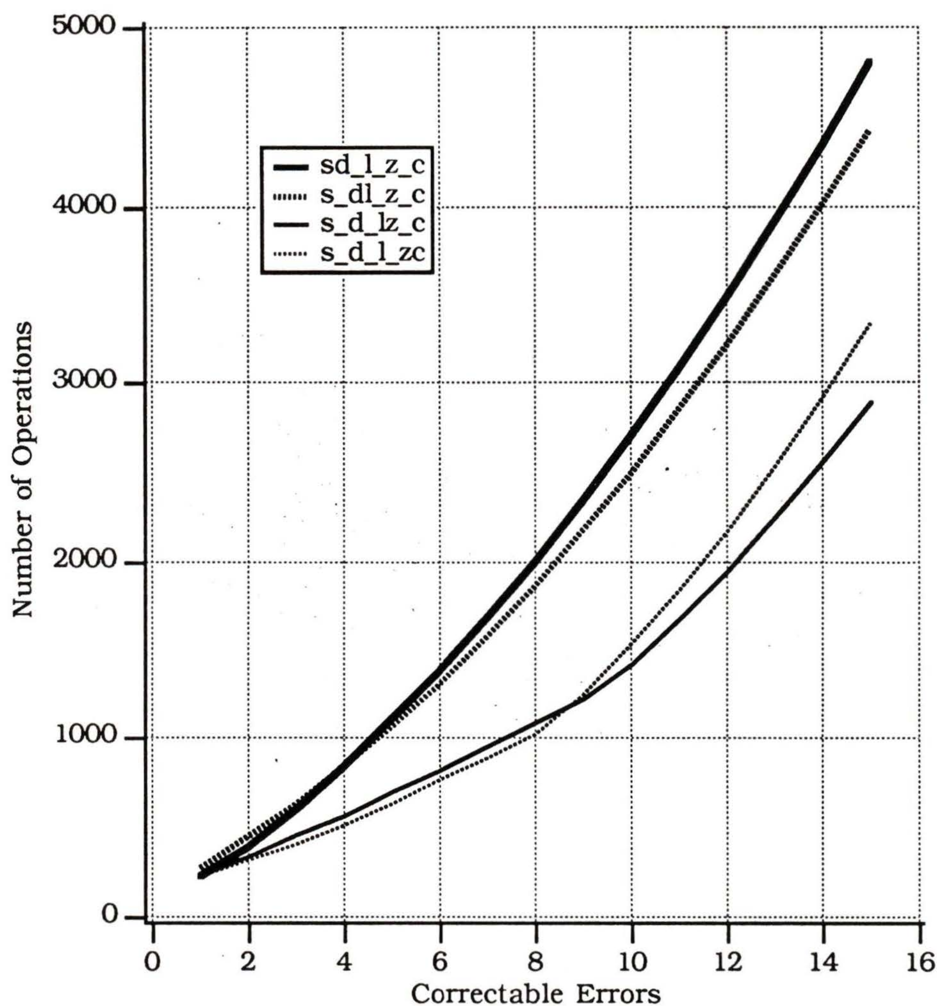


Figure 3.3.5: Number of steps needed to complete possible four stage pipelining option combinations, per number of correctable errors, for direct passing of arrays.

From Figure 3.3.5, it is seen that none of the direct passing four stage pipelining options is faster than the other options for all values of correctable errors. The option s-d-l-zc is the fastest for high rate codes (t_e = 1 to 8), and the option s-d-lz-c is the fastest for low rate codes (t_e = 9 to 15).

RAM Passing of Arrays

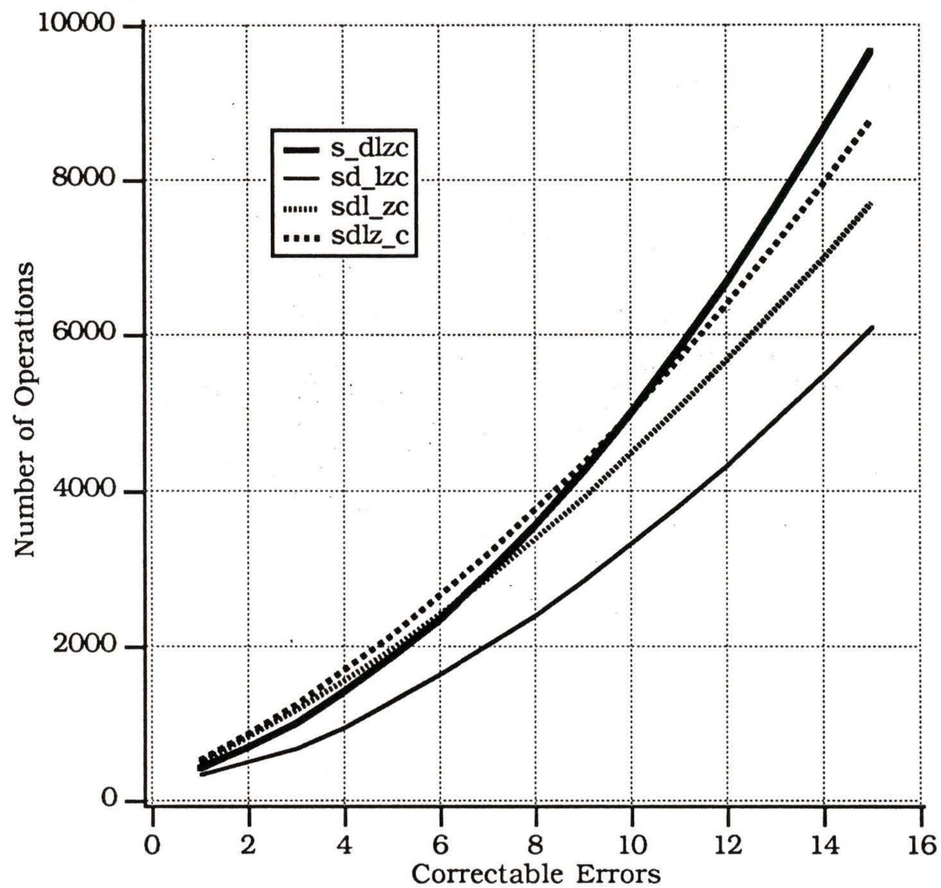


Figure 3.3.6: Number of steps needed to complete possible two stage pipelining option combinations, per number of correctable errors, for RAM passing of arrays.

From Figure 3.3.6, it is seen that the best two stage pipelining option for RAM passing of arrays is sd-lze, which is about 15% faster than the fastest of the remaining options.

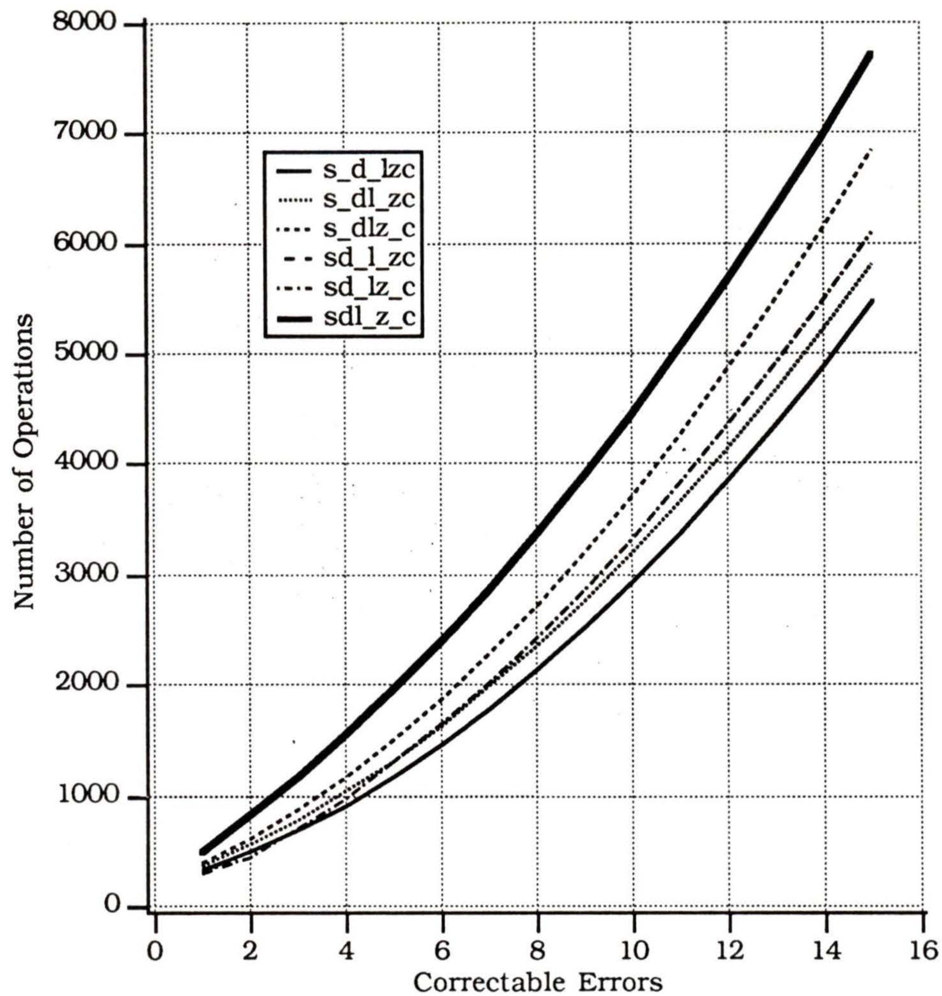


Figure 3.3.7: Number of steps needed to complete possible three stage pipelining option combinations, per number of correctable errors, for RAM passing of arrays.

From Figure 3.3.7, it is seen that none of the RAM passing three stage pipelining options is faster than the other options for all values of correctable errors. The option sd-l-zc is the fastest for high rate codes (t_e = 1 to 3), and the option s-d-lzc is the fastest for mid and low rate codes (t_e = 4 to 15).

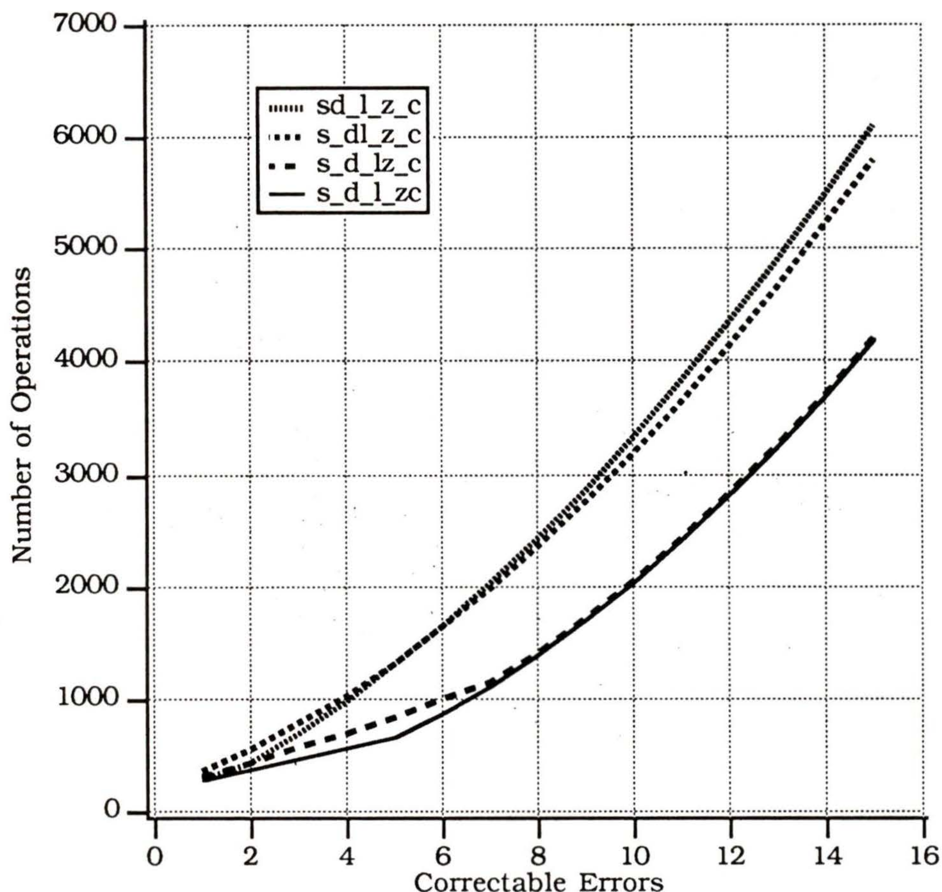


Figure 3.3.8: Number of steps needed to complete possible four stage pipelining option combinations, per number of correctable errors, for RAM passing of arrays.

From Figure 3.3.8, it is seen that the best four stage pipelining option for RAM passing of arrays is s-d-l-zc, although s-d-lz-c has the same speed for low rate codes (t_e = 8 to 15).

3.3.3 Hardware Requirements of Passing Options

The number of CLB's used by each array passing option must also be examined, to ensure that the fastest options do not take significantly more space than slightly slower options. For RAM passing of arrays, the number of CLB's does not vary among options of the same number

of stages. In the case of the direct array passing, hardware is minimized if modules which share intermediate calculations are combined. The amount of intermediate array memory in bits needed by each possible pipelining combination is given in Table 3.3.3.

Pipelining Combination	Memory (CLB's)
sdlzc	1360
s-dlzc	1670
sdlz-c	1700
sd-lzc	1735
sdl-zc	1835
s-dlz-c	1985
s-dl-zc	2040
s-d-lzc	2060
sd-lz-c	2065
sdl-z-c	2140
sd-l-ze	2220
sd-l-z-c	2530
s-dl-z-c	2530
s-d-lz-c	2600
s-d-l-zc	2675
s-d-l-z-c	2840

Table 3.3.3: Intermediate array hardware requirements for pipelining combinations.

The range in CLB requirements is 10% for the two stage options, 12% for the three stage options, and 6% for the four stage options.

3.3.4 Summary of Passing Options

For the RAM passing options , there are no differences in hardware requirements to consider. The fastest two stage (sd-lzc) pipelining and four stage (s-d-l-zc) pipelining options are the best options. In the case of the three stage pipelining options, sd-l-ze is 15% faster for t_e = 1 and 2, is about as fast as s-d-lzc for t_e = 3, but is about 10% slower for te = 4 to 15 than s-d-lzc. In general, the best three stage

option is then s- d- lzc.

For the direct passing options, the fastest two stage (sd-lzc) option only requires 5% more CLB's than the smallest but slowest two stage option(s-dlzc), and so is the best option. In the case of the three stage pipelining options, there is about 1% difference in size between the three fastest options (s-dl-zc, s-d-lzc and sd-lz-c), all of which are about 5% larger than the smallest, but considerably slower, option s-dlz-c. The option sd-l-zc is the fastest or within 5% of the fastest for the highest rate codes (t_e = 1 to 3), but is slower than the fastest three stage option by 8% for lower rate codes. The option s-dl-zc is the fastest or within 5% of the fastest for all but the high rate codes (for t_e = 5 to 15), but is 15% and 25% slower respectively for the highest rate codes ($t_e = 2$, $t_e = 1$). The best all around three stage pipelining option is probably s-d-lzc, which is 15% slower than the fastest three stage option for $t_e = 1$, is the fastest or within 5% of the fastest for mid-range codes (te = 2 to 11), and is less than 10% slower than the fastest for low rate codes (t_e = 12 to 15).

In the case of the four stage pipelining options, s-d-lz-c is about 3% smaller than s-d-l-zc, and is faster by 10 to 15% for low rate codes (t_e = 11 to 15). However, s-d-l-zc is faster than s-d-lz-c by about 7 to 14% for high rate codes (t_e = 1 to 8), and about the same for te = 9, and so is a better all around code.

The best pipelining options, for both direct and RAM passing, are given in Table 3.3.4 . The overall number of (31, k) RS decoder operations for the five best pipelining options as a function of correctable errors are given in Figure 3.3.9 for direct passing of

arrays, and Figures 3.3.10 for RAM passing of arrays.

Pipeline Option	Logical Modules
no pipelining	• syndrome & delta & location & z(X) & correction
two stage pipelining	• syndrome & delta • location & Z(X) & correction
three stage pipelining	• syndrome • delta • location & Z(X) & correction
four stage pipelining	• syndrome • delta • location • Z(X) & correction
five stage	• syndrome • delta • location • z(X) • error

Table 3.3.4: Pipelining options giving approximately equal numbers of steps per stage.

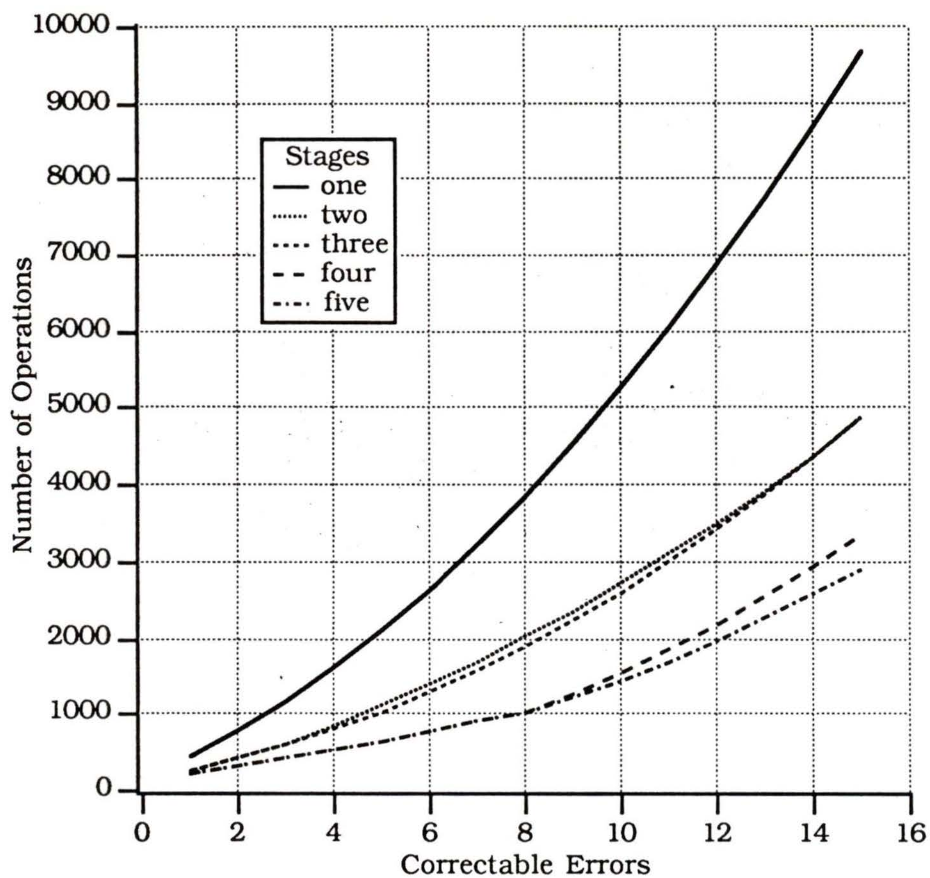


Figure 3.3.9 : Number of operations vs correctable errors per pipelining option for direct passing of arrays.

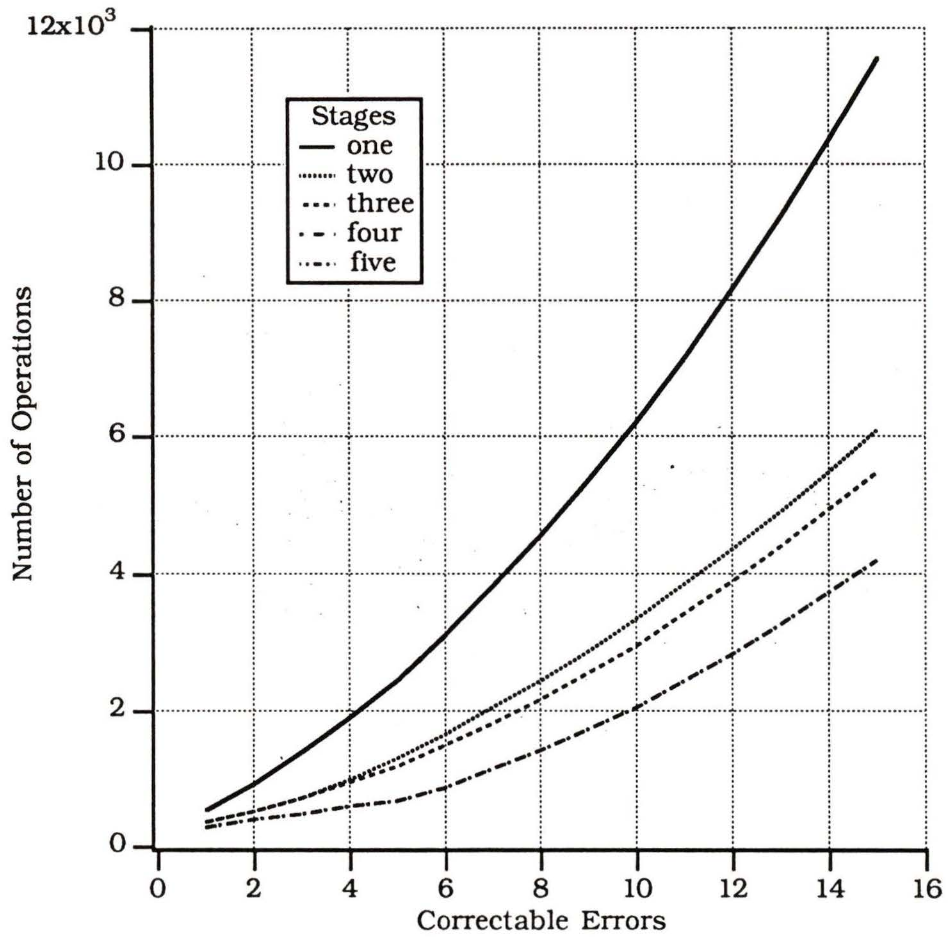


Figure 3.3.10: Number of operations vs correctable errors per pipelining option for RAM passing of arrays.

4 (31,k) RS Codec Designs

Software implementations of the (n,k) RS Codec for n = 15, 31, 63, 127, and 255 were previously written and tested in the C programming language [29], as a first step towards a Logic Cell Array (LCA) design for the (31,k) RS Codec. Using the C implementation as a basis, a (31,k) Reed-Solomon Codec with power representation of the Galois field GF(2^5) elements, five stage pipelining, and external RAM intermediate array storage was designed for the Xilinx XC3000 Series. The Xilinx design for the (31,k) RS Codec was subdivided into three sections: the Encoder, the Decoder modules, and an intermodule communication section.

4.1 Implementation in C

The C source code (see Appendix A) consists of the RS encoder program, the RS decoder program, and the Galois Field arithmetic functions program. The source code supports (n,k) RS codes over any Galois field. The present software implementation allows the user to choose between RS (15,k), RS (31,k), RS (63,k), RS(127,k) and RS (255,k) codecs.

Galois field arithmetic for the field GF(2^m) was implemented using the power representation, with α^0 represented by 0. The power representation allows the Galois field elements to be used as counters in calculating the syndrome components in the first module, simplifying the algorithm. Galois field multiplication is implemented as integer addition modulo 2^m-1, Galois field inversion is implemented by the XOR operation discussed in Chapter 3, and Galois

field addition is implemented using a look-up table. Note that the use of the power representation results in simple implementations of multiplication and inversion and a complex implementation of addition, whereas implementation of a vector representation has a simple implementation of addition and a complex implementation of multiplication and inversion.

The modular approach allows the function reading the received block from the demodulator to operate independently of the rest of the decoder program; it is then a simple task to change the input format. Changes in the algorithm of any of the modules are similarly easy to implement.

4.2 Xilinx Designs

Xilinx Inc. [19] produces lines of Programmable Gate Arrays (PGA's), with user programmable I/O functions, logic and storage functions, and interconnections. In particular, the Xilinx XC3000 Series has

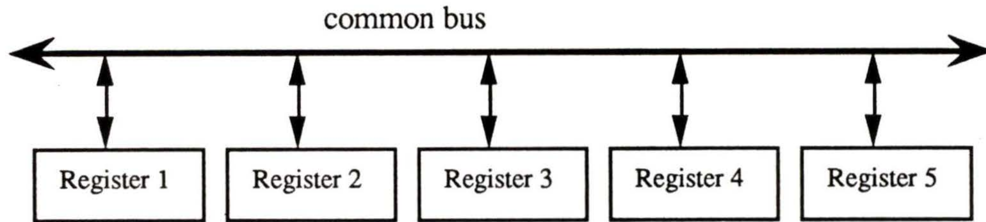
5-input logic functions, 2 flip-flops per CLB and three-state drivers on some interconnection lines. The Xilinx XC3000 Series also comes in three flip-flop toggle rates: 50 MHz, 70 MHz and 100 MHz, and five chip sizes (see Table 4.2.1).

Xilinx Chip	CLB's	I/O Pins
3020	64	64
3030	100	80
3042	144	96
3064	224	120
3090	320	144

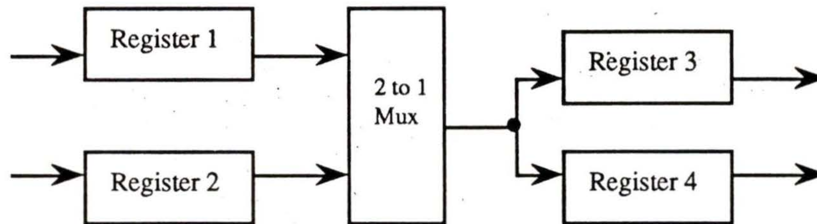
Table 4.2.1: Xilinx chip sizes.

Implementation of logic can be done either in terms of user defined five function logic, or using a schematic editor. The use of a schematic editor avoids the need for detailed design, but does not allow the same degree of customization as does the individual programming of the CLB's and the interconnections between them; an applicable analogy is the difference between programming in a high level programming language and in assembler language. The five function logic is particularly well suited to working in Galois fields $GF(2^m)$, where $m \leq 5$.

The 5-bit buses required for $GF(2^5)$ arithmetic can be implemented either as common buses using Xilinx's tri-state capabilities, or as buses between each set of communicating decoder components or storage block (Figure 4.2.1). The tri-state capabilities of the Xilinx XC3000 Series is quite limited, and it was found that implementing the buses between each set of communicating decoder components uses less chip space.



a) common bus structure.



b) buses between communicating registers.

Figure 4.2.1: Examples of bus hook-ups for XC 3000 series implementations of the RS codec.

4.3 Xilinx Design for the (31,k) RS Encoder

An (n,k) RS Encoder, with k constant, may be implemented as a linear feedback shift register (LFSR) as shown in Fig.2.3.1 [2,20]. However, to support a user selectable value of k, the single generator polynomial g(X) of the fixed k (n,k) RS encoder

$$g(X) = g_0 + g_1X + g_2X^2 + \ldots + g_{2te}X^{2te}$$

must be replaced by the $2^{m-1}-1$ generator polynomials:

$$g_1(X) = g_{01} + g_{11}X + g_{21}X^2$$

$$g_2(X) = g_{02} + g_{12}X + g_{22}X^2 + g_{32}X^3 + g_{42}X^4$$

$$\ldots$$

$$g_{te}(X) = g_{0te} + g_{1te}(X) + g_{2te}(X^2) + \ldots + g_{2tete}X^{2te}, \qquad 1 \le t_e \le 2^{m-1}-1.$$

In the case of the maximum number of correctable errors, $t_{e(max)}$ =

$2^{m-1}-1$, all the coefficients equal α^0. For $t_e < t_{e(max)}$, the $2t_e+1$ coefficients g_{ite} will not be identical. As well, the ith coefficient g_{ite} of the generator polynomial $g_{te}(X)$ will not be identical for different numbers of correctable errors t_e. For an (n,k) RS encoder, (n-3)(n-5)...(3) + 1 coefficients need be stored, as opposed to $2t_e$ coefficients in the case of the non-programmable encoder. Thus 211 g_{ite} coefficients must be stored in the implementation of a (31,k) RS Encoder.

Implementation of the fixed k (31,k) RS encoder as a simple LFSR is desirable because of its simplicity, and because it takes only 31 steps to code a word using the LFSR. However, n-k-1 GF adders and GF multipliers are needed for such an implementation, a considerable amount of hardware. A slower, but more compact alternative uses one GF adder, and one GF multiplier, and stores the products B_i of the coefficients g_{ite} and the Gate register (see Figure 4.3.1) in memory. In the case of the programmable RS encoder, there are further advantages in using memory storage of the B_i values.

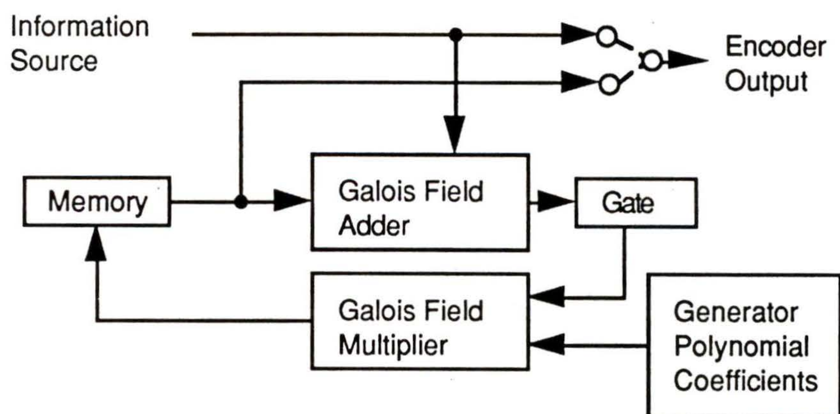


Figure 4.3.1: Block diagram of modified LFSR implementation of the RS encoder.

Two constants of the fixed k (31,k) RS encoder become variables in the programmable (31,k) RS encoder: the number of storage elements B_i and the coefficients g_{ite} of the generator polynomial. In order to implement the programmable (31,k) RS encoder as a simple LFSR, control logic is needed to choose the coefficient g_{ip} corresponding to the number of correctable errors t_e. For coefficients g_{ip} where $p > t_e$, the coefficient g_{ip} could be set to 0, or logic could be introduced to cut off the gates corresponding to g_{ip} for $p > t_e$; in either case the simplicity of the LFSR design is lost, and an external memory is a simple solution.

A modified LFSR implementation is proposed here for the (31,k) RS encoder, using the design of Figure 4.3.1, and storing the generator polynomial coefficients g_{ip} in ROM. The modified LFSR control logic selects the required generator polynomial coefficients from memory, and loads it into the appropriate GF arithmetic unit, to be operated on with the partial result of the current calculations held in a storage register (the Gate).

The number of steps required by the modified LFSR (31,k) encoder, the 5 Stage direct passing algorithm and the 5 Stage RAM passing algorithm are given in Table 4.3.1. Comparing columns two, three and four in Table 4.3.1, it can be seen that the modified LFSR (31,k) RS encoder algorithm requires, for $t_e < 6$, more steps than either of the 5 stage decoding algorithms. This creates timing problems, which are discussed in Section 4.5.

correctable errors	RS Encoder	5 Stage Direct Pass	5 Stage RAM Pass
1	237	220	284
2	387	314	378
3	513	408	472
4	615	509	566
5	693	636	660
6	747	763	872
7	777	890	1121
8	783	1017	1400
9	765	1200	1709
10	723	1433	2048
11	657	1686	2417
12	567	1959	2816
13	453	2252	3245
14	315	2565	3704
15	153	2898	4193

Table 4.3.1: Number of steps vs correctable errors for modified LFSR (31,k) encoder algorithm and 5 stage pipelined (31,k) RS decoder.

Correctable Errors t_e	Range of Order X (0 ->)	Address Range
1	1	0 -> 1
2	3	2 -> 5
3	5	6 -> 11
4	7	12 -> 19
5	9	20 -> 29
6	11	30 -> 41
7	13	42 -> 55
8	15	56 -> 71
9	17	72 -> 89
10	19	90 -> 109
11	21	110 -> 131
12	23	132 -> 155
13	25	155 -> 181
14	27	182 -> 209
15	29	210 -> 239

Table 4.3.2: Example of mapping from i-te indicing to linear addressing.

In order to call up the generator polynomial coefficient g_{ik} required for t_e correctable errors, the generator polynomial coefficients must be indexed in some fashion. The obvious choice is to

use two indices, the order i of $g_i(X)$, and the number of correctable errors t_e, to reference the generator polynomial coefficients g_{ite}. For the (31,k) RS encoder, i is a five bit integer and t_e is a four bit integer. Hence (i,t_e) indexing can be implemented using a 10 address line ROM. However, the (i,t_e) indexing is wasteful, using only 211 of the 1024 memory locations provided on the ROM. If a (63,k) or larger RS encoder were desired, or if it were desirable to store the generator polynomial coefficients in gate logic, some sort of mapping from the (i,t_e) indicing scheme to a more efficient addressing should be used. An example mapping is shown in Table 4.3.2.

The data-flow block diagram for the Xilinx implementation of the programmable (31, k) RS encoder, which can be implemented on an XC3090, is given in Figure 4.3.3. The key for the encoder and decoder data flow diagrams, and a description of the module components, is given in Appendix C. All lines are 5 bit buses unless labelled otherwise. All registers, except for *t_error* (4 bits) and *bgate* (30 GF elements = 150 bits) are 5 bit registers.

The control sequence for the RS (31,k) encoder is given in Appendix B. The number of correctable errors t_e is set by the user, and stored in the register *t_error*. The 4 bits in *t_error* are passed to the upper 4 bits of the registers *two_t -1* and *two_t -2*, which are decremented so that they hold $2t_e - 1$ and $2t_e - 2$, respectively. The 30 GF elements in *bgate* are reset to 0 (31 in the power representation), and *index* is reset to 30. Two nested loops are then performed to calculate the redundancy symbols. Details of the encoder operation are given in Appendix B.

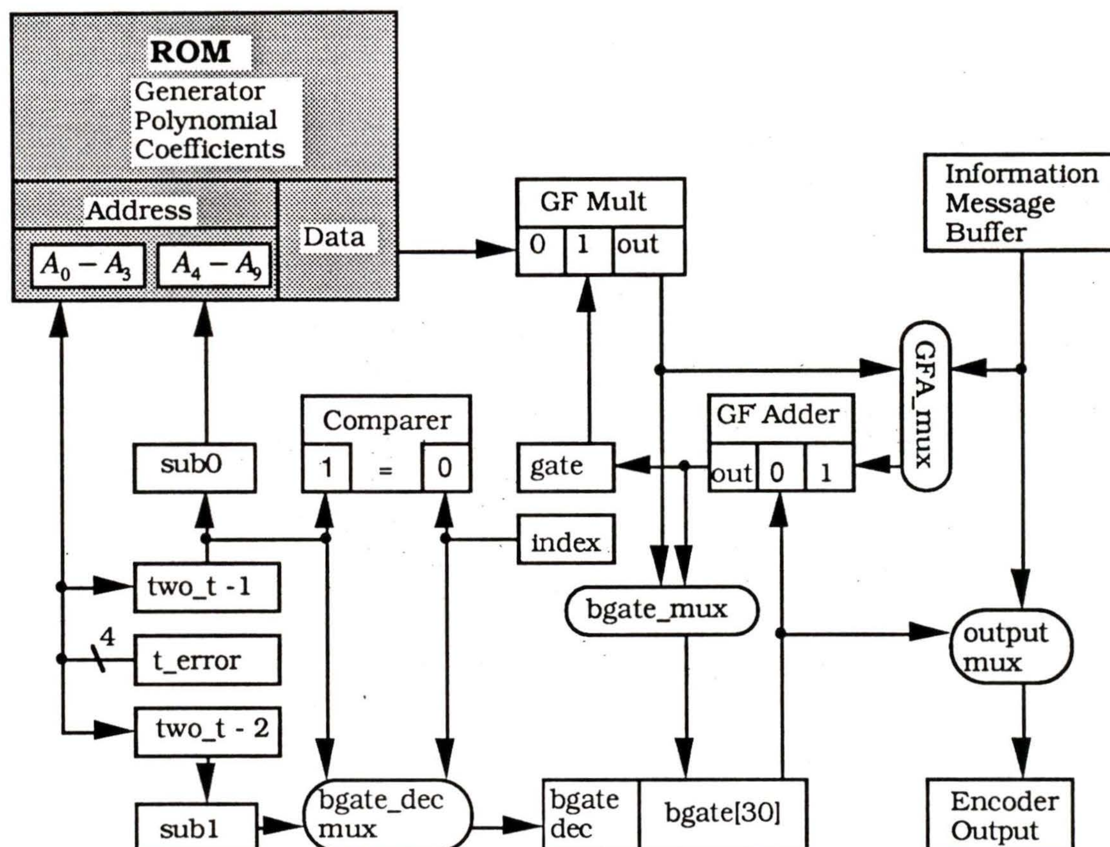


Figure 4.3.3: Data flow block diagram of Xilinx implementation of the programmable (31,k) RS encoder.

4.4 Xilinx Designs for the (31,k) RS Decoder

The (31,k) RS Decoder is subdivided into five physical modules: calculating the syndrome, finding the error-location polynomial δ(X), finding the roots of the error-location polynomial, finding the intermediate polynomial Z(X), and correcting the errors. Designs are made for both direct passing and RAM passing of arrays; however, only the RAM passing architecture is chosen to be written using the design editor XACT [19], and simulated using PC-Silos [30]. The design is intended to keep the modules from having to know the external

memory design. For example, the syndrome module should not have to know to which RAM bank its output is going; the routing of data between modules is the responsibility of the memory logic. This allows the same module design to be used for different numbers of pipelined stages. The control logic used to implement the module algorithms is given in Appendix B.

4.4.1 Xilinx Designs for the Syndrome Stage

The syndrome components S_i (see Section 2.4) may be calculated using a LFSR as shown in Figure 4.4.1 [2,20].

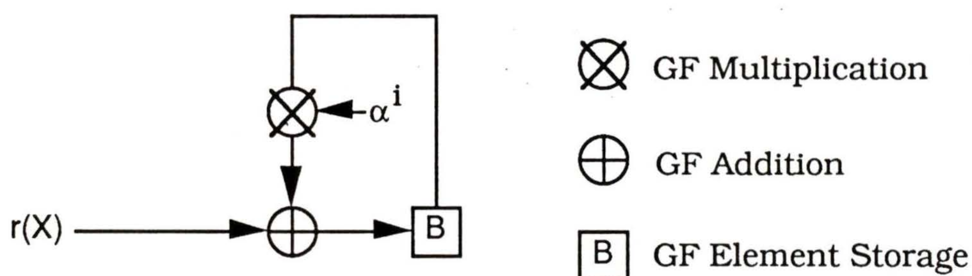


Figure 4.4.1: Syndrome calculations using a LFSR.

Since the syndrome algorithm always results in an even number of syndrome components S_i , the syndrome algorithm may be implemented such that two syndrome components, S_i and S_{i+1}, are calculated in offset calculations (see Figure 4.4.2).

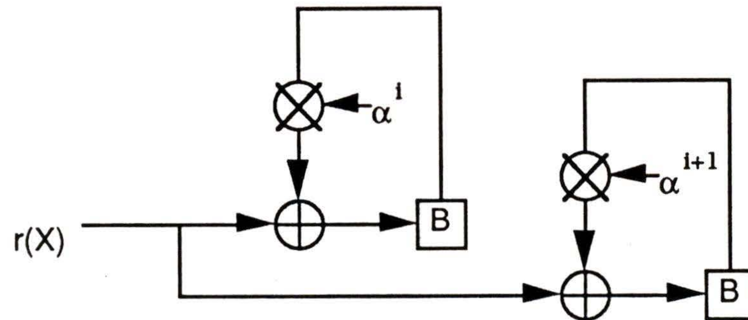


Figure 4.4.2: Paired LFSR calculations of syndrome components.

If both sets of syndrome calculations were performed simultaneously, the calculation would be twice as fast, but would require twice as much hardware. But because most of the hardware is in the GF Arithmetic units, by offsetting the sequence of the two calculations, only one GF adder and one GF multiplier are needed. This off-set syndrome design requires about two-thirds as many operations as does calculating each syndrome component separately, and requires only 10% more hardware. The use of the power representation allows the Galois field elements to be used as a counter in finding the syndrome, further simplifying the syndrome module design. The data flow block diagram for the RAM passing option, which can be implemented on a Xilinx XC3030, is shown in Figure 4.4.3. The data flow block diagram for the direct passing option, is shown in Figure 4.4.4. See Appendix B for the control algorithm, and Appendix C for the key. All lines are 5 bit buses unless labelled otherwise. All registers, except for *t_error* (4 bits), *received* (31 GF elements = 155 bits), and *syndrome* (31 GF elements = 155 bits) are 5 bit registers. The control logic used to implement the module algorithms is given in Appendix B.

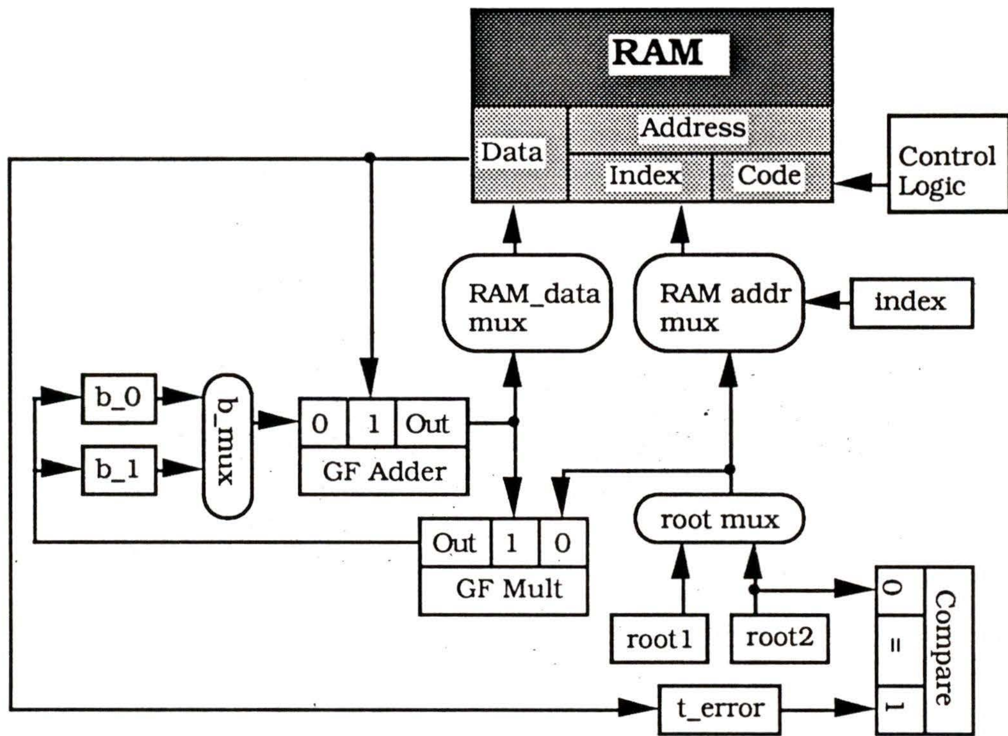


Figure 4.4.3: Data flow block diagram of module used to calculate syndrome for RAM passing.

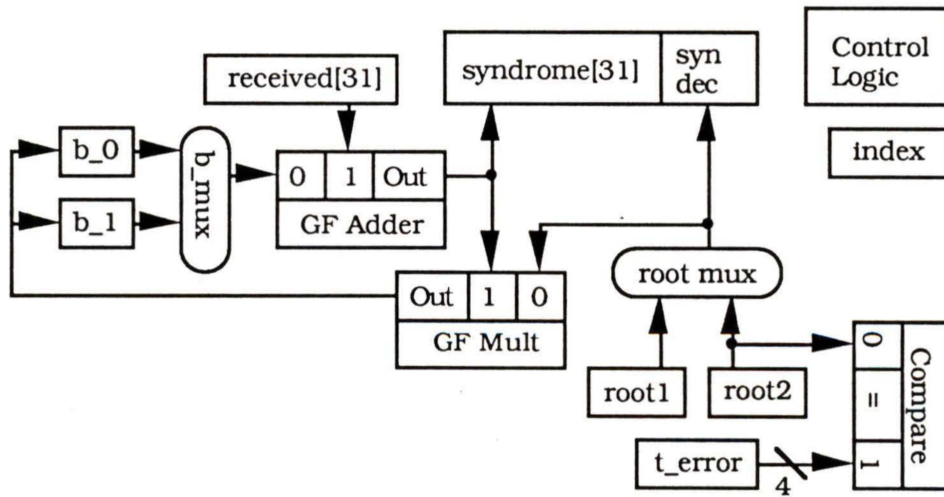


Figure 4.4.4: Data flow block diagram of module used to calculate syndrome for direct passing.

4.4.2 Xilinx Designs for the Delta Stage

Finding the error-locator polynomial is the most complex step in the RS decoding algorithm, as is reflected in the complexity of the hardware layout for the module. Three 16 symbol arrays, *p_delta [ ]*, *shifted_delta [ ]*, and *delta_mu_1 [ ]*, are needed to store data used only in this module; the three registers require a total space of 240 storage bits. These three arrays can be stored in the external RAM along with the intermediate arrays passed between modules. Since five intermediate arrays are stored in external RAM, adding three more arrays would not require adding an extra control bit, so the external memory control logic would not become more complex. However, external RAM storage of the internal arrays would add a memory access step to the algorithm whenever one of the arrays was accessed, and makes it impossible to directly transfer all 80 bits from one register to another in a single step.

The data flow block diagram of the delta module is given in Figure 4.4.5 for the RAM passing architecture, and in Figure 4.4.6 for the direct passing architecture. In order to implement the delta module on a single Xilinx XC3090, it is necessary to store some passive registers in I/O blocks instead of CLB's. It is also possible to break the delta stage into two physical modules, with connections between them. This allows a Xilinx XC3042 and a Xilinx XC3064 to be used instead of a single Xilinx XC3090. See Appendix B for the control algorithm, and Appendix C for the key. All lines are 5 bit buses unless labelled otherwise. All registers are 5 bit registers, except for *sub_zero* (1 bit), *t_error* (4 bits), *syndrome [ ]* (31 GF elements = 155

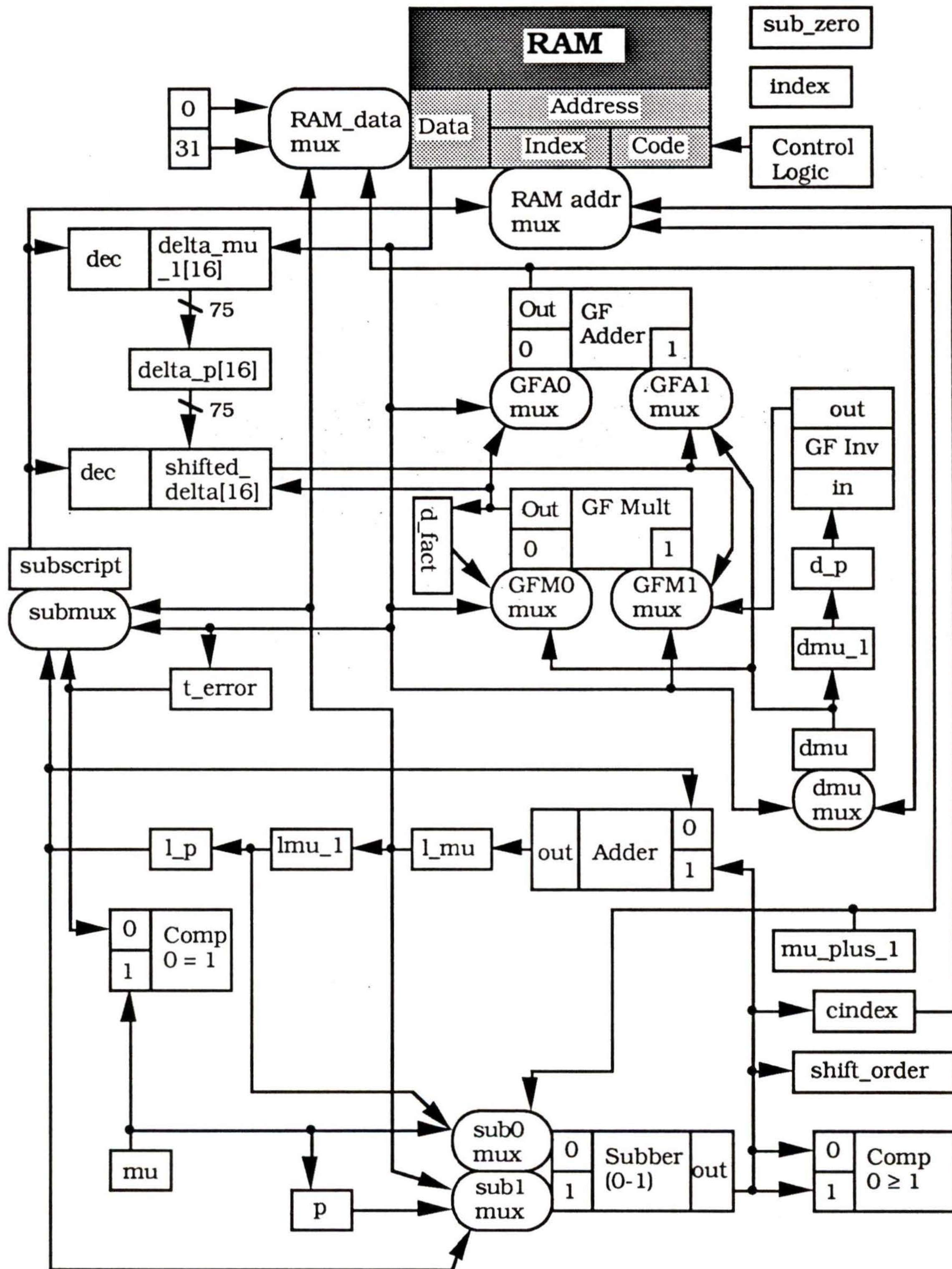


Figure 4.4.5: Data flow block diagram of module used to determine ∂(X) for RAM passing.

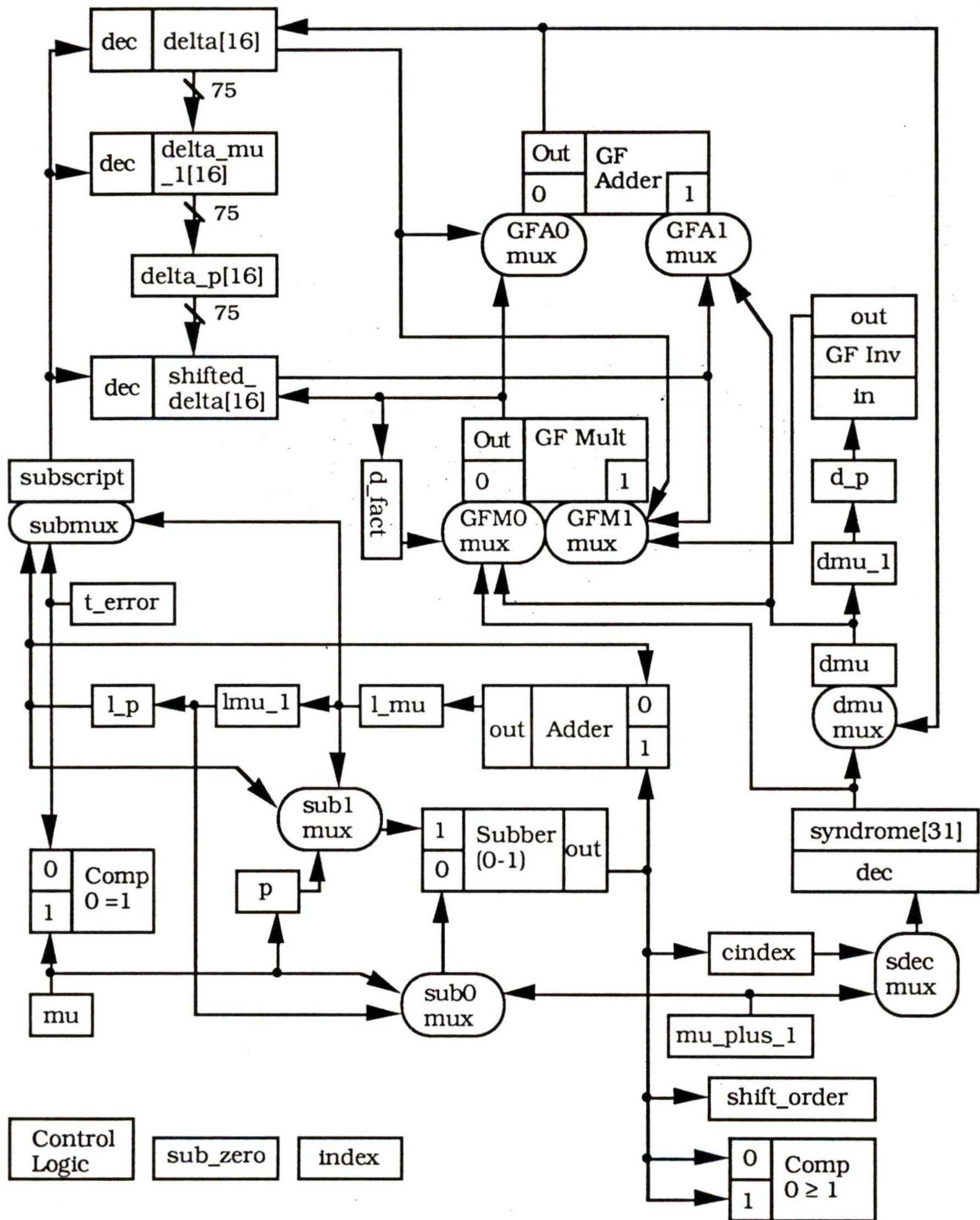


Figure 4.4.6 : Data flow block diagram of module used to determine δ(X) for direct passing.

bits), and *delta [ ]*, *delta_p [ ]*, *delta_mu_1 [ ]* and *shifted_delta [ ]* (16 GF elements = 80 bits). The control logic used to implement the

module algorithms is given in Appendix B.

4.4.3 Xilinx Designs for the Location Stage

Finding the positions of the transmitted errors requires finding the zeros of the error-location polynomial $\delta(X)$. The algorithm used inserts the error locations into the error-location polynomial $\delta(X)$, and records those error locations that give $\delta(X) = 0$. The algorithm could be stopped when the number of correctable errors reaches t_e. However, since the modules must be synchronized for the maximum number of steps for each module, no saving in the total number of decoder steps is obtained by stopping the location module when the number of errors reaches t_e.

The block diagram for the module finding the error locations, which can be implemented on a Xilinx XC3030, is given in Figure 4.4.7 for the RAM passing of arrays, and in Figure 4.4.8 for direct passing of arrays. See Appendix B for the control algorithm, and Appendix C for the key. All lines are 5 bit buses, and all registers are 5 bit registers, except for *delta [ ]* and *location_of_errors [ ]* (16 GF elements = 80 bits). The control logic used to implement the module algorithms is given in Appendix B.

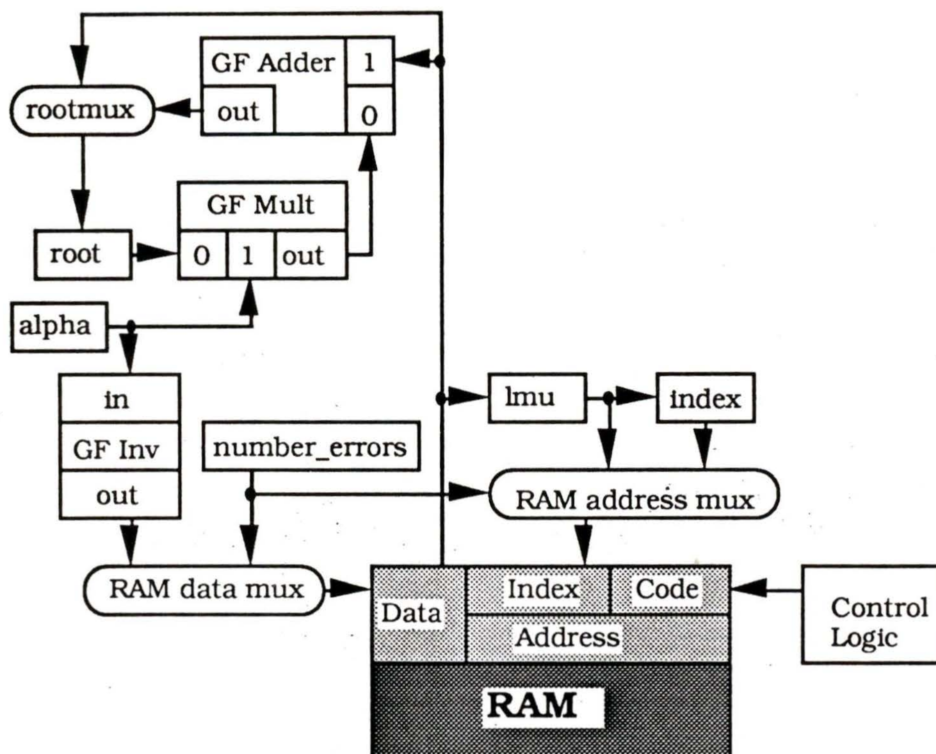


Figure 4.4.7: Data flow block diagram of module used to find the error locations for RAM passing.

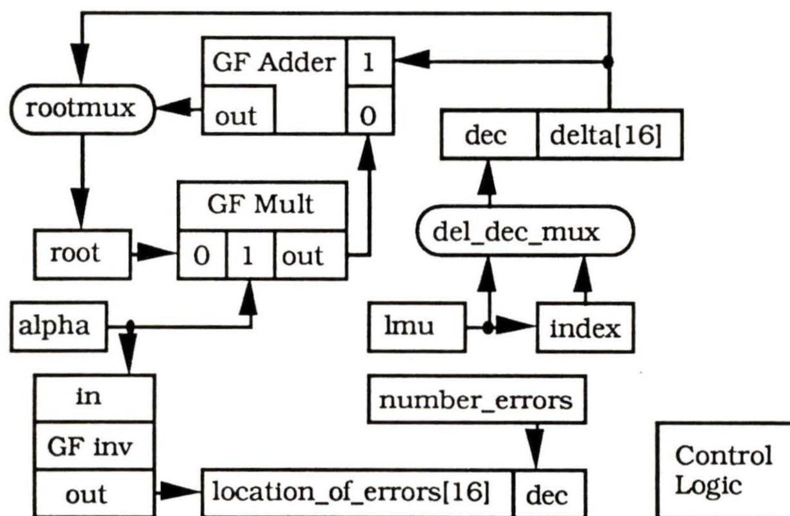


Figure 4.4.8: Data flow block diagram of module used to find the error locations for direct passing.

4.4.4 Xilinx Designs for the Z(X) Stage

The block diagram for the module finding Z(X), which can be implemented on a Xilinx XC3030, is given in Figure 4.4.9 for RAM passing of arrays, and in Figure 4.4.10 for direct passing of arrays. Two comparators are used in the design, in order to simplify the routing. Because each comparator requires only $2^1/_2$ CLB's, this also requires less space than adding a multiplex to each end of a comparator. See Appendix B for the control algorithm, and Appendix C for the key. All lines are 5 bit buses, and all registers are 5 bit registers, except for *syndrome [ ]* (31 GF elements = 155 bits) and *z_of_x [ ]* (16 GF elements = 80 bits). The control logic used to implement the module algorithms is given in Appendix B.

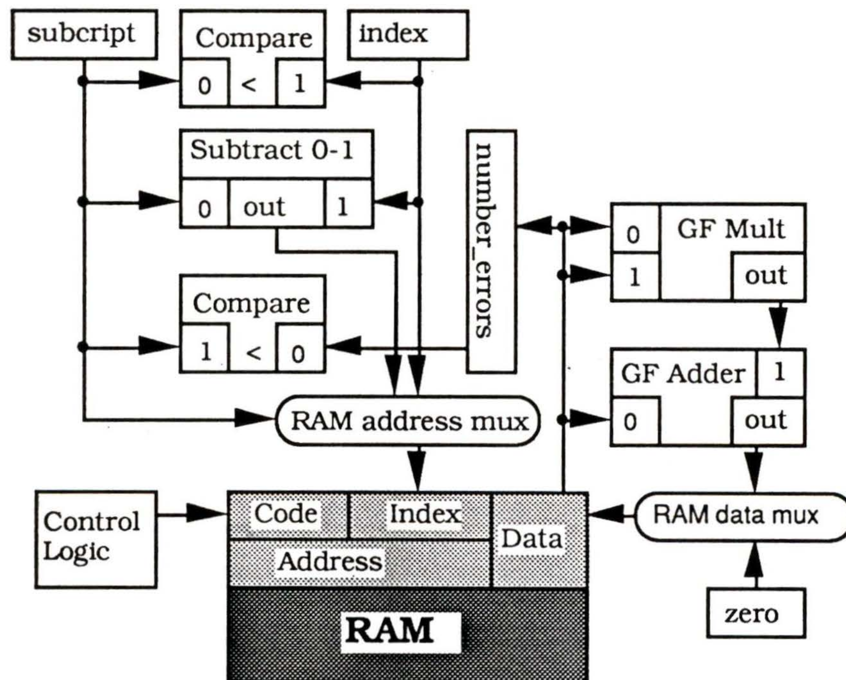


Figure 4.4.9: Data flow block diagram used to calculate Z(X) for RAM passing .

Figure 4.4.10: Data flow block diagram used to calculate Z(X) for direct passing.

4.4.5 Xilinx Design for the Correction Stage

The correction block calculates the corrected symbol for each error location, and stores the corrected symbol in the received code word. The data flow block diagram for the module correcting the errors, which can be implemented on a Xilinx XC3042, is given in Figure 4.4.11 for RAM passing of arrays, and in Figure 4.4.12 for direct passing of arrays. See Appendix B for the control algorithm, and Appendix C for the key. All lines are 5 bit buses, and all registers are 5 bit registers, except for *error []*, *location_of_errors [ ]*, and *z_of_x [ ]* (16 GF elements = 80 bits), and *received [ ]* (31 GF elements = 155 bits). The control logic used to implement the module algorithms is given in Appendix B.

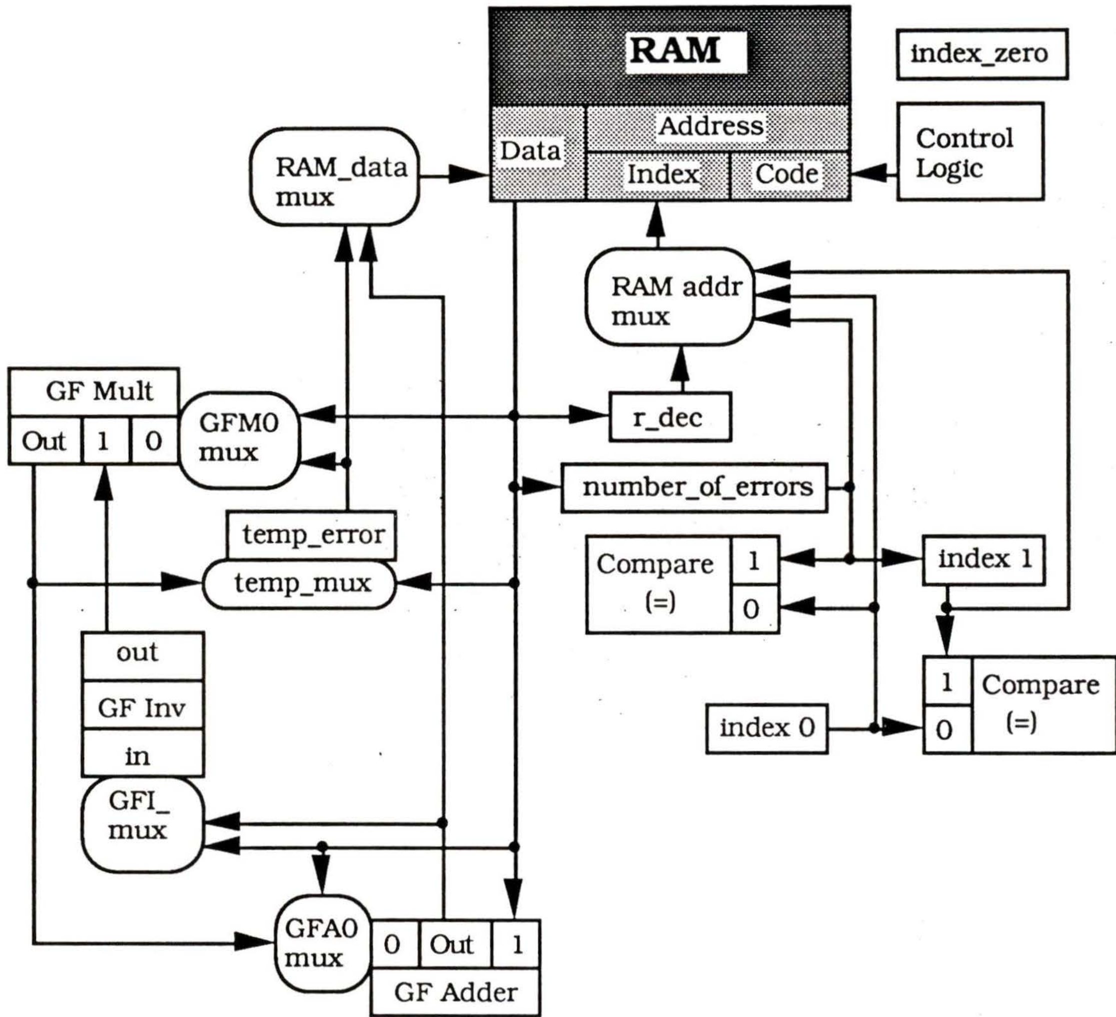


Figure 4.4.11: Data flow block diagram of module used to correct errors and transmit the corrected information message for RAM passing.

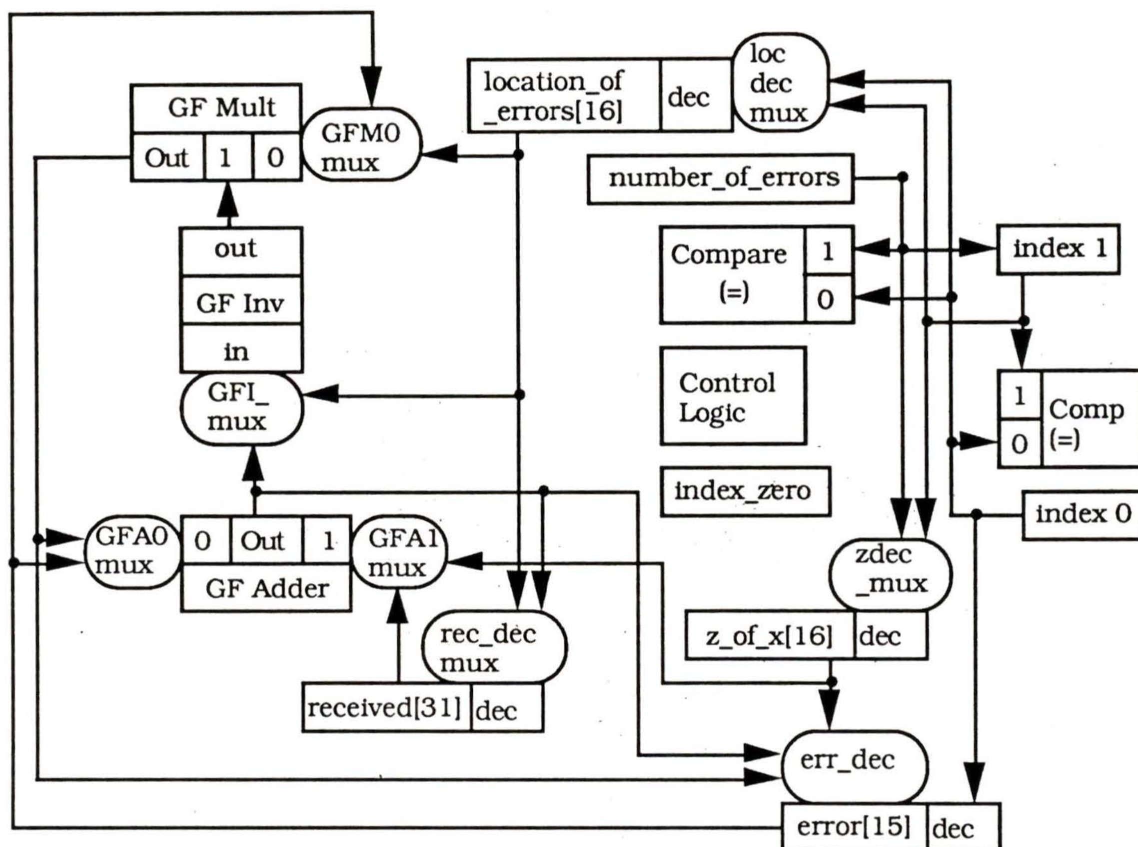


Figure 4.4.12: Data flow block diagram of module used to correct errors and transmit the corrected information message for direct passing.

4.5 Array Passing Logic and Handshaking

The stages of the RS encoder and decoder have been described. It is now necessary to describe the passing of information between stages of the decoder, the handshaking between the information source and the encoder, and the handshaking between the decoder and the information user.

The RS decoding algorithm requires a code word to be completely received before it can be decoded. Thus an input buffer is required to store the incoming code word while the syndrome module is decoding the previous code word. As well, logic is needed to switch the code word between the input buffer and the syndrome module.

4.5.1 Direct Passing of Arrays

If the direct passing architecture is chosen, an input buffer is needed to store the incoming code word while the syndrome module is decoding the previous word, and logic is needed to switch the code word between the input buffer and the syndrome module.

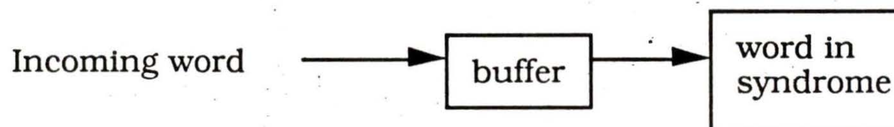


Figure 4.5.1: Input buffer for direct passing of arrays.

Ideally, when the input buffer is full it should be transferred in one step to the received word buffer associated with the syndrome module. Unfortunately, this requires 155 data lines between the buffers, which is impractable for a Xilinx implementation. A simpler option is to switch the incoming word line, and the five data lines used by the syndrome module, so that each buffer alternates between receiving the code word, and passing the word to the syndrome module (see Figure 4.5.2).

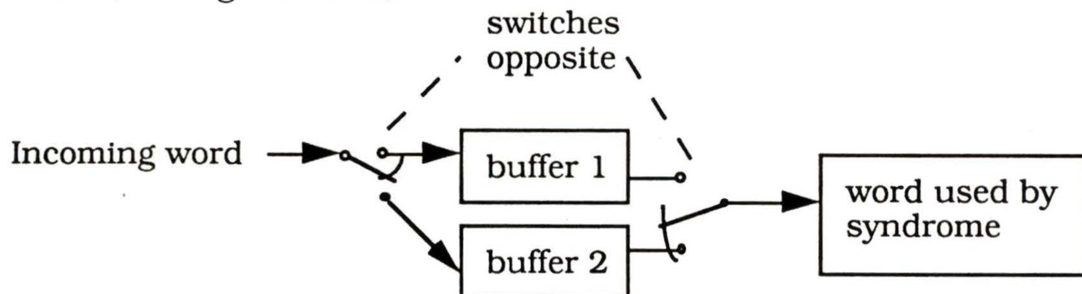


Figure 4.5.2: Double buffered input buffer for direct passing of arrays.

4.5.2 External RAM Array Passing

The received block is used in the first (syndrome) and last (correction) stages of the RS decoder (see Figure 3.2.1). This means that if the incoming code word is to be stored in RAM, two received code words must be stored in each RAM: the present code word and the next code word. A flag is also required in each module to choose between the two code words (see Figure 4.5.3).

A five bit buffer is used to store the incoming GF(2^5) symbol from the digital demodulator. This requires the digital demodulator either to send a 5-ary symbol, or if the source is a binary line, the decoder must store bits until it can transfer five bits simultaneously. Since the modules may require RAM access simultaneously with the input buffer, some way of insuring non-conflict is needed. The easiest way of implementing this is to disengage the module clock whenever the input buffer needs to access the RAM. From Figure 3.3.11, we see that from 200 to 10000 module steps are required in the time that 31 symbols must be entered into the RAM; thus the decoder is slowed by between 14% and .3% by disengaging the clock during symbol loading.

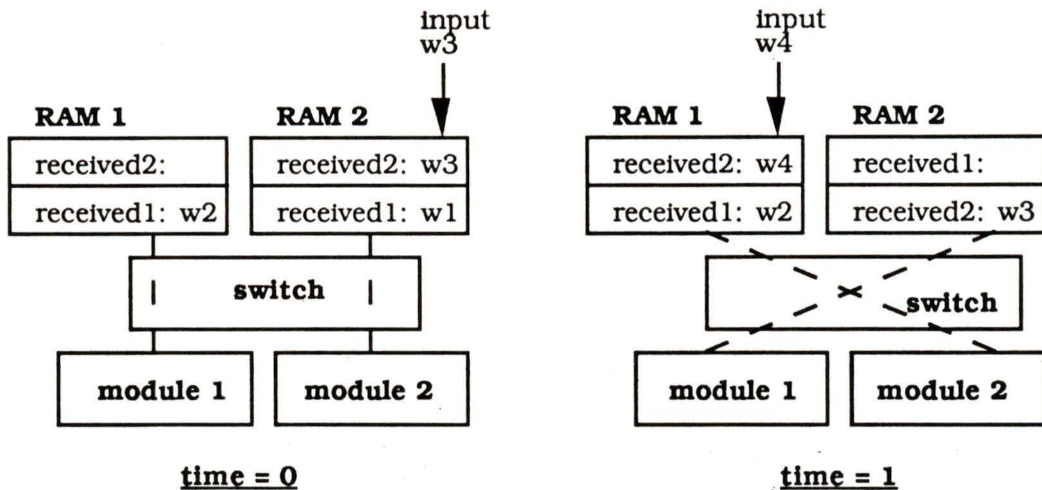


Figure 4.5.3: Diagram of input of incoming symbols into RAM, for the two module RS decoder.

4.5.3 Switching Circuit

The switching circuit is used to connect the physical decoder modules to the external memory associated with each pipelining stage. Since the data associated with a given word stays in the same memory chip, the connections between the physical stages and the memory chips rotate. Only the switching circuit for the five stage pipelined RS decoder will be examined here.

The five physical modules each have 20 lines associated with them, consisting of 5 input data lines, 5 output data lines, 8 address lines and 2 control lines. Six arrays and three five bit integers are passed between modules as shown in Chapter 3, and listed in Table 4.5.1. The arrays are given address codes from 0 to 6, the three integers are given the address code 7. The address code makes up the upper 3 bits of the 8 bit address for the array or integer in the external RAM. The lower 5 bits are the array index address; for the integers the index is given in Table 4.5.1.

Code	Register	Address
0	received [31] #1	0-30
1	received [31] #2	0-30
2	syndrome [31]	0-30
3	delta [16]	0-15
4	location_error [16]	0-15
5	Z [16]	0-15
6	error [16]	0-15
7	lmu	0
7	t_error	1
7	number_of_errors	2

Table 4.5.1: Intermediate values passed between modules.

Separate input and output data lines are needed because of the lack of efficient 3 state logic lines in the Xilinx XC3000 Series. The RAM chips have 15 data lines associated with them, consisting of 5 data lines, 8 address lines and 2 control lines. The 5 data lines between the RAM and the switching circuit must be bi-directional, and so use the inefficient 3-state logic lines. The RAM has three control lines:

~E = chip enable

~W = write enable

~G = output enable.

The control line signals to read and write are shown in Table 4.5.2.

Pin	Write Signal	Read Signal
~E	0	0
~W	0	1
~G	1	0

Table 4.5.2: RAM read and write signals.

Since ~E is down for read and for write, it may be grounded, and need not be included as a control line for the switching circuit.

Altogether 35 I/O pins are needed for each of the five stages, plus one I/O pin for the clock, and 6 lines from the input buffer, for a total of 182 I/O pins.

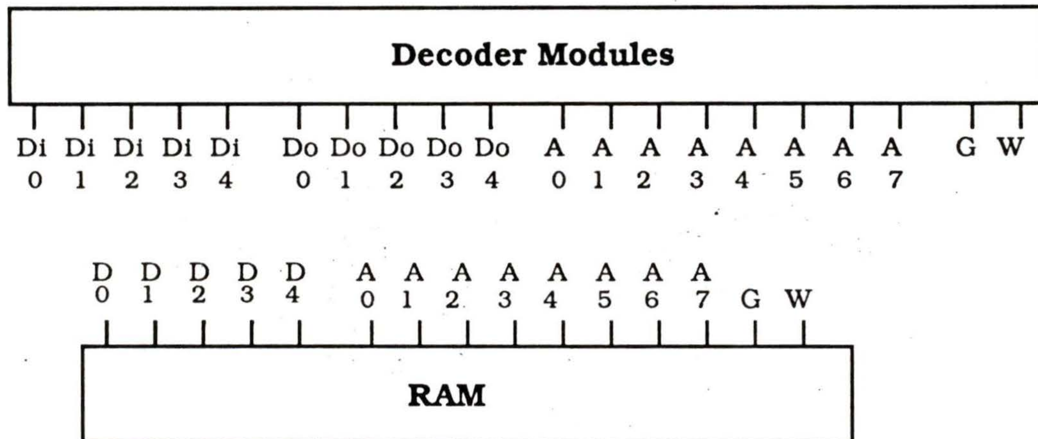
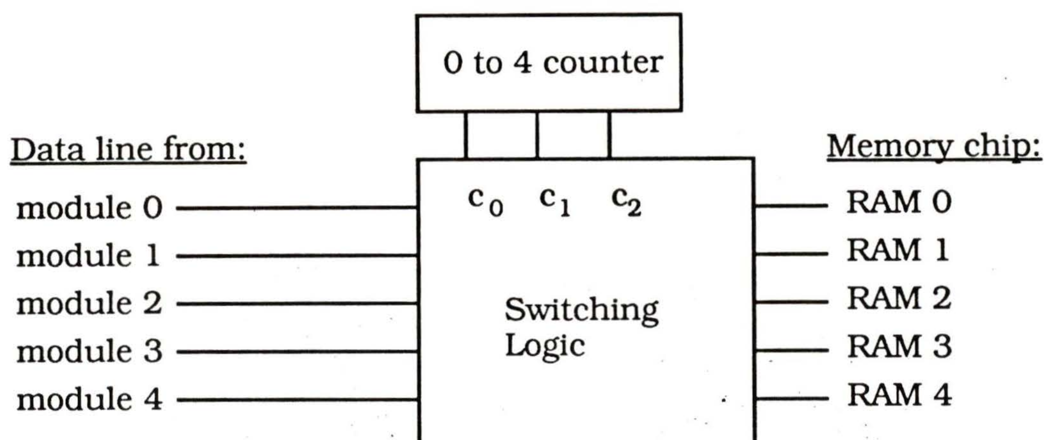


Figure 4.5.4 : Decoder Module and RAM I/O pins.

Each Xilinx XC3000 Series CLB has 5 inputs and 2 outputs; therefore the most efficient switching method is for each output pin to choose one of five input pins, depending on the point in the decoding cycle (see Figure4.5.5). The choice of input pin could be controlled either by a five state counter (3 bit) or by a shift (one bit). The shift is more efficient in terms of the number of CLB's required, but does not keep track of the cycle position, and so is unable to correct itself if a shift-signal error occurs.



$$RAM_0 = m_0 \bullet \sim c_0 \bullet \sim c_1 \bullet \sim c_2 + m_1 \bullet c_0 \bullet \sim c_1 \bullet \sim c_2 + m_2 \bullet \sim c_0 \bullet c_1 \bullet \sim c_2 + m_3 \bullet c_0 \bullet c_1 \bullet \sim c_2 + m_4 \bullet \sim c_0 \bullet \sim c_1 \bullet c_2$$

$$RAM_1 = m_1 \bullet \sim c_0 \bullet \sim c_1 \bullet \sim c_2 + m_2 \bullet c_0 \bullet \sim c_1 \bullet \sim c_2 + m_3 \bullet \sim c_0 \bullet c_1 \bullet \sim c_2 + m_4 \bullet c_0 \bullet c_1 \bullet \sim c_2 + m_0 \bullet \sim c_0 \bullet \sim c_1 \bullet c_2$$

$$RAM_2 = m_2 \bullet \sim c_0 \bullet \sim c_1 \bullet \sim c_2 + m_3 \bullet c_0 \bullet \sim c_1 \bullet \sim c_2 + m_4 \bullet \sim c_0 \bullet c_1 \bullet \sim c_2 + m_0 \bullet c_0 \bullet c_1 \bullet \sim c_2 + m_1 \bullet \sim c_0 \bullet \sim c_1 \bullet c_2$$

$$RAM_3 = m_3 \bullet \sim c_0 \bullet \sim c_1 \bullet \sim c_2 + m_4 \bullet c_0 \bullet \sim c_1 \bullet \sim c_2 + m_0 \bullet \sim c_0 \bullet c_1 \bullet \sim c_2 + m_1 \bullet c_0 \bullet c_1 \bullet \sim c_2 + m_2 \bullet \sim c_0 \bullet \sim c_1 \bullet c_2$$

$$RAM_4 = m_4 \bullet \sim c_0 \bullet \sim c_1 \bullet \sim c_2 + m_0 \bullet c_0 \bullet \sim c_1 \bullet \sim c_2 + m_1 \bullet \sim c_0 \bullet c_1 \bullet \sim c_2 + m_2 \bullet c_0 \bullet c_1 \bullet \sim c_2 + m_3 \bullet \sim c_0 \bullet \sim c_1 \bullet c_2$$

Figure 4.5.5: Choosing proper RAM chip. (see Appendix D for details of Xilinx implementation).

4.5.4 External Interfaces

The programmable (31,k) RS encoder and decoder both have the external interfaces shown in Figure 4.5.6.

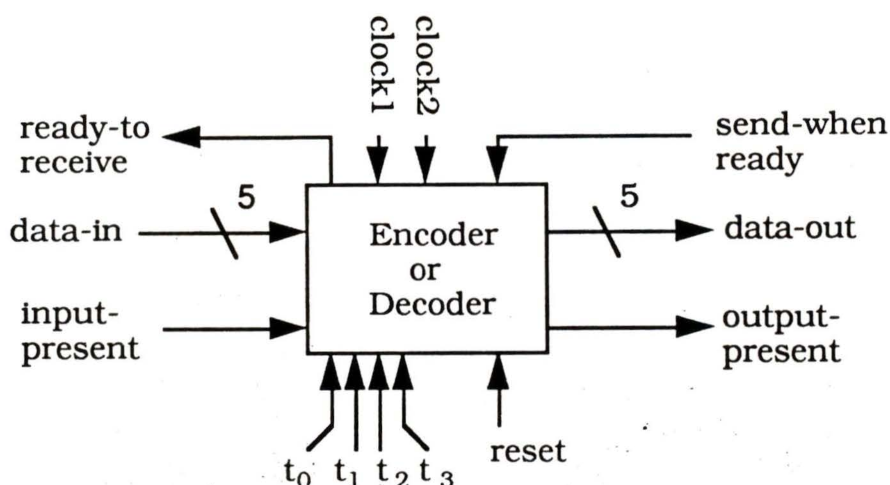


Fig. 4.5.6: External pins for encoder and decoder.

Data-in and data-out are 5-bit lines, carrying the GF(2^5) symbols, input-present and output-present are single lines going high when data is present. Clock1 is the clock for the data-in line, clock2 is the clock used for the modules. The lines t_0, t_1, t_2, t_3 give the number of correctable errors associated with the received word.

The RS encoder adds redundant bits to the information bits it receives from the information source. Thus the bit rate of the encoded data must be higher than the bit rate of the information source. Two methods have been considered here for adding the parity check bits into the stream of information bits. The first method of adding parity check bits into a stream of information bits assumes that the information source sends a steady stream of bits, without waiting for a go-ahead from the encoder. The RS encoder must then store the incoming bits in a buffer while calculating the parity check bits and outputting the encoded code word. The second method considered here of adding parity check bits into a stream of information bits

assumes that the information source waits for a go-ahead signal from the encoder before sending out new information bits. The encoder accepts information bits from the information source until the information block is full, and then sends a signal to the information source telling it not to send any further information bits. The encoder then calculates and outputs the parity check bits, until the encoded block is finished; after the old block is finished the encoder sends a signal to the information source telling it to continue sending information bits.

Since an encoder that expects the information source to wait until the encoder sends it a go ahead signal cannot run with a steady stream information source, the method one implementation is more general, and is used here.

For the RS decoder, the output information rate is less than that of the incoming code word rate. If the correction stage of the decoder outputs the information message, the result will be bursts at the rate of its internal clock. One solution is to have a circuit within the switching circuit responsible for outputting the corrected information. If this circuit was to take the information from RAM, the correction stage would have to be interrupted while the information access was going on. Alternatively, the switching circuit could hold the information symbols in a buffer after receiving them from RAM at the end of the previous module cycle, and output them according to a clock fed into it. This later method has been implemented here.

The switching circuit disengages the internal clock of the modules on the rising edge of the data-present line for one clock cycle, as the

incoming symbol is stored in RAM.

4.5.5 Clocks

The number of operations required to complete a decoding stage varies with the number of correctable errors, and the pipelining option chosen (see Table 4.5.3). In this design only the five-stage pipelining option will be considered; however, the clock design is easily modified for any of the other pipelining options.

Thus, some way of ensuring that the encoder rate equals the decoder rate is required. The problem has a further complication, in that the codec should be able to change the number of correctable errors for each information word.. Because for most values of t_e, and most pipelining options, the encoder requires fewer operations than the decoder, the problem of fitting the encoder speed to the decoder speed will only be examined.

correct. errors	encoder	one stage	two stages	three stages	four stages	five stages
1	237	574	366	366	315	315
2	387	966	532	532	409	409
3	513	1419	729	729	503	503
4	615	1933	1007	957	597	597
5	693	2508	1323	1216	691	691
6	747	3144	1669	1506	903	903
7	777	3841	2045	1827	1152	1152
8	783	4599	2451	2179	1431	1431
9	765	5418	2887	2562	1740	1740
10	732	6298	3353	2976	2079	2079
11	657	7239	3849	3421	2448	2448
12	567	8241	4375	3897	2847	2847
13	453	9304	4931	4404	3276	3276
14	315	10428	5517	4942	3735	3735
15	153	11613	6133	5511	4224	4224

Table 4.5.3: Number of operations required to complete encoder and RAM passing decoder options., including symbol loading of RAM.

The easiest solution is to assume the encoder and decoder can communicate with each other via a feedback channel. The encoder then accepts information messages from the information source, and sends coded words, as long at it receives a send-when-ready signal from the decoder. The decoder only sends a send-when-ready signal when its input buffer is free to receive a coded word. However, the encoder and decoder may be used in situations where it is not easy to relay control signals between themselves, and so a method may be required to match the encoder output bit-rate and the decoder input bit-rate.

One alternative is to use the rate of the highest rate code ($t_e = 1$) as the codec rate, and use counters to pause the codec for lower rate codes. That is, an internal counter would load in the difference in steps to completion between the five stage decoder and the encoder. When the encoder completes, it would pause until the counter counts down to zero. From Table 4.5.3 we see that for $t_e = 3$ to 5 the five stage decoder runs faster than the encoder, so that the encoder clock must run 2% faster than the decoder clock. For example, using an encoder clock 2% faster than the decoder clock, we must introduce 315 - 232 = 83 wait-cycles in the encoder for $t_e = 1$, 409- 380 = 29 wait-cycles for $t_e = 2$, no wait-cycles for $t_e = 3$ or 4, etc. However, using this approach, the encoder sends bursts of bits at its maximum bit-rate, which does not allow a lower-bandwidth line to be used, and increases the probability of transmission error through burst errors.

correct. errors	ratio decoder/ encoder	frequency ratio used
1	.89	.85
2	1.02	1
3	1.10	1
4	1.10	1
5	1.05	1
6	.85	.85
7	.69	.69
8	.59	.59
9	.45	.42
10	.36	.35
11	.27	.25
12	.21	.21
13	.14	.13
14	.085	.085
15	.036	.035

Table 4.5.4: Ratio of encoder clock frequency to that of decoder.

A second alternative, and the one chosen here, is to have the encoder clock speed dependent on the number of correctable errors. The encoder-clock would take as its input the number of correctable errors, which would be used to select an appropriate clock. The clock ratios (decoder clock frequency = 1) for the five stage option is given in table 4.5.4. The number of clocks required may be reduced by using a clock of frequency ratio 1 for te = 2, 3, 4, 5, a clock of frequency ratio .85 for te = 1, 6, a clock of frequency ratio .7 for te = 7 and a clock of frequency ratio .6 for te = 8. Dividers may be used to obtain the frequency ratios .42, .35, .25, .21, .13, .085 and .035.

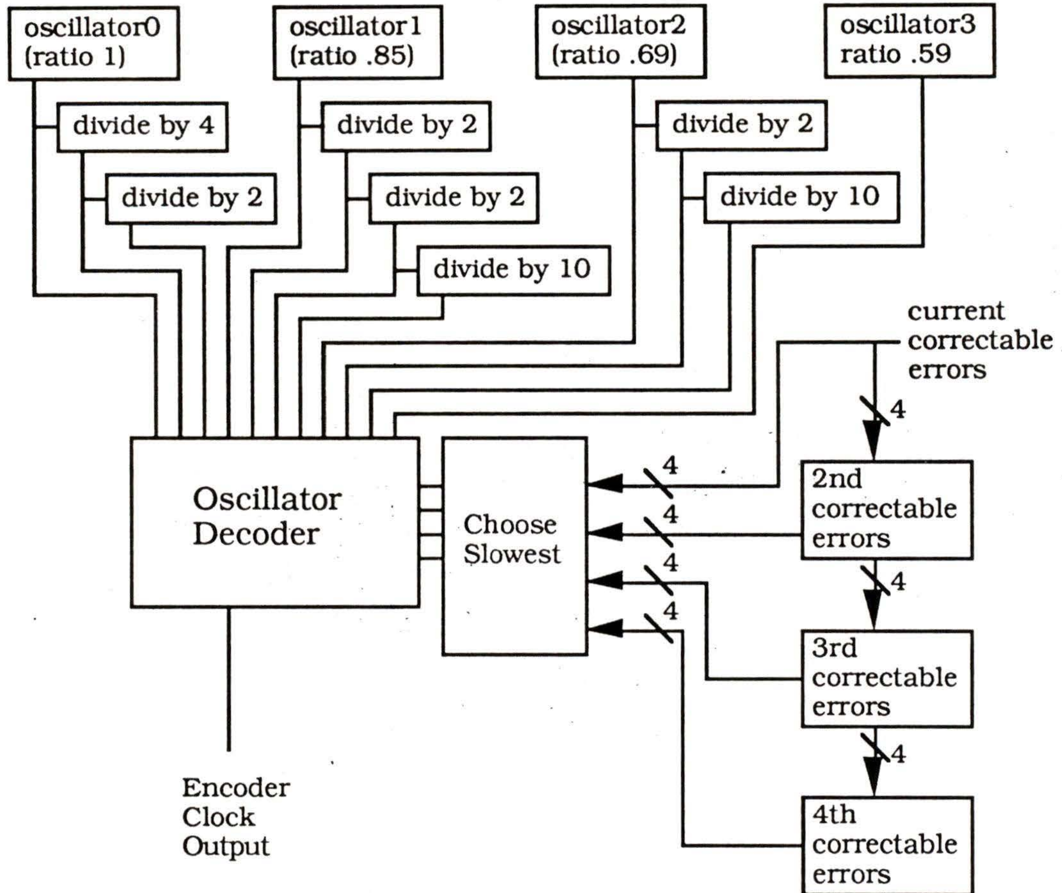


Figure 4.5.7 : Block diagram of encoder clock circuit, where the oscillator ratios are given in terms of the RS decoder clock frequency.

In order to switch between the number of correctable errors, given the different code rates, the encoder must either wait until the words coded at the previous number of correctable error has cleared the five decoder stages, or run at the rate of the lowest rate code in the decoder. Five registers in the clock keep track of the current and previous four clock rates (coded from 1 to 11), and a comparator switch chooses the slowest clock rate. A block diagram of the clock circuit is shown in Figure 4.5.7.

4.6 Verification and Summary

The XACT design editor [19] was used to make the data files that store the data used by the Xilinx chips. Complete designs were made for the (31,k) RS encoder, the (31,k) RS decoder RAM passing modules, and the switching circuit, on the Xilinx chip given in Table 4.6.1.

Module	Xilinx Chip
encoder	XC 3090
syndrome	XC 3030
delta	XC 3090
location	XC 3030
z(X)	XC 3030
correction	XC 3042
switching	XC 3090

Table 4.6.1 : Xilinx chips used for implementation of RAM passing (31,k) RS codec.

Each of the above were tested using the simulator program PC-SIlos [30]. Xilinx chips with a flip-flop toggle speed of 100 MHz are available, however, due to signal delays in the internal routing lines, the fastest safe internal clock speed is 5 MHz. Figure 4.6.1 shows the input bit rates for the five RAM passing RS (31,k) decoder options. The input speed of the four stage and of the five stage pipelining options, in the case of the RAM passing architecture, are the same. The range in bit rates for each of the pipelining options is given in Table 4.6.2. As the degree of pipelining increases from one stage to four stages, there is a slight (2%) decrease in the range of bit rates over the number of correctable errors.

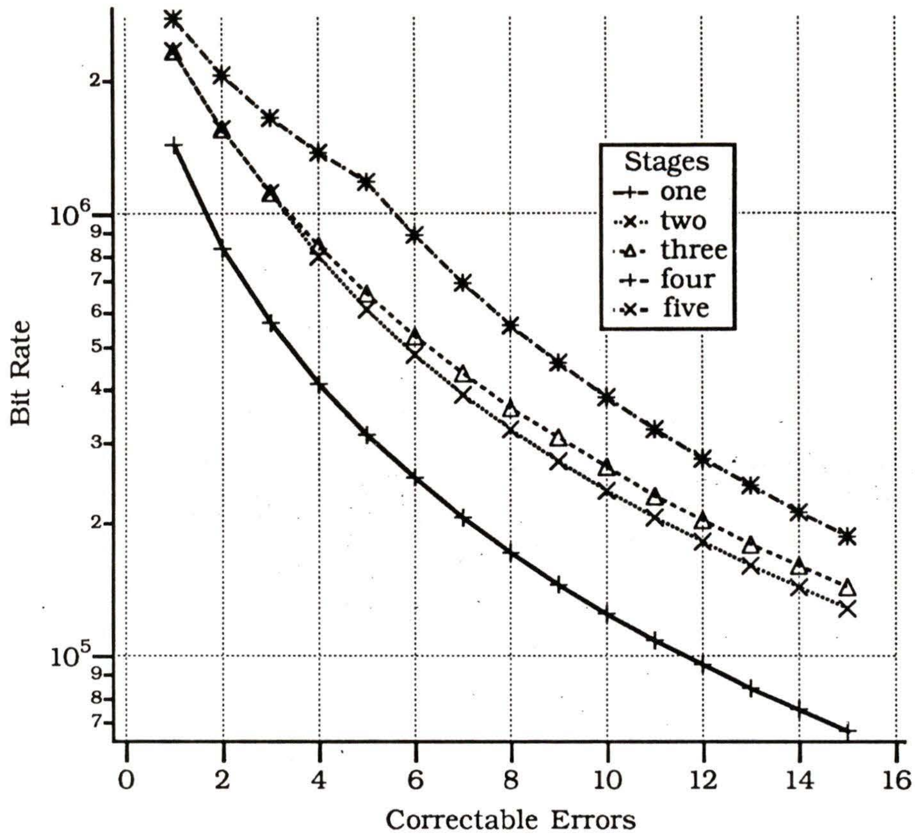


Figure 4.6.1: Decoder speeds per pipelining option for RAM passing decoder.

Pipelining Option	speed t_e = 15 as % of speed t_e = 1
one	4.7
two	5.6
three	6.1
four	6.8
five	6.8

Table 4.6.2: The speed of pipelining options at t_e = 15 as a percentage of the speed at t_e = 1.

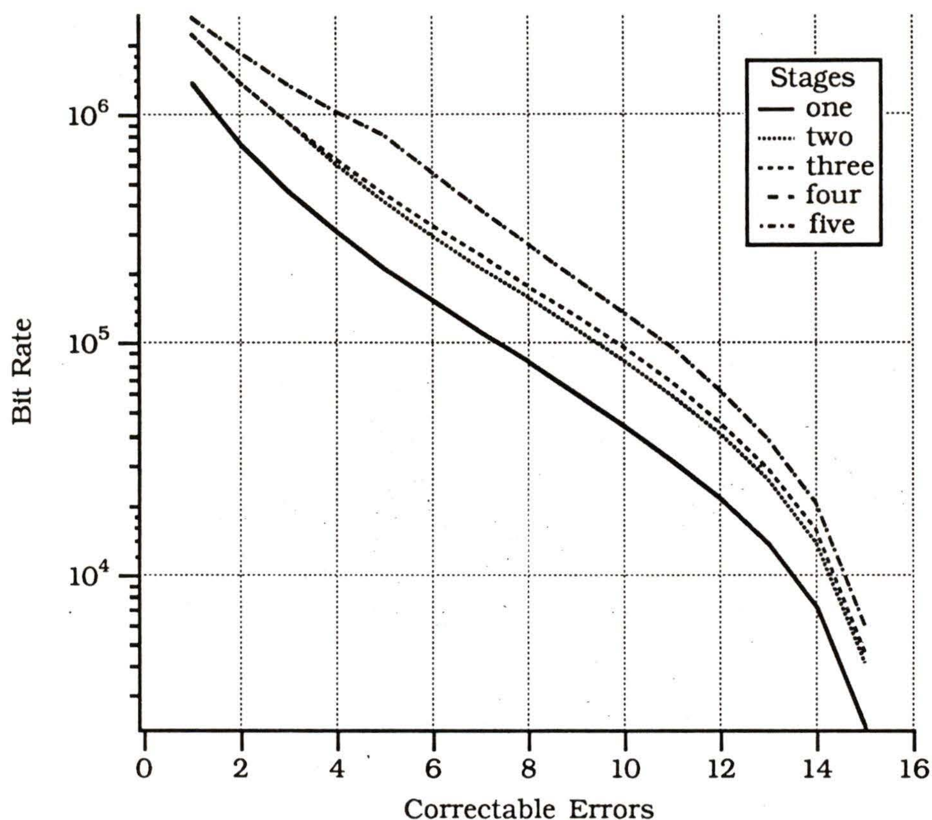


Figure 4.6.2: Decoder speeds per pipelining option for RAM passing decoder.

The input and output bit-rates differ because while the input block is always 31 GF(2^5) symbols (155) bits, the output information block varies with the number of correctable errors t_e. The size of the output block is ($31-2t_e$) GF(2^5) symbols; ie, it ranges from 145 to 5 bits as t_e goes from 1 to 15. Figure 4.6.2 shows the output bit rates for the five pipelining options at a clock speed of 5 MHz.

Summary

VLSI designs for pipelined RS decoders have been given for both frequency [9] and time domains [10], and a time domain programmable RS decoder has recently been implemented on a Gate-

Array by Shayan, Le-Ngoc and Bhargava [12]. The pipeline RS decoders designs are difficult to modify to be user programmable for correctable errors. The programmable RS decoder, using Blahut's universal RS decoder algorithm [6] has the advantage of being independent of the number of correctable errors in the RS code, and of correcting both errors and erasures. A decoder using Blahut's universal decoder algorithm requires n(n+1) operations to decoder a (n,k) RS code, or 992 operations to decode a (31,k) RS code word. The five stage (31,k) RS decoders proposed here require between 315 and 4224 operations per code word for the RAM passing architecture (see Table 3.3.9 plus 31 input operations per word), while the direct passing architecture requires between 251 and 2929 operations (see Table 3.3.5 plus 31 input operations per word).

The designs presented here use fewer operations for high rate codes (te $\leq$ 6 for RAM passing, te $\leq$ for direct passing); however, they require more operations for low rate codes, and the variable rate itself may be a problem in some applications. As well, the decoder by Shayan et al[25] has been implemented on a Gate-Array, allowing a clock speed of 20 MHz and an input bit rate of 3 MHz, while the 5 MHz clock possible on the Xilinx design gives a range for the input bit rate of between 2.8 MHz and 180 kHz for the RAM passing option.

An increase of 200% (t_e = 1) to 300% (t_e = 15) over the speed of the single stage decoder in the bit-rate of the RAM passing architecture for the RS (31,k) decoder is achieved by introducing four stages of pipelining. The cost of introducing pipelining is an additional RAM chip for each stage. RAM passing also has the

advantage of increasing the modularity of the design. This allows the same decoder modules to be used, with only the switching circuits changed, for all the pipelining options. The RAM passing architecture also makes it simple to change the number of correctable errors without stopping the codec.

If the direct passing architecture is used, the speed of the decoder increases by about 30%, but the size of the decoder doubles in going from one stage to five stages. It is possible to combine two XC3030 chips on a XC3090 chip, thus decreasing the number of chips needed for the direct passing decoders. However, because the passing of arrays is dependent on the degree of pipelining, the modules have to be altered to switch between different levels of pipelining. Using the power representation results in a saving of 10% in CLB's for $GF(2^5)$ arithmetic, and reduces the design complexity of the design by allowing counters to also serve as GF elements.

The overall speed of both the RAM passing and direct passing decoders could be increased by increasing the frequency of their internal clocks. The interconnections between the cells in the Xilinx XC3000 Series are routed through the use of switches, each of which introduces a signal propagation delay. As the design on a given chip becomes more complex, signal paths must be routed through more switches, reducing the possible clock speed. An expensive method of reducing the signal delay due to the crowding of routes is to use larger Xilinx chips than necessary (for example, using an XC3042 instead of an XC3030). In the case of the RAM passing five stage decoder, if the module calculating the error-location (delta stage) polynomial were

implemented on two XC3064's, RAM with an access time of less than 140 ns is used, and the remaining stages are implemented on chips one size larger than necessary, the decoder could operate at a clock speed of 7 MHz instead of 5 MHz. The RAM passing and the direct passing decoder's speed, especially for low rate codes, is limited mainly by the module calculating the polynomial $\delta(X)$. As this is also the most complex module, a significant improvement to the design would be made if a more efficient and simpler algorithm could be found for this stage. The modular design of the decoder should make it possible to change this stage without altering the remainder of the design.

4.7 Conclusion

The major goal of the designs presented here is to provide a basis for a modular Reed-Solomon Codec. An algorithm was chosen which could be separated into five tasks, and implemented on five independent modules, using a Programmable Gate Array (PGA) platform. Further, each module is built from a library of components common to the modules (for example, the Galois Field Arithmetic units, multiplexors, etc.) The algorithm and platform chosen have several disadvantages:

1) The clock speed is different for different number of correctable errors.
2) The δ polynomial stage is considerably larger, more complex, and slower than the other stages.
3) The complete decoder cannot be implemented on a single chip.
4) The PGA chosen does not allow clock speeds of higher than 5MHz.

However, several important development advantages result from the modular design.

1) Inefficient modules may be improved without having to redesign other modules.
2) If an installed base of the codec existed, an improved modules could replace an original module for less than the cost of a new codec.

3) The library of components can be easily laid out on the Xilinx chip. Improvements on a component are easily transfered to existing designs.
4) The modules can be debugged on a component basis.
5) Many applications will not require a codec with the complete range of correctable errors. The design presented here is efficient for high rate codes ($te < 5$).

Furthermore, as Programmable Gate Array Technology advances, many the disadvantages of the design will decrease in importance, while the advantages of a modular design will increase.

References

1. A.M.Michelson and A.H.Levesque, *Error-Control Techniques for Digital Communications*, John Wiley & Sons, 1985.

2. Shu Lin and D.J.Costello,Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, 1983.

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

4. E.R.Berlekamp, *Algebraic Coding Theory*, McGraw-Hill Book Co. Inc., 1968.

5. J.L.Massey, "Shift-Register Synthesis and BCH Decoding", *IEEE Transactions on Information Theory*, Vol. IT-15, pp. 122-127, 1969.

6. R.E.Blahut,*Theory and Practice of Error Control Codes*, Addison-Wesley, 1983.

7. R.E.Blahut, "Transform Techniques for Error Control Codes", *IBM Journal of Research and Development*, Vol. 23, No. 3, May 1979.

8. E.R.Berlekamp, "The Technology of Error-Correcting Codes", *Proceedings of the IEEE*, Vol. 68, No.5, May 1980.

9. H.M.Shao, T.K.Truong, L.J.Deutsch, Y.H.Yuen, I.S.Reed, "A VLSI Design of a Pipeline Reed-Solomon Decoder", *IEEE Transactoions on Computers*, Vol.C-34, No.5, May 1985.

10. H.M.Shao and I.S.Reed, "On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays", *IEEE Transactions on Computers*, Vol.37, No.10., Oct.1988.

11. A.Katsaro, "A programmable decoder for Reed-Solomon codes", *International Journal of Electronics*, Vol.64, No.4,

1988.

12. Y.R.Shayan, T.Le-Ngoc, and V.K.Bhargava, "A Versatile Time-Domain Reed-Solomon Decoder", *IEEE Journal of Selected Areas in Communications*, Vol.8, No.8, Oct. 1990.

13. R.E.Blahut, "A universal Reed-Solomon decoder", *IBM Journal of Research and Development*, Vol.28, No.2, March 1984.

14. S.L.Hurst, *Custom-Specific Integrated Circuits:Design and Fabrication*, Marcel Dekker, Inc., 1985.

15. *The Programmable Gate Array Data Book*, Xilinx Inc., San Jose, Calif., 1988.

16. "1988 Semi Custom Design Guide", *VLSI System Design*, CMP Publications, 1988.

17. T.R.Blakeslee, *Digital Design with Standard MSI & LSI*, 2nd.Ed., John Wiley & Sons, 1979.

18. R.Freeman, "User-Programmable Gate Arrays", *IEEE Spectrum*, Dec.1988.

19. *The Programmable Gate Array Design Handbook*, Xilinx Inc., San Jose, Calif., 1988.

20. W.W.Peterson and E.J.Weldon,Jr., *Error-Correcting Codes*, MIT Press, 1972.

21. F.J.MacWilliams and N.J.A.Sloane, *The Theory of Error-correcting Codes*, North-Holland, 1977.

22. G.Birkhoff and S.MacLane, *A Survey of Modern Algebra*, Macmillan, 1953.

23. J.L.Massey and J.K.Omura, Patent Application of *Computational Method and Apparatus for Finite Field Arithmetic*, submitted in 1981.

24. 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(2m)", *IEEE Transactions on Computors*, Vol. C-34, pp.709-717, Aug.1985.

25. Y.R.Shayan and T.Le-Ngoc, "The Least Complex Parallel Massey-Omura Multiplier and its LCA & VLSI Designs", *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp.408-410, June 1989.

26. M.H.Kang, "Software Algorithms for the Reed-Solomon Code", M.S.E.E. Thesis, University of Illinois, Urbana-Champaign, 1985.

27. R.S.Brunton, "A Comparative Analysis of Reed-Solomon Decoding Techniques", M.A.Sc. Thesis, University of Waterloo, Waterloo, Ont., Aug. 1981.

28. C.Mead and L.Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, 1980.

29. B.W.Kernighan and D.Ritchie, *The C Programming Language*, Prentice Hall Software Series, 1988.

30. *P/C Silos User's Manual*, Simucad Inc., Menlo Park, Calif., 1988.

Appendix A: C Implemtation of (31,k) RS Codec

```
/*************************************************************

PROGRAM
        RS_decoder

USAGE
        RS_decoder  <field size>

        where <field size> may be 16, 32, 64, 128 or 256.
        If no field size is given, a field size of 32 is assumed.

FUNCTION
        RS_decoder reads the number of correctable errors k, then
        reads and decodes a (31,k) Reed-Solomon code word, outputting
        the k GF(32) symbols making the information message. The
        program continues until the number of errors to be corrected
        is read as 99.

        The corrected code word is left in received[], where the k high
        order symbols are the information message, and the 31-k low
        order symbols are the the parity check symbols.

        The power notation is used to represent elements of GF(32);
        that is, alpha**i <-> i, with the exception of 0 in GF(32),
        which is represented by the integer 31.

EXAMPLE
        info_source | RS_encoder | noise_source | RS_decoder > info_file

*************************************************************/

#include <stdio.h>
#Include "arith16.c"
#include "arith32.c"
#include "arith64.c"
#include "arith128.c"
#include "arith 256.c"

# define END_CHAR 99

int RS_word_read(), find_syndrome(), find_delta(), find_location();
int find_z(), find_error();
int t_error;                          /* the number of correctable errors */
int lmu;                                        /* the order of the delta
polynomial */
int number_of_errors;          /* the number of errors found */
int small_array_size, (*gfadder)(), (*gfmulter)(), (*gfinverter)();
int gfsize;
int *received;          /* the received coded word */
```

```
int *syndrome;              /* the syndrome components */
int *delta;                          /* the error-location polynomial */
int *location_of_errors; /* the location of the errors */
int *z_of_x;                    /* Z(X), an intermediary array */
int *error;                          /* the errors at the locations */

int main(argc, argv)
int argc;
char **argv;
{
        int index, chk;

        if (argc == 2) {
                gfsize = atoi(argv[1]);
                switch (gfsize) {
                        case 16:
                                gfadder = add16;
                                gfmulter = mult16;
                                gfinverter = inv16;
                                num_symbols = 15;
                                break;
                        case 32:
                                gfadder = add32;
                                gfmulter = mult32;
                                gfinverter = inv32;
                                num_symbols = 31;
                                break;
                        case 64:
                                gfadder = add64;
                                gfmulter = mult64;
                                gfinverter = inv64;
                                num_symbols = 63;
                                break;
                        case 128:
                                gfadder = add128;
                                gfmulter = mult128;
                                gfinverter = inv128;
                                num_symbols = 127;
                                break;
                        case 256:
                                gfadder = add256;
                                gfmulter = mult256;
                                gfinverter = inv256;
                                num_symbols = 255;
                                break;
                        default :
                                fprintf(stderr, "Allowable GF Sizes:
16,32,64,128,256\n");
                                exit(1);
                }
        } else {
                gfadder = add32;
                gfmulter = mult32;
```

```
        gfinverter = inv32;
        num_symbols = 31;
}
small_array_size = num_symbols/2 + 1;
if ((received = (int *)calloc(num_symbols,sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for received\n");
        exit(1);
}
if ((syndrome = (int *)calloc(num_symbols,sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for syndrome\n");
        exit(1);
}
if ((delta = (int *)calloc(small_array_size,sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for delta\n");
        exit(1);
}
if ((location_of_errors = (int *) calloc (small_array_size,

                                        sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for location_error\n");
        exit(1);
}
if ((z_of_x = (int *)calloc(small_array_size, sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for z_of_x\n");
        exit(1);
}
if ((error = (int *)calloc(small_array_size, sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for error\n");
        exit(1);
}
scanf("%d",&t_error);           /* t_error = # correctable errors */
printf("%d\n",t_error);

while (t_error != END_CHAR) {
        two_t = 2 * t_error;
        if ((chk = RS_word_read()) != 0) {
                        printf("Error in RS_word_read\n");
                exit(1);
}
if ((chk = find_syndrome()) != 0) {
                printf("Error in find_syndrome\n");
                exit(1);
}
if ((chk = find_delta()) != 0) {
                printf("Error in find_delta\n");
                exit(1);
}
if ((chk = find_location()) != 0) {
                printf("Error in find_location\n");
                exit(1);
}
if ((chk = find_z()) != 0) {
                printf("Error in find_z\n");
```

```
                exit(1);
        }
        if ((chk = find_error()) != 0) {
                printf("Error in find_error\n");
                exit(1);
            }
        /* print out the information message */
            for (index = 2*t_error; index < num_symbols; index++)
                printf("%d ",received[index]);
            scanf("%d",&t_error);
            printf("\n%d\n",t_error);

        }
        return 0;

}
```

```
/***********************************************************

Program  RS_word_read.c

Function
        word_read reads in the num_symbols characters in the (31,k)
        code word to be decoded by RS_decoder.
*/

int RS_word_read()
{
        register int index;
        int *received_ptr;;


        received_ptr = received+num_symbols-1;
        for(index = num_symbols; index >= 1; index--)
                scanf("%d",received_ptr--);
        return 0;
}

/***********************************************************

Program RS_syndrome.c

Function
        RS_syndrome calculates the syndrome  components S[index]
        for the Programmable Reed-Solomon (n,k) codec program
        decoder.

                If t_error is the number of correctable errors, there
                are 2*t_error syndrome components.

                The calculation uses the formula (example for GF(32):

        r(X) = (...((r30 * a + r29)*a + r28)*a + ... + r1)*a + r0

                where a denotes alpha**i, and r2 denotes the 2nd component
                of the received code word (ie, the 3rd received symbol).

                The elements of GF(32) are expressed in terms of the power
                notation.  Thus 1 = alpha**0 <-> 0, alpha**1 <-> 1, and
                0 <-> 31.

*/

int find_syndrome()
{
        register int alpha,order;

        two_t = 2 * t_error;
        syndrome[0] = 0;
```

```
        for (alpha = 1; alpha <= two_t; alpha++) {
                syndrome[alpha] = received[num_symbols-1];
                for (order = num_symbols-2; order >= 0; order--)
                        syndrome[alpha] = (*gfadder)(received[order],

        (*gfmulter)(syndrome[alpha],alpha));
        }
        return 0;
}

/********************************************************

Program RS_delta.c

Function
        RS_delta finds the function delta(X).  Berlekamp's
        algorithm, as found in "Error Control Coding", chapter 6,
        by Lin & Costello, is used.

        Notes:
                - two_t is 2*t_error.
                - the lines are numbered from 0 on instead of
                  from -1 on as in L&C; that is,
                  mu is mu + 1, needed because negative indices
                  aren't possible in C.
                -delta[] will hold final delta polynomial.
                -shifted_delta holds the polynomial
                  dmu * inv(dp) * X**(mu-p) * delta[p](X)
                -shift_order plays a double role:
                =       it holds the order of the lowest order term of:
                        delta[p](X) * X**(mu-p), ie holds mu - p
                =       after the shifted_delta is found, it holds the order
                        of the shifted_delta polynomial, ie it becomes l_s
*/

int dmu, dmu_1, lmu_1;
int p, l_p, d_p;
int *delta_mu_1,  *delta_p, *shifted_delta;
if  ((delta_mu_1 = (int *)calloc(small_array_size, sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for delta_mu_1\n");
        exit(1);
        }
if  ((delta_p = (int *)calloc(small_array_size, sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for delta_p\n");
        exit(1);
}
if  ((shifted_delta = (int *)calloc(small_array_size, sizeof(int))) == NULL)
        fprintf(stderr,"Unable to allocate space for shifted_delta\n");
        exit(1);
}

int find_delta()
{
```

```
        int index,  shift_order, d_fact, mu;
        register int subscript, counter;

        d_p = l_p = lmu = 0;
        delta_p[0] = delta[0] = 0;
        p = -1;
        for (index = 1; index <= t_error; index++) {
                delta_p[index] = num_symbols;           /* equals 0 in power notation */
                delta[index] = num_symbols;
        }
        dmu = syndrome[1];
        for (mu = 0; mu < 2*t_error;mu++) {
                dmu_1 = dmu;
                lmu_1 = lmu;
                for (index = 0; index <= t_error; index++)
                        delta_mu_1[index] = delta[index];
                if (dmu != num_symbols)  {
                        for (index = 0; index <= t_error; index++)
                                shifted_delta[index] = num_symbols;
                        shift_order = mu - p;
                        d_fact = (*gfmulter) (dmu, (*gfinverter)(d_p));
                        for (index = shift_order; index <= l_p + shift_order; index++)
                                shifted_delta[index] = (*gfmulter)(d_fact,

        delta_p[index - shift_order]);
                        for (subscript = 0; subscript <= t_error; subscript++)
                                delta[subscript] = (*gfadder)(delta[subscript],

        shifted_delta[subscript]);
                        if (lmu < (l_p+shift_order))
                                lmu = l_p + shift_order;
                        if (dmu_1 != num_symbols && (p-l_p < mu-lmu_1)) {
                                p = mu;
                                d_p = dmu_1;
                                l_p = lmu_1;
                                for (index = 0; index <= t_error; index++)
                                        delta_p[index] = delta_mu_1[index];
                        }
                }
                dmu = syndrome[mu+2];
                for (index = mu+1, counter = 1; counter <= lmu; index--,

                                                        counter++) {
                        dmu = (*gfadder)(dmu, (*gfmulter)(syndrome[index],

                                                        delta[counter]));

                }
        }
        return 0;
}

/************************************************************
        Program RS_location.c
```

```
        Function
                RS_location finds the location of the errors in the
                received word received[ ], and stores them in the
                array location_of_error[ ]; called by RS_decoder

                The locations are the roots of delta(alpha); the
                elements of GF(2^m) are substituted into delta using the
                calculation:

    delta(alpha) = (...(delta[last]*alpha + delta[last-1])*alpha +
                        ...+delta[1])*alpha + 1

                The values of alpha giving delta(alpha) = 0 are stored
                as roots in location_of_error.
*/

int find_location()
{
        register int sub_index,alpha;
        int root;

        number_of_errors = 0;
        for (alpha = 0; alpha < num_symbols ; alpha++) {
                root = delta[lmu];
                for (sub_index = lmu - 1; sub_index >= 0; sub_index--)
                        root = (*gfadder)(delta[sub_index],(*gfmulter)(root,alpha));
                if (root == num_symbols) {        /* 0 =num_symbols */
                        location_of_error[number_of_errors] = (*gfinverter)(alpha);
                        number_of_errors++;
                }
                /* if all correctable errors found, leave function */
                if (number_of_errors == t_error) { break;}
        }
        return 0;
}

/***********************************************************
        Program  RS_z_of_x.c

        Function
                RS_z_of_x calculates Z(X) for RS_decoder where Z(X) is
                an intermediate function used in calculating the errors
                in the code symbols at location_of_error[]. The function
                performs the calculation:

        Z(X) = 1 + (S1+delta[1])X + (S2+delta[1]*S1+delta[2])X**2 + ...
                  + (Sv+delta[1]*S[v-1]+delta[2]*S[v-2]+...+delta[v])X**v

                where v is the order of delta(X).

*/
```

```
int find_z()
{
        register int subscript, sub_index;

        z_of_x[0] = 0; /* 1 in GF(2^m) denoted by 0 in pow. notation */

        for (subscript = 1; subscript <= number_of_errors; subscript++) {
                z_of_x[subscript] = num_symbols;

                for (sub_index = 0; sub_index <= subscript; sub_index++) {
                        z_of_x[subscript] = (*gfadder)(z_of_x[subscript],
                (*gfmulter)(syndrome[sub_index], delta[subscript-sub_index]));

                }
        }
        return 0;
}

/************************************************************Program
RS_correction.c

Function
        RS_correction calculates the error at each of error locations
        for the program RS_decoder.  The corrected code word
        is left in received[ ], where the k high order symbols are the
        information message, and the n-k low order symbols are the
        the parity check symbols.
*/

int find_error()
{
        register int index0, index1;
        int temp ,numerator, inverse;

        for (index0 = 0; index0 < number_of_errors; index0++) {

/* find the numerator, store in error[] */
                inverse = (*gfinverter)(location_of_error[index0]);
                error[index0] = z_of_x[number_of_errors];
                for (index1 = number_of_errors-1; index1 >= 0; index1--) {
                        error[index0] = (*gfadder)(z_of_x[index1],

        (*gfmulter) (error[index0],inverse));
                }

/* divide the numerator by the denominator functions, leaving
  the error in the array error[] */
                for (index1 = 0; index1 < number_of_errors; index1++) {
                        if (index1 != index0) {
                                temp = (*gfmulter)(location_of_error[index1],

        (*gfinverter)(location_of_error[index0]));
                                error[index0] =
```

```
                (*gfmulter)((*gfinverter)((*gfadder)(0,temp)), error[index0]);
                        }
                }
        }

/* correct the error at each error location */
        for (index0 = 0; index0 < number_of_errors; index0++) {
                received[location_of_error[index0]] =
        (*gfadder)(received[location_of_error[index0]],error[index0]);
        }
        return 0;
}
```

```
/**************************************************************
---------------------------------------------------------------

PROGRAM
 RS_encoder.c

USAGE
 RS_encoder

FUNCTION
        RS_encoder reads in the number of errors to be corrected        followed by
the required number of information symbols, and        outputs a RS codeword of 31
characters. It expects 31 - 2 *     t_error information characters, and  t_error = 99 as the
stop signal.

   The incoming information digits are the high-order bits of the
   received vector. Thus the first info bit is the X**2t bit
   of the received vector, the second info bit is the X**2t+1...

   this means the info array is filled from postion 2t to 30 with
   the info bits, from low order at 2t to high order at 30

EXAMPLE
 i32_infomaker | e32_encoder | RS_decoder

BUGS
        The need to constantly check for the number of errors to be        corrected is a
development strategy; the program should be   changed so as to only  need a signal
character to change unless very          rapid changing is expected.

*/

#include <stdio.h>
#include "arith32.c"
#include "gen_poly_coef_z.c"
#define END_CHAR 99

int info[31];
int g_[31][30];
int t_error;
int two_t;

int main()
{
        int code_info();
        int index, *info_ptr;

        gen_poly_coef_z();

/*    the first info in info should be t_error              */

        scanf("%d",&t_error);
```

```
        two_t = 2 * t_error;
        while ( t_error != END_CHAR) {


/               info_ptr = index;
                for( index = two_t; index < 31; index++)
                        scanf("%d",info_ptr++);
                code_info();
                scanf("%d",&t_error);
                two_t = 2 * t_error;
        }
        printf("%d\n", END_CHAR);
        return 0;
}

int code_info()
{
        int add32(), mult32();
        int gate, b_[30];
        register int index, sub_ind;

        for( index = 0; index <= 29; index++ )
                b_[ index ] = 31;
        for( index = 30; index > two_t - 1; index--) {
                gate = add32( b_[ two_t - 1 ] , info[ index ] );
                for (sub_ind = two_t - 1; sub_ind > 0; sub_ind-- )
                        b_[sub_ind] = add32 ((mult32 (g_[t_error][sub_ind], gate),
b_[sub_ind-1]);
        }
        b_[0] = mult32 (g_[t_error][0], gate);
        for( index = two_t - 1; index >= 0; index--)
                info[ index ] = b_[index] ;
        printf("%d\n",t_error);
        for( index = 30; index >= 0; index--)
                printf("%d ",info[index]);
        printf("\n");
        return 0;
}
```

```
/*********************************************************
---------------------------------------------

Program arith32.c

Function
        arith32 provides the addition, multiplication and inversion
        functions for GF(32), using the power notation.

*/


int inv32 (elem)
int elem;
{
        int result;
        if  (elem == 31) {
                printf("inversion of zero element");
                result = 31;
  } else {
                result = (31- elem)%31;
        }
        return result;
}


int mult32 (elem0, elem1)
int elem0, elem1;
{
        int result;

        if (elem0 == 31 || elem1 == 31) {
                result = 31;
        } else {
                result = (elem0 + elem1)%31;
 }
        return result;
}

int add32 (el1,el2)
int el1, el2;
{
  int translate(), untranslate();
  int elet1, elet2, vec_sum, result;

  elet1 = translate (el1);
  elet2 = translate (el2);
  vect_sum = elet1 ^ elet2;
  result = untranslate(vect_sum);
  return result;
}
```

```
/* The exact translations from the power to vector and vector to power representations
are given in Appendix D, and so have been
removed from the program listing to save space (and trees) */

int translate(field_element)
int field_element;
{
        int elem;
        switch ( field_element ) {
                case 0x0 :
                        elem = 16;
                        break;
                                ...
                case 0x1f :
                        elem = 0;
                        break;
        }
        return elem;
}

int untranslate(trans)
int trans;
{
        int elem;
        switch ( trans ) {
                case 0x0 :
                        elem = 31;
                        break;
                                ...
                case 0x1f :
                        elem = 15;
                        break;
        }
        return elem;
}
```

Appendix B: Control Logic for (31,k) RS Codec

This Appendix contains the control logic signals for the RAM passing architecture (31,k) RS Codec.

B.1 Encoder

See Figure 4.3.3.

The outer loop is carried out while the contents of *index* and *two_t_1* are not equal. A GF element is read from the *Information Message Buffer*, and sent to the *encoder output*. The GF element is simultaneously passed to the GF Adder, where it is added with the GF element in *bgate* indexed by the decoder *bgate_dec*, and stored in *gate*. The contents of *two_t-2* and *two_t-1* are stored in *sub1* and *sub0*, respectively. The inner loop is then carried out while the contents of *sub0* is not zero. The coefficient g_{ik} is read from the ROM into the GF multiplier, where it is multiplied with the contents of *gate*, and sent to the GF Adder.

```
counter = 0
  reset  bgate[all]
  reset index
  read into comp[1]
  read into comp[0]

counter = 1
  if comp[0] = comp[1]
              goto eight
  GFA1_mux = information message
  read into GFA[0]
  bgate_dec_mux = two_t_2
  read into GFA[1]
  output_mux = information message

counter = 2
  read into gate
  read into sub0
  read into sub1

counter = 3
```

```
  if sub0 = 0
          goto six
  GFM[0] = g_[t_error][sub1]
  GFM[1] = gate

counter = 4
  GFA1_mux = GFmultiplier
  read into GFA[0]
  read into GFA[1]
  bgate_dec_mux = sub1
  bgate_dec = sub1

counter = 5
  bgate_mux = GFadder
  read into bgate
  sub0--
  sub1--
  goto three

counter = 6
  read into GFM[0]
  read into GFM[1]
  index--

counter = 7
  read into comp[0]
  bgate_dec_mux = sub1
  bgate_mux = GFmultiplier
  read into bgate
  goto one

counter = 8
  bgate_dec_mux = index
  output_mux = bgate
  index--

counter = 9
  if  index = 0
          goto end_end
  else
      goto eight

counter = 10
  bgate_dec_mux = index
  output_mux = bgate
```

B.2 DECODER

RAM_codes:
received[31]: 0
received[31]: 1
syndrome[31]: 2
delta[16]: 3
location_of_error[16]: 4
z_of_x[16]: 5
error[16]: 6
t_error: 7 index: 0
lmu: 7 index: 1
number_of_errors: 7 index: 2

B.2.1 Syndrome Stage

See Figure 4.4.3 and Figure 4.4.4.

The operation of the RAM passing architecture and the direct passing architecture are similar, differing only in that the RAM passing architecture searches RAM for the registers *received [ ]* and *syndrome [ ]* and the initial value *t_error*. In the direct passing architecture, *received [ ]* is a cyclic shift register, connected by a 5 bit bus (one GF element wide) to the GF Adder. A new GF element is read from *received [ ]* by performing a 5 bit shift on *received [ ]*. The 4 bits of *t_error* are stored as the upper 4 bits in the comparator, effectively being stored as $2t_e$. The registers *root1*, *root2*, and *index* are initially set to 1, 2, and 30, respectively. Two control loops are used to calculate the syndrome components.

The outer loop, which is continued while *root2* and *t_error* are not equal, starts with *b_1* and the GF element at the transfer bus in *received [ ]* being loaded into the GF Adder. The inner loop then loads the content of *root2* and the GF Adder output into the GF Multiplier. The output of the GF Multiplier is then stored in *b_1*, and the content

of *b_0* is loaded into the GF Adder to be added with the transferable GF element in *received [ ]*. The contents of *root1* is then loaded into the GF Multiplier, together with the GF Adder output. The output of the GF Multiplier is then stored in *b_0*, *index* is decremented, and in the case of the direct passing architecture, *received [ ]* is cyclically shifted by one GF element. If the content of *index* does not equal 0, the inner loop is repeated. Otherwise *index* is reset to 30, and *b_1* and the transferable GF element in *received [ ]* are loaded into the GF Adder. The output of the GF Adder is stored in *syndrome [ ]*, at the address pointed to by *root2*. Then *b_0* is loaded into the GF Adder to be summed with the transferable GF element in *received [ ]*, and the output is stored in *syndrome [ ]*, at the address pointed to by *root1*. If *root2* is equal to *t_error*, the module is completed, otherwise the outer loop is repeated.

```
counter = 0
  unenable_RAM
  reset RAM_addr_mux = 0
  RAM_code = 2
  reset data_to_RAM = 0
  reset RAM_addr_mux = 0
counter = 1
  write_to_RAM
  reset root1
  reset root2
  reset index

counter = 2
  unenable_RAM
  RAM_code = 6

counter = 3
  read_from_RAM
  read into t_error

counter = 4
  reset b_0
  reset b_1
```

```
  b_mux = b_1
  RAM_addr_mux = index
  RAM_code = 0

counter = 5
  read_from_RAM
  read into GFA[0]
  read into GFA[1]
  root_mux = root2

counter = 6
  unenable_RAM
  root_mux = root2
  read into GFM[0]
  read into GFM[1]

counter = 7
  read into b_1
  b_mux = b_0
  read into GFA[0]
  index--

counter = 8
  root_mux = root1
  read into GFM[0]
  read into GFM[1]
  RAM_addr_mux = index
  RAM_code = 0

counter = 9
  read_from_RAM
  read into b_0
  b_mux = b_1
  read into GFA[0]
  read into GFA[1]
  root_mux = root2
  if  index ≠ 0
          goto six


counter = 10
  unenable_RAM
  reset index = 30
  RAM_addr_mux = root_mux
  RAM_code = 2
  read into data_to_RAM

counter = 11
  write_to_RAM

counter = 12
  unenable_RAM
```

```
  b_mux = b_0
  read into GFA[0]

counter = 13
  root_mux = root1
  root2 = root2 + 2
  RAM_addr_mux = root_mux
  RAM_code = 2
   read into data_to_RAM

counter = 14
   write_to_RAM
  root1 = root1 + 2
  if comp_out ≠ 1
          goto four
```

B.2.2 Delta Stage

See Figure 4.4.5 and Figure 4.4.6.

The operation of the RAM passing architecture and the direct passing architecture are similar, differing only in that the RAM passing architecture accesses RAM for the registers *syndrome [ ]* and *delta [ ]*, and the initial value *t_error*. Register *delta [ ]* contains the coefficients of the delta polynomial after mu iterations of the delta stage algorithm, *delta_mu_1 [ ]* contains the coefficients of the delta polynomial after the (mu-1)-th iteration of the delta stage algorithm, *delta_p [ ]* contains the coefficients of the delta polynomial after the p-th iteration of the delta stage algorithm. *dmu*, *dmu_1*, and *d_p* contain the difference after the mu-th, (mu-1)-th, and p-th iterations of the delta stage algorithm. *lmu*, *lmu_1*, and *l_p* contain the order of the delta polynomial after the mu-th, (mu-1)-th, and p-th iterations of the delta stage algorithm. *mu_plus_1* contains mu + 1.

After the required initialization of registers *p*, *d_p*, *l_p*, *lmu*, *delta_p [ ]*, *delta [ ]*, and *dmu*, an outer loop is started, and continued while the contents of *mu* are less than $2t_e$. The values of the registers *dmu*,

lmu, and *delta [ ]* for the previous value of *mu* are saved in *dmu_1*, *lmu_1*, and *delta_mu_1 [ ]*, respectively.

If *dmu* is not equal to 31, new values of *lmu* and *delta [ ]* are calculated. The shift register *shifted_delta [ ]* is zeroed, and *shift_order* is set to the difference between *mu* and *p*. The GF inverse of *d_p* is taken, and loaded into the GF Multiplier with *dmu*. The output of the GF Multiplier is stored in *d_fact*. The GF product of the GF elements in *delta_p [ ]* and *d_fact* are stored in *shift_register [ ]*. After all the GF elements in *delta_p [ ]* have been multiplied, *shift_register [ ]* is shifted by one GF element (5 bits) *shift_order* times. The GF products of the GF elements in *delta [ ]* and *shift_register [ ]* are then stored in *delta [ ]*. If *lmu* is less than the sum of *l_p* and *shift_order*, lmu is set to the sum of *l_p* and *shift_order*. If *dmu_1* is not equal to 31, and the diffence of *p* and *l_p* is less than the difference between *mu* and *lmu_1*, the contents of *dmu_1*, *lmu_1*, and *delta_mu_1 [ ]* are stored in *d_p*, *l_p*, and *delta_p [ ]*, respectively. This completes the calculation of new values of *lmu* and *delta [ ]*.

The GF element in *syndrome [ ]*, at the (mu+1)-th location, is loaded into *dmu*. The difference between *mu_plus_1* and *lmu* is loaded into *cindex*, and the contents of *lmu* is loaded into *subscript*. A loop is then started to calculate *dmu*. The GF product is taken of the GF element in *syndrome [ ]* pointed at by *cindex*, and the GF element in *delta [ ]* pointed at by *subscript*, and loaded into the GF Adder together with the contents of *dmu*. The output of the GF Adder is loaded into *dmu*. *cindex* is incremented, *subscript* is decremented, and the loop is repeated until the content of *subscript* equals zero. The outer loop is

then repeated, unless mu equals $2t_e$.

```
counter = 0
  unenable_RAM
  reset RAM_addr_mux
  RAM_code = 6
  reset mu, mu_plus_1, p, d_p, l_p, lmu, delta_p

counter = 1
  read_from_RAM
  read into t_error
  subscript_mux = data_from_RAM
  read into subscript
  reset subs_zero

counter = 2
  unenable_RAM
  RAM_addr_mux = subscript
  RAM_code = 3
  if subscript = 0
    reset data_to_RAM
   reset subs_zero
  else
    reset data_to_RAM

counter = 3
  write_to_RAM
  subscript--
  if  subs_zero = 1
   goto four
  else
   goto two

counter = 4
  unenable_RAM
  RAM_addr_mux = mu_plus_1
  RAM_code = 2

counter = 5
  read_from_RAM
  dmu_mux = data_from_RAM
  read into dmu
  subscript_mux = t_error
  read into subscript
  reset subs_zero

counter = 6
  unenable_RAM
  RAM_addr_mux = subscript
  RAM_code = 3
  del_dec_mux = subscript
```

```
  if  subscript = 0
    reset subs_zero = 1

counter = 7
  read_from_RAM
  read into delta_mu_1
  subscript--
  if subs_zero = 0
    goto six

counter = 8
  unenable_RAM
  read into dmu_1
  read into lmu_1
  sub0_mux = mu
  sub1_mux = p
  if  mu = 2 * t_error
    reset RAM_addr_mux
    RAM_code = 7
    write_to_RAM
    read into data_to_RAM
    goto end_delta
  if dmu = 31
    goto twenty-four


counter = 9
  reset  shifted_delta[all]
  read into sub[0]
  read into sub[1]
  read into GFI_in

counter = 10
  read into shift_order
  read into add[0]
  read into add[1]
  GFM0_mux = dmu
  GFM1_mux = GFinverter
  read into GFM[0]
  read into GFM[1]

counter = 11
  read into d_fact
  if  lmu < adder_out
    read into lmu
    read into  shifted_delta[all]
  subscript_mux = l_p
  read into subscript

counter = 12
  read into shift_delta_dec
  GFM0_mux = d_fact
  GFM1_mux = shifted_delta
```

```
  read into GFM[0]
  read into GFM[1]
  if subscript = -1
    reset num_shifts
    goto fourteen

counter = 13
  read into shifted_delta
  subscript--
  goto twelve

counter = 14
  if shift_order = 0
    goto sixteen

counter = 15
  shift shift_delta
  shift_order--
  goto fourteen

counter = 16
  unenable_RAM
  subscript_mux = t_error
  read into subscript
  reset subs_zero
  read into comp[1]
  sub0_mux = lmu_1
  sub1_mux = l_p
  RAM_addr_mux = subscript
  RAM_code = 3

counter = 17
  read_from_RAM
  read into shift_delta_dec
  GFA0_mux = data_from_RAM
  GFA1_mux = shifted_delta
  read into GFA[0]
  read into GFA[1]
  if subscript = 0
    reset subs_zero

counter = 18
  write_to_RAM
  read into data_to_RAM
  subscript--
  if subs_zero = 1
    goto twenty

counter = 19
  unenable_RAM
  RAM_addr_mux = subscript
  RAM_code = 3
  goto seventeen
```

```
counter = 20
  unenable_RAM
  read into sub[0]
  read into sub[1]

counter = 21
  read into comp[0]

counter = 22
  if dmu_1 = 31 OR comp[0] ≥ comp[1]
    goto twenty-four

counter = 23
  read into p
  read into d_p
  read into l_p
  read into delta_p

counter = 24
  unenable_RAM
  mu++
  mu_plus_1++
  RAM_addr_mux = mu_plus_1
  RAM_code = 2

counter = 25
  read_from_RAM
  read into dmu
  read into subscript
  sub0_mux = mu_plus_1
  sub1_mux = lmu
  read into sub[0]
  read into sub[1]

counter = 26
  unenable_RAM
  read into cindex
  RAM_code = 3
  RAM_addr_mux = subscript

counter = 27
  read_from_RAM
  GFM1_mux = data_from_RAM
  read into GFM[1]
  if subscript = 0
    read into subscript
    reset subs_zero
    goto six

counter = 28
  unenable_RAM
  RAM_addr_mux = cindex
```

```
  RAM_code = 1

counter = 29
  read_from_RAM
  GFM0_mux = data_from_RAM
  read into GFM[0]

counter = 30
  GFA0_mux = GFmultiplier
  GFA1_mux = dmu
  read into GFA[0]
  read into GFA[1]
  subscript--
  cindex++

counter = 31
  read into dmu
  RAM_addr_mux = subscript
  RAM_code = 3
  goto twenty-seven

end_delta:
```

B.2.3 Location Stage

See Figure 4.4.7 and Figure 4.4.8.

The operation of the RAM passing architecture and the direct passing architecture are similar, differing only in that the RAM passing architecture searches RAM for the registers *delta [ ]*, *lmu*, and *location_of_errors [ ]*. *number_errors* and *alpha* are reset to zero. An outer loop is started, and continued until *alpha* has held all the non-zero elements of GF(2^5). *root* is loaded with the GF element in *delta [ ]* pointed at by *lmu*. *lmu* is loaded into *index*, which is decremented once, and an inner loop is repeated while *index* is greater than or equal to zero. The GF product of *root* and *alpha* is loaded into the GF Adder, together with the GF element in *delta [ ]* pointed to by *index*. The output of the GF Adder is stored in *root*, *index* is decremented, and the loop is repeated. After the inner loop ends, if the content of

root is 31, the GF inverse of *alpha* is stored in *location_of_errors [ ]*, and *number_errors* is incremented. *alpha* is then incremented, and the outer loop is repeated.

```
counter = 0
  unenable_RAM
  reset number_of_errors
   reset alpha
  reset RAM_addr_mux
  RAM_code = 7

counter = 1
  read_from_RAM
  read into lmu = data_from_RAM

counter = 2
  unenable_RAM
  RAM_addr_mux = lmu
  RAM_code = 3
  read into index

counter = 3
  read_from_RAM
  root_mux = data_from_RAM
  read into root
  if alpha = 31
    goto end_loc
  if index = 0
           goto eight

counter = 4
  index--
  reset index_zero

counter = 5
  unenable_RAM
  RAM_addr_mux = index
  RAM_code = 3
  read into GFM[0]
  read into GFM[1]
  if index = 0
    reset index_zero

counter = 6
  read_from_RAM
  read into GFA[0]
  read into GFA[1]

counter = 7
  unenable_RAM
```

```
  RAM_addr_mux = index
  RAM_code = 3
  root_mux = gf_adder_out
  read into root
  index--
  if index_zero
    goto eight
  goto five

counter = 8
  unenable_RAM
  read into GFI_in
  RAM_addr_mux = number_of_errors
  RAM_code = 4
  if root ≠ 31
    goto ten

counter = 9
  write_to_RAM
  RAM_data_mux = gf_inverter_out
  number_of_errors++

counter = 10
  unenable_RAM
  alpha++
  RAM_addr_mux = 2
  RAM_code = 7

counter = 11
  write_to_RAM
  RAM_data_mux = number_of_errors
  goto two

end_loc:
```

B.2.4 Z(X) Stage

See Figure 4.4.9 and Figure 4.4.10.

The operation of the RAM passing and the direct passing architectures are similar, differing only in that the RAM passing architecture searches RAM for the registers *syndrome [ ]*,*delta [ ]*, and *z_of_x [ ]*. *z_of_x [ ]* and *subscript* are reset, and a loop is started, which continues until the content of *subscript* is greater than the number of errors. The GF element in *z_of_x [ ]* pointed to by *subscript* is set to 31, and *index* is set to zero. An inner loop is started, and continued

while the contents of *index* is less than the contents of *subscript*. The GF Multiplier is loaded with the GF element in *syndrome [ ]* pointed to by *index*, and the GF element in *delta [ ]* pointed to by the difference of the contents of *subscript* and *index*. The output of the GF Multiplier is then loaded into the GF Adder, together with the GF element in *z_of_x [ ]* pointed to by *subscript*. The GF Adder output is then stored in *z_of_x [ ]*, at the location pointed to by *subscript*. *index* is then incremented, and the inner loop is repeated. After the inner loop completes, *subscript* is incremented, and the outer loop is repeated.

```
counter = 0
  reset subscript
  unenable_RAM
  RAM_code = 5
  RAM_addr_mux = subscript
   reset data_to_RAM

counter = 1
   write_to_RAM

counter = 2
  unenable_RAM
  subscript++

counter = 3
  unenable_RAM
  RAM_code = 5
  RAM_addr_mux = subscript
   reset data_to_RAM

counter = 4
   write_to_RAM
  reset index
  if subscript > number_of_errors
    goto end_z

counter = 5
  unenable_RAM
  RAM_code = 3
  RAM_addr_mux = subber_out

counter = 6
  read_from_RAM
```

```
 read into GFM[0]

counter = 7
 unenable_RAM
 RAM_code = 2
 RAM_addr_mux = index

counter = 8
 read_from_RAM
 read into GFM[1]
 if index > subscript
   goto twelve

counter = 9
 unenable_RAM
 RAM_code = 4
 RAM_addr_mux = subscript

counter = 10
 read_from_RAM
 read into GFA[0]
 read into GFA[1]

counter = 11
 write_to_RAM
 RAM_data_mux =  gf_adder_out
 index++
 goto five

counter = 12
 subscript++
 goto three

end_z:
```

B.2.5 Correction Stage

See Figure 4.4.11 and Figure 4.4.12.

The operation of the RAM passing and the direct passing architectures are similar, differing only in that the RAM passing architecture searches RAM for the registers *received [ ]*, *error [ ]*, *z_of_x [ ]*, and *location_of_errors [ ]*. *index0* is zeroed, and an outer loop is started. The contents of *location_of_error [ ]* pointed to by *index0* is loaded into the GF Inverter, and the GF element in *z_of_x [ ]* pointed to

by *number_of_errors* is loaded into *error [ ]* at the location pointed to by *index0*. *index1* is set to *number_of_errors*, and decremented.

An inner loop is started by loading the output of the GF Inverter and the GF element in *error [ ]* pointed to by *index0* into the GF Multiplier. The output of the GF Multiplier and the GF element in *z_of_x [ ]* pointed to by *index1* are loaded into the GF Adder, and the output of the GF Adder is loaded into *error [ ]* at the location pointed to by *index0*. *index1* is decremented, and the inner loop is repeated while *index1* is greater than or equal to zero.

index1 is reset to zero, and a second inner loop is started. If the contents of *index0* and *index1* are equal, *index1* is incremented and the inner loop is repeated while the contents of *index1* is less than the *number of errors*. Otherwise, the content of *location_of_errors [ ]* pointed to by *index0* is loaded into the GF Inverter. The output of the GF Inverter is loaded into the GF Multiplier, together with the content of *location_of_errors [ ]* pointed to by *index1*. The output of the GF Multiplier is loaded into the GF Adder, together the integer 0. The output of the GF Adder is loaded into the GF Inverter. The product of the output of the GF Inverter and the GF element in *error [ ]* pointed at by *index0* is then stored in *error [ ]* at the location pointed at by *index0*. *index1* is incremented, and the inner loop is repeated while the contents of *index1* is less than the number of errors. *index0* is then incremented, and the outer loop is repeated while the contents of *index0* is greater than the number of errors.

index0 is then reset to 0, and the GF element in *received [ ]* pointed to by the content of *location_of_error [ ]* pointed to by *index0* is loaded

into the GF Adder, together with the GF element in *error [ ]* pointed to by *index0*. The output of the GF Adder is stored in *received [ ]*, at the location pointed to by the content of *location_of_errors [ ]* pointed to by *index0*. *index0* is incremented, and the loop is repeated until *index0* equals the number of errors. After this loop, *received [ ]* holds the corrected code word.

```
counter = 0
  unenable_RAM
  RAM_addr_mux = 2
  RAM_code = 7

counter = 1
  read_from_RAM
  number_of_errors = data_from_RAM
  reset index0
  reset index1_zero

counter = 2
  unenable_RAM
  RAM_addr_mux = index0
  RAM_code = 4
  if index0 = number_of_errors
    goto twenty-one

counter = 3
  read_from_RAM
  GFI_mux = data_from_RAM
  read into GFI_in
  read into index1

counter = 4
  unenable_RAM
  RAM_addr_mux = number_of_errors
  RAM_code = 5

counter = 5
  read_from_RAM
  read into temp_error
  index1--

counter = 6
  unenable_RAM
  RAM_addr_mux = index1
  RAM_code = 5
  GFM0_mux = temp_error
```

```
  read into GFM[0]
  read into GFM[1]
  if index1 = 0
    reset index1_zero

counter = 7
  read_from_RAM
  GFA1_mux = data_from_RAM
  read into GFA[0]
  read into GFA[1]

counter = 8
  temp_mux = GFadder
  read into temp_error
  if index1_zero
    goto nine
  index1--
  goto six

counter = 9
  unenable_RAM
  reset index1_zero = 0
  RAM_addr_mux = index0
  RAM_code = 4
  if index1 = number_of_errors
    goto eighteen
  if index1 = index0
    goto seventeen

counter = 10
  read_from_RAM
  GFI_mux = data_from_RAM
  read into GFI_in

counter = 11
  unenable_RAM
  RAM_addr_mux = index1
  RAM_code = 4

counter = 12
  GFM0_mux = data_from_RAM
  read into GFM[0]
  read into GFM[1]

counter = 13
  GFA0_mux = GFmultiplier
  read into GFA[0]
  reset GFA[1]

counter = 14
  GFI_mux = GFadder
  read into GFI_in
```

```
counter = 15
  GFM0_mux = GFinverter
  read into GFM[0]
  read into GFM[1]

counter = 16
  temp_mux = GFmultiplier
  read into temp_error

counter = 17
  index1++
  goto nine

counter = 18
  unenable_RAM
  RAM_addr_mux = index0
  RAM_code = 6

counter = 19
  write_to_RAM
  RAM_data_mux = temp_error

counter = 20
  index0++
  goto two

counter = 21
  reset index0

counter = 22
  unenable_RAM
  RAM_addr_mux = index0
  RAM_code = 6
  if index0 = number_of_errors
    goto end_error

counter = 23
  read_from_RAM
  GFA0_mux = data_from_RAM
  read into GFA[0]

counter = 24
  unenable_RAM
  RAM_addr_mux = index0
  RAM_code = 4

counter = 25
  read_from_RAM
  r_dmp = data_from_RAM

counter = 26
  unenable_RAM
  RAM_addr_mux = r_dmp
```

```
  RAM_code = 0

counter = 27
  read_from_RAM
  read into GFA[1]

counter = 28
  write_to_RAM
  RAM_data_mux = GFadder
  index0++
  goto twenty-two
end_error:
```

Appendix C: Listing of Components

The components used in the design for the (31,k) RS Encoder and Decoder are listed below. The components are divided into three tables for each of the modules. The first table lists the Registers used, the width of the register in bits, the number of CLB's required to implement the register in the Xilinx XC3000 Series PGA's, and the features the register must have in addition to storage of data. For instance, the register bgate in the encoder includes a 5-32 Decoder, and each bit of bgate can be reset. The second table lists the muxes required by the module, the number of inputs in the mux, and the number of CLB's required to implement the mux in the Xilinx XC3000 Series PGA's. The third table lists additional components, and the number of CLB's required to implement the component in the Xilinx XC3000 Series PGA's. Components marked with d* or r* are only required in the direct passing architecture or RAM passing architecture, respectively.

C.1 Encoder Components

Register	Bits	CLB's	Features
bgate	150	150	5-32 Decoder & reset
sub0	5	6	decrement & reset
sub1	5	6	decrement & reset
index	5	6	decrement & reset
t_error	4	2	
two_t_1	4	4	decrement by 1
two_t_2	4	4	decrement by 2
gate	5	2.5	

Table C.1.1: Encoder registers, their sizes and features.

Mux	Inputs	CLB's
bgate_dec_mux	3	5
bgate_mux	2	2.5
GFA1_mux	2	2.5

Table C.1.2: Encoder muxes, and their sizes.

Component	CLB's
GF Adder	18
GF Multiplier	15
= Comparator	1.5
Control Logic	16

Table C.1.3: Other Encoder components, and their sizes.

C.2 Syndrome Components

Register	Bits	CLB's	Features
d*syndrome	155	155	5-32decoder
root1	5	6	increment by 2 & reset
root2	5	6	increment by 2 & reset
index	5	6	increment & reset
t_error	4	2	

Table C.2.1: Syndrome registers, their sizes and features.

Mux	Inputs	CLB's
b_mux	2	2.5
root_mux	2	2.5
r*RAM_data_mux	2	2.5
r*RAM_addr_mux	2	2.5

Table C.2.2: Syndrome muxes, and their sizes.

Component	CLB's
= Comparator	1.5
GF Multiplier	15
GF Adder	18
p*Control Logic	18
r*Control Logic	21

Table C.2.3: Other Syndrome components, and their sizes.

C.3 Delta Components

Register	Bits	CLB's	Features
p*delta	80	77.5	4-16 decoder
delta_mu_1	80	77.5	4-16 decoder
delta_p	80	40	
shifted_delta	80	77.5	4-16 decoder & cyclic shift
mu	5	6	increment & reset
mu_plus_1	5	6	increment & reset
subscript	5	6	decrement & reset
cindex	5	6	increment & reset
shift_order	5	6	decrement
p	5	2.5	reset
d_fact	5	2.5	
lmu	5	2.5	reset
lmu_1	5	2.5	
l_p	5	2.5	reset
dmu	5	2.5	
dmu_1	5	2.5	
d_p	5	2.5	reset
t_error	4	2	
sub_zero	1	1	two reset values

Table C.3.1: Decoder registers, their sizes and features.

Mux	Inputs	CLB's
p*syn_dec_mux	3	5
subscript_mux	3	5
dmu_mux	2	2.5
GFA0_mux	2	2.5
GFA1_mux	2	2.5
GF0_mux	3	5
GFM1_mux	3	5
sub0_mux	3	5
sub1_mux	3	5
r*RAM_data_mux	4	8
r*RAM_addr_mux	4	8

Table C.3.2: Decoder muxes, and their sizes.

Component	CLB's
GF Adder	18
GF Multiplier	15
GF Inverter	4
Adder	4.5
Subtractor	4.5
= Comparator	1.5
p*Control Logic	30
r*Control Logic	40

Table C.3.3: Other Decoder components, and their sizes.

C.4 Location Components

Register	Bits	CLB's	Features
p*location_of_errors	80	77.5	4-16 decoder
index	5	6	decrement & reset
alpha	5	6	increment & reset
root	5	2.5	
number_of_errors	5	5	increment & reset
lmu	5	2.5	

Table C.4.1: Location registers, their sizes and features.

Mux	Inputs	CLB's
root_mux	2	2.5
p*del_dec_mux	2	2.5
r*RAM_addr_mux	3	5
r*RAM_data_mux	2	2.5

Table C.4.2: Location muxes, and their sizes.

Component	CLB's
GF Multiplier	15
GF Adder	18
GF Inverter	4
p*Control Logic	15
r*Control Logic	18

Table C.4.3: Other Location components, and their sizes.

C.5. Z(X) Components

Registers	Bits	CLB's	Features
z_of_x	80	77.5	4-16 Decoder & reset
number_of_errors	4	2	
index	5	6	increment & reset
subscript	5	6	increment & reset
zero	1	1	

Table C.5.1: Z(X) registers, their sizes and features.

Mux	Inputs	CLB's
p*z_mux	2	2.5
r*RAM_addr_mux	3	5
r*RAM_data_mux	2	2.5

Table C.5.2: Z(X) muxes, and their sizes.

Component	CLB's
GF Multiplier	15
GF Adder	18
> Comparator0	2.5
> Comparator1	2.5
Subractor	4
p*Control Logic	14
r*Control Logic	19

Table C.5.3: Other Z(X) components, and their sizes.

C.6 Correction Components

Register	Bits	CLB's	Features
p*error	80	77.5	4-16 decoder
index0	5	6	increment & reset
index1	5	8	incr, decr & reset
temp_err	5	2.5	
r_dec	5	2.5	
number_of_errors	4	2	
index_zero	1	1	

Table C.6.1: Correction registers, their sizes and features.

Mux	Inputs	CLB's
root_mux	2	2.5
GFM0_mux	2	2.5
GFA0_mux	2	2.5
p*GFA1_mux	2	2.5
GFI_mux	2	2.5
r*temp_err_mux	2	2.5
rec_dec_mux	2	2.5
z_dec_mux	2	2.5
err_mux	2	2.5
loc_dec_mux	2	2.5
r*RAM_addr_mux	4	8
r*RAM_data_mux	2	2.5

Table C.6.2: Correction muxes, and their sizes.

Component	CLB's
GF Multiplier	15
GF Adder	18
GF Inverter	4
= Comparator0	1.5
= Comparator1	1.5
p*Control Logic	30
r*Control Logic	36

Table C.6.3: Other Correction components, and their sizes.

Appendix D: Power - Vector Representations

The mappings between the power representation and the vector representation, using the minimal polynomial $P(X) = X^5 + X^2 + 1$, are given.

Power Representation to Vector Representation:

$a_0 = (\sim a_0 \otimes a_1 \otimes \sim a_2 + (a_0 \oplus a_1) \otimes a_2) \otimes (a_3 \oplus a_4) + (\sim(a_0 \oplus a_1) + \sim a_0) \otimes a_2 \otimes a_3 \otimes a_4 + a_0 \otimes (\sim a_2 \otimes \sim a_3 + a_1 \otimes a_3 \otimes \sim a_4)$

$a_1 = (a_0 \oplus a_1) \otimes a_2 \otimes (\sim a_3 + a_3 \otimes a_4) + \sim a_2 \otimes \sim a_3 \otimes (\sim a_0 \otimes a_1 + a_0 \otimes \sim a_4) + (\sim a_0 \otimes a_2 \otimes \sim a_4 + \sim(a_0 \oplus a_1) \otimes \sim a_2 \otimes a_4)$

$a_2 = \sim((a_0 \oplus a_1) \oplus a_4) \otimes \sim a_2 \otimes a_3 + (a_1 \otimes \sim a_2 + a_0 \otimes a_2) \otimes \sim a_3 \otimes \sim a_4 + a_2 \otimes (\sim a_0 \otimes (a_1 + \sim a_1 \otimes a_4) + a_0 \otimes \sim a_1 \otimes a_3)$

$a_3 = \sim((a_0 \oplus a_1) \oplus a_4) \otimes a_2 \otimes a_3 + \sim a_2 \otimes ((\sim a_1 + a_1 \otimes \sim a_4) \otimes a_0 + \sim a_0 \otimes \sim a_3 \otimes a_4 + a_1 \otimes (\sim a_2 \otimes a_3 \otimes a_4 + a_2 \otimes \sim a_3 \otimes \sim a_4)$

$a_4 = a_0 \otimes (\sim a_2 \otimes (a_3 \otimes \sim a_4 + \sim a_1 \otimes \sim a_3 + a_1 \otimes a_4) + a_2 \otimes \sim a_1 \otimes a_3) + \sim a_0 \otimes (a_1 \otimes a_3 + \sim a_2 \otimes \sim a_3 \otimes \sim a_4) + \sim(a_0 \oplus a_1) \otimes a_2 \otimes \sim a_3 \otimes a_4$

Vector Representation to Power Representation:

$a_0 = a_0 \otimes (a_1 \otimes (\sim a_2 \otimes (a_3 \oplus a_4)) + a_2 \otimes (a_3 \otimes \sim a_3 \otimes \sim a_4) + \sim a_0 \otimes (a_1 \otimes \sim a_3 \otimes \sim a_4 + \sim a_2 \otimes \sim(a_3 \oplus a_4) + \sim a_1 \otimes a_2 \otimes a_4) + \sim a_1 \otimes \sim a_2 \otimes a_3$

$a_1 = a_0 \otimes a_2 \otimes (a_1 + a_4) + \sim a_2 \otimes (\sim(a_0 \oplus a_1) \otimes \sim a_4 + (a_0 \oplus a_1) \otimes \sim a_3 \otimes a_4) + \sim a_0 \otimes (\sim a_4 \otimes (a_2 \oplus a_3) + \sim a_1 \otimes a_2 \otimes a_3)$

$a_2 = \sim a_0 \otimes ((a_3 \oplus a_4) \otimes (a_2 + a_1 \otimes \sim a_2) + \sim a_1 \otimes (\sim a_2 \otimes \sim a_3 + a_3 \otimes a_4)) + a_3 \otimes ((a_0 \oplus a_1) \otimes \sim a_2 \otimes \sim a_4 + a_0 \otimes a_1 \otimes a_2) + a_0 \otimes \sim a_1 \otimes (a_2 \otimes \sim a_3 + a_3 \otimes \sim(a_2 \oplus a_4))$

$a_3 = a_2 \otimes a_3 \otimes a_4 + (a_0 \oplus a_1) \otimes (\sim a_2 \otimes a_4 + a_2 \otimes a_3) + \sim a_0 \otimes \sim a_3 \otimes (\sim a_1 \otimes \sim a_2 \otimes \sim a_4 + a_1 \otimes a_4) + a_0 \otimes (\sim a_2 \otimes a_3 \otimes a_4 + a_1 \otimes a_2 \otimes \sim a_3)$

$a_4 = a_0 \otimes (\sim a_2 \otimes (a_1 + a_3) + a_2 \otimes \sim a_2 \otimes a_4) + \sim(a_0 \oplus a_1) \otimes (\sim a_2 \otimes \sim a_3 \otimes \sim a_4 + a_3 \otimes (a_2 \oplus a_4)) + \sim a_0 \otimes a_1 \otimes (a_2 \otimes (\sim a_3 + a_3 \otimes a_4) + \sim a_2 \otimes \sim a_3 \otimes a_4)$

VITA

Surname: Dravnieks

Given Names: Olaf Olgert Walter

Place of Birth: Saskatoon, Saskatchewan, Canada.

Date of Birth: May 19, 1959.

Education Institutions Attended:

University of Victoria	1988 to 1992
University of Waterloo	1984 to 1986
University of Saskatchewan	1977 to 1984

Degrees Awarded:

B.Eng	Unversity of Saskatchewan	1984
B.Sc.	University of Saskatchewan	1983

PARTIAL COPYWRITE LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:
Logic Cell Array Designs for a (31,k) Reed-Solomon Codec.

Author

OLAF DRAVNIEKS

Jun 29/93