 By Syed Shahzad Shah, Saqib Yaqub, and Faisal Suleman, Chameleon Logics

NOW THAT YOU'VE BRUSHED UP ON ERROR-CORRECTING CODES IN *PART 1: VITERBI CODECS* (FEB 15, 2001), UNCOVER THE THEORY AND HARDWARE IMPLEMENTATION OF A REED-SOLOMON CODEC.

Self-correcting codes conquer noise Part 2: Reed-Solomon codecs

Reed-Solomon coding is a type of forward-error correction that is used in data-transmission (vulnerable to channel noise) plus data-storage and -retrieval systems. By adding redundant data before transmission, Reed-Solomon codecs (encoders/decoders) can detect and correct errors within blocks of data. The design methodology for the implementation of a Reed-Solomon codec requires you to:

- understand Reed-Solomon codecs,
- implement the entire codec process in Matlab,
- break down the codec process into modules,
- develop hardware for each module,
- tailor the process for hardware efficiency,
- test each module,
- connect the modules, and
- test and verify the Reed-Solomon codec.

Each of these steps is important, and missing one results in developing hardware that does not work the first time and must be recreated. For example, it is critical to understand the mathematics behind the Reed-Solomon-codec process. Simulation programs, such as Matlab, help you quickly and completely understand the Reed-Solomon process without knowledge of the hardware. Once you build your confidence, you can then develop hardware to represent each of the equations.

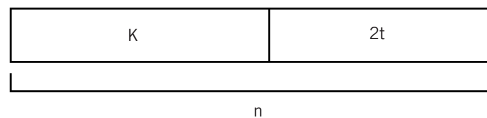
BLOCK CODES

Reed-Solomon codecs operate on blocks of data in which information is divided into frames (blocks), and the encoder uses only the current frame to produce its output. These codes are generally designated as (n, K) block codes: K is the number of information symbols input per block, and n is the number of symbols per block that the encoder outputs. The term "symbol" may represent one bit or a number of bits. Your concern is a subclass of block codes called linear block codes. You can define linear codes as those codes in which the sum of two code words is another code word, and the product of any code word by a scalar (field element) is also a code word. Further, an important subclass of linear block codes is cyclic codes. A code is cyclic if by cyclically shifting the components of the code word one place to the right, you get another valid code word (**Reference 1**).

GALOIS FIELDS

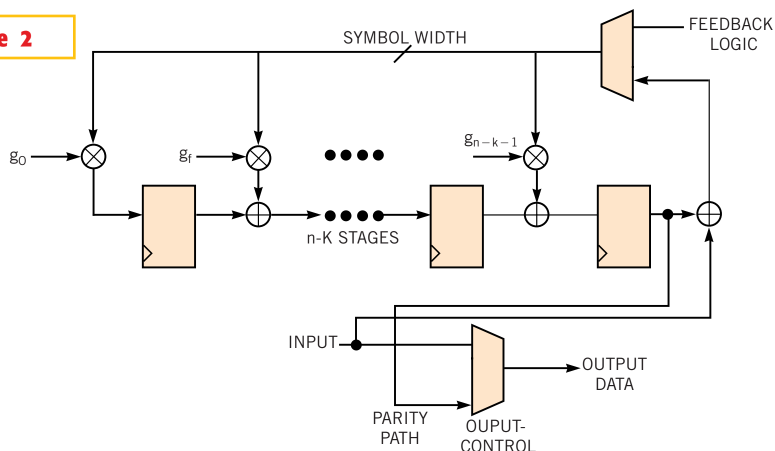
A powerful algebraic structure, known as a field, is a set in which you can add, subtract, multiply, and divide. Several well-known fields are the set of real numbers and the set of complex numbers. This ar-

Figure 1



Reed-Solomon codes are a special nonbinary subclass of BCH codes.

Figure 2

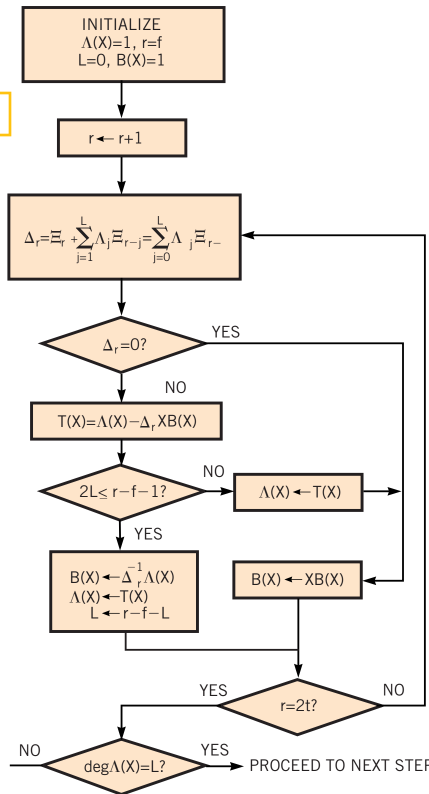


Multipliers, adders, and control circuitry make up the general architecture of a Reed Solomon(n,k) encoder.

ticle concerns fields with a finite number of elements. You call a field with q elements a finite field or a Galois field and denote it by the label GF(q). For example, GF(2) has two elements {0,1}. The most powerful and important ideas of coding theory are based on the arithmetic systems of the Galois fields. (**Reference 1** contains a detailed analysis of Galois fields.)

An extension field, denoted as $GF(p^m)$, is a finite field in which the number of elements is an integer power of a prime number, p (**Reference 2**). To determine the properties of extension fields, consider the extension fields for p=2. In $GF(2^m)$ fields, there is always a primitive element $\propto$, such that you can express every element of $GF(2^m)$ except zero as a power of $\propto$. You can generate every field $GF(2^m)$ using a primitive polynomial over GF(2), and the arithmetic performed in the $GF(2^m)$ field is modulo this primitive polynomial.

Figure 3



A flow chart for the Berlekamp-Massey algorithm can correct both errors and erasures.

Addition and subtraction operations are the same in $GF(2^m)$ fields, and adding any field element to itself yields a zero field element. In the case of $GF(2^3)$, the primitive polynomial of degree 3 that you can use to generate the elements of $GF(2^3)$ is given as $p(X)= X^3+X+1$.

Note that p(X) is a polynomial over GF(2), as the coefficients of the variables X^3 and X are the members of GF(2), namely 0 or 1. Let $\propto$ be a primitive element of $GF(2^3)$. Because $\propto$ is a root of p(X), $\propto^3+\propto+1=0$, and according to the above properties, $\propto^3=\propto+1$.

Now, applying the above result and the above properties, you can generate all the elements of $GF(2^3)$ in terms of powers of $\propto$. The numerical value of $\propto$ is purely arbitrary. Using a polynomial representation, you can assign the values in **Table 1** to field elements.

The addition of two field elements is modulo the primitive polynomial p(X). You can represent every field element as a polynomial over GF(2) of a degree less than or equal to 2 (generally, m−1). Also, the polynomial representation of field elements facilitates the addition process: Simply add the polynomial coefficients of field elements modulo 2 to get the result. The binary notation is a direct consequence of polynomial notation, showing the coefficients of the polynomial for respective powers of X. For example, to add $\propto^2$ to $\propto^3$, modulo 2 add 100 and 011 to get $111=\propto^5$.

You multiply two field elements by adding the powers of $\propto$ modulo $2^3-1=7$ (generally, 2^m-1). For example, $\propto^4 X \propto^6=\propto^{10\bmod(7)}=\propto^3$. Higher powers of $\propto$ simply repeat the pattern. So, $\propto^7=\propto^0, \propto^8=\propto$, and so on.

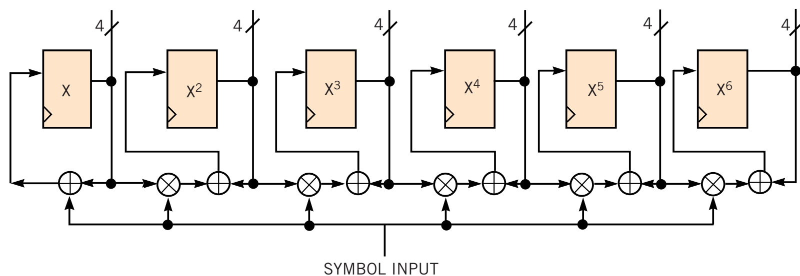
RS CODES

BCH (Bose-Chadhuri-Hocquenghem) codes are linear cyclic block codes that allow multiple-error correction. Reed-Solomon codes are a special nonbinary (more than 1 bit per symbol) subclass of BCH codes that achieve the largest possible code minimum distance. For nonbinary codes, you define the distance between two code words as the number of nonbinary symbols in which the sequence differs. Because the minimum distance is the distance between an all-zero code word and the code word closest to it, the minimum distance of a linear block code is equal to the minimum number of nonzero symbols occurring in any code word excluding the all-zero code word (**Reference 3**).

Reed-Solomon code is specified as an Reed-Solomon (n,K) code, where n is the number of symbols per block that the encoder outputs; K is the number of information symbols you input to the encoder; n−k=2t is the number of parity symbols that the encoder adds to each block; t=(n−k)/2 is the maximum number of symbol errors that the Reed-Solomon(n,K) code can correct in each block anywhere in the block; and minimum distance is $d_{MIN}=n-k+1$ (**Figure 1**).

Reed-Solomon codes are the algebraic codes in which the polynomials over

Figure 4



Field adders and multipliers dominate the hardware architecture necessary to calculate Γ(X) for a Reed-Solomon(15,9) code.

$GF(2^m)$ fields represent the information to be encoded and the encoded code word. Simply put, the input and output information consists of symbols, which are elements of $GF(2^m)$ $(0,1,\propto,\propto^2$, and others). These symbols are arranged as coefficients in the polynomial, and the power of the polynomial variable X indicates the order in which the encoder and decoder receive and output the associated symbol. For example, an information polynomial, $\propto X^2+\propto^3 X+\propto^6$, shows that $\propto$ is the symbol that you first input to the encoder, then $\propto^3$, and lastly, $\propto^6$. Also for Reed-Solomon(n,K) code, $n=2^m-1$ if $m=4$, $n=15$, and the Galois field you use to encode or decode the Reed-Solomon(n,K) code is $GF(2^4)$. All symbols are elements of $GF(2^4)$, all polynomials are polynomials over $GF(2^4)$, and you use the arithmetic for $GF(2^4)$ during the encoding and decoding process.

ENCODING REED SOLOMON(N,K) CODES

The key equation defining the systematic encoding operation for Reed-Solomon(n,K) code is:

$$c(X)=i(X)X^{n-k}+[i(X)X^{n-k}]\bmod g(X), \quad (1)$$

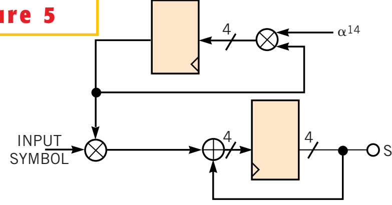
where c(X) is the code-word polynomial of degree $n-1$, i(X) is the information polynomial of degree $k-1$, $[i(X)X^{n-k}]\bmod g(X)$ is the parity polynomial of degree $n-k-1$, and g(X) is the code-generator polynomial of degree $n-k$.

Systematic encoding means that, after encoding, the resulting code word is such that the original information symbols are inserted at the higher order coefficients of the code word and then select parity symbols to force a legitimate code word c(X).

In general, the polynomial g(X) for Reed-Solomon(n,K) codes is: $g(X)=(X-\propto^j)(X-\propto^{j+1})\ldots\ldots(X-\propto^{j+2t-1})$, and, if $j=1$, $g(X)=g_{n-k}X^{n-k}+g_{n-k-1}X^{n-k-1}+\ldots\ldots g_2X^2+g_1X+g_0$.

You can choose any integer value for j; however, it's conventional to use $j=1$. Sometimes, by choosing $j=1$, you can reduce the number and the cost of circuit components. Note that all of the above polynomials, including g(X) are polynomials over $GF(2^m)$. An important property of g(X) is that it exactly divides c(X); that is, dividing c(X) by g(X) yields a remainder of zero.

Figure 5



The hardware for computing one S_j for a Reed-Solomon(15,9) code uses a finite-field-feedback multiplier.

To visualize hardware that implements **Equation 1**, you must understand the operations $i(X)X^{n-k}$ and $[i(X)X^{n-k}]\bmod g(X)$. As previously mentioned, for systematic encoding, you place information symbols as the higher power coefficients. So, $i(X)X^{n-k}$ means that you "shift" information symbols toward the higher powers of X, from $n-1$ down to $n-k$. You fill the remaining positions from power $n-k-1$ to 0 with zeros. Consider, for example, the same polynomial as above:

$$i(X)=\propto X^2+\propto^3X+\propto^6.$$

Multiplying the above equation by X^4 yields:

$$i(X)X^4=\propto X^6+\propto^3X^5+\propto^6X^4+0X^3+0X^2+0X+0.$$

The second term of **Equation 1**, $[i(X)X^{n-k}]\bmod g(X)$, is the remainder when you divide polynomial $i(X)X^{n-k}$ by the polynomial g(X). Therefore, you need to design a circuit that performs two operations: a division and a shift to a higher power of X. Linear-feedback shift registers enable you to easily implement both operations.

Figure 2 shows a general diagram of the encoder for Reed-Solomon(n,K) code. The main design task is to implement the $GF(2^m)$ multiplication and addition circuits, apart from some control circuitry or logic. Remember that you can add any two elements from the $GF(2^m)$ field by modulo 2 adding their binary notations, which resembles the XOR hardware operation. So your finite-field adder simply consists of XOR gates. A $GF(2^4)$ adder, for example, needs four two-input XOR gates to add two symbols over $GF(2^4)$. You can easily map the equation for the XOR operation into the LUT (look-up-table) architecture of an FPGA.

The finite-field multiplier requires you to put some calculations in equation form. This task may become tedious when m increases in $GF(2^m)$ and thus increases the hardware requirements. As an example, consider the multiplication of two field elements from $GF(2^3)$. The elements are represented in polynomial form as:

$$\beta_1=\gamma_2z^2+\gamma_1z+\gamma_0, \text{ and}$$
$$\beta_2=\lambda_2z^2+\lambda_1z+\lambda_0,$$

where the coefficients $\gamma_2,\lambda_2,\gamma_1,\lambda_1,\gamma_0,\lambda_0$ can be 0 or 1. Also, recall that the primitive polynomial for $GF(2^3)$ is $p(z)=z^3+z+1$. Use this primitive polynomial to reduce the product polynomial's degree to 2:

$$\beta_1\beta_2=(\gamma_2z^2+\gamma_1z+\gamma_0)(\lambda_2z^2+\lambda_1z+\lambda_0)=\gamma_2\lambda_2z^4+\gamma_2\lambda_1z^3+\gamma_2\lambda_0z^2+\gamma_1\lambda_2z^3+\gamma_1\lambda_1z^2+\gamma_1\lambda_0z+\gamma_0\lambda_2z^2+\gamma_0\lambda_1z+\gamma_0\lambda_0=\gamma_2\lambda_2z^4+(\gamma_2\lambda_1+\gamma_1\lambda_2)z^3+(\gamma_2\lambda_0+\gamma_1\lambda_1+\gamma_0\lambda_2)z^2+(\gamma_1\lambda_0+\gamma_0\lambda_1)z+\gamma_0\lambda_0.$$

Now, using $z^3=z+1$, you can normalize the powers down to 2:

$$\beta_1\beta_2=\gamma_2\lambda_2(z^2+z)+(\gamma_2\lambda_1+\gamma_1\lambda_2)(z+1)+(\gamma_2\lambda_0+\gamma_1\lambda_1+\gamma_0\lambda_2)z^2+(\gamma_1\lambda_0+\gamma_0\lambda_1)z+\gamma_0\lambda_0=(\gamma_2\lambda_0+\gamma_1\lambda_1+\gamma_0\lambda_2+\gamma_2\lambda_2)z^2+(\gamma_1\lambda_0+\gamma_0\lambda_1+\gamma_2\lambda_2+\gamma_2\lambda_1+\gamma_1\lambda_2)z+(\gamma_0\lambda_0+\gamma_2\lambda_1+\gamma_1\lambda_2). \quad (2)$$

The above equation implements the variable field multiplication in $GF(2^3)$. The coefficient with z^2 represents the most significant bit of the product, and the last coefficient represents the least significant bit. Note that the plus sign, adding various terms of a coefficient, represents an XOR operation, not an OR operation. The product terms, however, refer to an AND operation. This polynomial approach reduces the labor of finding a general equation for multiplication. Otherwise, you would have to construct a

TABLE 1—FIELD-ELEMENT-VALUE ASSIGNMENTS USING A POLYNOMIAL REPRESENTATION

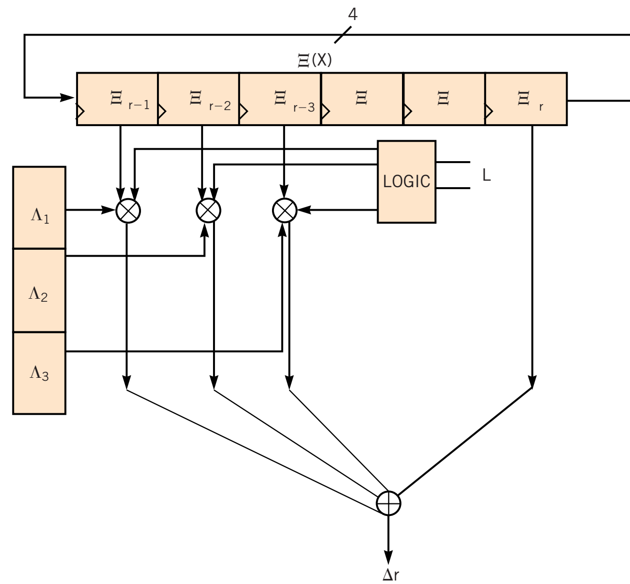
Exponential notation	Polynomial notation	Binary notation
0	0	000
α0	1	001
α	X	010
α2	X²	100
α3	X+1	011
α4	X²+X	110
α5=α3+α2=α2+α+1	X²+X+1	111
α6=α3+α2+α=α2+1	X²+1	101

Boolean truth table with six inputs, three outputs, and maps for each output. For larger values of m, however, either approach seems monotonous.

You can easily map **Equation 2** into the LUT architecture of an FPGA. You can manually optimize the recurring terms in various coefficients, or use a synthesis tool to do it automatically. In general, a Boolean equation of four variables consumes one four-input LUT of an FPGA with a LUT-based architecture. You map the larger variables to multiple LUTs. The key to speed and area optimization is understanding how your synthesis tool maps the equations into the LUT. The adders and multipliers will help both the encoding and decoding processes.

The encoder block diagram shows that one input to each multiplier is a constant field element, which is a coefficient of the polynomial g(X). For a particular block, you input the information polynomial i(X) to the encoder symbol by symbol. These symbols appear at the output of the encoder after a desired latency, where control logic feeds it back through an adder to produce the related parity. This process continues until all of the K symbols of i(X) are input to the encoder. During this time, the control logic at the output enables only the input data path, while keeping the parity path disabled. With an output latency of, say, one clock cycle, the encoder outputs the last information symbol at K+1th clock pulse. Also, during the first K clock cycles, the feedback control logic allows you to feed the adder output to the bus. After you have input the last symbol into the encoder (at the Kth clock pulse), you cannot immediately start the next block; you must wait at least n−K clock cycles. During this waiting time, the feedback control logic disables the adder output from being fed back and supplies a constant zero symbol to the bus. (Remember that this step corresponds with the operation $i(X)X^{n-k}$.) Also, the output control logic disables the input data path and allows the encoder to output the parity symbols (K+2th to n+1th clock pulse). Hence, you can start a new block at the n+1th clock pulse.

Figure 6



The hardware that implements a Reed-Solomon(15,9) code includes multipliers feeding the J and Λ registers, an output adder representing the Σ symbol, and control logic.

RS DECODERS

Although you can use the conventional techniques for decoding the cyclic codes to decode the BCH codes, several better decoding algorithms have been developed. The decoding is based on the Peterson-Gorenstein-Zierler decoder algorithm (**Reference 1**). This algorithm requires matrix inversion, which may be cumbersome if you are designing a large error-correcting decoder. Therefore, other techniques were developed. Elwyn Berlekamp found one such technique by exploiting the organized structure of the aforementioned matrix. You can use Berlekamp's technique to solve the key equation (that is, to find the error locator/connection polynomial) in the decoding process (**Reference 1**). (The following approach deals only with the necessary equations and their hardware implementation. See **references 1** and **4** for a more mathematical treatment of the subject.)

The decoding procedure for Reed-Solomon codes involves determining the locations and magnitudes of the errors in the received polynomial r(X). Locations are those powers of X (X^2, X^3, and others) in the received polynomials whose coefficients are in error. Magnitudes are symbols that you add to the corrupted symbol to find the original encoded symbol. These locations and magnitudes constitute the so-called error polynomial. Also, if you have built the decoder to support erasure decoding, then you must also find the erasure polynomial. An erasure is an error with a known location. Thus, your only task is to find the magnitudes of the erasures. A Reed-Solomon(n,K) code can successfully correct as many as 2t=n−K erasures if no errors are present. With both errors and erasures present, the decoder can successfully decode if n−K≥2v+f, where v is the number of errors, and f is the number of erasures.

The received polynomial r(X) is:

$$r(X)=r_{n-1}X^{n-1}+r_{n-2}X^{n2}+\dots\dots\dots\dots +r_2X^2+r_1X+r_0.$$

The degree of r(X) is n−1, which is the same as c(X). The following equations show the decoding procedure and hardware implementation of the main equations.

Assume that the received vector r(X) has f erasures and that the erasure locators are: $Y_1=\propto^{j1}$, $Y_2=\propto^{j2}$, ... $Y_f=\propto^{jf}$.

The receiver/demodulator provides these erasure locations. You decode the received polynomial according to the following steps:

1. Compute the erasure locator polynomial according to the equation:

$$\Gamma(X)=\prod_{l=1}^{f}(1-Y_lX).$$

2. Replace the erased coordinates (symbols) with zeros and compute the 2t syndromes. You find the syndromes by evaluating the received polynomial r(X) (with erasure positions filled with zero symbols) at the 2t roots of the generator

polynomial g(X):

$S_j = r(\propto^j)$ j=1,2.....2t.

3. Compute the modified syndrome polynomial according to equation:

$$\Xi(X)=(\Gamma(X)[1+S(X)]-1)\bmod X^{2t+1}, \quad (3)$$

where S(X) is the syndrome polynomial defined by

$$S(X)=\sum_{j=1}^{2t} S_j X^j = S_1X+S_2X^2+\cdots\cdots+S_{2t}X^{2t}. \quad (4)$$

4. Now, to solve the key equation (to find the error locations), apply the Berlekamp-Massey algorithm. Doing so gives you the error locator/connection polynomial Λ(X). **Figure 3** contains a flow chart to implement the Berlekamp-Massey algorithm for both errors and erasures.

5. Find the roots of Λ(X) to find the error locations (X_k^{-1} and X_k) . This step is known as a Chien search.

6. Determine the error/erasure polynomial using the equation:

$$\Psi(X)=\Lambda(X)\Gamma(X). \quad (5)$$

7. Determine the error magnitude polynomial using :

$$\Omega(X)=\Lambda(X)[1+\Xi(X)]\bmod X^{2t+1}. \quad (6)$$

8. Determine the error and erasure magnitudes (Forney's algorithm) using:

$$e_k=\frac{-X_k\Omega(X_k^{-1})}{\Psi'(X_k^{-1})}, \text{ and}$$

$$f_k=\frac{-Y_k\Omega(Y_k^{-1})}{\Psi'(Y_k^{-1})}.$$

The derivative of any polynomial over GF(2^m) is given by:

$$u(X)=u_vX^v+u_{v-1}X^{v-1}+..........+u_2X^2+u_1X+u_0,$$

which is a vth degree polynomial. You can then define u'(X) as :

$$u'(X)=vu_vX^{v-1}+(v-1)u_{v-1}X^{v-2}+\cdots\cdots+u_5X^4+u_3X^2+u_1=\sum_{j=0}^{v}(ju_j)X^{j-1}. \quad (7)$$

where ju_j=0 when j is even and $ju_j=u_j$ when j is odd, for example, for the polynomial $u(X)=\propto X^4+\propto^2X^3+\propto X^2+X+\propto^5$, $u'(X) = \propto^2X^2+1$.

9. By combining the error (erasure) magnitudes and locations, you can construct the error (erasure) polynomials and add them to r(X) to retrieve the code word.

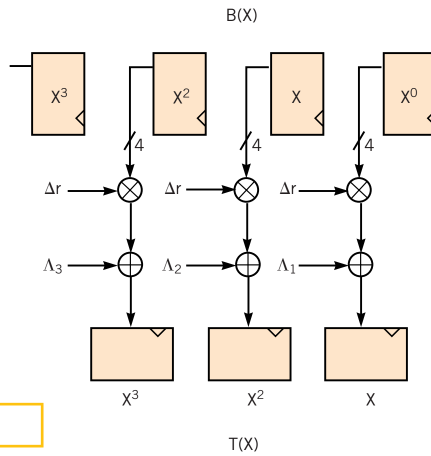


Figure 7

This hardware calculates T(X) for a Reed-Solomon (15,9) code, according to the equation $T(X) = \Lambda(X)-\Delta_r XB(X)$.

Some basic points can help you understand the possible hardware architecture for the above equations. Because you are dealing mostly with the polynomials, in hardware you need (v+1)xm flip-flops to represent a polynomial of degree v with coefficients (symbols) of m bits. Also, the subscripts you use to represent coefficients in any polynomial are the same as their corresponding degree of variable X (for example, u_2X^2 and $u_{n-1}X^{n-1}$). You carry out polynomial multiplication normally, except that you perform additions and multiplications over GF(2^m) in all terms.

Also, all the multiplication and division operations in the above equations are defined over the GF(2^m) fields, not over ordinary multiplication or division operators, and, therefore, plus and minus signs are interchangeable.

Now, consider a brief interpretation and hardware realization of the aforementioned equations.

$$\Gamma(X)=\prod_{l=1}^{f}(1-Y_lX).$$

requires the computation of an f-degree polynomial. The erasure locations are expressed as powers of ∝. For example, if you declare that the demodulator should erase the coefficient of X^4 in the received polynomial, then the corresponding Y_i is $\propto^4$. Because you know the maximum value of f (which is 2t) for a Reed-Solomon(n,k) code, you know the maximum degree of Γ(X), which also equals 2t. Therefore, you need (2t+1)m flip-flops, arranged in groups of m bit registers, to represent the polynomial Γ(X). Further, in this case, the lowest degree coefficient (Γ_0) is always symbol 1, so you need not calculate it. This simplification reduces the number of flip-flops to 2tm. A practical decoder receives the erasure locations one by one, so the degree of Γ(X) increases by 1 every time you add an erasure. The actual degree of Γ(X) thus indicates the number of erasures received. If, for example, you receive f erasures, where f<2t, then the degree of Γ(X) is f, meaning the coefficient of X^f is a nonzero symbol, and the coefficients of degrees of X greater than f are zero symbols. Expanding this equation term by term, you detect symmetry in the equation's architecture, which helps you reduce the amount of hardware. This reduction is suitable for FPGAs. As stated earlier, you can find field adders and multipliers in almost all of the above equations. **Figure 4** shows a diagram of the hardware architecture to calculate Γ(X) for a Reed-Solomon(15,9) code.

The equation $S_j = r(\propto^j)$ for j=1,2...2t is basically a power-sum computation. It involves the dynamic multiplication of the incoming symbol with powers of ∝ plus an accumulation process to compute one S_j. For example, consider an evaluation of r(X) at $\propto, \propto^2$, and so on; replace any erasure locations with zero symbols:

$S_1=r(\propto)=r_{n-1}\propto^{n-1}+r_{n-2}\propto^{n-2}+\ldots+r_2\propto^2+r_1\propto+r_0$, and $S_2=r(\propto^2)=r_{n-1}\propto^{n-2}+r_{n-2}\propto^{n-4}+\ldots+r_2\propto^4+r_1\propto^2+r_0$.

As you can see, the main task is to dynamically compute various powers of ∝. You can accomplish this computation with finite-field-feedback multipliers. These multipliers use the fact that $\propto^{n-1}\times\propto^{n-1}=\propto^{2n-2}=\propto^{(2n-2)\bmod n}=\propto^{-2}=1.\ \propto^{-2}=\propto^n\times\propto^2=\propto^{n-2}$. These equations show that if you tie one input of a multiplier to the power of ∝ that you see with r_{n-1}, for a particular S_j; initialize the multiplier output with the same power of ∝; and feed back the multiplier output to its

second input, then the next output will be the power of $\propto$ that S_j requires. **Figure 5** shows the hardware for computing one S_j for a Reed-Solomon(15,9) code. Parallel implementation requires 2t units. You can also compute the syndromes by using minimal polynomials (**Reference 1**).

As stated earlier, $\Xi(X)=(\Gamma(X)[1+S(X)]-1)\text{mod } X^{2t+1}$.

In the previous equation, S(X) is the syndrome polynomial containing S_j. You can arrange these S_j terms as a polynomial as per **Equation 4.** You can now expand the term $(\Gamma(X)[1+S(X)]-1)$ to yield a 2t+f-degree equation, and the mod X^{2t+1} operation reduces the degree to 2t. This reduction means you need only compute the remaining terms in hardware. Also, careful inspection of **Equation 3** reveals that Ξ_0 is always symbol 0. You can implement the final form of **Equation 3** in parallel either by employing as many adders and multipliers as the equation indicates or by computing $\Xi(X)$ in a more controlled or serial fashion to minimize the hardware requirements.

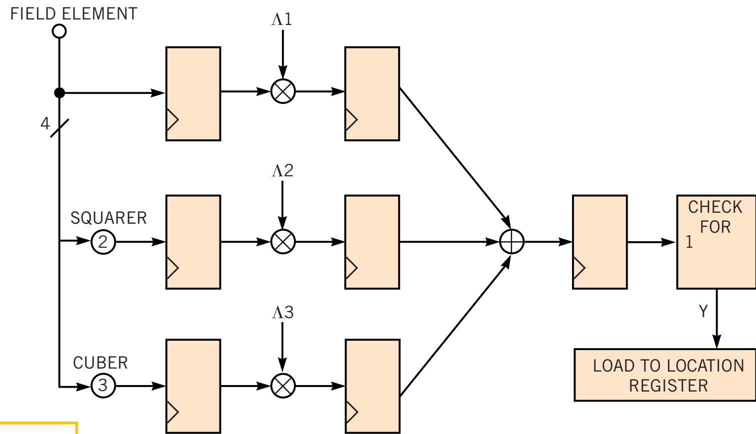


Figure 8

This hardware finds the roots of $\Lambda(X)$ for a Reed-Solomon(15,9) decoder, according to the equation $T(X) = \Lambda(X)-\Delta_r XB(X)$.

THE BERLEKAMP-MASSEY ALGORITHM

The most important and perhaps the most difficult part of Reed-Solomon decoding is solving the key equation using the Berlekamp-Massey algorithm. You can also use the Euclidean algorithm (**Reference 1**). The flow chart in **Figure 3** derives from **references 1** and **4.** Here again B(X), T(X), and $\Lambda(X)$ are polynomials; Δ_r represents a field element, so it requires m bits in hardware; L is a register whose bit size you can select by looking at the value of t (maximum correctable errors); and r controls the algorithm flow. In hardware, it depicts the control designed for the algorithm. B(X) is a t degree polynomial, so it requires (t+1)m flip-flops. T(X) and $\Lambda(X)$ are also t-degree polynomials, but they require just tm flip-flops, because $T_0=\Lambda_0=1$. The major equation in **Figure 3** computes the discrepancy Δ_r, given by:

$$\Delta_r = \Xi_r + \sum_{j=1}^{L} \Lambda_j \Xi_{r-j}.$$

The second term on the right reduces to a zero symbol when L=0. To obtain the hardware implementation of this equation, you use the same variable field multipliers but with an enable control, so that the multiplier gives the normal product at the output when its enable is high and gives a zero symbol at the output when enable is low. You construct this enable logic using the bits representing L. Initially, you load the polynomial $\Xi(X)$ into a register with m LSBs of the register containing the coefficient Ξ_{f+1}. With each iteration of r, this register is circular-right shifted over m bits. In general, you need t multipliers to form the products inside the summation sign. **Figure 6** shows the connections of the multiplier inputs to the Ξ and Λ registers, along with the output adder representing the Σ and the control logic for a Reed-Solomon(15,9) code. Note that every time the BM algorithm starts, the variables are initialized as in the first block in **Figure 3**, and you load polynomial $\Xi(X)$ in the register as described above. The more erasures, the fewer iterations that the BM algorithm takes. The maximum number of iterations when f=0 is 2t.

The equation $T(X)=\Lambda(X)-\Delta_r XB(X)$ is simple to implement. As stated earlier, multiplying X by a polynomial means shifting the polynomial left so as to move every coefficient to one higher power position. You can implement this step in hardware without actually shifting B(X), because you always insert an m-bit zero symbol at the m LSBs of the B(X) register during the XB(X) operation. **Figure 7** shows simple hardware calculating T(X) for a Reed-Solomon(15,9) code. You can implement the remaining equations in the same way.

Although you don't shift B(X) to compute T(X), you must shift B(X) when you perform operation B(X)=XB(X), because you must change the value of B(X). During this shift operation, you lose the m MSBs of B(X) register. To find the inversion of a field element (for example, Δ_r^{-1}), it is useful to construct a Boolean truth table and find the equations of the output bits. All of the operational blocks in the BM unit work in the required sequence by a perfectly designed control.

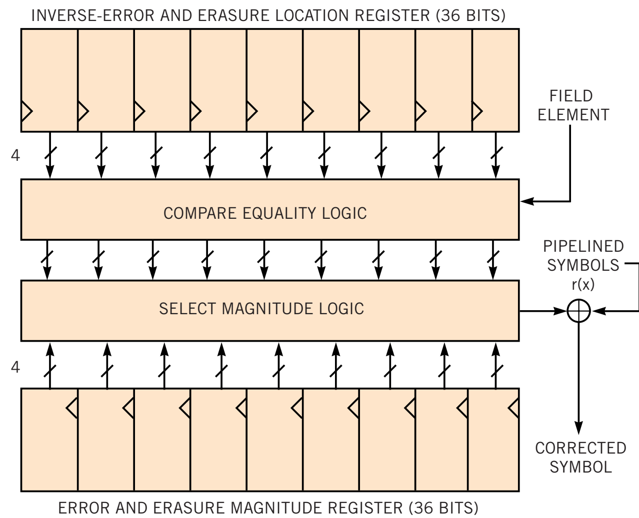
The end result of the BM unit is a polynomial $\Lambda(X)$. The check "deg $\Lambda(X)=L$" provides a clue as to whether the decoder will successfully decode r(X) or declare a failure. Remember that although this check is necessary (to proceed further), it is insufficient to determine whether the decoder will decode correctly. For more on failures, see **Reference 1**. The degree of $\Lambda(X)$ indicates the number of errors that actually occurred (provided errors are less than or equal to t, meaning that the decoder will decode correctly). Note that in hardware, to check the degree of a polynomial, locate the first nonzero symbol starting from the most significant position of the polynomial register. Once you find it, you can easily determine the actual degree of the polynomial.

FINDING ERROR LOCATIONS

In the Chien search, you evaluate the polynomial $\Lambda(X)$ at all the field elements

of $GF(2^m)$ you use for a given Reed-Solomon(n,k) code. You can exclude the zero element because it cannot be a root. For example, the roots $\propto^2$ ($=X_1^{-1}$) and $\propto^4$($=X_2^{-1}$) for a Reed-Solomon(15,9) code are actually inverse error locations (that is, $X_1=\propto^{-2}=1.\ \propto^{-2}=\propto^{15}.\propto^{-2}=\propto^{13}$ and similarly $X_2=\propto^{-4}=\propto^{11}$). The coefficients of X^{13} and X^{11} are, therefore, corrupted in the received polynomial r(X). An inability to find as many roots as the degree of $\Lambda(X)$ in a given $GF(2^m)$ reflects a decoding failure (**Reference 1**). Remember that the decoder may signal a decoding failure depending on the number and distribution of errors in a block.

Figure 9



Hardware to compute the final addition of the error and erasure polynomials to the received polynomial results in a corrected output polynomial.

The hardware's task is to find the value of $\Lambda(X)$ at different powers of $\propto$. One way to find it is by using circuits that generate the squares, cubes, and other powers of an input field element, multiplying by respective values of Λ_j and adding all the terms. If the sum evaluates to a zero symbol, then you regard that field element as a root of $\Lambda(X)$. For example, consider the hardware that finds roots of $\Lambda(X)$ for Reed-Solomon(15,9) code, and assume that the highest degree of $\Lambda(X)$ is t=3. Boolean tables can help you construct the square- and cube-generator circuits in the same manner as the inversion circuit. Because Λ_0 always equals 1, you can determine whether the addition result excluding symbol 1 equals 1. **Figure 8** shows the general hardware that finds $\Lambda(X)$ roots for a Reed-Solomon(15,9) code.

Determine the error/erasure polynomial using the equation:

$$\Psi(X)=\Lambda(X)\Gamma(X).$$

Determine the error magnitude polynomial using :

$$\Omega(X)=\Lambda(X)[1+\Xi(X)]\bmod X^{2t+1},$$
$$\text{and } \Psi(X)=\Lambda(X)\Gamma(X).$$

Notice that you don't need **Equation 5**'s $\Psi(X)$ in later calculations. You need its derivative $\Psi'(X)$. So you can always instead implement the simpler equation that directly evaluates the derivative $\Psi'(X)$ according to **Equation 7**. You can implement **Equation 6** in a similar fashion as **Equation 3**. Again, you can adopt a parallel or serial approach according to your design's requirements. The approach you choose directly affects the FPGA resources.

THE FORNEY'S ALGORITHM

Using the following equations:

$$e_k=\frac{-X_k\Omega(X_k^{-1})}{\Psi'(X_k^{-1})},\text{ and}$$

$$f_k=\frac{-Y_k\Omega(Y_k^{-1})}{\Psi'(Y_k^{-1})},$$

you can compute the error/erasure magnitudes using the corresponding locations. The magnitudes are definitely field elements. The equations are identical, and by carefully choosing hardware, you can merge many operations together in the two equations. The previously discussed Chien-search unit directly evaluates the inverse error locations X_k^{-1}, and you also have access to the erasure locations Y_k beginning at the start of decoding. By rearranging the above two equations, you get:

$$e_k=\frac{-\Omega(X_k^{-1})}{X_k^{-1}\Psi'(X_k^{-1})},\text{ and}$$

$$f_k=\frac{-\Omega(Y_k^{-1})}{Y_k^{-1}\Psi'(Y_k^{-1})}.$$

The numerators determine the value of $\Omega(X)$ at inverse error and erasure locations, so the same unit can handle both numerators. The hardware to determine values of $\Omega(X)$ at X_k^{-1}/Y_k^{-1} resembles the hardware in **Figure 8** for the Chien-search unit. Similarly, the denominators also resemble each other. The hardware is the same for the numerator, but you multiply the result by the corresponding X_k^{-1}/Y_k^{-1}. Note that rearranging the equations eliminates the need to determine X_k from X_k^{-1}, although you must still evaluate Y_k^{-1} from Y_k. Finally, you can divide the numerator by the denominator by inverting the denominator and multiplying it by the numerator. A register stores the resulting error and erasure magnitudes. If a denominator term evaluates to a zero symbol, meaning there is no error or erasure at that location, you can assign a zero symbol to its inversion. Although mathematicians might disagree, assigning zero to the inversion won't introduce errors, and hence the error/erasure magnitude is zero at this X_k^{-1}/Y_k^{-1}.

As stated in the beginning of the decoder section, you retrieve the code word c(X) by adding the error and erasure polynomials to the received polynomial r(X). You, therefore, must also pipeline the polynomial r(X) separately in addition to passing it through the main decoding units (a zero symbol is pipelined for an erasure). In FPGAs, you can easily accomplish this pipelining by using RAMs as shift registers, which greatly reduces hardware. This unit's hardware implementation is tricky. Because you know error and erasure locations and magnitudes, your task is to construct the error and erasure polynomials. On paper, this task is easy, but in hardware it is difficult and costly. Assume that you have the following polynomials for error and erasures for Reed-Solomon(15,9) code: $e(X)=\propto^2X^8+\propto^5X^3$, $f(X)=\propto^{12}X^{10}+\propto X^2$, and $c(X)=r(X)+e(X)+f(X)$. These polynomials show that you add $\propto^2$, $\propto^5$, $\propto^{12}$, and $\propto$ to r(X) at the locations X^8,

X^3, X^{10}, and X^2 to retrieve $\alpha(X)$. Note that error/erasure direct or inverse locations can be any field elements except zero (for example, location $\propto^0$ corresponds to X^0 is a constant term in a polynomial), but because hardware is fixed to accommodate maximum error/erasure locations, a zero symbol in a register indicates the state of a register, not an error/erasure location. Also, all error and erasure locations are unique. The technique used here employs the inverse error/erasure locations to perform the operations in the above equations. The idea is to generate field symbols $\propto^{n-1}, \propto^{n-2}, \ldots \propto, \propto^0$ and dynamically compare every element with symbols of the register containing inverse error and erasure locations. Only a maximum of one comparison can succeed at a particular power of $\propto$, thereby enabling $\propto$'s corresponding magnitude symbol in the error and erasure magnitude registers. Logic controls the timing of pipelined data r(X) so that the coefficient with the proper power of X arrives at the adder input at the same time that its corresponding error/erasure magnitude (if any) arrives at the other input of the adder. (Remember that the decoder first outputs the coefficient with X^{n-1}.) **Figure 9** shows a simplified hardware diagram for the aforementioned technique.

Programmable-hardware devices are the best choice for Reed-Solomon-codec implementation, because these devices contain an abundance of the registers that the hardware-implementation process requires. They also allow you to implement a pipelined design. Secondly, parallel realization of the equations is possible and helps you meet speed constraints. You can easily map the equations used for multipliers and power/inversion circuits into the LUT architecture of FPGAs. Moreover, you can use RAMs to further reduce hardware resources. The algorithms presented above support erasure decoding, which consumes more resources than error-only decoding. The equations presented above are general in nature, and from them you can easily extract the equations for error-only decoding. Control also plays an important role in Reed-Solomon-codec implementation. The timing of various functional blocks should be immaculate to properly transfer data and results between various blocks. □

References

1. Blahut, Richard E, *Theory and Practice of Error Control Codes*, Addison Publishing Company, 1983.

2. Sweeney, Peter, *Error Control Coding: An Introduction*, Prentice Hall, 1991.

3. Sklar, Bernard, *Digital Communications Fundamentals and Applications*, Prentice-Hall International, Inc,1988.

4. Matache, Adina, *Encoding/Decoding Reed Solomon Codes*, Department of Electrical Engineering, University of Washington, http://drake.ee.Washington.edu/~adina/rsc/slide/slide.html.

Authors' biographies

Syed Shahzad Shah is the CEO of Chameleon Logics (Islamabad, Pakistan); Saqib Yaqub and Faisal Suleman are FPGA core engineers for the firm.