Low-Gate-Count

Error-and-Erasure-Correcting

Reed-Solomon (RS)

Encoder and Decoder

Preview Final Document

(Customized to Meet Customer's Requirements)

By

ECC Technologies, Inc. ("ECC Tek")
4750 Coventry Road East
Minnetonka, MN 55345-3909
www.ecctek.com
Phone: 952-935-2885
Fax: 952-935-2491
Email: phil.white@ecctek.com

© 2011 ECC Technologies, Inc. All Rights Reserved.

Notice

This document is a "Preview" document and does not contain material which ECC Tek considers confidential so everyone is free to copy and distribute this document without any obligations whatsoever to ECC Tek.

The Reed-Solomon (RS) designs described in this document, the copyright to this document, the copyright to the C source code and Verilog source code which simulate and implement the RS encoder and decoder designs, and US Patent Number 5,754,563 entitled "*Byte-Parallel System For Implementing Reed-Solomon Error-Correcting Codes*" (the "PRS Patent") protecting these designs are owned by

ECC Technologies, Inc. (ECC Tek)
4750 Coventry Road East
Minnetonka, MN 55345-3909

Phone: 952-935-2885
Fax: 952-935-2491
E-mail: phil.white@ecctek.com
Website: www.ecctek.com.

The RS encoder and decoder designs described in this document must be licensed from ECC Tek in order to legally implement them.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Revision History

01-21-05	pew	Created initial version of this document based on previous documents and drawings.
05-11-05	pew	Edited and added text and inserted drawings as Figures.
05-13-05	pew	Edited and added text and inserted drawings as Figures.
05-16-05	pew	Edited and added text and inserted drawings as Figures.
05-17-05	pew	Edited and added text and inserted drawings as Figures.
05-17-05	pew	Edited and added text and inserted drawings as Figures.
02-25-11	pew	Edited to bring it up to date for 8-bit symbols and to make it as similar as possible to the binary BCH documentation.
05-04-11	pew	Edited and added text.
05-04-11	pew	Added drawing for DECODERA-D architecture.
05-04-11	pew	Replaced error-only EBM algorithm flowchart with errors-plus-erasures flowchart.
05-05-11	pew	Restructured document, edited and added text, added how to use DECODERs A-D, and corrected figures.
05-05-11	pew	Corrected encoder section. Added table for decoder latencies.
05-11-11	pew	Added section on finite fields and Reed-Solomon codes.
05-11-11	pew	Added erasure locations drawing for SYN.
05-11-11	pew	Corrected text and Figures in PPU Section.
05-12-11	pew	Corrected EVAL Figure and text.
05-12-11	pew	Added L-CPU, V-CPU and X-CPU Figures.
05-12-11	pew	Added Section regarding initializing L(x) and V(x) in DECODERsA-D.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Sections

© 2011 ECC Technologies, Inc. All Rights Reserved.

Figures

© 2011 ECC Technologies, Inc. All Rights Reserved.

Tables

© 2011 ECC Technologies, Inc. All Rights Reserved.

1. Finite Fields and Reed-Solomon (RS) Codes

Very little knowledge of finite fields is needed to understand RS codes implemented in digital logic.

Finite fields exist with p^m elements where p is any prime number and m is any integer. For RS codes implemented in digital logic, we are only interested in finite fields where $p = 2$ so the number of elements is $2, 4, 8, ..., 64, 128, ..., etc.$. Each element is a set of m bits.

Every finite field has at least one primitive element. Most finite fields have many primitive elements, and in some finite fields all the elements except 0 and 1 are primitive. It is customary to call any primitive element α. When you multiply a primitive element by itself multiple times, it generates every nonzero element in the field.

In order for an error-correcting code to correct t symbol errors, $2t$ consecutive powers of α must be roots of the generator polynomial, $g(x)$. For Reed-Solomon codes,

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

Where i is the first root power which is usually 0 so the roots are

$1, \alpha, \alpha^2, \alpha^3, \alpha^4, ..., \alpha^{2t-1}$.

Assume $i = 0$.

RS encoders input a set of data symbols and output a set of codeword symbols. If you consider the output symbols to be coefficients of a codeword polynomial, $c(x)$, then every codeword polynomial is a multiple of $g(x)$ so that roots of $g(x)$ are also roots of $c(x)$.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Since $1, \alpha, \alpha^2, \alpha^3, \alpha^4, ..., \alpha^{2t-1}$ are roots of $c(x)$,

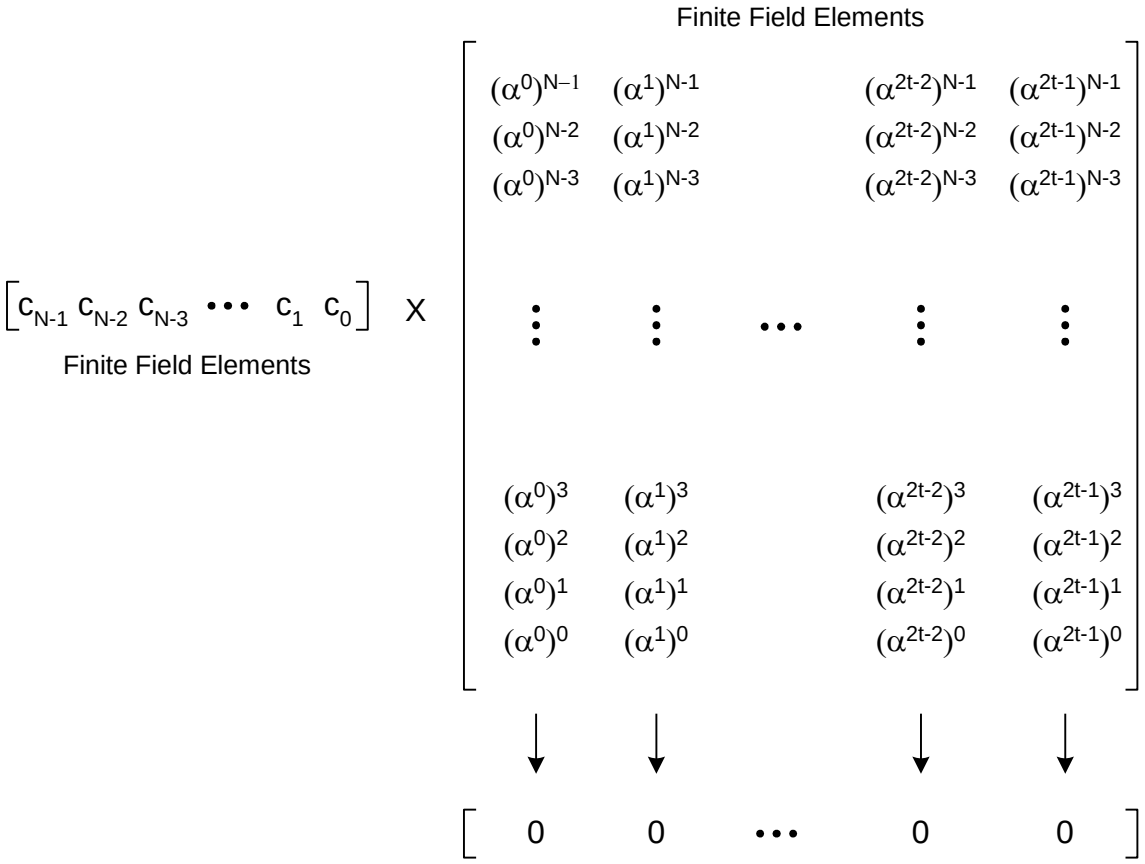


Figure 1 Codeword Multiplied by Parity-Check Matrix v1

© 2011 ECC Technologies, Inc. All Rights Reserved.

Or,

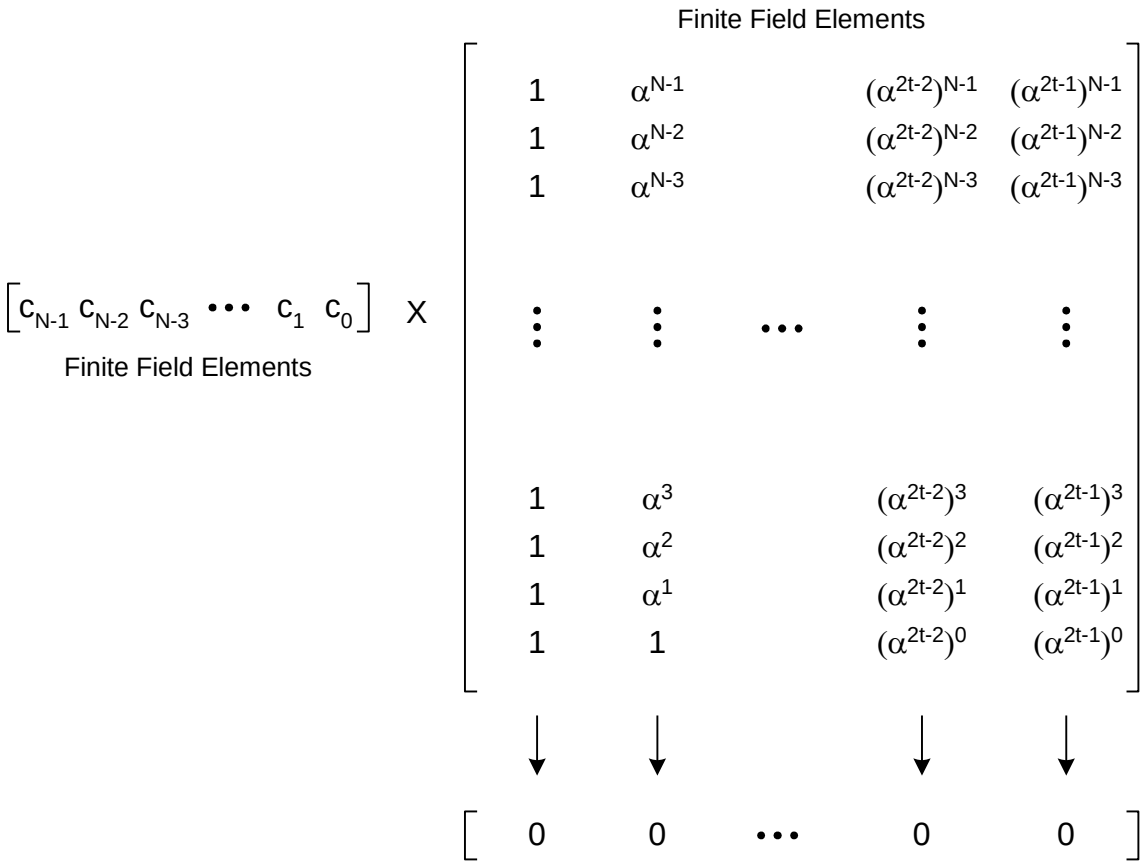


Figure 2 Codeword Multiplied by Parity-Check Matrix v2

© 2011 ECC Technologies, Inc. All Rights Reserved.

Now, let's call the elements in the second column the locations, L_i, of the codeword or received word symbols/coefficients as illustrated below where $r(x)$ is the received word/polynomial possibly with errors added and $S(x)$ is the syndrome.

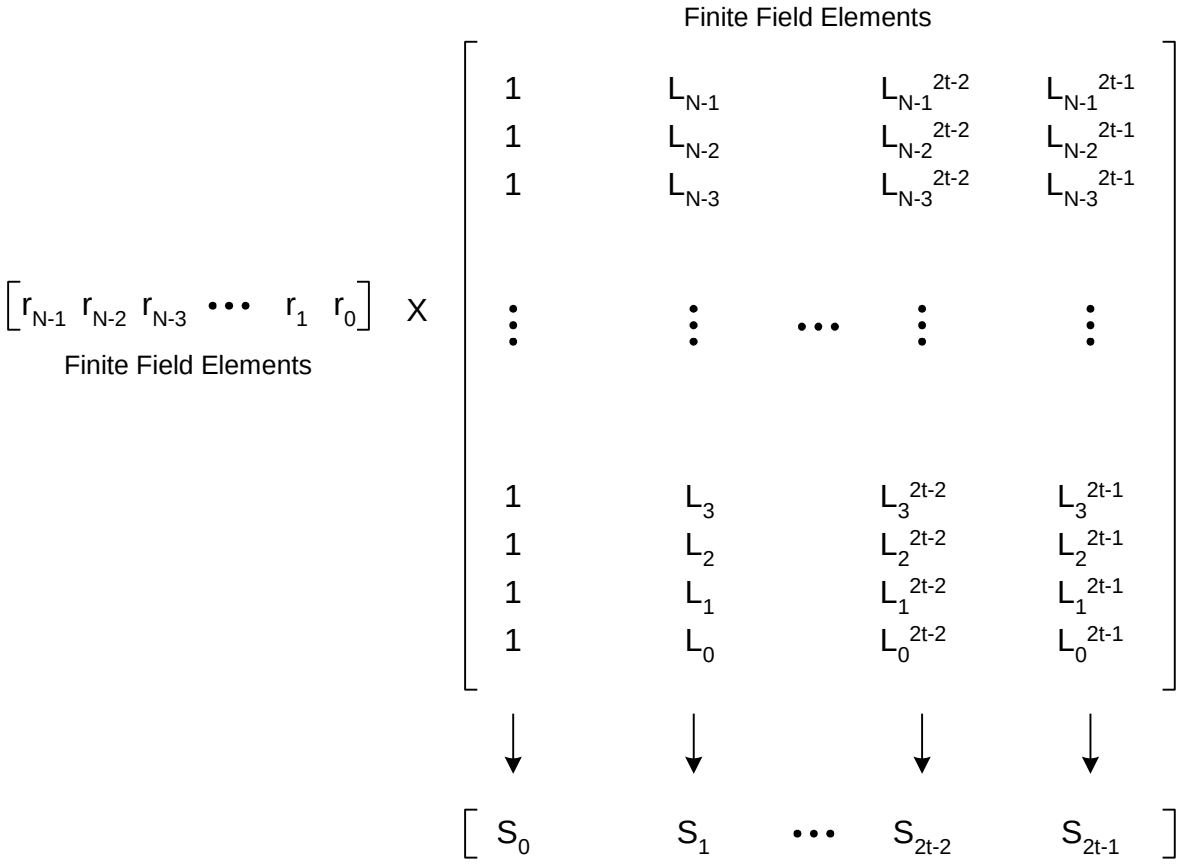


Figure 3 Received Word Multiplied by Parity Check Matrix

$$r(x) = c(x) + e(x)$$

Where $e(x)$ is the error polynomial or error pattern.

Since the syndrome for $c(x) = 0$, the syndrome of $r(x)$ only depends on the error pattern.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Suppose the error pattern has nonzero error magnitude values of V_a and V_b in locations, L_a and L_b.

Then,

(1) $S_0 = V_a + V_b$

(2) $S_1 = V_a L_a + V_b L_b$

(3) $S_2 = V_a L_a^2 + V_b L_b^2$

(4) $S_3 = V_a L_a^3 + V_b L_b^3$

etc.

With two errors, there are 4 unknowns so we need 4 simultaneous equations to solve for the 4 unknowns. If the locations of two errors are known, then we only need 2 simultaneous equations since there are only 2 unknowns. In general, to correct t errors, 2t redundant symbols are needed. To correct s erasures, s redundant symbols are required, and to simultaneously correct t errors and s erasures, 2t+s redundant symbols are required.

RS codes are "almost perfect" so only a minimum number of redundant symbols are required.

RS decoders find the unknown error location and error magnitude values. For erasures, the error locations are known and only the error magnitudes are unknown. It is possible for the error magnitude value to be 0 for an erasure, but not for an error.

It took many mathematicians many years to discover practical methods for solving the above simultaneous equations. The good news is that today a number of different practical methods exist that are easily implemented in digital logic. It is not easy to understand why the methods work, but it is easy to understand how to implement the methods. Manufacturers do not need to understand the mathematics of why methods work, they just need to be sure the methods work. Mathematicians have proved that the methods will always work.

ECC Tek makes use of Willard Eastman's Euclideanized version of the Berlekamp-Massey algorithm to find error locations and values.

Most RS and BCH decoders make use of what is called the key equation/congruency. An error locator polynomial, $L(x)$, and an error evaluator polynomial, $V(x)$ are defined and the key equation is

$$S(x)L(x) \equiv V(x) \bmod x^{2t}.$$

© 2011 ECC Technologies, Inc. All Rights Reserved.

The B-M algorithm finds the lowest degree $L(x)$ and $V(x)$ given $S(x)$ that satisfies the key congruency. *Mod* x^{2t} means the highest degree of $V(x)$ is $2t-1$. When the algorithm is finished, the roots of $L(x)$ indicate the locations of the errors and a simple formula is used to compute the values of the errors.

2. Overview of RS Encoder and Decoder Designs

This document describes digital logic designs for encoding and decoding a Reed-Solomon (RS) code which can correct t symbol errors and s erased symbols as long as $2t + s \leq 16$ in codewords of 118 symbols. The RS code being implemented uses K = 102 message (or data) symbols, N = 118 codeword symbols and R = 16 redundant parity check symbols.

An encoder overview is shown in Figure 4.

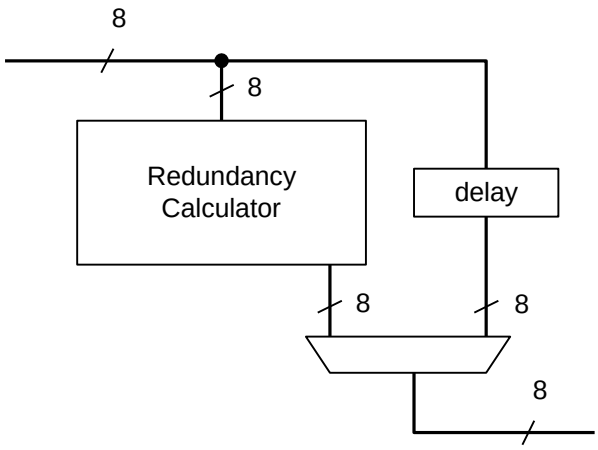


Figure 4 RS Encoder Overview

The encoder maps messages (or data words) which contain 102 symbols into codewords that contain 118 symbols by calculating and appending 16 symbols of redundancy.

Two versions of the encoder were created called ENCODER1 and ENCODER2.

© 2011 ECC Technologies, Inc. All Rights Reserved.

A decoder overview is shown in Figure 5.

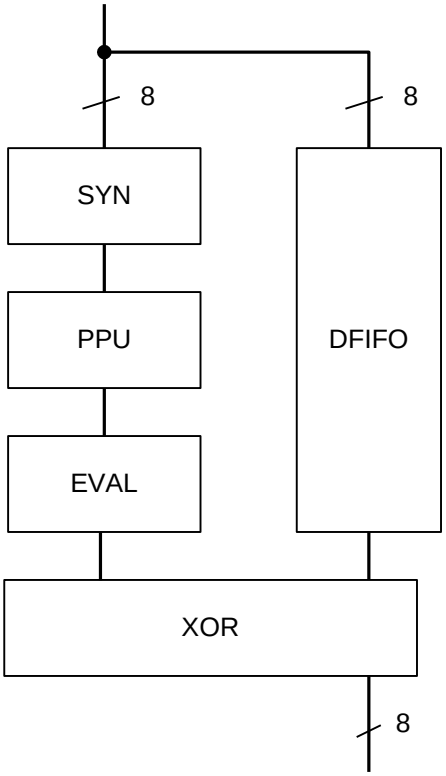


Figure 5 RS Decoder Overview

The decoder outputs a most-likely data word and error count indicating whether 0, 1, 2, … , 15 or 16 symbols have been corrected in the received word and if the decoder has failed because $2t + s > R$. If $2t + s \leq 16$, the most likely data word outputted by the decoder will be the correct original message (or data word).

In Figure 5, DFIFO stands for Data FIFO, SYN stands for Syndrome Generator, PPU stands for Polynomial Processing Unit, EVAL stands for Evaluator and XOR stands for Exclusive OR.

Five versions of the decoder were created called DECODER6, DECODERA, DECODERB, DECODERC and DECODERD. DECODERs A-D were created by modifying DECODER6 so this document will first describe DECODER6 and then describe how DECODER6 was modified to create DECODERs A-D.

3. Generator Polynomial

In order for an error-correcting code to simultaneously correct t errors and s erasures, 2t consecutive powers of a primitive finite field element α must be roots of g(x).

© 2011 ECC Technologies, Inc. All Rights Reserved.

Sixteen consecutive powers of the primitive element α = 0x02 are used to form the RS generator polynomial, g(x). g(x) has degree 16.

The generator polynomial,

$g(x) = (x-\alpha^0)(x-\alpha^1)(x-\alpha^2)\ldots(x-\alpha^{14})(x-\alpha^{15})$

$= g_0 + g_1x + g_2x^2 + g_3x^3 + \ldots + g_{14}x^{14} + g_{15}x^{15} + g_{16}x^{16}$

The coefficients are shown in the C code.

The "first root power" is equal to 0. The first root power can be any integer, but 0 is used in most standard RS schemes because it simplifies the decoder logic.

Any primitive element can be used as α. However, it is conventional to use "x" or 02 hex as the primitive element, so that is what is used in the RS designs.

4. Versions of the C Code and Verilog Code

The simulation and testing C code and the synthesizable Verilog code have been developed in versions.

Versions of the C software have suffixes V1, V2, V3, etc.

Verilog modules have module names with suffixes of 1, 2, 3, or A, B, C etc. The suffix is the version identifier. For example, the first version of SYND is called SYND1. Sometimes in this document a Verilog module is referred to, for example, as EVALAx. The "x" indicates all versions of EVALA.

5. Issues Involved in Licensing Synthesizable Verilog Code

ECC Tek is in the business of licensing synthesizable Verilog code which describes designs of ECC encoders and decoders at a high level. Licensing synthesizable Verilog code presents a unique set of challenges.

Since our Customers do synthesis, ECC Tek has no control over what synthesis tools are used or how they are used. ECC Tek does not know what its Customer's synthesis tools will do in optimizing logic or if one Customer's tools will do the same things or different things than another Customer's tools.

For example, suppose ECC Tek creates a number of Verilog modules so that the output of one module feeds the input of another. Assume we have "upstream" and "downstream" modules. If a downstream module forces a signal to a constant value, will the synthesis software, in all cases, be intelligent enough to know that certain upstream logic can be deleted because it is unneeded? ECC Tek has no way of knowing the answer to that question.

Another example is a Galois Field multiplier. If ECC Tek designs a GF multiplier to multiply two variables, will all of the synthesis software tools reduce and minimize the logic needed if one of the

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

inputs is a constant or should ECC Tek design specific multipliers to multiply by specific constants? ECC Tek has no way of knowing the answer to this question either. Most likely, most synthesis software will properly reduce the logic, but, because of that uncertainty, ECC Tek has created many different multipliers that multiply by a constant such as the ones delivered to Customer called MULT_BY_XX.

Gate counts and chip surface area required by these designs depend upon the circuit technology each licensee uses and the synthesis software they use.

6. Finite Field Generator Polynomial

The irreducible binary polynomial used to generate a finite field with 2^8 elements is

$p(x) = 11D \text{ hex} = 100011101 = x^8 + x^4 + x^3 + x^2 + 1.$

Any primitive element can be used as α, but it is conventional to use "x" or 0002 hex as the primitive element, so that is what is used.

7. What This Document Does

This document describes the final versions of a RS encoder and decoder which correct all error patterns with 8 or fewer symbols in error and is intended to help the reader understand the RS encoder and decoder hardware designs which are described in synthesizable Verilog and the C modeling software so that the designs can be easily modified and maintained.

8. What This Document Does Not Do

This document is not intended to be a substitute for a textbook on coding theory. It takes a lot of time, effort and study to understand algebraic coding theory. There are many good books on coding theory which should be consulted if the reader intends to gain an in-depth understanding of the math and theory associated with Reed-Solomon codes. It would be impossible for this document to cover all of the topics that a good book on coding theory covers so no attempt is made to do that.

This document does not provide mathematical proofs of mathematical results that have already been proven in textbooks or papers on coding theory. However, the C and Verilog models of the RS encoder and decoder designs can be used to verify that the RS system does, in fact, correctly correct all correctable error patterns. The C and Verilog models can also be used to determine the frequency of miscorrection.

This document presents and uses a version of the well-known Berlekamp-Massey ("BM") algorithm created by Willard Eastman (the "EBM" algorithm) to solve the key decoding equation without presenting a proof of its validity. The BM algorithm has been proven to be valid in various textbooks and papers on coding theory.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

The term "key equation" is more correctly called the "key congruency" but the term "key equation" is used in most of the textbooks on coding theory so it will be used in this document also. The key equation is $S(x)L(x) \equiv V(x) \bmod x^{2t}$ or, in words, S(x) times L(x) is congruent to V(x) modulo x^{2t} where S(x) is the syndrome polynomial, L(x) is the error locator polynomial and V(x) is the error evaluator polynomial. For the RS code described in this document, 2t = 16.

When operations are performed on polynomials modulo x^{16}, all of the terms resulting from a polynomial operation, such as multiplication of two polynomials, of the form $c_i x^i$ are 0 if i is greater than or equal to 16 because x^{16} is a factor of all terms of the form $c_i x^i$ when i is greater than or equal to 16 and x^{16} is congruent to 0.

It is well-known among coding theorists and stated in some textbooks on coding theory that the integer variable used in the EBM algorithm which ECC Tek refers to as "degLx" will be the actual degree of L(x) when decoding is successful. This fact is used to determine when the decoder has failed, but a mathematical proof of this fact not provided.

It has been proven in many textbooks that the degree of V(x) must be strictly less than the degree of L(x) when decoding is successful, and, if L(x) = 1, then V(x) must be 0. These facts are used to simplify the decoder design but, again, are not mathematically proven.

This document does not provide a detailed description of the C code because most of the C code is self-explanatory and also because the primary purpose of this document is to describe the hardware design – not the C software.

9. References

This document refers to other documents which are listed below:

1. Customer's "*RS Design Requirements Document*" referred to in this document as "**Customer's Requirements Document**"

2. A paper entitled *"Euclideanization of the Berlekamp-Massey Algorithm"* by Willard L. Eastman from Proceedings of the 1988 Tactical Communications Conference, Vol. 1 (1988) pp. 295-303. Referred to in this document as **"Eastman's Paper"**.

3. United States Patent Number 5,754,563 entitled *"Byte-Parallel System for Implementing Reed-Solomon Error-correcting Codes"*, May 19, 1998. Referred to in this document as **"the RS Patent"**.

4. Reuse Methodology Manual, Third printing 1999. Referred to in this document as "**the RMM**".

5. The book entitled *"Algebraic Coding Theory"*, written in 1968 by Elwyn Berlekamp. Referred to in this document as **"Berlekamp's book"**.

If there is a conflict or contradiction between what this documents states, indicates or implies and what the Verilog code is, the Verilog code should be taken to be the final authority.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

10. Notation

To simplify the notation used in this document, in the C code and in the Verilog code, English letters are used as symbols instead of Greek letters as much as possible.

A summary of the notation used by ECC Tek, Eastman and Berlekamp for the polynomials involved in solving the key decoding equation is given in Table 1.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Table 1 Comparison of Notation used by ECC Tek, Eastman and Berlekamp

	ECC Tek's Notation	Eastman's Notation (and Notation in the PRS Patent)	Berlekamp's Notation
Error Locator Polynomial	L(x)	$b^T(x)$, $\lambda(x)$ and $\Pi(x)$	$\sigma(z)$
Auxiliary Locator Polynomial	aL(x)	$b^O(x)$	$\tau(z)$
Error Evaluator Polynomial	V(x)	$p^T(x)$	$\omega(z)$
Auxiliary Evaluator Polynomial	aV(x)	$p^O(x)$	$\gamma(z)$
Last L(x) for Software Program	last_L(x)	$b^T(x)_{j-1}$	NA
Last V(x) for Software Program	last_V(x)	$p^T(x)_{j-1}$	NA
Number of Erasures	ne	μ	NA
Erasure Location i	eli	Wi	Xi
Current Discrepancy	dj	dj	$\Delta_1^{(k)}$
Last Nonzero Discrepancy	lnd	δ	NA
Number of Erasures Index	nei	λ	NA
Degree L(x) Variable	degLx	l	D(k)

© 2011 ECC Technologies, Inc. All Rights Reserved.

Table 2 defines symbols, abbreviations and terms used in this document, in ECC Tek's C and Verilog code, in Eastman's paper, in Berlekamp's book and in other papers and books on coding theory.

Table 2 Definition of Symbols, Abbreviations and Terms

aL(x)	In ECC Tek's notation, the auxiliary error Locator polynomial.
aV(x)	In ECC Tek's notation, auxiliary error eValuator polynomial.
BM	Berlekamp-Massey – Elwyn Berlekamp and James Massey who are credited with discovering or inventing an efficient algorithm for solving the key decoding equation for RS codes. There are numerous versions of the BM algorithm. The version presented in Eastman's paper is used in this document and referred to as the EBM algorithm.
c[i]	In ECC Tek's C code, the codeword symbol in position i.
ci	In ECC Tek's Verilog code, the codeword symbol in position i.
dj	The current discrepancy. In this document, the current discrepancy = Vj where j is the BM algorithm iteration number. If the current L(x) being calculated in the decoder pipeline is the correct L(x), the discrepancy will be 0 from this point until the end of the Berlekamp-Massey algorithm is reached.
degLx	The degree of L(x). The variable degLx is the maximum degree – or an upper bound on the degree of L(x) – while L(x) is being formed from S(x) using the BM algorithm. Once L(x) has been formed from S(x), the degLx variable is the actual degree of L(x). The degLx variable is used by the final stages of the decoder to determine if the error pattern calculated by the decoder is legitimate.
el[i]	In the C code, erasure location i.
eli	In the Verilog code, erasure location i.
g(x)	The generator polynomial for an algebraic block code such as a RS code. $g(x) = g_0 + g_1x+g_2x^2=\ldots+g_Rx^R$ where R is the number of redundant symbols. g_R is always 1.
gLx[]	In the C code, a global variable for storing the coefficients of L(x).
K	The number of symbols in each message or data field.

© 2011 ECC Technologies, Inc. All Rights Reserved.

L[i]	In the C code, the coefficient of L(x) in position i.
Li	In the Verilog code, the coefficient of L(x) in position i.
Li-1	In the Verilog code, the coefficient of L(x) in position (i-1).
lnd	In the Verilog and C code, the last nonzero discrepancy calculated while executing the BM algorithm. The lnd variable is initialized to 1.
Lp[i]	In the C code, the coefficient in position i of the polynomial which is the formal derivative of the error locator polynomial. The formal derivative of L(x) is represented as L'(x) or Lpx or Lp(x).
Lpx_eval	The evaluation of Lp(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation Lp(x) at "α^{-i}" is denoted as Lpx_eval or Lp(α^{-i}).
Lpxi	In the Verilog code, the coefficient in the polynomial which is the formal derivative of the error locator polynomial which is in the Th position.
Lx_eval	The evaluation of L(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation L(x) at "α^{-i}" is denoted as Lx_eval or L(α^{-i}).
mlci	most likely codeword symbol in the Th position in the Verilog code.
mle[i]	In the C code, the most likely error symbol (symbol) in position i.
mlei	In the Verilog code, the most likely error pattern symbol in position i.
mlmi	In the Verilog code, the most likely message or data symbol i.
N	The number of symbols (symbols) in a codeword.
nei	The "number of erasures initiated" index.
R	The number of redundant symbols (symbols) in a codeword. For the RS code described in this document, R = 5.

© 2011 ECC Technologies, Inc. All Rights Reserved.

r[i]	In the C code, the received symbol in position i.
ri	In the Verilog code, the received symbol in position i.
RS	Reed-Solomon – Dr. Irving Reed and Dr. Gustave Solomon who are credited with discovering RS codes in 1960.
s	The number of erasures the code can correct.
S(x)	The syndrome polynomial.
S[i]	In the C code, the coefficient of S(x) in position i.
Si	In the Verilog code, the coefficient of S(x) in position i.
t	In most books on coding theory, the number of errors a block code can correct.
Vi	In the Verilog code, the coefficient of V(x) in position i.
Vi-1	In the Verilog code, the coefficient of V(x) in position (i-1).
Vx_eval	The evaluation of V(x) at some specific value for x. An evaluation is the value obtained by replacing the indeterminate "x" in a polynomial with an actual finite field element and evaluating the resulting expression. The evaluation V(x) at "α^{-i}" is denoted as Vx_eval or V(α^{-i}).
Wj	In Eastman's paper and in some versions of the C code, the jth erasure location. The W can be thought of as standing for the W in "Where".
Π(x)	In Eastman's paper, the error locator polynomial. Π probably stands for "position".
Ω(x)	In Eastman's paper, the error evaluator or error magnitude polynomial. Omega – where "m" stands for magnitude.
α	A primitive element of a finite field. In this document $\alpha = 2$ which is one of many primitive elements in a finite field with 1024 elements.

© 2011 ECC Technologies, Inc. All Rights Reserved.

γ(z)	In Berlekamp's book, the auxiliary error evaluator or auxiliary error magnitude polynomial which ECC Tek refers to as aV(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish between binary and nonbinary polynomials.
μ	In Eastman's paper, the number of erasures.
τ(z)	In Berlekamp's book, the auxiliary error locator polynomial which ECC Tek refers to as aL(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish binary and nonbinary polynomials.
ω(z)	In Berlekamp's book, the error evaluator or error magnitude polynomial which ECC Tek refers to as V(x). Berlekamp uses the indeterminate "z" instead of "x" for polynomials with nonbinary coefficients I believe so that the reader can quickly distinguish between binary and nonbinary polynomials.

11. C Code

The C modeling program is used to test the encoder and decoder algorithms and to generate input data and expected results files for debugging the Verilog code. The C code can be used to see if the decoder correctly corrects correctable error patterns and to determine the probability of miscorrection when the number of errors in a received word exceeds the capability of the code to correct.

The WriteFiles() function in the C program writes testbench files. The other functions in the C program are used to test the encoder and decoder algorithms.

After being compiled with a standard ANSI C compiler, the C program should run on any PC or Mac with no alteration.

The C program is very basic. Printf() statements can be inserted at various points to view variables as the program executes.

Many of the variables in the C program are global variables which simplifies modification of the functions. The message array, m[i], the codeword array, c[i], and the most likely error and codeword arrays, mle[i] and mlc[i], are some of the primary global variables. Normally global variables are undesirable in software, but, in this case, they are helpful because different functions can operate on the same data without having to pass arguments from function to function. Different encoding, decoding and testing functions can be selected by calls from main() to perform various operations. Each possible encoding function assigns values to the global codeword array, c[i]. Errors can be added to the

© 2011 ECC Technologies, Inc. All Rights Reserved.

codeword c[i] array to create the received word r[i] array. Then any selected decoding function can be used to decode the r[i] array and generate the most likely error and codeword arrays, mle[i] and mlc[i].

The primary functions in the C program are the Encode() and Decode() functions. Other testing and multiplication functions allow the Encode() and Decode() functions to be tested individually or tied together and tested as a system.

There are three read-only tables in the programs. Each of the tables is initialized by main(). Although algorithms could be used instead of tables, tables are much faster and allow many more error patterns to be tested in the same amount of time than what could be tested if algorithms were used.

The first table is a list of finite field reciprocals. When a reciprocal of a finite field element is needed, this table is accessed.

The second table is a table of powers of a primitive element, α^i, where i is the index or address of a one-dimensional array and α^i is the value stored in the table or array at that address. This table is used by the decoder to quickly determine α^i given i where i = 0x00, 0x01,...,oxff.

The third table is a table of negative powers of a primitive element, α^{-i}, where i is the index or address of a one-dimensional array and α^{-i} is the value stored at that address or location. This table is used by the decoder to quickly determine α^{-i} given i. Received word and codeword locations are labeled as α^{-i} where i runs from 0x00 to 0xff.

The Decode() function determines the most likely error word, mle[i], and the most likely codeword, mlc[i], for a particular received word, r[i].

When the Decode() function detects an abnormal condition, it sets the decode_Fail flag to TRUE (or 1) indicating that the decoding operation has failed. The decode Fail flag is used to detect error patterns that exceed the capability of the decoder to correct.

Some error patterns that exceed the correction capability of the code will either not be detected or will be miscorrected. There is no way around this fact. All decoders for algebraic block codes have this property. If a severe error pattern happens to also be a codeword, no decoder can detect it or correctly correct the received word because the error pattern will map one legitimate codeword into another legitimate codeword and the syndrome will be all zeros indicating no errors.

The probability of miscorrection can be measured and quantified by using the C testing code. Error patterns can be generated that exceed the correction capability of the code and then the Decode() function can be executed to determine the frequency of miscorrection.

11.1. C Functions for Generating Verilog Input Data and Expected Results Files

The WriteFiles() function generates Verilog input data and expected results files for use with testbenches.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Actual results generated by running Verilog simulations are compared with the contents of the expected results files generated by the WriteFiles() function. If the Verilog simulation results are correct, they will match the results generated by the WriteFiles() function.

11.2. C Functions for Testing the Encoder, Decoder and System

The testing functions are useful for verifying that the decoder can correctly correct all correctable error patterns and for determining how frequently the decoder will declare a decoding failure when error patterns exceed the correction capability of the code. For example, the testing C code can be used to determine the probability of the decoder declaring a failure conditioned on the fact that a 9-symbol error pattern has occurred.

There are a number of error pattern generator functions in the C code that can be used to generate error patterns with a particular number of errors.

Theoretically, it is possible to exhaustively test most of the decoder by assuming that an all-zero message has been encoded into an all-zero codeword because, if the syndrome generator is working correctly, the syndrome does not depend upon what message or codeword has been sent. The syndrome only depends upon the error pattern so, once it has been determined that the syndrome generator is working correctly, the remainder of the decoder can be tested by assuming an all-zero codeword was sent. Most of the C functions that generate error patterns use the error pattern generated to be the received word and assume that an all-zero codeword was sent. If the user does not feel comfortable assuming an all-zero codeword was sent, then random messages can be generated, encoded into codewords and the error pattern can be added to the codeword to form the received word.

The C functions included in the C program are listed in Table 3 with a brief description of what they do.

Table 3 C Functions Included in the C Code and What They Do

Decode()	Determines the most likely code word, mlc, for a particular received word, r.
Encode()	Generates a code word, c, from a message, m.
Initialize_alphaToThe()	Initializes the α^i table.
Initialize_alphaToTheMinus()	Initializes the α^{-i} table.
Initialize_Mult()	Initializes the finite field multiplication table.
Initialize_Recip()	Initializes the table of reciprocals.
Manually_Test_System()	Manually tests the encoder and decoder system.
Multiply()	Multiplies a_of_x by b_of_x to form c_of_x. a_of_x, b_of_x and c_of_x are global variables.

© 2011 ECC Technologies, Inc. All Rights Reserved.

SystemTest2()	Tests the encoder and decoder system.
Test_Encoder()	Tests the encoder.
TestR_Decode()	Test decoder's ability to correct equally probable random error patterns.
TestRn_Decode()	Test decoder's ability to correct n symbol/symbol random error patterns.
WriteFiles()	Writes Verilog input data and expected results testbench files.

The SystemTest() function tests both the encoder and decoder as a system. A message is generated at random and then encoded into a codeword. Errors are added to the codeword to form the received word and the received word is decoded to see if the decoder corrects the errors correctly. The decoder test functions are used to test the decoder against error patterns with 1 symbol in error, 2 symbols in error, 3 symbols in error, 4 symbols in error, 5 symbols in error, 8 symbols in error and a random number of bytes in error. For the RS code, all single, double and triple symbol error patterns have been tested.

It is not possible to exhaustive test decoders which correct many symbols when codewords are long because there are too many possible error patterns.

The purpose of the TestR() functions is to determine the probability of miscorrection when the error pattern contains a large number of random errors.

Most of the decoder test functions assume the codeword is the all-zeros codeword. A nonzero received word is created by the test function for the decoder to decode. Each received word, r[i], created by the test function has a nonzero symbol value (Value 1 = V1) in Location 1 (L1). There also may be L2, V2, L3, V3, etc. symbols.

The WriteFiles() function writes input data and expected results files for use with testbenches.

12. Introduction to the Hardware Designs Described in Verilog

ECC Tek's hardware design strategy is to use only a very small number of basic Verilog language constructs as building blocks in developing RS encoder and decoder designs to ensure that the designs will be synthesizable and will achieve a high level of performance.

The basic Verilog constructs used by ECC Tek are as follows:

- Constructs that ECC Tek knows will synthesize into Registers
- Constructs that ECC Tek knows will synthesize into Multiplexers

© 2011 ECC Technologies, Inc. All Rights Reserved.

- Constructs that ECC Tek knows will synthesize into State Machines
- Constructs that ECC Tek knows will synthesize into Simple Combinatorial Logic

ECC Tek does not view Verilog as a programming language such as C, but as a way to easily simulate and synthesize circuits that have been previously laid out in block diagrams as shown in the Figures of this document.

The way ECC Tek designs digital logic is by first creating block diagrams (pictures) of the data flow required for a particular design to achieve a desired level of performance. For example, most of the block diagrams shown in the Figures in this document were created before any of the Verilog code was written, and the resulting performance of the encoder and decoder was predicted based upon previous experience in designing encoders and decoders.

Writing and debugging the Verilog code was the last step in the design process.

By adhering to this discipline, ECC Tek is able to quickly create and debug designs that can be customized to each Customer's unique requirements.

It is ECC Tek's belief that, if ECC Tek did not develop designs using only a few well-proven constructs, then the time to finish a design and the risk of failure would be much higher than it is using this design strategy.

In addition to the benefits of reducing development time and development risk, this design strategy also results in designs that are easy-to-understand, easy-to-modify and easy-to-maintain by the Customer.

The symbol used in the drawings for a register, R, the associated Verilog code and the circuit ECC Tek knows will be synthesized from the Verilog code are illustrated in Figure 6.

© 2011 ECC Technologies, Inc. All Rights Reserved.

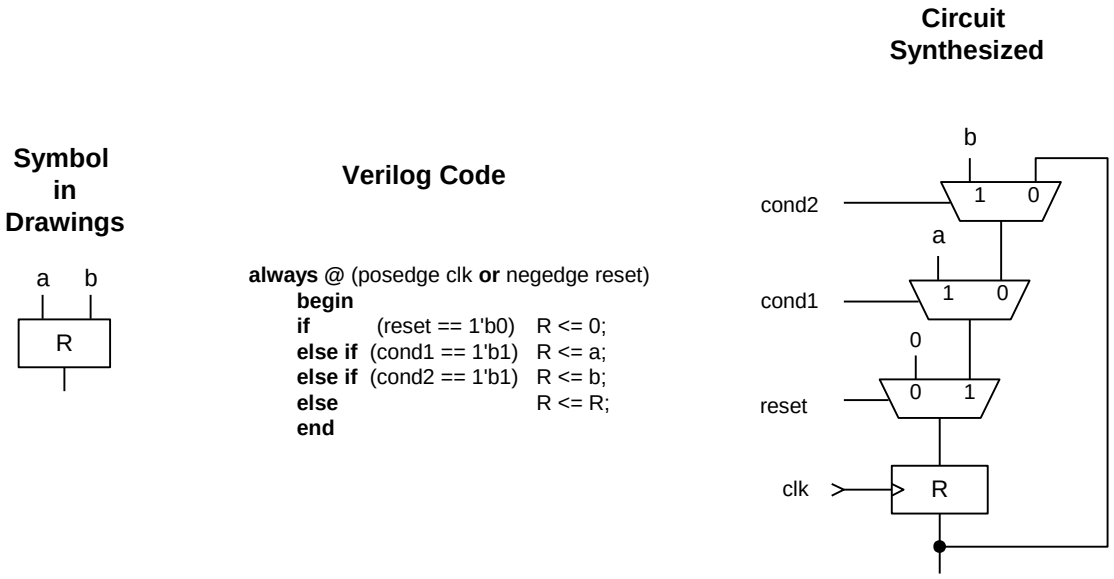


Figure 6 Registers

The symbol used in the drawings to represent a multiplexer (mux), the associated Verilog code and the circuit ECC Tek knows the Verilog code will be synthesized into are shown in Figure 7.

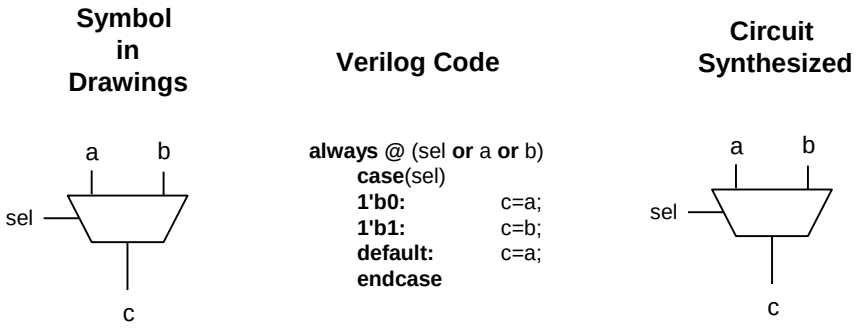


Figure 7 Multiplexers (Muxes)

State Machines combine a state register with a set of muxes and ECC Tek knows the Verilog code will synthesize into a circuit of the form shown in Figure 8. A sample of the Verilog code for implementing a state machine can be seen by looking at the SM Verilog code in the PPU module, for example.

© 2011 ECC Technologies, Inc. All Rights Reserved.

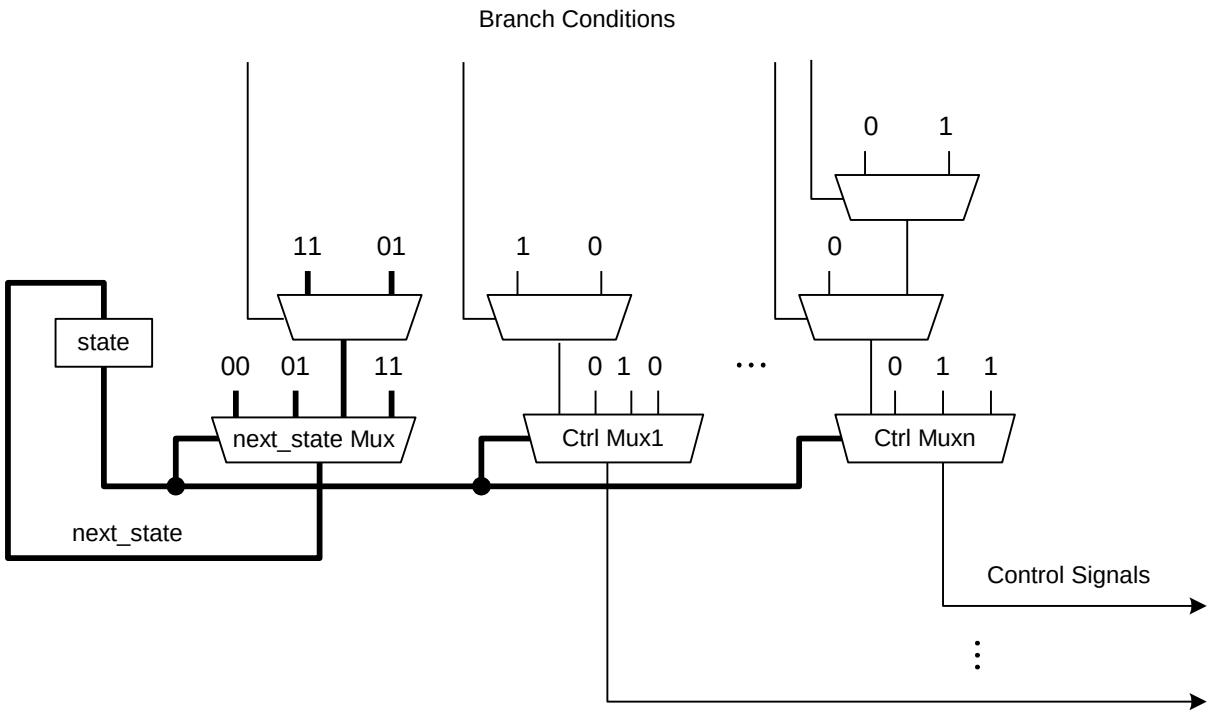


Figure 8 Generic State Machine

A generic control unit is shown in Figure 9. It consists of a number of registers (often up and down counters) used for control purposes and, usually, a state machine ("SM") to implement the control logic. In come cases the control is done completely by combinatorial logic and there is no state machine. The CONVERT and SYN modules have no state machine – only combinatorial logic to control the flow of data.

© 2011 ECC Technologies, Inc. All Rights Reserved.

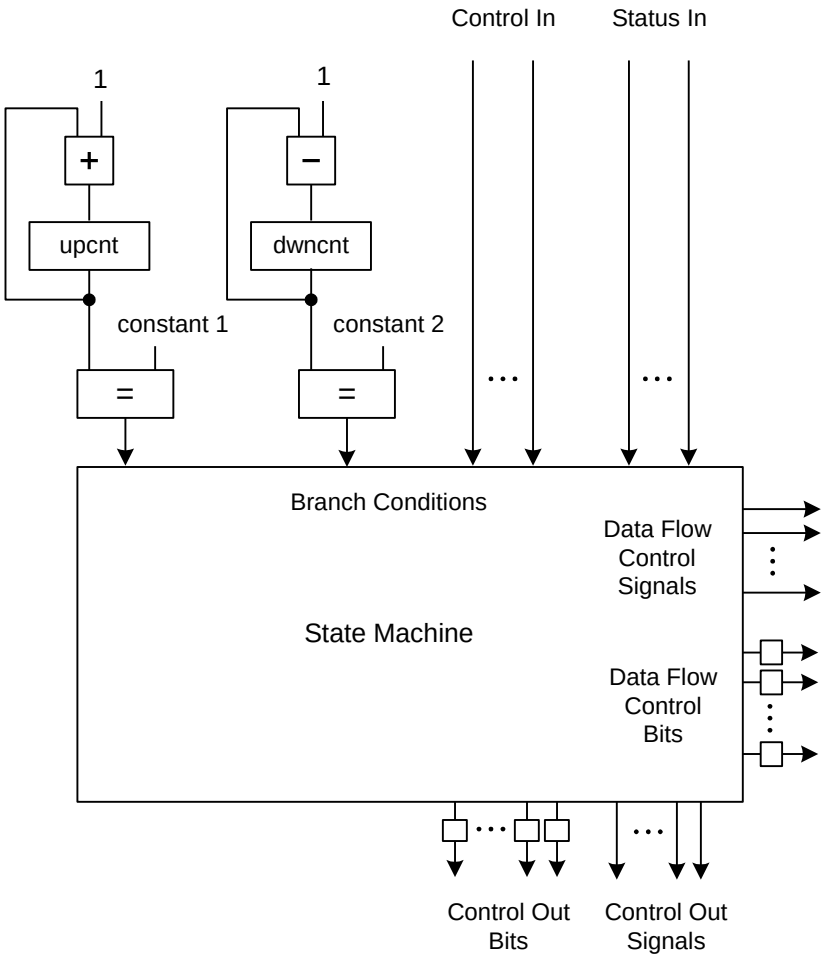


Figure 9 Generic Control Unit with a State Machine

13. Verilog Coding Style and Signal Naming Conventions

The encoder and decoder Verilog code has been written to, for the most part, comply with the Verilog coding style guidelines recommended in the RMM.

The Verilog code is written in a synthesizable form so that synthesizer software can be used to synthesize the actual circuits. This makes the RS designs independent of IC vendor.

The Verilog code can be synthesized to be implemented in an ASIC, a structured ASIC or in an FPGA.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Each Verilog module is in a separate file. The names of the files are the same as the names of the modules they contain with the addition of the ".v" filename extension. All filenames use primarily capital letters. For example, the file named "MULT1.v" contains the Verilog module named MULT1.

All signal names in Verilog use lowercase letters.

All Verilog module inputs are prefixed with "i_" to indicate "input" and all Verilog module outputs are prefixed with "o_" for "output".

The Verilog code is written in a style that makes it easy to spot errors and make changes. All input and output ports to all of the modules are on separate lines and all register and wire type variable declarations are on separate lines.

14. General Hardware Design Development Methodology

The design methodology used to develop the RS encoder and decoder hardware designs was to develop them in "steps" as follows:

1. Use a variation of the well-proven and efficient Berlekamp-Massey ("BM") algorithm for decoding. ECC Tek uses one of many variations of the BM algorithm for decoding as described by Willard Eastman in Eastman's paper.

2. Implement encoder and decoder algorithms in C to prove they work correctly and to generate input data and expected results files for debugging the Verilog code.

3. Thoroughly test the C models.

4. Create an initial "skeleton" version of the decoder that works, but does not have all of the final features required.

5. Add features to the "skeleton" version of the decoder to create a series of decoder versions until all of the required features have been successfully implemented. Each version is one "step". If we are unsuccessful in implementing a "next step", then we can always go back to the previously successful last step and try again.

6. Repeat the above development process for the encoder.

7. Use the C models and Verilog testbenches to debug and validate the designs.

15. Basic Logic Design Concepts Used by ECC Tek

All of the blocks of logic in the encoder and decoder can be divided into two parts – one part is the "control structure", "control unit" or just "control" and the other part is the "data flow structure", "data flow unit" or just "data flow" as illustrated in Figure 10. The control unit receives status signals from

© 2011 ECC Technologies, Inc. All Rights Reserved.

the data flow unit and sends control signals to the data flow unit to control the flow of data. Control signals are generally fed into and out of the control unit and data is fed into and out of the data flow unit.

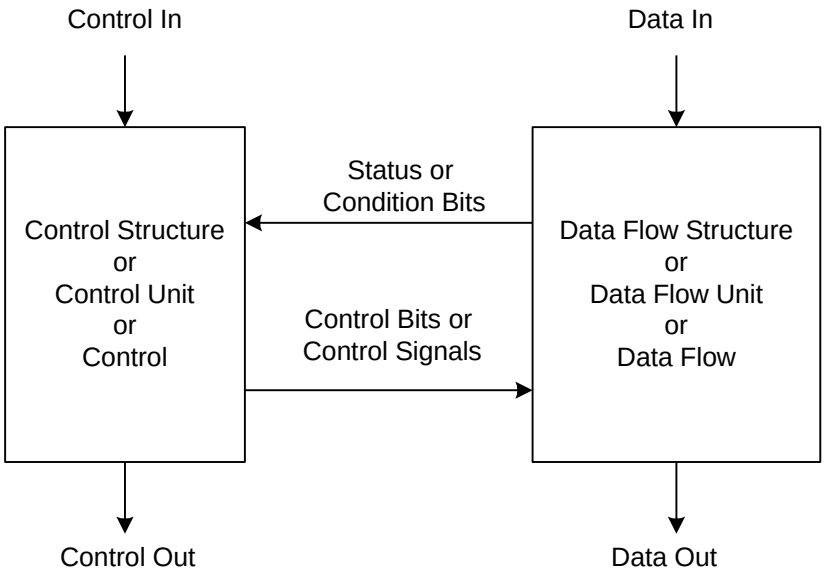


Figure 10 General Form for all Logic Blocks

State machines are described in Verilog in a standard format as described in the RMM so that the Verilog code for a SM will synthesize into a logic circuit of the form shown in Figure 8.

Probably the best way to understand the SM logic is to look at the Verilog code. State diagrams can also be drawn from the Verilog code, but viewing the Verilog code is how ECC Tek creates the SM. The Verilog statements are very easy to follow and understand. All of the state machines used in the encoder and decoder are very simple and usually only have a few states.

16. Drawing Conventions

In the drawings, the data flow is usually top down.

The control unit for a block of logic is usually drawn to the left of the data flow unit.

All Galois Field full multipliers are drawn with a dot (“.”) inside a box. Partial Galois Field multipliers are drawn with a star (“*”) inside a box.

© 2011 ECC Technologies, Inc. All Rights Reserved.

17. Implementing Finite Field Operations in Digital Logic

Both the encoder and the decoder perform operations in a finite (or Galois) field with 1024 elements where each element is a 10-bit-wide quantity referred to in this document as a "symbol". The irreducible and primitive binary polynomial used to generate the finite field is p(x)= 0x11D.

$p(x) = x^8 + x^4 + x^3 + x^2 + 1 = 0x11D.$

The finite field operations used for encoding and decoding RS codes are as follows:

- addition of two variable finite field elements,
- multiplication of a variable finite field element by a constant finite field element,
- multiplication of two variable finite field elements, and
- finding the reciprocal (or inverse) of a finite field element.

Addition of finite field elements is a bit-wise exclusive-or (XOR) operation.

The finite field multiplier that multiplies two variable finite field elements is a Verilog module called MULT1. MULT1 is a highly structured full multiplier as described in the RS Patent and illustrated in the top part of Figure 11.

If n multipliers are used and one of the inputs is common to all n multipliers, then the common multiplier logic that would normally be implemented inside each multiplier can be pulled out and implemented outside the multiplier only one time rather than n times. ECC Tek calls the resulting multiplier with the missing piece a "partial" multiplier. Partial multipliers are illustrated in bottom part of Figure 11.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

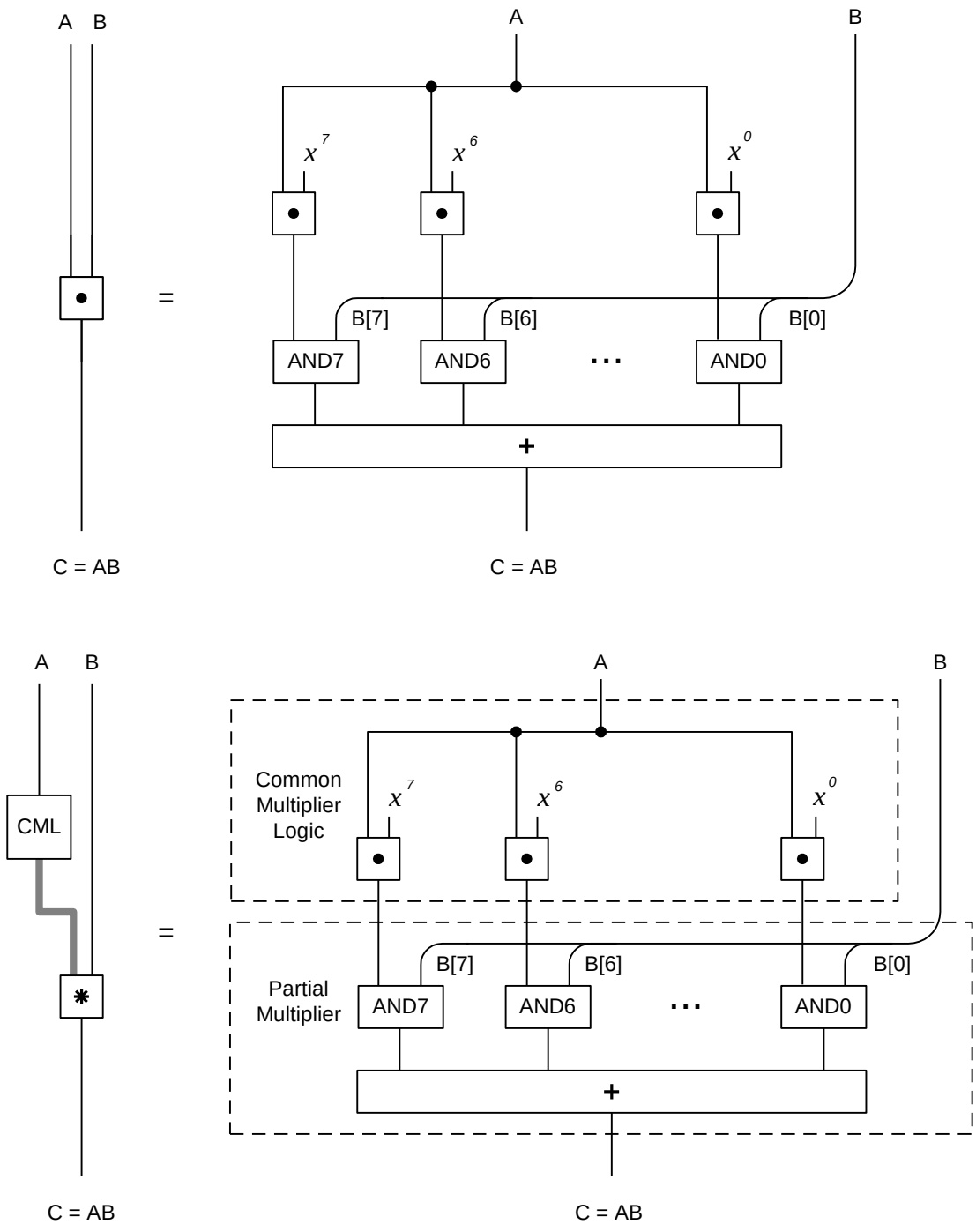


Figure 11 Full Multiplier (top) and Partial Multiplier (bottom)

If an intelligent synthesizer is used, the synthesizer should also recognize that part of the multiplier is common and the result of synthesizing full multipliers and partial multipliers should be the same. Since the synthesis result is synthesizer-dependent, ECC Tek provides both full and partial multiplier versions of the Verilog modules which require multiple multipliers.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Multipliers that multiply a variable finite field element by a constant field element are implemented as shown in Figure 12. The constant, B, determines the connections in Figure 12.

Multipliers that multiply a variable by a constant are separate Verilog modules. The Verilog module that multiplies by 0ab is called MULT_BY_0ab.

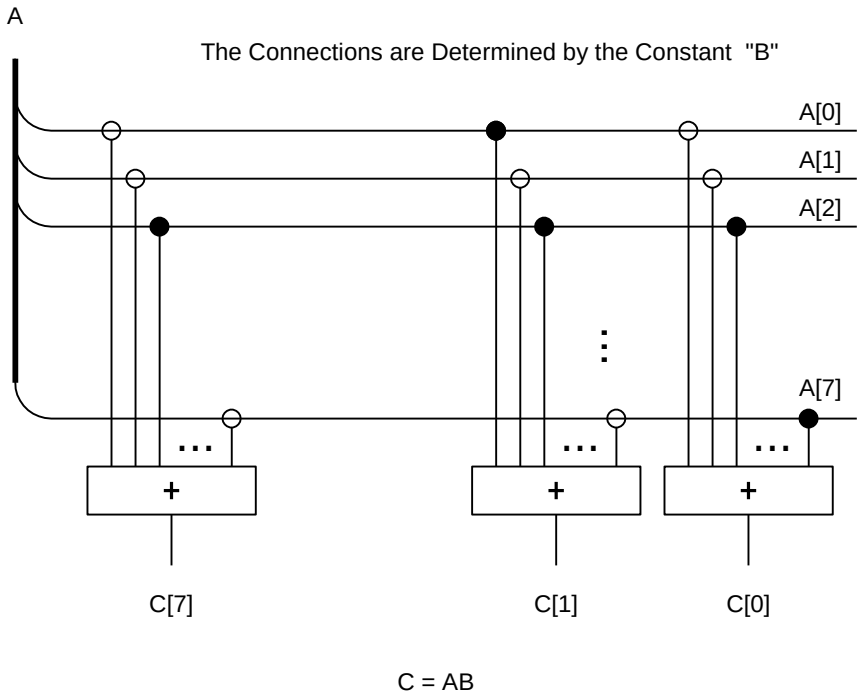


Figure 12 Multiplier that Multiplies the Variable "A" by the Constant "B"

Reciprocals of finite field elements are stored in a ROM. Only one reciprocal ROM is needed.

18. ENCODER2

A high-level block diagram of the encoder is shown in Figure 13.

© 2011 ECC Technologies, Inc. All Rights Reserved.

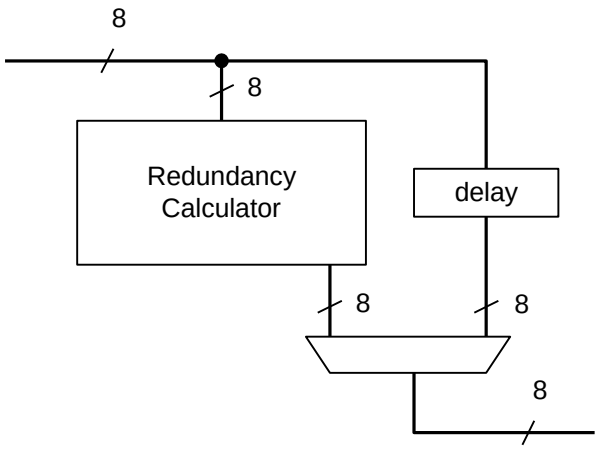


Figure 13 High-Level Block Diagram of Encoder

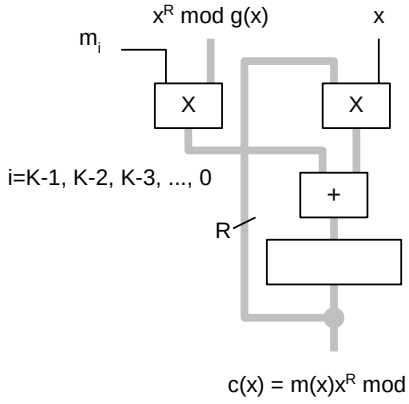
The encoder multiplies m(x) by x^R, divides x^R m(x) by g(x), finds the remainder, and appends the remainder onto m(x) as the redundancy. The mathematics involved in encoding is shown in Figure 14.

Figure 14 Encoder Mathematics

The message symbols are shown as a row vector that is multiplied by a generator matrix. Multiplication of the row vector by the generator matrix results in generating the required redundancy which is $m(x)\ x^R \bmod g(x)$.

An iterative circuit for encoding is shown in Figure 15.

© 2011 ECC Technologies, Inc. All Rights Reserved.



$$m(x)x^R \bmod g(x) = \left(\left(\left(\bullet\bullet\bullet \left(m_{K-1}\, x^R \bmod g(x) \right) x + m_{K-2}\, x^R \bmod g(x) \right) x + m_{K-3}\, x^R \bmod g(x) \right) x + \bullet\bullet\bullet + m_0\, x^R \bmod g(x) \right)$$

Figure 15 Encoder Iterative Circuit

An overview block diagram of the encoder was shown in Figure 13. A more detailed data flow diagram for the encoder is shown in Figure 16.

Figure 16 Encoder Data Flow

Internally, the encoder receives messages of K symbols labeled m_0, m_1, m_2, …, m_{K-1} (m_{K-1} is received first) serially and appends R redundant symbols c_0, c_1, c_2, …, c_{R-1} onto each message to form an N-symbol codeword c_0, c_1, c_2, …, c_{N-1}.

© 2011 ECC Technologies, Inc. All Rights Reserved.

19. ENCODER Signal Definitions

Input signals have a prefix of "i_". Output signals have a prefix of "o_". For example, i_symbol_valid is an input signal and o_symbol_valid is an output signal.

Signal Name	Description
i_rst	Asynchronous reset, active low.
i_clk	Synchronous clock, active on rising edge.
i_byte[7:0]	Input byte.
i_byte_valid	Input byte valid.
o_byte[7:0]	Output byte.
o_byte_valid	Output byte valid.

Table 4 ENCODER Signal Definitions

© 2011 ECC Technologies, Inc. All Rights Reserved.

20. Decoder Latency

All of the decoders described in this document have a fixed latency.

Latency is defined as the number of clock cycles from the last input byte received by the decoder to the first byte outputted.

DECODER	Latency
DECODER6	97
DECODERA	419
DECODERB	691
DECODERC	419
DECODERD	691

Table 5 Decoder Latencies

For DECODERA-D, the next input cannot be received until after the current data is output which is 118 clock cycles more than the latency. For DECODERA and DECODERC, there must be a gap between input records of 419+118=537 clock cycles. For DECODERB and DECODERD, there must be a gap between input records of 691+118=809 clock cycles.

21. DECODER6

A top-level block diagram overview of DECODER6 was shown in Figure 5 and is repeated below as Figure 17 . The following sections explain the components of the decoder.

© 2011 ECC Technologies, Inc. All Rights Reserved.

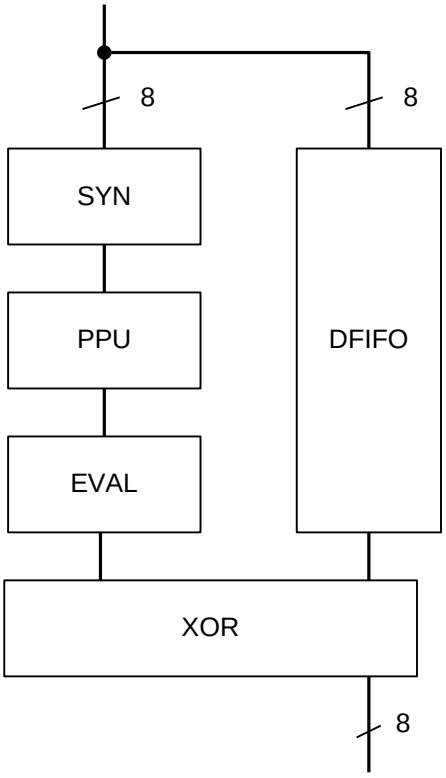


Figure 17 Decoder Block Diagram

22. Decoder Clocking Options

22.1. Decoder Clocks

Most of ECC Tek's previous decoder designs have used only one clock, but it is possible to have separate and independent clocks for the input logic, the PPU and the output logic. If the clocks are not synchronized, then logic would have to be added to resynchronize signals that crossed clock domains.

Clocking the PPU at a higher rate reduces the decoder latency/delay and reduces the DFIFO size needed.

DECODER6 has a clock input and a PPU_clock input. The PPU_clk must be 3x the clock input in order for DECODER6 to handle continuous input with no gaps between input records. The DECODER6 will work OK with a slower PPU_clk, but will not be able to handle continuous data input.

Clocking the output with a separate clock allows the output to be paused by gating the clock.

© 2011 ECC Technologies, Inc. All Rights Reserved.

22.2. Gating the Clocks

ECC Tek normally operates under the assumption that gating the clock is not acceptable. If gating the clock is acceptable, a number of options are opened up such as not clocking either the encoder or decoder if they are not being used and pausing the output by gating the output clock.

© 2011 ECC Technologies, Inc. All Rights Reserved.

23. DECODER Signal Definitions

Input signals have a prefix of "i_". Output signals have a prefix of "o_". For example, i_symbol_valid is an input signal and o_symbol_valid is an output signal.

Signal Name	Description
i_rst	Asynchronous reset, active low
i_clk	Synchronous clock, active on rising edge
i_byte[7:0]	Input byte
i_byte_valid	Input byte valid
o_byte_valid	Output byte valid
o_byte[7:0]	Corrected output byte
o_decode_fail	Decoder failed
o_error_count[4:0]	Number of errors corrected in a page. When o_last_byte is asserted, the o_error_count[4:0] is updated for the current page
o_error_cnt_valid	Error count valid

Table 6 DECODER Signal Definitions

24. DFIFO

Normally, ECC Tek's licensees replace ECC Tek's DFIFO with their own.

The DFIFO, as implemented by ECC Tek, is configured to write the first K bytes of data to a FIFO and skip the redundant symbols. The DFIFO module contains a FIFO component as illustrated in Figure 18. For decoder designs that allow for the pausing of the output, a register as shown in Figure 18 is needed to store the output of the FIFO immediately after the i_pause signal is asserted otherwise one input byte will get lost when the operation of the decoder is resumed after a pause. The saved byte is used when resuming operation of the decoder after i_pause is deasserted.

© 2011 ECC Technologies, Inc. All Rights Reserved.

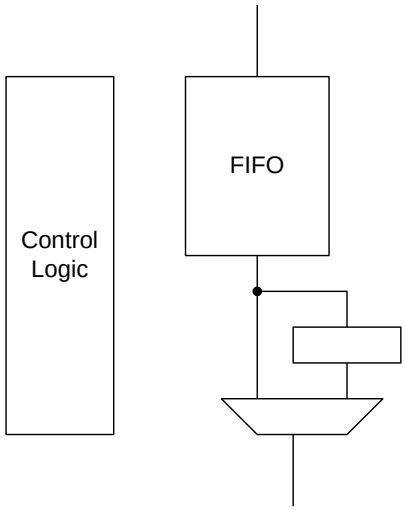


Figure 18 DFIFO

The FIFO shown in Figure 18 can be implemented using a dual-port memory as illustrated in Figure 19 or it can be implemented by two "ping-pong" single-port memories as illustrated in Figure 20. The term "ping-pong" is used to indicate that writing and reading of the RAMs ping-pongs alternately from one buffer to the other. When data is input to the decoder continuously, one buffer is written while the other is read.

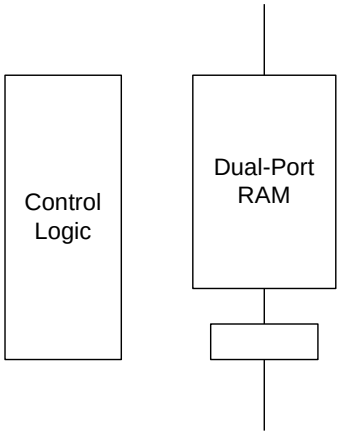


Figure 19 FIFO Implemented with a Dual-Port RAM

© 2011 ECC Technologies, Inc. All Rights Reserved.

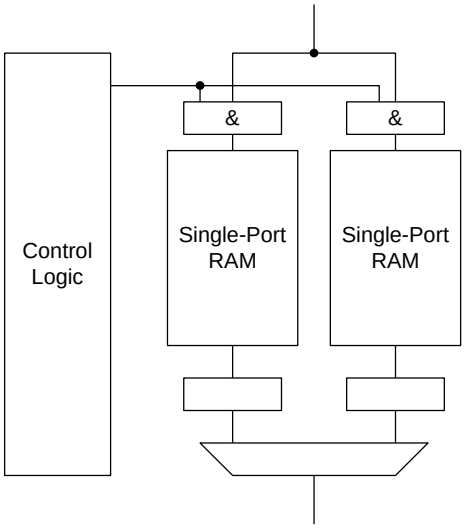


Figure 20 FIFO Implemented with Two Ping-Pong Single-Port RAMS

25. Determining the Minimum Size of the FIFO

ECC Tek determines the minimum size of the dual-port RAM FIFO needed by continuously inputting records which contain the maximum number of errors the decoder can correct and then monitoring the maximum full level of the FIFO. The maximum full level will be how large the FIFO needs to be as long as the decoder is not paused.

The DFIFO needs to hold at least 198 bytes for DECODER6 and 102 bytes for DECODERs A-D.

26. Syndrome Generator Mathematics

The syndrome generator performs the mathematical operations shown in Figure 21. The received word symbols, r_i, as a row vector are multiplied by the transpose of the parity check matrix, H^T, as shown in Figure 21. Another way to view the syndrome generation process is that the received word polynomial, r(x), is evaluated at α^0, α^1, α^2, …, α^{2t-1}.

© 2011 ECC Technologies, Inc. All Rights Reserved.

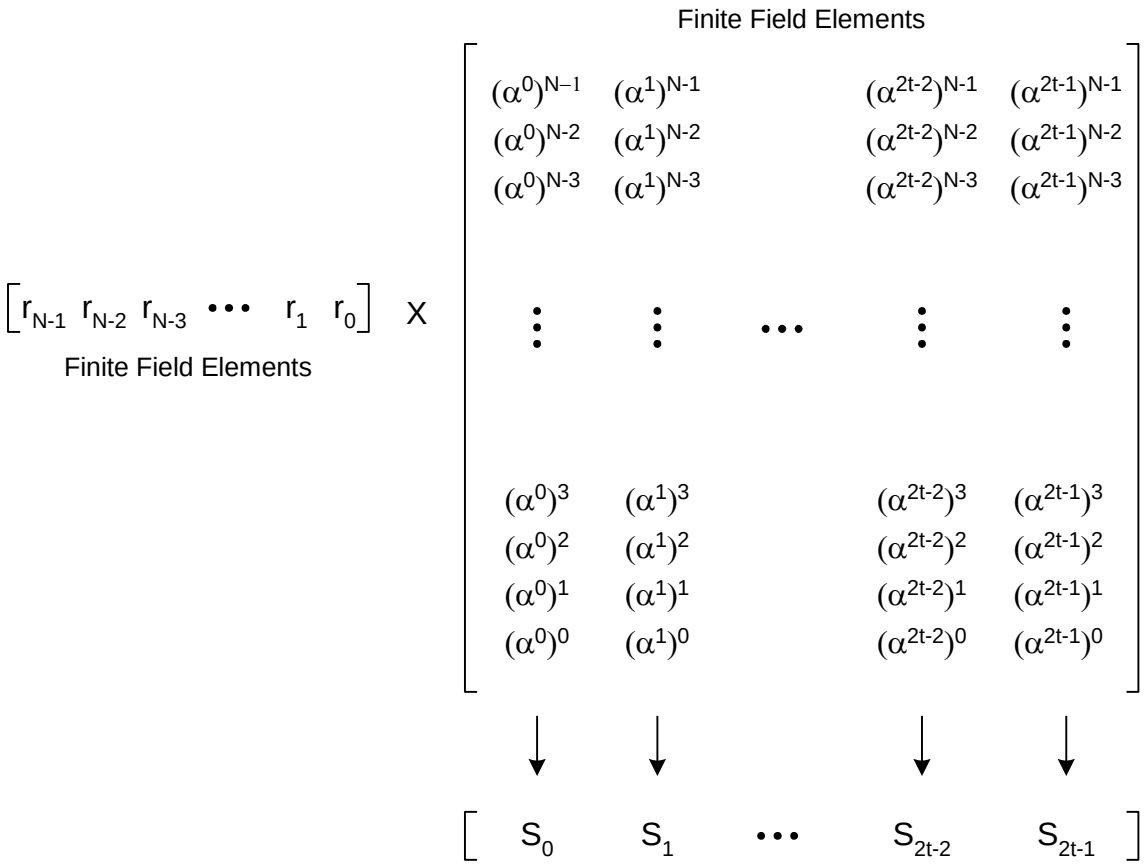


Figure 21 Syndrome Generator Mathematics

27. Syndrome Generator Circuits

Serial syndrome generators input one symbol at a time and, after the last symbol has been received, the syndrome has been calculated. ECC Tek refers to this type of implementation as having a parallelization level of 1.

Parallel syndrome generators input two or more symbols at a time. Different methods can be used to parallelize the syndrome generator. If a syndrome generator inputs p symbols simultaneously, ECC Tek defines the parallelization level to be p.

Fully parallel syndrome generators input all of the received symbols simultaneously. They have a parallelization level of N.

28. SYN

The SYN (Syndrome Generator) data flow is shown in Figure 22.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Figure 22 SYN Data Flow

The SYN control logic consists of combinatorial logic.

A counter is used to count the number of valid symbols as they are being received. Once N valid symbols have been received, the control logic disables the feedback so that the first symbol of the next received word can be directly loaded into the syndrome registers. Whenever the last symbol of a received word is received, SYN asserts an output "go" signal to indicate to the PPU that a valid syndrome is available.

The PPU must immediate transfer each new valid syndrome from SYN into the PPU otherwise it will be lost since SYN has no output holding registers.

Since the PPU requires less than N cycles to complete the calculation of L(x), V(x) and degLx from S(x), the PPU will always be ready to receive the next new valid syndrome.

The syndrome calculator evaluates the received word, r(x), at α^0, α^1, … , α^{15}. The codeword was constructed by the encoder so that α^0, α^1, …, α^{15} are roots of each codeword polynomial, c(x). Therefore, if no errors occur, the syndrome should be all 0s. If the syndrome is not all 0's, then the received word contains errors.

The syndrome generator generates the syndrome as the data is being received and stored in DFIFO. The syndrome, S(x), is available for use by the PPU after all of the symbols of the received word have been received.

The SYN control logic generates a go signal to the PPU when the syndrome for the current received word has been calculated and is available in the syndrome registers. The PPU must not be busy at that time so the PPU can immediately transfer the contents of the syndrome registers into the PPU's V(x) registers.

SYN also contains logic for generating erasure locations as illustrated in Figure 23. Initially the current location is set to α^{-117} which is the inverse of the location of the first codeword symbol received. The location of the next symbol is computed by multiplying the current location by α. The set of the inverses of the erasure locations, eli, are inputted by the PPU.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Figure 23 Erasure Location Generation Logic

29. PPU

The PPU transfers the inverses of the erasure locations and the syndrome from SYN to the PPU's eli and V(x) registers when SYN issues a go signal to the PPU.

The PPU then calculates the error locator polynomial, L(x), the error evaluator polynomial, V(x), and the degree of L(x), degLx, from the inverses of the error locations, eli, and the syndrome polynomial, S(x) according to the EBM algorithm as shown in Figure 24.

Figure 24 EBM Algorithm for Errors Plus Erasures

The flowchart shown in Figure 24 can be used to develop RS decoder software for a conventional processor where only one value is assigned to one variable at one time. Since a conventional processor is not capable of multiple assignments simultaneously as can be done in digital logic, two polynomial variables (arrays) called last_L(x) and last_V(x) are needed in a software implementation to store previous polynomial values. In hardware implementations of the PPU, the last_L(x) and last_V(x) array type variables are not needed.

The key equation is $S(x)L(x) \equiv V(x) \bmod x^{2t}$.

Initially, the BM algorithm assumes that 0 errors have occurred or, in other words, that the correct L(x) is L(x) = 1. This is a reasonable assumption because, in most situations, 0 errors is the most likely error pattern.

If the initial L(x) is L(x) = 1, then the initial V(x) = S(x) because, if we substitute L(x) = 1 into the key equation, the key equation becomes $S(x)1 \equiv V(x) \bmod x^{2t}$, and V(x) must be equal to S(x) to satisfy the key equation.

However, we know from coding theory that in order for L(x) = 1 to be the correct or valid L(x), V(x) must be 0, which means the syndrome must be all zeros.

The BM algorithm examines one coefficient of V(x) in each iteration of the algorithm. The V(x) coefficient being focused on for the jth iteration of the BM algorithm is called the jth "discrepancy" or dj. Initially, j = 0 and d0 = V0 = S0.

© 2011 ECC Technologies, Inc. All Rights Reserved.

Initially, if d0 = V0 = S0 $\neq 0$, the BM algorithm knows that the correct or valid L(x) is not L(x) = 1, that the initial guess that L(x) = 1 must be modified and that the degree of L(x) must be increased. The BM algorithm specifies how L(x) must be modified and how this iterative procedure continues until the correct L(x), the correct V(x) and the correct degLx are all determined. It is very difficult to understand why the BM algorithm works correctly, but it has been proven to be valid in many papers and textbooks on coding theory. Fortunately, even though it is difficult to understand why the BM algorithm works, it is not difficult to implement the BM algorithm as shown in Figure 24.

The BM algorithm requires the use of two "auxiliary" polynomials which ECC Tek calls aL(x) and aV(x). One auxiliary polynomial is associated with L(x) and the other with V(x).

It should be fairly easy for the reader to see from the PPU data flow diagram how the PPUs can be used to implement the BM algorithm shown in Figure 24.

The PPU implements the EBM algorithm. A top level block diagram of a low-gate-count PPU is presented in Figure 25.

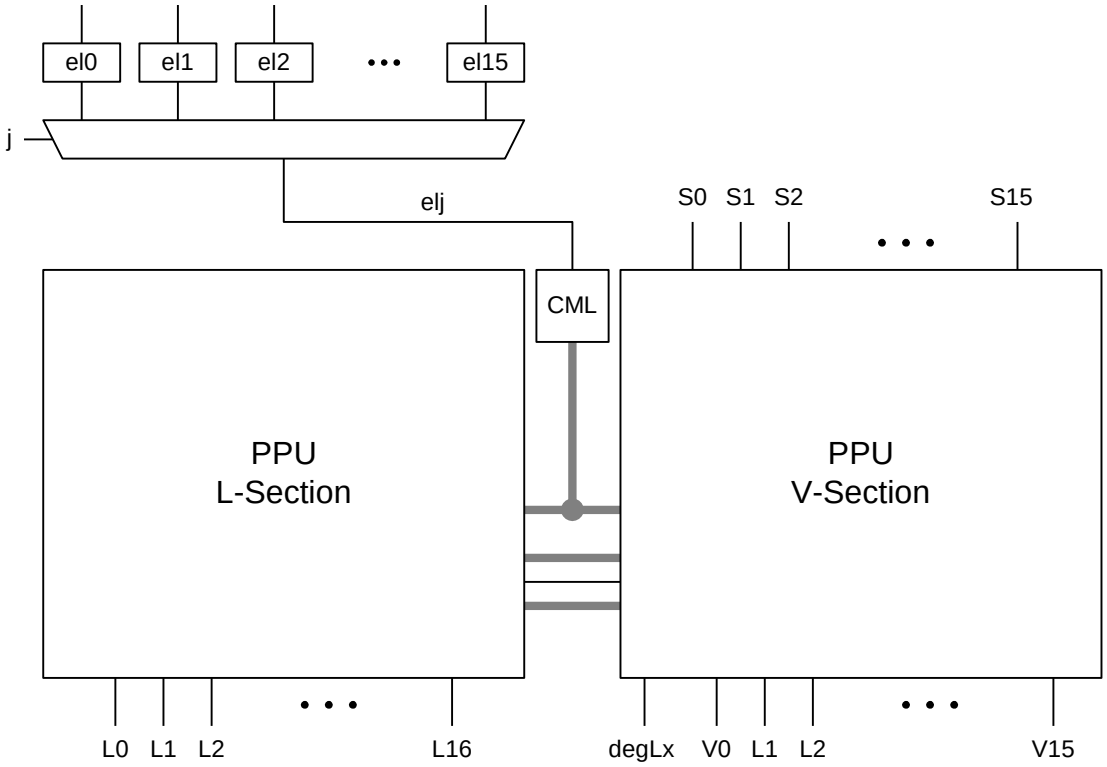


Figure 25 PPU Top Level Block Diagram

Details of what is in the boxes in Figure 25 are shown in the following Figures.

© 2011 ECC Technologies, Inc. All Rights Reserved.

For low-gate-count PPUs, the L-Section and P-Section each contain one Coefficient Processing Unit (CPU).

To understand how ECC Tek has defined "PPU parallelization level", the L-Section of a PPU with a parallelization level of p=1 is shown in Figure 26. One L CPU generates one new polynomial coefficient for L(x) and one new coefficient for aL(x) simultaneously. ECC Tek has defined PPU parallelization level to mean that a PPU with a parallelization level of p contains p CPUs in the L-Section and p CPUs in the V-Section so that p new coefficients of L(x), aL(x), V(x) and aV(x) are all computed simultaneously.

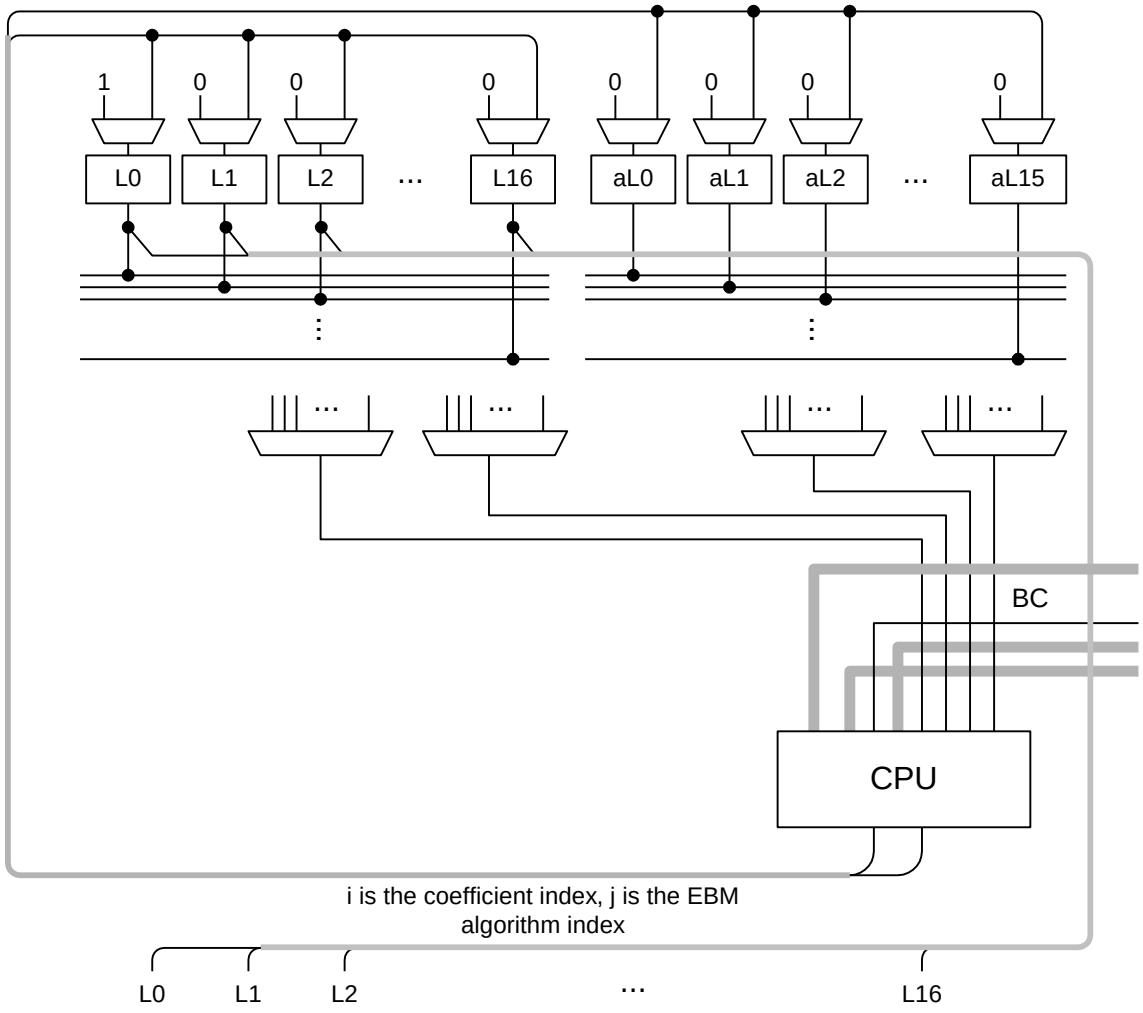


Figure 26 RS PPU L-Section with p=1

© 2011 ECC Technologies, Inc. All Rights Reserved.

The PPU V-Section with p=1 is shown in Figure 27.

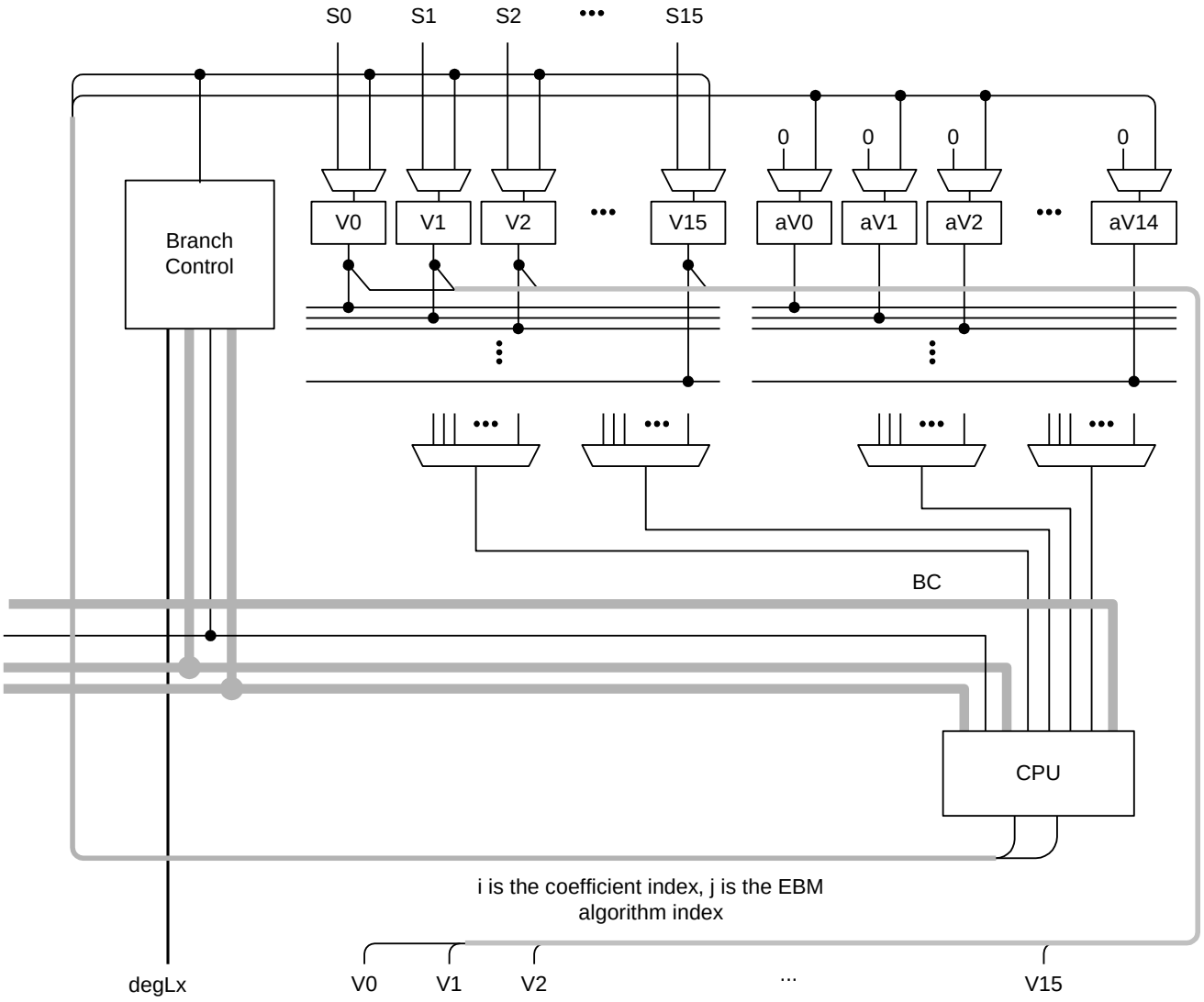


Figure 27 RS PPU V-Section with p=1

© 2011 ECC Technologies, Inc. All Rights Reserved.

The L-CPU for errors and erasures is shown in Figure 28.

Figure 28 L-CPU for Errors and Erasures

The V-CPU for Errors and Erasures is shown in Figure 29.

Figure 29 V-CPU for Errors and Erasures

The Branch Control logic for the p=1 PPU is shown in Figure 30.

Figure 30 PPU Branch Control

When operations are performed on polynomials, operations are being performed on each coefficient of each polynomial. For example, if we add two polynomials, we are adding the coefficients. If we multiply a polynomial by a value, we are multiplying each of the coefficients of the polynomial by that value.

A p=1 PPU generates 1 coefficient of the next L(x) and 1 coefficient of the next aL(x) in one clock cycle and, simultaneously generates 1 coefficient of the next V(x) and 1 coefficient of the next aV(x). Four new polynomial coefficients are generated simultaneously in one clock cycle.

When the PPU has finished calculating L(x) and the degLx, the PPU State Machine sets an output control bit called "o_PPU_full" which indicates to the next module in the pipeline that next L(x), V(x) and degLx values are available.

After the PPU SM sets the "o_PPU_full" bit, the PPU SM waits for the next module in the pipeline to transfer the L(x), V(x) and degLx values. The EVAL module transfers the new L(x) and degLx values into its registers from the PPU and clears the "o_PPU_full" bit which triggers the PPU SM to stop waiting for the got bit to be cleared and start waiting for the next "go" signal.

The PPU control unit contains two index registers called "i" and "j". The j index register corresponds to the "j" variable in the BM algorithm shown in Figure 24. The i index register is used to indicate the coefficient of L(x), aL(x), V(x), and aV(x) the PPU is currently computing. The highest power coefficient is computed first so i is initially set to 16 and coefficients 16, 15, 14, 13, … , 0 are computed.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

The degree of L(x) is initially 0. The degree of L(x) increases as needed and, when correction is successful, the degree of L(x) will be equal to the value of the "degLx" variable. The degLx value should be the same as the number of times L(x) evaluates to 0 when correction takes place in the XOR module. If the degLx is not the same as the number of times L(x) evaluates to 0, then the decoder has detected an uncorrectable error pattern.

30. EVAL

A block diagram of the EVAL data flow is shown in Figure 31. The top level of multipliers evaluate L(x) and V(x) at the inverse of first codeword location which is α^{117}. The first evaluations of L(x) and V(x) are loaded into the registers and then the registers are clocked to evaluate L(x) and V(x) at every other codeword location. If there is an error or erasure at any location, the evaluation of L(x) will be 0. When L(x) evaluates to 0, then the error value (or error magnitude) at that location is [Lx_eval/Lpx_eval]*Vx_eval where Lx_eval is L(x) evaluated at the inverse of the current location, Lpx is L'(x) (L'(x) is the formal derivative of L(x)) evaluated at the inverse of the current location and Vx_eval is V(x) evaluated at the inverse of the current location. When the first root power of the generator polynomial is 0, as it is in this case, then the evaluation of L'(x) is the sum of the odd coefficients of the evaluation of L(x) as illustrated in Figure 31 which is the simplification resulting from choosing the first root power to be equal to 0.

Figure 31 EVAL Data Flow

EVAL evaluates L(x), L'(x) and V(x) at all of the codeword symbol locations and calculates the most likely error symbols, mle[i] for each data symbol.

Since received symbols 0 - 15 correspond to the redundant symbols, they are not corrected. However, it is still necessary to evaluate L(x) for those symbols because the decode fail logic needs to know if L(x) evaluates to zero at any of those locations.

31. XOR

The XOR module receives a "go" signal from the EVAL module and initiates the reading of the DFIFO. The XOR unit receives the most likely error symbol as an input from EVAL and also the degLx value and the o_fail bit generated by the PPU.

The XOR module counts the number of times L(x) evaluates to 0 and, at the end of the outputting the current received word, compares the actual error count with the degLx value which originated in the PPU. If the two values are not equal, the XOR unit has detected an uncorrectable error pattern or a decoding failure.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

32. DECODERA-D

The decoders described in this document were designed for applications where clock frequencies are low and emphasis was put on minimizing gate count. In an effort to further minimize gate count, DECODERs A-D were created by modifying DECODER6.

DECODER6 was modified so that the SYN, PPU and EVAL functions could share the same set of registers. Sharing the same set of registers means that the SYN, PPU and EVAL operations must be performed serially and that the decoder cannot input the next received word until any errors in the present received word have been corrected and the corrected data outputted. Therefore, DECODERA-D cannot handle continuous input data as DECODER6 can, but have reduced complexity.

Sharing the same set of registers can be a little confusing. For example, the syndrome is calculated using the V(x) registers and there are no separate registers for the syndrome, S(x), as there are in DECODER6.

32.1. How DECODER6 was Modified to Create DECODERA-D

In DECODER6, each of the functions shown in Figure 17 are implemented in separate Verilog modules and operate independently. Each function has its own state machine or control logic. When one function is completed with its computation, it outputs a go signal to the following function which then transfers the output data to its input registers. DECODER6 is fully pipelined so that data can be received continuously.

The architecture of DECODERA-D is illustrated in Figure 32. For these decoders, the SYN, PPU and EVAL functions are implemented in one Verilog module called SPE. These decoders are not pipelined and cannot handle continuous input data. With these designs, there must be a gap between input records. The necessary gap between records is shorter for DECODERA and DECODERC than for DECODERB and DECODERD. They were developed in an effort to reduce the gate count as much as possible without dramatically reducing the performance of the decoders.

The state machines that were in SYN, PPU and EVAL are collapsed into one SPE state machine. The SYN, PPU and EVAL functions share the same set of registers so the operations must be done serially.

© 2011 ECC Technologies, Inc. All Rights Reserved.

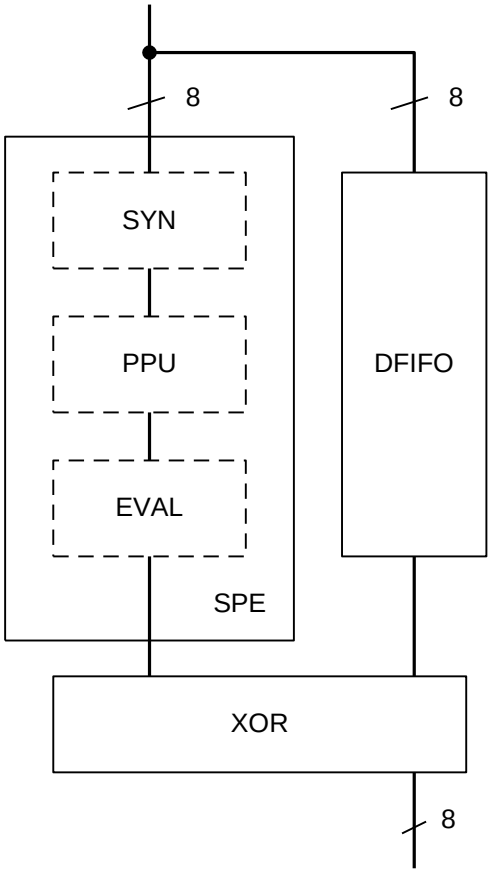


Figure 32 Architecture of DECODERs A-D

DECODERA and DECODERC use 2 CPUs as illustrated in Figure 26 and Figure 27 – one in the PPU L- Section and one in the PPU V-Section. DECODERB and DECODERD use PPUs with only 1 CPU which has input muxes to select either L or V as illustrated in Figure 33 and therefore it takes DECODERB and DECODERD twice as much time to execute the EBM algorithm as it takes DECODERA and DECODERC, and the necessary gap between input records is larger.

Figure 33 X-CPU and Muxes used in DECODERB and DECODERD

DECODERA uses Verilog constructs with a large number of if else, else if, else if, … statements which will be synthesized into a priority sequence of muxes as illustrated in Figure 6. Since the priority property is not needed, the if else, else if, statements were replaced with Verilog code which will be synthesized into muxes in DECODERC. Therefore, DECODERA and DECODERC are functionally equivalent. The only difference between them is in how the Verilog code is structured which will affect what circuits the synthesizer software will synthesize the code into. Theoretically, DECODERC should take slightly fewer gates than DECODERA, but, on the other hand, the Verilog code for DECODERA seems easier to understand.

© 2011 ECC Technologies, Inc. All Rights Reserved.

DECODERB also uses Verilog constructs with a large umber of if else, else if statements as in DECODERA, but only uses 1 CPU instead of 2.

DECODERD uses muxes in place of the else if statements used in DECODERB as is done in DECODERC and uses only one CPU with L or V input muxes to the CPU.

An extra register called the "mode" register is installed in DECODERC and D. The mode register is used as the select input to the L(x) and V(x) registers input muxes. When calculating the syndrome, for example, the mode is SYN and those inputs needed to calculate the syndrome are selected as inputs to V(x).

32.2. No L(x) and V(x) Initialization Multipliers in DECODERA-D

DECODER6 uses additional gated to evaluate the L(x) and V(x) polynomials computed by the PPU at α^{-177} so that the Li values can be immediately calculated for the received word. Unlike DECODER6, DECODERsA-D to not initially evaluate the L(x) and V(x) polynomials at α^{-177} but the L(x) and V(x) registers are evaluated first at $\alpha^{-255} = 1$ which does not require any additional circuitry and then the evaluation circuits are clocked 255-117=138 times (each clock multiplies by α) until L(x) and V(x) are evaluated at α^{-177}.

We could decrease the latency by 138 by adding circuitry to DECODERsA-D to initialize L(x) and V(x) the same way that DECODER6 does.

32.3. How to Use DECODERsA-D

For applications where DECODERA-D can be clocked at a higher frequency than the data, the method illustrated in Figure 34 can be used so that data can be continuously decoded.

As long as symbols are taken out of a FIFO at a rate faster than they are put in, the FIFO full level of the FIFO and the capacity needed for the FIFO will be limited.

© 2011 ECC Technologies, Inc. All Rights Reserved.

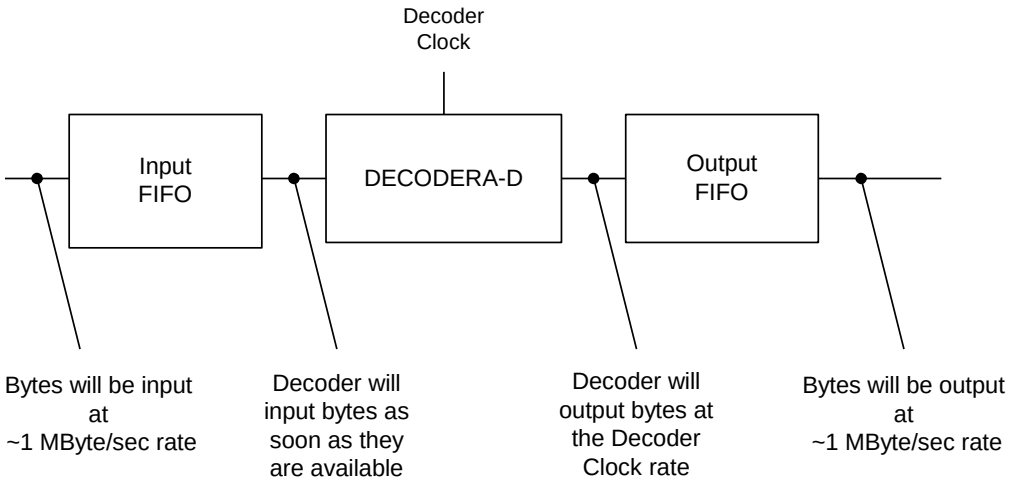


Figure 34 DECODERs A-D with Input and Output FIFOs

Consider DECODERD that has a latency of 691. The time from last byte received by decoder to when the next byte can be received by the decoder is 691+118=809 clock cyles.

The decoder clock must be fast enough so that decoding of the current record is completed before the next record has been completely received.

DECODERA and DECODERC can be clocked as low as 5f where f is the frequency of the data clock and DECODERB and DECODERD can be clocked as low as 7f.

The capacity requirements for the input and output FIFOs shift depending on the frequency of the decoder clock. In other words, if the frequency of the decoder clock is at the low end, the input FIFO needs to be larger, but the output FIFO can be smaller. The capacity of the input FIFO never needs to be more than 118 symbols/bytes and will normally be < 118. The capacity of the output FIFO never needs to be more than 102 symbols/bytes and will normally be < 102 as long as the clock frequencies are at least 5f or 7f.

The extra memory bits required by DECODERs A-D over DECODER6 is probably not very large because DECODER6 requires an internal FIFO with capacity of 199 symbols/bytes and DECODERs A-D require an internal FIFO with a capacity of only 103 symbols/bytes.

The extra power required by DECODERs A-D over DECODER6 probably is not significant if the decoder clock frequency is at the low end.

© 2011 ECC Technologies, Inc. All Rights Reserved. May 12, 2011

33. Further Gate Count Reduction

It is possible to reduce the gate count a little further in DECODERD (which is already the smallest). The technique applied to the L(x) and V(x) registers has not yet been applied to the aL(x) and aV(x) registers because.

34. Implementation of DECODERsA-D

If DECODERsA-D were implemented with input and output FIFOs, they should probably be modified to remove the input valid signals because, if data was put in the input FIFO, it should be valid so the decoder should look for the condition that the input FIFO is not empty and assume each value in the input FIFO is valid.

© 2011 ECC Technologies, Inc. All Rights Reserved.