INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A VHDL Code Generator for Reed-Solomon Encoders and Decoders

Vladimir Glavac

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

April 2003

Abstract

A VHDL Code Generator for Reed-Solomon Encoders and Decoders

Vladimir Glavac

Reed-Solomon codes are error correcting codes that are used in many applications such as satellite communications, digital audio tape, and in CDROMs. Such diverse applications call for the use of many different Reed-Solomon codes. The topic of this thesis is the development of a program to produce synthesizable VHDL code for an arbitrary Reed-Solomon encoder or decoder. A novel extension of the Massey-Berlekamp algorithm for solving the key equation is presented. This modified algorithm is a key aspect of the Reed-Solomon decoder designs discussed in this thesis. The details of the design of both RS encoders and decoders are presented in detail. A program written in a high level language was designed so as to generate the VHDL code that corresponds to the algorithms for encoding and decoding. Several encoders and decoders were synthesized for the Xilinx XCV1000 series of field programmable gate arrays (FPGAs). The resulting area and speed metrics are presented for several designs of Reed-Solomon encoders and decoders.

Dedicated to my parents, and my children Emilie and Jeremy.

ACKNOWLEDGEMENTS

I would like to thank Dr. Reza Soleymani for having inspired me, and guided me during my graduate studies and my thesis. I would also like to thank the management of EMS Technologies Canada Limited for having encouraged me to go ahead and do this work. I would especially like to thank all of my friends and colleagues at work who provided me with many hours of lively discussions on digital communications and mathematics. Finally, I would like to thank my parents, for having given all that they could, and who have taught me that perseverance pays off.

Table of Contents

List of Figures

List of Tables

Table of Acronyms

ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
BCH	Bose-Chaudhuri-Hocquenghem
CAD	Computer Aided Design
CPLD	Complex Programmable Logic Devices
DSP	Digital Signal Processing
FEC	Forward Error Correction
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
IP	Intellectual Property
LCM	Least Common Multiple
LSI	Large Scale Integrated Circuit
MSI	Medium Scale Integrated Circuit
PAL	Programmable Array Logic
PCB	Printed Circuit Board
PLD	Programmable Logic Device
RAM	Random Access Memory
RS	Reed-Solomon
SSI	Small Scale Integrated Circuit
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integrated Circuit

1 Introduction

Reed-Solomon codes are error-correcting codes that are used in many applications such as satellite communications, digital audio tape, and in CDROMs. Such diverse applications call for the use of many different Reed-Solomon codes. The modern digital systems designer is then faced with the problem of implementing these different error-correcting codes. The difficulty of this task has been greatly reduced through the use of Field Programmable Gate Arrays (FPGAs) or gate arrays and through the use of a hardware description language, such as VHDL. The approach is to write the functionality of the system in a top-down approach using VHDL, mapping the code into hardware elements through the process of synthesis, and finally to simulate the resulting netlist using a suitable testbench. The synthesized design can then be loaded into an FPGA and used, or a gate array may then be produced.

This approach is facilitated by the use of pre-written sections of design, called cores, and then just "plugging" them into the design. These cores, which provide the complete functionality for desired components, such as, for example, FIR filters, UARTs, and interrupt controllers, are typically available free of charge from FPGA vendors for many of the simpler functions, or at a cost from Intellectual Property (IP) vendors, for more complex functions, such as Reed-Solomon encoders/decoders. This thesis will describe a program that is a "core generator" for Reed-Solomon encoders and decoders. The user enters the desired parameters of the RS (Reed-Solomon) code, such as code word length,

error correcting capability, initial root of the code generator polynomial, the size of the Galois field elements, and the field generator polynomial. The program then will produce the VHDL code for either a RS encoder or decoder, as well as the VHDL code for the required testbench.

The rest of this chapter will provide an introduction to error-correcting codes, followed by a short history of digital design methodologies. Finally, the contents and contributions of this thesis will be presented.

1.1 Error-Correcting Codes

Consider the block diagram of a digital communication system as shown in Figure 1 [11]. In this system, a data source sends its digital data (bits) to the data sink over the communication channel. The channel can come in various forms, such as a pair of twisted wires, a fiber optic bundle, or space itself in the case of radio or satellite communications. It may even be a storage medium, such as a floppy disc, or a computer memory. In this case, the "reception" may happen at a much later time than the transmission. This is in fact communication in time as opposed to the usual communication that happens in the spatial domain. In any case, the desired result is to transmit this data as reliably as possible from source to destination (sink).

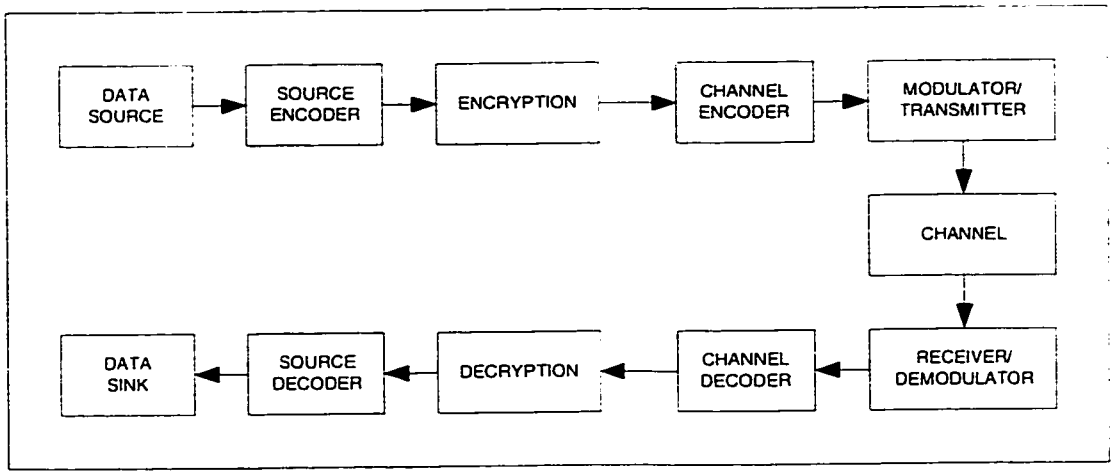

Figure 1. Block Diagram of a Digital Communication System.

The source encoder block, also called the data compression unit, is used to remove redundant information. This has the benefit of having to transfer fewer bits for the same source, thus either requiring less bandwidth, or taking less time to transmit for a given transmission rate. At this point, the data stream may be encrypted, if necessary. The channel encoder adds redundant information to the bit stream in order to be able to detect and correct errors in transmission, through the use of a carefully chosen "error correcting code". This digital bit stream is then sent to the modulator/transmitter which converts the digital data stream into a signal that can be transmitted over the "channel". It is the channel that is not in the control of the communications engineer. The channel is the medium over which the signal is sent from point A to point B. As such, it is the block that introduces noise into signal.

The actions of the blocks described above on the transmitter side must be reversed at the receiver side. First the receiver/demodulator takes the analog signal from the channel and converts it into symbols that the channel decoder can understand. It is in the channel decoder that the decoding algorithm for the error correcting code is applied. This results in a corrected digital data stream, in the case of there being a number of errors less than or equal to the error correcting capability of the code, or it may result in a corrupted digital data stream if the number of errors exceeded the error correcting capability of the

code. The resultant data stream is then decrypted, if encryption was applied, and the decompressed.

Broadly speaking, error-correcting codes may be classified into block codes, and convolutional codes. When block codes are used, the data stream is broken down into a predetermined and fixed size blocks. A block code then adds a fixed number of additional bits to the block, resulting in a larger block, which is then transmitted. On the other hand, a convolutional code adds one or more redundant streams, which are transmitted in a time sequential manner, one bit from each stream at a time. This is a continuous process. The next few sections will discuss these two classifications in more detail. It should be noted that this is not an exhaustive treatment of the subject.

1.1.1 Linear Block Codes

Block codes take "*k*" bits from the data stream, and add "*n-k*" parity bits, resulting in "*n*" bits of encoded data. If the code is linear, the calculation of the codeword can be represented compactly by a matrix multiplication. Define the original message as a vector of k bits :

$$m = \left(m_0, m_1, \ldots, m_{k-1}\right) \tag{1.1}$$

Multiplying the vector m by a generator matrix, G, forms the codeword.

$$c = (c_0, c_1, \ldots, c_{n-1}) = m \bullet G \tag{1.2}$$

This generator matrix depends one the code chosen, but is of the following form.

$$G = [P \vdots I_k] \tag{1.3}$$

I_k is the k dimensional identity matrix, while P is a k x (n-k) matrix describing the equations of the parity bits. The use of the identity matrix ensures that a direct copy of the original message appears in the codeword; this is called systematic encoding. The resulting 2^k codewords of length n form a subspace over the 2^n possible vectors. The modulo-2 sum of any 2 codewords also results in a codeword. A parity-check matrix, H, constructed in the following manner can be used to test if a received vector is indeed a codeword.

$$H = [I_{n-k} \vdots -P^T] \tag{1.4}$$

This consists of a (n-k) dimensional identity matrix, and the transpose of the submatrix P given in the generator matrix G. H is an (n-k) x n matrix. It can be shown [11] that any codeword c, multiplied by the transpose of H results in a (n-k) dimensional all zero vector.

$$c \bullet H^T = 0 \tag{1.5}$$

If the received codeword contains errors, then we can use the previous result to determine the error location. Since the received vector, **r**, is the sum of the original codeword **c**, and an error vector v. We calculate the syndrome **s**, of the received vector to determine error location.

$$s = r \bullet H^T = (c+e) \bullet H^T = c \bullet H^T + e \bullet H^T = 0 + e \bullet H^T = e \bullet H^T \tag{1.6}$$

This is the basis of syndrome decoding. Of particular interest in the choice of a block code are the code rate and the error correcting capability of the code. The code rate is the ratio of unencoded bits to encoded bits.

$$\text{code rate} = \frac{k}{n} \tag{1.7}$$

The error correcting capability of a code depends very much on the code chosen. Several examples of block codes will be shown in the following sections, including their decoding algorithms.

1.1.1.1 N-Repetition Codes

An "*n*" repetition code is the simplest of block codes. As the name suggests, it repeats every input bit a total of "*n*" times. Thus, this is a (n,1) block code. The algorithm that the channel decoder applies is called a majority vote. The number of 1's and 0's are counted, and the symbol with the highest count is chosen as the most likely candidate for the transmitted symbol. In order for no tie to occur, n is taken as an odd number. If we let $n = 2k + 1$, then this code can correct up to k errors in transmission. The code rate for such a code is a dismal $1/n$, thus precluding it from almost all practical use.

1.1.1.2 Single Error Detecting Code - Parity Codes

One of the first applications of error control codes was to detect if an error was present; so was born the single error detecting code, also called a parity code. A parity code takes "k" bits of information and appends 1 parity bit. This is a (k+1,k) code. Two types of parity code are used, even parity and odd parity. In an even parity code, the parity bit is chosen so that the total number of 1's is even; in an odd parity code, the parity bit is chosen so that the total number of 1's is odd. If one error occurs in the transmission, then the number of 1's will change. The decoding algorithm is to count the number of 1's. If it is even (for an even parity code), then no error was detected. However, if the number of 1's is odd, then an error has been detected. Note that this code can only detect an error, it cannot correct the error. Instead, if an error is detected, a retransmission can be asked for from the sender.

1.1.1.3 Single Error Correcting Code – Hamming Codes

The first practical error correcting codes were invented by Hamming in 1948, and were published in 1950 [12]. These were a class of linear block codes capable of single error correction. The parameters of these codes are as follows :

Codeword length :	$n = 2^m - 1$
Number of information bits :	$k = 2^m - m - 1$
Number of parity bits :	$n - k = m$
Error-correcting capability :	$t = 1$

The parity-check matrix of these codes is as follows :

$$H = [I_m \vdots Q] \tag{1.8}$$

I_m the m x m identity matrix, and Q is made up of k columns of m-tuples, each having at least two 1's. The (7,4) Hamming code (m=3) is a simple example of a single error correcting code. Let the parity check matrix be

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{1.9}$$

The corresponding generator matrix is

$$G = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.10}$$

Each separate single bit error corresponds to a unique syndrome value, as shown in the following table.

error pattern	Syndrome value
0000000	000
0000001	111
0000010	110
0000100	101
0001000	011
0010000	001
0100000	010
1000000	100

Calculating the syndrome, and adding the corresponding error pattern to the received vector accomplish error correction.

1.1.1.4 Cyclic Codes

A very important class of blocks codes are cyclic codes. These codes have a rich inherent algebraic structure that simplifies their encoding and decoding. More than 1 error can be corrected by the choice of a proper cyclic code. Codes are defined as being cyclic if the cyclic shift of every codeword is also a codeword, as shown below, where all three vectors are codewords.

$$\begin{aligned} codeword_1 &= (c_{n-1}, c_{n-2}, ..., c_2, c_1, c_0) \\ codeword_2 &= (c_{n-2}, ..., c_2, c_1, c_0, c_{n-1}) \\ codeword_3 &= (c_0, c_{n-1}, c_{n-2}, ..., c_2, c_1) \end{aligned} \tag{1.11}$$

If we consider the n dimensional codeword as the coefficients of a polynomial of degree n-1, c(x), with coefficients from GF(2), and the message vector similarly defined as polynomial of degree k-1, m(x), then their relationship can be expressed as follows :

$$c(x) = m(x) \bullet g(x) \tag{1.12}$$

where g(x), called the generator polynomial, is a polynomial of degree n-k. There is a restriction placed on g(x), namely, it must be a factor of $x^n - 1$. [11] Once a generator polynomial is chosen, the parity polynomial h(x) can be determined by the equation :

$$g(x) \bullet h(x) = x^n - 1 \tag{1.13}$$

The syndrome polynomial, s(x), is found by the following

$$s(x) = r(x) \bullet h(x) \text{ modulo } (x^n - 1) \tag{1.14}$$

If s(x) is equal to the 0 polynomial, the conclusion is that there is no error.

Two important types of cyclic codes are BCH and Reed-Solomon codes, which are discussed below.

1.1.1.4.1 BCH Codes

BCH codes have the following parameters :

Codeword length :	$n = 2^m - 1$ (bits)
Error-correcting capability :	t
Number of parity bits :	$n - k \leq m \bullet t$

Central to the theory of BCH codes is the mathematics of Galois fields. Consider a set F of elements, which has 2 operations defined on it called addition "+" and multiplication "*". The set F along with the two operations is called a field if [10]:

1. F is commutative under addition. The identity element for addition, called the *zero* element or the additive identity, is denoted by 0.
2. The set of non-zero elements in F is a commutative group under multiplication. The identity element for multiplication, called the *unit* element or the multiplicative identity, is denoted by 1.
3. Multiplication is distributive over addition; that is for any three elements a,b, and c in F,

$$a*(b+c) = a*b + a*c$$

The number of elements of a field is called the order of the field, and is designated by the letter q. The order of an element a in F is the smallest positive integer n such that $a^n=1$. An element from F is called a primitive element if its order is equal to $q-1$. Powers of primitive elements generate all the nonzero elements of F. The minimal polynomial for an element a is the polynomial of smallest degree over GF(2) such that a is a root.

Let α be a primitive element from $GF(2^m)$. Then the generator polynomial g(x) for a BCH code is the lowest-degree polynomial over GF(2), which has 2t roots which are consecutive powers of α. If we designate $\Phi_i(x)$ as the minimal polynomial of α^i, then g(x) is

$$g(x) = LCM\left\{\Phi_1(x), \Phi_2(x), ..., \Phi_{2t}(x)\right\} \tag{1.15}$$

where LCM is the least common multiple [10]. Clearly, this a "t" error correcting code. However, we cannot increase t indefinitely. The following table (taken from [10]) illustrates some BCH codes and their error correcting capabilities.

n	k	t	n	k	t
7	4	1	127	120	1
15	11	1		113	2
	7	2		106	3
	5	3		99	4
31	26	1		92	5
	21	2		85	6
	16	3		78	7
	11	5		71	9
	6	7		64	10
63	57	1		57	11
	51	2		50	13
	45	3		43	14
	39	4		36	15
	36	5		29	21
	30	6		22	23
	24	7		15	27
	18	10		8	31
	16	11			
	10	13			
	7	15			

Table 1. BCH Codes Generated by Primitive Elements of Order Less Than 27.

The syndrome polynomial can be calculated by polynomial multiplication. For large values of t, the error correcting capability of the code, the degree of the syndrome polynomial becomes proportionally large. As a consequence, the syndrome lookup table approach to error correction becomes impractical.

The error locator polynomial, $\sigma(x)$, is a polynomial whose roots are the locations of the errors. An iterative procedure, called the Massey-Berlekamp [3] algorithm, can be used to calculate the error locator polynomial from the syndrome polynomial. This algorithm requires 2t iterations to determine the coefficients of the error locator polynomial. Once this is done, each location in the received polynomial is tested, and if it is a root of the error locator polynomial, then the corresponding bit is reversed, thus correcting the error.

1.1.1.4.2 Reed-Solomon Codes

BCH codes are binary codes. Reed-Solomon codes are q-ary BCH codes. The parameters of these codes are :

Codeword length :	$n = 2^m - 1$ (symbols)
Symbol length :	m
Error-correcting capability :	t (symbols)
Number of parity symbols :	$n - k = 2 \cdot t$ (symbols)

Note that for every additional error that can be corrected, only 2t additional parity symbols are needed. This is an improvement over BCH codes. Let α be a primitive element from GF(2^m). Then the generator polynomial g(x) is the lowest-degree polynomial over GF(2) which has 2t roots which are consecutive powers of α.

$$g(x) = (x+\alpha)(x+\alpha^2)\cdots(x+\alpha^{2t}) \tag{1.16}$$

The syndrome polynomial can be computed as before, however, now we are dealing with operations over GF(2^m) instead of GF(2). Efficient shift register circuit configurations have been found to calculate the coefficients of the syndrome polynomial, as discussed in section 7.2.

Once the syndrome polynomial has been found, the error locator polynomial, $\Lambda(x)$, and the error evaluator polynomial, $\Omega(x)$, (also called the error magnitude polynomial) must be computed. The error evaluator polynomial is needed, because knowing the error location is not sufficient to correct an error. Each symbol is m bits wide, thus one of the 2^m-1 possible error patterns must be added to the symbols at the error locations.

Several algorithms for the error locator polynomial have been devised, such as the Peterson-Gorenstein-Zieler algorithm [1], Euclid's algorithm [12], and the non-binary Massey-Berlekamp algorithm [12]. The error evaluator polynomial can be computed as follows :

$$\Lambda(x)S(x) \equiv \Omega(x) \bmod(x^{2t}) \tag{1.17}$$

Improvements have been made to the Massey-Berlekamp algorithm where both the error locator polynomial and the error evaluator polynomial are computed at the same time. This algorithm requires performing Galois field inversion, which is the most difficult of the basic operations of addition, multiplication and inversion. Recently, an inversionless Massey-Berlekamp algorithm was discovered [6], however, it only computes the error locator polynomial. An extension to this algorithm which calculates both polynomials at the same time without the use of inversion was discovered by this author [13], and is described in detail in section 2.6.3.

Once these two polynomials have been found, the roots of the error locator polynomial are found. An efficient method of performing this function is called the Chien search, and is described in section 2.3. The error value that must be added to the symbol at the error location is given by Forney's algorithm as discussed in section 2.6.3.

1.1.2 Convolutional Codes

Convolutional codes work on a continuous stream of input bits, as opposed to a block of bits. Convolutional codes are described by three parameters (n, k, m), where there are n output, k input, and memory size m. Refer to the following figure which shows the structure of a (4,3,4) convolutional encoder.

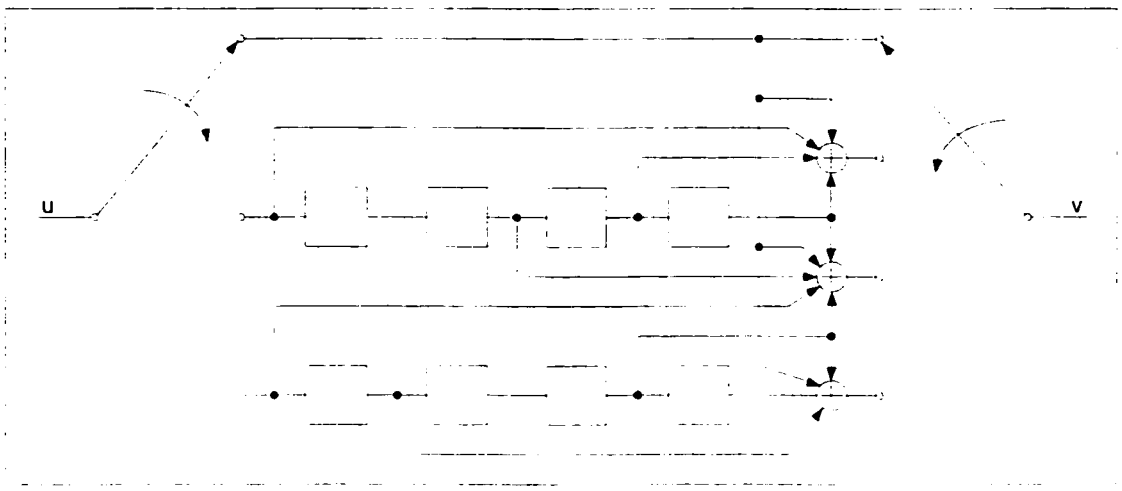

Figure 2. Block Diagram of a (4,3,4) Convolutional Encoder.

A convolutional encoder generates n encoded bits for every k information bits. From the figure we see that the top chain does not contain any memory elements. This is the systematic part of the code. The remaining outputs are functions of the input and the memory elements in the (m-1) memory chains. Thus an additional piece of information that is required is which of the inputs and memory elements are added together (modulo) to form each output. Note that each input bit at time t can possibly affect the outputs up to m time units later.

Decoding of convolutional codes is done by performing the Viterbi algorithm. This algorithm is a maximum likelihood decoding algorithm, and is equivalent to a dynamic programming solution to the problem [10]. The details of this algorithm are beyond the

scope of this thesis, but the reader is referred to texts in error control coding such as [10] and [11] for further information.

1.2 Digital Design Implementation Technologies

The implementation of digital designs has changed greatly over the last 25 years. In this section, we shall explore the development of the methods of implementing a design. This ranges from the use of integrated circuits in individual packages, to large ASICs capable of handling an entire design, and finally to the recent innovations in programmable logic.

1.2.1 Integrated Circuits (SSI, MSI, LSI, VLSI)

At the beginning of the electronics age, after transistors were first crafted in silicon, designers had to create designs out of basic building blocks such as AND gates and flip-flops. These functions were found inside integrated circuits, which are grouped into 4 categories, namely

Technology type	Equivalent gate count, C
Small Scale Integrated (SSI) circuits	$C \leq 12$
Medium Scale Integrated (MSI) circuits	$12 < C < 99$
Large Scale Integrated (LSI) circuits	$100 \leq C \leq 999$
Very Large Scale Integrated (VLSI) circuits	$C \geq 1000$

Table 2. Equivalent Gate Count for SSI / MSI / LSI / VLSI.

The basic functions such as AND gates, OR gates, inverters, and single or dual flip-flops are found in Small Scale Integrated (SSI) circuits. The line between MSI and LSI is more of a blur, but MSI was typically made up of shift registers, multipliers, adders, quad, hex and octal flip-flops, counters, multiplexers and demultiplexers and decoders. LSI, with its higher gate count and functionality includes FIFOs, memory (SRAM and

ROM), and bit-slice processors. VLSI is the realm of microprocessors and their support chips.

The interconnection of these elements was determined by hand and written onto schematics. Reduction of Boolean equations to a minimal set was done by hand. Each pin of device had to be accounted for. Once a schematic was produced, a printed circuit board (PCB) was made. This was the physical implementation of the design interconnection. If a design change was required, the process started by modifying the schematic, and then making a new PCB. Needless to say, this was a costly and time consuming cycle. An example of a small portion of schematic is shown in the following figure.

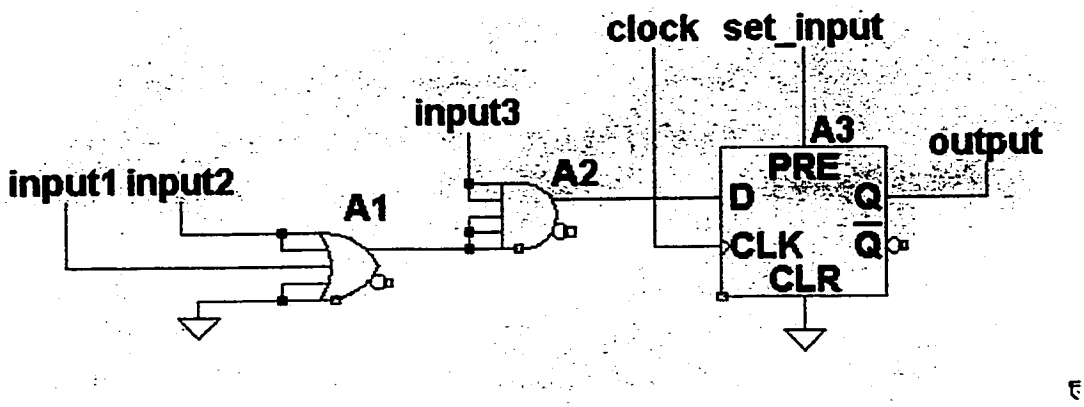

Figure 3. Schematic Diagram Example.

1.2.2 Application Specific Integrated Circuits (ASICs)

The evolution of digital design implementation has culminated in the Application Specific Integrated circuits (ASICs). In these devices, a designer may implement an entire design consisting of many thousands, or even millions of gates. ASICs may be partitioned into 3 groups.

In order of increasing performance they are :

- standard cell
- full custom.
- gate arrays

With full custom ASICs, the designer creates the geometries which make up transistors. Very high densities and speeds can be achieved, but a disadvantage of full custom is that they are very costly and require a very long design time. Non-recurring costs can easily run into the hundreds of thousands or a million dollars. With the standard cell, a library of standard cells is used, and an automatic place and route tool performs the layout. Design time is usually faster than full custom for this reason. In addition, the manufacturing process is the same as for full custom, so the cost is still high. With gate arrays, a library of standard cells is mapped onto an array of transistors already present on a wafer, which can be produced in quantity ahead of time. A routing tool is used to create the masks for routing. Design time is fast, and manufacturing costs a less than for full custom or standard cell. Performance can approach that of standard cell and full custom.

When ASICs were first introduced, the design was entered by schematics, and tools would make the necessary mapping from logic to transistors. For large designs, the schematics could be hundreds of pages long. While the clutter of a large design could be mitigated by performing a top down design approach, eventually, the burden of detail could overwhelm the designer. To accommodate the increasing complexity of designs, an alternative approach was invented.

Thus was born VHDL or VHSIC Hardware Description Language, where VHSIC means Very High Speed Integrated Circuit. Using this method, a descriptive language is used to specify the functionality of a digital circuit at a higher level. Synthesis and place and route tools are used to perform Boolean minimizations, state machine encoding and other low level tasks, and to map the design to the technology to be used. Actions to be performed are written in processes, with many levels of if-then-else possible. This makes

a complicated behavior possible in just a few lines of code. Here is an example of a counter in VHDL. References [24] and [25] provide excellent background to the VHDL language.

```
Signal counter : std_logic_vector (7 downto 0);

Process (clk)
Begin
  If (clk'event and clk='1') then
    If (reset_n='0') or (count_reset='1') then
      Counter <= (others => '0');
    Elsif (count_enable='1') then
      Counter <= Counter + 1;
    End if;
  End if;
End process;
```

Figure 4. VHDL Example For a Counter.

1.2.3 Programmable Logic

Programmable logic was invented to give the designer added flexibility, and the ability to integrate many small functions into one device. In addition, a company could stock pile one or two types of programmable logic, and entirely stop using much of the SSI and MSI functions, since they could be replicated within the programmable logic.

The first programmable devices were called PALs (programmable array logic) and PLDs (programmable logic devices). These could implement about 100 gates. They are arranged as an array of logic in sum of products form. The designer would make the design using a low level hardware language, such as ABEL or CUPL, which specifies the desired logic in sum of products form. The output of the logic could also be registered or not as needed.

As the complexity of designs increased, the need for larger programmable devices became apparent. This resulted in the development of Complex Programmable Logic Devices (CPLDs). These are essentially several PALs on one chip, with an interconnect matrix between them.

The next development was the introduction of Field Programmable Gate Arrays (FPGAs). In this technology, there are a large number of programmable gates, and a programmable interconnect between the gates. Some FPGAs have included other features such a built-in hardware multipliers, embedded memory, and even microprocessor. FPGAs have now reached the level of performance coming close to that of ASICs. For example, the largest Xilinx FPGA at present is the XC2V8000 which has the equivalent of 8 million system gates, 3024 Kbits of RAM (SelectRAM) and 168 (18 bit x 18 bit) hardware multipliers [14]. Internal clock speeds for register to register transfers can exceed 200 MHz.

1.3 Contributions and Contents of the Thesis

The structure of the thesis report is as follows. Chapter 2 will provide the necessary background into the theory of Reed-Solomon codes. This chapter will feature an improvement to the efficiency of the key equation solver by the use of an "extended inversionless Massey-Berlekamp algorithm". Since Galois field inverters are very complex, several times more so than a Galois field multiplier, a key equation solver that eliminates the need for such an inverter can yield savings in decoder latency. Such an algorithm is the inversionless Massey-Berlekamp algorithm by Reed, Shih, and Truong [6]. However, the drawback to this algorithm is that it only computes the error-locator polynomial. The error-evaluator polynomial is computed then by multiplying the error-locator polynomial by the syndrome polynomial. This results in extra latency of the order of the error-locator polynomial, and as a result an extra number of gates and registers are needed, as will be shown. The extended inversionless Massey-Berlekamp algorithm overcomes this extra latency by computing both the error-locator polynomial

and the error-evaluator polynomial at the same time. The description and proof of this improved algorithm are included in this thesis.

Chapter 3 discusses the structure of the program that generates the VHDL code for an encoder or a decoder. Chapter 4 discusses the basic operations of addition, multiplication and inversion in Galois fields, and how they are mapped into VHDL code. Chapter 5 examines in detail the structure and VHDL implementation for an arbitrary RS encoder. This is followed by a description of the synthesis results for the encoder in Chapter 6.

The design of a RS decoder is broken down into decoders for 3 or more errors, and decoders for 1 or 2 errors. The reason for such a breakdown is due to a simplification in the decoding process for 1 or 2 error correcting codes. Chapter 7 will describe the structure and VHDL implementation for a RS decoder for 1 or 2 errors. It will be shown that a simplification to the decoding process is possible, namely the removal of the key equation solver, and a resulting small increase in complexity in the Chien search and error correction block.

Chapter 8 describes the structure and VHDL implementation for a RS decoder for 3 or more errors; the major blocks are the syndrome calculation, key equation solver, and the Chien search and error correction block. Chapter 9 will describe the synthesis results, in terms of area and speed, for all of the RS decoders described in Chapters 7 and 8 generated by the "RS core generator". Chapters 11 will present the conclusions. The target technology to be examined is Xilinx FPGAs, a very common class of devices used to implement DSP and communications related algorithms.

1.4 Previous Work

This section will examine previous that has been done in the area of VHDL code generation and high-level synthesis for FEC circuits. Two of the sources are Master's

theses [26, 27]. Reference 26, "High-level Synthesis and Its Application in the Design of Reed-Solomon Decoders", describes the use of high-level synthesis in the design of Reed-Solomon decoders. A Computer-Aided-Design (CAD) tool is implemented showing the advantages of high-level design techniques as applied to the versatile time-domain RS(n,k) decoder [32]. Techniques for optimizing performance are presented. An implementation of these methods to the versatile RS decoder is discussed and the area and speed metrics are shown. The basic Galois field operations are discussed in detail, since these functions form the basis of RS decoding.

Reference 27, "The Design of a VHDL Based Synthesis Tool For BCH Codecs", describes a VHDL code generator for Bose, Chaudhuri and Hocquenghem (BCH) codes. These codes are binary and are less complex in nature. Since they are binary, only the error location needs to be found. The decoding procedure is very close to that required to decode Reed-Solomon codes. First, the syndromes must be calculated from the incoming codeword. The Massey-Berlekamp algorithm can be used to find the error locator polynomial. Finally, the Chien search is used to find the roots of the error locator polynomial, and hence, the error locations. The corresponding bit is then inverted to correct the error.

In a sense, this document is closest to the topic of this thesis. Fast and efficient algorithms are developed for the basic operations, including a dual-polynomial basis multiplier, a parallel polynomial basis multiplier and a circuit for raising field elements to the third power. As in this thesis, the user enters the relevant parameters for the BCH code desired, and the core generator, written in C, produces VHDL code that is synthesizable. It presents different implementations for 1 error, 2 error, and multiple error correcting BCH codes, in order to optimize performance. Area and speed metrics are shown for BCH decoders of varying error-correcting capability.

Reference 28, "Very High-Level Synthesis of Control and Datapath Structures for Reconfigurable Logic Devices", describes an approach to the synthesis of inner loops that manipulate array variables using affine index access functions written in C directly to

VHDL. The approach uses analysis techniques that allow a variety of design implementations for space or execution time reduction. The approach is automatic, and it implements the data path and the control path separately. This separation allows the generation of several control strategies. The design of the VHDL core generator for Reed-Solomon encoders and decoders also applies this methodology of separating the data path from the control path. This permits the generation of VHDL code that is clearer and easier to debug.

The paper also describes two approaches to control design, namely, a state table driven approach, where the control signals are defined by simple setting or clearing depending on the state, and secondly, a counter based approach, where control signals are set or cleared based on the value of a counter which counts up from 0 when a "start" signal is observed. These two approaches are used in the VHDL core generator for Reed-Solomon encoders and decoders. The state table approach is used in the design of the syndrome calculation for RS decoders. The counter based approach is used in the other modules of the RS decoder, and is used in the design of the RS encoder.

Reference 29, "Code Generation Tools For Hardware Implementation of FEC Circuits", is a brief paper which describes a VHDL code generator for Reed-Solomon decoders using Euclid's algorithm. The VHDL code is synthesizable, and the area and speed metrics for several decoder designs is presented, targeted to Xilinx Field Programmable Gate Arrays. The core generator also generates the vectors for the test bench. The code generator is capable of generating VHDL code for most binary FEC decoders, such as Hamming, Cyclical Redundancy Check (CRC), and BCH codes. In the design of the RS decoder, a bank of Galois field multipliers is used, as opposed to a dedicated multiplier where it is needed. This results in less area, but requires tristate buses and reliable scheduling of the steps to avoid collisions.

Reference 30, "A Soft IP Compiler For Reed-Solomon Decoder", a program that generates synthesizable VHDL code for Reed-Solomon decoders is presented. Erasure decoding is supported. The method of solving the key equation is the inversionless

Massey-Berlekamp algorithm, or a modified version called the reformulated inversionless Massey-Berlekamp algorithm. The choice of the algorithm is user controlled. The area and speed metrics are provided for several designs. The average throughput achieved is 500 Mbits/second in an ASIC. This is comparable to the results achieved by the core generator of this thesis when applied to Xilinx FPGAs.

Reference 31, "Design of a Reed Solomon Decoder using Partial Dynamic Reconfiguration of XILINX Virtex FPGAs – A Case Study", describes a methodology and design flow for VIRTEX FPGAs which enable designers to efficiently use the VIRTEX pRTR (partial run-time reconfigurability) features. The methodology and results are demonstrated on the design of a Reed-Solomon Coder/Decoder.

2 Reed-Solomon Encoding and Decoding Algorithms

2.1 RS Encoding

A Reed-Solomon code is a q^m-ary BCH code of length q^m-1. Code symbols are elements of the corresponding Galois field GF(q^m). In this thesis, we will be dealing with extensions of binary codes, i.e., q=2. This field is generated by an primitive irreducible polynomial of degree m. Let α be a primitive element from GF(2^m). The generator polynomial for the code capable of correcting "t" errors is of degree 2t, and has as its roots, 2t consecutive powers of α, that is :

$$g(x) = \prod_{i=0}^{2t-1} (x + \alpha^{m_0 + i}) \tag{2.1}$$

where m_0 is the log of the initial root. The message of k*m bits is divided into k symbols, each symbol being an m-bit element from GF(2^m). Each symbol is regarded as a coefficient of a k-1 degree polynomial, m(x). A total of 2t parity symbols are appended to the message, thus making a systematic encoding format. Such an RS code is denoted by 2 parameters, n and k, and is written as RS(n,k). The distance of RS codes is :

$$\text{distance} = n - k + 1 = 2t+1$$

Here there are k message symbols, and n-k = 2t parity symbols, for a total of n symbols. RS codes may be shortened to RS(n',k'), where n'=n-l, and k'=k-l. In this case, the distance property d = 2t+1 still holds. The encoded message, c(x), is formed as follows [10] :

1. multiply the message m(x) by x^{2t}
2. form the parity symbols, b(x), by dividing the above result by the generator polynomial.
3. $c(x) = m(x) \cdot x^{2t} + b(x)$

Thus, in order to specify an RS code completely, the following items must be described.

1. The degree of the field generator polynomial, m, and its coefficients.
2. The error-correcting capability, t, of the code.
3. The number of message symbols, k.
4. The log of the initial root of the code generator polynomial, m_0.

Note that there is a restriction placed on these parameters. First, we can calculate n, the total number symbols as n=k+2t. This number must be less than or equal to the maximum number of allowable symbols, namely (2^m-1). The higher order symbols are usually transmitted first.

2.2 Syndrome Calculation

The syndrome calculation is the first step in the decoding process. For a t-error correcting RS code, there are 2t syndromes that must be calculated. They are calculated as follows :

$$S(j) = r(\alpha^{m_0+j}), \text{ where } j \in \{0..2t-1\} \tag{2.2}$$

In the equation above, r(x) is the received codeword polynomial, and m_0 is the log of the initial root of the code generator polynomial. Since the received codeword polynomial is the sum of the message polynomial and the error polynomial, we can reduce this to

$$S(j) = c(\alpha^{m_0+j}) + e(\alpha^{m_0+j}) = e(\alpha^{m_0+j}) = \sum_{i=0}^{t} Y_i \bullet X_i^{j+m_0}, \text{ where } j \in \{0..2t-1\} \tag{2.3}$$

where the Y_i are the error values at the error locations X_i. This information will be used later in the decoding algorithms for 1 and 2-error correcting codes.

2.3 Chien Search

Another crucial step in the decoding process is to find the location of the errors in the received codeword. An iterative process, called the Chien search is the most efficient means of doing this. Consider the error locator polynomial, $\sigma(x)$, which is a t degree polynomial whose roots are the error locations, X_i.

$$\sigma(x) = \prod_{i=1}^{t} (x + X_i) = x^t + \sigma_1 \bullet x^{t-1} + \cdots + \sigma_{t-1} \bullet x + \sigma_t \qquad (2.4)$$

Now, it can be shown [5], that the coefficients of the error-locator polynomial are related to the syndrome values, S(j), through a set of relations called Newton's identities :

$$S(t+j) + \sigma_1 \cdot S(t+j-1) + \ldots + S(j) \cdot \sigma_t = 0 \qquad \forall j \qquad (2.5)$$

This fact will be called upon later in the next 2 sections when the equations for the error values for 1-error and 2-error correcting codes are derived. Any root of $\sigma(x)$ will also satisfy the equation

$$\sigma_1 \bullet x^{-1} + \sigma_2 \bullet x^{-2} + \cdots + \sigma_t \bullet x^{-t} = 1 \qquad (2.6)$$

The circuit shown in Figure 5 will implement such a search, one at a time (per clock cycle). Initially, the values of the registers are loaded with the coefficients of the error locator polynomial. A sum of the registers is formed, resulting in the Chien sum, and if it is 0, then an error has been located.

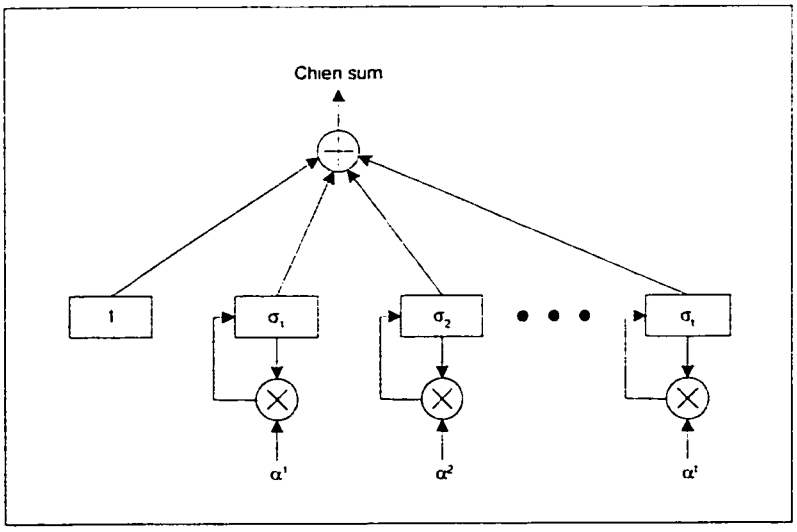

Figure 5. Chien Search Block Diagram

Since the received message polynomial, r(x), is received one symbol at a time, starting with the symbol for the first position, a pipelined approach to the decoder is possible. This is the approach taken in this thesis.

2.4 Decoding for 1 Error

When error correcting capability of a Reed-Solomon code is limited to one error, that is, n – k = 2, the process of computing the error-locator polynomial and the error-evaluator polynomial is not necessary. Instead, one can proceed with the Peterson-Gorenstein-Zierler decoding algorithm [1]. The derivation for a general 1-error correcting RS code will now be shown.

The generator polynomial for a 1-error correcting code is given by $g(x) = \prod_{i=0}^{1} (x + \alpha^{m_0 + i})$. Consider now the syndrome equations for such an RS code. There are 2t=2 syndromes, given by :

$$S(0) = r(\alpha^{m_0}) \text{ and } S(1) = r(\alpha^{1+m_0}) \tag{2.7}$$

where m_0 is the log of the initial root of the RS field generator polynomial. From this, we proceed as follows. Let the error-locator polynomial be given by $\sigma(x) = \prod_{i=1}^{1} (x + X_i) = x + \sigma_1$, where X_1 is the location of the single error. The syndrome equations can be related to the error-locator polynomial via some algebraic manipulations, to arrive at the so called Newton's identities [5]. In general, Newton's identities are in the form of :

$$S(t+j) + \sigma_1 \cdot S(t+j-1) + \ldots + S(j) \cdot \sigma_t = 0, \forall j \tag{2.8}$$

For the case of a 1-error correcting code, t=1, we get : $S(1)+\sigma_1 \cdot S(0)=0$. Thus $\sigma(x)=x+\sigma_1$ is the error locator polynomial, where $\sigma_1=-\frac{S(1)}{S(0)}$. So far we have calculated the position of the error, using just the syndrome values. Now, we must also calculate the error value from the syndromes, as follows :

$$S(0)=r(\alpha^{m_1})=Y_1 \cdot X_1^{m_1} \Rightarrow Y_1=S(0) \cdot X_1^{-m_1} \tag{2.9}$$

where Y_1 is the error value, and X_1 is the error location.

The entire algorithm for a single error correcting code is then stated as follows :

1. Calculate the syndromes $S(0)=r(\alpha^{m_1})$ and $S(1)=r(\alpha^{1+m_1})$.
2. Let D_1=S(0). If D_1=0, then there is no error, and STOP. If ($D_1 \neq 0$) then there is an error.
3. Perform a Chien search on the entire received message. NOTE: the Chien search requires both syndrome values. If the element at position (X_1) is found to be the location of an error, that is, when the Chien sum is 0, then
   a. calculate the error value : $Y_1=S(0) \cdot X_1^{-m_1}$
   b. correct the error : corrected_element = received_element + Y_1
4. STOP

Note that for Galois fields of the form GF(2^m), that is, those that are extensions of the binary field, -a = a, and so we may modify the equation for the error location to $X_1=\frac{S(1)}{S(0)}$. The equation for the error value remains the same.

It should be noted that equations have been derived to calculate the coefficients of the error locator polynomial and the error value directly from the syndrome values and the

error locations, as can be found in [5]. These equations, however, are only valid for a single value of m_0. Moreover, the size of these equations increases exponentially with the error correcting capability, and quickly become too cumbersome to use, since all of the error locations must be known before applying the error correction formulae. In addition, the Chien search is suited for use in a stream of data corresponding to a codeword, as has been assumed in this thesis. As an example, consider the use of the Chien search for the 1-error case. Referring to Figure 5, the 1-error correction case reduces to the following.

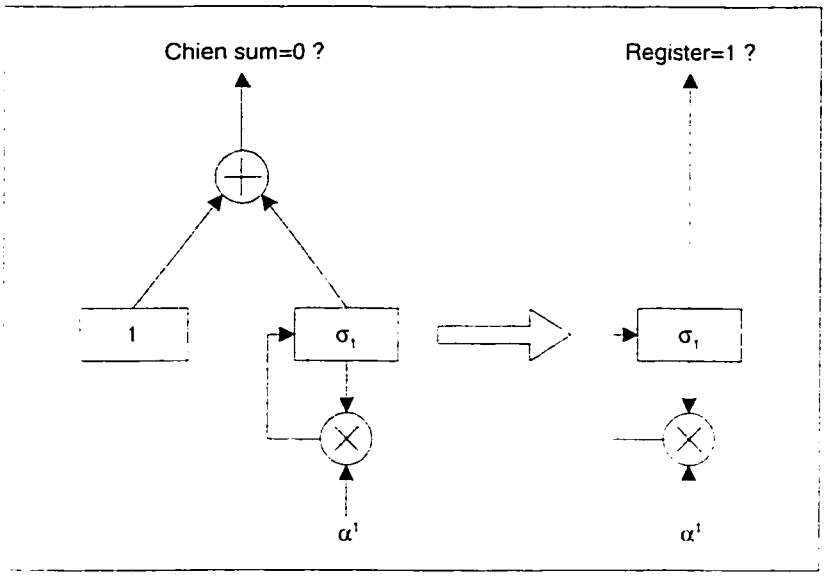

Figure 6. Reduced Form of Chien Search For Single Error Correcting Code

In this figure, 2 registers are used. One is loaded with the value of "1", and the other register is loaded with the value calculated for the error location, namely, $X_1 = \frac{S(1)}{S(0)}$. Every clock cycle, this register is multiplied by α, and added to the register containing "1", resulting in the Chien sum. If the Chien sum is "0", then the location is the error location. A further simplification can be done by simply removing the register containing "1", and testing the remaining register for the value of "1". These are equivalent systems, and the second method is that used in the design of the single error correcting decoders in this thesis.

Also note, that this method is actually simpler than using a counter to determine the position of the error. This is due to the fact that error location calculated is the Galois field representation of the position. Therefore, one would have to perform a logarithm operation to determine the correct error position in terms of integer values, and then to use a counter to count to this value. Calculating the logarithm of finite field element is more complex than multiplying by α.

A block diagram of a pipelined single error correcting RS decoder is shown in Figure 7 below. An input strobe, RS_data_in_start is used to signify the beginning of a new input codeword. The codeword enters the decoder in bitwise parallel format, one symbol at a time. It first enters into the Calculate_Syndrome block, where the 2 syndromes are calculated. In addition, a control signal Error_Present is also shown, and is set to '0' (inactive) if both syndromes are equal to 0, otherwise it is set to '1' (active). Finally, a strobe indicating the completion of the syndrome calculation is sent to the Chien_Search block. A delay block for the incoming RS symbols is necessary to compensate for the delays in the Calculate_Syndrome block, and any startup delays in the Chien_Search block. The Chien_Search block performs the function of finding the error locations, and correcting the data when the Chien sum is 0. The corrected data and an output data strobe are the outputs of the RS decoder.

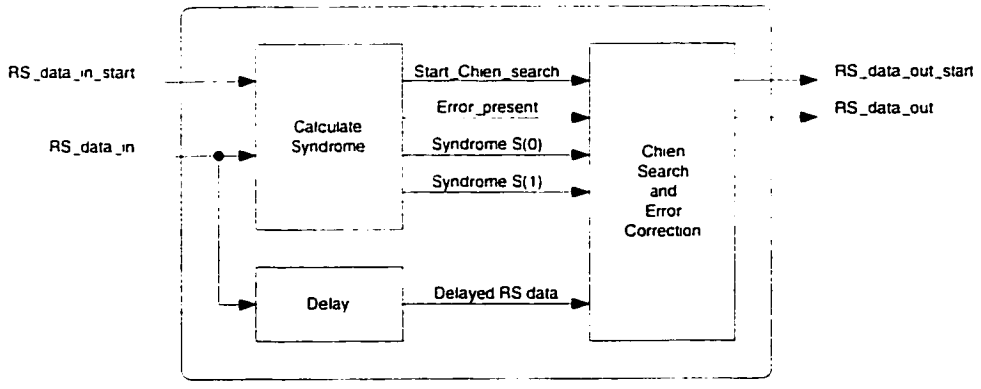

Figure 7. Single Error Correcting RS Decoder Block Diagram

2.5 Decoding for 2 Errors

When the error correcting capability of a Reed-Solomon code is limited to two errors, the process of computing the error-locator polynomial and the error-evaluator polynomial is also not necessary. As in the previous section, one can proceed with the Peterson-Gorenstein-Zierler decoding algorithm [1]. The derivation for a general 2-error correcting RS code will now be shown.

The generator polynomial for a 2-error correcting code is given by $g(x)=\prod_{i=0}^{3}(x+\alpha^{m_0+i})$. Consider now the syndrome equations for such an RS code. There are 2t=4 syndromes, given by :

$$\begin{aligned} S(0) &= r(\alpha^{m_0}) \\ S(1) &= r(\alpha^{1+m_0}) \\ S(2) &= r(\alpha^{2+m_0}) \\ S(3) &= r(\alpha^{3+m_0}) \end{aligned} \tag{2.10}$$

where m_0 is the log of the initial root of the RS field generator polynomial. From this, we proceed as follows. Let the error-locator polynomial be given by $\sigma(x)=\prod_{i=1}^{2}(x+X_i)=x^2+\sigma_1 \bullet x+\sigma_2$, where X_i is the location of the i^{th} error. Once again, we use Newton's identities :

$$S(t+j)+\sigma_1 \cdot S(t+j-1)+\ldots+S(j)\cdot\sigma_t=0, \forall j .$$

In matrix form, the 2 equations in 2 variables are :

$$\begin{bmatrix} S(0) & S(1) \\ S(1) & S(2) \end{bmatrix} \cdot \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S(2) \\ -S(3) \end{bmatrix} \tag{2.11}$$

Let $D_2 = S(0) \cdot S(2) + S(1)^2$. This is the determinant of the 2x2 matrix given above. If D_2 is not 0, then 2 errors are present, and we proceed by solving for the coefficients of the error locator polynomial, $\sigma(x) = x^2 + \sigma_1 \cdot x + \sigma_2$. This is accomplished by solving this system of 2 equations in 2 variables. Using Cramer's rule, the result is :

$$\sigma_1 = \frac{\begin{vmatrix} S(0) & S(2) \\ S(1) & S(3) \end{vmatrix}}{D_2} = \frac{S(0) \cdot S(3) + S(1) \cdot S(2)}{D_2} = \frac{S(0) \cdot S(3) + S(1) \cdot S(2)}{S(0) \cdot S(2) + S(1)^2}$$
$$\sigma_2 = \frac{\begin{vmatrix} S(2) & S(1) \\ S(3) & S(2) \end{vmatrix}}{D_2} = \frac{S(2)^2 + S(1) \cdot S(3)}{D_2} = \frac{S(2)^2 + S(1) \cdot S(3)}{S(0) \cdot S(2) + S(1)^2} \tag{2.12}$$

So far we have calculated the position of the error, using just the syndrome values. Now, we must also calculate the error value from the syndromes, as follows. The syndrome equations for the first 2 syndromes are :

$$S(0) = r(\alpha^{m_0}) = Y_1 \cdot X_1^{m_0} + Y_2 \cdot X_2^{m_0}$$
$$S(1) = r(\alpha^{m_0+1}) = Y_1 \cdot X_1^{m_0+1} + Y_2 \cdot X_2^{m_0+1} \tag{2.13}$$

where Y_1 is the error value, and X_1 is the error location of the first error, and Y_2 is the error value, and X_2 is the error location of the second error. Putting this into matrix form, we get

$$\begin{bmatrix} X_1^{m_0} & X_2^{m_0} \\ X_1^{m_0+1} & X_2^{m_0+1} \end{bmatrix} \cdot \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} S(0) \\ S(1) \end{bmatrix} \tag{2.14}$$

We can now solve for the Y_i's in term of the syndrome values S(0) and S(1), and the error locations X_1 and X_2. First, we define the determinant.

Let $\Delta = X_1^{m_0} \cdot X_2^{m_0+1} + X_1^{m_0+1} \cdot X_2^{m_0} = X_1^{m_0} \cdot X_2^{m_0} \cdot (X_1 + X_2)$. Then the Y_i's are found to be :

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} \dfrac{S(0) \cdot X_2^{m_0+1} + S(1) \cdot X_2^{m_0}}{\Delta} \\ \dfrac{S(1) \cdot X_1^{m_0} + S(0) \cdot X_1^{m_0+1}}{\Delta} \end{bmatrix} = \begin{bmatrix} \dfrac{S(0) \cdot X_2^{m_0+1} + S(1) \cdot X_2^{m_0}}{X_1^{m_0} \cdot X_2^{m_0} \cdot (X_1 + X_2)} \\ \dfrac{S(1) \cdot X_1^{m_0} + S(0) \cdot X_1^{m_0+1}}{X_1^{m_0} \cdot X_2^{m_0} \cdot (X_1 + X_2)} \end{bmatrix}$$

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} \dfrac{S(0) \cdot X_2 + S(1)}{X_1^{m_0} \cdot (X_1 + X_2)} \\ \dfrac{S(1) + S(0) \cdot X_1^1}{X_2^{m_0} \cdot (X_1 + X_2)} \end{bmatrix} \qquad (2.15)$$

The values of error locations are found during the Chien search. The Chien search goes through every location and tests it if it is an error location [5]. This gives us the value of X_1, at any one location. Since there are 2 errors, the error locator polynomial is a quadratic, and we may solve for the remaining X_2.

$$\sigma(x) = x^2 + \sigma_1 \cdot x + \sigma_2 = (x + X_1) \bullet (x + X_2) = x^2 + (X_1 + X_2) \cdot x + X_1 \cdot X_2$$
$$\Rightarrow \sigma_1 = X_1 + X_2, \text{or solving for } X_2, \quad X_2 = \sigma_1 + X_1 \qquad (2.16)$$

Given this, the equation for the error at location X_1 during the Chien search is :

$$Y_1 = \frac{S(0) \cdot (X_1 + \sigma_1) + S(1)}{X_1^{m_0} \cdot \sigma_1} \qquad (2.17)$$

where X_1 is the present location, S(0) and S(1) are the first 2 syndrome values, and σ_1 is the coefficient of the first power of x in the error locator polynomial, which has been previously calculated as $\sigma_1 = \dfrac{S(0) \cdot S(3) + S(1) \cdot S(2)}{S(0) \cdot S(2) + S(1)^2}$.

This is the result for 2 errors. If the determinant D_2 was indeed 0, then we proceed as in the case for a single error, as discussed in section 2.4.

The entire algorithm for a double error correcting code is then stated as follows :

1. Calculate the 4 syndromes :

$$S(0) = r(\alpha^{m_0})$$
$$S(1) = r(\alpha^{1+m_0})$$
$$S(2) = r(\alpha^{2+m_0})$$
$$S(3) = r(\alpha^{3+m_0})$$

2. Based on the syndromes, calculate the determinant $D_2 = S(0)\cdot S(2)+S(1)^2$. If this value is 0, goto step 6.
3. Calculate the coefficients of the error locator polynomial from the syndromes :

$$\sigma_1 = \frac{S(0)\cdot S(3)+S(1)\cdot S(2)}{S(0)\bullet S(2)+S(1)^2}, \text{ and } \sigma_2 = \frac{S(2)^2+S(1)\cdot S(3)}{S(0)\bullet S(2)+S(1)^2}$$

4. Perform a Chien search on the entire received message. If the element at position (X_1) is found to be the location of an error, that is, when the Chien sum is 0, then
   a. calculate the error value : $Y_1 = \frac{S(0)\cdot(X_1+\sigma_1)+S(1)}{X_1^{m_0}\cdot\sigma_1}$
   b. correct the error : corrected_element = received_element + Y_1
5. STOP
6. Let D_1=S(0), the value of the first syndrome. If D_1=0, then there is no error, and we must STOP. If ($D_1 \neq 0$) then there is an error, and we continue.
7. Perform a Chien search on the entire received message. If the element at position (X_1) is found to be the location of an error, that is, when the Chien sum is 0, then
   a. calculate the error value : $Y_1 = S(0)\cdot X_1^{-m_0}$
   b. correct the error : corrected_element = received_element + Y_1
8. STOP

Note that it was assumed that we are dealing with Galois fields of the form $GF(2^m)$, that is, those that are extensions of the binary field. These are the fields of any practical use. A block diagram of a pipelined double error correcting RS decoder is shown in Figure 8 below. As before, an input strobe, RS_data_in_start is used to signify the beginning of a

new input codeword. The codeword enters the decoder in bitwise parallel format, one symbol at a time. It first enters into the Calculate_Syndrome block, where the 4 syndromes are calculated. The control signal Error_Present is also generated. Finally, a strobe indicating the completion of the syndrome calculation is sent to the Chien_Search block. A delay block for the incoming RS symbols is necessary to compensate for the delays in the Calculate_Syndrome block, and any startup delays in the Chien_Search block. The Chien_Search block performs the function of finding the error locations, and correcting the data when the Chien sum is 0. Note that the Chien_Search block in this case is more complex than the for the single error correcting case, as has been described in the paragraphs above. The corrected data and an output data strobe are the outputs of the RS decoder.

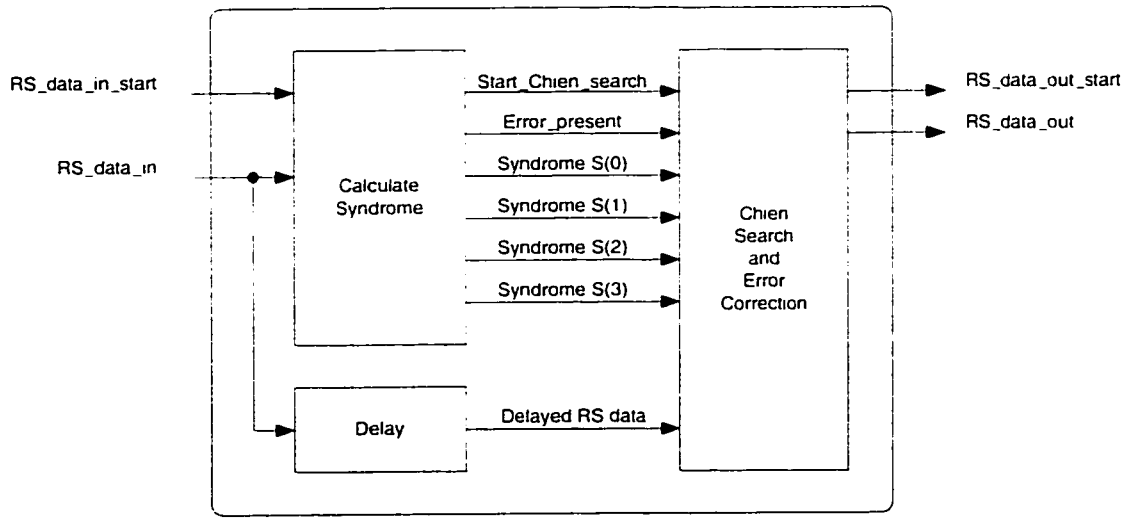

Figure 8. Double Error Correcting RS Decoder Block Diagram

2.6 Decoding For 3 or More Errors

This section is devoted to the case of RS decoding for 3 or more errors. In this case, the method outlined in Sections 2.4 and 2.5 are not appropriate for a pipelined design. For 3 or more errors, the resulting "t" x "t" sets of simultaneous equations becomes cumbersome and inefficient. A more elegant and efficient way to solve the key equations is the Massey-Berlekamp algorithm.

In this algorithm, the error-locator and the error-evaluator polynomial can be simultaneously computed. Once this is done, the Chien search is performed, and the error can be calculated using Forney's algorithm [8]. One drawback of the Massey-Berlekamp algorithm is the need to perform Galois field inversion every iteration. There are two approaches to mitigate this. First, one may pipeline the algorithm, so that only either a (mulitplication+addition) operation is performed, or an inversion is performed. This will reduce the maximum time needed between clock cycles, but increase the total number of clock cycles, thus increasing latency. The other approach is to put the inversion in the same clock cycle as the other operations, resulting in a slower clock speed.

An improvement to the Massey-Berlekamp algorithm is the inversionless Massey-Berlekamp algorithm [6]. In this algorithm, the need for the Galois field inversion has been eliminated by a clever modification to the equations used in every iteration. This results in an algorithm that requires the same number of iterations as the original, yet, since it does not perform inversion, can be speeded up. Unfortunately, the output of this algorithm is only one of the required polynomials, namely, the error-locator polynomial. The error-evaluator polynomial is calculated afterwards by the following equation :

$$\Lambda(x)S(x) \equiv \Omega(x) \bmod (x^{2t}) \tag{2.18}$$

Extra clock cycles are needed to perform such a polynomial multiplication, once again, resulting in a longer latency.

One of the important results of this thesis is an extension of the inversionless Massey-Berlekamp algorithm, which computes both the error-locator polynomial and the error-evaluator polynomial simultaneously. Thus latency is reduced, resulting in a faster decoder. The three algorithms will now be described, with a proof given for the extended inversionless Massey-Berlekamp algorithm.

2.6.1 Original Massey-Berlekamp Algorithm

Consider an RS code capable of correcting up to 't' errors, and without loss of generality, let m_0=0. The original Massey-Berlekamp algorithm is defined as follows [7]. First, initializations are made, let :

$$\Lambda(x)^{(0)} = 1, B(x)^{(0)} = 1, \Gamma(x)^{(0)} = 0, A(x)^{(0)} = x^{-1}, L^{(0)} = 0 \tag{2.19}$$

In the equations listed above, $\Lambda(x)$ is the error locator polynomial, B(x) is the error-locator support polynomial, $\Gamma(x)$ is the error-evaluator polynomial, and A(x) is the error-evaluator support polynomial. L is an integer variable. The algorithm proceeds iteratively, and the superscripts define the iteration level. Let the syndromes be represented by the syndrome polynomial :

$$S(x) = \sum_{j=0}^{2t-1} S_j \cdot x^j \tag{2.20}$$

The algorithm iterates for 2t steps. At the $(k+1)^{st}$ step, calculate the following term:

$$\Delta^{(k+1)} = \sum_{j=0}^{L^{(k)}} \Lambda_j^{(k)} \cdot S_{k-j} \tag{2.21}$$

Then, let

$$\Lambda(x)^{(k+1)} = \Lambda(x)^{(k)} - \Delta^{(k+1)} \cdot B(x)^{(k)} \cdot x \tag{2.22}$$

$$\Gamma(x)^{(k+1)} = \Gamma(x)^{(k)} - \Delta^{(k+1)} \cdot A(x)^{(k)} \cdot x \tag{2.23}$$

If $\Delta^{(k+1)}=0$ or $2L^{(k)}>k$ then,

$$B(x)^{(k+1)} = x \cdot B(x)^{(k)} \tag{2.24}$$

$$A(x)^{(k+1)} = x \cdot A(x)^{(k)} \tag{2.25}$$

$$L^{(k+1)} = L^{(k)} \tag{2.26}$$

otherwise

$$B(x)^{(k+1)} = \frac{\Lambda(x)^{(k)}}{\Delta^{(k+1)}} \tag{2.27}$$

$$A(x)^{(k+1)} = \frac{\Gamma(x)^{(k)}}{\Delta^{(k+1)}} \tag{2.28}$$

$$L^{(k+1)} = k + 1 - L^{(k)} \tag{2.29}$$

Galois field inversion is required in steps (2.27) and (2.28), however it is the inverse of the same value, $\Delta^{(k+1)}$. Thus, only one inverter is needed.

2.6.2 Inversionless Massey-Berlekamp Algorithm

An improvement to the Massey-Berlekamp algorithm was made in [6], by removing the need for performing the Galois field inversion. The original inversionless Massey-Berlekamp algorithm calculates the error-locator polynomial. The error-evaluator polynomial is calculated after. The algorithm can be stated as follows. Let :

$$l^{(0)} = 0 \text{ , and } \gamma^{(k)} = 1 \text{ if } k \leq 0$$

$$\mu(x)^{(0)} = 1, \lambda(x)^{(0)} = 1 \tag{2.30}$$

This inversionless algorithm also iterates for 2t steps. At the $(k+1)^{st}$ step, calculate the following term :

$$\delta^{(k+1)} = \sum_{j=0}^{l^{(k)}} \mu_j^{(k)} \cdot S_{k-j} \tag{2.31}$$

Then, let

$$\mu(x)^{(k+1)} = \gamma^{(k)} \cdot \mu(x)^{(k)} - \delta^{(k+1)} \cdot \lambda(x)^{(k)} \cdot x \tag{2.32}$$

If $\delta^{(k+1)}=0$ or $2l^{(k)}>k$ then,

$$\lambda(x)^{(k+1)} = x \cdot \lambda(x)^{(k)} \tag{2.33}$$

$$l^{(k+1)} = l^{(k)} \tag{2.34}$$

$$\gamma^{(k+1)} = \gamma^{(k)} \tag{2.35}$$

otherwise

$$\lambda(x)^{(k+1)} = \mu(x)^{(k)} \tag{2.36}$$

$$l^{(k+1)} = k+1-l^{(k)} \tag{2.37}$$

$$\gamma^{(k+1)} = \delta^{(k+1)} \tag{2.38}$$

As can be seen, in this algorithm it is not required to perform inversion. When the algorithm terminates, the error-locator polynomial is $\mu(x)$. At this point, the error-evaluator polynomial, $\Omega(x)$, can be calculated from the syndrome polynomial and the error-locator polynomial as shown below :

$$S(x) \cdot \mu(x) \equiv \Omega(x) \bmod x^{2t} \tag{2.39}$$

This operation takes place after the error locator polynomial is found, and thus, it adds to the latency of the decoder. The next section describes a new result, the extended inversionless Massey-Berlekamp algorithm, which computes both polynomials in tandem, thus saving the final step.

2.6.3 Extended Inversionless Massey-Berlekamp Algorithm

The inversionless Massey-Berlekamp algorithm can be extended to yield both the error-locator and the error-evaluator polynomial at the same time, also without the use of Galois field inverters. This eliminates the need for performing the required polynomial multiplication to determine the error-evaluator polynomial. The extension to the inversionless Massey-Berlekamp algorithm consists of the addition of the following steps to the inversionless algorithm :

$$\Omega(x)^{(k+1)} = \gamma^{(k)} \cdot \Omega(x)^{(k)} - \delta^{(k+1)} \cdot a(x)^{(k)} \cdot x \tag{2.40}$$

If $\delta^{(k+1)}=0$ or $2l^{(k)}>k$ then,

$$a(x)^{(k+1)} = x \cdot a(x)^{(k)} \tag{2.41}$$

otherwise

$$a(x)^{(k+1)} = \Omega(x)^{(k)} \tag{2.42}$$

The initial conditions are the same as those in the original Massey-Berlekamp algorithm, that is :

$$\Omega(x)^{(0)} = 0, a(x) = x^{-1} \tag{2.43}$$

In the equations above, $\Omega(x)$ is the error-evaluator polynomial , and $a(x)$ is the error-evaluator support polynomial. These are similar equations as that shown in (2.32), (2.33), and (2.36) of the inversionless Massey-Berlekamp algorithm. The following theorem is postulated, and then proved.

Theorem : the polynomials and scalars computed in the extended inversionless Massey-Berlekamp algorithm and the original Massey-Berlekamp algorithm are related as follows:

$$\mu(x)^{(k)} = \prod_{i=-1}^{k-1} \gamma^{(i)} \cdot \Lambda(x)^{(k)} \tag{2.44}$$

$$\Omega(x)^{(k)} = \prod_{i=-1}^{k-1} \gamma^{(i)} \cdot \Gamma(x)^{(k)} \tag{2.45}$$

$$\lambda(x)^{(k)} = \gamma^{(k)} \cdot B(x)^{(k)} \tag{2.46}$$

$$a(x)^{(k)} = \gamma^{(k)} \cdot A(x)^{(k)} \tag{2.47}$$

$$l^{(k)} = L^{(k)} \tag{2.48}$$

The proof for (2.43), (2.45), and (2.47) has been shown in the paper describing the inversionless Massey-Berlekamp algorithm which determines the error-locator polynomial [6]. The proof for (2.44), and (2.46) is presented here. The proof proceeds along similar lines as in [6], and is by induction. Clearly, for k=0, equations (2.44) and (2.46) hold. Consider the k^{th} iteration, the value calculated for δ.

$$\delta^{(k+1)} = \sum_{j=0}^{L^{(k)}} \mu_j^{(k)} \cdot S_{k-j} \tag{2.49}$$

Substituting for $\mu(x)$ from (2.43), and interchanging the summation and product, we find

$$\delta^{(k+1)} = \prod_{i=-1}^{k-1} \gamma^{(i)} \cdot \sum_{j=0}^{L^{(k)}} \Lambda_j^{(k)} \cdot S_{k-j} \tag{2.50}$$

However, by examining (2.20), this is equal to

$$\delta^{(k+1)} = \prod_{i=-1}^{k-1} \gamma^{(i)} \cdot \Delta^{(k+1)} \tag{2.51}$$

By definition (2.39), we have

$$\Omega(x)^{(k+1)} = \gamma^{(k)} \cdot \Omega(x)^{(k)} - \delta^{(k+1)} \cdot a(x)^{(k)} \cdot x \tag{2.52}$$

Substituting for $\Omega(x)$ from (2.44), $\delta^{(k+1)}$ from (2.50), a(x) from (2.46) we get :

$$\begin{aligned} \Omega(x)^{(k+1)} &= \prod_{i=-1}^{k} \gamma^{(i)} \cdot \Gamma(x)^{(k)} - \prod_{i=-1}^{k} \gamma^{(i)} \cdot \Delta^{(k+1)} \cdot A(x)^{(k)} \cdot x \\ \Omega(x)^{(k+1)} &= \prod_{i=-1}^{k} \gamma^{(i)} \cdot \left[\Gamma(x)^{(k)} - \Delta^{(k+1)} \cdot A(x)^{(k)} \cdot x \right] \end{aligned} \tag{2.53}$$

but from (2.22) the term $\left(\Gamma(x)^{(k)} - \Delta^{(k+1)} \cdot A(x)^{(k)} \cdot x \right)$ is equal to $\Gamma(x)^{(k+1)}$, thus

$$\Omega(x)^{(k+1)} = \prod_{i=-1}^{k} \gamma^{(i)} \cdot \Gamma(x)^{(k+1)} \tag{2.54}$$

This concludes the proof for equation (2.44). We now proceed on to the proof of equation (2.46). There are 2 cases to consider. Let us consider first the case where $\delta^{(k+1)}=0$. From (2.50) we see that $\delta^{(k+1)}=0$ if $\Delta^{(k+1)}=0$, and that $\delta^{(k+1)}\neq 0$ if $\Delta^{(k+1)} \neq 0$, because $\gamma^{(i)} \neq 0$ for i=-1,0,...,k-1. Hence, if $\delta^{(k+1)}= \Delta^{(k+1)}=0$ or if $2l^{(k)}=2L^{(k)}>k$, then by equation (2.40)

$$a(x)^{(k+1)} = x \cdot a(x)^{(k)} \tag{2.55}$$

Substituting for $a(x)^{(k)}$ from (2.46), and noting that for this case $\gamma^{(k+1)} = \gamma^{(k)}$ (see 2.34) and $A(x)^{(k+1)}=x \cdot A(x)^{(k)}$, (see 2.24) we get

$$a(x)^{(k+1)} = \gamma^{(k+1)} \cdot A(x)^{(k+1)} \tag{2.56}$$

This proves equation (2.46) for the case $\delta^{(k+1)}= 0$ or if $2 \cdot l^{(k)}>k$. Consider now the case where $\delta^{(k+1)} \neq 0$ and $2 \cdot l^{(k)} \leq k$. From (2.41)

$$a(x)^{(k+1)} = \Omega(x)^{(k)} \tag{2.57}$$

Substituting for $\Omega(x)$ from (2.44)

$$a(x)^{(k+1)} = \prod_{i=-1}^{k-1} \gamma^{(i)} \cdot \Gamma(x)^{(k)} \tag{2.58}$$

Substituting for $\gamma^{(i)}$ from (2.50) we get

$$a(x)^{(k+1)} = \frac{\delta^{(k+1)}}{\Delta^{(k+1)}} \cdot \Gamma(x)^{(k)} \tag{2.59}$$

For this case, from (2.38) we see that $\delta^{(k+1)}= \gamma^{(k+1)}$. Also, from (2.27) we have

$$A(x)^{(k+1)} = \frac{\Gamma(x)^{(k)}}{\Delta^{(k+1)}} \tag{2.60}$$

Thus (2.58) is reduced to :

$$a(x)^{(k+1)} = \gamma^{(k+1)} \cdot A(x)^{(k+1)} \tag{2.61}$$

This proves equation (2.46) for the case $\delta^{(k+1)}= 0$ or if $2\cdot l^{(k)}>k$. Thus the theorem is proved. This algorithm has been simulated and verified in the C language, and implemented successfully in VHDL.

At this point, both the error-locator and the error-evaluator polynomials have been computed. The Chien search will find the zeros of the error-locator polynomial, i.e., the error locations. The next step is to correct the errors. This is done using Forney's algorithm [8], which, for extensions of binary fields is :

$$e_{i_m} = \alpha^{i_m(1-m_0)} \frac{\Omega(\alpha^{-i_m})}{\Lambda'(\alpha^{-i_m})} \tag{2.62}$$

In this equation, α^{i_m} is the present location of the Chien search, X_i. The entire algorithm for a multi-error (t>2) correcting code is then stated as follows :

1. Calculate the 2t syndromes :

$$S(j) = r(\alpha^{j+m_0}), \quad j \in \{0...(2t-1)\} \tag{2.63}$$

2. If all of the syndromes are 0, then there is no error, and we STOP.
3. Calculate the error-locator and the error-evaluator polynomials using the extended inversionless Massey-Berlekamp algorithm.
4. Perform a Chien search on the entire received message. If the element at position (X_i) is found to be the location of an error, that is, when the Chien sum is 0, then
    a. calculate the error value Y_i using Forney's algorithm
    b. correct the error : corrected_element = received_element + Y_i

A block diagram of a pipelined multi-error (t>2) correcting RS decoder is shown in Figure 9 below. As before, an input strobe, RS_data_in_start is used to signify the beginning of a new input codeword. The codeword enters the decoder in bitwise parallel format, one symbol at a time. It first enters into the Calculate_Syndrome block, where the 2t syndromes are calculated. The control signal Error_Present is also generated. Finally, a strobe indicating the completion of the syndrome calculation is sent to the Key_Equation_Solver block, which implements the extended inversionless Massey-

Berlekamp algorithm. A delay block for the incoming RS symbols is necessary to compensate for the delays in the Calculate_Syndrome and Key_Equation_Solver blocks, and any startup delays in the Chien_Search block. The Key_Equation_Solver block calculates the error-locator and the error-evaluator polynomials. The Chien_Search block performs the function of finding the error locations, and correcting the data, using Forney's algorithm, when the Chien sum is 0. The evaluation of the polynomials needed for error correction can be pipelined. The corrected data and an output data strobe are the outputs of the RS decoder.

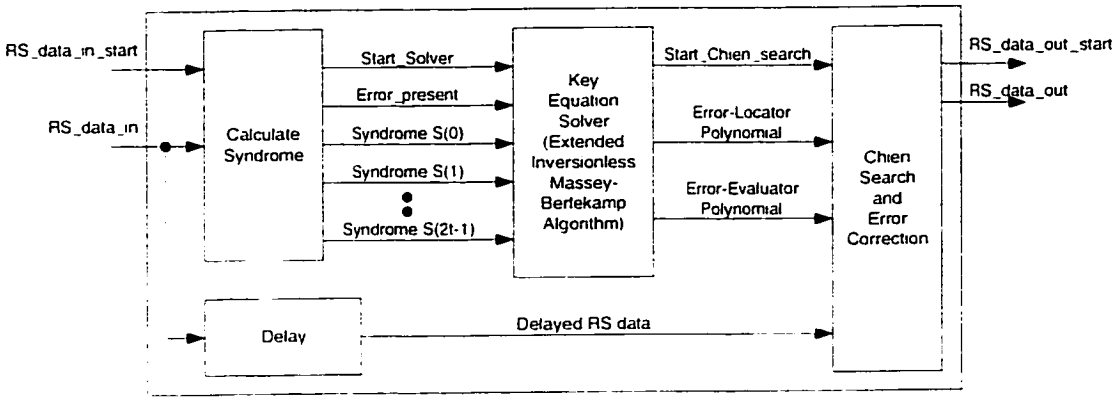

Figure 9. Multi-Error (>2) Correcting RS Decoder Block Diagram

3 Structure of the RS Encoder/Decoder Core Generator

This Chapter describes the contents of the source files in terms of the procedures and functions within the files.

3.1 File Structure

The entire functionality of the RS Encoder/Decoder Core Generator is contained in 5 files, as shown in the table below. These files contain 9864 lines of code.

filename	description
Rs_enc.pas	Core generator for an RS encoder
Rs_dec_t1_t2.pas	Core generator for an RS decoder for 1 or 2 errors.
Rs_dec.pas	Core generator for an RS decoder for more than 2 errors.
Rs_tb.pas	Core generator for an RS encoder or RS decoder testbench.
rs_utils.pas	Contains Functions and Procedures used by all other 4 files.

Table 3. RS Encoder/Decoder Core Generator Files.

The parameters of the desired encoder/decoder are entered via an ASCII file. The structure of this input file is as follows. The user enters the size, m, of the Galois field of the basic field elements from $GF(2^m)$ on the first line. The second line contains the coefficients of the irreducible polynomial, p(x), used to generate the field. These coefficients are from GF(2). These describe the specific RS(n,k) code to be designed for. The third line contains the value of n, and the fourth line contains the error correcting capability of the code, t. Note that k = n - 2t. The fifth line contains the value of the log of the first root, m_0, of the generator polynomial, g(x), formed by :

$$g(x) = \prod_{i=0}^{2t-1} (x + \alpha^{m_0 + i})$$

An example of such an input file is shown in the table below.

Line	Value	description
Line 1:	4	Galois field is GF(2^4)
Line 2:	1 0 0 1 1	Field generator polynomial is : $P(x) = x^4 + x + 1$
Line 3:	12	N=12
Line 4:	2	T=2 => RS(12,8)
Line 5:	0	$m_0 = 0,\ g(x) = \prod_{i=0}^{3}(x+\alpha^{i+1})$

Table 4. An Example of an RS Encoder/Decoder Core Generator Parameter File.

3.2 VHDL Core Generator for a RS Encoder

FILE : rs_enc.pas

procedure EncoderProcess;

There is only one procedure in this file. This procedure writes VHDL code for an RS encoder, using the algorithms specified in Section 2.1. The design of the code in VHDL is discussed in Section 5.2.

3.3 VHDL Core Generator for a RS Decoder for 1 or 2 Errors

The following table describes the procedures in file **rs_dec_t1_t2.pas.** This file is used to generate the VHDL code for a 1-error or a 2-error correcting RS decoder. The design of the code in VHDL is discussed in Section 7.

FILE : rs_dec_t1_t2.pas	
procedure	description
InputProcess	This procedure writes VHDL code for the syndrome calculation portion of an RS decoder, using the algorithms specified in Section 2.2.
ChienSearchProcess_T1	This procedure writes VHDL code for the Chien search portion of a 1-error correcting RS decoder, using the algorithms specified in Section 2.3.
RSTopProcess_T1	This procedure writes VHDL code which interconnects InputProcess and ChienSearchProcess_T1. In addition, internal signals from the InputProcess are brought out to be verified by the testbench, as well as the outputs of the ChienSearchProcess_T1.
RSTopProcess_T1_syn	This procedure writes VHDL code which interconnects InputProcess and ChienSearchProcess_T1. No other internal signals are brought out, making this the VHDL code used for synthesis for a 1-error correcting RS decoder.
ChienSearchProcess_T2	This procedure writes VHDL code for the Chien search portion of a 2-error correcting RS decoder, using the algorithms specified in Section 2.3.
RSTopProcess_T2	This procedure writes VHDL code which interconnects InputProcess and ChienSearchProcess_T2. In addition, internal signals from the InputProcess are brought out to be verified by the testbench, as well as the outputs of the ChienSearchProcess_T2.
RSTopProcess_T2_syn	This procedure writes VHDL code which interconnects InputProcess and ChienSearchProcess_T2. No other internal signals are brought out, making this the VHDL code used for synthesis for a 2-error correcting RS decoder.

3.4 VHDL Core Generator for a RS Decoder for More Than 2 Errors

The following table describes the procedures in file **rs_dec.pas.** This file is used to generate the VHDL code for a RS decoder capable of correcting more than 2 errors. The design of the code in VHDL is discussed in section 8.

FILE : rs_dec.pas	
procedure	description
InputProcess	This procedure writes VHDL code for the syndrome calculation portion of an RS decoder, using the algorithms specified in Section 2.2.
MBSolverProcess	This procedure writes VHDL code for the extended inversionless Massey-Berlekamp portion of the RS decoder, using the algorithms specified in Section 2.6.3.
ChienSearchProcess	This procedure writes VHDL code for the Chien search portion of the RS decoder, using the algorithms specified in Section 2.3.
RenameProcess	This procedure renames internal signals for verification by the testbench.
RSTopProcess	This procedure writes VHDL code which interconnects InputProcess, MBSolverProcess and ChienSearchProcess. In addition, internal signals from the InputProcess and MBSolverProcess are brought out to be verified by the testbench, as well as the outputs of the ChienSearchProcess.
RSTopProcessSyn	This procedure writes VHDL code which interconnects InputProcess, MBSolverProcess and ChienSearchProcess. No other internal signals are brought out, making this the VHDL code used for synthesis for a greater than 2-error correcting RS decoder.

3.5 Test Bench

FILE : rs_tb.pas

There are no procedures within this file. It calls the appropriate testbench procedure within the utility file rs_utils.pas, depending on whether an RS encoder or a RS decoder has been selected. These procedures generate test vectors for the VHDL testbench. The testbench has been written in a generic manner; all that needs to be specified are the values of number of bits in the Galois field (GFPower), the size of the codeword, RS_N, and the error-correcting capability of the code, RS_T. These are written to an ASCII file, named **generics.txt**, which is read in by the VHDL simulator. An example of such a file is shown below :

```
assign 8 GFPower
assign 22 RS_N
assign 2 RS_T
```

Figure 10. Example of a Generics.txt File Used by the Testbench.

The structure of the testbenches are discussed in section 6.4.

3.6 Utility Functions and Procedures

The file **rs_utils.pas** contains all of the utility functions and procedures used by the rest of the program. They include the basic low level functions such performing Galois field addition, multiplication and inversion, and all the way to procedures which generate the Galois field, or generate testbench vectors. The following section lists the functions and procedures in rs_utils.pas, along with a brief description.

function add(a, b : integer) : integer; This function adds 2 Galois field elements. The Galois field must have been specified as described in Section 3.1.

function antilog(a : integer) : integer; This function finds the antilog of an integer, a, returning the Galois field element which is α^a.

function bin(ix : integer; length : integer) : string; This function returns a string of the binary representation of an integer value for a given length. This is used to generate std_logic_vector (standard logic vector) values in the VHDL code.

function gfdiv(a, b : integer) : integer; This function returns the result of a Galois field division.

function hex(a : integer) : char; This function returns a hex character for an integer in the range of 0 to 15;

function hex2(a : integer) : string; This function returns 2 hex characters for an integer in the range of 0 to 255;

function inv(a : integer) : integer; This function returns the inverse of a Galois field element.

function log(a : integer) : integer; This function returns the logarithm of a Galois field element.

function mul(a, b : integer) : integer; This function returns the result of a Galois field multiplication.

function rand_code : integer; This function generates a random Galois field element.

function rand_error : integer; This function generates a random error value.

function rand_index(N:integer) : integer; This function generates a random index. Since an RS codeword is represented as an array of integers, this functions points point to one of the Galois field elements in the codeword.

function StrAlpha(a:integer) : string; This function returns a string representing the power to a Galois field element. For example, for the value of α^{27}, the function returns " a27".

procedure AddRoot(root : integer); This procedure adds a root to an existing polynomial. This is used to generate the code generator polynomial.

procedure CalculateSyndrome(rx : codeword; var syndrome : syndrome_type); This procedure calculates the syndrome of a codeword.

procedure ChienSearch; This procedure performs the Chien search for a codeword. The algorithm used is discussed in section 2.3.

procedure decode (inp : codeword; var out : codeword); This procedure decodes a codeword. First the syndrome is calculated. If there are errors, the key equation is solved using the extended inversionless Massey-Berlekamp algorithm. Finally, a Chien search is used to find the error locations, and Forney's algorithm is used to correct the errors.

procedure encode (inp : codeword; var out : codeword); This procedure encodes an incoming set of k Galois field elements into a (n,k) Reed-Solomon codeword. In the process, the 2t = n-k parity symbols are appended.

procedure ExtendedInversionlessMasseyBerlekamp(synd : syndrome_type); This procedure solves the key equation from the syndromes of an incoming codeword using the extended inversionless Massey-Berlekamp algorithm, as discussed in section 2.6.3.

procedure GenerateField(GFP : integer; FP : field_polynomial_type); This procedure generates the Galois field based on the field generator polynomial. In the process, the log and antilog tables are generated. These tables are used to implement Galois field multiplication, inversion, and division.

procedure GetPolynomial; This procedure calculates the code generator polynomial using the power of the initial root, m_0, and the error correcting capability of the code, t.

procedure get_parameters(filename : string); This procedure reads in the parameters of the Reed-Solomon code, as discussed in section 3.1.

procedure inject_RS_errors(N:integer); This procedure injects a given number of errors in a codeword. This is used in generating test vectors for the RS decoder.

procedure tb_decoder; This procedure generates the input test vectors and expected values for a RS decoder.

procedure tb_encoder; This procedure generates the input test vectors and expected values for a RS encoder.

procedure VHDL_1_GFConstant(var f1 : text; power : integer); This procedure generates the VHDL code corresponding to the constant definition of a particular Galois field value.

procedure VHDL_GFConstants(var f1 : text; min_element, max_element:integer); This procedure generates the VHDL code corresponding to the constant definition of a series of sequential Galois field values.

procedure VHDL_GFConstants_0_1(var f1 : text); This procedure generates the VHDL code corresponding to the constant definition of 0 and 1.

procedure VHDL_GFConstants_from_generator_polynomial(var fl : text); This procedure writes VHDL comments describing the RS code generator polynomial. Each coefficient if printed out in power form.

procedure VHDL_add(var fl : text); This procedure generates the VHDL code for the Galois field addition of 2 elements, as discussed in Section 4.1.

procedure VHDL_mul(var fl : text); This procedure generates the VHDL code for the Galois field multiplication of 2 elements, as discussed in Section 4.2.

procedure VHDL_inv(var fl : text); This procedure generates the VHDL code for the Galois field inversion of an element, as discussed in Section 4.3.

4 VHDL Implementation of Galois Field Operations

This Chapter will discuss the VHDL implementation of the basic Galois field operations that are needed in either a Reed-Solomon encoder or decoder. These basic operations consist of addition, multiplication, and inversion over GF(2^m). The information required to implement these operations are the value of m, since the operations are performed over the Galois field GF(2^m), and the field generator polynomial p(x). The field generator polynomial p(x) is an irreducible polynomial of degree m-1, with coefficients from GF(2). The equations for each of these operations will be discussed for a general value of m and p(x), and VHDL code examples will be given for specific cases. The discussions of the VHDL code, for all of the code presented in this thesis, will feature code walkthroughs, and block diagrams.

4.1 VHDL Implementation of a Galois Field Adder

Addition and subtraction are the simplest of Galois field operators over GF(2^m). Let **a** and **b** be two elements from GF(2^m), that is,

$$a = a_{n-1} \bullet x^{n-1} + \cdots + a_1 \bullet x + a_0 = \sum_{i=0}^{n-1} a_i \bullet x^i, \text{ and } b = b_{n-1} \bullet x^{n-1} + \cdots + b_1 \bullet x + b_0 = \sum_{i=0}^{n-1} b_i \bullet x^i$$

Let **c** be the result of the addition of **a** and **b**; c is just the polynomial addition of the polynomial representations of **a** and **b** on a term-by-term basis.

$$c = a + b = c_{n-1} \bullet x^{n-1} + \cdots + c_1 \bullet x + c_0 = \sum_{i=0}^{n-1} c_i \bullet x^i = \sum_{i=0}^{n-1} (a_i + b_i) \bullet x^i \qquad (3.1)$$

where the addition is performed over GF(2). The addition table for 2 elements A and B from GF(2) is shown in Table 5 below :

A	B	C=A+B
0	0	0
0	1	1
1	0	1
1	1	0

Table 5. Binary addition.

In terms of a Boolean equation, we see that C = A xor B, where xor is the binary exclusive-or operation. This is performed on a bitwise basis. An example of an addition function over GF(2^8) written in VHDL is shown below.

```
Constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);

function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  d(6) := (b(6) xor c(6));
  d(7) := (b(7) xor c(7));
  return d;
end add;
```

Figure 11. VHDL Code for Galois Field Addition, GF(2^8)

Since we are dealing with elements from GF(2^m), subtraction and addition are equivalent, and hence the same VHDL code can be used for both addition and subtraction.

4.2 VHDL Implementation of a Galois Field Multiplier

The generation of VHDL code for the multiplication of 2 elements over GF(2^m) is accomplished by first forming the product of the polynomial representations of **a** and **b**

modulo the primitive irreducible polynomial. and the converting the expression into combinatorial equations for binary addition and multiplication. Let **a** and **b** be two elements from GF(2^m), that is,

$$a = a_{n-1} \bullet x^{n-1} + \cdots + a_1 \bullet x + a_0 = \sum_{i=0}^{n-1} a_i \bullet x^i, \text{ and } b = b_{n-1} \bullet x^{n-1} + \cdots + b_1 \bullet x + b_0 = \sum_{i=0}^{n-1} b_i \bullet x^i$$

Let **c** be the result of the multiplication of **a** and **b**.

$$c = a \bullet b = c_{2n-2} \bullet x^{2n-2} + \cdots + c_1 \bullet x + c_0 = \sum_{i=0}^{2n-2} c_i \bullet x^i, \text{ with } c_i = \sum_{\substack{0 \le j,k \le n-1 \\ j+k=i}} a_j b_k \qquad (3.2)$$

where the additions and multiplications are performed over GF(2). The resulting polynomial is then reduced modulo p(x). The addition table for 2 elements A and B from GF(2) was shown previously. The multiplication table for 2 elements A and B from GF(2) is shown in Table 6 below :

A	B	C=AB
0	0	0
0	1	0
1	0	0
1	1	1

Table 6. Binary multiplication.

In terms of a Boolean equation, C = A and B. An example of this process will clarify the method. Consider a field GF(2^4) formed by the primitive irreducible polynomial $p(x)=x^4+x+1$. The elements of this field are shown in the following table in vector form, and both the exponential (power) and polynomial forms.

Elements from GF(2^4), p(x)=x^4+x+1					
power	Polynomial	vector = (a_3, a_2, a_1, a_0)			
		a_3	a_2	a_1	a_0
0	0	0	0	0	0
1	1	0	0	0	1
α	α	0	0	1	0
α^2	α^2	0	1	0	0
α^3	α^3	1	0	0	0
α^4	$\alpha+1$	0	0	1	1
α^5	$\alpha^2+\alpha$	0	1	1	0
α^6	$\alpha^3+\alpha^2$	1	1	0	0
α^7	$\alpha^3+\alpha+1$	1	0	1	1
α^8	α^2+1	0	1	0	1
α^9	$\alpha^3+\alpha$	1	0	1	0
α^{10}	$\alpha^2+\alpha+1$	0	1	1	1
α^{11}	$\alpha^3+\alpha^2+\alpha$	1	1	1	0
α^{12}	$\alpha^3+\alpha^2+\alpha+1$	1	1	1	1
α^{13}	$\alpha^3+\alpha^2+1$	1	1	0	1
α^{14}	α^3+1	1	0	0	1

Table 7. Elements of GF(2^4), p(x)=x^4+x+1

Let **a** and **b** be two elements from GF(2^4), that is, $a = a_3 \bullet x^3 + a_2 \bullet x^2 + a_1 \bullet x + a_0$, and $b = b_3 \bullet x^3 + b_2 \bullet x^2 + b_1 \bullet x + b_0$. Let **c** be the result of the multiplication of **a** and **b**.

$c = a \bullet b = c_6 \bullet x^6 + c_5 \bullet x^5 + c_4 \bullet x^4 + c_3 \bullet x^3 + c_2 \bullet x^2 + c_1 \bullet x + c_0$, with $c_i = \sum_{\substack{0 \le j,k \le 3 \\ j+k=i}} a_j b_k$, that is,

$c_0 = a_0 \bullet b_0$
$c_1 = a_0 \bullet b_1 + a_1 \bullet b_0$
$c_2 = a_0 \bullet b_2 + a_1 \bullet b_1 + a_2 \bullet b_0$
$c_3 = a_0 \bullet b_3 + a_1 \bullet b_2 + a_2 \bullet b_1 + a_3 \bullet b_0$
$c_4 = a_1 \bullet b_3 + a_2 \bullet b_2 + a_3 \bullet b_1$
$c_5 = a_2 \bullet b_3 + a_3 \bullet b_2$
$c_6 = a_3 \bullet b_3$

In order to reduce the polynomial $c = a \bullet b = c_6 \bullet x^6 + c_5 \bullet x^5 + c_4 \bullet x^4 + c_3 \bullet x^3 + c_2 \bullet x^2 + c_1 \bullet x + c_0$ modulo the primitive irreducible polynomial $p(x)=x^4+x+1$, we note the following :

1. $x^4 = x + 1$
2. $x^5 = x^2 + x$
3. $x^6 = x^3 + x^2$

Thus, the polynomial $c = a \bullet b = c_6 \bullet x^6 + c_5 \bullet x^5 + c_4 \bullet x^4 + c_3 \bullet x^3 + c_2 \bullet x^2 + c_1 \bullet x + c_0$ can be reduced to $c = a \bullet b = (c_3 + c_6) \bullet x^3 + (c_2 + c_5 + c_6) \bullet x^2 + (c_1 + c_4 + c_5) \bullet x + (c_0 + c_4)$, or

$$
\begin{aligned}
c = {} & (a_0 \bullet b_3 + a_1 \bullet b_2 + a_2 \bullet b_1 + a_3 \bullet b_0 + a_3 \bullet b_3) \bullet x^3 \\
& + (a_0 \bullet b_2 + a_1 \bullet b_1 + a_2 \bullet b_0 + a_2 \bullet b_3 + a_3 \bullet b_2 + a_3 \bullet b_3) \bullet x^2 \\
& + (a_0 \bullet b_1 + a_1 \bullet b_0 + a_1 \bullet b_3 + a_2 \bullet b_2 + a_3 \bullet b_1 + a_2 \bullet b_3 + a_3 \bullet b_2) \bullet x \\
& + (a_0 \bullet b_0 + a_1 \bullet b_3 + a_2 \bullet b_2 + a_3 \bullet b_1)
\end{aligned}
\qquad (3.3)
$$

These are the bitwise combinatorial equations for multiplication over $GF(2^4)$, with the primitive irreducible polynomial $p(x)=x^4+x+1$. The resulting VHDL code is shown below, with the binary "xor" used for bitwise addition, and the binary "and" used for bitwise multiplication :

```
constant GFPower : integer := 4;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(3)) xor (b(2) and c(2))
          xor (b(3) and c(1));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(1) and c(3))
          xor (b(2) and c(2)) xor (b(3) and c(1)) xor (b(2) and c(3))
          xor (b(3) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(2) and c(0)) xor
          (b(2) and c(3)) xor (b(3) and c(2)) xor (b(3) and c(3));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor
          (b(3) and c(0)) xor (b(3) and c(3));
  return d;
end mul;
```

Figure 12. VHDL Code for Galois Field Multiplication, $GF(2^4)$, $p(x)=x^4+x+1$

4.3 VHDL Implementation of a Galois Field Inverter

Unlike Galois field multiplication, Galois field inversion is generally difficult, if not impossible, to reduce into reasonable sized Boolean equation. The easiest approach is to list the table of inversion, and let the synthesis tool generate the required equations. This results in a lookup table of size 2^m words with m bits per word, where m is the degree of the field generator polynomial. For inversion over $GF(2^4)$, with $p(x)=x^4+x+1$, the VHDL code using "`case`" statements is shown in Figure 13. An alternate implementation using "`if-then-else`" statements is shown in Figure 14. Both methods yield identical simulation results, but the "`if-then-else`" method yielded better synthesis results for Xilinx FPGAs.

```
function inv (b : in std_logic_vector (3 downto 0)) return std_logic_vector is
  variable d : std_logic_vector (3 downto 0);
begin
  d := "0000";
  case b is
      when "0000" => d := "0000";
      when "0001" => d := "0001";
      when "0010" => d := "1001";
      when "0011" => d := "1110";
      when "0100" => d := "1101";
      when "0101" => d := "1011";
      when "0110" => d := "0111";
      when "0111" => d := "0110";
      when "1000" => d := "1111";
      when "1001" => d := "0010";
      when "1010" => d := "1100";
      when "1011" => d := "0101";
      when "1100" => d := "1010";
      when "1101" => d := "0100";
      when "1110" => d := "0011";
      when "1111" => d := "1000";
      when others   => d := "0000";
  end case;
  return d;
end inv;
```

Figure 13. VHDL Code for Galois Field Inversion, $GF(2^4)$, $p(x)=x^4+x+1$

```
function inv (b : in std_logic_vector) return std_logic_vector is
  variable d : std_logic_vector (3 downto 0);
begin
  d := "0000";
     if (b="0001") then d := "0001";
  elsif (b="0010") then d := "1001";
  elsif (b="0011") then d := "1110";
  elsif (b="0100") then d := "1101";
  elsif (b="0101") then d := "1011";
  elsif (b="0110") then d := "0111";
  elsif (b="0111") then d := "0110";
  elsif (b="1000") then d := "1111";
  elsif (b="1001") then d := "0010";
  elsif (b="1010") then d := "1100";
  elsif (b="1011") then d := "0101";
  elsif (b="1100") then d := "1010";
  elsif (b="1101") then d := "0100";
  elsif (b="1110") then d := "0011";
  elsif (b="1111") then d := "1000";
  end if;
  return d;
end inv;
```

Figure 14. Alternate VHDL Code for Galois Field Inversion, $GF(2^4)$, $p(x)=x^4+x+1$

5 VHDL Design of a RS Encoder

The topic of this thesis was to write a program that would generate synthesizable VHDL code for any arbitrary Reed-Solomon encoder or decoder. The previous sections have concentrated on the theoretical aspects of encoding and decoding. One of the goals of the program was also to generate "generic" VHDL code as much as possible, and rely on the definition of constants and signals as the primary method of creating a specific encoder or decoder. This section will discuss how the different parts of a RS encoder are implemented in VHDL.

The parameters that must be entered, in order to specify an RS encoder, are :

1. the size of the extension of the Galois field GF(2) into GF(2^m), i.e., the value m.
2. the irreducible primitive polynomial used to generate the field, p(x), by specifying the coefficients of the polynomial from GF(2)
3. the log of the initial root of the code generator polynomial, m_0.
4. the error-correcting capability of the code.

The number of message symbols, k, will be made programmable. In the following sections, a brief discussion of the encoder structure will be followed by a detailed exposure of the VHDL code for the various parts of the encoder.

5.1 Encoder Overview

Figure 15 illustrates a block diagram of an RS encoder. The 2t storage elements are m-bit registers, labeled b_0 through to b_{2t-1}, again where m is the degree of the field generator polynomial, and t is the error correcting capability of the code. These are also called the parity registers. The circuit performs polynomial division of the message polynomial, m(x) by the field generator polynomial, g(x). The remainder of the division, b(x) is

stored in the 2t parity registers. The codeword polynomial, c(x), is the concatenation of m(x) followed by b(x).

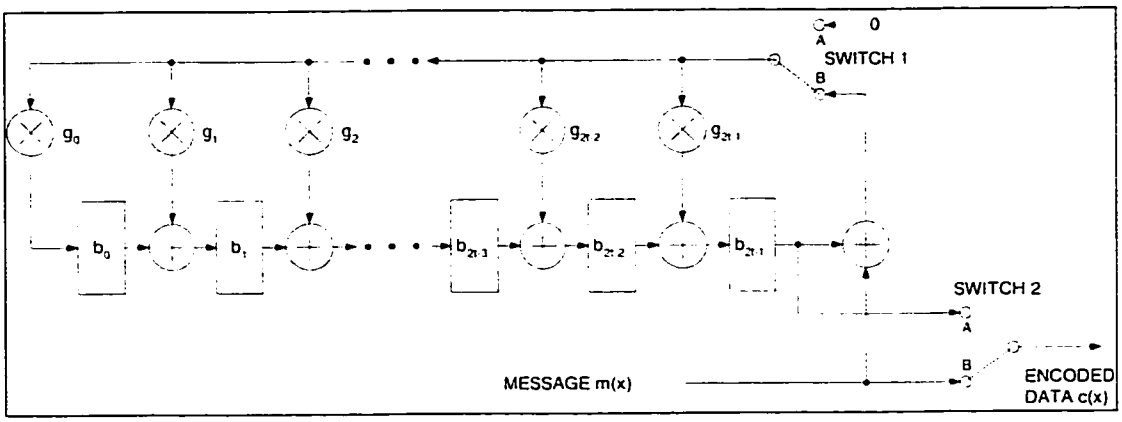

Figure 15. Block Diagram of a Generic Reed-Solomon Encoder

The operation of the circuit, as can be found in any textbook in error control coding [10], is described as follows :

1. The initial state of the registers is 0 for each register.
2. Without lack of generality, we assume that the message is divided equally into m-bit words, where each word is now considered a Galois field element. Each m-bit word is then associated with an increasing power of x, starting with $x^0=1$, and ending in x^{k-1}, thus forming a polynomial, m(x) over $GF(2^m)$. The message polynomial is of degree k-1, that is, there are k coefficients. Each coefficient enters the circuit one coefficient every clock cycle, with the most significant coefficient entering first.
3. For the first k clock cycles, corresponding to when the message is entering and the remainder is being calculated, switches 1 and 2 are in position B. The encoded data thus corresponds with the message polynomial for the first k clock cycles. During these first k clock cycles, the remainder is being calculated in the registers.
4. When the message has finished entering into the encoder, switches 1 and 2 are set to position A. Since the output of switch 1 is 0, the resulting multiplications are

0, and the resulting additions, and consequently the inputs to the registers are just the values of the preceding register. In the case of the first register, its input is 0, since it sees the output of the switch. The output of switch 2 is now the output of the last register. Thus the entire remainder polynomial, b(x) is shifted out one element at a time. Once all the register contents have been shifted out, the codeword generation is complete.

5.2 VHDL Implementation of an RS Encoder

5.2.1 Encoder Timing Diagram.

Consider now the implementation of the above algorithm in VHDL. The design is partitioned into data flow and control signals. The timing diagram showing the input/output interface of the RS encoder is shown in Figure 16. A signal called `input_strobe` is used to start the encoding process. It is a single clock cycle pulse that precedes the message data, called `data_in`, by one clock cycle. The data consists of the k coefficients of the message polynomial. Another requirement for the encoder is the size of the message; this encoder can accommodate different sized messages. The message size is conveyed through the signal `data_size`. The timing diagram shows two control signals that are used internally, namely, `do_calc` and `count`. These will be discussed in more detail in the next section. Finally, there are two output signals, `output_strobe`, and `data_out`, whose names clearly describe their function. The signal `output_strobe` is active for 1 clock cycle, and it occurs 1 clock cycle before the output data.

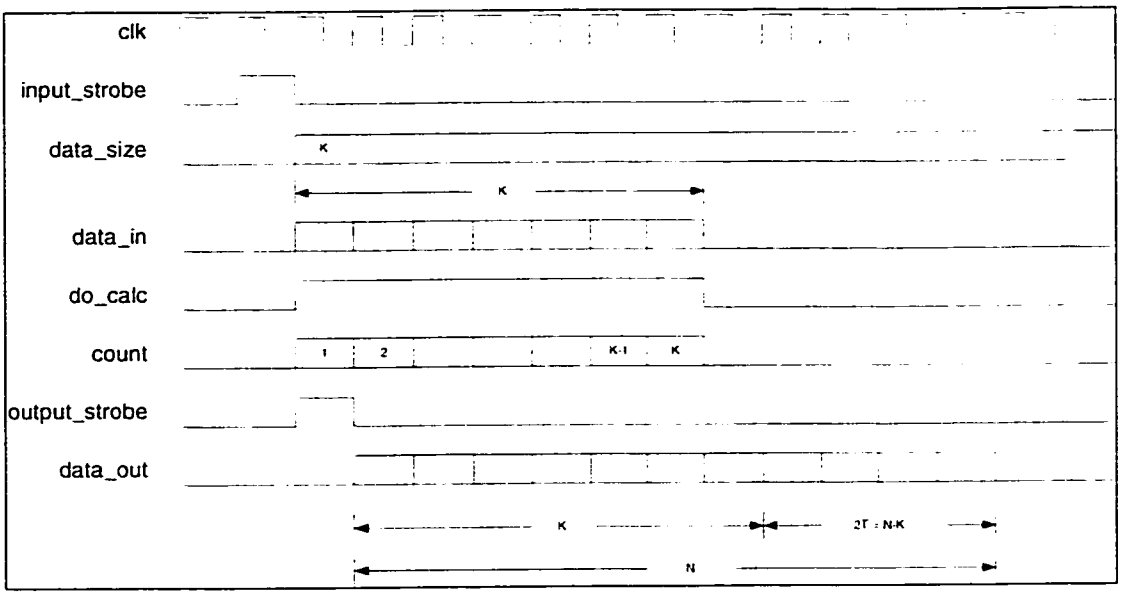

Figure 16. RS Encoder Timing Diagram

5.2.2 Encoder Control Logic

Control of the encoder is accomplished through the `do_calc` control signal and the counter `count`. These signals, along with the registers are used to perform the polynomial division. The signal `input_strobe` starts off the encoding process. When it arrives, the counter `count` is set to the value 1, and then is incremented every clock cycle, as long as `do_calc` is '1', otherwise it is set to the value 0. In addition, the `input_strobe` causes the signal `do_calc` to be set to '1'. The signal `do_calc` is then only set to '0' when the counter `count` reaches the value prescribed by `data_size`. A separate process is used for each of these signals. The VHDL code for these two processes is shown in Figure 17 for the case specific case of m=4, $p(x)=x^4+x+1$. Notice that the code for these two processes is written in such a manner that it can be reused for any size of Galois field element, only the initial definitions require adjustment for specific codes. Also notice that all sequential signals are coded to perform a synchronous reset when the signal `reset_n` is '0'. This is true for all sequential signals in either the encoder or decoder design. The initialization section defines the signals needed, as well as constants, types and subtypes. In particular, the

size of the Galois field elements, m, is denoted by the constant GFPower, indicating the power of the Galois field. The subtype Galois_Field_element is then defined using the constant GFPower. The constant RS_T defines the error-correcting capability of the code. Other constants include Galois field equivalents of 0, 1 and the field generator polynomial coefficients. Finally, the parity registers are defined as an array of Galois field elements.

```
constant GFPower : integer := 4;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant    RS_T  : integer := 1;
constant     zero : Galois_Field_element := "0000";
constant      one : Galois_Field_element := "0001";
constant   alpha6 : Galois_Field_element := "1100";
constant  alpha14 : Galois_Field_element := "1001";
signal DoCalc : std_logic;
signal count : std_logic_vector ((GFPower-1) downto 0);
signal sum : Galois_Field_element;
type parity_reg_type is array(0 to (2*RS_T-1)) of Galois_Field_element;
signal parity_reg : parity_reg_type;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      DoCalc <= '0';
    elsif (input_strobe = '1') then
      DoCalc <= '1';
    elsif (count=data_size) then
      DoCalc <= '0';
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      count <= (others => '0');
    elsif (input_strobe = '1') then
      count <= one;
    elsif (DoCalc = '1') then
      count <= count + 1;
    else count <= (others => '0');
    end if;
  end if;
end process;
```

Figure 17. VHDL Code of Encoder Control Signals

5.2.3 Parity Registers and Output Signals

The encoder consists of a set (2t) of parity registers, each being m bits wide. When the signal input_strobe arrives it sets the parity registers to 0. Then, when the do_calc signal is '1', the parity registers perform the required multiplications and additions using the combinatorial signal sum, which is the sum of the input data and the last parity register, as shown in Figure 15. This actions is the heart of the polynomial division process. When the division is complete, the parity registers are shifted out one at a time. The VHDL code for the single error correcting code is shown in Figure 18.

```
process (parity_reg, data_in)
begin
  sum <= add(parity_reg((2*RS_T-1)),data_in);
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to (2*RS_T-1) loop
        parity_reg(i) <= (others=>'0');
      end loop;
    elsif (input_strobe = '1') then
      for i in 0 to (2*RS_T-1) loop
        parity_reg(i) <= (others=>'0');
      end loop;
    elsif (DoCalc = '1') then
      parity_reg(  1) <= add(parity_reg(  0),mul(sum,alpha14));
      parity_reg(  0) <= mul(sum,alpha6);
    else
      for i in (2*RS_T-1) downto 1 loop
        parity_reg(i) <= parity_reg(i-1);
      end loop;
      parity_reg(0) <= (others=>'0');
    end if;
  end if;
end process;
```

Figure 18. VHDL Code of Encoder Parity Registers

Figure 19 shows the VHDL code for the output data (data_out), and the signal output_strobe. The output data is set to the input data when the signal do_calc is '1', otherwise it is set to the output of the last parity register. This has the effect of concatenating the message polynomial with the parity elements, thus forming the complete codeword. The signal output_strobe is simply a delayed version of the input strobe, thus satisfying the requirement of the output interface.

```
process(clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      data_out <= (others => '0');
    elsif (DoCalc='1') then
      data_out <= data_in;
    else
      data_out <= parity_reg((2*RS_T-1));
    end if;
  end if;
end process;

process(clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      output_strobe <= '0';
    else
      output_strobe <= input_strobe;
    end if;
  end if;
end process;
```

Figure 19. VHDL Code of Encoder Output Signals

The complete VHDL code for one RS encoder is shown in Appendix A. This includes the entity-architecture definitions as well as the functional code.

6 Synthesis and Test Results for RS Encoders

6.1 General Remarks

This Chapter will discuss the synthesis results for RS encoders in terms of speed and area. The technology that was chosen for synthesis is that of the Xilinx Virtex series of Field Programmable Gate Arrays (FPGAs), specifically the XCV1000E. This series of FPGAs is a good candidate for the implementation of communications related algorithms. They consist of parts that have an equivalent gate count of 1 million gates, and register-to-register speeds of well over 150 MHz. The speed is given by the maximum clock frequency in MHz, while the area is measured by the number of slices. Slices are the basic building block of the FPGA, consisted of 2 flip-flops and 2 5-bit LUTs for implementing combinatorial logic. The choice of synthesis target is arbitrary; in fact ASICs or others FPGAs, such as Actel FPGAs are also valid targets. Thus, it is not the absolute performance to be considered, although in terms of absolute performance the suggested target technology must be capable of implementing a typical RS encoder or decoder. Rather, it is the relative performance that will be discussed, taking into perspective the parameters of the particular encoder/decoder.

RS encoder designs were synthesized for different error-correcting ability codes across 3 values of the degree of the field generator polynomial, m, namely, 4, 6 and 8. The results are discussed in the next 2 sections.

6.2 RS Encoder Synthesis Speed Results

Several RS encoder designs were chosen as candidate designs. The error-correcting capability ranged from 1 to 7. This was applied to codes from $GF(2^4)$, $GF(2^6)$, and $GF(2^8)$. The synthesis flow consisted of the Synplicity synthesis tool (from Synplicity) and Xilinx software for the final place-and-route and timing analysis. The speed results are tabulated in Table 8 and shown graphically in Figure 20.

Error-correcting ability	$GF(2^4)$		$GF(2^6)$		$GF(2^8)$	
	Clock speed (MHz)	Bit Rate (Mbps)	Clock speed (MHz)	Bit Rate (Mbps)	Clock speed (MHz)	Bit Rate (Mbps)
1	192.5	770.0	188.4	1130.4	169.2	1353.6
2	168.1	672.5	168.1	1008.7	140.4	1123.1
3	211.1	844.6	168.1	1008.7	140.4	1123.1
4	196.4	785.5	140.4	842.3	140.4	1123.1
5	185.8	743.2	157.0	941.9	159.6	1276.5
6	168.1	672.5	159.6	957.4	159.6	1276.5
7	180.3	721.1	159.6	957.4	149.4	1194.9

Table 8. Maximum Encoder Speed (MHz) vs. Error Correcting Ability

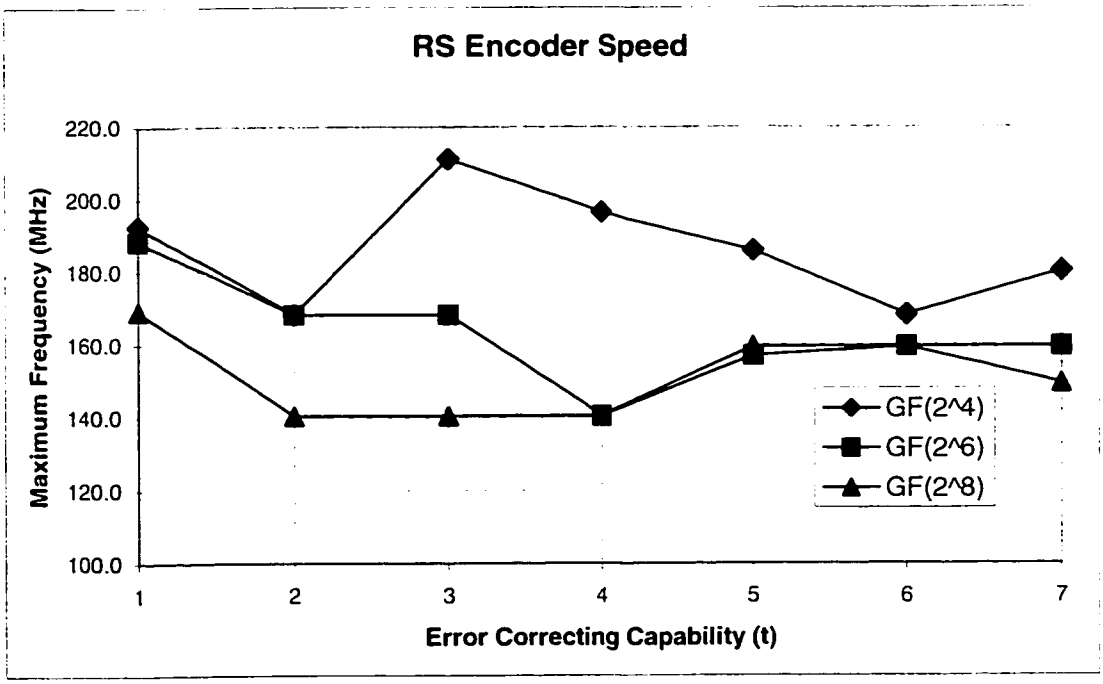

Figure 20. Reed-Solomon Encoder Maximum Speed

The results show that for a given field generator polynomial, the maximum speed does not vary with the error-correcting capability. There is also very little variation with respect to the field generator polynomial, although, it can be said that on average, a code

with a larger field generator polynomial runs slightly slower than one with a smaller field generator polynomial. This can be attributed to the fact that the Galois field multipliers become increasingly more complex, and as a result, increasingly more slower with larger field generator polynomials. It can be seen that encoder bit rates of 10^9 bits per second are achievable.

6.3 RS Encoder Synthesis Area Results

The area results are tabulated in Table 9 and shown graphically in Figure 21. It is clear from the results that area is a linear function of the error-correcting capability. This is to be expected, since all that is being added for increasing error-correcting capability is the number of parity registers. The encoder area increases with increasing error-correcting capability, again, as a consequence of ever more complex Galois field multipliers.

Error-correcting ability	GF(2^4)	GF(2^6)	GF(2^8)
1	15	24	32
2	25	47	67
3	31	59	84
4	36	64	102
5	41	78	112
6	46	82	132
7	74	97	149

Table 9. Encoder Area (in slices) vs. Error Correcting Ability

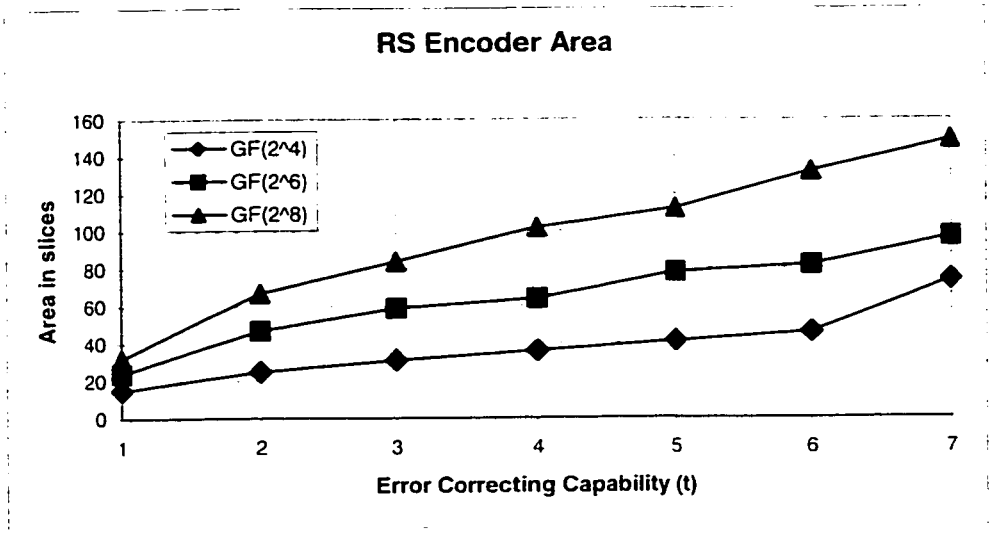

Figure 21. Reed-Solomon Encoder Area (in slices)

6.4 Encoder Testbench

The performance of the encoders described above was verified with a VHDL testbench. The testbench has the structure shown in Figure 22. The clock and stimulus generator provides the 4 input signals to the encoder. The clock `clk` is just a square wave of 10 MHz. The other 3 signals are generated by reading an ASCII text file containing the logic levels for the signals. The encoding algorithm was written in a high level language, which was then used to generate the stimulus test vectors (`input_strobe`, `data_size`, and `data_in`), and the expected vectors. The response verifier reads the ASCII text file corresponding to the expected values of the output signal `data_out`, and compares it to actual output data. Any discrepancy is flagged, and an error message is displayed on the simulator. Figure 23 shows a typical encoder simulation, showing inputs, outputs, and internal control signals. The VHDL code for the encoder testbench is shown in Appendix E.

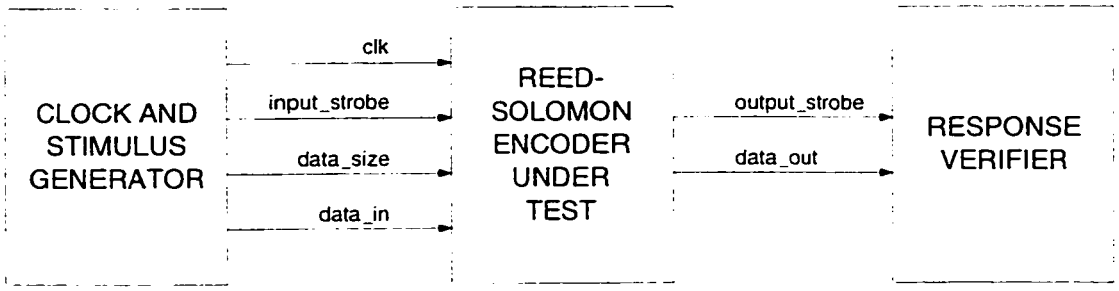

Figure 22. RS Encoder Testbench Structure

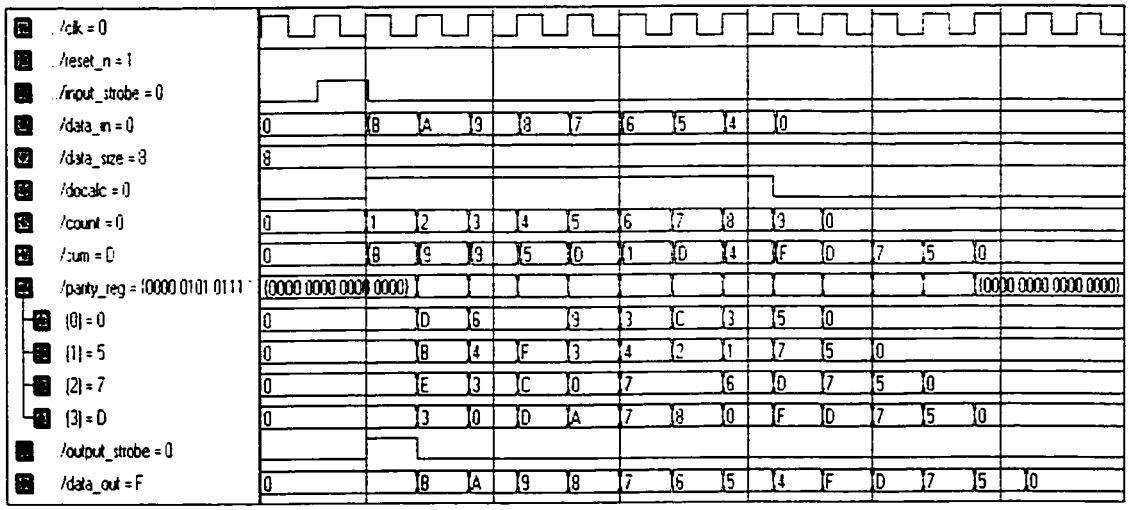

Figure 23. RS Encoder Simulation Waveforms

7 VHDL Design of a General RS Decoder (1 or 2 errors)

7.1 Decoder Overview

As was discussed in Section 2.4, a major simplification is possible if the error-correcting capability of a code is either 1 or 2. The resulting simplification is the removal of the key equation solver, and simpler Chien search block. The syndrome calculation is still required. These 2 blocks are discussed below.

7.2 Syndrome Calculation

The syndrome calculation for an RS decoder capable of correcting 1 or 2 errors is identical to that of the syndrome calculation for 3 or more errors, except that the number of syndromes values is less, since the error-correcting capability is correspondingly less. For a 1 error-correcting code, 2 syndrome values are produced, while 4 syndrome values are produced for a 2 error-correcting code. A block for a generic syndrome value, S(j), is shown in Figure 24, and represents the calculation of the syndrome as per the equations in section 2.2.

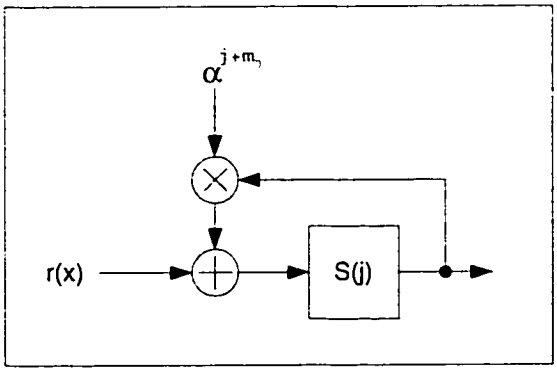

Figure 24. Syndrome Calculation Block Diagram

7.2.1 Syndrome Calculation Timing Diagram

Figure 25 shows the timing diagram for a syndrome calculation block. An input strobe called `rs_data_in_start` occurs at the start of the N symbols of rs_data. When all the N `rs_data` symbols have been received, the syndrome calculation has been completed. An output strobe called `syndrome_calc_done` occurs immediately after the data has entered. One clock cycle later, the value of the 2t syndromes and a status signal called `errors_present`, which is active high, are ready.

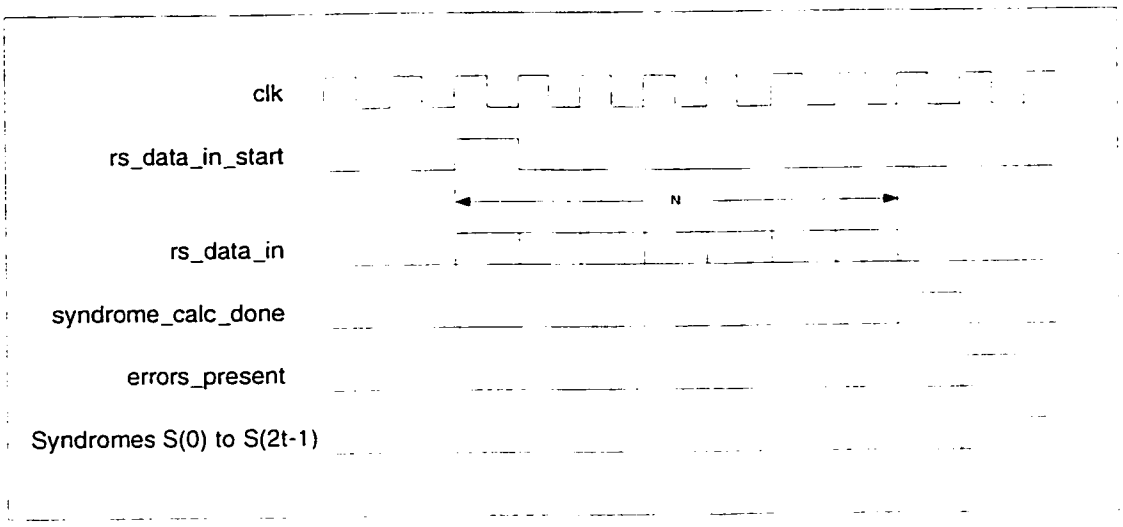

Figure 25. Syndrome Calculation Timing Diagram

7.2.2 Syndrome Calculation Constants, Signals, and Control Logic

A number of constants and signals are needed in the syndrome calculation block. Figure 26 shows the VHDL code for a RS(22,18) code using GF(2^7). This is specific code, but the generic case will be discussed as needed. First, the value of m=7 is defined as the constant `GFPower`. This is then used to define a subtype called `Galois_Field_element` which is used to define all of the data related signals, namely, the syndrome values `IntS`. The name IntS is used to because these are the internal syndrome values. The constant for N, t, and m_0 are next defined. A constant

called `Two_T_minus_1` is defined and will be used in the functional code later. The constant for 0 and 1 (`zero` and `one`) are next defined and are specific to the RS code. In this case, the error correcting capability is t=2. Thus there are 4 constants required in the syndrome calculation, the powers of alpha from 33 to 36, since m_0=33. A counter `IntCount` is required in the control of the syndrome calculation. Its size is based on the largest possible value, proportional to N. All of the other signals listed are needed in the syndrome calculation block, but can written in the manner shown for any particular RS code.

```
constant GFPower : integer := 7;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 2;
constant RS_N : integer := 22;
constant RS_M0 : integer := 33;
constant Two_T_minus_1 : integer := 2*RS_T-1;
constant     zero : Galois_Field_element := "0000000";
constant      one : Galois_Field_element := "0000001";
constant  alpha33 : Galois_Field_element := "0001100";
constant  alpha34 : Galois_Field_element := "0011000";
constant  alpha35 : Galois_Field_element := "0110000";
constant  alpha36 : Galois_Field_element := "1100000";
type IntS_type is array(0 to Two_T_minus_1) of Galois_Field_element;
type IntEP_type is array(0 to Two_T_minus_1) of std_logic;
type state_type is (Idle, RxData, XferData);
signal present_state : state_type;
signal IntCount : std_logic_vector (4 downto 0);
signal internal_errors_present : std_logic;
signal CountEnable : std_logic;
signal CountReset : std_logic;
signal StartXfer : std_logic;
signal XferSyndrome : std_logic;
signal DoCalc : std_logic;
signal IntS : IntS_type;
signal IntEP : IntEP_type;
```

Figure 26. Syndrome Calculation - VHDL Constant and Signal Definition

Figure 27 shows the VHDL code for the control logic of the syndrome calculation block for the example given above. A state machine is used to control the data flow ; the state of the state machine is called `present_state`. The state machine also controls a counter `IntCount` using the control signals `CountEnable` and `CountReset`. The data flow is controlled through the signals `DoCalc` and `XferSyndrome`. Initially, the state machine is in the `Idle` state, and all control signals are in their inactive (low) state, except for the `CountReset`, which is resetting the counter.

When the input strobe `rs_data_in_start` arrives, the state machine switches to the `RxData` state. In this state, the `DoCalc` signal is active (high), enabling the syndrome calculation, and the `CountEnable` is active, incrementing the counter every clock cycle. All other control signals are inactive. When the counter reaches the value 19 (decimal) = N-3, the `StartXfer` signal is set to 1. The next clock cycle, the state machine changes to the `XferData` state. During this state, the output strobe `syndrome_calc_done` is set to active for 1 clock cycle. The state machine then goes back to the `Idle` state immediately thereafter. Note that the VHDL code is written in such a manner as to minimize the changes needed for specific RS codes. Indeed, the only line of code that needs changing is the check for the maximum counter value.

```
InputControlSD_Idle : process (clk)
begin
if (clk'event and clk = '1') then
  if (reset_n = '0') then
    XferSyndrome<='0';  DoCalc<='0';
    CountReset<='1';    CountEnable<='0';
    present_state <= Idle;
  else
    case present_state is
      when Idle =>
        if (rs_data_in_start = '1') then
          DoCalc<='1';  CountReset<='0';
          CountEnable<='1';  present_state <= RxData;
        else
          present_state <= Idle;
        end if;
      when RxData =>
        if (StartXfer = '1') then
          XferSyndrome<='1'; DoCalc<='0';
          CountReset<='1';  CountEnable<='0';
          present_state <= XferData;
        else
          present_state <= RxData;
        end if;
      when XferData =>
        XferSyndrome<='0'; DoCalc<='0';
        CountReset<='1'; CountEnable<='0';
        present_state <= Idle;
      when others =>
        XferSyndrome<='0'; DoCalc<='0';
        CountReset<='1';  CountEnable<='0';
        present_state <= Idle;
    end case;
  end if;
end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (CountReset = '1')) then
      IntCount <= (others=>'0');
    elsif (CountEnable = '1') then
      IntCount <= IntCount + 1;
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      StartXfer <= '0';
    elsif (IntCount="10011") then -- RS(22,18) : max_count = 19
      StartXfer <= '1';
    else
      StartXfer <= '0';
    end if;
  end if;
end process;

syndrome_calc_done <= XferSyndrome;
```

Figure 27. Syndrome Calculation - VHDL Code For Control Logic

7.2.3 Syndrome Registers

Figure 29 shows the VHDL code for the syndrome registers, and the determination of if there is an error present. The intermediate syndrome values are stored in the IntS register array. On reset, they are set to zero. When the input strobe `rs_data_in_start` arrives, these registers are set to the input data. Thereafter, while the control signal `DoCalc` is 1, the syndromes are calculated as per previous discussions. When the control signal `XferSyndrome` is 1, i.e., at the end of the input data, the output syndrome registers and the `errors_present` control bit are set to their correct values. The signal `errors_present` is not used by the Chien search block, but is an output for the testbench. The unused outputs are removed during the synthesis process, so that resources are not used for unwanted signals. Once again, note that only a few lines will differ depending on the RS code.

Figure 28 shows the simulation waveforms for the example cited above.

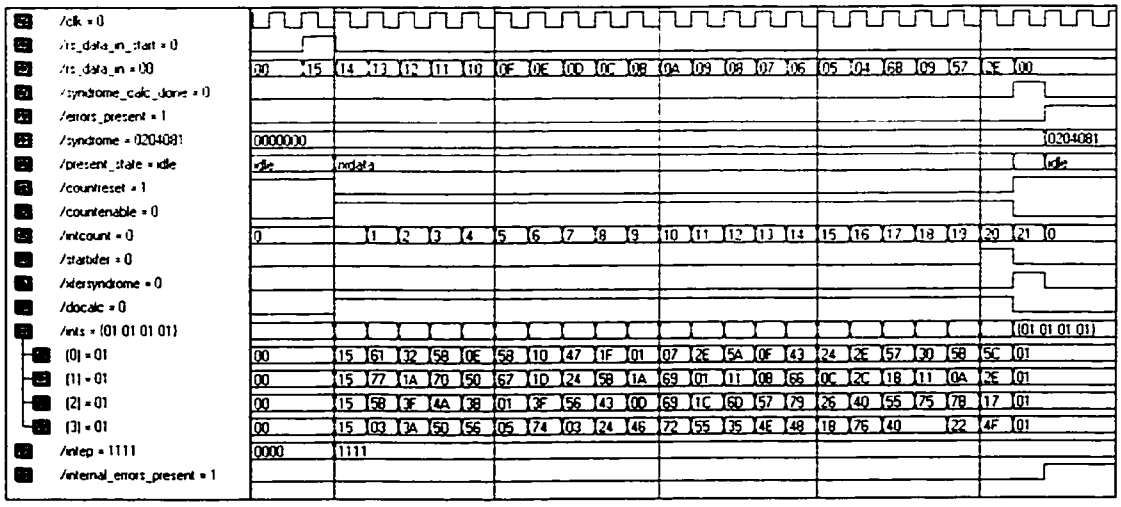

Figure 28. Syndrome Calculation Simulation Waveforms

```
process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= zero;
      end loop;
    elsif (rs_data_in_start = '1') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= rs_data_in;
      end loop;
    elsif (DoCalc='1') then
        IntS(0) <= add(mul(alpha33,IntS(0)),rs_data_in);
        IntS(1) <= add(mul(alpha34,IntS(1)),rs_data_in);
        IntS(2) <= add(mul(alpha35,IntS(2)),rs_data_in);
        IntS(3) <= add(mul(alpha36,IntS(3)),rs_data_in);
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      syndrome <= (others=>'0');
      internal_errors_present <= '0';
    elsif (XferSyndrome='1') then
      syndrome <=
        IntS(3) & IntS(2) & IntS(1) & IntS(0) ;
      internal_errors_present <=
        IntEP(0) or IntEP(1) or IntEP(2) or IntEP(3) ;
    end if;
  end if;
end process;

process (IntS)
begin
  for i in 0 to Two_T_minus_1 loop
    IntEP(i)<='1';
    if (IntS(i)=zero) then IntEP(i)<= '0'; end if;
  end loop;
end process;

errors_present <= internal_errors_present;
```

Figure 29. Syndrome Calculation - VHDL Code for Syndrome Registers

7.3 Chien Search and Error Correction for 1 Error

This section will describe the details of the VHDL implementation of the Chien search and the error correction for an RS decoder capable of correcting 1 error. The relevant theory has been described previously in Section 2.4. The input/output timing diagram of the Chien search block is shown in Figure 30. The internal signals are not shown here. Refer to the simulation waveforms below to see their relationship with the input and output signals.

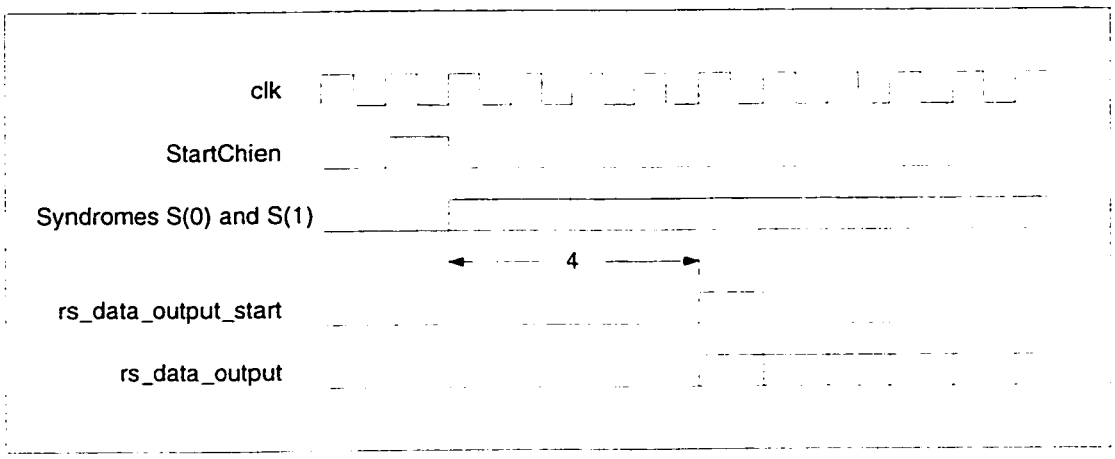

Figure 30. Chien Search Timing Diagram – 1 error

7.3.1 Constants and Signal Definition

The value of m, ($GF(2^m)$), is defined as an integer `GFPower`. The subtype `Galois_Field_element` is then defined from `GFPower`. The constant for the error correcting capability, `RS_T`, the encoded message size in symbols, `RS_N`, and the log of the initial root of the code generator polynomial, m_0, are defined as integers. These are then used to define the remaining constants. The delay through the Chien search pipeline is defined. It has a value of 3, since the delay starts from 0 and ends at 3, giving a 4 clock delay, as shown in Figure 30. These together with RS_N define the size of the delay block shift_register_delay, `SR_max`. A counter `Count` is used to aid the generation of control signals. The maximum count value for the control signals are defined as

constants. The signals Count, CountEnable, InitChien, InitChien_dl, DoChien, and OutputEnable are control signals, while the remaining signals are related to the data path.

```
constant GFPower : integer := 6;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T  : integer := 1;
constant RS_N  : integer := 22;
constant RS_M0 : integer := 55;
constant ChienSearch_pipeline_delay : integer := 3;
constant SR_max : integer := RS_N + ChienSearch_pipeline_delay;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant OutputEnableStopCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant DoChienCountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-2, GFPower+1) ;
constant CountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant      zero : Galois_Field_element := "000000";
constant       one : Galois_Field_element := "000001";
constant Temp1_initial_value : Galois_Field_element := "111010"; -- alpha(-(RS_N-1))
constant  x0a_mul : Galois_Field_element := "101110"; -- alpha(RS_M0) =  a55
constant   alpha1 : Galois_Field_element := "000010";
constant x0a_initial_value : Galois_Field_element := "111010"; -- alpha(-(RS_N-1)*RS_M0)
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
signal shift_register_delay : shift_register_delay_type;
signal RS_data_delayed : Galois_Field_element;
signal CORR_FACTOR : Galois_Field_element;
signal Count : std_logic_vector(GFPower downto 0);
signal CountEnable : std_logic;
signal InitChien : std_logic;
signal InitChien_dl : std_logic;
signal DoChien : std_logic;
signal OutputEnable : std_logic;
signal inv_s0 : Galois_Field_element;
signal x0a : Galois_Field_element;
signal Temp1 : Galois_Field_element;
type syndrome_type is array(0 to 2*RS_T-1) of Galois_Field_element;
signal S : syndrome_type ;
```

Figure 31. Chien Search For 1 Error – VHDL Constants and Signals

7.3.2 Control Signals

Figure 33 shows the code for the control signals. A counter Count is set to 0 when the input strobe StartChien is active. Thereafter, it increments every clock cycle. When it reaches the constant CountMax, the enable to the counter is removed, and the counter stops counting. The signal StartChien is just a delayed version (1 clock delay) of the input strobe, and is only used to control the counter. The signal InitChien is just a delayed version (1 clock delay) of StartChien, and is used to initialize the data path. The signal OutputEnable is used to enable the output data while the data stream is

leaving, otherwise the output data is set to 0. The signal DoChien is used to control the path as well. As can be seen, these 2 control signals are controlled by the input strobe and the counter value. Note that the code is written in such a manner so that no change is required for different RS codes. The parameterization is accomplished through the use of constants.

The code for the output strobe is shown in Figure 32. The output strobe is controlled by the counter, and is set to 1 for only one clock period. It is timed to coincide with the first output data symbol.

```
RS_Data_out_Start_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;
```

Figure 32. Chien Search For 1 Error - Output Strobe

```
System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;

System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;

InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
      InitChien_dl <= '0';
    else
      InitChien <= StartChien;
      InitChien_dl <= InitChien;
    end if;
  end if;
end process;

OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien_dl = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;
```

Figure 33. Chien Search For 1 Error - Internal Control Signals

7.3.3 Delay Block

Figure 34 shows the VHDL code for the data delay. The required delay is equal to the delay through the Syndrome_Calculation block (N) and the Chien search pipeline delay. This length determines the constant SR_max. The delay is implemented by an array from 0 to SR_max of back-to-back registers. Input data is sent to the 0^{th} array element, and the delayed data is taken from the SR_max^{th} array element.. Note that the code is written in such a manner so that no change is required for different RS codes. The parameterization is accomplished through the use of constants.

```
Delay_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to SR_max loop
        shift_register_delay(i) <= zero;
      end loop;
    else
      for i in SR_max downto 1 loop
        shift_register_delay(i) <= shift_register_delay(i-1);
      end loop;
      shift_register_delay(0) <= RS_Data_In;
    end if;
  end if;
end process;
RS_data_delayed <= shift_register_delay(SR_max);
```

Figure 34. Chien Search For 1 Error - Data Delay

7.3.4 Chien Search, Error Calculation and Error Correction

A block diagram of the data flow of the Chien search block is shown in Figure 35. The corresponding VHDL code is shown in Figure 36. When the `StartChien` arrives, the Temp1 register is loaded with the initial value $\alpha^{-(N-1)}$, the inverse of the present location. It is then multiplied by the syndrome S(1) then next clock cycle (`InitChien=1`), and by the inverse of syndrome S(0) the following clock cycle, when `InitChien_d1` is 1. Every clock cycle thereafter, the `Temp1` register is multiplied by the constant α. In this manner, every location is stepped through. When the location X=S(1)/S(0) is processed, the value of `Temp1` will be the symbol corresponding to 1. In this manner, the location of the error is found.

The register x0a is part of the error correction calculation. When the InitChien_d1 signal is 1, it is set to its initial value of $\alpha^{-(N-1)\cdot m_0}$. Every clock cycle thereafter, it value is multiplied by α^{m_0}. The register x0a is multiplied by the syndrome S(0), to yield a streaming set of correction factors of the form $Y_i = S(0) \cdot X_i^{-m_0}$ as specified in section 2.4. The correction factor is applied to the delayed data only when the Chien search has found the error, i.e., when `Temp1` is 1. This results in corrected data being sent out of the decoder.

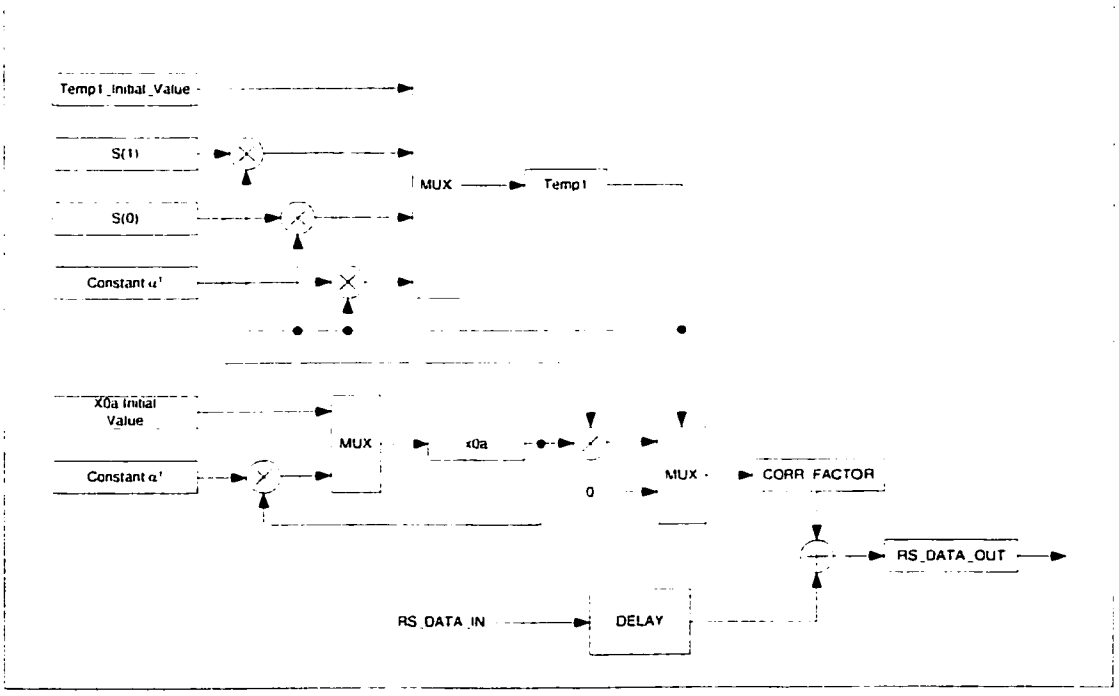

Figure 35. Chien Search For 1 Error - Data Flow Block Diagram

```
S(1) <= syndromes(11 downto 6);
S(0) <= syndromes(5 downto 0);

inv_s0_process : process(S)
begin
  inv_s0 <= inv(S(0));
end process;

Calc_Temp_Registers : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1 <= zero;
    elsif (StartChien = '1') then
      Temp1 <= Temp1_initial_value;
    elsif (InitChien = '1') then
      Temp1 <= mul(S(1),Temp1);
    elsif (InitChien_d1 = '1') then
      Temp1 <= mul(inv_s0,Temp1);
    elsif (DoChien = '1') then
      Temp1 <= mul(alpha1,Temp1);
    end if;
  end if;
end process;

x0a_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x0a <= zero;
    elsif (InitChien_d1 = '1') then
      x0a <= x0a_initial_value;
    else
      x0a <= mul(x0a,x0a_mul);
    end if;
  end if;
end process;

Calc_Correction_Factor : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= zero;
    elsif (rs_enable = '1') and (Temp1=one) then
      CORR_FACTOR   <= mul(S(0),x0a);
    else
      CORR_FACTOR <= zero;
    end if;
  end if;
end process;

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif (OutputEnable = '1') then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    else
      RS_Data_out <= zero;
    end if;
  end if;
end process;
```

Figure 36. Chien Search For 1 Error – Chien Search, and Correction

7.4 Chien Search and Error Correction for 2 Errors

This section will describe the details of the VHDL implementation of the Chien search and the error correction for an RS decoder capable of correcting 2 errors. The input/output timing diagram of the Chien search block is shown in Figure 37. The internal signals are not shown here. Refer to the simulation waveforms below to see their relationship with the input and output signals.

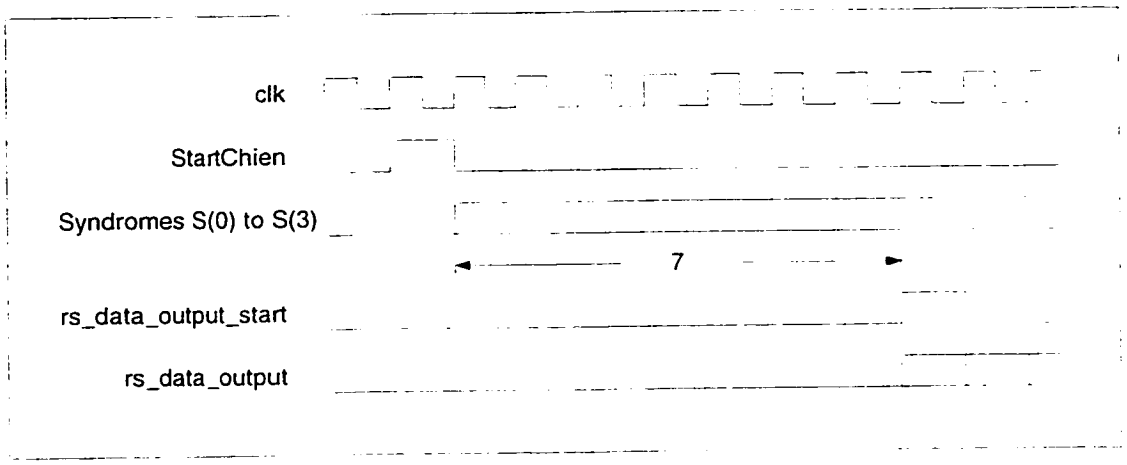

Figure 37. Chien Search Timing Diagram – 2 errors

7.4.1 Constants and Signal Definition

The code for the VHDL constant s is shown in Figure 38, while the signal definitions are shown in Figure 39. As before, the constant `GFPower` is defined, along with constants for m_0 (`RS_M0`), N (`RS_N`) and t (`RS_T`). The `ChienSearch_pipeline_delay` is defined. These basic constants are then used to define constants for control signal (`CountMax` to `RS_Data_out_StartCount`). The remaining constants, `Temp1_mul_factor` to `zero` are related to the data flow logic. Here is where the characteristics specific to a RS code required in the Chien search are defined.

```
constant GFPower : integer := 7;
constant RS_M0   : integer := 33;
constant RS_N    : integer := 22;
constant RS_T    : integer := 2;
constant ChienSearch_pipeline_delay : integer := 7;
constant SR_max : integer   := RS_N + ChienSearch_pipeline_delay;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
type syndrome_type is array(0 to 2*RS_T-1) of Galois_Field_element;
constant CountMax                : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant DoChienCountMax         : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-3, GFPower+1) ;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant OutputEnableStopCount  : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant Temp1_mul_factor   : Galois_Field_element := "0111000"; -- alpha(-(RS_N-1)*1)
constant Temp2_mul_factor   : Galois_Field_element := "0011010"; -- alpha(-(RS_N-1)*2)
constant alpha1             : Galois_Field_element := "0000010";
constant alpha2             : Galois_Field_element := "0000100";
constant one                : Galois_Field_element := "0000001";
constant x0a_initial_value  : Galois_Field_element := "0101101"; --alpha(-(RS_N-1)*RS_M0)
constant x0a_mul            : Galois_Field_element := "0001100"; -- alpha(RS_M0)
constant x1_initial_value   : Galois_Field_element := "1101101"; -- alpha(RS_N-1)
constant x1_mul             : Galois_Field_element := "1000100"; -- alpha(-1)
constant x1m_initial_value  : Galois_Field_element := "1101110"; -- alpha((RS_N-1)*RS_M0)
constant x1m_mul            : Galois_Field_element := "0011110"; -- alpha(-RS_M0)
constant zero               : Galois_Field_element := "0000000";
```

Figure 38. Chien Search For 2 Error – VHDL Constants

The signals shown in Figure 39 can be separated into two categories. The control logic related signals are grouped from `Count` to `there_is_one_error`. The data flow related signals are those from `RS_data_delayed` to `x1m`.

```
signal Count                : std_logic_vector(GFPower downto 0);
signal CountEnable          : std_logic;
signal DoChien              : std_logic;
signal InitChien            : std_logic;
signal InitChien_d1         : std_logic;
signal InitChien_d2         : std_logic;
signal InitChien_d3         : std_logic;
signal InitChien_d4         : std_logic;
signal InitChien_d5         : std_logic;
signal OutputEnable         : std_logic;
signal there_are_two_errors : std_logic;
signal there_is_one_error   : std_logic;
signal RS_data_delayed      : Galois_Field_element;
signal CORR_FACTOR          : Galois_Field_element;
signal CORR_FACTOR_1_error  : Galois_Field_element;
signal CORR_FACTOR_2_errors : Galois_Field_element;
signal ChienSum             : Galois_Field_element;
signal D1                   : Galois_Field_element;
signal D2                   : Galois_Field_element;
signal Temp1                : Galois_Field_element;
signal Temp1_num_1_error    : Galois_Field_element;
signal Temp1_num_2_errors   : Galois_Field_element;
signal Temp2                : Galois_Field_element;
signal Temp2_num_2_errors   : Galois_Field_element;
signal denom                : Galois_Field_element;
signal inv_D2               : Galois_Field_element;
signal inv_denom            : Galois_Field_element;
signal inv_s0               : Galois_Field_element;
signal num                  : Galois_Field_element;
signal num_d1               : Galois_Field_element;
signal shift_register_delay : shift_register_delay_type;
signal sigma1               : Galois_Field_element;
signal syndrome             : syndrome_type;
signal x0a                  : Galois_Field_element;
signal x1                   : Galois_Field_element;
signal x1m                  : Galois_Field_element;
```

Figure 39. Chien Search For 2 Error2 – VHDL Signals

7.4.2 Control Signals

Figure 40 shows the VHDL code for some internal control signals. A counter `Count` is reset when the signal `StartChien` is 1, i.e., at the beginning. The enable for the counter is set to 1 (active) also when `StartChien` is 1. The counter increments every clock cycle as long as the enable is active. The enable is deactivated when the counter has reached the `CountMax` value defined in the constants section. Figure 41 shows the VHDL code for the determination of a single, double or no error case. The result is as per the discussions in Section 2.5. Figure 42 shows the code for the control signal `OutputEnable` and the output strobe RS_Data_out_Start. Both of these signals are controlled by the counter and constants defined previously. Note the code for all of these control signals is completely general through the use of constants.

```
System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;

System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;

InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
      InitChien_d1 <= '0';
      InitChien_d2 <= '0';
      InitChien_d3 <= '0';
      InitChien_d4 <= '0';
      InitChien_d5 <= '0';
    else
      InitChien <= StartChien;
      InitChien_d1 <= InitChien;
      InitChien_d2 <= InitChien_d1;
      InitChien_d3 <= InitChien_d2;
      InitChien_d4 <= InitChien_d3;
      InitChien_d5 <= InitChien_d4;
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien_d4 = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;
```

Figure 40. Chien Search Internal Control Signals

```
Error_Count_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      there_is_one_error   <= '0';
      there_are_two_errors <= '0';
    elsif ((InitChien = '1') and (IsNotZero(D1)='1')) then
      there_is_one_error   <= '1';
      there_are_two_errors <= '0';
    elsif ((InitChien_d1 = '1') and (IsNotZero(D2)='1')) then
      there_is_one_error   <= '0';
      there_are_two_errors <= '1';
    end if;
  end if;
end process;
```

Figure 41. Single/Double Error Determination

```
OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

RS_Data_out_Start_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;
```

Figure 42. Output Control Signals

7.4.3 Delay Block

The VHDL code for the delay block for a 2-error correcting RS code is identical to that of the 1-error correcting code, except that the Chien search pipeline delay is larger (7). This makes the constant SR_max correspondingly larger. The VHDL code is shown in Figure 34.

7.4.4 Chien Search, Error Calculation and Error Correction

Figure 43 shows the code for the calculation of the determinants D_1 and D_2, as well as the inverse of D_2, as defined in section 2.5. Only the code for the remapping of the syndromes bus and the determinant D_1 require modification for other RS codes. However, this change is relatively trivial. The calculation of these values is predicated by their position in the overall pipeline, using the control signals defined above.

```
syndrome(3) <= syndromes( 27 downto  21);
syndrome(2) <= syndromes( 20 downto  14);
syndrome(1) <= syndromes( 13 downto   7);
syndrome(0) <= syndromes(  6 downto   0);

D1 <= syndromes(6 downto  0);

Calculate_D2 : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      D2 <= zero;
    elsif (InitChien = '1') then
      D2 <= add(mul(syndrome(0),syndrome(2)),mul(syndrome(1),syndrome(1)));
    end if;
  end if;
end process;

Calculate_inv_D2 : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      inv_D2 <= zero;
    elsif (InitChien_d1 = '1') then
      inv_D2 <= inv(D2);
    end if;
  end if;
end process;
```

Figure 43. Determinant Calculation

Figure 44 shows the VHDL code for the calculation of the error corrections factor, according to the equations defined in Section 2.5. The calculation for both the single error and the double error case are shown. The x0a_process calculate the single error correction factor $Y_1 = S(0) \cdot X_1^{-m_0}$. This is done by first initializing the x0a register to $S(0) \cdot \alpha^{-(N-1) \cdot m_0}$. This corresponds to a single error correction factor for the first position of the Chien search. Every clock cycle thereafter, the register is multiplied by α^{m_0}, thus generating a sequence of single error correction factors for each location.

The x1_process, x1m_process and the CORR_FACTOR_2_errors_process show the code for calculating the correction factors for double errors, as described in 2.5, namely

$$Y_1 = \frac{S(0)\cdot(X_1+\sigma_1)+S(1)}{X_1^{m_0}\cdot\sigma_1} \tag{6.1}$$

The x1_process generates successive values of X_1. This is done by initializing the x1 register to $\alpha^{(N-1)}$, and then multiplying the register by every clock cycle thereafter. This produces the sequence $\alpha^{(N-1)},\alpha^{(N-2)},\alpha^{(N-3)},\ldots,\alpha^2,\alpha,1$. In a similar manner, the x1m_process generates successive values of $X_1^{m_0}$. The register x1m is initialized to $\alpha^{(N-1)\cdot m_0}$, and is then multiplied by α^{-m_0} every clock cycle.

The CORR_FACTOR_2_errors_process completes the calculation of the double error correction factor. In parallel, it calculates the numerator $S(0)\cdot(X_1+\sigma_1)+S(1)$, and the denominator, $X_1^{m_0}\cdot\sigma_1$. The denominator is then inverted, and multiplied with the numerator, to yield the value Y_1, as defined above.

Figure 45 shows the code for choosing which error correction factor to use, single or double. This is done by examining two control signals `there_is_one_error` and `there_are_two_errors`, whose name describe their function. The generation of these two control signals is based on the values of the determinants D_1 and D_2, and is shown in Figure 41. The correction factor is then applied when the Chien sum is 0, that is, when an error location has been found. As can be seen, the VHDL code written for these processes is completely generic, and does not require modification for other RS codes.

```
x0a_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x0a <= zero;
    elsif (InitChien_d5 = '1') then
      x0a <= mul(x0a_initial_value,syndrome(0));
    else
      x0a <= mul(x0a,x0a_mul);
    end if;
  end if;
end process;
CORR_FACTOR_1_error <= x0a;

x1_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x1 <= zero;
    elsif (InitChien_d2 = '1') then
      x1 <= x1_initial_value;
    else
      x1 <= mul(x1,x1_mul);
    end if;
  end if;
end process;

x1m_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x1m <= zero;
    elsif (InitChien_d2 = '1') then
      x1m <= x1m_initial_value;
    else
      x1m <= mul(x1m,x1m_mul);
    end if;
  end if;
end process;

CORR_FACTOR_2_errors_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      num        <= zero;
      num_d1     <= zero;
      denom      <= zero;
      inv_denom <= zero;
      CORR_FACTOR_2_errors <= zero;
    else
      num        <= add(mul(add(x1,sigma1),syndrome(0)),syndrome(1));
      num_d1     <= num;
      denom      <= mul(x1m,sigma1);
      inv_denom <= inv(denom);
      CORR_FACTOR_2_errors <= mul(num_d1,inv_denom);
    end if;
  end if;
end process;
```

Figure 44. Correction Factor Calculation

```
Calc_Correction_Factor : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= zero;
    elsif (rs_enable = '1') and (ChienSum=zero) then
      if (there_is_one_error='1') then
        CORR_FACTOR   <= CORR_FACTOR_1_error;
      elsif (there_are_two_errors='1') then
        CORR_FACTOR   <= CORR_FACTOR_2_errors;
      end if;
    else
      CORR_FACTOR <= zero;
    end if;
  end if;
end process;

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif (OutputEnable = '1') then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    else
      RS_Data_out <= zero;
    end if;
  end if;
end process;
```

Figure 45. Error Correction

Figure 46 and Figure 47 show the VHDL code that implements the actual Chien search. In Figure 46, the calculation of the coefficients of the error-locator polynomial is shown. The Chien search uses two registers `Temp1` and `Temp2` to perform the root test. For the single error case, `Temp2` is set to zero, while `Temp1` is set to $\frac{S(1)}{S(0)} \bullet \alpha^{-(N-1)}$, since the error-locator polynomial is linear. For the double error case, Temp2 is set to $\frac{S(2)^2 + S(1) \cdot S(3)}{S(0) \bullet S(2) + S(1)^2} \bullet \alpha^{-(N-1)-2}$, while Temp1 is set to $\frac{S(0) \cdot S(3) + S(1) \cdot S(2)}{S(0) \bullet S(2) + S(1)^2} \bullet \alpha^{-(N-1)}$, since the error-locator polynomial is quadratic. These equations are those derived in Section 2.5. The Temp1 register is then multiplied by α, and Temp2 is multiplied by α^2 every clock cycle. Finally, the Chien is formed by adding the registers `Temp1`, `Temp2` and the constant one. An error location is found when the Chien search is 0.

```
inv_s0_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      inv_s0 <= zero;
    elsif (InitChien = '1') then
      inv_s0 <= inv(syndrome(0));
    end if;
  end if;
end process;

Temp1_num_1_error_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1_num_1_error <= zero;
    elsif (InitChien_d1 = '1') then
      Temp1_num_1_error <= mul(inv_s0,syndrome(1));
    end if;
  end if;
end process;

Temp1_num_2_errors_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1_num_2_errors <= zero;
    elsif (InitChien = '1') then
      Temp1_num_2_errors <=
add(mul(syndrome(0),syndrome(3)),mul(syndrome(1),syndrome(2)));
    end if;
  end if;
end process;

Temp2_num_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp2_num_2_errors <= zero;
    elsif (InitChien = '1') then
      Temp2_num_2_errors <=
add(mul(syndrome(1),syndrome(3)),mul(syndrome(2),syndrome(2)));
    elsif (InitChien_d2 = '1') then
      Temp2_num_2_errors <= mul(Temp2_num_2_errors,inv_d2);
    end if;
  end if;
end process;

Temp2_2_error_value_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      sigma1 <= zero;
    elsif (InitChien_d2 = '1') then
      sigma1 <= mul(Temp1_num_2_errors,inv_D2);
    end if;
  end if;
end process;
```

Figure 46. Chien Search Initialization Values Calculation

```
Calc_Temp_Registers_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') or (StartChien='1') then
      Temp1 <= zero;
      Temp2 <= zero;
    elsif (InitChien_d4 = '1') then
      if (IsNotZero(D2) = '1') then
        Temp1 <= mul(Temp1_mul_factor,sigma1);
        Temp2 <= mul(Temp2_mul_factor,Temp2_num_2_errors);
      elsif (IsNotZero(D1) = '1') then
        Temp1 <= mul(Temp1_mul_factor,Temp1_num_1_error);
        Temp2 <= zero;
      else
        Temp1 <= zero;
        Temp2 <= zero;
      end if;
    elsif (DoChien = '1') then
        Temp1 <= mul(alpha1,Temp1);
        Temp2 <= mul(alpha2,Temp2);
      end if;
  end if;
end process;

ChienSum_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      ChienSum <= zero;
    elsif (DoChien = '1') then
      ChienSum <= add(one,add(Temp1,Temp2));
    end if;
  end if;
end process;
```

Figure 47. Chien Search Process

Figure 48 shows the simulation waveforms of a Chien search for 1 error, while Figure 49 shows the waveforms for 2 errors.

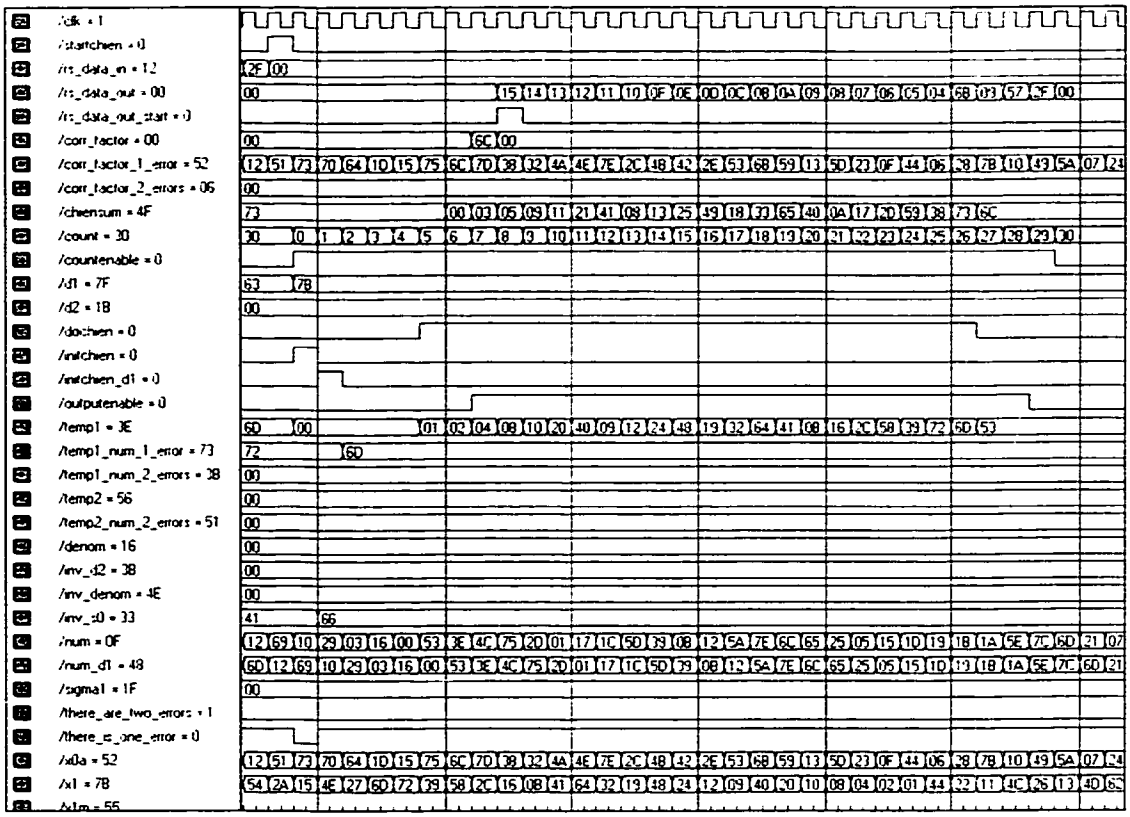

Figure 48. Simulation Waveforms – 1 error

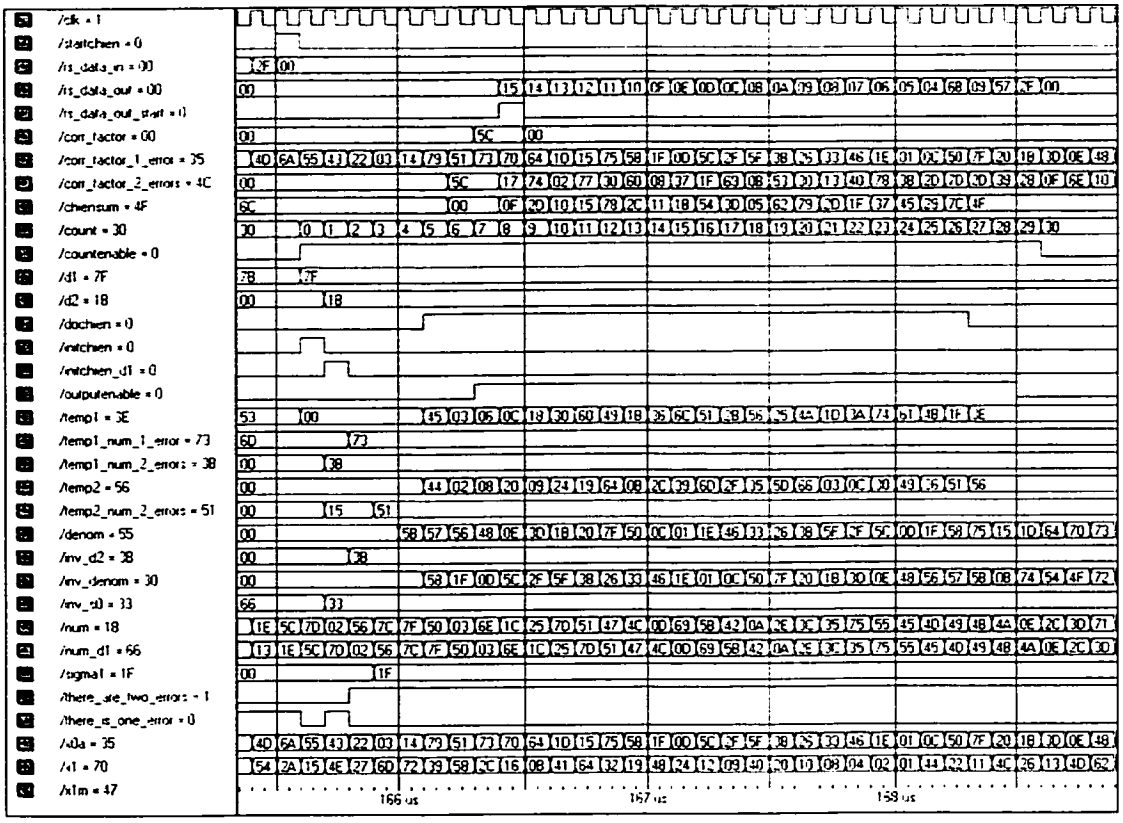

Figure 49. Simulation Waveforms – 2 errors

8 VHDL Design of a General RS Decoder (3 or more errors)

8.1 Decoder Overview

A generic RS decoder for 3 or more errors can be broken down in three major sections, namely the syndrome calculation block, the key equations solver, for which the extended inversionless Massey-Berlekamp is chosen, and finally, the Chien search. The exact algorithmic description has been given in Sections 2.2, 2.3, and 2.6. The implementation of the syndrome calculation block in VHDL has been discussed in detail in Section 7.2 and will not be repeated here.

8.2 Key Equation Solver

The VHDL implementation of the extended inversionless Massey-Berlekamp will be described in this section. Figure 50 shows the timing diagram. The block algorithm starts when the input strobe XferSyndrome arrives. The outputs of this block are the coefficients of the error-locator polynomial (`lambda_poly`) and the error-evaluator polynomial (`omega_poly`). The latency is (2t+1) where "t" is the error correcting capability of the RS code.

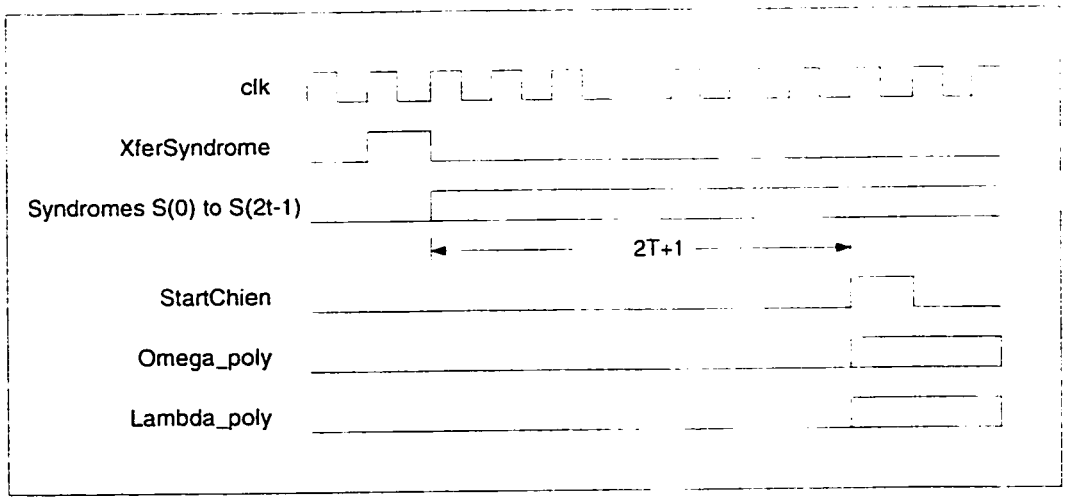

Figure 50. Extended Inversionless Massey-Berlekamp Timing Diagram

Figure 51 shows the VHDL code for the constants, signal definition, and 3 functions, convolution_term_mul, is_not_0 and is_0. The value of m, ($GF(2^m)$), defined as an integer GFPower. The subtype Galois_Field_element is then defined from GFPower. The constant for the error correcting capability, RS_T, the encoded message size in symbols, RS_N, and the log of the initial root of the code generator polynomial, m_0, are defined as integers. These are then used to define the remaining constants. The control logic related signals are those listed from current_state to StoreNewPolys. The data flow related signals are those listed from N to Convolution_Term_Multipler.

The function convolution_term_mul is used to help in the calculation of the signal delta, as will be described below. The function is_not_0 is used to determine if a Galois field element is not 0. This is done by OR'ing all of the bits of the element. The function is_0 is simply the negation of is_not_0.

Figure 52 shows the state machine that generates the control signals to perform the iteration of the extended inversionless Massey-Berlekamp algorithm. It generates four control logic signals, Initialize, StoreNewPolys, StartChien, and CountEnable. This state machine implements the looping from 0 to N-1 for the extended inversionless Massey-Berlekamp algorithm, through the use of the counter N. The mapping between the internal array Mu into a single bus called lambda_poly for the error-locator polynomial coefficients is shown, along with the mapping of the internal array Zomega into a single bus called omega_poly for the error-evaluator polynomial coefficients.

```
constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 4;
constant RS_N : integer := 22;
constant RS_M0 : integer := 1;
constant    zero : Galois_Field_element := "00000000";
constant     one : Galois_Field_element := "00000001";
constant size_of_N : integer := 4;
type SRegType is array(0 to 3*RS_T-1) of Galois_Field_element;
type Poly1Type is array(0 to RS_T) of Galois_Field_element;
type Poly2Type is array(0 to 2*RS_T) of Galois_Field_element;
type Poly3Type is array(1 to RS_T) of Galois_Field_element;
type state_type is (Idle,ChienSearchStart,Init,Synchronize,Update_Polys);

signal current_state : state_type;
signal KeepOldL : std_logic;
signal L : std_logic_vector((size_of_N-1) downto 0);
signal TwoL : std_logic_vector((size_of_N-1) downto 0);
signal Initialize : std_logic;
signal CountEnable : std_logic;
signal StoreNewPolys : std_logic;

signal N : std_logic_vector((size_of_N-1) downto 0);
signal SReg: SRegType;
signal Delta : Galois_Field_element;
signal Convolution_Term : Poly1Type;
signal Post_Convolution_Term : Poly1Type;
signal Gamma : Galois_Field_element;
signal Lambda : Poly1Type;
signal ZOmega  : Poly1Type;
signal Mu, GammaMu : Poly1Type;
signal DeltaLambda : Poly3Type;
signal B    : Poly1Type;
signal Convolution_Term_Multipler : std_logic_vector(RS_T downto 0);

function convolution_term_mul (b : in Galois_Field_element; c : in std_logic) return
Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0):=b(0) and c;
  d(1):=b(1) and c;
  d(2):=b(2) and c;
  d(3):=b(3) and c;
  d(4):=b(4) and c;
  d(5):=b(5) and c;
  d(6):=b(6) and c;
  d(7):=b(7) and c;
  return d;
end convolution_term_mul;

function is_not_0 (b : in Galois_Field_element) return std_logic is
  variable d : std_logic;
begin
  d := b(0) or b(1) or b(2) or b(3) or b(4) or b(5) or b(6) or b(7) ;
  return d;
end is_not_0;

function is_0 (b : in Galois_Field_element) return std_logic is
  variable d : std_logic;
begin
  d := not is_not_0(b);
  return d;
end is_0;
```

Figure 51. Constant, Signal Definitions, and Funtions

```
process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Initialize <= '0';  StoreNewPolys <= '0';
      StartChien <= '0';  CountEnable <= '0';
      current_state <= Idle;
    else
      case current_state is
        when Idle =>
          if (XferSyndrome = '1') then
            Initialize <= '1'; StoreNewPolys <= '0';
            StartChien <= '0'; CountEnable <= '0';
            current_state <= Init;
          else
            current_state <= Idle;
          end if;
        when ChienSearchStart =>
          Initialize <= '0';   StoreNewPolys <= '0';
          StartChien <= '0';   CountEnable <= '0';
          current_state <= Idle;
        when Init =>
          if (ErrorsPresent = '0') then
            Initialize <= '0';  StoreNewPolys <= '0';
            StartChien <= '0';  CountEnable <= '1';
            current_state <= Synchronize;
          else
            Initialize <= '0';  StoreNewPolys <= '1';
            StartChien <= '0';  CountEnable <= '1';
            current_state <= Update_Polys;
          end if;
        when Synchronize =>
          if (N = "0111") then
            Initialize <= '0';  StoreNewPolys <= '0';
            StartChien <= '1';  CountEnable <= '0';
            current_state <= ChienSearchStart;
          else
            current_state <= Synchronize;
          end if;
        when Update_Polys =>
          if (N = "0111") then
            Initialize <= '0';  StoreNewPolys <= '0';
            StartChien <= '1';  CountEnable <= '0';
            current_state <= ChienSearchStart;
          else
            current_state <= Update_Polys;
          end if;
        when others =>
          Initialize <= '0';  StoreNewPolys <= '0';
          StartChien <= '0';  CountEnable <= '0';
          current_state <= Idle;
      end case;
    end if;
  end if;
end process;

omega_poly  <=  ZOmega(4) & ZOmega(3) & ZOmega(2) & ZOmega(1) ;
lambda_poly <=  Mu(4) & Mu(3) & Mu(2) & Mu(1) & Mu(0) ;
```

Figure 52. Control State Machine

Figure 53 shows the VHDL code for the calculation of the signal delta. This corresponds to equation 2.31, and is accomplished in four processes. The RS code chosen in this example is a 4-error correcting code; hence, the signal L, which controls the length of summation, can range from 0 to 4. First, the SReg array, which is 3t elements long, is loaded with 0 for the first t elements, and with the 2t syndrome elements for the remainder. All 5 convolution terms are formed from the Mu and the SReg arrays. These terms are then multiplied by the 5 convolutional term multipliers, which depend on the value of signal L, as shown in then third process called Convolution_Term_Multiplier_process. This results in the Post_Convolution_Term array, which is 5 elements long, some of which are 0, and some of which correspond to the convolution terms. These 5 terms are then added to yield the delta signal. The SReg array is shifted over every iteration of the algorithm, thus forming the convolution from a different starting point.

Figure 54 shows the VHDL code for the variables L, N and for the Gamma array. The process L_process implements the function of equations 2.34 and 2.37. The process TwoL_process generates the value of 2L that is needed in the L_process: it is simply a multiply by 2 function. The Gamma_process implements Equations 2.35 and 2.38. The signal KeepOldL is used by both the Gamma_process and the L_process. It is set to '1' if $\delta^{(k+1)}=0$ or $2l^{(k)}>k$, otherwise it is set to '0'. The N_Process implements the counter from 0 to (2t-1)=7 via the CountEnable control signal from the state machine described earlier.

Figure 55 shows the code for the Lambda_Process and the B_process. The Lambda_process implements the equations 2.33 and 2.36, while the B_process implements equations 2.41 and 2.42, where there has been a renaming f the variable. Figure 56 shows the VHDL code for the Zomega_process which implements equation 2.40. The GammuMu_process, DeltaLambda_process and the Mu_process implement equation 2.32. This corresponds to the error-locator polynomial. When there is no error, as indicated by the input signal ErrorsPresent,

the all of the coefficients of the error-locator polynomial are set to 0 except the first coefficient , which is set to 1. This will prevent the Chien search from finding a root, and thus the data is never changed. Figure 57 shows the simulation waveforms for the extended inversionless Massey-Berlekamp algorithm.

```
SReg_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to (3*RS_T-1) loop
        SReg(i) <= zero;
      end loop;
    elsif (Initialize = '1') then
      for i in 0 to (RS_T-1) loop
        SReg(i) <= zero;
      end loop;
      for i in 0 to (2*RS_T-1) loop
        SReg(i+RS_T) <= syndrome_poly((((i+1)*GFPower)-1) downto (i*GFPower));
      end loop;
    elsif (StoreNewPolys = '1') then
      for i in 0 to (3*RS_T-2) loop
        SReg(i) <= SReg(i+1);
      end loop;
        SReg((3*RS_T-1)) <= zero;
    end if;
  end if;
end process SReg_Process;

Convolution_Term_Process : process (SReg, Mu)
begin
  for i in 0 to RS_T loop
    Convolution_Term(i) <= mul(SReg(RS_T-i),Mu(i));
  end loop;
end process Convolution_Term_Process;

Convolution_Term_Multipler_Process : process (L)
begin
  case L is
    when "0000" => Convolution_Term_Multipler <= "00001";
    when "0001" => Convolution_Term_Multipler <= "00011";
    when "0010" => Convolution_Term_Multipler <= "00111";
    when "0011" => Convolution_Term_Multipler <= "01111";
    when "0100" => Convolution_Term_Multipler <= "11111";
    when others  => Convolution_Term_Multipler <= "11111";
  end case;
end process Convolution_Term_Multipler_Process;

Post_Convolution_Term_Process : process (Convolution_Term_Multipler, Convolution_Term)
begin
  for i in 0 to RS_T loop
    Post_Convolution_Term(i) <=
convolution_term_mul(Convolution_Term(i),Convolution_Term_Multipler(i));
  end loop;
end process Post_Convolution_Term_Process;

Delta_Process : process (Post_Convolution_Term)
begin
   delta <= add(Post_Convolution_Term(0),
            add(Post_Convolution_Term(1),
            add(Post_Convolution_Term(2),
            add(Post_Convolution_Term(3), Post_Convolution_Term(4)))));
end process Delta_Process;
```

Figure 53. Delta Process

```
L_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      L <= (others=>'0');
    elsif (Initialize = '1') then
      L <= (others=>'0');
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='0') then
        L <= N + 1 - L;
      end if;
    end if;
  end if;
end process L_Process;

TwoL_Process : process(L)
begin
  TwoL <= L((size_of_N-2) downto 0)&'0';
end process TwoL_Process;

KeepOldL_Process : process(TwoL, N, Delta)
begin
  if ((TwoL>N) or (is_0(Delta)='1')) then
    KeepOldL <= '1';
  else
    KeepOldL <= '0';
  end if;
end process KeepOldL_Process;

Gamma_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      Gamma <= (others=>'0');
    elsif (Initialize = '1') then
      Gamma <= one;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='0') then
        Gamma <= Delta;
      end if;
    end if;
  end if;
end process Gamma_Process;

N_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      N <= (others=>'0');
    elsif (Initialize = '1') then
      N <= (others=>'0');
    elsif (CountEnable = '1') then
      N <= N + 1;
    end if;
  end if;
end process N_Process;
```

Figure 54. Gamma and Variables L and N Process

```
Lambda_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        Lambda(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      Lambda(0) <= one;
      for i in 1 to RS_T loop
        Lambda(i) <= zero;
      end loop;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='1') then
        for i in 1 to RS_T loop
          Lambda(i) <= Lambda(i-1);
        end loop;
        Lambda(0) <= zero;
      else
        for i in 0 to RS_T loop
          Lambda(i) <= Mu(i);
        end loop;
      end if;
    end if;
  end if;
end process Lambda_Process;

B_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        B(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      B(0) <= one;
      for i in 1 to RS_T loop
        B(i) <= zero;
      end loop;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='1') then
        for i in 1 to RS_T loop
          B(i) <= B(i-1);
        end loop;
        B(0) <= zero;
      else
        for i in 0 to RS_T loop
          B(i) <= ZOmega(i);
        end loop;
      end if;
    end if;
  end if;
end process B_Process;
```

Figure 55. Lambda and Polynomial B Process

```
ZOmega_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        ZOmega(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      for i in 0 to RS_T loop
        ZOmega(i) <= (others=>'0');
      end loop;
    elsif (StoreNewPolys = '1') then
      for i in 1 to RS_T loop
        ZOmega(i) <= add(mul(Gamma,ZOmega(i)),mul(Delta,B(i-1)));
      end loop;
      ZOmega(0) <= mul(Gamma,ZOmega(0));
    end if;
  end if;
end process ZOmega_Process;

GammaMu_Process : process (Gamma, Mu)
begin
  for i in 0 to RS_T loop
    GammaMu(i) <= mul(Gamma, Mu(i));
  end loop;
end process GammaMu_Process;

DeltaLambda_Process : process (Delta, Lambda)
begin
  for i in 1 to RS_T loop
    DeltaLambda(i) <= mul(Delta, Lambda(i-1));
  end loop;
end process DeltaLambda_Process;

Mu_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        Mu(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') or (ErrorsPresent='0') then
      for i in 1 to RS_T loop
        Mu(i) <= (others=>'0');
      end loop;
      Mu(0) <= one;
    elsif (StoreNewPolys = '1') then
      for i in 1 to RS_T loop
        Mu(i) <= add(GammaMu(i),DeltaLambda(i));
      end loop;
      Mu(0) <= GammaMu(0);
    end if;
  end if;
end process Mu_Process;
```

Figure 56. Z Omega and Polynomial Mu Process

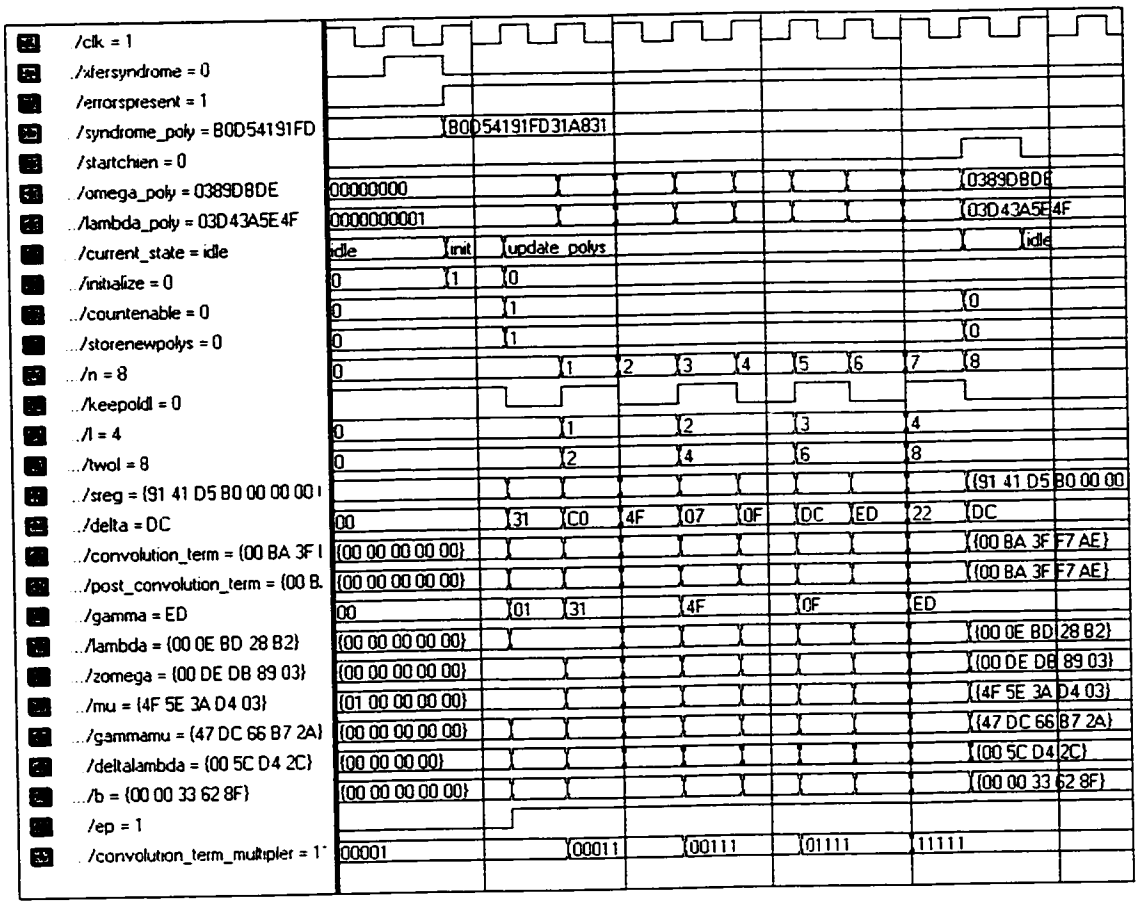

Figure 57. Extended Inversionless Massey-Berlekamp Simulation Waveforms

8.3 Chien Search and Error Correction

When the error correcting capability of an RS code is greater than two, the calculation of the correction factor requires evaluation of the derivative of the error-locator polynomial and the error-evaluator polynomial. The Chien search requires the evaluation of the error-locator polynomial. The evaluations are best handled in a pipeline for polynomial evaluation. This next section discusses the pipelines that are needed and their size. The pipeline for the error-locator polynomial and the error-evaluator polynomial will be called the EVAL_LAMBDA_PRIME and EVAL_OMEGA pipelines in the VHDL code. The Chien search pipeline is called the EVAL_CHIENSUM pipeline in the VHDL code.

Given a t-error correcting RS code, the error-locator polynomial, $\Lambda(x)$, and the error-evaluator polynomial, $\Omega(x)$ obtained from the extended inversionless Massey-Berlekamp algorithm are defined as follows :

$$\Lambda(x) = \sum_{i=0}^{t} \Lambda_i \cdot x^i \tag{7.1}$$

and

$$\Omega(x) = \sum_{i=0}^{t-1} \Omega_i \cdot x^i \tag{7.2}$$

The derivative of the error-locator polynomial, $\Lambda'(x)$, is used in Forney's algorithm to compute the error correction factor.

$$\Lambda'(x) = \sum_{i=0}^{k-1} \Lambda_{2i+1} \cdot x^{2i} \tag{7.3}$$

for t = 2k (that is, t is even)

$$\Lambda'(x) = \sum_{i=0}^{k-1} \Lambda_{2i+1} \cdot x^{2i} \tag{7.4}$$

for t = 2k + 1 (that is, t is odd).

Clearly then, the EVAL_LAMBDA_PRIME and EVAL_OMEGA pipeline stages for computing the error term must necessarily have a different relationship with respect to one another, when t is even or odd.

Specifically, for t odd, t = 2k + 1, the EVAL_OMEGA pipeline will be of length t-1 = 2k, while the EVAL_LAMBDA_PRIME pipeline will be of length t-3, and with 1 stage for the K factor multiplication and 1 stage for the inversion, brings up the total to t-1 stages for the denominator. However, since the EVAL_LAMBDA_PRIME pipeline starts one pipeline stage after the EVAL_OMEGA pipeline, a 1 delay stage is added to the EVAL_OMEGA pipeline to compensate. Note that, if no K factor multiplication stage is needed, then it is replaced by a pure delay stage, in order to match the pipeline delay of the EVAL_OMEGA pipeline.

For t even, t = 2k, the EVAL_OMEGA pipeline will be of length t-1, as before, while the EVAL_LAMBDA_PRIME pipeline will be of length t-2, and with 1 stage for the K factor multiplication and 1 stage for the inversion, brings up the total to t stages for the denominator. As an example, for t=8, we have :

$$\Lambda(x) = \Lambda_8 x^8 + \Lambda_7 x^7 + \Lambda_6 x^6 + \Lambda_5 x^5 + \Lambda_4 x^4 + \Lambda_3 x^3 + \Lambda_2 x^2 + \Lambda_1 x + \Lambda_0 \tag{7.5}$$

$$\Lambda'(x) = \Lambda_7 x^6 + \Lambda_5 x^4 + \Lambda_3 x^2 + \Lambda_1 \tag{7.6}$$

$$\Omega(x) = \Omega_7 x^7 + \Omega_6 x^6 + \Omega_5 x^5 + \Omega_4 x^4 + \Omega_3 x^3 + \Omega_2 x^2 + \Omega_1 x + \Omega_0 \tag{7.7}$$

whereas for t=7, we have :

$$\Lambda(x) = \Lambda_7 x^7 + \Lambda_6 x^6 + \Lambda_5 x^5 + \Lambda_4 x^4 + \Lambda_3 x^3 + \Lambda_2 x^2 + \Lambda_1 x + \Lambda_0 \tag{7.8}$$

$$\Lambda'(x) = \Lambda_7 x^6 + \Lambda_5 x^4 + \Lambda_3 x^2 + \Lambda_1 \tag{7.9}$$

$$\Omega(x) = \Omega_6 x^6 + \Omega_5 x^5 + \Omega_4 x^4 + \Omega_3 x^3 + \Omega_2 x^2 + \Omega_1 x + \Omega_0 \tag{7.10}$$

Note that the degree of $\Lambda`(x)$ is the same in both cases.

The following table summarizes the pipeline lengths and any delays needed for several values of t, and for general t.

Pipeline			t=8		t=7		t=6		t even t=2k		t odd t=2k+1	
X delayed (X1_D1, X1_D2, ..., X1_D5)	pipeline starts at	A	2		2		2		2		2	
	length of pipeline	B	5		4		3		t-3		t-3	
	pipeline ends at	A+B-1	6		5		4		$t-2^3$		$t-2^3$	
Powers of X (X2, X3, ..., X7)	pipeline starts at	A	1		1		1		1		1	
	length of pipeline	B	7		6		5		t-1		t-1	
	pipeline ends at	A+B-1	7		6		5		t-1		t-1	
EVAL_CHIENSUM	pipeline starts at	A	1		1		1		1		1	
	length of pipeline	B	9		8		7		t+1		t+1	
	delay stage[1]	C	0	0	1	0	0	0	0	0	1	0
	pipeline ends at	Sum(A..C)-1	**9**	**9**	**9**	**8**	**7**	**7**	**t+1**	**t+1**	**t+2**	**t+1**
EVAL_OMEGA	pipeline starts at	A	1		1		1		1		1	
	length of pipeline	B	8		7		6		t		t	
	delay stage[1]	C	1	1	2	1	1	1	1	1	2	1
	pipeline ends at	Sum(A..C)-1	**9**	**9**	**9**	**8**	**7**	**7**	**t+1**	**t+1**	**t+2**	**t+1**
EVAL_LAMBDAPRIME	pipeline starts at	A	2		2		2		2		2	
	length of pipeline	B	6		6		4		t-2		t-1	
	K_val stage[2]	C	1	0	1	0	1	0	1	0	1	0
	delay stage[1]	D	0	1	0	0	0	1	0	1	0	0
	inversion stage	E	1		1		1		1		1	
	pipeline ends at	Sum(A..E)-1	**9**	**9**	**9**	**8**	**7**	**7**	**t+1**	**t+1**	**t+2**	**t+1**

Table 10. Chien Search Pipeline Lengths.

NOTES :

1) In all cases, the three pipelines, EVAL_CHIENSUM , EVAL_OMEGA , and EVAL_LAMBDAPRIME must end at the same time. See the rows labeled "pipeline ends at". The rule for the delay stages are as follows :

a. EVAL_CHIENSUM delay stage = 1 if t is odd, and a K_val stage is needed, otherwise it is 0.

b. EVAL_OMEGA delay stage = 2 if t is odd, and a K_val stage is needed, otherwise it is 1.

c. EVAL_LAMBDAPRIME delay stage = 1 if t is even, and a K_val stage is not needed, otherwise it is 0.

2) The K_val stage corresponds to the multiplication of the EVAL_LAMBDA_PRIME pipeline by $\alpha^{-t(1-m_0)}$. It is needed if m_0, the log of the initial root is not 1. Forney's algorithm for the error, e_i, at position α^i is :

$$e_i = \frac{\alpha^{i(1-m_0)} \bullet \Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})} = \frac{\Omega(\alpha^{-i})}{\alpha^{-i(1-m_0)} \bullet \Lambda'(\alpha^{-i})} \tag{7.11}$$

Clearly, when m_0 is 1, the first term in the denominator becomes $\alpha^0 = 1$, and so, in this case : $e_i = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})}$, and thus the K_val stage is not needed.

The initial value of K_val is determined as follows. The initial value would be $\alpha^{(RS\_N-1)\bullet(1-m_0)}$, if there were no pipeline delay. The register for K_val is at the same pipeline stage as the last register of the EVAL_LAMBDAPRIME pipeline. For t even, the final register is at stage t-1, and so the initial value of K_val is

$$\alpha^{(RS\_N-1+(t-2))\bullet(1-m_0)} = \alpha^{(RS\_N+t-3)\bullet(1-m_0)} \tag{7.12}$$

For t odd, the final register is at stage t, and so the initial value of K_val is

$$\alpha^{(RS\_N-1+(t-1))\bullet(1-m_0)} = \alpha^{(RS\_N+t-2)\bullet(1-m_0)} \tag{7.13}$$

The coefficients of the error-locator polynomial, $\Lambda(x)$, are used in the Chien search. For a given value of t, the EVAL_CHIENSUM pipeline is of length t. To illustrate the various pipeline lengths, Figure 58, Figure 59, Figure 60, and Figure 61 show the Chien

various Chien search pipelines for t=4 and 5, with m_0=0 and 1. Figure 62 shows the block diagram for the error correction.

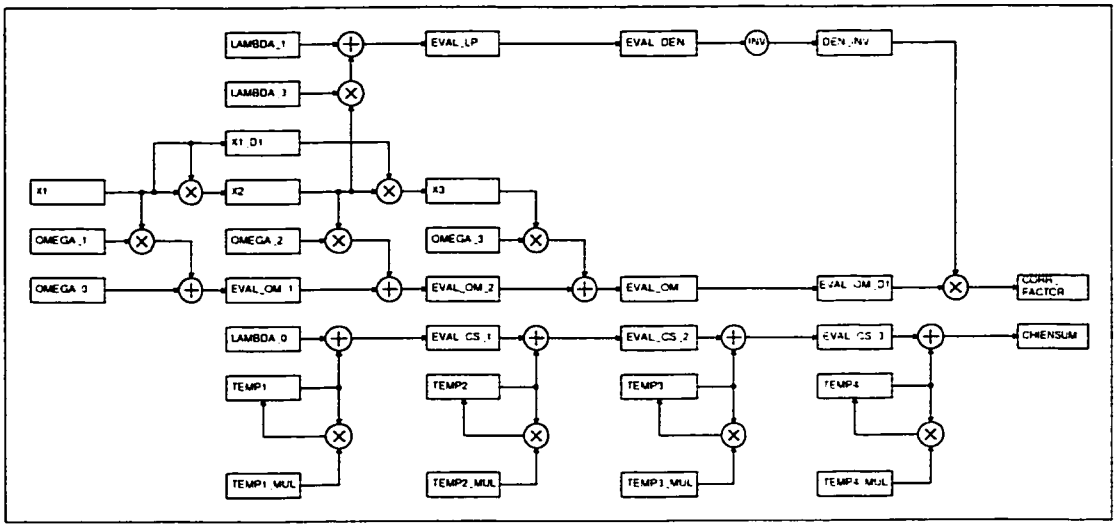

Figure 58. Chien Search Pipeline Block Diagram, t=4, m_0=1

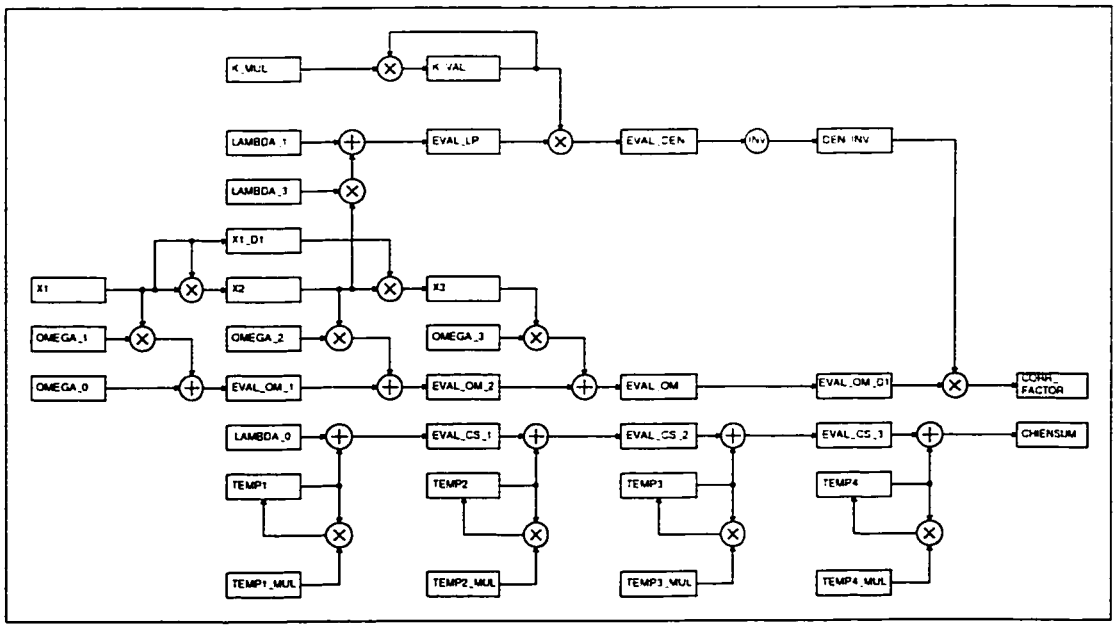

Figure 59. Chien Search Pipeline Block Diagram, t=4, m_0 not 1

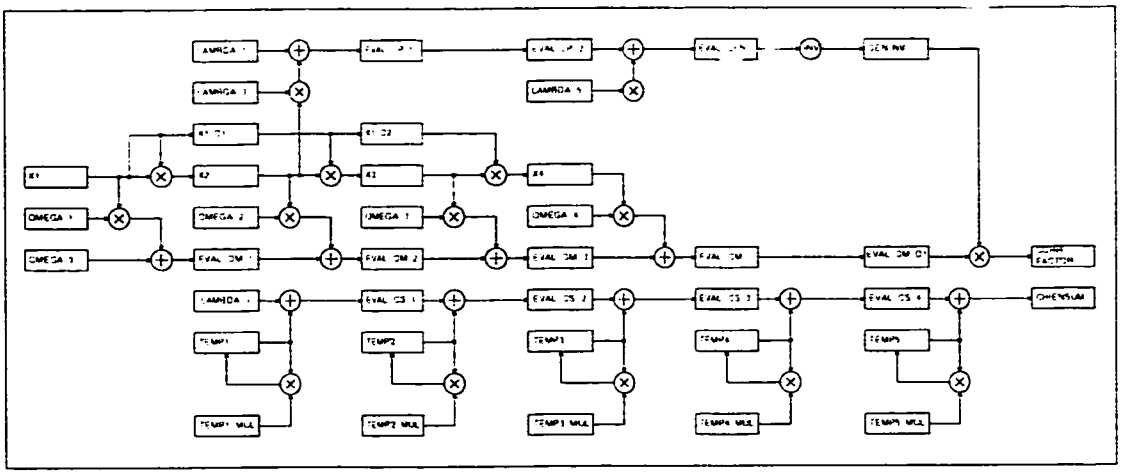

Figure 60. Chien Search Pipeline Block Diagram, t=5, $m_0 = 1$

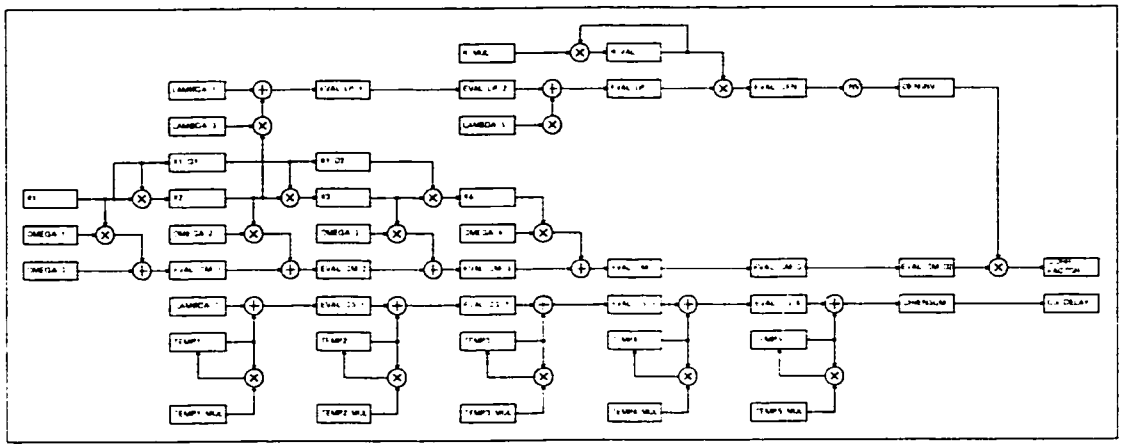

Figure 61. Chien Search Pipeline Block Diagram, t=5, m_0 not 1

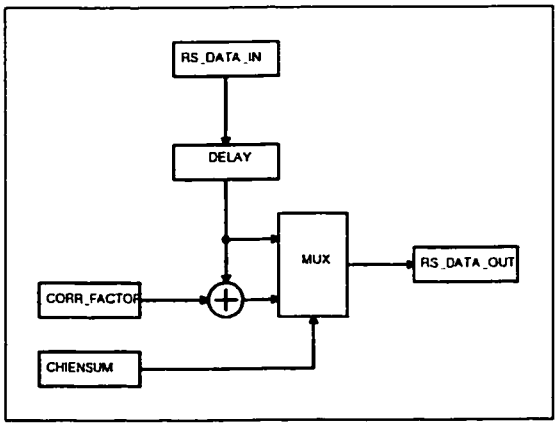

Figure 62. Error Correction

Figure 63 shows the VHDL constants and signal definitions for the Chien search block. In this case, m=8, N=22, t=4, and m_0=1. The control signals are those listed from `Count` to `OutputEnable`. The other signals are data flow related. Note that the only changes needed for other RS codes are the constants mentioned above, and the 2t powers of α that are needed in the actual Chien search. The remainder of the code can be written as is.

Figure 64 shows the VHDL code for the initialization of the local signals `IntLambda` and `IntOmega`. Theses signals are set to the incoming Lambda and Omega polynomial coefficients when the input strobe `StartChien` arrives. Also shown is the code for the system counter and the counter enable. As can be seen, the system counter, `Count`, is set to 0 when input strobe `StartChien` arrives, and increments every clock cycle as long as the `CountEnable` is active. The control signal is set to '1' (active) when `StartChien` arrives, and is set to '0' (inactive) when the counter reaches the maximum value as defined by the constant `CountMax`, which is equal to the sum of the total number of symbols, N, and the Chien search pipeline delay, t+1.

Figure 65 shows the code for two control signals, `InitChien` and `DoChien`, and the output strobe, `RS_Data_out_Start`. The signal `InitChien` is a one clock cycle delayed version of `StartChien`, and `DoChien` is a one clock strobe that happens after `InitChien`. The output strobe is a one clock period pulse that occurs when the counter reaches the value defined by `RS_Data_out_StartCount`, namely, the Chien search pipeline delay, t+1.

```
constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector(GFPower-1) downto 0);
constant RS_T  : integer := 4;
constant RS_N  : integer := 22;
constant RS_M0 : integer := 1;
constant key_equation_solver_delay : integer := 2*RS_T+2;
constant ChienSearch_pipeline_delay : integer := RS_T+1;
constant SR_max : integer := RS_N + key_equation_solver_delay +
ChienSearch_pipeline_delay;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant OutputEnableStopCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant DoChienCountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant CountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant     zero : Galois_Field_element := "00000000";
constant   alpha1 : Galois_Field_element := "00000010";
constant   alpha2 : Galois_Field_element := "00000100";
constant   alpha3 : Galois_Field_element := "00001000";
constant   alpha4 : Galois_Field_element := "00010000";
constant alpha234 : Galois_Field_element := "11111011";
constant alpha211 : Galois_Field_element := "10110010";
constant alpha186 : Galois_Field_element := "01101110";
constant alpha159 : Galois_Field_element := "01110011";

type Poly1Type is array(0 to RS_T) of Galois_Field_element;
type Poly2Type is array(0 to RS_T-1) of Galois_Field_element;
type Poly3Type is array(1 to RS_T) of Galois_Field_element;
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
signal Count : std_logic_vector(GFPower downto 0);
signal CountEnable : std_logic;
signal InitChien : std_logic;
signal DoChien : std_logic;
signal OutputEnable : std_logic;
signal shift_register_delay : shift_register_delay_type;
signal RS_data_delayed : Galois_Field_element;
signal Lambda : Poly1Type;
signal Omega  : Poly2Type;
signal IntLambda_0 : Galois_Field_element;
signal IntLambda_1 : Galois_Field_element;
signal IntLambda_3 : Galois_Field_element;
signal IntOmega  : Poly2Type;
signal X1 : Galois_Field_element;
signal X2 : Galois_Field_element;
signal X3 : Galois_Field_element;
signal X1_D1 : Galois_Field_element;
signal EVAL_OM_1 : Galois_Field_element;
signal EVAL_OM_2 : Galois_Field_element;
signal EVAL_OM     : Galois_Field_element;
signal EVAL_OM_D1  : Galois_Field_element;
signal EVAL_LP   : Galois_Field_element;
signal EVAL_DEN  : Galois_Field_element;
signal DEN_INV   : Galois_Field_element;
signal CORR_FACTOR : Galois_Field_element;
signal Temp1 : Galois_Field_element;
signal Temp2 : Galois_Field_element;
signal Temp3 : Galois_Field_element;
signal Temp4 : Galois_Field_element;
signal ChienSum : Galois_Field_element;
signal EVAL_CS_1 : Galois_Field_element;
signal EVAL_CS_2 : Galois_Field_element;
signal EVAL_CS_3 : Galois_Field_element;
```

Figure 63. Chien Search Constants and Signals

```
Lambda(  0) <= lambda_poly (   7 downto    0);
Lambda(  1) <= lambda_poly (  15 downto    8);
Lambda(  2) <= lambda_poly (  23 downto   16);
Lambda(  3) <= lambda_poly (  31 downto   24);
Lambda(  4) <= lambda_poly (  39 downto   32);
 Omega(  0) <=  omega_poly (   7 downto    0);
 Omega(  1) <=  omega_poly (  15 downto    8);
 Omega(  2) <=  omega_poly (  23 downto   16);
 Omega(  3) <=  omega_poly (  31 downto   24);

Init_IntLambda_and_IntOmega : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      IntLambda_0 <= zero;
      IntLambda_1 <= zero;
      IntLambda_3 <= zero;
      for i in 0 to RS_T-1 loop
        IntOmega(i) <= zero;
      end loop;
    elsif (StartChien = '1') then
      IntLambda_0 <= Lambda(0);
      IntLambda_1 <= Lambda(1);
      IntLambda_3 <= Lambda(3);
      for i in 0 to RS_T-1 loop
        IntOmega(i) <= Omega(i);
      end loop;
    end if;
  end if;
end process;

System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;

System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;
```

Figure 64. Internal Lambda and Omega Process and System Counter

```
InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
    else
      InitChien <= StartChien;
    end if;
  end if;
end process;

OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

RS_Data_out_Start_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;
```

Figure 65. Internal Control Signal Processes

Figure 66 shows the code for the X1 pipeline and the powers of X1 pipeline. The signal X1 represents the inverse of the present location in the Chien search. Its initial value is equal to α^{-N}. Every clock cycle after initialization, it is multiplied by α. The X1_D signals are delayed copies of X1, and are used to generate the powers of X1. The powers_of_X1 pipeline generates all of the powers up to t-1. These values are used in the polynomial evaluation of the derivative of the error-locator polynomial and the error-evaluator polynomial. The code is generic, except for the reference to the initial value of X1.

```
X1_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      X1    <= alpha234;
      X1_D1 <= zero;
    else
      X1    <= mul(X1,alpha1);
      X1_D1 <= X1;
    end if;
  end if;
end process;

Powers_of_X1_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      X2 <= zero;
      X3 <= zero;
    else
      X2 <= mul(X1,X1);
      X3 <= mul(X2,X1_D1);
    end if;
  end if;
end process;
```

Figure 66. Present Location X1 and Powers of X1 Pipelines

Figure 67 show the code for the evaluation of the derivative of the error-locator polynomial (EVAL_LP_Pipeline) and the error-evaluator polynomial (EVAL_OM_Pipeline). The EVAL_LP_Pipeline include the required inversion. Also included is the code for the correction factor calculation. This portion of the VHDL code is not generic, and must be tailored for each RS code as per Table 10.

Figure 68 shows the code for the Chien search pipeline. Since the Chien sum is formed by a pipeline, the Temp registers follow the pattern of Temp_i = $\alpha^{-(N+i-1)\cdot i}$. This ensures that the Chien sum at the end of the pipeline is correct. This portion of the VHDL code is also not generic, and must be tailored for each RS code as per Table 10.

Figure 69 shows the code for the data delay, and the error correction. The incoming delayed data is corrected by the correction factor when the Chien sum is 0. The length of the data delay, SR_max, is equal to RS_N + key_equation_solver_delay + ChienSearch_pipeline_delay. The code for these two processes is generic and can be reused for other RS codes without modification.

```
EVAL_OM_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_OM_1  <= zero;
      EVAL_OM_2  <= zero;
      EVAL_OM    <= zero;
      EVAL_OM_D1 <= zero;
    else
      EVAL_OM_1  <= add(IntOmega(0),mul(IntOmega(1),X1));
      EVAL_OM_2  <= add(EVAL_OM_1,mul(IntOmega(2),X2));
      EVAL_OM    <= add(EVAL_OM_2,mul(IntOmega(3),X3));
      EVAL_OM_D1 <= EVAL_OM;
    end if;
  end if;
end process;

EVAL_LP_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_LP   <= zero;
      EVAL_DEN  <= zero;
      DEN_INV   <= zero;
    else
      EVAL_LP   <= add(IntLambda_1,mul(IntLambda_3,X2));
      EVAL_DEN  <= EVAL_LP; -- RS_T_is_even, K_mul stage is not needed, EVAL_LAMBDAPRIME
delay stage = 1
      DEN_INV   <= inv(EVAL_DEN); -- K_mul stage is needed or EVAL_LAMBDAPRIME delay
stage = 1
    end if;
  end if;
end process;

Calc_Correction_Factor : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= (others=>'0');
    elsif (rs_enable = '1') then
      CORR_FACTOR   <= mul(EVAL_OM_D1,DEN_INV);
    else
      CORR_FACTOR <= (others=>'0');
    end if;
  end if;
end process;
```

Figure 67. Evaluation of Omega and Lambda Prime Pipelines, and Calculation of the Correction Factor

```
Calc_Temp_Registers : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1 <= zero;
      Temp2 <= zero;
      Temp3 <= zero;
      Temp4 <= zero;
    elsif (InitChien = '1') then
      Temp1 <= mul(alpha234,Lambda(1));  --  234 = (256 -  22)*  1 mod 255 =  234 mod 255
      Temp2 <= mul(alpha211,Lambda(2));  --  211 = (256 -  23)*  2 mod 255 =  211 mod 255
      Temp3 <= mul(alpha186,Lambda(3));  --  186 = (256 -  24)*  3 mod 255 =  186 mod 255
      Temp4 <= mul(alpha159,Lambda(4));  --  159 = (256 -  25)*  4 mod 255 =  159 mod 255
    elsif (DoChien = '1') then
      Temp1 <= mul(alpha1,Temp1);
      Temp2 <= mul(alpha2,Temp2);
      Temp3 <= mul(alpha3,Temp3);
      Temp4 <= mul(alpha4,Temp4);
    end if;
  end if;
end process;

Chien_Sum_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_CS_1   <= zero;
      EVAL_CS_2   <= zero;
      EVAL_CS_3   <= zero;
      ChienSum    <= zero;
    elsif (DoChien = '1') then
      EVAL_CS_1   <= add(IntLambda_0,Temp1);
      EVAL_CS_2   <= add(EVAL_CS_1,Temp2);
      EVAL_CS_3   <= add(EVAL_CS_2,Temp3);
      ChienSum    <= add(EVAL_CS_3,Temp4);
    end if;
  end if;
end process;
```

Figure 68. Chien Sum Pipeline

```
Delay_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to SR_max loop
        shift_register_delay(i) <= (others=>'0');
      end loop;
    else
      for i in SR_max downto 1 loop
        shift_register_delay(i) <= shift_register_delay(i-1);
      end loop;
      shift_register_delay(0) <= RS_Data_In;
    end if;
  end if;
end process;
RS_data_delayed <= shift_register_delay(SR_max);

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif ((ChienSum = zero) and (OutputEnable = '1')) then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    elsif (OutputEnable = '1') then
      RS_Data_out <= RS_data_delayed;
    else
      RS_Data_out <= zero;
    end if;
  end if;
end process;
```

Figure 69. Data Delay, and Error Correction Processes

9 Synthesis and Test Results for RS Decoders

RS decoder designs were synthesized for codes with different error-correcting capabilities across 3 values of the degree of the field generator polynomial, m, namely, 4, 6 and 8. The results are discussed in the next 2 sections.

9.1 RS Decoder Synthesis Speed Results

Several RS decoder designs were chosen as candidate designs. The error-correcting capability ranged from 1 to 7. This was applied to codes from $GF(2^4)$, $GF(2^6)$, and $GF(2^8)$. The synthesis flow consisted of the Synplicity synthesis tool (from Synplicity) and Xilinx software for the final place-and-route and timing analysis. The speed results are tabulated in Table 11 and shown graphically in Figure 70. As can be seen from the data, as the value of m increases, the maximum frequency decreases, due to the increasing complexity of the Galois field multipliers and inverters. Also, when t=1 or t=2, the maximum frequency is higher than when $t > 2$, due to the simpler decoding process, that is, no key equation solver. As t increases, when $t>2$, the maximum frequency is relatively flat. The input bit rate is shown along with the clock speed. Since the decoder is working on a Galois field symbol by symbol basis, the equivalent input bit rate is the clock speed multiplied by the Galois field symbol width 4, 6, or 8 as the case may be.

Error-correcting ability	$GF(2^4)$		$GF(2^6)$		$GF(2^8)$	
	Clock Speed (MHz)	Input Bit Rate (Msps)	Clock Speed (MHz)	Input Bit Rate (Msps)	Clock Speed (MHz)	Input Bit Rate (Msps)
1	97.94	391.8	105.33	632.0	89.53	716.2
2	134.37	537.5	129.89	779.3	98.14	785.1
3	90.33	361.3	82.37	494.2	60.53	484.3
4	84.60	338.4	76.98	461.9	59.03	472.3
5	84.60	338.4	74.52	447.1	59.38	475.1
6	75.47	301.9	74.52	447.1	59.38	475.1
7	82.37	329.5	70.57	423.4	59.59	476.8

Table 11. Maximum Decoder Speed (MHz) vs. Error Correcting Ability

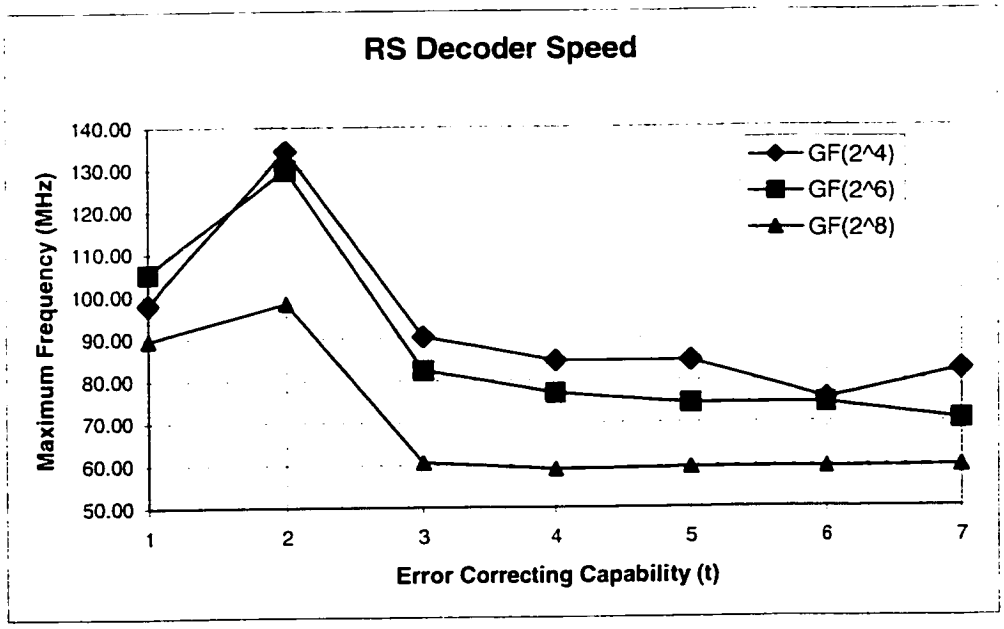

Figure 70. Reed-Solomon Decoder Maximum Speed

9.2 RS Decoder Synthesis Area Results

The area results are tabulated in Table 12 and shown graphically in Figure 71. It is clear from the results that area is a linear function of the error-correcting capability. There is a small kink in the curve at t=2 due to the change over in the decoding algorithms used. The decoder area increases with increasing error-correcting capability, again, as a consequence of evermore complex Galois field multipliers and inverters. As can be compared to the encoders, the RS decoders are bigger than the RS encoders by a factor of 6 to 20, or roughly, an order of magnitude.

Error-correcting ability	GF(2^4)	GF(2^5)	GF(2^8)
1	124	159	279
2	200	446	941
3	448	719	1336
4	535	929	1656
5	653	1110	2026
6	759	1352	2358
7	963	1473	2840

Table 12. Decoder Area (in slices) vs. Error Correcting Ability

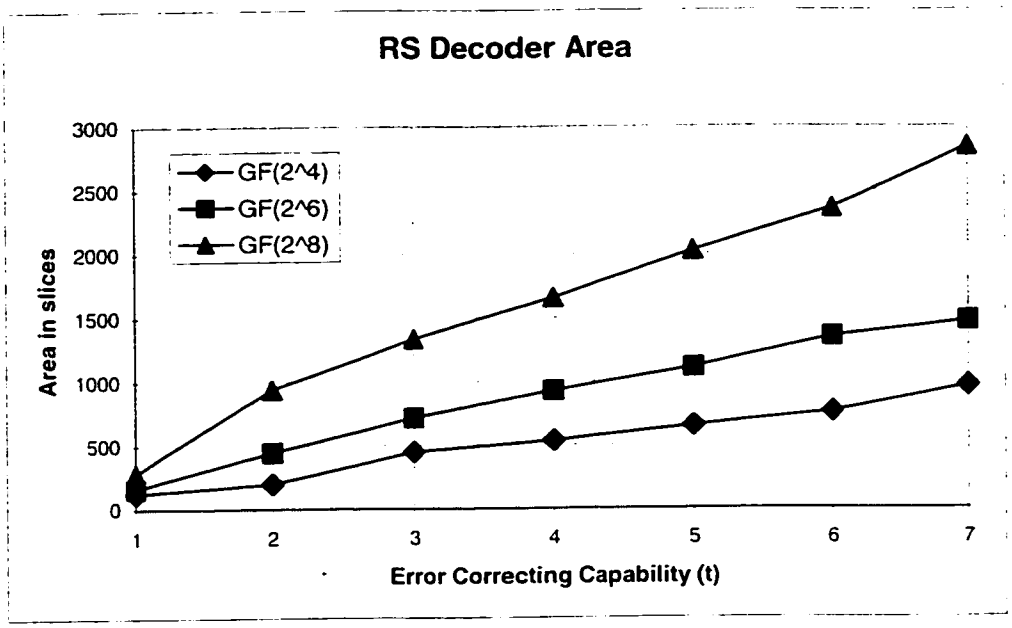

Figure 71. Reed-Solomon Decoder Area (in slices)

9.3 Decoder Testbench

As for the encoders, the performance of the decoders described above was verified with a VHDL testbench. Fixed and random error patterns were injected, and the output of the syndrome calculation block, the key equation solver, and the Chien search were verified. The VHDL code for the decoder testbench is shown in Appendix F.

10 Comparison of Generated Cores to Available Cores

10.1 Encoder Cores

This section will compare the area and speed metrics obtained for the generated encoder cores with those of other commercially available encoder cores. Table 13 summarizes the area and speed metrics for several encoder cores. The encoders listed use GF(2^8), and are 8 error-correcting RS codes. The information regarding these cores is limited to the applicable data sheets. As such, the target technology may not be exactly the same as that listed for the encoder cores listed in this thesis. Two of the commercial cores have targeted reconfigurable hardware, (Actel and Xilinx). The other two target ASIC technology. It should be noted that the target technology used in this thesis, namely the Xilinx XCV1000E Field Programmable Gate Array, is not the fastest series of FPGA from Xilinx. The XCV1000E is part of the Virtex I series. Over the last 2 years, this has been superceded by the Virtex II and the Virtex II Pro series, both of which include parts with higher density and better performance. However, even considering this fact, the observed performance in terms of clock frequency is comparable with commercial cores.

Company	Target Technology	Speed	Area	Reference
Machine Learning Laboratory Inc.,	Actel Axcelerator AX500-3	119 MHz	7%	7%
Memec Design	Xilinx V50-6	113 MHz	94 CLBs	[20]
4i2i Communications	ASIC 0.35 micron	318 MHz	2260 gates	[21]
Telecommunications and Information Technologies	ASIC 0.6 micron	55 MHz	2464 gates	[22]

Table 13. Device Utilization and Performance of Existing Encoder Cores

Table 14 shows the characteristics of the Xilinx Virtex FPGA Family of which the XCV1000E is a member. The quoted [14] system performance is 200 MHz, and an equivalent 1,124,022 system gates. Table 15 shows the characteristics of the Actel Axcelerator Family of which the AX500-3 is a member. In terms of a direct comparison, the equivalent system gates will be used. For an encoder using GF(2^8) and correcting up to 8 errors, the number of system gates is 3080 , running at 140 MHz. The Actel core 35000 system gates, running at 119 MHz. While the frequencies of operation are comparable, the equivalent system gates between the two technologies differ considerably.

The most direct comparison can be made with the Memec core, which is targeted, to the Xilinx Virtex V50-6 FPGA. This core runs at 113 MHz and uses 94 CLBs, whereas the generated cores run at 140 MHz and use 80 CLBs. The next 2 cores are targeted to ASIC technology, and we can see that in the case of the core from 4i2i Communications is over 2 times the speed of the FPGA core, which is to be expected. ASICs are still faster than the best FPGAs. A comparison of area is not possible since the target technologies are too radically different.

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+™ Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576
XCV100	108,904	20x30	2,700	180	40,960	38,400
XCV150	164,674	24x36	3,888	260	49,152	55,296
XCV200	236,666	28x42	5,292	284	57,344	75,264
XCV300	322,970	32x48	6,912	316	65,536	98,304
XCV400	468,252	40x60	10,800	404	81,920	153,600
XCV600	661,111	48x72	15,552	512	98,304	221,184
XCV800	888,439	56x84	21,168	512	114,688	301,056
XCV1000	1,124,022	64x96	27,648	512	131,072	393,216

Table 14. Xilinx Virtex FPGA Family Members

	AX125	AX250	AX500	AX1000	AX2000
Equiv. System Gates	125,000	250,000	500,000	1,000,000	2,000,000
Typical Gates	82,000	154,000	286,000	612,000	1,060,000
Total RAM Bits	29,184	71,680	95,232	198,912	338,688
Max Registers	1,344	2,816	5,376	12,096	21,504
Total Modules	2,016	4,224	8,064	18,144	32,256
Dedicated Registers	672	1,408	2,688	6,048	10,752
RAM Bocks	4	12	16	36	64
PerPin FIFOs	168	248	336	516	684
Max No. of LVDS Pairs	84	124	168	258	342
PLL's	8	8	8	8	8
I/O Registers	504	744	1,008	1,548	2,052
User I/O's	168	248	336	516	684
Packages	CS180 FG256 FG324	FG256 FG484	FG484 FG676	FG484 FG676 BG729 FG896	FG896 FG1152

Table 15. Axcelerator Family Selection Guide

10.2 Decoder Cores

This section will compare the area and speed metrics obtained for the generated decoder cores with those of other commercially available decoder cores. Table 16 summarizes the area and speed metrics for several decoder cores. The decoders listed use GF(2^8), and are 8 error-correcting RS codes. Four of the commercial cores have targeted reconfigurable hardware, (Altera and Xilinx). The other core was targeted to ASIC technology.

The equivalent core produced by the VHDL core generator resulted in the use 3070 slices, and 77.6 MHz frequency of operation in a Xilinx XC2V500-4. This is more area compared to the average of the 3 cores that are specified for use with Xilinx FPGAs. In terms of frequency of operation, it is comparable with 4 of the cores. The first core from Machine Learning Laboratory Inc is quite fast, running at 143 MHz. In general, the core produced by the VHDL core generator is close to the average speed mentioned in the cores listed in Table 16. The area is approximately 50% more than the average.

Company	Target Technology	Speed	Area	Reference
Machine Learning Laboratory Inc.,	Xilinx X2V1000-6	143 MHz	1950 slices	[17]
Memec Design	Xilinx XC2V250-6	77 MHz	1201 slices	[18]
Paxonet Communications	Xilinx XC2V500-4	97.9 MHz	2545 slices	[19]
4i2i Communications	ASIC 0.35 micron	75 MHz	37000 gates	[21]
Altera	Altera FLEX 10KE	61 MHz	2431 logic elements	[23]

Table 16. Device Utilization and Performance of Existing Decoder Cores

11 Conclusion

This thesis has introduced and proved the extended inversionless Massey-Berlekamp algorithm, which is a modest improvement over the inversionless Massey-Berlekamp algorithm for solving the key equation in a Reed-Solomon decoder. The error location and correction equations for an arbitrary RS code with an error correcting capability of less than or equal to 2 errors was also presented. Using this information, VHDL based designs for specific RS codes were implemented for both encoder and decoder. Once this was done, a program was written to automatically generate the VHDL code for either an RS encoder, or an RS decoder for an arbitrary RS code based on user inputs. It was shown that much of the VHDL code could be reused as is for any RS code, be it encoder or decoder. The rest of the code needed only minor modifications, or specifications of different constants to implement the new functionality. The design of the RS encoder and decoder were pipelined, enabling a substantial bit rate of operation. For the larger decoders, typical of those used in modern digital communication systems, bit rates over 450 Mbits/sec were achieved on FPGAs. For the encoders, bit rates of over 1 Gbits/sec were achieved on FPGAs. The end result is thus a VHDL core generator for Reed-Solomon encoders and decoders.

The following items may be considered for future work.

1. The VHDL code generated was written specifically to be as generic as possible, that is, no specific FPGA attributes were coded for. As such, delay blocks were implemented with shift register arrays. One could, however, code for specific use of the Xilinx FPGAs, namely, the use of BlockRAM, a large amount of internal dual port RAM available within a FPGA, which is there whether or not you use it. These RAMs could be used to implement delay blocks, and also the Galois field inverter. This will result in fewer CLB's (logic blocks) being used.
2. The representation of the Galois field elements is that standard polynomial form. Other representations exist, such as dual basis representation used in the

Consultative Committee for Space Data Systems (CCSDS) standard [15], or the composite field representation suggested by C. Paar [16]. The composite field representation is or particular interest since it has been shown that faster and smaller Galois field multipliers and inverters are possible using this technique.

3. There exist several other algorithms for solving the key equation, namely, Euclid's algorithm, and the original Massey-Berlekamp algorithm. These could be substituted for the extended inversionless Massey-Berlekamp algorithm presented here.

12 References

1. D. Gorenstein And N. Zierler, "A Class of Error Correcting Codes in p^m Symbols", *Journal of the Society of Industrial and Applied Mathematics*, Vol. 9, pp. 207-214, June 1961.
2. Y. Sugiyama, Y. Kasahara, S. Hirasawa, And T. Namekawa, "A Method for Solving Key Equation for Goppa Codes", *Information and Control*, Vol. 27, pp. 87-89, 1975.
3. E.R. Berlekamp, "*Algebraic Coding Theory*", (McGraw-Hill, New York, 1968), chapter 7.
4. J.L. Massey, "Shift-register synthesis and BCH decoding", IEEE Transactions on Information Theory, 1969, pp. 122-127.
5. Arnold M. Michelson And Allen H. Levesque, "*Error Control Techniques for Digital Communication*", John Wiley & Sons, 1985.
6. I.S. Reed, M.T.Shih, T.K. Truong, "*VLSI design of Inverse-free Berlekamp-Massey algorithm*", IEE Proceedings-E, Vol.138, No.5, September 1991.
7. Richard E. Blahut, "*Algebraic Methods for Signal Processing and Communications Coding*", (Springer-Verlag, New York, 1992), chapter 8.
8. George C. Clark, Jr. And J. Bibb Cain, "*Error-Correction Coding for Digital Communications*", (Plenum Press, New York, 1981), chapter 5.
9. G.D. Forney, Jr., "*On Decoding BCH Codes*", IEEE Transactions on Information Theory, 1965, pp. 549-557.
10. S. Lin And D.J. Costello Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice Hall, 1983
11. Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, NJ: Prentice Hall, 1983
12. R. W. Hamming, *Error Detecting and Error Correcting Codes*, Bell System Technical Journal, Vol. 29, pp. 147-160, 1950.

13. V. Glavac and M.R. Soleymani, "An Extended Inversion-less Massey-Berlekamp Algorithm," 2000 Conference on Systemics, Cybernetics and Informatics, SCI2000, Orlando, Florida, July 23-26, 2000.

14. Xilinx Inc., "Virtex-II Data Sheet, DS031-2 Virtex-II 1.5 V FPGAs; Introduction and Ordering Information" July 2000, http://www.xilinx.com/partinfo/ds031-1.pdf.

15. Consultative Committee for Space Data Systems Standard CCSDS 101.0-B-4, Recommendations for Space Data System Standards, Telemetry Channel Coding, NASA, May 1999.

16. C. Paar and M. Rosner, "Comparison of Arithmetic Architectures for Reed-Solomon Decoders in Reconfigurable Hardware", FCCM '97

17. Machine Learning Laboratory Inc., "Reed-Solomon Code Encoder/Decoder", http://www.ml-labo.com/eng/ipcore_rs.htm

18. Xilinx Alliance Core – Memec Design, "XF_RSDEC Data Sheet", http://www.xilinx.com/products/logicore/alliance/memec/xf_rsdec.pdf

19. Xilinx Alliance Core – Paxonet Communications, "G.709 Compliant FEC Core". http://www.jp.xilinx.com/products/logicore/alliance/coreel/paxonet_cc345.pdf

20. Xilinx Alliance Core – Memec Design, "XF_RSENC Data Sheet", http://www.xilinx.com/products/logicore/alliance/memec/xf_rsenc.pdf

21. 4i2i Communications Ltd., "Reed-Solomon Core Data Sheet". http://www.4i2i.com/4i2iRS_V2_220402.PDF

22. Telecommunications and Information Technologies, "Reed-Solomon Encoder Data Sheet", http://www.us.design-reuse.com/EXCHANGER/TTI/RSencoder.doc

23. Altera Corporation, "Reed-Solomon Compiler MegaCore Function", http://www.altera.ru/Altera/sb/sb048.pdf

24. Peter J. Ashenden, "The Designer's Guide to VHDL", Morgan Kaufmann, 2001.

25. J. Bhasker, Jayaram Bhasker, "A VHDL Primer", 3rd edition, Prentice-Hall, 1998.

26. Shadia Hijazie, "High-level Synthesis and Its Application in the Design of Reed-Solomon Decoders", Master's Thesis, Concordia University, July 1997.

27. Ernest Jamro, "The Design of a VHDL Based Synthesis Tool For BCH Codecs", Master's Thesis, University of Huddersfield, July 1997.

28. Pablo Moisset, Joonseok Park and Pedro Diniz. "Very High-Level Synthesis of Contol and Datapath Structures for Reconfigurable Logic Devices" In Proceedings of the Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99). Oct. 99, Washington, DC, USA

29. Christian Schuler, "Code Generation Tools For Hardware Implementation of FEC Circuits", http://www.fokus.gmd.de/research/cc/mobis/publ/doc/1999/code_gen.pdf.

30. Jong Kang Park and Jong Tae Kim. "A Soft IP Compiler For Reed-Solomon Decoder", http://www.sipac.org/ap-soc/index_k/index_a/source/A04_lec.pdf.

31. A. Haase, C. Kretzschmar, R. Siegmund, D. Muller, J, Schneider, M. Boden, M. Langer, "Design of a Reed Solomon Decoder using Partial Dynamic Reconfiguration of XILINX Virtex FPGAs – A Case Study", http://people.eas.iis.fhg.de/users/jsch/veroeffentlichungen/date2002_df.pdf.

32. Y. Shayan, T. Le-Ngoc and V. Bhargava, "A Versatile Time-Domain Reed-Solomon Decoder", IEEE JSAC, Vol. 8, No. 8, October 1990.

13 Appendix A - RS Encoder VHDL Code

This appendix contains the VHDL code generated for a RS encoder for a 3-error correcting code over $GF(2^4)$, with $p(x) = x^4 + x + 1$.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  rs_encoder  is
port (reset_n : in std_logic ;
      data_out : out std_logic_vector (3 downto 0);
      data_in : in std_logic_vector (3 downto 0);
      input_strobe : in std_logic ;
      clk : in std_logic ;
      output_strobe : out std_logic ;
      data_size : in std_logic_vector (3 downto 0));
end;

architecture  RTL  of  rs_encoder  is
constant GFPower : integer := 4;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant   RS_T  : integer := 3;
constant     zero : Galois_Field_element := "0000";
constant      one : Galois_Field_element := "0001";
constant   alpha4 : Galois_Field_element := "0011";
constant   alpha6 : Galois_Field_element := "1100";
constant   alpha9 : Galois_Field_element := "1010";
constant  alpha10 : Galois_Field_element := "0111";
constant  alpha14 : Galois_Field_element := "1001";
signal DoCalc : std_logic;
signal count : std_logic_vector ((GFPower-1) downto 0);
signal sum : Galois_Field_element;
type parity_reg_type is array(0 to (2*RS_T-1)) of Galois_Field_element;
signal parity_reg : parity_reg_type;

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor (b(3) and c(1));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor
          (b(2) and c(3)) xor (b(3) and c(1)) xor (b(3) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(2) and c(0)) xor (b(2) and c(3)) xor
          (b(3) and c(2)) xor (b(3) and c(3));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor (b(3) and c(0)) xor
          (b(3) and c(3));
  return d;
end mul;

function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  return d;
end add;

begin
process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      count <= (others => '0');
    elsif (input_strobe = '1') then
      count <= one;
    elsif (DoCalc = '1') then
      count <= count + 1;
    else count <= (others => '0');
    end if;
  end if;
end process;
```

```
process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      DoCalc <= '0';
    elsif (input_strobe = '1') then
      DoCalc <= '1';
    elsif (count=data_size) then
      DoCalc <= '0';
    end if;
  end if;
end process;

process (parity_reg, data_in)
begin
  sum <= add(parity_reg((2*RS_T-1)),data_in);
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to (2*RS_T-1) loop
        parity_reg(i) <= (others=>'0');
      end loop;
    elsif (input_strobe = '1') then
      for i in 0 to (2*RS_T-1) loop
        parity_reg(i) <= (others=>'0');
      end loop;
    elsif (DoCalc = '1') then
      parity_reg(  5) <= add(parity_reg(  4),mul(sum,alpha10));
      parity_reg(  4) <= add(parity_reg(  3),mul(sum,alpha14));
      parity_reg(  3) <= add(parity_reg(  2),mul(sum,alpha4));
      parity_reg(  2) <= add(parity_reg(  1),mul(sum,alpha6));
      parity_reg(  1) <= add(parity_reg(  0),mul(sum,alpha9));
      parity_reg(  0) <= mul(sum,alpha6);
    else
      for i in (2*RS_T-1) downto 1 loop
        parity_reg(i) <= parity_reg(i-1);
      end loop;
      parity_reg(0) <= (others=>'0');
    end if;
  end if;
end process;

process(clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      data_out <= (others => '0');
    elsif (DoCalc='1') then
      data_out <= data_in;
    else
      data_out <= parity_reg((2*RS_T-1));
    end if;
  end if;
end process;

process(clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      output_strobe <= '0';
    else
      output_strobe <= input_strobe;
    end if;
  end if;
end process;
end;
```

14 Appendix B - RS Decoder VHDL Code (1 error)

This appendix contains the VHDL code generated for an RS decoder for a 1-error correcting code over GF(2^6), with $p(x) = x^6 + x + 1$. The value of N is 14 and the value of the log of the initial root of the code generator polynomial is 25. This is a RS(14,12) code.

```
----------------------------------------
-- RS Decoder Core Generator
-- Version 1.0
-- written by Vladimir Glavac 4182200
--
-- Reed-Solomon Decoder Parameters
--
-- RS_N  =  14 = length of codeword
--
-- RS_T  =   1 = error correcting capability
--
-- RS_M0 =  37 = power of initial root of code generator polynomial
--
-- code generator polynomial g(x) =
--
--      a0 * x^2
--   +  a43 * x^1
--   +  a12
--
--
-- field generator polynomial p(x) = [1000011] = x^6 + x^1 + 1
--
----------------------------------------

----------------------------------------
-- Start of InputProcess
----------------------------------------


library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  InputProcess  is
port (
      reset_n : in std_logic ;
      clk : in std_logic ;
      rs_data_in : in std_logic_vector (5 downto 0);
      rs_data_in_start : in std_logic ;
      syndrome_calc_done : out std_logic ;
      errors_present : out std_logic ;
      syndrome : out std_logic_vector (11 downto 0)
);
end;

architecture  RTL  of  InputProcess  is
constant GFPower : integer := 6;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 1;
constant RS_N : integer := 14;
constant RS_M0 : integer := 37;
constant Two_T_minus_1 : integer := 2*RS_T-1;
constant     zero : Galois_Field_element := "000000";
constant      one : Galois_Field_element := "000001";
constant  alpha37 : Galois_Field_element := "101100";
constant  alpha38 : Galois_Field_element := "011011";
type IntS_type is array(0 to Two_T_minus_1) of Galois_Field_element;
type IntEP_type is array(0 to Two_T_minus_1) of std_logic;
type state_type is (Idle, RxData, XferData);
signal present_state : state_type;
signal IntCount : std_logic_vector (3 downto 0);
signal internal_errors_present : std_logic;
signal CountEnable : std_logic;
signal CountReset : std_logic;
signal StartXfer : std_logic;
signal XferSyndrome : std_logic;
signal DoCalc : std_logic;
signal IntS : IntS_type;
signal IntEP : IntEP_type;
```

```
function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  return d;
end add;

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(4) and c(2)) xor (b(5) and c(1));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor
          (b(2) and c(5)) xor (b(3) and c(3)) xor (b(3) and c(4)) xor (b(4) and c(2)) xor
          (b(4) and c(3)) xor (b(5) and c(1)) xor (b(5) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(2) and c(0)) xor (b(2) and c(5)) xor
          (b(3) and c(4)) xor (b(3) and c(5)) xor (b(4) and c(3)) xor (b(4) and c(4)) xor
          (b(5) and c(2)) xor (b(5) and c(3));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor (b(3) and c(0)) xor
          (b(3) and c(5)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(5) and c(3)) xor
          (b(5) and c(4));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor (b(3) and c(1)) xor
          (b(4) and c(0)) xor (b(4) and c(5)) xor (b(5) and c(4)) xor (b(5) and c(5));
  d(5) := (b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(3) and c(2)) xor
          (b(4) and c(1)) xor (b(5) and c(0)) xor (b(5) and c(5));
  return d;
end mul;

begin

process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (CountReset = '1')) then
      IntCount <= (others=>'0');
    elsif (CountEnable = '1') then
      IntCount <= IntCount + 1;
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      StartXfer <= '0';
    elsif (IntCount="1011") then -- RS(14,12) : max_count = 11
      StartXfer <= '1';
    else
      StartXfer <= '0';
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= zero;
      end loop;
    elsif (rs_data_in_start = '1') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= rs_data_in;
      end loop;
    elsif (DoCalc='1') then
```

```
        IntS(0) <= add(mul(alpha37,IntS(0)),rs_data_in);
        IntS(1) <= add(mul(alpha38,IntS(1)),rs_data_in);
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      syndrome <= (others=>'0');
      internal_errors_present <= '0';
    elsif (XferSyndrome='1') then
      syndrome <=
        IntS(1) & IntS(0) ;
      internal_errors_present <=
        IntEP(0) or IntEP(1) ;
    end if;
  end if;
end process;

process (IntS)
begin
  for i in 0 to Two_T_minus_1 loop
    IntEP(i)<='1';
    if (IntS(i)=zero) then IntEP(i)<= '0'; end if;
  end loop;
end process;

InputControlSD_Idle : process (clk)
begin
if (clk'event and clk = '1') then
  if (reset_n = '0') then
    XferSyndrome<='0';
    DoCalc<='0';
    CountReset<='1';
    CountEnable<='0';
    present_state <= Idle;
  else
    case present_state is
      when Idle =>
        if (rs_data_in_start = '1') then
          DoCalc<='1';
          CountReset<='0';
          CountEnable<='1';
          present_state <= RxData;
        else
          present_state <= Idle;
        end if;
      when RxData =>
        if (StartXfer = '1') then
          XferSyndrome<='1';
          DoCalc<='0';
          CountReset<='1';
          CountEnable<='0';
          present_state <= XferData;
        else
          present_state <= RxData;
        end if;
      when XferData =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
      when others =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
```

```
    end case;
  end if;
end if;
end process;

syndrome_calc_done <= XferSyndrome;
errors_present <= internal_errors_present;

end;

----------------------------------------
-- End of InputProcess
----------------------------------------


----------------------------------------
-- Start of ChienSearchProcess
----------------------------------------

library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity  ChienSearchProcess  is
port (
      clk : in std_logic ;
      reset_n : in std_logic ;
      syndromes : in std_logic_vector (11 downto 0);
      StartChien : in std_logic ;
      rs_enable : in std_logic ;
      RS_Data_In : in std_logic_vector (5 downto 0);
      RS_Data_out : out std_logic_vector (5 downto 0);
      RS_Data_out_Start : out std_logic
  );
end;

architecture  RTL  of  ChienSearchProcess  is
constant GFPower : integer := 6;
subtype Galois_Field_element is std_logic_vector(GFPower-1 downto 0);
constant RS_T  : integer := 1;
constant RS_N  : integer := 14;
constant RS_M0 : integer := 37;
constant ChienSearch_pipeline_delay : integer := 3;
constant SR_max : integer := RS_N + ChienSearch_pipeline_delay;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant OutputEnableStopCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant DoChienCountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-2, GFPower+1) ;
constant CountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant     zero : Galois_Field_element := "000000";
constant      one : Galois_Field_element := "000001";
constant Temp1_initial_value : Galois_Field_element := "110100"; -- alpha(-(RS_N-1)) =
a50
constant  x0a_mul : Galois_Field_element := "101100"; -- alpha(RS_M0) =  a37
constant   alpha1 : Galois_Field_element := "000010";
constant x0a_initial_value : Galois_Field_element := "101001"; -- alpha(-(RS_N-1)*RS_M0)
=  a23
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
signal shift_register_delay : shift_register_delay_type;
signal RS_data_delayed : Galois_Field_element;
signal CORR_FACTOR : Galois_Field_element;
signal Count : std_logic_vector(GFPower downto 0);
signal CountEnable : std_logic;
signal InitChien : std_logic;
signal InitChien_d1 : std_logic;
signal DoChien : std_logic;
```

```
signal OutputEnable : std_logic;
signal inv_s0 : Galois_Field_element;
signal x0a : Galois_Field_element;
signal Temp1 : Galois_Field_element;
type syndrome_type is array(0 to 2*RS_T-1) of Galois_Field_element;
signal S : syndrome_type ;

function inv (b : in std_logic_vector) return std_logic_vector is
  variable d : std_logic_vector (5 downto 0);
begin
  d := "000000";
     if (b="000001") then d := "000001";
  elsif (b="000010") then d := "100001";
  elsif (b="000011") then d := "111110";
  elsif (b="000100") then d := "110001";
  elsif (b="000101") then d := "101011";
  elsif (b="000110") then d := "011111";
  elsif (b="000111") then d := "101100";
  elsif (b="001000") then d := "111001";
  elsif (b="001001") then d := "100101";
  elsif (b="001010") then d := "110100";
  elsif (b="001011") then d := "011100";
  elsif (b="001100") then d := "101110";
  elsif (b="001101") then d := "101000";
  elsif (b="001110") then d := "010110";
  elsif (b="001111") then d := "011001";
  elsif (b="010000") then d := "111101";
  elsif (b="010001") then d := "110110";
  elsif (b="010010") then d := "110011";
  elsif (b="010011") then d := "100111";
  elsif (b="010100") then d := "011010";
  elsif (b="010101") then d := "100011";
  elsif (b="010110") then d := "001110";
  elsif (b="010111") then d := "011000";
  elsif (b="011000") then d := "010111";
  elsif (b="011001") then d := "001111";
  elsif (b="011010") then d := "010100";
  elsif (b="011011") then d := "100010";
  elsif (b="011100") then d := "001011";
  elsif (b="011101") then d := "110101";
  elsif (b="011110") then d := "101101";
  elsif (b="011111") then d := "000110";
  elsif (b="100000") then d := "111111";
  elsif (b="100001") then d := "000010";
  elsif (b="100010") then d := "011011";
  elsif (b="100011") then d := "010101";
  elsif (b="100100") then d := "111000";
  elsif (b="100101") then d := "001001";
  elsif (b="100110") then d := "110010";
  elsif (b="100111") then d := "010011";
  elsif (b="101000") then d := "001101";
  elsif (b="101001") then d := "101111";
  elsif (b="101010") then d := "110000";
  elsif (b="101011") then d := "000101";
  elsif (b="101100") then d := "000111";
  elsif (b="101101") then d := "011110";
  elsif (b="101110") then d := "001100";
  elsif (b="101111") then d := "101001";
  elsif (b="110000") then d := "101010";
  elsif (b="110001") then d := "000100";
  elsif (b="110010") then d := "100110";
  elsif (b="110011") then d := "010010";
  elsif (b="110100") then d := "001010";
  elsif (b="110101") then d := "011101";
  elsif (b="110110") then d := "010001";
  elsif (b="110111") then d := "111100";
  elsif (b="111000") then d := "100100";
  elsif (b="111001") then d := "001000";
  elsif (b="111010") then d := "111011";
  elsif (b="111011") then d := "111010";
  elsif (b="111100") then d := "110111";
```

```
  elsif (b="111101") then d := "010000";
  elsif (b="111110") then d := "000011";
  elsif (b="111111") then d := "100000";
  end if;
  return d;
end inv;


function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  return d;
end add;


function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(4) and c(2)) xor (b(5) and c(1));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor
          (b(2) and c(5)) xor (b(3) and c(3)) xor (b(3) and c(4)) xor (b(4) and c(2)) xor
          (b(4) and c(3)) xor (b(5) and c(1)) xor (b(5) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(2) and c(0)) xor (b(2) and c(5)) xor
          (b(3) and c(4)) xor (b(3) and c(5)) xor (b(4) and c(3)) xor (b(4) and c(4)) xor
          (b(5) and c(2)) xor (b(5) and c(3));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor (b(3) and c(0)) xor
          (b(3) and c(5)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(5) and c(3)) xor
          (b(5) and c(4));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor (b(3) and c(1)) xor
          (b(4) and c(0)) xor (b(4) and c(5)) xor (b(5) and c(4)) xor (b(5) and c(5));
  d(5) := (b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(3) and c(2)) xor
          (b(4) and c(1)) xor (b(5) and c(0)) xor (b(5) and c(5));
  return d;
end mul;


function IsNotZero(b  : in std_logic_vector) return std_logic is
       variable d : std_logic;
begin
 d :=
      b(0) or
      b(1) or
      b(2) or
      b(3) or
      b(4) or
      b(5) ;
 return d;
end IsNotZero;

begin

S(1) <= syndromes(11 downto 6);
S(0) <= syndromes(5 downto 0);

System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;
```

```
System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;

InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
      InitChien_d1 <= '0';
    else
      InitChien <= StartChien;
      InitChien_d1 <= InitChien;
    end if;
  end if;
end process;

OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

RS_Data_out_Start_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien_d1 = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;

Calc_Correction_Factor : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= zero;
    elsif (rs_enable = '1') and (Temp1=one) then
      CORR_FACTOR   <= mul(S(0),x0a);
```

```
    else
      CORR_FACTOR <= zero;
    end if;
  end if;
end process;

x0a_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x0a <= zero;
    elsif (InitChien_d1 = '1') then
      x0a <= x0a_initial_value;
    else
      x0a <= mul(x0a,x0a_mul);
    end if;
  end if;
end process;

Delay_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to SR_max loop
        shift_register_delay(i) <= zero;
      end loop;
    else
      for i in SR_max downto 1 loop
        shift_register_delay(i) <= shift_register_delay(i-1);
      end loop;
      shift_register_delay(0) <= RS_Data_In;
    end if;
  end if;
end process;
RS_data_delayed <= shift_register_delay(SR_max);

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif (OutputEnable = '1') then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    else
      RS_Data_out <= zero;
    end if;
  end if;
end process;

inv_s0_process : process(S)
begin
  inv_s0 <= inv(S(0));
end process;

Calc_Temp_Registers : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1 <= zero;
    elsif (StartChien = '1') then
      Temp1 <= Temp1_initial_value;
    elsif (InitChien = '1') then
      Temp1 <= mul(S(1),Temp1);
    elsif (InitChien_d1 = '1') then
      Temp1 <= mul(inv_s0,Temp1);
    elsif (DoChien = '1') then
      Temp1 <= mul(alpha1,Temp1);
    end if;
  end if;
end process;
```

```
end;

---------------------------------------
-- End of ChienSearchProcess
---------------------------------------


---------------------------------------
-- Start of RSTopProcess
---------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity rs_dec_top is
  generic (
           GFPower : INTEGER := 6;
           RS_T : INTEGER := 1
           );
  port (
        clk : in std_logic;
        reset_n : in std_logic;
        RS_Data_In : in std_logic_vector(GFPower - 1 downto 0 );
        rs_data_in_start : in std_logic;
        RS_Data_out : out std_logic_vector(GFPower - 1 downto 0 );
        RS_Data_out_Start : out std_logic;
        rs_enable : in std_logic
        );


end rs_dec_top;

use work.all;
architecture RTL of rs_dec_top is

  signal errors_present : std_logic;
  signal syndrome : std_logic_vector(2 * GFPower * RS_T - 1 downto 0 );
  signal syndrome_calc_done : std_logic;
  component InputProcess
      port (
            reset_n : in std_logic;
            clk : in std_logic;
            rs_data_in : in std_logic_vector(5 downto 0 );
            rs_data_in_start : in std_logic;
            syndrome_calc_done : out std_logic;
            errors_present : out std_logic;
            syndrome : out std_logic_vector(11 downto 0 )
            );
  end component;
  component ChienSearchProcess
      port (
            clk : in std_logic;
            reset_n : in std_logic;
            syndromes : in std_logic_vector(11 downto 0 );
            StartChien : in std_logic;
            rs_enable : in std_logic;
            RS_Data_In : in std_logic_vector(5 downto 0 );
            RS_Data_out : out std_logic_vector(5 downto 0 );
            RS_Data_out_Start : out std_logic
            );
  end component;


begin

  inst_InputProcess: InputProcess
    port map (
              reset_n => reset_n,
              clk => clk,
              rs_data_in => RS_Data_In(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start,
              syndrome_calc_done => syndrome_calc_done,
```

```
            errors_present => errors_present,
            syndrome => syndrome(2 * GFPower * RS_T - 1 downto 0));

  inst_ChienSearchProcess: ChienSearchProcess
    port map (
            clk => clk,
            reset_n => reset_n,
            syndromes => syndrome(2 * GFPower * RS_T - 1 downto 0),
            StartChien => syndrome_calc_done,
            rs_enable => rs_enable,
            RS_Data_In => RS_Data_In(GFPower - 1 downto 0),
            RS_Data_out => RS_Data_out(GFPower - 1 downto 0),
            RS_Data_out_Start => RS_Data_out_Start);

end RTL;


----------------------------------------
-- End of RSTopProcess
----------------------------------------
```

15 Appendix C - RS Decoder VHDL Code (2 errors)

This appendix contains the VHDL code generated for an RS decoder for a 2-error correcting code over GF(2^5), with $p(x) = x^5 + x^2 + 1$. The value of N is 23 and the value of the log of the initial root of the code generator polynomial is 17. This is a RS(23,19) code.

```
----------------------------------------
-- RS Decoder Core Generator
-- Version 1.0
-- written by Vladimir Glavac 4182200
--
-- Reed-Solomon Decoder Parameters
--
-- RS_N  =  23 = length of codeword
--
-- RS_T  =   2 = error correcting capability
--
-- RS_M0 =  17 = power of initial root of code generator polynomial
--
-- code generator polynomial g(x) =
--
--      a0 * x^4
--   +  a9 * x^3
--   + a20 * x^2
--   + a15 * x^1
--   + a12
--
--
-- field generator polynomial p(x) = [100101] = x^5 + x^2 + 1
--
----------------------------------------

----------------------------------------
-- Start of InputProcess
----------------------------------------


library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  InputProcess  is
port (
      reset_n : in std_logic ;
      clk : in std_logic ;
      rs_data_in : in std_logic_vector (4 downto 0);
      rs_data_in_start : in std_logic ;
      syndrome_calc_done : out std_logic ;
      errors_present : out std_logic ;
      syndrome : out std_logic_vector (19 downto 0)
);
end;

architecture  RTL  of  InputProcess  is
constant GFPower : integer := 5;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 2;
constant RS_N : integer := 23;
constant RS_M0 : integer := 17;
constant Two_T_minus_1 : integer := 2*RS_T-1;
constant     zero : Galois_Field_element := "00000";
constant      one : Galois_Field_element := "00001";
constant  alpha17 : Galois_Field_element := "10011";
constant  alpha18 : Galois_Field_element := "00011";
constant  alpha19 : Galois_Field_element := "00110";
constant  alpha20 : Galois_Field_element := "01100";
type IntS_type is array(0 to Two_T_minus_1) of Galois_Field_element;
type IntEP_type is array(0 to Two_T_minus_1) of std_logic;
type state_type is (Idle, RxData, XferData);
signal present_state : state_type;
signal IntCount : std_logic_vector (4 downto 0);
signal internal_errors_present : std_logic;
signal CountEnable : std_logic;
signal CountReset : std_logic;
signal StartXfer : std_logic;
signal XferSyndrome : std_logic;
signal DoCalc : std_logic;
signal IntS : IntS_type;
```

```
signal IntEP : IntEP_type;

function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  return d;
end add;

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(3) and c(2)) xor
          (b(4) and c(1)) xor (b(4) and c(4));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(4) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(4)) xor (b(2) and c(0)) xor
          (b(2) and c(3)) xor (b(3) and c(2)) xor (b(3) and c(4)) xor (b(4) and c(1)) xor
          (b(4) and c(3)) xor (b(4) and c(4));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor (b(2) and c(4)) xor
          (b(3) and c(0)) xor (b(3) and c(3)) xor (b(4) and c(2)) xor (b(4) and c(4));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor (b(3) and c(1)) xor
          (b(3) and c(4)) xor (b(4) and c(0)) xor (b(4) and c(3));
  return d;
end mul;

begin

process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (CountReset = '1')) then
      IntCount <= (others=>'0');
    elsif (CountEnable = '1') then
      IntCount <= IntCount + 1;
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      StartXfer <= '0';
    elsif (IntCount="10100") then -- RS(23,19) : max_count = 20
      StartXfer <= '1';
    else
      StartXfer <= '0';
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= zero;
      end loop;
    elsif (rs_data_in_start = '1') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= rs_data_in;
      end loop;
    elsif (DoCalc='1') then
        IntS(0) <= add(mul(alpha17,IntS(0)),rs_data_in);
        IntS(1) <= add(mul(alpha18,IntS(1)),rs_data_in);
        IntS(2) <= add(mul(alpha19,IntS(2)),rs_data_in);
```

```
        IntS(3) <= add(mul(alpha20,IntS(3)),rs_data_in);
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      syndrome <= (others=>'0');
      internal_errors_present <= '0';
    elsif (XferSyndrome='1') then
      syndrome <=
        IntS(3) & IntS(2) & IntS(1) & IntS(0) ;
      internal_errors_present <=
        IntEP(0) or IntEP(1) or IntEP(2) or IntEP(3) ;
    end if;
  end if;
end process;

process (IntS)
begin
  for i in 0 to Two_T_minus_1 loop
    IntEP(i)<='1';
    if (IntS(i)=zero) then IntEP(i)<= '0'; end if;
  end loop;
end process;

InputControlSD_Idle : process (clk)
begin
if (clk'event and clk = '1') then
  if (reset_n = '0') then
    XferSyndrome<='0';
    DoCalc<='0';
    CountReset<='1';
    CountEnable<='0';
    present_state <= Idle;
  else
    case present_state is
      when Idle =>
        if (rs_data_in_start = '1') then
          DoCalc<='1';
          CountReset<='0';
          CountEnable<='1';
          present_state <= RxData;
        else
          present_state <= Idle;
        end if;
      when RxData =>
        if (StartXfer = '1') then
          XferSyndrome<='1';
          DoCalc<='0';
          CountReset<='1';
          CountEnable<='0';
          present_state <= XferData;
        else
          present_state <= RxData;
        end if;
      when XferData =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
      when others =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
    end case;
```

```
  end if;
end if;
end process;

syndrome_calc_done <= XferSyndrome;
errors_present <= internal_errors_present;

end;

----------------------------------------
-- End of InputProcess
----------------------------------------


----------------------------------------
-- Start of ChienSearchProcess
----------------------------------------

library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity  ChienSearchProcess  is
port (
      clk : in std_logic ;
      reset_n : in std_logic ;
      syndromes : in std_logic_vector (19 downto 0);
      StartChien : in std_logic ;
      rs_enable : in std_logic ;
      RS_Data_In : in std_logic_vector (4 downto 0);
      RS_Data_out : out std_logic_vector (4 downto 0);
      RS_Data_out_Start : out std_logic
  );
end;

architecture  RTL  of  ChienSearchProcess  is
constant GFPower : integer := 5;
constant RS_M0    : integer := 17;
constant RS_N     : integer := 23;
constant RS_T     : integer := 2;
constant ChienSearch_pipeline_delay : integer := 7;
constant SR_max : integer    := RS_N + ChienSearch_pipeline_delay;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
type syndrome_type is array(0 to 2*RS_T-1) of Galois_Field_element;
constant CountMax                : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant DoChienCountMax         : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-3, GFPower+1) ;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant OutputEnableStopCount   : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant Temp1_mul_factor    : Galois_Field_element := "11010"; -- alpha(-(RS_N-1)*1) =
a9
constant Temp2_mul_factor    : Galois_Field_element := "00011"; -- alpha(-(RS_N-1)*2) =
a18
constant alpha1              : Galois_Field_element := "00010";
constant alpha2              : Galois_Field_element := "00100";
constant one                 : Galois_Field_element := "00001";
constant x0a_initial_value   : Galois_Field_element := "01001"; -- alpha(-(RS_N-1)*RS_M0)
=  a29
constant x0a_mul             : Galois_Field_element := "10011"; -- alpha(RS_M0) =  a17
constant x1_initial_value    : Galois_Field_element := "10101"; -- alpha(RS_N-1) =  a22
constant x1_mul              : Galois_Field_element := "10010"; -- alpha(-1) =  a30
constant x1m_initial_value   : Galois_Field_element := "00100"; -- alpha((RS_N-1)*RS_M0) =
a2
constant x1m_mul             : Galois_Field_element := "11101"; -- alpha(-RS_M0) =  a14
constant zero                : Galois_Field_element := "00000";
```

```
signal CORR_FACTOR          : Galois_Field_element;
signal CORR_FACTOR_1_error  : Galois_Field_element;
signal CORR_FACTOR_2_errors : Galois_Field_element;
signal ChienSum             : Galois_Field_element;
signal Count                : std_logic_vector(GFPower downto 0);
signal CountEnable          : std_logic;
signal D1                   : Galois_Field_element;
signal D2                   : Galois_Field_element;
signal DoChien              : std_logic;
signal InitChien            : std_logic;
signal InitChien_d1         : std_logic;
signal InitChien_d2         : std_logic;
signal InitChien_d3         : std_logic;
signal InitChien_d4         : std_logic;
signal InitChien_d5         : std_logic;
signal InitChien_d6         : std_logic;
signal OutputEnable         : std_logic;
signal RS_data_delayed      : Galois_Field_element;
signal Temp1                : Galois_Field_element;
signal Temp1_num_1_error    : Galois_Field_element;
signal Temp1_num_2_errors   : Galois_Field_element;
signal Temp2                : Galois_Field_element;
signal Temp2_num_2_errors   : Galois_Field_element;
signal denom                : Galois_Field_element;
signal inv_D2               : Galois_Field_element;
signal inv_denom            : Galois_Field_element;
signal inv_s0               : Galois_Field_element;
signal num                  : Galois_Field_element;
signal num_d1               : Galois_Field_element;
signal shift_register_delay : shift_register_delay_type;
signal sigma1               : Galois_Field_element;
signal syndrome             : syndrome_type;
signal there_are_two_errors : std_logic;
signal there_is_one_error   : std_logic;
signal x0a                  : Galois_Field_element;
signal x1                   : Galois_Field_element;
signal x1m                  : Galois_Field_element;

function inv (b : in std_logic_vector) return std_logic_vector is
  variable d : std_logic_vector (4 downto 0);
begin
  d := "00000";
     if (b="00001") then d := "00001";
  elsif (b="00010") then d := "10010";
  elsif (b="00011") then d := "11100";
  elsif (b="00100") then d := "01001";
  elsif (b="00101") then d := "10111";
  elsif (b="00110") then d := "01110";
  elsif (b="00111") then d := "01100";
  elsif (b="01000") then d := "10110";
  elsif (b="01001") then d := "00100";
  elsif (b="01010") then d := "11001";
  elsif (b="01011") then d := "10000";
  elsif (b="01100") then d := "00111";
  elsif (b="01101") then d := "01111";
  elsif (b="01110") then d := "00110";
  elsif (b="01111") then d := "01101";
  elsif (b="10000") then d := "01011";
  elsif (b="10001") then d := "11000";
  elsif (b="10010") then d := "00010";
  elsif (b="10011") then d := "11101";
  elsif (b="10100") then d := "11110";
  elsif (b="10101") then d := "11010";
  elsif (b="10110") then d := "01000";
  elsif (b="10111") then d := "00101";
  elsif (b="11000") then d := "10001";
  elsif (b="11001") then d := "01010";
  elsif (b="11010") then d := "10101";
  elsif (b="11011") then d := "11111";
  elsif (b="11100") then d := "00011";
  elsif (b="11101") then d := "10011";
```

```
  elsif (b="11110") then d := "10100";
  elsif (b="11111") then d := "11011";
  end if;
  return d;
end inv;


function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  return d;
end add;


function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(3) and c(2)) xor
          (b(4) and c(1)) xor (b(4) and c(4));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(4) and c(2));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(4)) xor (b(2) and c(0)) xor
          (b(2) and c(3)) xor (b(3) and c(2)) xor (b(3) and c(4)) xor (b(4) and c(1)) xor
          (b(4) and c(3)) xor (b(4) and c(4));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(2) and c(1)) xor (b(2) and c(4)) xor
          (b(3) and c(0)) xor (b(3) and c(3)) xor (b(4) and c(2)) xor (b(4) and c(4));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(2) and c(2)) xor (b(3) and c(1)) xor
          (b(3) and c(4)) xor (b(4) and c(0)) xor (b(4) and c(3));
  return d;
end mul;


function IsNotZero(b  : in std_logic_vector) return std_logic is
        variable d : std_logic;
begin
 d :=
      b(0) or
      b(1) or
      b(2) or
      b(3) or
      b(4) ;
 return d;
end IsNotZero;

begin

syndrome(3) <= syndromes( 19 downto  15);
syndrome(2) <= syndromes( 14 downto  10);
syndrome(1) <= syndromes(  9 downto   5);
syndrome(0) <= syndromes(  4 downto   0);

System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;

System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
```

```
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;

InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
      InitChien_d1 <= '0';
      InitChien_d2 <= '0';
      InitChien_d3 <= '0';
      InitChien_d4 <= '0';
      InitChien_d5 <= '0';
      InitChien_d6 <= '0';
    else
      InitChien <= StartChien;
      InitChien_d1 <= InitChien;
      InitChien_d2 <= InitChien_d1;
      InitChien_d3 <= InitChien_d2;
      InitChien_d4 <= InitChien_d3;
      InitChien_d5 <= InitChien_d4;
      InitChien_d6 <= InitChien_d5;
    end if;
  end if;
end process;

OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

RS_Data_out_Start_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien_d4 = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;

Calc_Correction_Factor : process (clk)
begin
```

```
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= zero;
    elsif (rs_enable = '1') and (ChienSum=zero) then
      if (there_is_one_error='1') then
        CORR_FACTOR   <= CORR_FACTOR_1_error;
      elsif (there_are_two_errors='1') then
        CORR_FACTOR   <= CORR_FACTOR_2_errors;
      end if;
    else
      CORR_FACTOR <= zero;
    end if;
  end if;
end process;

D1 <= syndromes(4 downto  0);

Calculate_D2 : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      D2 <= zero;
    elsif (InitChien = '1') then
      D2 <= add(mul(syndrome(0),syndrome(2)),mul(syndrome(1),syndrome(1)));
    end if;
  end if;
end process;

Calculate_inv_D2 : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      inv_D2 <= zero;
    elsif (InitChien_d1 = '1') then
      inv_D2 <= inv(D2);
    end if;
  end if;
end process;

Error_Count_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      there_is_one_error   <= '0';
      there_are_two_errors <= '0';
    elsif ((InitChien = '1') and (IsNotZero(D1)='1')) then
      there_is_one_error   <= '1';
      there_are_two_errors <= '0';
    elsif ((InitChien_d1 = '1') and (IsNotZero(D2)='1')) then
      there_is_one_error   <= '0';
      there_are_two_errors <= '1';
    end if;
  end if;
end process;

x0a_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x0a <= zero;
    elsif (InitChien_d5 = '1') then
      x0a <= mul(x0a_initial_value,syndrome(0));
    else
      x0a <= mul(x0a,x0a_mul);
    end if;
  end if;
end process;
CORR_FACTOR_1_error <= x0a;

x1_process : process (clk)
begin
```

```
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x1 <= zero;
    elsif (InitChien_d2 = '1') then
      x1 <= x1_initial_value;
    else
      x1 <= mul(x1,x1_mul);
    end if;
  end if;
end process;

x1m_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      x1m <= zero;
    elsif (InitChien_d2 = '1') then
      x1m <= x1m_initial_value;
    else
      x1m <= mul(x1m,x1m_mul);
    end if;
  end if;
end process;

CORR_FACTOR_2_errors_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      num       <= zero;
      num_d1    <= zero;
      denom     <= zero;
      inv_denom <= zero;
      CORR_FACTOR_2_errors <= zero;
    else
      num       <= add(mul(add(x1,sigma1),syndrome(0)),syndrome(1));
      num_d1    <= num;
      denom     <= mul(x1m,sigma1);
      inv_denom <= inv(denom);
      CORR_FACTOR_2_errors <= mul(num_d1,inv_denom);
    end if;
  end if;
end process;

Delay_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to SR_max loop
        shift_register_delay(i) <= zero;
      end loop;
    else
      for i in SR_max downto 1 loop
        shift_register_delay(i) <= shift_register_delay(i-1);
      end loop;
      shift_register_delay(0) <= RS_Data_In;
    end if;
  end if;
end process;
RS_data_delayed <= shift_register_delay(SR_max);

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif (OutputEnable = '1') then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    else
      RS_Data_out <= zero;
    end if;
  end if;
```

```
end process;

inv_s0_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      inv_s0 <= zero;
    elsif (InitChien = '1') then
      inv_s0 <= inv(syndrome(0));
    end if;
  end if;
end process;

Temp1_num_1_error_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1_num_1_error <= zero;
    elsif (InitChien_d1 = '1') then
      Temp1_num_1_error <= mul(inv_s0,syndrome(1));
    end if;
  end if;
end process;

Temp1_num_2_errors_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1_num_2_errors <= zero;
    elsif (InitChien = '1') then
      Temp1_num_2_errors <=
add(mul(syndrome(0),syndrome(3)),mul(syndrome(1),syndrome(2)));
    end if;
  end if;
end process;

Temp2_num_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp2_num_2_errors <= zero;
    elsif (InitChien = '1') then
      Temp2_num_2_errors <=
add(mul(syndrome(1),syndrome(3)),mul(syndrome(2),syndrome(2)));
    elsif (InitChien_d2 = '1') then
      Temp2_num_2_errors <= mul(Temp2_num_2_errors,inv_d2);
    end if;
  end if;
end process;

Temp2_2_error_value_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      sigma1 <= zero;
    elsif (InitChien_d2 = '1') then
      sigma1 <= mul(Temp1_num_2_errors,inv_D2);
    end if;
  end if;
end process;

Calc_Temp_Registers_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') or (StartChien='1') then
      Temp1 <= zero;
      Temp2 <= zero;
    elsif (InitChien_d4 = '1') then
      if (IsNotZero(D2) = '1') then
        Temp1 <= mul(Temp1_mul_factor,sigma1);
        Temp2 <= mul(Temp2_mul_factor,Temp2_num_2_errors);
```

```
      elsif (IsNotZero(D1) = '1') then
        Temp1 <= mul(Temp1_mul_factor,Temp1_num_1_error);
        Temp2 <= zero;
      else
        Temp1 <= zero;
        Temp2 <= zero;
      end if;
    elsif (DoChien = '1') then
        Temp1 <= mul(alpha1,Temp1);
        Temp2 <= mul(alpha2,Temp2);
      end if;
  end if;
end process;

ChienSum_process : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      ChienSum <= zero;
    elsif (DoChien = '1') then
      ChienSum <= add(one,add(Temp1,Temp2));
    end if;
  end if;
end process;

end;


----------------------------------------
-- End of ChienSearchProcess
----------------------------------------


----------------------------------------
-- Start of RSTopProcess
----------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity rs_dec_top is

  generic (
           GFPower : INTEGER := 5;
           RS_T : INTEGER := 2
           );

  port (
        clk : in std_logic;
        reset_n : in std_logic;
        RS_Data_In : in std_logic_vector(GFPower - 1 downto 0);
        rs_data_in_start : in std_logic;
        RS_Data_out : out std_logic_vector(GFPower - 1 downto 0);
        RS_Data_out_Start : out std_logic;
        rs_enable : in std_logic
        );

end rs_dec_top;


use work.all;
architecture RTL of rs_dec_top is

  signal errors_present : std_logic;
  signal syndrome : std_logic_vector(2 * GFPower * RS_T - 1 downto 0);
  signal syndrome_calc_done : std_logic;

  component InputProcess
      port (
            reset_n : in std_logic;
            clk : in std_logic;
            rs_data_in : in std_logic_vector(4 downto 0);
```

```
            rs_data_in_start : in std_logic;
            syndrome_calc_done : out std_logic;
            errors_present : out std_logic;
            syndrome : out std_logic_vector(19 downto 0)
            );
  end component;

  component ChienSearchProcess
      port (
            clk : in std_logic;
            reset_n : in std_logic;
            syndromes : in std_logic_vector(19 downto 0);
            StartChien : in std_logic;
            rs_enable : in std_logic;
            RS_Data_In : in std_logic_vector(4 downto 0);
            RS_Data_out : out std_logic_vector(4 downto 0);
            RS_Data_out_Start : out std_logic
            );
  end component;

begin

  inst_InputProcess: InputProcess
    port map (
              reset_n => reset_n,
              clk => clk,
              rs_data_in => RS_Data_In(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start,
              syndrome_calc_done => syndrome_calc_done,
              errors_present => errors_present,
              syndrome => syndrome(2 * GFPower * RS_T - 1 downto 0));

  inst_ChienSearchProcess: ChienSearchProcess
    port map (
              clk => clk,
              reset_n => reset_n,
              syndromes => syndrome(2 * GFPower * RS_T - 1 downto 0),
              StartChien => syndrome_calc_done,
              rs_enable => rs_enable,
              RS_Data_In => RS_Data_In(GFPower - 1 downto 0),
              RS_Data_out => RS_Data_out(GFPower - 1 downto 0),
              RS_Data_out_Start => RS_Data_out_Start);

end RTL;


---------------------------------------
-- End of RSTopProcess
---------------------------------------
```

16 Appendix D - RS Decoder VHDL Code (8 errors)

This appendix contains the VHDL code generated for an RS decoder for an 8-error correcting code over GF(2^8), with $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. This is an RS(179,163) code.

```
----------------------------------------
-- RS Decoder Core Generator
-- Version 1.0
-- written by Vladimir Glavac 4182200
--
-- Reed-Solomon Decoder Parameters
--
-- RS_N  = 179 = length of codeword
--
-- RS_T  =   8 = error correcting capability
--
-- RS_M0 = 212 = power of initial root of code generator polynomial
--
-- code generator polynomial g(x) =
--
--      a0 * x^16
--   + a77 * x^15
--   + a18 * x^14
--   + a233 * x^13
--   + a192 * x^12
--   + a142 * x^11
--   + a158 * x^10
--   + a30 * x^9
--   + a169 * x^8
--   + a214 * x^7
--   + a16 * x^6
--   + a184 * x^5
--   + a163 * x^4
--   + a133 * x^3
--   + a102 * x^2
--   + a90 * x^1
--   + a197
--
--
-- field generator polynomial p(x) = [100011101] = x^8 + x^4 + x^3 + x^2 + 1
--
----------------------------------------

----------------------------------------
-- Start of InputProcess
----------------------------------------


library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  InputProcess  is
port (
      reset_n : in std_logic ;
      clk : in std_logic ;
      rs_data_in : in std_logic_vector (7 downto 0);
      rs_data_in_start : in std_logic ;
      syndrome_calc_done : out std_logic ;
      errors_present : out std_logic ;
      syndrome : out std_logic_vector (127 downto 0)
);
end;

architecture  RTL  of  InputProcess  is
constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 8;
constant RS_N : integer := 179;
constant RS_M0 : integer := 212;
constant Two_T_minus_1 : integer := 2*RS_T-1;
constant     zero : Galois_Field_element := "00000000";
constant      one : Galois_Field_element := "00000001";
constant alpha212 : Galois_Field_element := "01111001";
constant alpha213 : Galois_Field_element := "11110010";
constant alpha214 : Galois_Field_element := "11111001";
constant alpha215 : Galois_Field_element := "11101111";
```

```
constant alpha216 : Galois_Field_element := "11000011";
constant alpha217 : Galois_Field_element := "10011011";
constant alpha218 : Galois_Field_element := "00101011";
constant alpha219 : Galois_Field_element := "01010110";
constant alpha220 : Galois_Field_element := "10101100";
constant alpha221 : Galois_Field_element := "01000101";
constant alpha222 : Galois_Field_element := "10001010";
constant alpha223 : Galois_Field_element := "00001001";
constant alpha224 : Galois_Field_element := "00010010";
constant alpha225 : Galois_Field_element := "00100100";
constant alpha226 : Galois_Field_element := "01001000";
constant alpha227 : Galois_Field_element := "10010000";
type IntS_type is array(0 to Two_T_minus_1) of Galois_Field_element;
type IntEP_type is array(0 to Two_T_minus_1) of std_logic;
type state_type is (Idle, RxData, XferData);
signal present_state : state_type;
signal IntCount : std_logic_vector (7 downto 0);
signal internal_errors_present : std_logic;
signal CountEnable : std_logic;
signal CountReset : std_logic;
signal StartXfer : std_logic;
signal XferSyndrome : std_logic;
signal DoCalc : std_logic;
signal IntS : IntS_type;
signal IntEP : IntEP_type;

function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  d(6) := (b(6) xor c(6));
  d(7) := (b(7) xor c(7));
  return d;
end add;

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(7)) xor (b(2) and c(6)) xor (b(3) and c(5)) xor
          (b(4) and c(4)) xor (b(5) and c(3)) xor (b(5) and c(7)) xor (b(6) and c(2)) xor
          (b(6) and c(6)) xor (b(6) and c(7)) xor (b(7) and c(1)) xor (b(7) and c(5)) xor
          (b(7) and c(6)) xor (b(7) and c(7));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(7)) xor (b(3) and c(6)) xor
          (b(4) and c(5)) xor (b(5) and c(4)) xor (b(6) and c(3)) xor (b(6) and c(7)) xor
          (b(7) and c(2)) xor (b(7) and c(6)) xor (b(7) and c(7));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(7)) xor (b(2) and c(0)) xor
          (b(2) and c(6)) xor (b(3) and c(5)) xor (b(3) and c(7)) xor (b(4) and c(4)) xor
          (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(5)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(4)) xor (b(6) and c(6)) xor (b(6) and c(7)) xor
          (b(7) and c(1)) xor (b(7) and c(3)) xor (b(7) and c(5)) xor (b(7) and c(6));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(1) and c(7)) xor (b(2) and c(1)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(0)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(4) and c(7)) xor
          (b(5) and c(3)) xor (b(5) and c(4)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(1) and c(7)) xor (b(2) and c(2)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(1)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(0)) xor (b(4) and c(4)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(4)) xor
          (b(5) and c(5)) xor (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(4)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor (b(7) and c(7));
  d(5) := (b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(2) and c(7)) xor
          (b(3) and c(2)) xor (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(1)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor (b(5) and c(0)) xor
          (b(5) and c(4)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(6) and c(3)) xor
```

```
          (b(6) and c(4)) xor (b(6) and c(5)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor
          (b(7) and c(4));
  d(6) := (b(0) and c(6)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(3) and c(7)) xor (b(4) and c(2)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor
          (b(5) and c(1)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(0)) xor (b(6) and c(4)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(3)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(7) := (b(0) and c(7)) xor (b(1) and c(6)) xor (b(2) and c(5)) xor (b(3) and c(4)) xor
          (b(4) and c(3)) xor (b(4) and c(7)) xor (b(5) and c(2)) xor (b(5) and c(6)) xor
          (b(5) and c(7)) xor (b(6) and c(1)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(6) and c(7)) xor (b(7) and c(0)) xor (b(7) and c(4)) xor (b(7) and c(5)) xor
          (b(7) and c(6));
  return d;
end mul;

begin

process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (CountReset = '1')) then
      IntCount <= (others=>'0');
    elsif (CountEnable = '1') then
      IntCount <= IntCount + 1;
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      StartXfer <= '0';
    elsif (IntCount="10110000") then -- RS(179,163) : max_count = 176
      StartXfer <= '1';
    else
      StartXfer <= '0';
    end if;
  end if;
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= zero;
      end loop;
    elsif (rs_data_in_start = '1') then
      for i in 0 to Two_T_minus_1 loop
        IntS(i) <= rs_data_in;
      end loop;
    elsif (DoCalc='1') then
        IntS(0) <= add(mul(alpha212,IntS(0)),rs_data_in);
        IntS(1) <= add(mul(alpha213,IntS(1)),rs_data_in);
        IntS(2) <= add(mul(alpha214,IntS(2)),rs_data_in);
        IntS(3) <= add(mul(alpha215,IntS(3)),rs_data_in);
        IntS(4) <= add(mul(alpha216,IntS(4)),rs_data_in);
        IntS(5) <= add(mul(alpha217,IntS(5)),rs_data_in);
        IntS(6) <= add(mul(alpha218,IntS(6)),rs_data_in);
        IntS(7) <= add(mul(alpha219,IntS(7)),rs_data_in);
        IntS(8) <= add(mul(alpha220,IntS(8)),rs_data_in);
        IntS(9) <= add(mul(alpha221,IntS(9)),rs_data_in);
        IntS(10) <= add(mul(alpha222,IntS(10)),rs_data_in);
        IntS(11) <= add(mul(alpha223,IntS(11)),rs_data_in);
        IntS(12) <= add(mul(alpha224,IntS(12)),rs_data_in);
        IntS(13) <= add(mul(alpha225,IntS(13)),rs_data_in);
        IntS(14) <= add(mul(alpha226,IntS(14)),rs_data_in);
        IntS(15) <= add(mul(alpha227,IntS(15)),rs_data_in);
    end if;
  end if;
```

```
end process;

process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n = '0') then
      syndrome <= (others=>'0');
      internal_errors_present <= '0';
    elsif (XferSyndrome='1') then
      syndrome <=
        IntS(15) & IntS(14) & IntS(13) & IntS(12) & IntS(11) & IntS(10) &
        IntS(9) & IntS(8) & IntS(7) & IntS(6) & IntS(5) & IntS(4) &
        IntS(3) & IntS(2) & IntS(1) & IntS(0) ;
      internal_errors_present <=
        IntEP(0) or IntEP(1) or IntEP(2) or IntEP(3) or IntEP(4) or IntEP(5) or
        IntEP(6) or IntEP(7) or IntEP(8) or IntEP(9) or IntEP(10) or IntEP(11) or
        IntEP(12) or IntEP(13) or IntEP(14) or IntEP(15) ;
    end if;
  end if;
end process;

process (IntS)
begin
  for i in 0 to Two_T_minus_1 loop
    IntEP(i)<='1';
    if (IntS(i)=zero) then IntEP(i)<= '0'; end if;
  end loop;
end process;

InputControlSD_Idle : process (clk)
begin
if (clk'event and clk = '1') then
  if (reset_n = '0') then
    XferSyndrome<='0';
    DoCalc<='0';
    CountReset<='1';
    CountEnable<='0';
    present_state <= Idle;
  else
    case present_state is
      when Idle =>
        if (rs_data_in_start = '1') then
          DoCalc<='1';
          CountReset<='0';
          CountEnable<='1';
          present_state <= RxData;
        else
          present_state <= Idle;
        end if;
      when RxData =>
        if (StartXfer = '1') then
          XferSyndrome<='1';
          DoCalc<='0';
          CountReset<='1';
          CountEnable<='0';
          present_state <= XferData;
        else
          present_state <= RxData;
        end if;
      when XferData =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
      when others =>
        XferSyndrome<='0';
        DoCalc<='0';
        CountReset<='1';
        CountEnable<='0';
        present_state <= Idle;
```

```
    end case;
  end if;
end if;
end process;

syndrome_calc_done <= XferSyndrome;
errors_present <= internal_errors_present;

end;

----------------------------------------
-- End of InputProcess
----------------------------------------


----------------------------------------
-- Start of MBSolverProcess
----------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity  MBSolver  is
port (reset_n : in std_logic ;
      clk : in std_logic;
      ErrorsPresent : in std_logic ;
      XferSyndrome : in std_logic ;
      StartChien : out std_logic ;
      syndrome_poly : in std_logic_vector (127 downto 0);
      omega_poly : out std_logic_vector (63 downto 0);
      lambda_poly : out std_logic_vector (71 downto 0)
);
end;

architecture  RTL  of  MBSolver  is

constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T : integer := 8;
constant RS_N : integer := 179;
constant RS_M0 : integer := 212;
constant    zero : Galois_Field_element := "00000000";
constant     one : Galois_Field_element := "00000001";
constant size_of_N : integer := 5;

type SRegType is array(0 to 3*RS_T-1) of Galois_Field_element;
type Poly1Type is array(0 to RS_T) of Galois_Field_element;
type Poly2Type is array(0 to 2*RS_T) of Galois_Field_element;
type Poly3Type is array(1 to RS_T) of Galois_Field_element;
type state_type is (Idle,ChienSearchStart,Init,Synchronize,Update_Polys);

signal current_state : state_type;
signal Initialize : std_logic;
signal CountEnable : std_logic;
signal StoreNewPolys : std_logic;
signal N : std_logic_vector((size_of_N-1) downto 0);
signal SReg: SRegType;
signal Delta : Galois_Field_element;
signal Convolution_Term : Poly1Type;
signal Post_Convolution_Term : Poly1Type;
signal Gamma : Galois_Field_element;
signal L : std_logic_vector((size_of_N-1) downto 0);
signal TwoL : std_logic_vector((size_of_N-1) downto 0);
signal Lambda : Poly1Type;
signal ZOmega  : Poly1Type;
signal Mu, GammaMu : Poly1Type;
signal DeltaLambda : Poly3Type;
signal B    : Poly1Type;
signal KeepOldL : std_logic;
signal Convolution_Term_Multipler : std_logic_vector(RS_T downto 0);
```

```
function convolution_term_mul (b : in Galois_Field_element; c : in std_logic) return
Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0):=b(0) and c;
  d(1):=b(1) and c;
  d(2):=b(2) and c;
  d(3):=b(3) and c;
  d(4):=b(4) and c;
  d(5):=b(5) and c;
  d(6):=b(6) and c;
  d(7):=b(7) and c;
  return d;
end convolution_term_mul;

function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  d(6) := (b(6) xor c(6));
  d(7) := (b(7) xor c(7));
  return d;
end add;

function is_not_0 (b : in Galois_Field_element) return std_logic is
  variable d : std_logic;
begin
  d := b(0) or b(1) or b(2) or b(3) or b(4) or b(5) or b(6) or b(7) ;
  return d;
end is_not_0;

function is_0 (b : in Galois_Field_element) return std_logic is
  variable d : std_logic;
begin
  d := not is_not_0(b);
  return d;
end is_0;

function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(7)) xor (b(2) and c(6)) xor (b(3) and c(5)) xor
          (b(4) and c(4)) xor (b(5) and c(3)) xor (b(5) and c(7)) xor (b(6) and c(2)) xor
          (b(6) and c(6)) xor (b(6) and c(7)) xor (b(7) and c(1)) xor (b(7) and c(5)) xor
          (b(7) and c(6)) xor (b(7) and c(7));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(7)) xor (b(3) and c(6)) xor
          (b(4) and c(5)) xor (b(5) and c(4)) xor (b(6) and c(3)) xor (b(6) and c(7)) xor
          (b(7) and c(2)) xor (b(7) and c(6)) xor (b(7) and c(7));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(7)) xor (b(2) and c(0)) xor
          (b(2) and c(6)) xor (b(3) and c(5)) xor (b(3) and c(7)) xor (b(4) and c(4)) xor
          (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(5)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(4)) xor (b(6) and c(6)) xor (b(6) and c(7)) xor
          (b(7) and c(1)) xor (b(7) and c(3)) xor (b(7) and c(5)) xor (b(7) and c(6));
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(1) and c(7)) xor (b(2) and c(1)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(0)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(4) and c(7)) xor
          (b(5) and c(3)) xor (b(5) and c(4)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(1) and c(7)) xor (b(2) and c(2)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(1)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(0)) xor (b(4) and c(4)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(4)) xor
          (b(5) and c(5)) xor (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(4)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor (b(7) and c(7));
```

```
  d(5) := (b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(2) and c(7)) xor
          (b(3) and c(2)) xor (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(1)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor (b(5) and c(0)) xor
          (b(5) and c(4)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(6) and c(3)) xor
          (b(6) and c(4)) xor (b(6) and c(5)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor
          (b(7) and c(4));
  d(6) := (b(0) and c(6)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(3) and c(7)) xor (b(4) and c(2)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor
          (b(5) and c(1)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(0)) xor (b(6) and c(4)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(3)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(7) := (b(0) and c(7)) xor (b(1) and c(6)) xor (b(2) and c(5)) xor (b(3) and c(4)) xor
          (b(4) and c(3)) xor (b(4) and c(7)) xor (b(5) and c(2)) xor (b(5) and c(6)) xor
          (b(5) and c(7)) xor (b(6) and c(1)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(6) and c(7)) xor (b(7) and c(0)) xor (b(7) and c(4)) xor (b(7) and c(5)) xor
          (b(7) and c(6));
  return d;
end mul;

begin

SReg_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to (3*RS_T-1) loop
        SReg(i) <= zero;
      end loop;
    elsif (Initialize = '1') then
      for i in 0 to (RS_T-1) loop
        SReg(i) <= zero;
      end loop;
      for i in 0 to (2*RS_T-1) loop
        SReg(i+RS_T) <= syndrome_poly((((i+1)*GFPower)-1) downto (i*GFPower));
      end loop;
    elsif (StoreNewPolys = '1') then
      for i in 0 to (3*RS_T-2) loop
        SReg(i) <= SReg(i+1);
      end loop;
        SReg((3*RS_T-1)) <= zero;
    end if;
  end if;
end process SReg_Process;

L_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      L <= (others=>'0');
    elsif (Initialize = '1') then
      L <= (others=>'0');
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='0') then
        L <= N + 1 - L;
      end if;
    end if;
  end if;
end process L_Process;

TwoL_Process : process(L)
begin
  TwoL <= L((size_of_N-2) downto 0)&'0';
end process TwoL_Process;

KeepOldL_Process : process(TwoL, N, Delta)
begin
  if ((TwoL>N) or (is_0(Delta)='1')) then
    KeepOldL <= '1';
  else
    KeepOldL <= '0';
  end if;
```

```
end process KeepOldL_Process;

Gamma_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      Gamma <= (others=>'0');
    elsif (Initialize = '1') then
      Gamma <= one;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='0') then
        Gamma <= Delta;
      end if;
    end if;
  end if;
end process Gamma_Process;

N_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      N <= (others=>'0');
    elsif (Initialize = '1') then
      N <= (others=>'0');
    elsif (CountEnable = '1') then
      N <= N + 1;
    end if;
  end if;
end process N_Process;

Convolution_Term_Process : process (SReg, Mu)
begin
  for i in 0 to RS_T loop
    Convolution_Term(i) <= mul(SReg(RS_T-i),Mu(i));
  end loop;
end process Convolution_Term_Process;

Convolution_Term_Multipler_Process : process (L)
begin
  case L is
    when "00000" => Convolution_Term_Multipler <= "000000001";
    when "00001" => Convolution_Term_Multipler <= "000000011";
    when "00010" => Convolution_Term_Multipler <= "000000111";
    when "00011" => Convolution_Term_Multipler <= "000001111";
    when "00100" => Convolution_Term_Multipler <= "000011111";
    when "00101" => Convolution_Term_Multipler <= "000111111";
    when "00110" => Convolution_Term_Multipler <= "001111111";
    when "00111" => Convolution_Term_Multipler <= "011111111";
    when "01000" => Convolution_Term_Multipler <= "111111111";
    when others  => Convolution_Term_Multipler <= "111111111";
  end case;
end process Convolution_Term_Multipler_Process;

Post_Convolution_Term_Process : process (Convolution_Term_Multipler, Convolution_Term)
begin
  for i in 0 to RS_T loop
    Post_Convolution_Term(i) <=
convolution_term_mul(Convolution_Term(i),Convolution_Term_Multipler(i));
  end loop;
end process Post_Convolution_Term_Process;

Delta_Process : process (Post_Convolution_Term)
begin
   delta <= add(Post_Convolution_Term(0),
            add(Post_Convolution_Term(1),
            add(Post_Convolution_Term(2),
            add(Post_Convolution_Term(3),
            add(Post_Convolution_Term(4),
            add(Post_Convolution_Term(5),
            add(Post_Convolution_Term(6),
            add(Post_Convolution_Term(7), Post_Convolution_Term(8)))))))));
```

```
end process Delta_Process;

Lambda_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        Lambda(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      Lambda(0) <= one;
      for i in 1 to RS_T loop
        Lambda(i) <= zero;
      end loop;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='1') then
        for i in 1 to RS_T loop
          Lambda(i) <= Lambda(i-1);
        end loop;
        Lambda(0) <= zero;
      else
        for i in 0 to RS_T loop
          Lambda(i) <= Mu(i);
        end loop;
      end if;
    end if;
  end if;
end process Lambda_Process;

B_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        B(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      B(0) <= one;
      for i in 1 to RS_T loop
        B(i) <= zero;
      end loop;
    elsif (StoreNewPolys = '1') then
      if (KeepOldL='1') then
        for i in 1 to RS_T loop
          B(i) <= B(i-1);
        end loop;
        B(0) <= zero;
      else
        for i in 0 to RS_T loop
          B(i) <= ZOmega(i);
        end loop;
      end if;
    end if;
  end if;
end process B_Process;

ZOmega_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        ZOmega(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') then
      for i in 0 to RS_T loop
        ZOmega(i) <= (others=>'0');
      end loop;
    elsif (StoreNewPolys = '1') then
      for i in 1 to RS_T loop
        ZOmega(i) <= add(mul(Gamma,ZOmega(i)),mul(Delta,B(i-1)));
      end loop;
```

```
      ZOmega(0) <= mul(Gamma,ZOmega(0));
    end if;
  end if;
end process ZOmega_Process;

GammaMu_Process : process (Gamma, Mu)
begin
  for i in 0 to RS_T loop
    GammaMu(i) <= mul(Gamma, Mu(i));
  end loop;
end process GammaMu_Process;

DeltaLambda_Process : process (Delta, Lambda)
begin
  for i in 1 to RS_T loop
    DeltaLambda(i) <= mul(Delta, Lambda(i-1));
  end loop;
end process DeltaLambda_Process;

omega_poly  <=
                ZOmega(8) & ZOmega(7) & ZOmega(6) & ZOmega(5) & ZOmega(4) & ZOmega(3) &
                ZOmega(2) & ZOmega(1) ;
lambda_poly  <=
                Mu(8) & Mu(7) & Mu(6) & Mu(5) & Mu(4) & Mu(3) & Mu(2) & Mu(1) &
                Mu(0) ;

Mu_Process : process (clk)
begin
  if (clk'event and clk='1') then
    if (reset_n='0') then
      for i in 0 to RS_T loop
        Mu(i) <= (others=>'0');
      end loop;
    elsif (Initialize = '1') or (ErrorsPresent='0') then
      for i in 1 to RS_T loop
        Mu(i) <= (others=>'0');
      end loop;
      Mu(0) <= one;
    elsif (StoreNewPolys = '1') then
      for i in 1 to RS_T loop
        Mu(i) <= add(GammaMu(i),DeltaLambda(i));
      end loop;
      Mu(0) <= GammaMu(0);
    end if;
  end if;
end process Mu_Process;

process (clk)
begin

  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Initialize <= '0';
      StoreNewPolys <= '0';
      StartChien <= '0';
      CountEnable <= '0';
      current_state <= Idle;
    else

      case current_state is
        when Idle =>
          if (XferSyndrome = '1') then
            Initialize <= '1';
            StoreNewPolys <= '0';
            StartChien <= '0';
            CountEnable <= '0';
            current_state <= Init;
          else
            current_state <= Idle;
          end if;
```

```
        when ChienSearchStart =>
          Initialize <= '0';
          StoreNewPolys <= '0';
          StartChien <= '0';
          CountEnable <= '0';
          current_state <= Idle;

        when Init =>
          if (ErrorsPresent = '0') then
            Initialize <= '0';
            StoreNewPolys <= '0';
            StartChien <= '0';
            CountEnable <= '1';
            current_state <= Synchronize;
          else
            Initialize <= '0';
            StoreNewPolys <= '1';
            StartChien <= '0';
            CountEnable <= '1';
            current_state <= Update_Polys;
          end if;

        when Synchronize =>
          if (N = "01111") then
            Initialize <= '0';
            StoreNewPolys <= '0';
            StartChien <= '1';
            CountEnable <= '0';
            current_state <= ChienSearchStart;
          else
            current_state <= Synchronize;
          end if;

        when Update_Polys =>
          if (N = "01111") then
            Initialize <= '0';
            StoreNewPolys <= '0';
            StartChien <= '1';
            CountEnable <= '0';
            current_state <= ChienSearchStart;
          else
            current_state <= Update_Polys;
          end if;

        when others =>
          Initialize <= '0';
          StoreNewPolys <= '0';
          StartChien <= '0';
          CountEnable <= '0';
          current_state <= Idle;
      end case;
    end if;
  end if;
end process;

end;


----------------------------------------
-- End of MBSolverProcess
----------------------------------------

--
--
-- Chien search parameters
--
-- RS_N  = 179 ;  RS_T  = 8 ;  RS_M0 = 212
-- RS_T_is_odd           = FALSE
-- RS_T_is_even          = TRUE
-- K_mul_stage_is_needed = TRUE
-- K_VAL_is_needed       = TRUE
```

```
-- ChienSum_D1_needed     = FALSE
-- eval_om_d2_needed      = FALSE
-- eval_lp_d1_needed      = FALSE
--

----------------------------------------
-- Start of ChienSearchProcess
----------------------------------------

library ieee; use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity  ChienSearchProcess  is
port (
      clk : in std_logic ;
      reset_n : in std_logic ;
      lambda_poly : in std_logic_vector (71 downto 0);
      omega_poly  : in std_logic_vector (63 downto 0);
      StartChien : in std_logic ;
      rs_enable : in std_logic ;
      RS_Data_In : in std_logic_vector (7 downto 0);
      RS_Data_out : out std_logic_vector (7 downto 0);
      RS_Data_out_Start : out std_logic
  );
end;


architecture  RTL  of  ChienSearchProcess  is
constant GFPower : integer := 8;
subtype Galois_Field_element is std_logic_vector((GFPower-1) downto 0);
constant RS_T  : integer := 8;
constant RS_N  : integer := 179;
constant RS_M0 : integer := 212;
constant key_equation_solver_delay : integer := 2*RS_T+2;
constant ChienSearch_pipeline_delay : integer := RS_T+1;
constant SR_max : integer := RS_N + key_equation_solver_delay +
ChienSearch_pipeline_delay;
constant OutputEnableStartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay-1, GFPower+1) ;
constant RS_Data_out_StartCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(ChienSearch_pipeline_delay, GFPower+1) ;
constant OutputEnableStopCount : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant DoChienCountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay-1, GFPower+1) ;
constant CountMax : std_logic_vector(GFPower downto 0) :=
CONV_STD_LOGIC_VECTOR(RS_N+ChienSearch_pipeline_delay, GFPower+1) ;
constant     zero : Galois_Field_element := "00000000";
constant   alpha1 : Galois_Field_element := "00000010";
constant   alpha2 : Galois_Field_element := "00000100";
constant   alpha3 : Galois_Field_element := "00001000";
constant   alpha4 : Galois_Field_element := "00010000";
constant   alpha5 : Galois_Field_element := "00100000";
constant   alpha6 : Galois_Field_element := "01000000";
constant   alpha7 : Galois_Field_element := "10000000";
constant   alpha8 : Galois_Field_element := "00011101";
constant  alpha77 : Galois_Field_element := "00111100";
constant alpha152 : Galois_Field_element := "01001001";
constant alpha225 : Galois_Field_element := "00100100";
constant  alpha41 : Galois_Field_element := "11010100";
constant alpha110 : Galois_Field_element := "01100111";
constant alpha177 : Galois_Field_element := "11011011";
constant alpha242 : Galois_Field_element := "10110000";
constant  alpha50 : Galois_Field_element := "00000101";
type Poly1Type is array(0 to RS_T) of Galois_Field_element;
type Poly2Type is array(0 to RS_T-1) of Galois_Field_element;
type Poly3Type is array(1 to RS_T) of Galois_Field_element;
type shift_register_delay_type is array(0 to SR_max) of Galois_Field_element;
signal shift_register_delay : shift_register_delay_type;
signal RS_data_delayed : Galois_Field_element;
```

```
signal Lambda : Poly1Type;
signal Omega  : Poly2Type;
signal IntLambda_0 : Galois_Field_element;
signal IntLambda_1 : Galois_Field_element;
signal IntLambda_3 : Galois_Field_element;
signal IntLambda_5 : Galois_Field_element;
signal IntLambda_7 : Galois_Field_element;
signal IntOmega  : Poly2Type;
signal X1 : Galois_Field_element;
signal X2 : Galois_Field_element;
signal X3 : Galois_Field_element;
signal X4 : Galois_Field_element;
signal X5 : Galois_Field_element;
signal X6 : Galois_Field_element;
signal X7 : Galois_Field_element;
signal X1_D1 : Galois_Field_element;
signal X1_D2 : Galois_Field_element;
signal X1_D3 : Galois_Field_element;
signal X1_D4 : Galois_Field_element;
signal X1_D5 : Galois_Field_element;
signal EVAL_OM_1 : Galois_Field_element;
signal EVAL_OM_2 : Galois_Field_element;
signal EVAL_OM_3 : Galois_Field_element;
signal EVAL_OM_4 : Galois_Field_element;
signal EVAL_OM_5 : Galois_Field_element;
signal EVAL_OM_6 : Galois_Field_element;
signal EVAL_OM     : Galois_Field_element;
signal EVAL_OM_D1  : Galois_Field_element;
signal EVAL_LP_1 : Galois_Field_element;
signal EVAL_LP_2 : Galois_Field_element;
signal EVAL_LP_3 : Galois_Field_element;
signal EVAL_LP_4 : Galois_Field_element;
signal EVAL_LP   : Galois_Field_element;
signal EVAL_DEN  : Galois_Field_element;
signal DEN_INV   : Galois_Field_element;
signal CORR_FACTOR : Galois_Field_element;
signal Count : std_logic_vector(GFPower downto 0);
signal CountEnable : std_logic;
signal InitChien : std_logic;
signal DoChien : std_logic;
signal OutputEnable : std_logic;
signal Temp1 : Galois_Field_element;
signal Temp2 : Galois_Field_element;
signal Temp3 : Galois_Field_element;
signal Temp4 : Galois_Field_element;
signal Temp5 : Galois_Field_element;
signal Temp6 : Galois_Field_element;
signal Temp7 : Galois_Field_element;
signal Temp8 : Galois_Field_element;
signal ChienSum : Galois_Field_element;
signal EVAL_CS_1 : Galois_Field_element;
signal EVAL_CS_2 : Galois_Field_element;
signal EVAL_CS_3 : Galois_Field_element;
signal EVAL_CS_4 : Galois_Field_element;
signal EVAL_CS_5 : Galois_Field_element;
signal EVAL_CS_6 : Galois_Field_element;
signal EVAL_CS_7 : Galois_Field_element;
constant K_MUL : Galois_Field_element := "11101110"; -- alpha44
constant K_VAL_initial_value : Galois_Field_element := "10100001"; -- alpha63
signal K_VAL : Galois_Field_element;

function inv (b : in std_logic_vector) return std_logic_vector is
  variable d : std_logic_vector (7 downto 0);
begin
  d := "00000000";
     if (b="00000001") then d := "00000001";
  elsif (b="00000010") then d := "10001110";
  elsif (b="00000011") then d := "11110100";
  elsif (b="00000100") then d := "01000111";
  elsif (b="00000101") then d := "10100111";
  elsif (b="00000110") then d := "01111010";
```

```
elsif (b="00000111") then d := "10111010";
elsif (b="00001000") then d := "10101101";
elsif (b="00001001") then d := "10011101";
elsif (b="00001010") then d := "11011101";
elsif (b="00001011") then d := "10011000";
elsif (b="00001100") then d := "00111101";
elsif (b="00001101") then d := "10101010";
elsif (b="00001110") then d := "01011101";
elsif (b="00001111") then d := "10010110";
elsif (b="00010000") then d := "11011000";
elsif (b="00010001") then d := "01110010";
elsif (b="00010010") then d := "11000000";
elsif (b="00010011") then d := "01011000";
elsif (b="00010100") then d := "11100000";
elsif (b="00010101") then d := "00111110";
elsif (b="00010110") then d := "01001100";
elsif (b="00010111") then d := "01100110";
elsif (b="00011000") then d := "10010000";
elsif (b="00011001") then d := "11011110";
elsif (b="00011010") then d := "01010101";
elsif (b="00011011") then d := "10000000";
elsif (b="00011100") then d := "10100000";
elsif (b="00011101") then d := "10000011";
elsif (b="00011110") then d := "01001011";
elsif (b="00011111") then d := "00101010";
elsif (b="00100000") then d := "01101100";
elsif (b="00100001") then d := "11101101";
elsif (b="00100010") then d := "00111001";
elsif (b="00100011") then d := "01010001";
elsif (b="00100100") then d := "01100000";
elsif (b="00100101") then d := "01010110";
elsif (b="00100110") then d := "00101100";
elsif (b="00100111") then d := "10001010";
elsif (b="00101000") then d := "01110000";
elsif (b="00101001") then d := "11010000";
elsif (b="00101010") then d := "00011111";
elsif (b="00101011") then d := "01001010";
elsif (b="00101100") then d := "00100110";
elsif (b="00101101") then d := "10001011";
elsif (b="00101110") then d := "00110011";
elsif (b="00101111") then d := "01101110";
elsif (b="00110000") then d := "01001000";
elsif (b="00110001") then d := "10001001";
elsif (b="00110010") then d := "01101111";
elsif (b="00110011") then d := "00101110";
elsif (b="00110100") then d := "10100100";
elsif (b="00110101") then d := "11000011";
elsif (b="00110110") then d := "01000000";
elsif (b="00110111") then d := "01011110";
elsif (b="00111000") then d := "01010000";
elsif (b="00111001") then d := "00100010";
elsif (b="00111010") then d := "11001111";
elsif (b="00111011") then d := "10101001";
elsif (b="00111100") then d := "10101011";
elsif (b="00111101") then d := "00001100";
elsif (b="00111110") then d := "00010101";
elsif (b="00111111") then d := "11100001";
elsif (b="01000000") then d := "00110110";
elsif (b="01000001") then d := "01011111";
elsif (b="01000010") then d := "11111000";
elsif (b="01000011") then d := "11010101";
elsif (b="01000100") then d := "10010010";
elsif (b="01000101") then d := "01001110";
elsif (b="01000110") then d := "10100110";
elsif (b="01000111") then d := "00000100";
elsif (b="01001000") then d := "00110000";
elsif (b="01001001") then d := "10001000";
elsif (b="01001010") then d := "00101011";
elsif (b="01001011") then d := "00011110";
elsif (b="01001100") then d := "00010110";
elsif (b="01001101") then d := "01100111";
```

```
elsif (b="01001110") then d := "01000101";
elsif (b="01001111") then d := "10010011";
elsif (b="01010000") then d := "00111000";
elsif (b="01010001") then d := "00100011";
elsif (b="01010010") then d := "01101000";
elsif (b="01010011") then d := "10001100";
elsif (b="01010100") then d := "10000001";
elsif (b="01010101") then d := "00011010";
elsif (b="01010110") then d := "00100101";
elsif (b="01010111") then d := "01100001";
elsif (b="01011000") then d := "00010011";
elsif (b="01011001") then d := "11000001";
elsif (b="01011010") then d := "11001011";
elsif (b="01011011") then d := "01100011";
elsif (b="01011100") then d := "10010111";
elsif (b="01011101") then d := "00001110";
elsif (b="01011110") then d := "00110111";
elsif (b="01011111") then d := "01000001";
elsif (b="01100000") then d := "00100100";
elsif (b="01100001") then d := "01010111";
elsif (b="01100010") then d := "11001010";
elsif (b="01100011") then d := "01011011";
elsif (b="01100100") then d := "10111001";
elsif (b="01100101") then d := "11000100";
elsif (b="01100110") then d := "00010111";
elsif (b="01100111") then d := "01001101";
elsif (b="01101000") then d := "01010010";
elsif (b="01101001") then d := "10001101";
elsif (b="01101010") then d := "11101111";
elsif (b="01101011") then d := "10110011";
elsif (b="01101100") then d := "00100000";
elsif (b="01101101") then d := "11101100";
elsif (b="01101110") then d := "00101111";
elsif (b="01101111") then d := "00110010";
elsif (b="01110000") then d := "00101000";
elsif (b="01110001") then d := "11010001";
elsif (b="01110010") then d := "00010001";
elsif (b="01110011") then d := "11011001";
elsif (b="01110100") then d := "11101001";
elsif (b="01110101") then d := "11111011";
elsif (b="01110110") then d := "11011010";
elsif (b="01110111") then d := "01111001";
elsif (b="01111000") then d := "11011011";
elsif (b="01111001") then d := "01110111";
elsif (b="01111010") then d := "00000110";
elsif (b="01111011") then d := "10111011";
elsif (b="01111100") then d := "10000100";
elsif (b="01111101") then d := "11001101";
elsif (b="01111110") then d := "11111110";
elsif (b="01111111") then d := "11111100";
elsif (b="10000000") then d := "00011011";
elsif (b="10000001") then d := "01010100";
elsif (b="10000010") then d := "10100001";
elsif (b="10000011") then d := "00011101";
elsif (b="10000100") then d := "01111100";
elsif (b="10000101") then d := "11001100";
elsif (b="10000110") then d := "11100100";
elsif (b="10000111") then d := "10110000";
elsif (b="10001000") then d := "01001001";
elsif (b="10001001") then d := "00110001";
elsif (b="10001010") then d := "00100111";
elsif (b="10001011") then d := "00101101";
elsif (b="10001100") then d := "01010011";
elsif (b="10001101") then d := "01101001";
elsif (b="10001110") then d := "00000010";
elsif (b="10001111") then d := "11110101";
elsif (b="10010000") then d := "00011000";
elsif (b="10010001") then d := "11011111";
elsif (b="10010010") then d := "01000100";
elsif (b="10010011") then d := "01001111";
elsif (b="10010100") then d := "10011011";
```

```
elsif (b="10010101") then d := "10111100";
elsif (b="10010110") then d := "00001111";
elsif (b="10010111") then d := "01011100";
elsif (b="10011000") then d := "00001011";
elsif (b="10011001") then d := "11011100";
elsif (b="10011010") then d := "10111101";
elsif (b="10011011") then d := "10010100";
elsif (b="10011100") then d := "10101100";
elsif (b="10011101") then d := "00001001";
elsif (b="10011110") then d := "11000111";
elsif (b="10011111") then d := "10100010";
elsif (b="10100000") then d := "00011100";
elsif (b="10100001") then d := "10000010";
elsif (b="10100010") then d := "10011111";
elsif (b="10100011") then d := "11000110";
elsif (b="10100100") then d := "00110100";
elsif (b="10100101") then d := "11000010";
elsif (b="10100110") then d := "01000110";
elsif (b="10100111") then d := "00000101";
elsif (b="10101000") then d := "11001110";
elsif (b="10101001") then d := "00111011";
elsif (b="10101010") then d := "00001101";
elsif (b="10101011") then d := "00111100";
elsif (b="10101100") then d := "10011100";
elsif (b="10101101") then d := "00001000";
elsif (b="10101110") then d := "10111110";
elsif (b="10101111") then d := "10110111";
elsif (b="10110000") then d := "10000111";
elsif (b="10110001") then d := "11100101";
elsif (b="10110010") then d := "11101110";
elsif (b="10110011") then d := "01101011";
elsif (b="10110100") then d := "11101011";
elsif (b="10110101") then d := "11110010";
elsif (b="10110110") then d := "10111111";
elsif (b="10110111") then d := "10101111";
elsif (b="10111000") then d := "11000101";
elsif (b="10111001") then d := "01100100";
elsif (b="10111010") then d := "00000111";
elsif (b="10111011") then d := "01111011";
elsif (b="10111100") then d := "10010101";
elsif (b="10111101") then d := "10011010";
elsif (b="10111110") then d := "10101110";
elsif (b="10111111") then d := "10110110";
elsif (b="11000000") then d := "00010010";
elsif (b="11000001") then d := "01011001";
elsif (b="11000010") then d := "10100101";
elsif (b="11000011") then d := "00110101";
elsif (b="11000100") then d := "01100101";
elsif (b="11000101") then d := "10111000";
elsif (b="11000110") then d := "10100011";
elsif (b="11000111") then d := "10011110";
elsif (b="11001000") then d := "11010010";
elsif (b="11001001") then d := "11110111";
elsif (b="11001010") then d := "01100010";
elsif (b="11001011") then d := "01011010";
elsif (b="11001100") then d := "10000101";
elsif (b="11001101") then d := "01111101";
elsif (b="11001110") then d := "10101000";
elsif (b="11001111") then d := "00111010";
elsif (b="11010000") then d := "00101001";
elsif (b="11010001") then d := "01110001";
elsif (b="11010010") then d := "11001000";
elsif (b="11010011") then d := "11110110";
elsif (b="11010100") then d := "11111001";
elsif (b="11010101") then d := "01000011";
elsif (b="11010110") then d := "11010111";
elsif (b="11010111") then d := "11010110";
elsif (b="11011000") then d := "00010000";
elsif (b="11011001") then d := "01110011";
elsif (b="11011010") then d := "01110110";
elsif (b="11011011") then d := "01111000";
```

```
  elsif (b="11011100") then d := "10011001";
  elsif (b="11011101") then d := "00001010";
  elsif (b="11011110") then d := "00011001";
  elsif (b="11011111") then d := "10010001";
  elsif (b="11100000") then d := "00010100";
  elsif (b="11100001") then d := "00111111";
  elsif (b="11100010") then d := "11100110";
  elsif (b="11100011") then d := "11110000";
  elsif (b="11100100") then d := "10000110";
  elsif (b="11100101") then d := "10110001";
  elsif (b="11100110") then d := "11100010";
  elsif (b="11100111") then d := "11110001";
  elsif (b="11101000") then d := "11111010";
  elsif (b="11101001") then d := "01110100";
  elsif (b="11101010") then d := "11110011";
  elsif (b="11101011") then d := "10110100";
  elsif (b="11101100") then d := "01101101";
  elsif (b="11101101") then d := "00100001";
  elsif (b="11101110") then d := "10110010";
  elsif (b="11101111") then d := "01101010";
  elsif (b="11110000") then d := "11100011";
  elsif (b="11110001") then d := "11100111";
  elsif (b="11110010") then d := "10110101";
  elsif (b="11110011") then d := "11101010";
  elsif (b="11110100") then d := "00000011";
  elsif (b="11110101") then d := "10001111";
  elsif (b="11110110") then d := "11010011";
  elsif (b="11110111") then d := "11001001";
  elsif (b="11111000") then d := "01000010";
  elsif (b="11111001") then d := "11010100";
  elsif (b="11111010") then d := "11101000";
  elsif (b="11111011") then d := "01110101";
  elsif (b="11111100") then d := "01111111";
  elsif (b="11111101") then d := "11111111";
  elsif (b="11111110") then d := "01111110";
  elsif (b="11111111") then d := "11111101";
  end if;
  return d;
end inv;


function add (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) xor c(0));
  d(1) := (b(1) xor c(1));
  d(2) := (b(2) xor c(2));
  d(3) := (b(3) xor c(3));
  d(4) := (b(4) xor c(4));
  d(5) := (b(5) xor c(5));
  d(6) := (b(6) xor c(6));
  d(7) := (b(7) xor c(7));
  return d;
end add;


function mul (b , c : in Galois_Field_element) return Galois_Field_element is
  variable d : Galois_Field_element;
begin
  d(0) := (b(0) and c(0)) xor (b(1) and c(7)) xor (b(2) and c(6)) xor (b(3) and c(5)) xor
          (b(4) and c(4)) xor (b(5) and c(3)) xor (b(5) and c(7)) xor (b(6) and c(2)) xor
          (b(6) and c(6)) xor (b(6) and c(7)) xor (b(7) and c(1)) xor (b(7) and c(5)) xor
          (b(7) and c(6)) xor (b(7) and c(7));
  d(1) := (b(0) and c(1)) xor (b(1) and c(0)) xor (b(2) and c(7)) xor (b(3) and c(6)) xor
          (b(4) and c(5)) xor (b(5) and c(4)) xor (b(6) and c(3)) xor (b(6) and c(7)) xor
          (b(7) and c(2)) xor (b(7) and c(6)) xor (b(7) and c(7));
  d(2) := (b(0) and c(2)) xor (b(1) and c(1)) xor (b(1) and c(7)) xor (b(2) and c(0)) xor
          (b(2) and c(6)) xor (b(3) and c(5)) xor (b(3) and c(7)) xor (b(4) and c(4)) xor
          (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(5)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(4)) xor (b(6) and c(6)) xor (b(6) and c(7)) xor
          (b(7) and c(1)) xor (b(7) and c(3)) xor (b(7) and c(5)) xor (b(7) and c(6));
```

```
  d(3) := (b(0) and c(3)) xor (b(1) and c(2)) xor (b(1) and c(7)) xor (b(2) and c(1)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(0)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(4) and c(4)) xor (b(4) and c(5)) xor (b(4) and c(7)) xor
          (b(5) and c(3)) xor (b(5) and c(4)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(4) := (b(0) and c(4)) xor (b(1) and c(3)) xor (b(1) and c(7)) xor (b(2) and c(2)) xor
          (b(2) and c(6)) xor (b(2) and c(7)) xor (b(3) and c(1)) xor (b(3) and c(5)) xor
          (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(0)) xor (b(4) and c(4)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(5) and c(3)) xor (b(5) and c(4)) xor
          (b(5) and c(5)) xor (b(6) and c(2)) xor (b(6) and c(3)) xor (b(6) and c(4)) xor
          (b(7) and c(1)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor (b(7) and c(7));
  d(5) := (b(0) and c(5)) xor (b(1) and c(4)) xor (b(2) and c(3)) xor (b(2) and c(7)) xor
          (b(3) and c(2)) xor (b(3) and c(6)) xor (b(3) and c(7)) xor (b(4) and c(1)) xor
          (b(4) and c(5)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor (b(5) and c(0)) xor
          (b(5) and c(4)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(6) and c(3)) xor
          (b(6) and c(4)) xor (b(6) and c(5)) xor (b(7) and c(2)) xor (b(7) and c(3)) xor
          (b(7) and c(4));
  d(6) := (b(0) and c(6)) xor (b(1) and c(5)) xor (b(2) and c(4)) xor (b(3) and c(3)) xor
          (b(3) and c(7)) xor (b(4) and c(2)) xor (b(4) and c(6)) xor (b(4) and c(7)) xor
          (b(5) and c(1)) xor (b(5) and c(5)) xor (b(5) and c(6)) xor (b(5) and c(7)) xor
          (b(6) and c(0)) xor (b(6) and c(4)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(7) and c(3)) xor (b(7) and c(4)) xor (b(7) and c(5));
  d(7) := (b(0) and c(7)) xor (b(1) and c(6)) xor (b(2) and c(5)) xor (b(3) and c(4)) xor
          (b(4) and c(3)) xor (b(4) and c(7)) xor (b(5) and c(2)) xor (b(5) and c(6)) xor
          (b(5) and c(7)) xor (b(6) and c(1)) xor (b(6) and c(5)) xor (b(6) and c(6)) xor
          (b(6) and c(7)) xor (b(7) and c(0)) xor (b(7) and c(4)) xor (b(7) and c(5)) xor
          (b(7) and c(6));
  return d;
end mul;


function IsNotZero(b  : in std_logic_vector) return std_logic is
       variable d : std_logic;
begin
 d :=
      b(0) or
      b(1) or
      b(2) or
      b(3) or
      b(4) or
      b(5) or
      b(6) or
      b(7) ;
 return d;
end IsNotZero;

begin

Lambda(  0) <= lambda_poly (   7 downto    0);
Lambda(  1) <= lambda_poly (  15 downto    8);
Lambda(  2) <= lambda_poly (  23 downto   16);
Lambda(  3) <= lambda_poly (  31 downto   24);
Lambda(  4) <= lambda_poly (  39 downto   32);
Lambda(  5) <= lambda_poly (  47 downto   40);
Lambda(  6) <= lambda_poly (  55 downto   48);
Lambda(  7) <= lambda_poly (  63 downto   56);
Lambda(  8) <= lambda_poly (  71 downto   64);
 Omega(  0) <=  omega_poly (   7 downto    0);
 Omega(  1) <=  omega_poly (  15 downto    8);
 Omega(  2) <=  omega_poly (  23 downto   16);
 Omega(  3) <=  omega_poly (  31 downto   24);
 Omega(  4) <=  omega_poly (  39 downto   32);
 Omega(  5) <=  omega_poly (  47 downto   40);
 Omega(  6) <=  omega_poly (  55 downto   48);
 Omega(  7) <=  omega_poly (  63 downto   56);

Init_IntLambda_and_IntOmega : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
```

```
      IntLambda_0 <= zero;
      IntLambda_1 <= zero;
      IntLambda_3 <= zero;
      IntLambda_5 <= zero;
      IntLambda_7 <= zero;
      for i in 0 to RS_T-1 loop
        IntOmega(i) <= zero;
      end loop;
    elsif (StartChien = '1') then
      IntLambda_0 <= Lambda(0);
      IntLambda_1 <= Lambda(1);
      IntLambda_3 <= Lambda(3);
      IntLambda_5 <= Lambda(5);
      IntLambda_7 <= Lambda(7);
      for i in 0 to RS_T-1 loop
        IntOmega(i) <= Omega(i);
      end loop;
    end if;
  end if;
end process;

System_Counter : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      Count <= (others => '0');
    elsif (CountEnable = '1') then
      Count <= Count + 1;
    end if;
  end if;
end process;

System_Count_Enable : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CountEnable <= '0';
    elsif (StartChien = '1') then
      CountEnable <= '1';
    elsif (Count = CountMax) then
      CountEnable <= '0';
    end if;
  end if;
end process;

InitChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      InitChien <= '0';
    else
      InitChien <= StartChien;
    end if;
  end if;
end process;

OutputEnable_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      OutputEnable <= '0';
    elsif (Count = OutputEnableStartCount) then
      OutputEnable <= '1';
    elsif (Count = OutputEnableStopCount) then
      OutputEnable <= '0';
    end if;
  end if;
end process;

RS_Data_out_Start_Control : process (clk)
begin
```

```
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out_Start <= '0';
    elsif (Count = RS_Data_out_StartCount) then
      RS_Data_out_Start <= '1';
    else
      RS_Data_out_Start <= '0';
    end if;
  end if;
end process;

DoChien_Control : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (Count = DoChienCountMax)) then
      DoChien <= '0';
    elsif (InitChien = '1') then
      DoChien <= '1';
    end if;
  end if;
end process;

X1_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      X1    <= alpha77;
      X1_D1 <= zero;
      X1_D2 <= zero;
      X1_D3 <= zero;
      X1_D4 <= zero;
      X1_D5 <= zero;
    else
      X1    <= mul(X1,alpha1);
      X1_D1 <= X1;
      X1_D2 <= X1_D1;
      X1_D3 <= X1_D2;
      X1_D4 <= X1_D3;
      X1_D5 <= X1_D4;
    end if;
  end if;
end process;

Powers_of_X_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      X2 <= zero;
      X3 <= zero;
      X4 <= zero;
      X5 <= zero;
      X6 <= zero;
      X7 <= zero;
    else
      X2 <= mul(X1,X1);
      X3 <= mul(X2,X1_D1);
      X4 <= mul(X3,X1_D2);
      X5 <= mul(X4,X1_D3);
      X6 <= mul(X5,X1_D4);
      X7 <= mul(X6,X1_D5);
    end if;
  end if;
end process;

EVAL_OM_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_OM_1  <= zero;
      EVAL_OM_2  <= zero;
      EVAL_OM_3  <= zero;
```

```
      EVAL_OM_4   <= zero;
      EVAL_OM_5   <= zero;
      EVAL_OM_6   <= zero;
      EVAL_OM     <= zero;
      EVAL_OM_D1 <= zero;
    else
      EVAL_OM_1   <= add(IntOmega(0),mul(IntOmega(1),X1));
      EVAL_OM_2   <= add(EVAL_OM_1,mul(IntOmega(2),X2));
      EVAL_OM_3   <= add(EVAL_OM_2,mul(IntOmega(3),X3));
      EVAL_OM_4   <= add(EVAL_OM_3,mul(IntOmega(4),X4));
      EVAL_OM_5   <= add(EVAL_OM_4,mul(IntOmega(5),X5));
      EVAL_OM_6   <= add(EVAL_OM_5,mul(IntOmega(6),X6));
      EVAL_OM     <= add(EVAL_OM_6,mul(IntOmega(7),X7));
      EVAL_OM_D1 <= EVAL_OM;
    end if;
  end if;
end process;

EVAL_LP_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_LP_1 <= zero;
      EVAL_LP_2 <= zero;
      EVAL_LP_3 <= zero;
      EVAL_LP_4 <= zero;
      EVAL_LP   <= zero;
      EVAL_DEN  <= zero;
      DEN_INV   <= zero;
    else
      EVAL_LP_1 <= add(IntLambda_1,mul(IntLambda_3,X2));
      EVAL_LP_2 <= EVAL_LP_1;
      EVAL_LP_3 <= add(EVAL_LP_2,mul(IntLambda_5,X4));
      EVAL_LP_4 <= EVAL_LP_3;
      EVAL_LP   <= add(EVAL_LP_4,mul(IntLambda_7,X6));
      EVAL_DEN  <= mul(EVAL_LP,K_VAL); -- K_mul stage is needed
      DEN_INV   <= inv(EVAL_DEN); -- K_mul stage is needed or EVAL_LAMBDAPRIME delay
stage = 1
    end if;
  end if;
end process;

K_VAL_Process : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      K_VAL <= K_VAL_initial_value;
    else
      K_VAL <= mul(K_VAL,K_MUL);
    end if;
  end if;
end process;

Calc_Correction_Factor : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      CORR_FACTOR <= (others=>'0');
    elsif (rs_enable = '1') then
      CORR_FACTOR   <= mul(EVAL_OM_D1,DEN_INV);
    else
      CORR_FACTOR <= (others=>'0');
    end if;
  end if;
end process;

Delay_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      for i in 0 to SR_max loop
```

```
        shift_register_delay(i) <= (others=>'0');
      end loop;
    else
      for i in SR_max downto 1 loop
        shift_register_delay(i) <= shift_register_delay(i-1);
      end loop;
      shift_register_delay(0) <= RS_Data_In;
    end if;
  end if;
end process;
RS_data_delayed <= shift_register_delay(SR_max);

Correct_RS_Data : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      RS_Data_out <= zero;
    elsif ((ChienSum = zero) and (OutputEnable = '1')) then
      RS_Data_out <= add(CORR_FACTOR,RS_data_delayed);
    elsif (OutputEnable = '1') then
      RS_Data_out <= RS_data_delayed;
    else
      RS_Data_out <= zero;
    end if;
  end if;
end process;

Calc_Temp_Registers : process (clk)
begin
  if (clk'event and clk = '1') then
    if (reset_n = '0') then
      Temp1 <= zero;
      Temp2 <= zero;
      Temp3 <= zero;
      Temp4 <= zero;
      Temp5 <= zero;
      Temp6 <= zero;
      Temp7 <= zero;
      Temp8 <= zero;
    elsif (InitChien = '1') then
      Temp1 <= mul(alpha77,Lambda(1));    --   77 = (256 - 179)*  1 mod 255 =   77 mod 255
      Temp2 <= mul(alpha152,Lambda(2));   --  152 = (256 - 180)*  2 mod 255 =  152 mod 255
      Temp3 <= mul(alpha225,Lambda(3));   --  225 = (256 - 181)*  3 mod 255 =  225 mod 255
      Temp4 <= mul(alpha41,Lambda(4));    --   41 = (256 - 182)*  4 mod 255 =   41 mod 255
      Temp5 <= mul(alpha110,Lambda(5));   --  110 = (256 - 183)*  5 mod 255 =  110 mod 255
      Temp6 <= mul(alpha177,Lambda(6));   --  177 = (256 - 184)*  6 mod 255 =  177 mod 255
      Temp7 <= mul(alpha242,Lambda(7));   --  242 = (256 - 185)*  7 mod 255 =  242 mod 255
      Temp8 <= mul(alpha50,Lambda(8));    --   50 = (256 - 186)*  8 mod 255 =   50 mod 255
    elsif (DoChien = '1') then
      Temp1 <= mul(alpha1,Temp1);
      Temp2 <= mul(alpha2,Temp2);
      Temp3 <= mul(alpha3,Temp3);
      Temp4 <= mul(alpha4,Temp4);
      Temp5 <= mul(alpha5,Temp5);
      Temp6 <= mul(alpha6,Temp6);
      Temp7 <= mul(alpha7,Temp7);
      Temp8 <= mul(alpha8,Temp8);
    end if;
  end if;
end process;

Chien_Sum_Pipeline : process (clk)
begin
  if (clk'event and clk = '1') then
    if ((reset_n = '0') or (StartChien = '1')) then
      EVAL_CS_1   <= zero;
      EVAL_CS_2   <= zero;
      EVAL_CS_3   <= zero;
      EVAL_CS_4   <= zero;
      EVAL_CS_5   <= zero;
      EVAL_CS_6   <= zero;
```

```
      EVAL_CS_7    <= zero;
      ChienSum     <= zero;
    elsif (DoChien = '1') then
      EVAL_CS_1    <= add(IntLambda_0,Temp1);
      EVAL_CS_2    <= add(EVAL_CS_1,Temp2);
      EVAL_CS_3    <= add(EVAL_CS_2,Temp3);
      EVAL_CS_4    <= add(EVAL_CS_3,Temp4);
      EVAL_CS_5    <= add(EVAL_CS_4,Temp5);
      EVAL_CS_6    <= add(EVAL_CS_5,Temp6);
      EVAL_CS_7    <= add(EVAL_CS_6,Temp7);
      ChienSum     <= add(EVAL_CS_7,Temp8);
    end if;
  end if;
end process;


end;


----------------------------------------
-- End of ChienSearchProcess
----------------------------------------


----------------------------------------
-- Start of RSTopProcess
----------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity RS_Decoder_top is
  generic (GFPower : INTEGER := 8; RS_N : INTEGER := 179; RS_T : INTEGER := 8);
  port (
        clk : in std_logic;
        reset_n : in std_logic;
        rs_data_in : in std_logic_vector(GFPower - 1 downto 0 );
        rs_data_in_start : in std_logic;
        RS_Data_out : out std_logic_vector(GFPower - 1 downto 0 );
        RS_Data_out_Start : out std_logic;
        rs_enable : in std_logic
        );
end RS_Decoder_top;


use work.all;
architecture RS_Decoder_top of RS_Decoder_top is

  signal errors_present : std_logic;
  signal lambda_poly : std_logic_vector((RS_T + 1) * GFPower - 1 downto 0 );
  signal omega_poly : std_logic_vector(RS_T * GFPower - 1 downto 0 );
  signal StartChien : std_logic;
  signal syndrome : std_logic_vector(2 * GFPower * RS_T - 1 downto 0 );
  signal syndrome_calc_done : std_logic;
  component MBSolver
      port (
            reset_n : in std_logic;
            clk : in std_logic;
            ErrorsPresent : in std_logic;
            XferSyndrome : in std_logic;
            StartChien : out std_logic;
            syndrome_poly : in std_logic_vector(2 * GFPower * RS_T - 1 downto 0 );
            omega_poly : out std_logic_vector(RS_T * GFPower - 1 downto 0 );
            lambda_poly : out std_logic_vector((RS_T + 1) * GFPower - 1 downto 0 )
            );
  end component;
  component InputProcess
      port (
            reset_n : in std_logic;
            clk : in std_logic;
            rs_data_in : in std_logic_vector(GFPower - 1 downto 0 );
            rs_data_in_start : in std_logic;
```

```
            syndrome_calc_done : out std_logic;
            errors_present : out std_logic;
            syndrome : out std_logic_vector(2 * GFPower * RS_T - 1 downto 0 )
            );
  end component;
  component ChienSearchProcess
      port (
            clk : in std_logic;
            reset_n : in std_logic;
            lambda_poly : in std_logic_vector((RS_T + 1) * GFPower - 1 downto 0 );
            omega_poly : in std_logic_vector(RS_T * GFPower - 1 downto 0 );
            StartChien : in std_logic;
            rs_enable : in std_logic;
            RS_Data_In : in std_logic_vector(GFPower - 1 downto 0 );
            RS_Data_out : out std_logic_vector(GFPower - 1 downto 0 );
            RS_Data_out_Start : out std_logic
            );
  end component;

begin

  inst_MBSolver: MBSolver
    port map (
              reset_n => reset_n,
              clk => clk,
              ErrorsPresent => errors_present,
              XferSyndrome => syndrome_calc_done,
              StartChien => StartChien,
              syndrome_poly => syndrome(2 * GFPower * RS_T - 1 downto 0),
              omega_poly => omega_poly(RS_T * GFPower - 1 downto 0),
              lambda_poly => lambda_poly((RS_T + 1) * GFPower - 1 downto 0));

  inst_InputProcess: InputProcess
    port map (
              reset_n => reset_n,
              clk => clk,
              rs_data_in => rs_data_in(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start,
              syndrome_calc_done => syndrome_calc_done,
              errors_present => errors_present,
              syndrome => syndrome(2 * GFPower * RS_T - 1 downto 0));

  inst_ChienSearch: ChienSearchProcess
    port map (
              clk => clk,
              reset_n => reset_n,
              lambda_poly => lambda_poly((RS_T + 1) * GFPower - 1 downto 0),
              omega_poly => omega_poly(RS_T * GFPower - 1 downto 0),
              StartChien => StartChien,
              rs_enable => rs_enable,
              RS_Data_In => rs_data_in(GFPower - 1 downto 0),
              RS_Data_out => RS_Data_out(GFPower - 1 downto 0),
              RS_Data_out_Start => RS_Data_out_Start);
end RS_Decoder_top;

----------------------------------------
-- End of RSTopProcess
```

17 Appendix E - RS Encoder Testbench VHDL Code

This appendix contains the VHDL code generated for the RS encoder testbench.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity  tb_validate_encoder  is

generic (
           encoder_expected_filename  : string := "enc_exp.txt";
           report_filename    : string := "enc_report.txt";
           GFPower : integer := 4;
           RS_N : integer := 15;
           RS_T : integer := 3
         );

port (encoder_data_out : in std_logic_vector (7 downto 0);
      encoder_output_strobe : in std_logic ;
      clk : in std_logic ;
      reset_n : in std_logic );
end;
architecture  RTL  of  tb_validate_encoder  is

file encoder_expected_file : text is in encoder_expected_filename;
file report_file : text is out report_filename;
type pass_fail_type is (pass, fail);
signal sampling_encoder : std_logic;
signal overall_pass_fail_status : pass_fail_type := pass;

procedure write_output_separator is
variable l1: line;
begin
  write(l1,
string'("|=============================================================================|"));
  writeline(output,l1);
end write_output_separator;

procedure write_report_separator is
variable l1: line;
begin
  write(l1,
string'("|=============================================================================|"));
  writeline(report_file,l1);
end write_report_separator;

procedure make_pass_fail_line (RS_check_name : in string; cell_count : in integer;
pass_fail_status : in pass_fail_type; l1 : inout line ) is
begin
  write(l1,string'("| "));
  write(l1,RS_check_name);
  write(l1,string'(" codeword "));
  write(l1,cell_count,justified=>right,field=>4);
  if (pass_fail_status=pass) then
    write(l1,string'("                ; PASS/FAIL status = |  PASS  |"));
  else
    write(l1,string'("                ; PASS/FAIL status = |  FAIL  | ******** FAIL"));
  end if;
end make_pass_fail_line;

procedure data_compare (variable_name : in string; actual_data : in integer; exp_data :
in integer;
                        status : inout pass_fail_type ) is
variable l1: line;
begin
  write(l1,string'("| "));
  write(l1,variable_name);
  write(l1,string'(": expected = "));
  write(l1,exp_data,justified=>right,field=>4);
```

```
  write(l1,string'(" : actual = "));
  write(l1,actual_data,justified=>right,field=>4);
  if (actual_data=exp_data) then
    write(l1, string'("       |  PASS  |"));
  else
    write(l1, string'("       |  FAIL  |"));
    status := fail;
  end if;
  writeline(report_file,l1);
end data_compare;

begin


verify_encoder_outptut : process

variable l1, l2 : line;
variable cell_count : integer;
variable actual_encoder_data_int : integer;
variable expected_encoder_data_int : integer;
variable pass_fail_status : pass_fail_type;

begin

  pass_fail_status := pass;
  overall_pass_fail_status <= pass;
  cell_count := 0;
  sampling_encoder <= '0';
  write_output_separator;

  while (true) loop

  wait until encoder_output_strobe'event and encoder_output_strobe='1';
  wait until clk'event and clk='0';
  wait until clk'event and clk='0';
  cell_count := cell_count + 1;
  pass_fail_status := pass;
  sampling_encoder <= '0';

  write_report_separator;
  for i in RS_N-1 downto 0 loop
    readline(encoder_expected_file, l1);
    read(l1,expected_encoder_data_int);
    sampling_encoder <= '1';
    actual_encoder_data_int := conv_integer('0'&encoder_data_out);
    data_compare(string'("encoder_data_out
"),actual_encoder_data_int,expected_encoder_data_int,pass_fail_status);
    if (pass_fail_status=fail) then overall_pass_fail_status <= fail; end if;
    wait for 1 ns;
    sampling_encoder <= '0';
    wait until clk'event and clk='0';
  end loop;
  write_report_separator;

  make_pass_fail_line(string'("encoder"),cell_count,pass_fail_status,l2);
  writeline(report_file,l2);
  write_report_separator;
  write(l2,string'(" "));
  writeline(report_file,l2); -- this writes a blank line
  make_pass_fail_line(string'("encoder"),cell_count,pass_fail_status,l2);
  writeline(output,l2);

  wait until clk'event and clk='0';
  sampling_encoder <= '0';

  if (endfile(encoder_expected_file)) then
    write_output_separator;
    write(l1,string'(" "));
    writeline(report_file,l1);
    write(l1,string'("End of processing at "));
```

```
    write(ll, now);
    writeline(report_file,ll);
    write(ll,string'(" "));
    writeline(output,ll);
    write(ll,string'("End of processing at "));
    write(ll, now);
    writeline(output,ll);

    if (overall_pass_fail_status=pass) then
      assert (false) report "Simulation Done. All tests have PASSED."  severity error;
    else
      assert (false) report "Simulation Done.  ATTENTION : There were some failures. --
FAIL FAIL FAIL"  severity error;
    end if;
    wait;
  end if;

  end loop; -- while (TRUE) loop

end process verify_encoder_outptut;


end;



library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use ieee.std_logic_arith.all;
use std.textio.ALL;

entity  tb_rs_encoder_stim  is
generic (
   GFPower : integer := 8; RS_N : integer := 20; RS_T : integer := 1
   );

port (reset_n : out std_logic ;
      clk : out std_logic ;
      data_size : out std_logic_vector ((GFPower-1) downto 0);
      rs_data_in : out std_logic_vector ((GFPower-1) downto 0);
      rs_data_in_start : out std_logic );
end;
architecture  RTL  of  tb_rs_encoder_stim  is
signal int_clk: std_logic;

constant in_filename: string := "enc_stim.txt";

begin

generate_Intsys_clk : process
begin
  int_clk <= '1';
  while true loop
    wait for 50 ns;
    int_clk <= not int_clk;
    wait for 50 ns;
    int_clk <= not int_clk;
  end loop;
end process;
clk <= int_clk;

process
begin
  reset_n <= '0';
  for i in 1 to 40 loop
    wait until (int_clk'event and int_clk='1');
  end loop;
  reset_n <= '1';
  wait;
end process;
```

```
generate_stimulus : process
variable stimulus_var : std_logic_vector(GFPower downto 0);
file infile: text is in in_filename;
variable stimulus_line  : line;
variable RS_N_int : integer;
variable RS_T_int : integer;

begin
  rs_data_in_start <= '0';
  rs_data_in <= (others => '0');
  data_size <= conv_std_logic_vector(RS_N-2*RS_T,GFPower);
for i in 1 to 50 loop
  wait until int_clk'event and int_clk='1';
end loop;

while not endfile(infile) loop
  wait for 5 ns;
  readline(infile, stimulus_line);
  read(stimulus_line, stimulus_var);
  rs_data_in_start <= stimulus_var(GFPower);
  rs_data_in <= stimulus_var((GFPower-1) downto 0);
  wait until int_clk'event and int_clk='1';
end loop; -- while not endfile(infile) loop

wait; -- forever
end process generate_stimulus;

end;

library ieee;
use ieee.std_logic_1164.all;
entity top_tb_RS_encoder is
  generic (
           GFPower : INTEGER := 8;
           RS_N : INTEGER := 15;
           RS_T : INTEGER := 1
           );


end top_tb_RS_encoder;


use work.all;
architecture RTL of top_tb_RS_encoder is

  signal clk : std_logic;
  signal data_size : std_logic_vector(GFPower - 1 downto 0 );
  signal encoder_data_out : std_logic_vector(GFPower - 1 downto 0 );
  signal encoder_output_strobe : std_logic;
  signal reset_n : std_logic;
  signal rs_data_in : std_logic_vector(GFPower - 1 downto 0 );
  signal rs_data_in_start : std_logic;
  component tb_rs_encoder_stim
      generic (
           GFPower : INTEGER := 8;
           RS_N : INTEGER := 15;
           RS_T : INTEGER := 1
               );
      port (
            reset_n : out std_logic;
            clk : out std_logic;
            data_size : out std_logic_vector((GFPower - 1) downto 0 );
            rs_data_in : out std_logic_vector((GFPower - 1) downto 0 );
            rs_data_in_start : out std_logic
            );
  end component;
  component rs_encoder
      port (
            reset_n : in std_logic;
            data_out : out std_logic_vector(GFPower - 1 downto 0 );
```

```
            data_in : in std_logic_vector(GFPower - 1 downto 0 );
            input_strobe : in std_logic;
            clk : in std_logic;
            output_strobe : out std_logic;
            data_size : in std_logic_vector(GFPower - 1 downto 0 )
            );
  end component;
  component tb_validate_encoder
      generic (
                encoder_expected_filename : STRING := "enc_exp.txt";
                report_filename : STRING := "enc_report.txt";
                GFPower : INTEGER := 4;
                RS_N : INTEGER := 15;
                RS_T : INTEGER := 3
                );
      port (
            encoder_data_out : in std_logic_vector(GFPower - 1 downto 0 );
            encoder_output_strobe : in std_logic;
            clk : in std_logic;
            reset_n : in std_logic
            );
  end component;


begin

  inst_tb_rs_encoder_stim: tb_rs_encoder_stim
    generic map (GFPower, RS_N, RS_T)
    port map (
              reset_n => reset_n,
              clk => clk,
              data_size => data_size(GFPower - 1 downto 0),
              rs_data_in => rs_data_in(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start);

  inst_rs_encoder: rs_encoder
    port map (
              reset_n => reset_n,
              data_out => encoder_data_out(GFPower - 1 downto 0),
              data_in => rs_data_in(GFPower - 1 downto 0),
              input_strobe => rs_data_in_start,
              clk => clk,
              output_strobe => encoder_output_strobe,
              data_size => data_size(GFPower - 1 downto 0));

  inst_tb_validate_encoder: tb_validate_encoder
    generic map ("enc_exp.txt",
                 "enc_report.txt",
                 GFPower,
                 RS_N,
                 RS_T)
    port map (
              encoder_data_out => encoder_data_out(GFPower - 1 downto 0),
              encoder_output_strobe => encoder_output_strobe,
              clk => clk,
              reset_n => reset_n);
end RTL;
```

18 Appendix F - RS Decoder Testbench VHDL Code

This appendix contains the VHDL code generated for the RS decoder testbench.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity  tb_validate_RS  is

generic (
           InputProcess_expected_filename  : string := "ip_exp.txt";
           MBSolver_expected_filename  : string := "mb_exp.txt";
           ChienSearch_expected_filename  : string := "cs_exp.txt";
           stimulus_filename  : string := "ip_stim.txt";
           report_filename    : string := "ip_report.txt";
           GFPower : integer := 4;
           RS_N : integer := 15;
           RS_T : integer := 3
        );

port (syndrome_calc_done : in std_logic ;
      errors_present : in std_logic ;
      StartChien : in std_logic ;
      lambda_poly : in std_logic_vector ((GFPower * (RS_T + 1) - 1) downto 0);
      omega_poly : in std_logic_vector ((GFPower * RS_T - 1) downto 0);
      syndrome : in std_logic_vector ((GFPower * RS_T * 2 - 1) downto 0);
      reset_n : in std_logic ;
      clk : in std_logic ;
      blk_out : in std_logic_vector (GFPower-1 downto 0);
      blk_sync_out : in std_logic  );
end;
architecture  RTL  of  tb_validate_RS  is

file InputProcess_expected_file : text is in InputProcess_expected_filename;
file MBSolver_expected_file : text is in MBSolver_expected_filename;
file ChienSearch_expected_file : text is in ChienSearch_expected_filename;
file report_file : text is out report_filename;
type pass_fail_type is (pass, fail);
signal sampling_IP : std_logic;
signal sampling_MB: std_logic;
signal sampling_CS : std_logic;
signal syndrome_element : std_logic_vector(GFPower-1 downto 0);
signal overall_IP_pass_fail_status : pass_fail_type := pass;
signal overall_MB_pass_fail_status : pass_fail_type := pass;
signal overall_CS_pass_fail_status : pass_fail_type := pass;

procedure write_output_separator is
variable l1: line;
begin
  write(l1,
string'("|================================================================|"));
  writeline(output,l1);
end write_output_separator;

procedure write_report_separator is
variable l1: line;
begin
  write(l1,
string'("|================================================================|"));
  writeline(report_file,l1);
end write_report_separator;

procedure make_pass_fail_line (RS_check_name : in string; cell_count : in integer;
pass_fail_status : in pass_fail_type; l1 : inout line ) is
begin
  write(l1,string'("| "));
  write(l1,RS_check_name);
  write(l1,string'(" codeword "));
  write(l1,cell_count,justified=>right,field=>4);
  if (pass_fail_status=pass) then
    write(l1,string'(" ; PASS/FAIL status = |  PASS  |"));
```

```
  else
    write(l1,string'(" ; PASS/FAIL status = |  FAIL  | ******** FAIL"));
  end if;
end make_pass_fail_line;

procedure data_compare (variable_name : in string; actual_data : in integer; exp_data :
in integer;
                                status : inout pass_fail_type ) is
variable l1: line;
begin
  write(l1,string'("| "));
  write(l1,variable_name);
  write(l1,string'(": expected = "));
  write(l1,exp_data,justified=>right,field=>4);
  write(l1,string'(" : actual = "));
  write(l1,actual_data,justified=>right,field=>4);
  if (actual_data=exp_data) then
    write(l1, string'("       |  PASS  |"));
  else
    write(l1, string'("       |  FAIL  |"));
    status := fail;
  end if;
  writeline(report_file,l1);
end data_compare;


begin -- architecture  RTL  of  tb_validate_IP

verify_InputProcess_outptut : process

variable l1, l2 : line;
variable cell_count : integer;
variable expected_syndrome_int : integer;
variable actual_syndrome_int : integer;
variable pass_fail_status : pass_fail_type;
variable first_bit : integer;
variable last_bit : integer;

begin

  syndrome_element <= (others=>'0');
  write(l1,string'("stimulus file : "));
  write(l1,stimulus_filename);
  writeline(output,l1);
  write(l1,string'("InputProcess expected file : "));
  write(l1,InputProcess_expected_filename);
  writeline(output,l1);
  write(l1,string'("    MBSolver expected file : "));
  write(l1,MBSolver_expected_filename);
  writeline(output,l1);
  write(l1,string'(" ChienSearch expected file : "));
  write(l1,ChienSearch_expected_filename);
  writeline(output,l1);
  write(l1,string'("               report file : "));
  write(l1,report_filename);
  writeline(output,l1);
  writeline(output,l1);

  write(l1,string'("stimulus file : "));
  write(l1,stimulus_filename);
  writeline(report_file,l1);
  write(l1,string'("InputProcess expected file : "));
  write(l1,InputProcess_expected_filename);
  writeline(report_file,l1);
  write(l1,string'("    MBSolver expected file : "));
  write(l1,MBSolver_expected_filename);
  writeline(report_file,l1);
  write(l1,string'(" ChienSearch expected file : "));
  write(l1,ChienSearch_expected_filename);
  writeline(report_file,l1);
  write(l1,string'("               report file : "));
```

```
  write(l1,report_filename);
  writeline(report_file,l1);
  writeline(report_file,l1);

  pass_fail_status := pass;
  overall_IP_pass_fail_status <= pass;
  cell_count := 0;
  sampling_IP <= '0';
  write_output_separator;

  while (true) loop

  wait until syndrome_calc_done'event and syndrome_calc_done='1';
  wait until clk'event and clk='0';
  wait until clk'event and clk='0';
  cell_count := cell_count + 1;
  pass_fail_status := pass;

  write_report_separator;
  sampling_IP <= '1';
  for i in 0 to 2*RS_T-1 loop
    readline(InputProcess_expected_file, l1);
    read(l1,expected_syndrome_int);
    first_bit := GFPower*(i+1) - 1;
    last_bit := GFPower*i;
    syndrome_element <= syndrome(first_bit downto last_bit);
    actual_syndrome_int := conv_integer('0'&syndrome(first_bit downto last_bit));
    write(l1,string'("  actual_syndrome : "));
    write(l1,syndrome(first_bit downto last_bit));
    write(l1,report_filename);
    data_compare(string'("syndrome
"),actual_syndrome_int,expected_syndrome_int,pass_fail_status);
    if (pass_fail_status=fail) then overall_IP_pass_fail_status <= fail; end if;
    first_bit := first_bit - GFPower;
  end loop;
  write_report_separator;
  make_pass_fail_line(string'("InputProcess     "),cell_count,pass_fail_status,l2);
  writeline(report_file,l2);
  make_pass_fail_line(string'("InputProcess     "),cell_count,pass_fail_status,l2);
  writeline(output,l2);
  wait for 1 ns;
  sampling_IP <= '0';
  wait until clk'event and clk='0';

  end loop; -- while (TRUE) loop

end process verify_InputProcess_outptut;



verify_MBSolver_outptut : process

variable l1, l2 : line;
variable cell_count : integer;
variable expected_lambda_int : integer;
variable actual_lambda_int : integer;
variable expected_omega_int : integer;
variable skip_MB_int : integer;
variable actual_omega_int : integer;
variable pass_fail_status : pass_fail_type;
variable first_bit : integer;
variable last_bit : integer;

begin

  overall_MB_pass_fail_status <= pass;
  pass_fail_status := pass;
  cell_count := 0;

  while (true) loop
```

```
  wait until StartChien'event and StartChien='1';
  wait until clk'event and clk='0';
  cell_count := cell_count + 1;
  pass_fail_status := pass;
  sampling_MB <= '0';

  write_report_separator;
  for i in 0 to RS_T loop
    readline(MBSolver_expected_file, l1);
    read(l1,expected_lambda_int);
    if (i=0) then read(l1,skip_MB_int); end if;
    first_bit := GFPower*(i+1) - 1;
    last_bit := GFPower*i;
    sampling_MB <= '1';
    actual_lambda_int := conv_integer('0'&lambda_poly(first_bit downto last_bit));
    if (skip_MB_int=0) then
      data_compare(string'("lambda
"),actual_lambda_int,expected_lambda_int,pass_fail_status);
    end if;
    if (pass_fail_status=fail) then overall_MB_pass_fail_status <= fail; end if;
    wait for 1 ns;
    sampling_MB <= '0';
    wait for 1 ns;
    first_bit := first_bit - GFPower;
  end loop;
  write_report_separator;

  for i in 0 to RS_T-1 loop
    readline(MBSolver_expected_file, l1);
    read(l1,expected_omega_int);
    first_bit := GFPower*(i+1) - 1;
    last_bit := GFPower*i;
    sampling_MB <= '1';
    actual_omega_int := conv_integer('0'&omega_poly(first_bit downto last_bit));
    if (skip_MB_int=0) then
      data_compare(string'("omega
"),actual_omega_int,expected_omega_int,pass_fail_status);
    end if;
    if (pass_fail_status=fail) then overall_MB_pass_fail_status <= fail; end if;
    wait for 1 ns;
    sampling_MB <= '0';
    wait for 1 ns;
    first_bit := first_bit - GFPower;
  end loop;
  write_report_separator;
  make_pass_fail_line(string'("MBSolver          "),cell_count,pass_fail_status,l2);
  writeline(report_file,l2);
  make_pass_fail_line(string'("MBSolver          "),cell_count,pass_fail_status,l2);
  writeline(output,l2);

  wait until clk'event and clk='0';
  sampling_MB <= '0';

  end loop; -- while (TRUE) loop

end process verify_MBSolver_outptut;


verify_ChienSearch_outptut : process

variable l1, l2 : line;
variable cell_count : integer;
variable actual_rs_data_int : integer;
variable expected_rs_data_int : integer;
variable pass_fail_status : pass_fail_type;

begin

  pass_fail_status := pass;
  overall_CS_pass_fail_status <= pass;
  cell_count := 0;
```

```
  while (true) loop

  wait until blk_sync_out'event and blk_sync_out='1';
  wait until clk'event and clk='0';
  cell_count := cell_count + 1;
  pass_fail_status := pass;
  sampling_CS <= '0';

  write_report_separator;
  for i in RS_N-1 downto 0 loop
    readline(ChienSearch_expected_file, l1);
    read(l1,expected_rs_data_int);
    sampling_CS <= '1';
    actual_rs_data_int := conv_integer('0'&blk_out);
    data_compare(string'("rs_data_out
"),actual_rs_data_int,expected_rs_data_int,pass_fail_status);
    if (pass_fail_status=fail) then overall_CS_pass_fail_status <= fail; end if;
    wait for 1 ns;
    sampling_CS <= '0';
    wait until clk'event and clk='0';
  end loop;
  write_report_separator;

  make_pass_fail_line(string'("ChienSearch      "),cell_count,pass_fail_status,l2);
  writeline(report_file,l2);
  write_report_separator;
  write(l2,string'(" "));
  writeline(report_file,l2); -- this writes a blank line
  make_pass_fail_line(string'("ChienSearch      "),cell_count,pass_fail_status,l2);
  writeline(output,l2);
  write_output_separator;

  wait until clk'event and clk='0';
  sampling_CS <= '0';

  if (endfile(ChienSearch_expected_file)) then
    write(l1,string'(" "));
    writeline(report_file,l1);
    write(l1,string'("End of processing at "));
    write(l1, now);
    writeline(report_file,l1);
    write(l2,string'(" "));
    writeline(output,l2);
    write(l2,string'("End of processing at "));
    write(l2, now);
    writeline(output,l2);

    if ((overall_IP_pass_fail_status=pass) and (overall_MB_pass_fail_status=pass) and
(overall_CS_pass_fail_status=pass)) then
      assert (false) report "Simulation Done. All tests have PASSED."  severity error;
    else
      assert (false) report "Simulation Done.  ATTENTION : There were some failures. --
FAIL FAIL FAIL"  severity error;
    end if;
    write(l1,string'("overall_IP_pass_fail_status = "));
    if (overall_IP_pass_fail_status=pass) then write(l1,string'("PASS")); else
write(l1,string'("FAIL")); end if;
    writeline(output,l1);
    write(l1,string'("overall_MB_pass_fail_status = "));
    if (overall_MB_pass_fail_status=pass) then write(l1,string'("PASS")); else
write(l1,string'("FAIL")); end if;
    writeline(output,l1);
    write(l1,string'("overall_CS_pass_fail_status = "));
    if (overall_CS_pass_fail_status=pass) then write(l1,string'("PASS")); else
write(l1,string'("FAIL")); end if;
    writeline(output,l1);
    assert (false) report "Simulation Done. End of Report."  severity error;
    wait;
  end if;
```

```
  end loop; -- while (TRUE) loop

end process verify_ChienSearch_outptut;

end;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use ieee.std_logic_arith.all;
use std.textio.ALL;

entity  tb_rs_stim  is
generic (
   GFPower : integer := 8
   );

port (reset_n : out std_logic ;
      clk : out std_logic ;
      rs_enable : out std_logic;
      rs_data_in : out std_logic_vector ((GFPower-1) downto 0);
      rs_data_in_start : out std_logic );
end;
architecture  RTL  of  tb_rs_stim  is
signal int_clk: std_logic;

constant in_filename: string := "ip_stim.txt";

begin

generate_Intsys_clk : process
begin
  int_clk <= '1';
  while true loop
    wait for 50 ns;
    int_clk <= not int_clk;
    wait for 50 ns;
    int_clk <= not int_clk;
  end loop;
end process;
clk <= int_clk;

process
begin
  reset_n <= '0';
  rs_enable <= '1';
  for i in 1 to 40 loop
    wait until (int_clk'event and int_clk='1');
  end loop;
  reset_n <= '1';
  wait;
end process;

generate_stimulus : process
variable stimulus_var : std_logic_vector(GFPower downto 0);
file infile: text is in in_filename;
variable stimulus_line  : line;
variable RS_N_int : integer;
variable RS_T_int : integer;

begin
  rs_data_in_start <= '0';
  rs_data_in <= (others => '0');

for i in 1 to 50 loop
  wait until int_clk'event and int_clk='1';
end loop;

while not endfile(infile) loop
  wait for 5 ns;
  readline(infile, stimulus_line);
```

```
  read(stimulus_line, stimulus_var);
  rs_data_in_start <= stimulus_var(GFPower);
  rs_data_in <= stimulus_var((GFPower-1) downto 0);
  wait until int_clk'event and int_clk='1';
end loop; -- while not endfile(infile) loop

wait; -- forever
end process generate_stimulus;

end;

library ieee;
use ieee.std_logic_1164.all;
entity top_tb_RS_decoder is
  generic (
           GFPower : INTEGER := 8;
           RS_N : INTEGER := 22;
           RS_T : INTEGER := 8
           );


end top_tb_RS_decoder;


use work.all;
architecture RTL of top_tb_RS_decoder is

  signal clk : std_logic;
  signal reset_n : std_logic;
  signal rs_data_in : std_logic_vector(GFPower - 1 downto 0 );
  signal rs_data_in_start : std_logic;
  signal RS_Data_out : std_logic_vector(7 downto 0 );
  signal RS_Data_out_Start : std_logic;
  signal rs_enable : std_logic;
  signal tb_errors_present : std_logic;
  signal tb_lambda_poly : std_logic_vector((RS_T + 1) * GFPower - 1 downto 0 );
  signal tb_omega_poly : std_logic_vector(RS_T * GFPower - 1 downto 0 );
  signal tb_StartChien : std_logic;
  signal tb_syndrome : std_logic_vector(2 * GFPower * RS_T - 1 downto 0 );
  signal tb_syndrome_calc_done : std_logic;
  component tb_rs_stim
      generic (
               GFPower : INTEGER := 8
               );
      port (
            reset_n : out std_logic;
            clk : out std_logic;
            rs_enable : out std_logic;
            rs_data_in : out std_logic_vector((GFPower - 1) downto 0 );
            rs_data_in_start : out std_logic
            );
  end component;
  component tb_validate_RS
      generic (
               InputProcess_expected_filename : STRING := "ip_exp.txt";
               MBSolver_expected_filename : STRING := "mb_exp.txt";
               ChienSearch_expected_filename : STRING := "cs_exp.txt";
               stimulus_filename : STRING := "ip_stim.txt";
               report_filename : STRING := "ip_report.txt";
               GFPower : INTEGER := 4;
               RS_N : INTEGER := 15;
               RS_T : INTEGER := 3
               );
      port (
            syndrome_calc_done : in std_logic;
            errors_present : in std_logic;
            StartChien : in std_logic;
            lambda_poly : in std_logic_vector((GFPower * (RS_T + 1) - 1) downto
            0 );
            omega_poly : in std_logic_vector((GFPower * RS_T - 1) downto 0 );
            syndrome : in std_logic_vector((GFPower * RS_T * 2 - 1) downto 0 );
```

```
            reset_n : in std_logic;
            clk : in std_logic;
            blk_out : in std_logic_vector(GFPower - 1 downto 0 );
            blk_sync_out : in std_logic
            );
  end component;
  component RS_Decoder_top
      generic (
                GFPower : INTEGER := 8;
                RS_N : INTEGER := 71;
                RS_T : INTEGER := 8
                );
      port (
            clk : in std_logic;
            reset_n : in std_logic;
            rs_data_in : in std_logic_vector(GFPower - 1 downto 0 );
            rs_data_in_start : in std_logic;
            RS_Data_out : out std_logic_vector(7 downto 0 );
            RS_Data_out_Start : out std_logic;
            rs_enable : in std_logic;
            tb_errors_present : out std_logic;
            tb_lambda_poly : out std_logic_vector((RS_T + 1) * GFPower - 1
             downto 0 );
            tb_omega_poly : out std_logic_vector(RS_T * GFPower - 1 downto 0 );
            tb_StartChien : out std_logic;
            tb_syndrome : out std_logic_vector(2 * GFPower * RS_T - 1 downto 0 );
            tb_syndrome_calc_done : out std_logic
            );
  end component;

begin

  inst_tb_rs_stim: tb_rs_stim
    generic map (GFPower)
    port map (
              reset_n => reset_n,
              clk => clk,
              rs_enable => rs_enable,
              rs_data_in => rs_data_in(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start);

  inst_tb_validate_RS: tb_validate_RS
    generic map ("ip_exp.txt",
                 "mb_exp.txt",
                 "cs_exp.txt",
                 "ip_stim.txt",
                 "ip_report.txt",
                 GFPower,
                 RS_N,
                 RS_T)
    port map (
              syndrome_calc_done => tb_syndrome_calc_done,
              errors_present => tb_errors_present,
              StartChien => tb_StartChien,
              lambda_poly => tb_lambda_poly((RS_T + 1) * GFPower - 1 downto 0),
              omega_poly => tb_omega_poly(RS_T * GFPower - 1 downto 0),
              syndrome => tb_syndrome(2 * GFPower * RS_T - 1 downto 0),
              reset_n => reset_n,
              clk => clk,
              blk_out => RS_Data_out(7 downto 0),
              blk_sync_out => RS_Data_out_Start);

  inst_RS_Decoder_top: RS_Decoder_top
    generic map (GFPower,
                 RS_N,
                 RS_T)
    port map (
              clk => clk,
              reset_n => reset_n,
              rs_data_in => rs_data_in(GFPower - 1 downto 0),
              rs_data_in_start => rs_data_in_start,
```

```
                RS_Data_out => RS_Data_out(7 downto 0),
                RS_Data_out_Start => RS_Data_out_Start,
                rs_enable => rs_enable,
                tb_errors_present => tb_errors_present,
                tb_lambda_poly => tb_lambda_poly((RS_T + 1) * GFPower - 1 downto 0 ),
                tb_omega_poly => tb_omega_poly(RS_T * GFPower - 1 downto 0),
                tb_StartChien => tb_StartChien,
                tb_syndrome => tb_syndrome(2 * GFPower * RS_T - 1 downto 0),
                tb_syndrome_calc_done => tb_syndrome_calc_done);
end RTL;
```