Application Report
SPRA686 - December 2000

Reed Solomon Decoder: TMS320C64x Implementation

Jagadeesh Sankaran — Digital Signal Processing Solutions

ABSTRACT

This application report describes a Reed Solomon decoder implementation on the TMS320C64x™ DSP family. Reed Solomon codes have been widely accepted as the preferred (ECC) error control coding scheme for ADSL networks, digital cellular phones and high-definition television systems. The reason for their widespread prevalence is their excellent robustness to burst errors. Programmable implementations of the Reed Solomon decoder offer the system designer the unique flexibility to trade off the data bandwidth and vary the error correcting capability that is desired for a given channel.

An efficient method to perform Reed Solomon decoding is the Peteren-Gorenstein-Zierler (PGZ) algorithm. Digital signal processors of the TMS320C6000™ DSP platform seek to exploit the data level and instruction level parallelism of algorithms by having multiple ALU units capable of working in tandem, to obtain extremely high levels of performance. The advanced set of code generation tools help the user in identifying the parallelism in the decoding algorithm for the multiple units of the DSP to exploit. This application note helps to overcome this challenge by showing the steps involved in the developing an efficient implementation of a complete (204,188,8) Reed Solomon decoder on the TMS320C64x DSP family.

This application report

- Identifies the various processing steps that are involved in the development of a Reed Solomon decoder.
- Discusses the instructions and features of the C6400 DSP used for implementing an efficient Reed Solomon decoder.
- Presents a complete implementation of a (204,188,8) Reed Solomon decoder on the C6400 DSP.

Contents

List of Figures

List of Tables

1 Introduction to Reed Solomon Codes

This section presents a brief overview of Reed Solomon codes and their associated terminology. It also discusses the advantages of a programmable implementation of the Reed Solomon decoder.

Reed Solomon codes are a particular case of non-binary BCH codes. They are extremely popular because of their capacity to correct burst errors. Their capacity to correct burst errors stems from the fact that they are word oriented rather than bit-oriented. A bit-oriented code such as a BCH code would treat this situation as many independent single-bit errors. To a Reed Solomon code, however a single error means any or all-incorrect bits within a single word. Therefore the RS (Reed Solomon) codes are designed to combat burst errors in a channel. In fact RS codes are a particular case of non-binary BCH codes.

The structure of a Reed Solomon code is specified by the following two parameters:

- The length of the code-word m in bits, often chosen to be 8,
- The number of errors to correct T.

A code-word for this code then takes the form of a block of m bit words. The number of words in the block is N, which is always equal to N = 2^m – 1 words, of which 2T words are parity or check words. For example, the m = 8, t = 3 RS code uses a block length of N = 255 bytes, of which 6 are parity and 249 are data bytes. The number of data bytes is usually referred to by the symbol K. Thus the RS code is usually described by a compact (N,K,T) notation. The RS code discussed above for example has a compact notation of (255,249,3). When the number of data bytes to be protected is not close to the block length of N defined by N = 2^m – 1 words a technique called shortening is used to change the block length. A shortened RS code is one in which both the encoder and decoder agree not to use part of the allowable code space. For example, a (204,188,8) code would only use 204 of the allowable 255 code words defined by the m = 8 Reed Solomon code. An error correcting code, such as an RS code, is said to be systematic if the user data to be encoded appears verbatim in the encoded code word. Thus a systematic (204,188,8) code would have the 188 data bytes provided by the user appearing verbatim in the encoded code word, appended by the 16 parity words of the encoder to form one block of 204 words. The choice of using a systematic code is merely from the point of simplicity as it lets the decoder recover the data bytes and strip off the parity bytes easily, because of the structure of the systematic code.

A programmable implementation of a RS encoder and decoder is an attractive solution as it offers the system designer the unique flexibility to trade-off the data bandwidth and the error correcting capability that is desired based on the condition of the channel. This can be done by providing the user the capability to vary the data bandwidth or the error correcting capability (T) that is required. The C6400 DSP offers a unique and rich instruction set that allows for the development of a high performance Reed Solomon decoder by minimizing the development time required without compromising on the flexibility that is desired. This application note will discuss in the following sections how to develop an efficient implementation of a complete (204,188,8) RS decoder solution on the C6400 DSP. This Reed Solomon code was chosen as an example because it is used widely as an FEC scheme in ADSL modems.

2 Galois Fields

This section presents a brief review of the properties of Galois fields. This section presents the utmost minimum detail that is required in order to understand RS encoding and decoding. A comprehensive review of Galois fields can be obtained from references on coding theory [1].

A field is a set of elements on which two binary operations can be performed. Addition and multiplication must satisfy the commutative, associative and distributive laws. A field with a finite number of elements is a finite field. Finite fields are also called Galois fields after their inventor [1]. An example of a binary field is the set {0,1} under modulo 2 addition and modulo 2 multiplication and is denoted GF(2). The modulo 2 addition and subtraction operations are defined by the tables shown in the following figure. The first row and the first column indicate the inputs to the Galois field adder and multiplier. For e.g. 1 + 1 = 0 and 1 * 1 = 1.

Modulo 2 Addition (XOR)

+	0	1
0	0	1
1	1	0

Modulo 2 Multiplication

*	0	1
0	0	0
1	0	1

Figure 1. Moulo 2 Finite Field Math

In general if p is any prime number then it can be shown that GF(p) is a finite field with p elements and that GF(p^m) is an extension field with p^m elements. In addition the various elements of the field can be generated as various powers of one field element α, by raising it to different powers. For example GF(256) has 256 elements which can all be generated by raising the primitive element 2 to the 256 different powers.

In addition, polynomials whose coefficients are binary belong to GF(2). A polynomial over GF(2) of degree m is said to be irreducible if it is not divisible by any polynomial over GF(2) of degree less than m but greater than zero. The polynomial $F(X) = X^2 + X + 1$ is an irreducible polynomial as it is not divisible by either X or X + 1. An irreducible polynomial of degree m which divides $X^{2m-1} + 1$, is known as a primitive polynomial. For a given m, there may be more than one primitive polynomial. An example of a primitive polynomial for m = 8, which is often used in most communication standards is $F(X) = 1 + X^2 + X^3 + X^4 + X^8$.

Galois field addition is easy to implement in software, as it is the same as modulo addition. For e.g. if 29 and 16 are two elements in GF(2^8) then their addition is done simply as an XOR operation as follows: 29 (11101) $\oplus$ 16 (10000) = 13 (01101).

Galois field multiplication on the other hand is a bit more complicated as shown by the following example, which computes all the elements of GF(2^4), by repeated multiplication of the primitive element α. To generate the field elements for GF(2^4) a primitive polynomial G(x) of degree m = 4 is chosen as follows $G(x) = 1 + X + X^4$. In order to make the multiplication be modulo so that the results of the multiplication are still elements of the field, any element that has the fifth bit set is brought back into a 4-bit result using the following identity $F(\alpha) = 1 + \alpha + \alpha^4 = 0$. This identity is used repeatedly to form the different elements of the field, by setting $\alpha^4 = 1 + \alpha$. Thus the elements of the field can be enumerated as follows:

$\{0,1,\alpha,\alpha^2,\alpha^3,1 + \alpha, \alpha + \alpha^2, \alpha^2 + \alpha^3, 1 + \alpha + \alpha^3,....1 + \alpha^3\}$

Since α is the primitive element for GF(2^4), it can be set to 2 to generate the field elements of GF(2^4) as {0,1,2,4,8,3,6,7,12,11,…9}.

3 Overview of the C6400 DSP

This section presents an overview of the C6400™ DSP. It discusses the specific architectural enhancements that have been made to significantly increase performance for Reed Solomon encoding and decoding.

The C6400 DSP is uniquely tailored for implementing Reed Solomon based error control coding because it provides hardware support for performing Galois field multiplies. In the absence of hardware to effectively perform Galois field math, previous DSP implementations made use of logarithms to perform multiplication in finite fields. This limited the performance of programmable implementations of Reed Solomon decoders on DSP architectures.

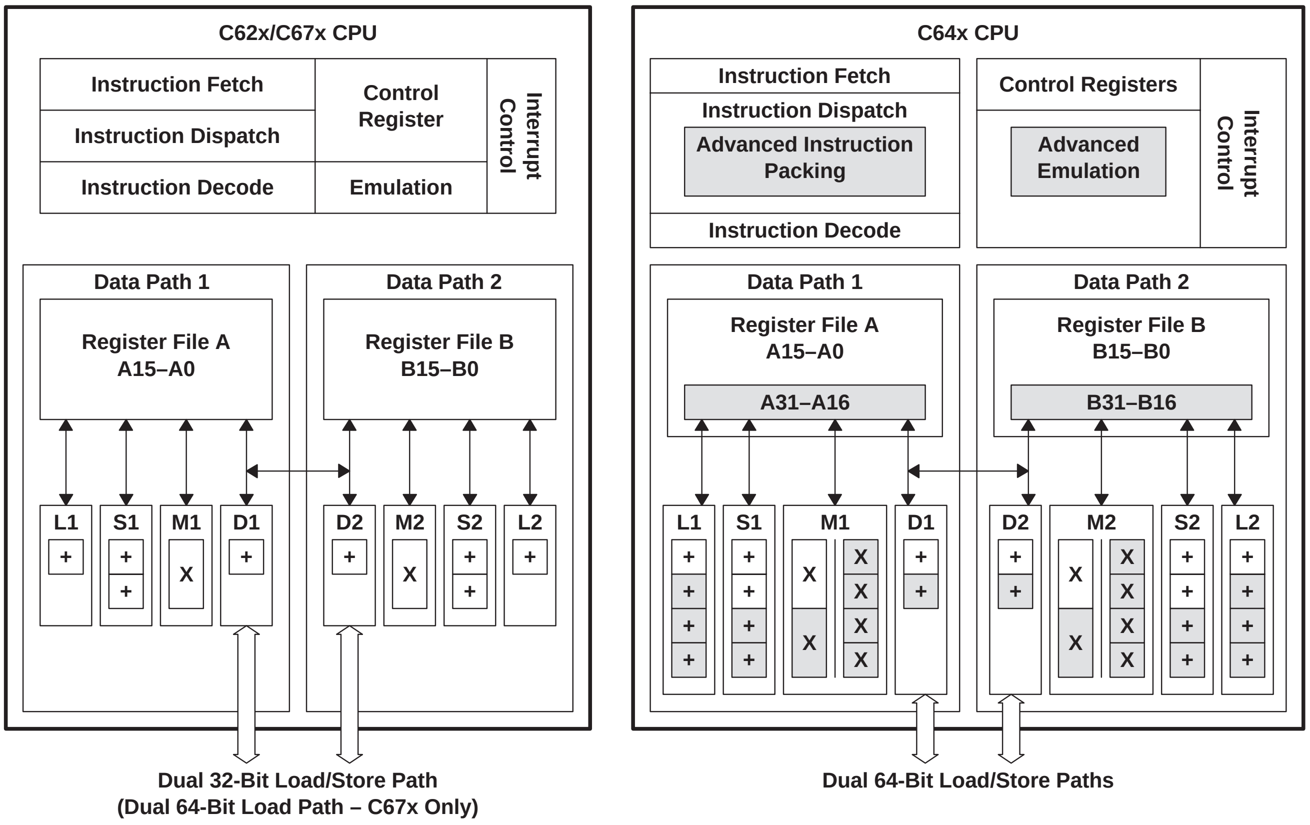

Figure 2. C62x/C67x and C64x CPU

The Galois field addition is performed by the use of the XOR operation, and the multiplication operation is performed by the use of the GMPY4 instruction. The C6400 DSP allows up to 24 8-bit XOR operations to be performed in parallel every cycle. In addition it has 64 general-purpose registers that allow the architecture to obtain extremely high levels of performance. The action of the Galois field multiplier is shown in the figure below. The Galois field multiplier accepts two integers, each of which contains 4 packed bytes and multiplies them as shown below to produce four packed bytes as an integer.

$C_0 = B_0 \otimes A_0, C_1 = B_1 \otimes A_1, C_2 = A_2 \otimes B_2, C_3 = B_3 \otimes A_3$, where $\otimes$ denotes Galois field multiplication.

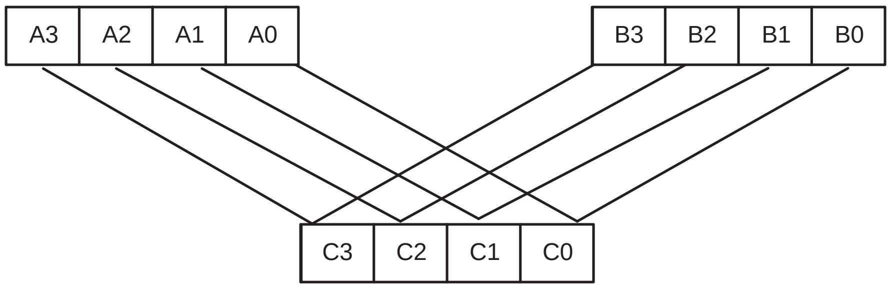

Figure 3. GMPY4 Operation on the C64x

The “GMPY4” instruction denotes that all four Galois field multiplies are being performed in parallel. The architecture can issue two such GMPY4s in parallel every cycle, thus performing up to eight Galois field multiplies in parallel. This provides the architecture the capability to attain new levels of performance for Reed Solomon based coding. In addition the Galois field to be used, can be programmed using the GFPGFR register. The ability to use these instructions directly from C by the use of “intrinsics” helps to considerably reduce the software development time.

Galois field division is not used often in finite field math operations, so that it can be implemented as a look-up table if required.

4 Examples of using GMPY4 for different GF(2^M)

The following C code fragment illustrates how the “gmpy4” instruction can be used directly from C to perform four Galois field multiplies in parallel. Previous DSPs that do not have this instruction, would typically perform the Galois field addition using logarithms. For example, two field elements a and b would be multiplied as a $\otimes$ b = exp[log[a] + log[b]]. It can be seen that three lookup-table operations have to be performed for each Galois field multiply. For some computational stages of the Reed-Solomon such as syndrome accumulate and Chien search one of the inputs to the multiplier is fixed, and hence one table look up can be avoided, thereby allowing 2 Galois field multiplies every cycle. The architectural capabilities of the C6400 directly give it a 4x boost in terms of Galois field multiplier capability. The C6400 DSP allows up to eight Galois field multiplies to be performed in parallel, by the use of two gmpy4 instructions, one on each data-path. This example performs Galois field multiplies in GF(256) with the generator polynomial defined as follows: $G(X) = 1 + X^2 + X^3 + X^4 + X^8$. The generator polynomial can be written out as a hex pattern as shown below (1+4+8+16) = 29 = 0x1D:

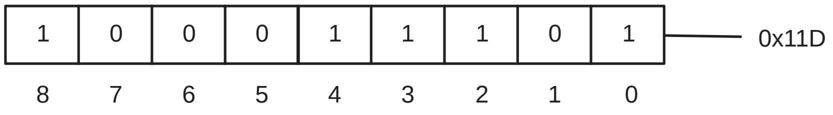

Figure 4. Default Polynomial for GFPGFR

The device comes up powered with the G(x) shown above as the generator polynomial for GF(256), as most communications standards make use of this polynomial for Reed Solomon based coding. If some other generator polynomial or some other GF(2^m) is desired then the user should initialize the GFPGFR (Galois field polynomial generator) [5]. The behavior of the GMPY4 instruction is controlled by programming the GFPGFR (Galois field polynomial generator). Two parameters are required to program the GFPGFR namely size and polynomial generator. The size field is three bits and is one smaller than the degree of the generator polynomial, in this case 8 – 1 = 7. The generator polynomial is an eight bit field and is computed from the 8 LSBs of the hex pattern represented by 0x11D in hexadecimal. The 9th bit is always 1 for GF(256) and hence only the 8 LSBs need to be represented as the generator polynomial in the control register. The behavior of the GMPY4 instruction is controlled by programming GFPGFR (Galois field polynomial generator). Two parameters are required to program the GFPGFR namely size and polynomial generator. The size field is seven bits and is one smaller than the degree of the generator polynomial , in this case 8 – 1 = 7. The generator polynomial is an eight bit field and is computed from the eight LSBs of the hex pattern represented by 0 x 1D in hexadecimal. The ninth bit is always 1 for GF(256) and hence only the eight LSBs need to be represented as the generator polynomial in the control register.

Example 1. Example Showing Galois Field Multiplies on a DSP

```
inline int GMPY( int op1, int op2 )
{
    /*------------------------------------------------------*/
    /* Operands a0 and b0 are in polynomial representation.  */
    /* GF multiplication is in power representation.         */
    /*------------------------------------------------------*/
    int t0 = exp_table2[log_table[op1] + log_table[op2]];
    if ((op1 == 0) || (op2 == 0)) t0 = 0;
    return(t0);
}
void main()
{
    int symbol_word0 = 0xFFCADEBA;
    int symbol_word1 = 0xABDE876E;
    /*-------------------------------------------------------*/
    /* Previous DSP’s would use logarithm tables to implement */
    /* Galois field multiplication.                           */
    /*-------------------------------------------------------*/
    unsigned char byte0 =  GMPY(0xBA, 0x6E);
    unsigned char byte1 =  GMPY(0xDE, 0x87);
    unsigned char byte2 =  GMPY(0xCA, 0xDE);
    unsigned char byte3 =  GMPY(0xFF, 0xAB);
   /*--------------------------------------------------------*/
   /* C6400 uses dedicated instruction accessible from C as   */
   /* shown below, and performs the four multiplies in        */
   /* parallel.                                               */
   /* symbol_word0 = 0xFFCADEBA symbol_word1 = 0xABDE876E     */
   /* prod_word=(0xFF *0xAB)(0xCA*0xDE)(0xDE*0x87)(0xBA*0x6E))*/
   /*--------------------------------------------------------*/
   int prod_word    = _gmpy4(symbol_word0, symbol_word1);
}
```

The example shown above illustrates the use of the _gmpy4 instruction in C64x DSP with the default generator polynomial in GF(256) from C code. The example shown on the next page illustrates how to use a different generator polynomial for the case of GF(256). In this case the generator polynomial is defined as follows: $G(x) = 1 + X^3 + X^5 + X^6 + X^8$ is shown below.
In this case the GFPGFR needs to be programmed, by writing a control word to the register. Two parameters are required to program this control register, size and polynomial generator. The size is one smaller than the degree of the generator polynomial, in this case 8 – 1 = 7. The generator polynomial is the hexadecimal pattern of the generator polynomial, as shown below. Note that this is an 8 bit field since X^8 is always assumed to be 1. This is shown in Figure 5 (8 bit field, bit 9 is always assumed to be 1)

$G(x) = 1 + X^3 + X^5 + X^6 + X^8 = (1 + 8 + 32 + 64) = 105 = 0x69$

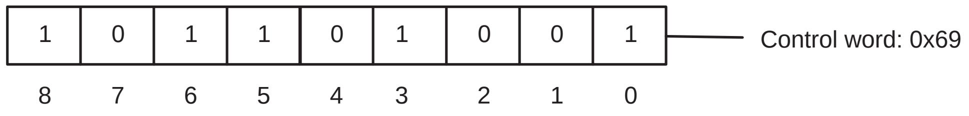

Figure 5. Generator Polynomial in GFPGFR for $G(x) = 1 + X^3 + X^5 + X^6 + X^8$

Example 2. Example Showing Use of GMPY4 Instruction on C64x DSP

```
#include <stdio.h>
#include <stdlib.h>
#include <c6x.h>
void main()
{
  /*---------------------------------------------------------*/
  /* Default Gen. Polynomial is 1 + X^2 + X^3 + X^4 + X^8     */
  /* Default Control word is: 700001D, 2 * 128 =(16+8+4+1)=29 */

  /* Control word for G(X): 1 + X^3 + X^5 + X^6 + X^8  = 0x69 */
  /* Field size: Polynomial degree – 1 = 8 – 1   = 7          */
  /*  Control word: 7000069,  2 *128  = (64+32+8+1) = 105     */

  /*---------------------------------------------------------*/
  /*----------------------------------------------------------*/
  /* Capture Default control word and Perform Galois field mpy */
  /*----------------------------------------------------------*/

  unsigned int control_word = 0x7000069;

  printf(”Default GFPGFR is %x \n”, GFPGFR);
  printf(”2  GMPY 128 is %d \n”, _gmpy4(2,128));

  /*---------------------------------------------------------*/
  /* Update GFPGFR and then perform Galois field multiply     */
  /*---------------------------------------------------------*/

  GFPGFR = control_word;
  printf(”GFPGFR is %x \n”, GFPGFR);
  printf(”2  GMPY 128 is %d \n”, _gmpy4(2,128));
}
```

The next example shown below illustrates how to use the _gmpy4 instruction for field sizes other than GF(256). The generator polynomial chosen in this case uses the Galois field GF(64) and is defined as G(x) = $1 + X + X^2 + X^5 + X^6$. In this case each field element is represented using 6 bits. In this case the generator polynomial is left justified so that the 8th bit corresponding to X^8 is always assumed to be 1. The control word is derived from this left shifted polynomial which is denoted as G’(x) = $X^8 + X^7 + X^4 + X^3 + X^2$. The control word can now be derived as shown below. The field size in this case is derived from the original polynomial as 6 – 1 = 5, and not from the left shifted polynomial G’(x). The control word is shown in Figure 6 (7 bit field, bit 8 is always assumed to be 1) (4 + 8 + 16 + 128) = 156 = 0x69:

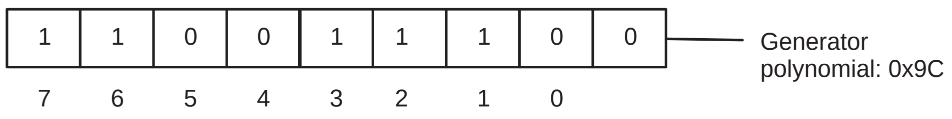

Figure 6. Programming GFPGFR for the Generator Polynomial G(x) = $1 + X + X^2 + X^5 + X^6$ for GF(64)

Example 3. Example Showing How to Program GFPGFR for a Generator Polynomial in GF(64)

```
#include <stdio.h>
#include <stdlib.h>
#include <c6x.h>

void main()
{

  /*----------------------------------------------*/
  /* Consider: GF(2^6) = 1 + X + X^2 + X^5 + X^6  */
  /* Left justify the G by X^2 to get:            */
  /* X^8 + X^7 + X^4 + X^3 + X^2                  */
  /* 1 1001 1100 = 9C                             */
  /* Field size: 5  = (order of poly – 1)         */
  /* 2 * 32 = 2 ^ 6 = 32 + 4 + 2 + 1 = 39         */
  /*                                              */
  /* The elements of the field need to be left    */
  /* justified by 2, hence 2*32 = _gmpy4(8,128)   */
  /* Result needs to be right shifted by 2 to     */
  /* obtain the final result.                     */
  /* All computations can be done with the set of */
  /* numbers that have been left justified, to    */
  /* produce a 8 bit number and finally the       */
  /* numbers can be right shifted to convert them */
  /* back to 6 bit numbers.                       */
  /*----------------------------------------------*/
  unsigned int control_word = 0x500009C;

  GFPGFR = control_word;
  printf(”2  GMPY4 32 is %d \n”, _gmpy4(8,128) >> 2);
}
```

5 Peterson-Gorenstein-Zierler (PGZ) Reed Solomon Decoder

The Peterson-Gorenstein-Zierler algorithm for decoding Reed Solomon codes consists of four steps as shown below:

1. Syndrome Computation
2. Berlekamp Massey Algorithm for solving the error locator polynomial
3. Chien Search Algorithm for solving for the roots of the error locator polynomial
4. Forney algorithm for computing the error magnitudes

The algorithmic details , and the computational complexity of each of these algorithms is discussed below.

5.1 Syndrome Computation

The syndrome accumulate is the first step in the Reed-Solomon decoding process. This is done to detect if there are any errors in the received code word. Since the code word is generated by multiplying with the generator polynomial, if the received code word is error free then its modulus with respect to the generator polynomial should evaluate to zero.

Let C be the code word without any errors, R be the received code word and E be any error that the channel introduces. The encoder takes the data D to be encoded and encodes it as a codeword C, which is a multiple of the generator polynomial G for the case of systematic codes by adding 2T parity words after the K data bytes. The syndrome S can then be defined as S = R mod G and is the remainder obtained by dividing the received code word by the generator polynomial G. The received code word R can then be expressed as R = C + E where E is the associated channel error. Since C is a multiple of G by definition, S actually evaluates to the following:

S = E mod G.

Conceptually this amounts to computing the Fourier transform of the received message at the 2T powers of the primitive element β. Evaluating the Fourier transform of the error polynomial at these 2T locations and checking for spectral nulls lets us know if the received code word is error free. For Galois fields of the form GF(2^m) the primitive element β is 2. Thus the 2T powers of β can be listed as {2, 4, 8, 16, 32, 64, 128, 29, 58, 116, 232, 205, 135, 19, 38} for the case of $G(X) = 1 + X^2 + X^3 + X^4 + X^8$. Thus for the case of a (N, K, T) code like a (204,188,8) GF(256) code there will be 16 (2*T) syndromes where the symbols stand for the following:

N is the number of words in the received code word.

K is the number of data bytes to be encoded.

T is the number of byte errors that the Reed Solomon code can correct.

2*T is the number of parity bytes added to the code,

N = K + 2*T,   GF(256): stands for a Galois field with 256 elements.

In this case the primitive element is 2 and all the elements in the field are generated as various powers of this primitive element by definition. The 16 syndromes can be expressed as follows:

S_i = R modG where i = (0,1,2,3,...15).

The received code word may be expressed in polynomial form as follows:,

$R_i = r_0 x^{N-1} + r_1 X^{N-2} + \ldots + r_{N-1}$

where the length of the received code word is N. Let the first 2T powers of beta be specified as follows:

Beta = $\{\beta_0, \beta_1, \ldots \beta_{15}\}$ where $\beta_i = (\beta^i)$ is the i'th power of β. The 16 syndromes can now be expanded as follows:

$$S_0 = r_0 \beta_0^{N-1} + r_1 \beta_0^{N-2} + r_2 \beta_0^{N-3} + \ldots + r_{N-2} \beta_0^1 + r_{N-1}$$

$$S_1 = r_1 \beta_1^{N-1} + r_1 \beta_1^{N-2} + r_2 \beta_1^{N-3} + \ldots + r_{N-2} \beta_1^1 + r_{N-1}$$

[illegible]

$$S_{15} = r_0 \beta_{15}^{N-1} + r_1 \beta_{15}^{N-2} + r_2 \beta_{15}^{N-3} + \dots + r_{N-2} \beta_{15}^{1} + r_{N-1}$$

Since all the elements involved in the computation belong to a Galois field the operations of addition and multiplication are also done over finite fields. Addition over finite fields is merely an XOR operation while multiplication over finite fields is a Galois field multiply. Thus

$\beta i = (\beta * \beta * \ldots itimes)$ where * denotes Galois field multiplication.

5.2 Berlekamp Massey Algorithm

It has been shown in the previous section, on syndrome computation, that the syndromes merely depend on the error polynomial. It is assumed that the error pattern e(X) has γ errors at the locations $X^{j_1 1} + X^{j2} + X^{j3} + \dots X^{j\gamma}$. The errors are assumed to have unit magnitude to keep the following equations simple to understand. Since $S_i = E(a^i)$, the syndromes can be written as shown below:

$$S_0 = \alpha^{j1} + \alpha^{j2} + \alpha^{j3} + \cdots + \alpha^{j\gamma},$$

$$S_1 = (\alpha^{j1})^2 + (\alpha^{j2})^2 + (\alpha^{j3})^2 \cdots + (\alpha^{j\gamma})^2,$$

$$S_{2T-1} = (\alpha^{j1})^{2T} + (\alpha^{j2})^{2T} + (\alpha^{j3})^{2T} + \dots + (\alpha^{j\gamma})^{2T},$$

where $[\alpha^{j1}, \alpha^{j2}, \dots, \alpha^{j\gamma}]$ are unknown. Any method for solving these equations is a decoding algorithm for the Reed Solomon codes. Once $[a^{j1}, \alpha^{j2}, \dots, \alpha^{j\gamma}]$ the values $j_1, j_2, \dots j^{\gamma}$ can be found, to tell us the locations where the error occurred. In general there are many solutions, to the equations shown above, however the solution that yields an error pattern with the smallest number of errors is the right solution, and represents the most probable error pattern e(X) caused by channel noise. Let us denote $\beta_1 = \alpha^1$ where $1 \leq l \leq \gamma$, then the equation shown above can also be equivalently expressed as:

$$S_0 = \beta_1 + \beta_2 + \dots + \beta_\gamma,\ S_1 = \beta_1^2 + \beta_2^2 + \dots + \beta_\gamma^2, \dots, S_{2T-1} = \beta_1^{2T} + \beta_2^{2T} + \beta_\gamma^{2T}.$$

The error locator polynomial can now be defined as follows:

Lambda(X) = $(1 + \beta_1 X)(1 + \beta_2 X)(1 + \beta_3 X)\dots(1 + \beta_\gamma X)$. It can be seen that the roots of this polynomial are the inverses of the error locations, and can be expanded as shown, *Lambda*(X) = $\sigma_0 + \sigma 01X + \sigma_2 0X^2 + \dots + \sigma_\gamma X\gamma$. The coefficients of $\sigma(X)$ can be related to the β's as shown below:

$$\sigma 0 = 1, \sigma_1 = \beta_1 + \beta_2 + \dots + \beta_\gamma, \sigma_2 = \beta_1\beta_2 + \beta_2\beta_3 + \dots + \beta_{\gamma-1}\beta_\gamma.$$

These σ_i's are known as elementary power symmetric functions of β_i's. In fact relating them to the syndromes gives the set of equations known as Newton's identities that are used by Berlekamp-Massey algorithm.

$$S_1 + \sigma_1 = 0,\ S_2 + \sigma_1 S_1 + 2\sigma_2 = 0,\ S_3 + \sigma_1 S_2 + 3\sigma_3 = 0, \dots S_{\gamma 1} + \sigma_2 S_{\gamma-1} + \sigma_2 S_{\gamma-2} + \dots + \gamma S_\gamma = 0$$

The Berlekamp-Massey serves as an iterative algorithm to solve for $\sigma(X)$ and is outlined below. The steps a) through i) are iterated 2T times in the Berlekamp algorithm.

a. Let the syndromes be denoted $S_1, S_2, S_3, \ldots S_{2T}$
b. Initialize the algorithm variables: k=0, $\lambda^{(0)}(x) = 1$, L = 0, and $T(x) = x$, where k is the degree of Lambda(x) at this iteration.
c. Set k = k+1, Compute the discrepancy $\Delta^k(x)$ as follows:

$$\Delta^k = S_k - \sum_{i=1}^{L} \lambda_i^{k-1} S_{k-1}$$

d. If $\Delta^k = 0$, then go to step h
e. Modify Lambda polynomial as follows: $\lambda^k(x) = \lambda^{(k-1)} - \Delta^k T(x)$
f. If ($2L \geq k$) then go to step h
g. Set L = k and $T(x) = \lambda^{(k-1)}(x)/\Delta^k$
h. Set $T(x) = x.T(x)$
i. If present iteration k is < 2T then go to c

It can be seen that the algorithm tries to iteratively solve for the error locator polynomial by solving one equation after another and updating the error locator polynomial. If it turns out that it cannot solve the equation at some step, then it computes the error and weights it by the last non-zero discriminant found, and delays the weighted result to increase the polynomial degree by 1.

5.3 Chien Search

The Chien search algorithm is an efficient technique for determining the zeroes of the error locator polynomial of degree lam_deg. The error locator polynomial is a polynomial whose roots are constructed to be the reciprocal of the locations where the errors occurred. The error locator polynomial is typically obtained by solving for a minimum degree solution that satisfies the Newton's power symmetric equations.

The error locator polynomial is thus a polynomial of degree υ depending on the number of errors that have occurred in the received code word. There is no closed form solution for solving for the roots of a υ^{th} degree polynomial. Since the root obviously has to be one of the elements of the field, an exhaustive search by substituting each of the field elements in the error locator polynomial is the only way out. Chien search is an effective algorithm to do this exhaustive search in an efficient manner.

5.4 Forney Algorithm

This algorithm addresses the problem of calculating the error magnitudes given the values of the error polynomial Lambda(X) and the syndromes. The Chien search determines the roots of the error locator polynomial, which enables us to construct a set of linear equations, to find the non-zero value of the coefficients. The error polynomial is non-zero except for the υ errors. The Fourier transform of the non-zero coefficients gives the following:

$$E_i = e(a^i) = \sum_{j=1}^{v} e_i \, a^{il_j}$$

This is a system of υ linear equations in υ unknowns, the unknown coefficients at the known error locations. In matrix form this can be written as:

$$\begin{bmatrix} {\alpha^{1\iota}}_1 & {\alpha^{1\iota}}_2 & \alpha^{1\upsilon} \\ {\alpha^{2\iota}}_1 & {\alpha^{2\iota}}_2 & {\alpha^{2\iota}}_\upsilon \\ \cdots & \cdots & \cdots \\ {\alpha^{\upsilon\iota}}_1 & \alpha^{\upsilon\iota_2} & {\alpha^{\upsilon\iota}}_\upsilon \end{bmatrix} \begin{bmatrix} e_{1\iota} \\ e_{\iota 2\upsilon} \\ \cdots \\ e\iota_\upsilon \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdots \\ E_\upsilon \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_\upsilon \end{bmatrix}$$

This can be solved using traditional methods of matrix inversion which would take $O(\upsilon^3)$, however the Forney algorithm presents a faster method to do the same. In order for this to be done, a new polynomial is introduced, which is referred to as the error evaluator polynomial $\Omega(X)$. From the definition of the error locator polynomial Lambda(x) and the error polynomial e(x) satisfy the following equation in time domain:

$e_k \gamma_k$ for k = 0,1,2,..,n–1. This equation can be transformed into the frequency domain as shown, E(x).Lambda(x) = 0 for $X = \alpha^0, \alpha^1, \cdots, \alpha^{2T}$. Multiplication of polynomials is equivalent to linear convolution and since the field is finite, it actually ends up being a circular convolution. Therefore the equation may be rewritten as shown: E(x).Lambda(x)mod(x^n–1) = 0. This implies that the product is a multiple of (x^n–1), the multiple being the error evaluator polynomial. Thus the equation can be re-written as follows: E(x).Lambda(x) = Ω(x)(x^n–1).

Therefore the equation can be solved for the error evaluator polynomial as follows:

$$E(x) = \frac{\Omega(x)(x^n - 1)}{Lambda(x)}$$

The coefficients of the error polynomial can be calculated by evaluating the inverse Fourier transform at the known error locations that can be computed from the results of Chien search as follows:

$$e_k = \frac{1}{n} \frac{\Omega(x)(x^n - 1)}{Lambda(x)} \text{ for } x = \alpha^{-k} \text{ for } k = l_1, l_2, \ldots, l_y.$$

However these are precisely the locations where both the numerator and the denominator evaluate to zero. Therefore applying L'Hospitals rule, it can be derived that

$$e_k = \frac{a^k \Omega(a^{-k}}{Lambda'(a^{-k})} \text{ for } x = \alpha^{-k} \text{ for } k = l_1, l_2, \ldots l_\lambda.$$

where Lambda' is the derivative of the error locator polynomial Lambda(x). It turns out that the derivative of any polynomial in GF(2) is simple to compute as odd powers can be zeroed out and even powers shifted down by one. This enables easy computation of the error magnitudes. With the knowledge of these four algorithms the decoder implementation on the C6400 DSP can be developed.

5.5 C6400 Implementation of Reed Solomon Decoder

The four steps required to perform Reed Solomon decoding were discussed earlier. This section focuses on how these different pieces can be implemented on the C6400 DSP to make full use of the available resources. In each algorithm we will aim to maximize the number of Galois field multiplies we can use. The Galois field multiply has a latency of 4 cycles, and hence certain loops need to be unrolled to take full advantage of the latency, so that two GMPY4s may be issued every cycle. In addition byte quantities need to be packed together to form a packed integer to serve as the 4 inputs to the multiplier.

5.6 Syndrome Computation:

The syndrome computation algorithm is first developed for the case of T = 8. This case involves computing 16 syndromes. This section also examines how the code developed can be re-used for the case of T = 4. This section also discusses how to implement the algorithm for odd N.

5.7 Case of T = 8

Since the GF(256) has 256 elements each element of the field can be expressed as a byte (8 bits). The Galois field multiplier that accepts two 32 bit quantities as input , each of which contains 4 packed field elements and returns the 4 output results of the multiply as a 32 bit quantity containing 4 packed field elements together. The syndrome can be formally defined as follows:

$S_i = R \ \mathrm{mod} G$ where i = (0,1,2,3,...15).

The received code word may be expressed in polynomial form as follows:

$R_i = r_0 x^{N-1} + r_1 x^{N-2} + ... + r_{N-1}$

where the length of the received code word is N. For the case of (204,188,8) code N is equal to 204. Let the first 2T powers of beta be specified as shown below, where Beta = $\{\beta_0, \beta_1, ...\beta_{15}\}$ where abs β_i^j is the j'th power of the i'th root of the generator polynomial. The 16 syndromes can now be expanded as follows:

$$S_0 = r_0 \beta_0^{N-1} + r_1 \beta_0^{N-2} + r_2 \beta_0^{N-3} + ... + r_{N-2} \beta_0^1 + r_{N-1}$$

$$S_1 = r_0 \beta_1^{N-1} + r_1 \beta_1^{N-2} + r_2 \beta_1^{N-3} + ... + r_{N-2} \beta_1^1 + r_{N-1}$$

$$S_{15} = r_0 \beta_{15}^{N-1} + r_1 \beta_{15}^{N-2} + r_2 \beta_{15}^{N-3} + ... + r_{N-2} \beta_{15}^1 + r_{N-1}$$

It can be seen that computing the syndromes amounts to polynomial evaluation at the roots as defined by Beta. This is done recursively as follows using Horner's rule. Using Horner's rule for example, $1 + x + x^2 + x^3 + x^4$ can be written recursively as shown below:

$1 + x + x^2 + x^3 + x^4 = (x(x(x(x + 1) + 1) + 1) + 1$ and serves as an efficient way of computing polynomial equations. We shall denote these as β_{0123}, β_{4567}, β_{891011} and $\beta_{12131415}$. The recursive computation of S_0 is shown below:

$$S_0 = (((((r_0\beta + r_1)\beta + r_2) + \beta) + ... r_{N-2})\beta + r_{N-1})$$

The computations that need to be performed can be seen from the following C code description of the algorithm:

```
int syn_acc_cn       (  const unsigned char *restrict  byte_i,
                        unsigned char  *restrict s1,
                        unsigned char  *restrict s2,
                        const unsigned char *restrict  beta,
                        int RS_N, int RS_2T
                     )
```

```
{
    int i,   index,  s1_i;
    int beta_i, tmp1, ret_val;
    int ret_val = 0;
    /*----------------------------------------------------------------------*/
    /* The syndrome accumulate routine contains two loops. The outer loop    */
    /* iterates 2T times  and evaluates 2T syndromes. The inner loop index   */
    /* iterates for the number of elements in the finite field and performs  */
    /* the syndrome calculation.                                             */
    /* tmp1 multiplies two terms together and s_I gets updated by adding the */
    /* next byte. Two iterations of this code are traced as shown below:     */
    /*                                                                       */
    /* Iteration 1: tmp1 = r0 * beta0  s1_i = r0 * beta0 + r1                */
    /* Iteration 2: tmp1 = (r0 * beta0 + r1) * beta0                         */
    /*  s1_I = r0*beta0*beta + r1*beta0 + r2                                 */
    /*----------------------------------------------------------------------*/

    for (i = 0; i < RS_2T; i++)
    {
         s1_i   = byte_i[0];
         beta_i = beta[i];
         for (index = 1; index < RS_N ; index++)
         {
              tmp1   = _gmpy4(s1_i, beta_i];
              s1_i = ( tmp1 ^ byte_i[index] );
         }
          s1[i] = s1_i;
    }

    /*-----------------------------------------------------------------------*/
    /* If all the syndromes are non–zero then the return value is set to zero */
    /* .  This indicates that the received codeword does not have any errors. */
    /* If the received codeword has any one of its 2T syndromes to be non–zero*/
    /* then the return value is set to one to indicate error.                 */
    /*-----------------------------------------------------------------------*/
    for (i = 0; i < RS_2T; i++)
    {
        if (s1[i] > 0)  ret_val = 1;
    }
    return(ret_val);
}


   /* ======================================================================= */
   /*  End of file:  syn_acc_c.c                                              */
   /* ----------------------------------------------------------------------- */
   /*            Copyright (c) 2000  Texas Instruments, Incorporated.         */
   /*                             All Rights Reserved.                        */
   /* ======================================================================= */
```

It can be clearly seen that the syndrome computation involves two loops, an outer loop that iterates once for every syndrome and an inner loop that iterates over all the field elements. In order to obtain the best performance from the architecture, we need to unroll and compute all the 2T (16 in this case) in parallel. The various powers of β can be read in as packed 8 bit quantities into a 32 bit register. In order to do this we shall use an approach similar to that of a radix 4 FFT. The received code word is read starting at locations 0, N/4, N/2 and 3N/ 4. Horner's rule is now applied recursively to all four of these packed beta-words using the input data read in all the four locations N/4 – 1 times. In the following equations $R_i = r_i\, r_i\, r_i\, r_i$ as a packed 32 bit quantity. In addition each term of the form $R_i\, \beta_{ijkl} = R_i * \beta_i, R_i * \beta_j, R_i * \beta_k, R_i * \beta_l$ where * denotes the gmpy4 instruction. The 16 syndromes are evaluated partially using the following 16 words as shown below. The operations required for the first four syndromes are contained in the terms s1_word0, s2_word0, s3_word0 and s4_word0. It can be seen that these are the partial results for the first four syndromes S_0, S_1, S_2, S_3.

$$s1\_word0 = R_0\beta_{0123}^{N/4-1} + R_1\beta_{0123}^{N/4-2} + \ldots\ldots\ldots\ldots + R_{N/4-2}\beta_{0123}^{1} + R_{N/4-1}$$

$$s2\_word0 = R_{N/4}\beta_{0123}^{N/4-1} + R_{N/4+1}\beta_{0123}^{N/4-2} + \ldots\ldots. + R_{N/2-2}\beta_{0123} + R_{N/2-1}$$

$$s3\_word0 = R_{N/2}\beta_{0123}^{N/4-1} + R_{N/2+1}\beta_{0123}^{N/4-2} + \ldots.. + R_{3N/4-2}\beta_{0123} + R_{3N/4-1}$$

$$s4\_word0 = R_{3N/4}\beta_{0123}^{N/4-1} + R_{3N/4+1}\beta_{0123}^{N/4-2} + \ldots.. + R_{N-2}\beta_{0123} + R_{N-1}$$

A comparison with the definition of the syndromes shows that these partial results need to be re-combined as shown below to yield the first four syndromes as a packed word S0123:

$$s_{0123} = s1\_word0\,\beta_{0123}^{3N/4} + S2\_word0\,\beta_{0123}^{N/2} + s3\_word0\beta_{0123}^{N/4} + s4\_word0$$

The equations that are shown below, outline a similar set of steps for the other syndromes, wherein four words of the type{s1_wordj, s2_wordj,s3_wordj,s4_wordj} contain partial results of the syndromes {4*j, 4*j+1,4*j+2,4*j+3}. The recombination equations for these partial results is also shown right after the expressions for the partial results. These partial words contained in 16 words are re-combined to form 4 words, where each word contains four syndromes. The re-combination equations are done outside the main loop. The main loop iterates N/4 – 1 times and produces 16 words with partial results.

$$s1\_word1 = R_0\beta_{4567}^{N/4-1} + R_1\beta_{4567}^{N/4-2} + \ldots\ldots\ldots\ldots + R_{N/4-2}\beta_{4567}^{1} + R_{N/4-1}$$

$$s2\_word1 = R_{N/4}\beta_{4567}^{N/4-1} + R_{N/4-1}\beta_{4567}^{N/4-2} + \ldots\ldots. + R_{N/2-2}\beta_{4567} + R_{N/2-1}$$

$$s3\_word1 = R_{N/2}\beta_{4567}^{N/4-1} + R_{N/2+1}\beta_{4567}^{N/4-2} + \ldots.. + R_{3N/4-2}\beta_{4567} + R_{3N/4-1}$$

$$s4\_word1 = R_{3N/4}\beta_{4567}^{N/4-1} + R_{3N/4+1}\beta_{4567}^{N/4-2} + \ldots.. + R_{N-2}\beta_{4567} + R_{N-1}$$

$$s_{4567} = s1\_word1\beta_{4567}^{3N/4} + s2\_word1\beta_{4567}^{N/2} + s3word1\beta_{4567}^{N/4} + s4\_word1$$

$$s1\_word2 = R_0\beta_{891011}^{N/4-1} + R_1\beta_{891011}^{N/4-2} + \dots\dots\dots\dots + R_{N/4-2}\beta_{891011}^{1} + R_{N/4-1}$$

$$s2\_word2 = R_{N/4}\beta_{891011}^{N/4-1} + R_{N/4+1}\beta_{891011}^{N/4-2} + \dots\dots. + R_{N/2-2}\beta_{891011} + R_{N/2-1}$$

$$s3\_word2 = R_{N/2}\beta_{891011}^{N/4-1} + R_{N/2+1}\beta_{891011}^{N/4-2} + \dots.. + R_{3N/4-2}\beta_{891011}^{1} + R_{3N/4-1}$$

$$s4\_word2 = R_{3N/4}\beta_{891011}^{N/4-1} + R_{3N/4+1}\beta_{891011}^{N/4-2} + \dots.. + R_{N-2}\beta_{891011}^{1} + R_{N-1}$$

$$s_{891011} = s1\_word2\beta_{891011}^{3N/4} + s2\_word2\beta_{891011}^{N/2} + s3word2\beta_{891011}^{N/4} + s4\_word2$$

$$s1\_word3 = R_0\beta_{12131415}^{N/4-1} + R_1\beta_{12131415}^{N/4-2} + \dots\dots\dots\dots + R_{N/4-2}\beta_{12131415}^{1} + R_{N/4-1}$$

$$s2\_word3 = R_{N/4}\beta_{12131415}^{N/4-1} + R_{N/4+1}\beta_{12131415}^{N/4-2} + \dots\dots. + R_{N/2-2}\beta_{12131415} + R_{N/2-1}$$

$$s3\_word3 = R_{N/2}\beta_{12131415}^{N/4-1} + R_{N/2+1}\beta_{12131415}^{N/4-2} + \dots.. + R_{3N/4-2}\beta_{12131415} + R_{3N/4-1}$$

$$s4\_word3 = R_{3N/4}\beta_{12131415}^{N/4-1} + R_{3N/4+1}\beta_{12131415}^{N/4-2} + \dots.. + R_{N-2}\beta_{12131415} + R_{N-1}$$

$$s_{12131415} = s1\_word3\beta_{12131415}^{3N/4} + s2\_word3\beta_{12131415}^{N/2} + s3\_word3\beta_{12131415}^{N/4} + s4\_word3$$

5.8 Case of T = 4

In the case of T = 4, there are 8 syndromes to compute. Instead of using a separate scheme to implement the calculation the same set of computational elements will be used as in the case of T = 8. However, in this case, the received data will be read at 8 locations starting from 0, N/8, N/4, 3N/8, N/2, 5N/8, 7N/8. This approach is similar to reading the input data for a radix 8 FFT. Horner's rule is applied recursively N/8 –1 times to compute these partial results. Once again, the set of equations involved for the first four syndromes are shown. For the case of T = 4, eight partial results are computed that need to be re-combined in a two-step re-combination. The first step re-uses the combination rules that were used for T = 8, so that the second step of re-combination can be coded as one additional step.

$$s1\_word0 = R_0\beta_{0123}^{N/8-1} + R_1\beta_{0123}^{N/8-2} + \dots\dots\dots + R_{N/8-2}\beta_{0123} + R_{N/8-1}$$

$$s2\_word0 = R_{N/4}\beta_{0123}^{N/8-1} + R_{N/4+1}\beta_{0123}^{N/8-2} + \dots. + R_{N/8-2}\beta_{0123} + R_{3N/8-1}$$

$$s3\_word0 = R_{N/2}\beta_{0123}^{N/8-1} + R_{N/2+1}\beta_{0123}^{N/8-2} + \dots.. + R_{5N/8-2}\beta_{0123} + R_{5N/8-1}$$

$$s4\_word0 = R_{3N/4}\beta_{0123}^{N/8-1} + R_{3N/4+1}\beta_{0123}^{N/8-2} + \dots + R_{7N/8-2}\beta_{0123} + R_{7N/8-1}$$

$$s1\_word2 = R_{N/8}\beta_{0123}^{N/8\ -\ 1} + R_{N/8+1}\beta_{0123}^{N/8\ -\ 2} + ... + R_{N/4-2}\beta_{0123} + R_{N/4-1}$$

$$s2\_word2 = R_{3N/8}\beta_{0123}^{N/8\ -\ 1} + R_{3N/8+1}\beta_{0123}^{N/8\ -\ 2} + ... + R_{N/2-2}\beta_{0123} + R_{N/2-1}$$

$$s3\_word2 = R_{5N/8}\beta_{0123}^{N/8\ -\ 1} + R_{5N/8+1}\beta_{0123}^{N/8\ -\ 2} + ...R_{3N/4-2}\beta_{0123} + R_{3N/4-1}$$

$$s4\_word2 = R_{7N/8}\beta_{0123}^{N/8\ -\ 1} + R_{7N/8+1}\beta_{0123}^{N/8\ -\ 2} + ...R_{N-2}\beta_{0123} + R_{N-1}$$

5.8.1 First Recombination Step Similar to the Case of T = 8

$$Sword0 = s1\_word0\,\beta_{0123}^{7N/8} + s2\_word0\beta_{0123}^{5N/8} + s3\_word0\beta_{0123}^{3N/8} + s4\_word0\beta_{0123}^{N/8}$$

$$Sword2 = s1\_word2\,\beta_{0123}^{3N/4} + s2\_word2\beta_{0123}^{N/2} + s3\_word2\beta_{0123}^{N/4} + s4\_word2$$

5.8.2 Second recombination step for T = 4

$$\mathrm{S}_{0123} = \mathrm{Sword0} + \mathrm{Sword2}$$

This second step of recombination gives us the first four syndromes. A similar set of equations can be written for the next four syndromes and is shown below. The two steps of recombination are also shown. This gives us the next four syndromes. The loop that produces the partial results iterates for N/8 – 1 times, which is half the iteration count for the case of T = 8. The results for the case of T = 4 are produced in half the time that is required for the case of T = 8.

$$s1\_word1 = R_0\beta_{4567}^{N/8\ -\ 1} + R_1\beta_{4567}^{N/8\ -\ 2} ....R_{N/8-2}\beta_{4567} + R_{N/8-1}$$

$$s2\_word1 = R_{N/4}\beta_{4567}^{N/8\ -\ 1} + R_{N/4+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{3N/8-2}\beta_{4567} + R_{3N/8-1}$$

$$s3\_word1 = R_{N/2}\beta_{4567}^{N/8\ -\ 1} + R_{N/2+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{5N/8-2}\beta_{4567} + R_{5N/8-1}$$

$$s4\_word1 = R_{3N/4}\beta_{4567}^{N/8\ -\ 1} + R_{3N/4+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{7N/8-2}\beta_{4567} + R_{7N/8-1}$$

$$s1\_word3 = R_{N/8}\beta_{4567}^{N/8\ -\ 1} + R_{N/8+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{N/4-2}\beta_{4567} + R_{N/4-1}$$

$$s2\_word3 = R_{3N/8}\beta_{4567}^{N/8\ -\ 1} + R_{3N/8+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{N/2-2}\beta_{4567} + R_{N/2-1}$$

$$s3\_word3 = R_{5N/8}\beta_{4567}^{N/8\ -\ 1} + R_{5N/8+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{3N/4-1}$$

$$s4\_word3 = R_{7N/8}\beta_{4567}^{N/8\ -\ 1} + R_{7N/8+1}\beta_{4567}^{N/8\ -\ 2} + ... + R_{N/4-2}\beta_{4567} + R_{N/4-1}$$

These 16 words that contain the partial results are re-combined as shown below. Unlike the T = 8 case there are two stages of re-combination for T = 4. The re-combination rules for the case of T = 4 are specified below:

$$Sword1 = s1\_word1\beta_{4567}^{7N/8} + s2\_word1\beta_{4567}^{5N/8} + s3\_word1\beta_{4567}^{3N/8} + s4\_word1\beta_{4567}^{N/8}$$

$$Sword3 = s1\_word3\,\beta_{4567}^{3N/4} + s2\_word3\beta_{4567}^{N/2} + s3\_word3\beta_{4567}^{N/4} + s4\_word3$$

$$\mathrm{S}_{4567} = \mathrm{Sword1} + \mathrm{Sword3}$$

In this way the same computational elements are used for computing syndromes for the case of T = 8 and T = 4.

5.9 Case of Odd N

The same algorithm can be used for computing syndromes for the case of odd N. In this case a certain number of zeroes are appended to the received code word R to make an augmented code word whose length is a multiple of 4 if T = 8 or a multiple of 8 if T = 4. However the section that computes the partial results using Horner’s rule uses the length of the augmented code word to compute N/4 –1 or N/8 –1 and hence has a slight overhead. None the less the same computational elements are used. For the case of a (255, 239, 8) code the first syndrome may be written as follows:

$$S_0 = r_0\beta_0^{254} + r_1\beta_0^{253} + ... + r_{254}$$

This can also be computed as an augmented code word of length 256 for the case of T = 8, by appending the original code word with one zero. In this case the first syndrome may be written as:

$$S_0 = 0.\beta_0^{255} + r_0\beta_0^{254} + r_1\beta_0^{253} + ... + r_{254}$$

The implementation thus keeps zeroing out the first byte, to prepare this augmented code word.

5.10 C Code Implementation

The C6400 DSP is a good compiler target and allows for obtaining the highest levels of performance directly from C code. As shown earlier the Galois field multiplier can be accessed by the _gmpy4 instruction. The syndrome accumulate requires 2T syndromes to be computed by computing the syndrome polynomial of size N for each. Hence the total number of galois field multiplies required is N * 2T. For the case of 204, 188, 8 code, the syndrome accumulate requires 204*16 = 3264 Galois field multiplies. The C64x can perform eight Galois field multiplies in a given cycle allowing for the core syndrome computation loop to be performed in about 408 cycles. The C6400 DSP allows the users to obtain this performance directly from C code. The piped loop kernel that performs this operation is a 8 cycle loop and for the case of T = 8 performs computations for all the 16 syndromes in parallel. The piped loop kernel performs 8 Galois field multiplies every cycle. The complete C code is provided below to enable a clear understanding of the various operations that are being performed in the core loop that computes the partial syndromes and the recombination loop that performs the final computation of all the syndromes. The resulting piped loop kernel is also shown below. The roots, various powers {N/4, N/2, 3N/4] powers are shown below. The 5N/4 th power table is not used for T = 8 and hence is set to 1s. For the case of T = 8 the number of iterations of the accumulate loop is N>> 2 and for the case of T = 4 the number of iterations is N>>3.

beta[ ]={beta0,beta1,beta2,beta3,beta4,beta5,beta6,......,beta15};
beta_RS_N_4={beta0^N/4,beta1^N/4,.......................beta15^N/4};
beta_RS_N_2={beta0^N/2,beta1^N/2,.......................beta15^N/2};
beta_RS_3N_4={beta0^3N/4,beta1^3N/4,beta2^3N/4,..........beta15^3N/4};
beta_RS_5N_4={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

These tables are shown for the case of the default generator polynomial, $G(X) = 1 + X^2 + X^3 + X^4 + X^8$ and are shown below:

beta[ ]   = { 1,   2,   4,   8,   16,   32,   64, 128,   29,   58, 116, 232, 205, 135, 19, 38};
beta_RS_N_4[ ] = {   1, 10, 68, 146, 221,   1,   10,   68, 146, 221,   1,   10,   68,   146, 221, 1};
beta_RS_N_2[ ] = {   1, 68, 221, 10, 146,   1,   68,   221, 10, 146,   1,   68,   221,   10, 146, 1};
beta_RS_3N–4[ ]= {   1, 146, 10, 221, 68,   1, 146,   10, 221, 68,   1,   146,   10,   221, 68,   1};

For the case of T = 4 each array is of length 16 but the data is massaged to get the right powers.

beta = {beta0123, beta4567, beta0123, beta4567};

beta_RS_N_4   =   beta0123^3N/8,beta4567^(3N/8),beta0123^(N/4),beta4567^(N/4)}
beta_RS_N_2 =   beta0123^5N/8,beta4567^5N/8,beta0123^N/2,beta4567^N/2}
beta_RS_3N_4 =   beta0123^7N/8,beta4567^7N/8,beta0123^3N/4,beta4567^3N/4}
beta_RS_5N_4 =   {beta0123^N/8,beta4567^N/8,beta0123^N/8,beta4567^N/8}

```
/* ======================================================================== */
/*  NAME: syn_acc –– Syndrome Accumulate for the REED–SOLOMON decoder       */
/*  USAGE                                                                   */
/*      This routine has the following C prototype:                         */
/*        int syn_acc       (  const unsigned char *restrict  byte_i,       */
/*                             unsigned char  *restrict s1,                 */
/*                             unsigned char  *restrict s2,                 */
/*                             const unsigned char *restrict  beta,         */
/*                             const unsigned char *restrict  beta_RS_N_4,  */
/*                             const unsigned char *restrict  beta_RS_N_2,  */
/*                             const unsigned char *restrict  beta_RS_3N_4, */
/*                             const unsigned char *restrict  beta_RS_5N_4, */
/*                             int RS_N, int RS_2T  );                      */
/*  The syn_accumulate routine  accepts the received codeword in byte_i     */
/*  array and computes  RS_2T syndromes and stores them in the output       */
/*  s1 array. s2 array is a scratch pad array of T characters. The beta     */
/*  arrays contain the various powers of the primitive element of the       */
/*  finite field.                                                           */
/*========================================================================*/
/*            Copyright (c) 2000 Texas Instruments, Incorporated.           */
/*                          All Rights Reserved.                            */
/* ======================================================================== */

int  syn_acc_c      ( const unsigned char *restrict byte_i,
                              unsigned char  *s1,
                              unsigned char  *s2,
                        const unsigned char *restrict beta,
                        const unsigned char *restrict beta_RS_N_4,
                        const unsigned char *restrict beta_RS_N_2,
                        const unsigned char *restrict beta_RS_3N_4,
                            const unsigned char *restrict beta_RS_5N_4,
                            int RS_N,
                            int RS_2T  )
    {
        double betadword0,  betadword1,  beta4dword0, beta4dword1;
        double beta2dword0, beta2dword1, beta3dword0, beta3dword1;
        double beta5dword0, beta5dword1;
        double *betaptr, *beta4ptr, *beta2ptr, *beta3ptr, *beta5ptr;

        unsigned int betaword0,  betaword1,  betaword2,  betaword3;
        unsigned int beta4word0, beta4word1, beta4word2, beta4word3;
        unsigned int beta2word0, beta2word1, beta2word2, beta2word3;
        unsigned int beta3word0, beta3word1, beta3word2, beta3word3;
        unsigned int beta5word0, beta5word1;

        int N, i, offset1, offset2, offset3;
        int offset4, offset5, offset6, offset7;

        const unsigned char *byte_iptr,    *byte_ioff1ptr, *byte_ioff2ptr;
        const unsigned char *byte_ioff3ptr,*byte_ioff4ptr, *byte_ioff5ptr;
        const unsigned char *byte_ioff6ptr, *byte_ioff7ptr;

        unsigned int   byteval,  byteval1, byteval2, byteval3;
        unsigned int   byteval4, byteval5, byteval6, byteval7;

        unsigned int   s0_startval, s1_startval, s2_startval, s3_startval;
        unsigned int   s4_startval, s5_startval, s6_startval, s7_startval;
        unsigned int   s1_w0,  s2_w0,  s3_w0,  s4_w0;
        unsigned int   s1_w1,  s2_w1,  s3_w1,  s4_w1;
        unsigned int   s1_w2,  s2_w2,  s3_w2,  s4_w2;
        unsigned int   s1_w3,  s2_w3,  s3_w3,  s4_w3;
        unsigned int   iters;

        int            r, t, st, status, modval, diff, val;
        int            zero = 0;
        int            ret0, ret1, ret2, ret3, ret10, ret23, ret;

        unsigned int   byte_i0,   byte_i1,    byte_i2, byte_i3;
        unsigned int   byte_i4,   byte_i5,    byte_i6, byte_i7;
        unsigned int   tmp1w0,    tmp1w1,     tmp1w2,  tmp1w3;
        unsigned int   tmp2w0,    tmp2w1,     tmp2w2,  tmp2w3;
        unsigned int   tmp3w0,    tmp3w1,     tmp3w2,  tmp3w3;
        unsigned int   tmp4w0,    tmp4w1,     tmp4w2,  tmp4w3;
        unsigned int   temp;
        unsigned int   temp1w0,   temp2w0,    temp3w0,  sum2w0, sum1w0, sumw0;
        unsigned int   temp1w1,   temp2w1,    temp3w1,  sum2w1, sum1w1, sumw1;
        unsigned int   temp1w2,   temp2w2,    temp3w2,  sum2w2, sum1w2, sumw2;
        unsigned int   temp1w3,   temp2w3,    temp3w3,  sum2w3, sum1w3, sumw3;
```

```
/*--------------------------------------------------------------------*/
/* Obtain an integer pointer to store off syndromes as word–wide quants*/
/* Load in the various powers of beta from the tables as dword wide    */
/* quantities and extract the low and high halves by the _lo and _hi   */
/* intrinsics.                                                         */
/* betaword0,.....betaword3:  Roots as described in Assumptions        */
/* beta4word0,....beta4word3: various powers of beta^N/4               */
/* beta2word0,....beta2word3: various powers of beta^N/2               */
/* beta3word0,.../beta3word3: various powers of beta^3N/4              */
/*--------------------------------------------------------------------*/

unsigned int *s1ptr = (unsigned int *) (s1);
betaptr  = (double *) (beta);
beta4ptr = (double *) (beta_RS_N_4);
beta2ptr = (double *) (beta_RS_N_2);
beta3ptr = (double *) (beta_RS_3N_4);
beta5ptr = (double *) (beta_RS_5N_4);

betadword0  = betaptr[0];         betadword1  = betaptr[1];
beta4dword0 = beta4ptr[0];        beta4dword1 = beta4ptr[1];
beta2dword0 = beta2ptr[0];        beta2dword1 = beta2ptr[1];
beta3dword0 = beta3ptr[0];        beta3dword1 = beta3ptr[1];
beta5dword0 = beta5ptr[0];

betaword0  = _lo(betadword0);    betaword1  = _hi(betadword0);
betaword2  = _lo(betadword1);    betaword3  = _hi(betadword1);
beta4word0  = _lo(beta4dword0);  beta4word1  = _hi(beta4dword0);
beta4word2  = _lo(beta4dword1);  beta4word3  = _hi(beta4dword1);
beta2word0  = _lo(beta2dword0);  beta2word1  = _hi(beta2dword0);
beta2word2  = _lo(beta2dword1);  beta2word3  = _hi(beta2dword1);
beta3word0  = _lo(beta3dword0);  beta3word1  = _hi(beta3dword0);
beta3word2  = _lo(beta3dword1);  beta3word3  = _hi(beta3dword1);
beta5word0  = _lo(beta5dword0);  beta5word1  = _hi(beta5dword0);

/*-------------------------------------------------------------------*/
/* Adjust N to be closest multiple of 4 for T = 8 and closest multiple*/
/* of 8 for T = 4. if T = 4 status = 1 else status = 0. This is done  */
/* by adding 8 or 4 to N. This modified N is now anded with a mask    */
/* that is the 1’s compliment of 8 or 4. If the result of and is zero */
/* then the result is reset to be either 8 or 4 and indicates that N  */
/* is already a multiple of 8 or 4. In this case N does not change.   */
/* The result of the AND represents the remainder obtained by dividing*/
/* the original N by 8 or 4. The difference between 8/4 and the       */
/* remainder is the number of zeroes that needs to be inserted.       */
/*-------------------------------------------------------------------*/

N = RS_N;
r = 4;
t = RS_2T >> 1;
status = (t < 5) ? 1:0;
if (status) r = 8;
temp = N + r;
val  = r – 1;
```

```
modval = temp & val;
if (!modval) modval = r;
diff = r – modval;
N    = N + diff;
/*--------------------------------------------------------------------*/
/* For the case of T = 8 set pointers to perform a radix4 Fourier     */
/* transform. For the case of T = 4 set pointers to perform a radix 8 */
/* transform. In order to reduce conditional code 8 pointers are      */
/* computed speculatively. If diff is non–zero these pointers are     */
/* subtracted from diff to offset them by the appropriate amount      */
/* These pointers are byte_iptr......byte_ioff7ptr                    */
/*--------------------------------------------------------------------*/

 offset1 = N >> 3;                    offset2 = N >> 2;  offset4 = N >> 1;
 offset3 = offset1 + offset2;         offset5 = offset1 + offset4;
 offset6 = offset2 + offset4;         offset7 = offset1 + offset6;
 offset1 –= diff;                     offset2 –= diff;
 offset3 –= diff;                     offset4 –= diff;
 offset5 –= diff;                     offset6 –= diff;
 offset7 –= diff;

 byte_iptr     = byte_i;              byte_ioff1ptr = byte_iptr + offset1;

 byte_ioff2ptr = byte_iptr + offset2;   byte_ioff3ptr = byte_iptr + offset3;
 byte_ioff4ptr = byte_iptr + offset4;   byte_ioff5ptr = byte_iptr + offset5;

 byte_ioff6ptr = byte_iptr + offset6;   byte_ioff7ptr = byte_iptr + offset7;
/*-------------------------------------------------------------------*/
/* Load data from these offsets. If diff is non–zero then zero out   */
/* first byte, otherwise start loading using byte_iptr. The values   */
/* at the various offsets are loaded speculatively into byteval,...  */
/* byteval7.                                                         */
/*-------------------------------------------------------------------*/

byteval  = 0;
if (!diff) byteval  = *byte_iptr++;
if (diff)  diff––;

byteval1 = *byte_ioff1ptr++;   byteval2 = *byte_ioff2ptr++;
byteval3 = *byte_ioff3ptr++;   byteval4 = *byte_ioff4ptr++;
byteval5 = *byte_ioff5ptr++;   byteval6 = *byte_ioff6ptr++;
byteval7 = *byte_ioff7ptr++;

/*-------------------------------------------------------------------*/
/* Adjust the loop iteration count to N/4 – 1 for T = 4 and N/8 – 1  */
/* for T = 8.                                                        */
/*-------------------------------------------------------------------*/

st = status;
iters  = N >> 2;
if (st) iters = N >> 3;
iters––;
```

```
/*-------------------------------------------------------------------*/
/* Prepare a packed quad with replicated bytes in each of the quad    */
/* using pack2 and packl4 instructions. Notice that if T = 8 then     */
/* the four words of si i={0,..3} are identical. For T = 4 only two   */
/* words of si are identical. If T = 8 s0_startval = s1–startval      */
/* If T = 4 then s0_strtval and s1_startval are different             */
/* Therefore if T = 8                                                 */
/* s0_startval = s1–startval = r0r0r0r0                               */
/* s2_startval  = s3_startval = rN/4 rN/4 rN/4 rN/4                    */
/* s4_startval = s5_startval = rN/2 rN/2 rN/2 rN/2                    */
/* s6–startval = s7_startval = r3N/4 r3N/4 r3N/4 r3N/4                */
/* If T = 4                                                           */
/* s0_startval = r0r0r0r0 and s1–startval = rN/8rn/8rN/8rN/8          */
/* s2_startval = rN/4rN/4rN/4rN/4 s3_startval = r3N/8r3N/8r3N/8r3N/8  */
/* s4_startval = rN/2rN/2rN/2rN/2 s5_startval = r5N/8r5N/8r5N/8r5N/8  */
/* s6–startval = r3N/4r3N/4r3N/4r3N/4 s7_startval=r7N/8r7N/8r7N/8r7N/8*/
/*-------------------------------------------------------------------*/

temp = _pack2(byteval,byteval);
s1_startval = _packl4(temp, temp);
s0_startval = s1_startval;
temp = _pack2(byteval1,byteval1);
if (st) s0_startval = _packl4(temp,temp);

temp = _pack2(byteval2,byteval2);
s3_startval = _packl4(temp, temp);
s2_startval = s3_startval;
temp = _pack2(byteval3,byteval3);
if (st) s2_startval = _packl4(temp,temp);

temp = _pack2(byteval4,byteval4);
s5_startval = _packl4(temp, temp);
s4_startval = s5_startval;
temp = _pack2(byteval5,byteval5);
if (st) s4_startval = _packl4(temp,temp);

temp = _pack2(byteval6,byteval6);
s7_startval = _packl4(temp, temp);
s6_startval = s7_startval;
temp = _pack2(byteval7,byteval7);
if (st) s6_startval = _packl4(temp,temp);

s1_w0 = s1_w1 = s1_startval;       s2_w0 = s2_w1 = s3_startval;
s3_w0 = s3_w1 = s5_startval;       s4_w0 = s4_w1 = s7_startval;

s1_w2 = s1_w3 = s0_startval;       s2_w2 = s2_w3 = s2_startval;
s3_w2 = s3_w3 = s4_startval;       s4_w2 = s4_w3 = s6_startval;

/*------------------------------------------------------------------*/
/* Compute the 4 words of each s1,s2,s3 and s4 for T = 8 or T = 4*/
/* Perform the Horner’s rule expansion using 16 partial words    */
/*------------------------------------------------------------------*/
```

```
for ( i = 0; i < iters; i++)
   {
       /*------------------------------------------------------------*/
       /* Load bytes at locations 0, N/8, N/4,....7N/8.  Note that loads*/
       /* to N/8, 3N/8, 5N/8 and 7N/8 are needed only if T = 4.  The   */
       /* first byte should be delayed till an appropriate number of   */
       /* zeroes has been inserted for the case of odd N.              */
       /*------------------------------------------------------------*/
       if (!diff) byteval =  *byte_iptr++;
       if (diff)  diff--;
       byteval2 = *byte_ioff2ptr++;
       byteval4 = *byte_ioff4ptr++;
       byteval6 = *byte_ioff6ptr++;
       if (st) byteval1 = *byte_ioff1ptr++;
       if (st) byteval3 = *byte_ioff3ptr++;
       if (st) byteval5 = *byte_ioff5ptr++;
       if (st) byteval7 = *byte_ioff7ptr++;

       temp    = _pack2(byteval, byteval);
       byte_i1 =  byte_i0 = _packl4(temp,temp);
       temp    = _pack2(byteval1,byteval1);
       if (st) byte_i1 = _packl4(temp,temp);

       temp    = _pack2(byteval2, byteval2);
       byte_i3 =  byte_i2 = _packl4(temp,temp);
       temp    = _pack2(byteval3,byteval3);
       if (st) byte_i3 = _packl4(temp,temp);

       temp    = _pack2(byteval4, byteval4);
       byte_i5 =  byte_i4 = _packl4(temp,temp);
       temp    = _pack2(byteval5,byteval5);
       if (st) byte_i5 = _packl4(temp,temp);

       temp    = _pack2(byteval6, byteval6);
       byte_i7 = byte_i6 = _packl4(temp,temp);
       temp    = _pack2(byteval7,byteval7);
       if (st) byte_i7 = _packl4(temp,temp);

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

/* $s1\_word0 = R_0\beta_{0123}^{N/4-1} + R_1\beta_{0123}^{N/4-2} + \ldots\ldots + R_{N/4-2}\beta_{0123}^{1} + R_{N/4-1}$ */

/* $s2\_word0 = R_{N/4}\beta_{0123}^{N/4-1} + R_{N/4+1}\beta_{0123}^{N/4-2} + \ldots\ldots + R_{N/2-2}\beta_{0123} + R_{N/2-1}$ */

/* $s3\_word0 = R_{N/2}\beta_{0123}^{N/4-1} + R_{N/2+1}\beta_{0123}^{N/4-2} + \ldots\ldots + R_{3N/4-2}\beta_{0123} + R_{N/4-1}$ */

/* $s4\_word0 = R_{3N/4}\beta_{0123}^{N/4-1} + R_{3N/4+1}\beta_{0123}^{N/4-2} + \ldots\ldots + R_{N-2}\beta_{0123} + R_{N-1}$ */

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

       tmp1w0     = _gmpy4(s1_w0, betaword0);
       s1_w0      = tmp1w0 ^ byte_i0;
       tmp2w0     = _gmpy4(s2_w0, betaword0);
       s2_w0      = tmp2w0 ^ byte_i2;
       tmp3w0     = _gmpy4(s3_w0, betaword0);
```

```
            s3_w0      = tmp3w0 ^ byte_i4;
            tmp4w0     = _gmpy4(s4_w0, betaword0);
            s4_w0      = tmp4w0 ^ byte_i6;
```

```
/*---------------------------------------------------------------*/
/* $s1\_word1 = R_0\beta_{4567}^{N/4-1} + R_1\beta_{4567}^{N/4-2} + \ldots\ldots\ldots. + R_{N/4-2}\beta_{4567}^{1} + R_{N/4-1}$ */
/* $s2\_word1 = R_{N/4}\beta_{4567}^{N/4-1} + R_{N/4-1}\beta_{4567}^{N/4-2} + \ldots\ldots. + R_{N/2-2}\beta_{4567}^{1} + R_{N/2-1}$ */
/* $s3\_word1 = R_{N/2}\beta_{4567}^{N/4-1} + R_{N/2+1}\beta_{4567}^{N/4-2} + \ldots.. + R_{3N/4-2}\beta_{4567}^{1} + R_{3N/4-1}$ */
/* $s4\_word1 = R_{3N/4}\beta_{4567}^{N/4-1} + R_{3N/4+1}\beta_{4567}^{N/4-2} + \ldots.. + R_{N-2}\beta_{4567}^{1} + R_{N-1}$ */
/*---------------------------------------------------------------*/

tmp1w1     = _gmpy4(s1_w1, betaword1);
s1_w1      = tmp1w1 ^ byte_i0;
tmp2w1     = _gmpy4(s2_w1, betaword1);
s2_w1      = tmp2w1 ^ byte_i2;
tmp3w1     = _gmpy4(s3_w1, betaword1);
s3_w1      = tmp3w1 ^ byte_i4;
tmp4w1     = _gmpy4(s4_w1, betaword1);
s4_w1      = tmp4w1 ^ byte_i6;

/*---------------------------------------------------------------*/
/* $s1\_word2 = R_0\beta_{891011}^{N/4-1} + R_1\beta_{891011}^{N/4-2} + \ldots\ldots\ldots. + R_{N/4-2}\beta_{891011}^{1} + R_{N/4-1}$ */
/* $s2\_word2 = R_{N/4}\beta_{891011}^{N/4-1} + R_{N/4+1}\beta_{891011}^{N/4-2} + \ldots\ldots. + R_{N/2-2}\beta_{891011} + R_{N/2-1}$ */
/* $s3\_word2 = R_{N/2}\beta_{891011}^{N/4-1} + R_{N/2+1}\beta_{891011}^{N/4-2} + \ldots.. + R_{3N/4-2}\beta_{891011}^{1} + R_{3N/4-1}$ */
/* $s4\_word2 = R_{3N/4}\beta_{891011}^{N/4-1} + R_{3N/4+1}\beta_{891011}^{N/4-2} + \ldots.. + R_{N-2}\beta_{891011}^{1} + R_{N-1}$ */
/*---------------------------------------------------------------*/

tmp1w2     = _gmpy4(s1_w2, betaword2);
s1_w2      = tmp1w2 ^ byte_i1;
tmp2w2     = _gmpy4(s2_w2, betaword2);
s2_w2      = tmp2w2 ^ byte_i3;
tmp3w2     = _gmpy4(s3_w2, betaword2);
s3_w2      = tmp3w2 ^ byte_i5;
tmp4w2     = _gmpy4(s4_w2, betaword2);
s4_w2      = tmp4w2 ^ byte_i7;

/*---------------------------------------------------------------*/
/* $s1\_word3 = R_0\beta_{12131415}^{N/4-1} + R_1\beta_{12131415}^{N/4-2} + \ldots\ldots\ldots. + R_{N/4-2}\beta_{12131415}^{1} + R_{N/4-1}$ */
/* $s2\_word3 = R_{N/4}\beta_{12131415}^{N/4-1} + R_{N/4+1}\beta_{12131415}^{N/4-2} + \ldots\ldots. + R_{N/2-2}\beta_{12131415} + R_{N/2-1}$ */
/* $s3\_word3 = R_{N/2}\beta_{12131415}^{N/4-1} + R_{N/2+1}\beta_{12131415}^{N/4-2} + \ldots.. + R_{3N/4-2}\beta_{12131415}^{1} + R_{3N/4-1}$ */
/* $s4\_word3 = R_{3N/4}\beta_{12131415}^{N/4-1} + R_{3N/4+1}\beta_{12131415}^{N/4-2} + \ldots.. + R_{N-2}\beta_{12131415}^{1} + R_{N-1}$ */
/*---------------------------------------------------------------*/
```

```
    tmp1w3     = _gmpy4(s1_w3, betaword3);
    s1_w3      = tmp1w3 ^ byte_i1;
    tmp2w3     = _gmpy4(s2_w3, betaword3);
    s2_w3      = tmp2w3 ^ byte_i3;
    tmp3w3     = _gmpy4(s3_w3, betaword3);
    s3_w3      = tmp3w3 ^ byte_i5;
    tmp4w3     = _gmpy4(s4_w3, betaword3);
    s4_w3      = tmp4w3 ^ byte_i7;
}
/*------------------------------------------------------------------- */
/* Completely unroll and perform re-combination loop using other     */
/* powers of beta.                                                    */

/*  $s_{0123} = s1\_word0\ \beta^{3N/4}_{0123} + S2\_word0\ \beta^{N/2}_{0123} + s3\_word0\beta^{N/4}_{0123} + s4\_word0\beta^{N/4}_{0123}$   */

/*  $s_{4567} = s1\_word1\beta^{3N/4}_{4567} + s2\_word1\beta^{N/2}_{4567} + s3word1\beta^{N/4}_{4567} + s4\_word1$   */

/*  $s_{891011} = s1\_word2\beta^{3N/4}_{891011} + s2\_word2\beta^{N/2}_{891011} + s3word2\beta^{N/4}_{891011} + s4\_word2$   */

/*  $s_{12131415} = s1\_word3\beta^{3N/4}_{12131415} + s2\_word3\beta^{N/2}_{12131415} + s3word2\beta^{N/4}_{12131415} + s4\_word3$ */

/*------------------------------------------------------------------- */

temp1w0        = _gmpy4(s1_w0, beta3word0);
temp2w0        = _gmpy4(s2_w0, beta2word0);
temp3w0        = _gmpy4(s3_w0, beta4word0);
sum2w0         =  temp1w0 ^ temp2w0;
if (st)  s4_w0 = _gmpy4(s4_w0,beta5word0);

sum1w0         =  temp3w0 ^ s4_w0;
sumw0          =  sum1w0 ^  sum2w0;

temp1w1         = _gmpy4(s1_w1, beta3word1);
temp2w1         = _gmpy4(s2_w1, beta2word1);
temp3w1         = _gmpy4(s3_w1, beta4word1);
sum2w1          =  temp1w1 ^ temp2w1;
if (st)  s4_w1  = _gmpy4(s4_w1,beta5word1);

sum1w1          =  temp3w1 ^ s4_w1;
sumw1           =  sum1w1 ^  sum2w1;

temp1w2         = _gmpy4(s1_w2, beta3word2);
temp2w2         = _gmpy4(s2_w2, beta2word2);
temp3w2         = _gmpy4(s3_w2, beta4word2);
sum2w2          =  temp1w2 ^ temp2w2;
sum1w2          =  temp3w2 ^ s4_w2;
sumw2           =  sum1w2 ^  sum2w2;

temp1w3         = _gmpy4(s1_w3, beta3word3);
temp2w3         = _gmpy4(s2_w3, beta2word3);
temp3w3         = _gmpy4(s3_w3, beta4word3);
sum2w3          =  temp1w3 ^ temp2w3;
sum1w3          =  temp3w3 ^ s4_w3;
sumw3           =  sum1w3 ^  sum2w3;
```

```
    /*----------------------------------------------------------------*/
    /* Perform one more level of re–combination for T = 4             */
    /*----------------------------------------------------------------*/

    if (st)  sumw0 = sumw0 ^ sumw2;
    if (st)  sumw1 = sumw1 ^ sumw3;

    *s1ptr++   = sumw0;           *s1ptr++   = sumw1;

    /*----------------------------------------------------------------*/
    /* Check if syndromes are non–zero by using split compares. If T  */
    /* is 4 just check sumw0 and sumw1 for the eight syndromes. If T  */
    /* is 8 check all 16 syndromes. If result is non–zero return 1.   */
    /*----------------------------------------------------------------*/

    ret0  = _cmpgtu4(sumw0, zero);   ret1  = _cmpgtu4(sumw1, zero);
    ret10 = ret0 + ret1;
    if (!st) *s1ptr++   = sumw2;     if (!st) *s1ptr++   = sumw3;
    ret2  = _cmpgtu4(sumw2, zero);   ret3  = _cmpgtu4(sumw3, zero);
    ret23 = ret2 + ret3;
    ret = 0;
    if (!st) ret = ret10 + ret23;
    if (st)  ret = ret10;
    if (ret) ret = 1;
    return(ret);
}
/*----------------------------------------------------------------------------*/
/*   End of file syn_acc_i.c                                                  */
/*----------------------------------------------------------------------------*/
/*            Copyright (c) 1999 Texas Instruments, Incorporated.            */
/*                           All Rights Reserved.                             */
/* ======================================================================== */
```

This piped loop kernel iterates N/4 –1 or N/8 –1 times depending on whether T = 8 or T = 4. The piped loop kernel computes 64 Galois field multiplies in 8 cycles maximizing the Galois field multiplier bandwidth. The compiler flags used were ” –k –o2 –mwtx –mv6400”. The numbers listed are the corresponding line numbers of the C code. It can be seen that in every cycle both the GMPY4 instructions are being used to perform eight multiplies in parallel.

```
L2:    ; PIPED LOOP KERNEL

Cycle 1:
   [ A2]   PACK2   .L2     B27,B27,B9         ; |538|
|| [ A2]   PACK2   .S1     A7,A7,A21          ; |544|
||         GMPY4   .M1     A20,A24,A17        ; |553|
||         GMPY4   .M2X    B17,A26,B19        ; |578|
||         PACKL4  .L1     A8,A8,A5           ; |529|
||         PACK2   .S2     B30,B30,B16        ; |535|
|| [ A2]   LDBU    .D2T2   *B4++,B27          ; |518|
||         LDBU    .D1T1   *A29++,A8          ; |515|

Cycle 2:
   [ A2]   PACK2   .S1     A6,A6,A22          ; |532|
||         GMPY4   .M1     A4,A25,A4          ; |571|
||         GMPY4   .M2X    B18,A26,B0         ; |574|
|| [!A0]   PACK2   .S2     B22,B22,B21        ; |511|
||         PACKL4  .L2     B16,B16,B18        ; |535|
||         PACKL4  .L1     A19,A19,A20        ; |541|
||         LDBU    .D2T2   *B8++,B30          ; |514|
||         LDBU    .D1T1   *A27++,A16         ; |513|

Cycle 3:
           GMPY4   .M1     A17,A26,A31        ; |580|
||         MV      .L1     A5,A4              ; |530|
||         XOR     .D1     A20,A16,A3         ; |562|
|| [ A1]   BDEC    .S1     L2,A1              ;
||         GMPY4   .M2X    B20,A25,B31        ; |565|
||         XOR     .S2     B18,B1,B29         ; |560|
||         MV      .D2     B18,B17            ; |536|
|| [!A0]   PACKL4  .L2     B21,B21,B5         ; |511|

Cycle 4:
           GMPY4   .M1     A18,A24,A16        ; |551|
||         MV      .S1     A20,A18            ; |542|
|| [ A0]   SUB     .D1     A0,1,A0            ; |512|
||         MV      .L2     B5,B2              ;
||         MV      .S2     B5,B20             ;
||         XOR     .D2X    A5,B0,B16          ; |558|
|| [ A2]   PACKL4  .L1     A22,A22,A4         ; |532|
||         GMPY4   .M2     B29,B23,B1         ; |560|

Cycle 5:
           XOR     .L2     B2,B31,B25         ; |556|
||         XOR     .S2X    A5,B30,B26         ; |549|
||         XOR     .S1     A4,A8,A30          ; |567|
||         XOR     .D1     A4,A16,A23         ; |576|
|| [ A2]   PACKL4  .L1     A21,A21,A18        ; |544|
||         GMPY4   .M1X    A3,B23,A16         ; |562|
|| [!A0]   LDBU    .D2T2   *B28++,B22         ; |511|
||         GMPY4   .M2     B16,B23,B0         ; |558|
Cycle 6:
   [ A2]   PACK2   .S2     B7,B7,B24          ; |526|
|| [ A2]   PACKL4  .L2     B9,B9,B17          ; |538|
||         XOR     .L1     A18,A4,A4          ; |571|
||         XOR     .S1X    B2,A19,A5          ; |547|
||         GMPY4   .M2     B25,B23,B31        ; |556|
||         GMPY4   .M1     A30,A25,A8         ; |567|
|| [ A2]   LDBU    .D1T1   *A9++,A7           ; |519|
Cycle 7:
           XOR     .S2     B17,B1,B19         ; |569|
||         XOR     .D2     B17,B19,B17        ; |578|
|| [ A2]   PACKL4  .L2     B24,B24,B20        ; |526|
||         GMPY4   .M2X    B26,A24,B30        ; |549|
||         GMPY4   .M1     A5,A24,A19         ; |547|
||         PACK2   .L1     A8,A8,A19          ; |541|
||         PACK2   .S1     A16,A16,A8         ; |529|
|| [ A2]   LDBU    .D1T1   *A28++,A6          ; |517|

Cycle 8:
           XOR     .L1X    B18,A16,A18        ; |551|
||         XOR     .S1     A18,A31,A17        ; |580|
||         XOR     .D1     A20,A17,A20        ; |553|
||         XOR     .L2     B20,B0,B18         ; |574|
||         XOR     .S2     B20,B31,B20        ; |565|
|| [ A2]   LDBU    .D2T2   *B6++,B7           ; |517|
||         GMPY4   .M2X    B19,A25,B1         ; |569|
||         GMPY4   .M1     A23,A26,A16        ; |576|
```

5.10.1 Berlekamp Massey Algorithm

The Berleykamp Massey Algorithm solves the lambda polynomial equation where

$$S_k - \sum_{i-1}^{L} \lambda_i^{k-1} S_{k-i} = 0 \qquad \forall \quad k$$

The convolution of lambda and syndromes is zero. The C code to solve for lambda given a particular syndrome is given below:

```
/*============================================================================*/
/*                                                                            */
/* TEXAS INSTRUMENTS, INC.                                                    */
/*                                                                            */
/* NAME                                                                       */
/*      bk_massey                                                             */
/*                                                                            */
/* USAGE                                                                      */
/*     This routine is C-callable and can be called as:                      */
/*                                                                            */
/*     void bk_massey_cn(unsigned char * s, unsigned int * GF_inv, int T,    */
/*                 int * fail_code, int * lam_deg, unsigned char * lambda); */
/*                                                                            */
/*      s         = pointer to syndromes                                       */
/*      GF_inv    = pointer to inverse table, each entry 4 inverses            */
/*      T         = number of errors to correct 1 to 8                         */
/*      fail_code = type of non zero failure                                   */
/*      lam_deg   = pointer to lambda degree                                   */
/*      lambda    = ptr to lamda polynomial                                    */
/*                                                                             */
/*       (See the C compiler reference guide.)                                 */
/*                                                                             */
/* DESCRIPTION                                                                 */
/*     The Berleykamp – Massey function solves the error locator polynomial */
/*     equation, Lambda * S = 0, where * denotes convolution. Both             */
/*     Lambda and Syndrome are polynomials of order T and 2*T respectively. */
/*                                                                             */
/*============================================================================*/
#define  RS_2T (16)
#define  RS_T  (8)
void bk_massey_cn(unsigned char * s, unsigned int * GF_inv, int T,
                  int * fail_code, int * lam_deg, unsigned char * lambda )
{
   unsigned char Told[RS_2T];
   unsigned char Tnew[RS_2T];
   unsigned char syn_rev[RS_T];     /* time reversed syndrome input */
   unsigned char delta, q;
   int i,j,k,L,case0, case1;

   for (i = 0; i <= RS_T; i++)
   {
     lambda[i] = 0;
     syn_rev[i] = 0;
     Told[i] = 0;
   }

   lambda[0] = 1;
   L = 0;
   Told[1] = 1;
   k = 1;

   delta = s[0];
   syn_rev[1] = s[0];
   for (j = 0; j < 2*T; j++)
   {
     case0 = (k – 2*L) >> 31;
     case1 = delta & ~case0;
     if (case1) L = k – L;

     for (i = RS_T; i > 0; i––) Tnew[i] = Told[i–1];

     q = (unsigned char) (0xff & GF_inv[0xff & delta]);
     for (i = RS_T; i > 0; i––)
     if ( case1) Tnew[i] = GMPY(lambda[i–1],q);

     for (i = RS_T; i > 0; i––) lambda[i] ^= GMPY(delta,Told[i]);
```

```
      for (i = RS_T; i > 0; i––) Told[i] = Tnew[i];

      delta = s[k];
      for (i = RS_T; i > 0; i––)
      delta ^= GMPY(lambda[i], syn_rev[i]);

      for (i = RS_T; i > 0; i––) syn_rev[i] = syn_rev[i–1];
      syn_rev[1] = s[k];

      k++;
    }
    *lam_deg = L;
    *fail_code = 0;

  }
/*============================================================================*/
/*      Copyright (C) 1997–2000 Texas Instruments Incorporated.           */
/*                      All Rights Reserved                                 */
/*============================================================================*/
```

The functions GMPY refer to the generic Galois field multiply which in the generic code is a call to a function using lookup tables. The GF_inv is a table containing the inverses for the elements of the Galois field used. This can contain up to 256 entries. The optimized intrinsic code below shows how this code is optimized.

```
/*============================================================================*/
/*                                                                          */
/* TEXAS INSTRUMENTS, INC.                                                  */
/*                                                                          */
/* NAME                                                                     */
/*       bk_massey                                                          */
/*                                                                          */
/* USAGE                                                                    */
/*     This routine is C-callable and can be called as:                     */
/*                                                                          */
/*     void bk_massey_c(unsigned char * s, unsigned int * GF_inv, int T,    */
/*                 int * fail_code, int * lam_deg, unsigned char * lambda); */
/*                                                                          */
/*      s         = pointer to syndromes                                    */
/*      GF_inv    = pointer to inverse table, each entry 4 inverses         */
/*      T         = number of errors to correct 1 to 8                      */
/*      fail_code = type of non zero failure                                */
/*      lam_deg   = pointer to lambda degree                                */
/*      lambda    = ptr to lamda polynomial                                 */
/*                                                                          */
/*       (See the C compiler reference guide.)                              */
/*                                                                          */
/* DESCRIPTION                                                              */
/*     The berlekamp – massey function solves the error locator polynomial  */
/*     equation, Lambda * S = 0, where * denotes convolution. Both          */
/*     Lambda and Syndrome are polynomials of order T and 2*T respectively. */
/*                                                                          */
/* ASSUMPTIONS                                                              */
/*     The input data and coeeficients are stored on double word aligned    */
/*     boundaries. The table GF_inv must be aligned to a 1K byte boundary.  */
/*     Each entry in the table is an inverse in the Galois field. Four      */
/*     inverses, are packed in each int.                                    */
/*     The GF_inv table is centered on a circular buffer using B4 as the    */
/*     circular buffer pointer.                                             */
/*     This code assumes LITTLE_ENDIAN system. The code is interrupt        */
/*     tolerant. Interupts are disabled and reenabled at the start and end  */
/*     of the code. RS_T = 8. The function works for any  2 <= T <= 8       */
/*                                                                          */
/* TECHNIQUES                                                               */
/*     A special table GF_inv has 4 inverses packed into a word. This       */
/*     simplifies the code slightly. Also the inner loops of order RS_T are */
/*     completely unrolled and placed in registers. SHLMB is used to        */
/*     advance the polynomial by alpha. No error is detected in this code.  */
/*     The inner loops only iterate for T times, this is to increase speed. */
/*     If an incorrect lamda is found, chien search will identify it.       */
/*                                                                          */
/* BIBLIOGRAPHY                                                             */
/*     This algorithm is taken from the University of Washington, Dept. of  */
/*     electrical engineering, A. Matache                                   */
/*     http://drake.ee.washington.edu/~adina/rsc/slide/node8.html           */
/*==========================================================================*/
/*    Copyright (C) 1997–1999 Texas Instruments Incorporated.               */
/*                    All Rights Reserved                                   */
/*==========================================================================*/
void bk_massey_c(unsigned char * s, unsigned int * GF_inv,
                 int T, int * fail_code, int * lam_deg,
                 unsigned char * lambda)
{
    unsigned int  k, i, L;
    unsigned int  delta;
    int  case0, case1;
    unsigned int  q3210;
    unsigned int  lam8765, lam4321_, lam4321, lam0;
    unsigned int  Tn8765, Tn4321;
    unsigned int  Tnew8765, Tnew4321;
    unsigned int  T8765, T4321;
    unsigned int  dlam8765, dlam4321;
    unsigned int  d0, d1, d2;
    unsigned int  s1234, s5678;
    unsigned int  d8765, d4321;
    unsigned int  del3210, del1032, del2301, del0123;
    unsigned int  delta_1, delta_2;

        lam8765 = lam4321 = 0;
        T8765 = T4321 = 0;
```

```
L = 0;
k = 1;
s1234 = 0;
s5678 = (unsigned int) s[0];
delta = s5678;
delta = _packl4(delta, delta);
delta = _packl4(delta, delta);
for (i =0; i < 2*T; i++)
{
  case0 = k – 2*L;
  case0 = case0 >> 31;
  case1 = delta & ~case0;
  if(case1) L = k – L;
  k += 1;
  q3210 = GF_inv[0xff & delta];
  lam4321_ = lam4321;
  Tn4321 = _gmpy4(lam4321_, q3210);
  Tn8765 = _gmpy4(lam8765, q3210);
  Tnew8765 = _shlmb(Tn4321, Tn8765);
  Tnew4321 = _shlmb(q3210,  Tn4321);

  if (!case1) Tnew8765 = _shlmb(T4321, T8765);
  if (!case1) Tnew4321 = T4321 << 8;

  dlam4321 = _gmpy4(delta, T4321);
  lam4321 = dlam4321 ^ lam4321;
  dlam8765 = _gmpy4(delta, T8765);
  lam8765 = dlam8765 ^ lam8765;

  T8765 = Tnew8765;
  T4321 = Tnew4321;

  d0 = (unsigned int) *++s;
  d1 = _packl4(d0, d0);
  d2 = _packl4(d1, d1);

  d8765 = _gmpy4(s1234, lam8765);
  d4321 = _gmpy4(s5678, lam4321);
  del3210 = d8765 ^ d4321;
  del1032 = _packlh2(del3210, del3210);
  del2301 = _swap4(del3210);
  del3210 = del3210 ^ d2;
  del0123 = _swap4(del1032);
  delta_1 = del3210 ^ del1032;
  delta_2 = del0123 ^ del2301;
  delta = delta_2 ^ delta_1;
  s1234  = _shlmb(s5678, s1234);
  s5678  = (s5678 << 8) + d0;
}
lam0 = 1;
lambda[0] = lam0;
_memd8(&lambda[1]) = _itod(lam8765, lam4321);
```

```
        lam_deg[0] = L;
        fail_code[0] = 0;
}
/*============================================================================*/
/*    Copyright (C) 1997-1999 Texas Instruments Incorporated.                 */
/*                    All Rights Reserved                                     */
/*============================================================================*/
```

A section out of the optimized assembly code shows the inner loop used :

```
LOOP:
        PACKL4  .L2     B_d0,       B_d0,       B_d1       ;expand syndrome x4
||      GMPY4   .M2X    B_delta,    A_T4321,    B_dlam4321 ;make lambda change
||      LDW     .D2T1   *B_GF_inv[B_delta],     A_q3210    ;q=inv_table[delta]
||      SHR     .S1     A_case0,    31,         A_case0    ;make signed mask

        PACKL4  .L2     B_d1,       B_d1,       B_d2       ;expand s[k] x4
||      GMPY4   .M2X    B_delta,    A_T8765,    B_dlam8765 ;make lamda changes

        ANDN    .S2X    B_delta,    A_case0,    B_case1    ;case1=delta&&case0

  [ B_case1]SUB .L1     A_k,        A_L,        A_L        ;update degree
;-
        XOR     .S2     B_dlam4321, B_lam4321,  B_lam4321  ;lambda new [4..1]

        GMPY4   .M2X    A_s5678,    B_lam4321,  B_d4321    ;see if lam's==0?
||      XOR     .D2     B_dlam8765, B_lam8765,  B_lam8765  ;update lam[8..5]
||      GMPY4   .M1X    B_lam8765,  A_q3210,    A_Tn8765   ;generate gradient T

        GMPY4   .M2X    A_s1234,    B_lam8765,  B_d8765    ;see if lam's ==0
||      GMPY4   .M1     A_lam4321,  A_q3210,    A_Tn4321   ;generate gradient T

        NOP             2                                  ;GMPY pipeline delay
;-
        BDEC    .S2     LOOP,       B_i                    ;for(i=0;i<2T;i++){
||      ADD     .D1X    A_s5678_,   B_d0,       A_s5678    ;shift syn window 1
||      SHLMB   .S1     A_s5678,    A_s1234,    A_s1234    ;shift syn window 1
||      ADD     .L1     A_k,        1,          A_k        ;update count
;-
        XOR     .L2     B_d8765,    B_d4321,    B_del3210  ;check for solution
||      SHLMB   .L1     A_q3210,    A_Tn4321,   A_Tnew4321 ;update T(x)
||      SHLMB   .S1     A_Tn4321,   A_Tn8765,   A_Tnew8765 ;update T(x)
||      LDBU    .D1T2   *++A_s[1],  B_d0                   ;get next s[k]
||      ROTL    .M1X    B_lam4321,  0,          A_lam4321  ;live too long
;-
        SWAP4   .L2     B_del3210,  B_del2301              ;check for solution
||      PACKLH2 .S2     B_del3210,  B_del3210,  B_del1032  ;check for solution
||[!B_case1]SHL .S1     A_T4321,    8,          A_Tnew4321 ;shift x.T(x) poly
||[!B_case1]SHLMB.L1    A_T4321,    A_T8765,    A_Tnew8765 ;shift x.T(x) poly

        SWAP4   .L2     B_del1032,  B_del0123              ;check for solution
||      XOR     .S2     B_del3210,  B_d2,       B_del3210  ;check for solution
||      MV      .L1     A_Tnew4321, A_T4321                ;update T poly
||      SHL     .S1     A_s5678,      8,          A_s5678_    ;part of syn window
;-
        XOR     .S2     B_del0123,  B_del2301,  B_delta_2   ;check for solution
||      XOR     .D2     B_del3210,  B_del1032,  B_delta_1   ;check for solution

        XOR     .L2     B_delta_2,  B_delta_1,  B_delta     ;check for solution
||      MV      .S1     A_Tnew8765, A_T8765                 ;update new T poly
||      SUBAH   .D1     A_k,        A_L,        A_case0     ;k –2*L > 0
```

The optimized assembly produces a 15 cycle loop. It uses a packed lookup table each entry has the inverse duplicated 4 times in a word. The table is shown here for GF(2^8) field generator polynomial 0x1D.

```
unsigned int GF_inv[GF8+PAD] = {
     0x00000000, 0x01010101, 0x8e8e8e8e, 0xf4f4f4f4,
     0x47474747, 0xa7a7a7a7, 0x7a7a7a7a, 0xbabababa,
     0xadadadad, 0x9d9d9d9d, 0xdddddddd, 0x98989898,
     0x3d3d3d3d, 0xaaaaaaaa, 0x5d5d5d5d, 0x96969696,
     0xd8d8d8d8, 0x72727272, 0xc0c0c0c0, 0x58585858,
     0xe0e0e0e0, 0x3e3e3e3e, 0x4c4c4c4c, 0x66666666,
     0x90909090, 0xdededede, 0x55555555, 0x80808080,
     0xa0a0a0a0, 0x83838383, 0x4b4b4b4b, 0x2a2a2a2a,
     0x6c6c6c6c, 0xedededed, 0x39393939, 0x51515151,
     0x60606060, 0x56565656, 0x2c2c2c2c, 0x8a8a8a8a,
     0x70707070, 0xd0d0d0d0, 0x1f1f1f1f, 0x4a4a4a4a,
     0x26262626, 0x8b8b8b8b, 0x33333333, 0x6e6e6e6e,
     0x48484848, 0x89898989, 0x6f6f6f6f, 0x2e2e2e2e,
     0xa4a4a4a4, 0xc3c3c3c3, 0x40404040, 0x5e5e5e5e,
     0x50505050, 0x22222222, 0xcfcfcfcf, 0xa9a9a9a9,
     0xabababab, 0x0c0c0c0c, 0x15151515, 0xe1e1e1e1,
     0x36363636, 0x5f5f5f5f, 0xf8f8f8f8, 0xd5d5d5d5,
     0x92929292, 0x4e4e4e4e, 0xa6a6a6a6, 0x04040404,
     0x30303030, 0x88888888, 0x2b2b2b2b, 0x1e1e1e1e,
     0x16161616, 0x67676767, 0x45454545, 0x93939393,
     0x38383838, 0x23232323, 0x68686868, 0x8c8c8c8c,
     0x81818181, 0x1a1a1a1a, 0x25252525, 0x61616161,
     0x13131313, 0xc1c1c1c1, 0xcbcbcbcb, 0x63636363,
     0x97979797, 0x0e0e0e0e, 0x37373737, 0x41414141,
     0x24242424, 0x57575757, 0xcacacaca, 0x5b5b5b5b,
     0xb9b9b9b9, 0xc4c4c4c4, 0x17171717, 0x4d4d4d4d,
     0x52525252, 0x8d8d8d8d, 0xefefefef, 0xb3b3b3b3,
     0x20202020, 0xecececec, 0x2f2f2f2f, 0x32323232,
     0x28282828, 0xd1d1d1d1, 0x11111111, 0xd9d9d9d9,
     0xe9e9e9e9, 0xfbfbfbfb, 0xdadadada, 0x79797979,
     0xdbdbdbdb, 0x77777777, 0x06060606, 0xbbbbbbbb,
     0x84848484, 0xcdcdcdcd, 0xfefefefe, 0xfcfcfcfc,
     0x1b1b1b1b, 0x54545454, 0xa1a1a1a1, 0x1d1d1d1d,
     0x7c7c7c7c, 0xcccccccc, 0xe4e4e4e4, 0xb0b0b0b0,
     0x49494949, 0x31313131, 0x27272727, 0x2d2d2d2d,
     0x53535353, 0x69696969, 0x02020202, 0xf5f5f5f5,
     0x18181818, 0xdfdfdfdf, 0x44444444, 0x4f4f4f4f,
     0x9b9b9b9b, 0xbcbcbcbc, 0x0f0f0f0f, 0x5c5c5c5c,
     0x0b0b0b0b, 0xdcdcdcdc, 0xbdbdbdbd, 0x94949494,
     0xacacacac, 0x09090909, 0xc7c7c7c7, 0xa2a2a2a2,
     0x1c1c1c1c, 0x82828282, 0x9f9f9f9f, 0xc6c6c6c6,
     0x34343434, 0xc2c2c2c2, 0x46464646, 0x05050505,
     0xcececece, 0x3b3b3b3b, 0x0d0d0d0d, 0x3c3c3c3c,
     0x9c9c9c9c, 0x08080808, 0xbebebebe, 0xb7b7b7b7,
     0x87878787, 0xe5e5e5e5, 0xeeeeeeee, 0x6b6b6b6b,
     0xebebebeb, 0xf2f2f2f2, 0xbfbfbfbf, 0xafafafaf,
     0xc5c5c5c5, 0x64646464, 0x07070707, 0x7b7b7b7b,
     0x95959595, 0x9a9a9a9a, 0xaeaeaeae, 0xb6b6b6b6,
     0x12121212, 0x59595959, 0xa5a5a5a5, 0x35353535,
     0x65656565, 0xb8b8b8b8, 0xa3a3a3a3, 0x9e9e9e9e,
     0xd2d2d2d2, 0xf7f7f7f7, 0x62626262, 0x5a5a5a5a,
     0x85858585, 0x7d7d7d7d, 0xa8a8a8a8, 0x3a3a3a3a,
     0x29292929, 0x71717171, 0xc8c8c8c8, 0xf6f6f6f6,
     0xf9f9f9f9, 0x43434343, 0xd7d7d7d7, 0xd6d6d6d6,
     0x10101010, 0x73737373, 0x76767676, 0x78787878,
     0x99999999, 0x0a0a0a0a, 0x19191919, 0x91919191,
     0x14141414, 0x3f3f3f3f, 0xe6e6e6e6, 0xf0f0f0f0,
     0x86868686, 0xb1b1b1b1, 0xe2e2e2e2, 0xf1f1f1f1,
     0xfafafafa, 0x74747474, 0xf3f3f3f3, 0xb4b4b4b4,
     0x6d6d6d6d, 0x21212121, 0xb2b2b2b2, 0x6a6a6a6a,
     0xe3e3e3e3, 0xe7e7e7e7, 0xb5b5b5b5, 0xeaeaeaea,
     0x03030303, 0x8f8f8f8f, 0xd3d3d3d3, 0xc9c9c9c9,
     0x42424242, 0xd4d4d4d4, 0xe8e8e8e8, 0x75757575,
     0x7f7f7f7f, 0xffffffff, 0x7e7e7e7e, 0xfdfdfdfd};
```

The performance of the algorithm is 30*T + 6 where T is the number of errors to be corrected.

5.10.2 Chien Search Algorithm

The Chien search algorithm solves for the roots of the error locator polynomial by plugging each field element one after the other and checks to see if it is a valid solution. To gain a clear understanding, of the algorithm the C code for the algorithm is shown below:

```
void ch_srch    (   restrict unsigned char lambda[],
                    int lam_deg,
                    restrict unsigned char  zeros[],
                    restrict int *fail,
                    const restrict unsigned char alpha[],
                    restrict int state[],
                    int RS_T,
                    const restrict unsigned char exp_table2[],
                    const restrict unsigned char log_table[],
                    const restrict unsigned char div_inv_table[]
                )
{
    int i, j, ptr,result0;
    int *state0;
    int  t0;
    const unsigned char *alpha0 = alpha ;
    state0 = state;
    ptr = 0;

    /*----------------------------------------------------------------------*/
    /* Initialize the elements of the zeros array to be zero.              */
    /* In addition compute lam1*alpha, lam2*alpha^2,...lamK*alpha^K        */
    /* where lami: ith term of lambda polynomial                           */
    /*       alpha: primitive element of the Galois field                  */
    /*       K:     degree of the lambda polynomial lam_deg < = T          */
    /* and store these in state[0], state[1], ....                         */
    /* The * operator denotes Galois field multiplication and the + operator*/
    /* denotes Galois field addition.                                      */
    /*----------------------------------------------------------------------*/

    for (j = 0; j < RS_T; j++)
    {
        zeros[j] = 0;
        t0 = _gmpy4(alpha0[j]] , lambda[j+1]);
        state0[j] = t0;
    }

    /*----------------------------------------------------------------------*/
    /* Now for each of the 256, elements in a GF(256) field compute the     */
    /* polynomial and check if it is a zero. Multiply each state entry      */
    /* state[j] by the corresponding power of alpha^j for use in computing  */
    /* the next iteration of the loop. If a zeros is found, store out the   */
    /* index where the zero was found.                                      */
    /*----------------------------------------------------------------------*/

    for (i = 1; i <= 256; i++)
    {
        result0 = 1;

        for (j = 0; j < RS_T; j++)
        {
            result0 = (result0^state0[j]);
            t0      = _gmpy4(state0[j],alpha0[j]);
            state0[j] = t0;
        }
        if (!result0) zeros[ptr++]= i;
    }

    /*----------------------------------------------------------------------*/
    /* Use the index where the zero was found to find the value of the zero,*/
    /* by using the exponentiation table. If the number of zeroes found is  */
    /* not equal to the degree of the lambda polynomial set a fail code of  */
    /* 3 and set all zeroes to 1. If lam_deg is greater than RS_T, the      */
    /* maximum number of errors that can be corrected, set a fail code of   */
    /* 2 and set all zeros to 1.                                            */
    /*----------------------------------------------------------------------*/
```

```
    for (i = 0; i < RS_T; i++)
    {
         zeros[i] = exp_table2[zeros[i]];
    }
}
```

The C code clearly shows that the Chien search algorithm has two loops, the outer loop that iterates once over all field elements, and an inner loop that iterates over all T. Since the C6400 DSP can perform up-to 8 Galois field multiplies in parallel, it is possible to completely unroll the inner loop and search the 256 elements of the field at the rate of 1 cycle/field element. The principal idea behind the efficient implementation of the Chien search is to maximize the use of the vector Galois field multiplies. This is done by splitting the search of N elements into four parallel searches of N/4 elements and performing each of them in parallel. Since N is a power of 2 by definition, N is divisible by 4 as in the case of GF(256) for example. Thus the efficient implementation of the Chien search splits the finite field of 256 elements into four sub-fields F1, F2, F3, F4 as shown below:

F1	{1,…….64}
F2	{65,……128}
F3	{129……192}
F4	{193,……256}

Note that one consequence of splitting the finite field into four sub-fields is that the order in which the zeroes are found is different from the one obtained by doing a straight search of N elements. However this is not a problem because the relative order of the zeroes is not important for the Reed Solomon decoding process. Thus the i’th iteration of the Chien search loop evaluates the error locator polynomial with the i’th element from the fields named F1, F2, F3 and F4.

Let the error locator polynomial Lambda(X) be assumed to have γ errors. The error locator polynomial can then be defined as shown below. In the worst case the degree of the polynomial γ is the same as T the number of errors the code can correct.

$\sigma(x) = 1 + \sigma_1 X + \sigma_2 X^2 + \sigma_3 X^3 + ... + \sigma_T X^T$. In order to gain a clear understanding of the steps involved the first two iterations of the loop and the associated values of the four possible zeroes in each iteration are shown.

1st iteration:

$Zf0 = 1 + \sigma_1\alpha + \sigma_2\alpha^{2} + \cdots\cdots \sigma_T\alpha^{T}$

$Zf1 = 1 + \sigma_1\alpha^{64} + \sigma_2\alpha^{128} + \cdots\cdots \sigma_T\alpha^{64T}$

$Zf2 = 1 + \sigma_1\alpha^{128} + \sigma_2\alpha^{256} + \cdots \sigma_T\alpha^{128T}$

$Zf3 = 1 + \sigma_1\alpha^{192} + \sigma_2\alpha^{384} + \cdots \sigma_T\alpha^{192T}$

2nd iteration:

$Zf0 = 1 + \sigma_1\alpha^{2} + \sigma_2\alpha^{4} + \cdots\cdots \sigma_T\alpha^{2T}$

$Zf1 = 1 + \sigma_1\alpha^{65} + \sigma_2\alpha^{130} + \cdots\cdots \sigma_T\alpha^{65T}$

$Zf2 = 1 + \sigma_1\alpha^{129} + \sigma_2\alpha^{258} + \cdots\sigma_T\alpha^{129T}$

$Zf3 = 1 + \sigma_1\alpha^{193} + \sigma_2\alpha^{386} + \cdots\sigma_T\alpha^{193T}$

It can be seen that the individual terms of Zf are multiplied by $\{\alpha,\alpha^2,\cdots\alpha^T\}$ to update the individual terms and are then summed together. The following C code is intended to show these operations so that a clear understanding of the different operations may be gained. This version of C code has also been optimized to handle the cases of T = 1 and T = 4.

```
/* ======================================================================= */
/*  NAME                                                                   */
/*      ch_srch -- Chien Search for Reed Solomon Decoding                  */
/*  USAGE                                                                  */
/*                                                                         */
/*      This routine has the following C prototype:                        */
/*                                                                         */
/*  void ch_srch_c  (   const restrict unsigned char lambda[],             */
/*                      int lam_deg,                                       */
/*                      restrict unsigned char  zeros[],                   */
/*                      restrict int *fail,                                */
/*                      const restrict unsigned char alpha[],              */
/*                      restrict int state[],                              */
/*                      int RS_T,                                          */
/*                      const restrict unsigned char exp_table2[],         */
/*                      const restrict unsigned char log_table[],          */
/*                      const restrict unsigned char div_inv_table[]       */
/*     where:                                                              */
/*           lambda:   Error locator polynomial                            */
/*           lam_deg:  Degree of the error locator polynomial. Can have a  */
/*                     maximum value of T + 1.                             */
/*           zeros:    Output array where the roots of the error locator   */
/*                     polynomial is to be stored.                         */
/*           fail:     Pointer where fail/pass state of algorithm is stored */
/*           alpha:    Array which is organized as follows                 */
/*                     { alpha0, alpha1,......... alpha7                   */
/*                       alpha0^64, alpha1^64,.....alpha7^64,              */
/*                       alpha0^128, alpha1^128,....alpha7^128,            */
/*                       alpha0^192,................alpha7^192}            */
/*                     a total of 32 elements that contain the 0,64,128 and */
/*                     192th powers of the various roots of the generator  */
/*                     polynomial. In the case of Reed Solomon, this is the */
/*                     various powers of the primitive element alpha.Thus  */
/*                     for the case of 204,188,8 code alpha is given below  */
/*                     unsigned char alpha[] = {                           */
/*                     2,   4,  8,   16,  32,  64,  128, 29,               */
/*                     190, 46, 100, 32,  94,  169, 28,  116,              */
/*                     23,  8,  184, 64,  169, 58,  33,  205,              */
/*                     25,  92, 47,  128, 28,  33,  30,  19   };           */
/*                                                                         */
/*           state:    Temporary scratch pad array used by other versions   */
/*           of C code.                                                    */
/*          RS_T:     The maximum number of errors that the Reed Solomon   */
/*          code can correct.                                              */
/*          exp_table2: An array that contains the various powers of the   */
/*          primitive element alpha.                                       */
/*          log_table: An array that contains various logarithms of the    */
/*          primitive element alpha.                                       */
/*          div_inv_table: A table containing the various multiplicative   */
/*          inverses of each field element.                                */
/*                                                                         */
/*  The chien search routine accepts an error locator polynomial lambda    */
/*  of degree lam_deg and finds the zeroes of the polynomial and returns   */
/*  these in the zeroes array. If the number of zeroes is > the degree of  */
/*  polynomial or T, then a fail code is returned.                         */
/*  ASSUMPTIONS                                                            */
/*  No specific alignment requirements are expected.                       */
/*  This code is Endian Neutral.                                           */
/*  The scratch pad array state is 36 bytes and does not alias with any    */
/*  other memory.                                                          */
/*  exp_table2: contains 512 values for each of the cases of exp[loga +logb]*/
/*  log_table:  contains 256 log values for each of the field element.     */
/*  div_inv_table: contains 256 values for each of the 256 elements.       */
/*                                                                         */
/*  NOTES                                                                  */
/*  If T = 4 is being used the alpha table needs to be modified.           */
/*  If a different GF field is being used the tables need to be modified.  */
/* ----------------------------------------------------------------------- */
/*           Copyright (c) 1999 Texas Instruments, Incorporated.           */
/*                          All Rights Reserved.                           */
/* ======================================================================= */

void ch_srch_c(    unsigned char restrict *lambda,
                   int lam_deg,
                   unsigned char  restrict *zeros,
                   int restrict *fail,
                   const unsigned char restrict *alpha,
                   int restrict *state,
                   int RS_T,
                   const unsigned char restrict *exp_table2,
                   const unsigned char restrict *log_table,
                   const unsigned char restrict *div_inv_table )
{
  /*-----------------------------------------------------------------------*/
  /* Set pointers to load the various powers of alphaI where:              */
  /* alpha0: reads the zeroth power of alpha0,...alpha7                    */
  /* alpha1: reads the sixty fourth power of alpha0...alpha7               */
  /* alpha2: reads the one hundred and twenty eight power of alpha0..alpha7 */
  /* alpha3: reads the one hundred and ninety second power of alpha0..alpha7*/
  /*                                                                       */
  /* If the alpha array is dword aligned then all these individual pointers*/
  /* will also be dword aligned and hence they can be cast as double point- */
  /* ers to perform loads using double word wide loads. The resulting low   */
  /* and high halves are loaded using intrinsics.                          */
  /*-----------------------------------------------------------------------*/
```

```
const unsigned char *alpha0 = alpha;
const unsigned char *alpha1 = alpha + 8;
const unsigned char *alpha2 = alpha + 16;
const unsigned char *alpha3 = alpha + 24;

double *alpha0ptr = (double *) (alpha0);
double *alpha1ptr = (double *) (alpha1);
double *alpha2ptr = (double *) (alpha2);
double *alpha3ptr = (double *) (alpha3);

double alpha0_dword, alpha1_dword, alpha2_dword, alpha3_dword;

unsigned int alpha00, alpha01, alpha10, alpha11;
unsigned int alpha20, alpha21, alpha30, alpha31;
unsigned int temp0,   temp1,   temp10,  temp11;
unsigned int temp2,   temp3,   temp12,  temp13;
unsigned int temp4,   temp5,   temp14,  temp15;
unsigned int temp6,   temp7,   temp16,  temp17;

unsigned int aword00, aword01, aword02, aword03;
unsigned int aword0,  aword1,  aword2, aword3;
unsigned int aword4,  aword5,  aword6, aword7;

unsigned int alpha0123word0, alpha0123word1, alpha0123word2;
unsigned int alpha0123word4, alpha0123word5, alpha0123word6;
unsigned int alpha0123word7, alpha0123word3;

double      *lamptr = (double *) (lambda);
double       lam_dword0, lam_dword1;
unsigned int lam0123,   lam4567,   lam8xxx;
unsigned int lamword00, lamword01, lamword02, lamword03;
unsigned int lamword0,  lamword1,  lamword2,  lamword3;
unsigned int lamword4,  lamword5,  lamword6,  lamword7;

unsigned int state0123word0, state0123word1, state0123word2;
unsigned int state0123word4, state0123word5, state0123word6;
unsigned int state0123word3, state0123word7;

unsigned int constant0;
unsigned int unityconstant;

unsigned char *zerosptr1, *zerosptr2;
unsigned char *zerosptr1orig, *zerosptr2orig;
unsigned int   result0123word0, result0123word1, result0123word2;
unsigned int   result0123word4, result0123word5, result0123word6;
unsigned int   result0123word3, result0123word7;

unsigned int result0, result1,   result2, result3;
unsigned int ptr,     limit;
unsigned int  i,    elements1, elements2;
unsigned int lam01, zero_val;
```

```
alpha0_dword = alpha0ptr[0];
alpha1_dword = alpha1ptr[0];
alpha2_dword = alpha2ptr[0];
alpha3_dword = alpha3ptr[0];

/*------------------------------------------------------------------*/
/* alpha01:alpha00 = A7 A6 A5 A4 A3 A2 A1 A0                        */
/* alpha11:alpha10 = B7 B6 B5 B4 B3 B2 B1 B0                        */
/* alpha21:alpha20 = C7 C6 C5 C4 C3 C2 C1 C0                        */
/* alpha31:alpha30 = D7 D6 D5 D4 D3 D2 D1 D0                        */
/*                                                                  */
/* Since the inner loop is unrolled completely, all four powers of  */
/* each alpha need to be packed together for the inner most loop.   */
/* For example the first word must contain, the powers of alpha0 in */
/* the following order: alpha0,alpha0^64,alpha0^128,alpha0^192      */
/* This requires packing the first words as D0 C0 B0 A0             */
/* This is done using a set of packs, namely _pack2/_packh2 and     */
/* _packl4 /_packh4                                                 */
/* This results in the following:                                   */
/* alpha0123word0 = D0 C0 B0 A0                                     */
/* alpha0123word1 = D1 C1 B1 A1                                     */
/* alpha0123word2 = D2 C2 B2 A2                                     */
/* alpha0123word3 = D3 C3 B3 A3                                     */
/* alpha0123word4 = D4 C4 B4 A4                                     */
/* alpha0123word5 = D5 C5 B5 A5                                     */
/* alpha0123word6 = D6 C6 B6 A6                                     */
/* alpha0123word7 = D7 C7 B7 A7                                     */
/*------------------------------------------------------------------*/

alpha00 = _lo(alpha0_dword);
alpha01 = _hi(alpha0_dword);
alpha10 = _lo(alpha1_dword);
alpha11 = _hi(alpha1_dword);
alpha20 = _lo(alpha2_dword);
alpha21 = _hi(alpha2_dword);
alpha30 = _lo(alpha3_dword);
alpha31 = _hi(alpha3_dword);
temp0   = _pack2(alpha10, alpha00);
temp1   = _pack2(alpha30, alpha20);
temp10  = _packl4(temp1, temp0);
temp11  = _packh4(temp1, temp0);
temp2  = _packh2(alpha10, alpha00);
temp3  = _packh2(alpha30, alpha20);
temp12 = _packl4(temp3, temp2);
temp13 = _packh4(temp3, temp2);
temp4  = _pack2(alpha11, alpha01);
temp5  = _pack2(alpha31, alpha21);
temp14 = _packl4(temp5, temp4);
temp15 = _packh4(temp5, temp4);
temp6  = _packh2(alpha11, alpha01);
temp7  = _packh2(alpha31, alpha21);
temp16 = _packl4(temp7, temp6);
temp17 = _packh4(temp7, temp6);
```

```
alpha0123word0 = temp10;
alpha0123word1 = temp11;
alpha0123word2 = temp12;
alpha0123word3 = temp13;
alpha0123word4 = temp14;
alpha0123word5 = temp15;
alpha0123word6 = temp16;
alpha0123word7 = temp17;
/*-----------------------------------------------------------------*/
/* In addition each state needs to be updated by multiplying by the */
/* appropriate power of alpha. Hence since all four of them need to */
/* be updated the various powers of alpha are packed together.      */
/* In this case the variables contain the foll:                     */
/*                                                                  */
/* aword0 = A0 A0 A0 A0                                             */
/* aword1 = A1 A1 A1 A1 where A1 = A0 ^ 2                           */
/* aword2 = A2 A2 A2 A2 where A2 = A0 ^ 3                           */
/* aword3 = A3 A3 A3 A3 where A3 = A0 ^ 4                           */
/* aword4 = A4 A4 A4 A4 where A4 = A0 ^ 5                           */
/* ...                                                              */
/* aword7 = A7 A7 A7 A7 where A7 = A0 ^ 8                           */
/*-----------------------------------------------------------------*/
aword00  = _pack2(alpha00, alpha00);
aword0   = _packl4(aword00, aword00);
aword1   = _packh4(aword00, aword00);
aword01  = _packh2(alpha00, alpha00);
aword2   = _packl4(aword01, aword01);
aword3   = _packh4(aword01, aword01);
aword02  = _pack2(alpha01,  alpha01);
aword4   = _packl4(aword02, aword02);
aword5   = _packh4(aword02, aword02);
aword03  = _packh2(alpha01, alpha01);
aword6   = _packl4(aword03, aword03);
aword7   = _packh4(aword03, aword03);
/*----------------------------------------------------------------------*/
/* In addition each lambda needs to be packed or replicated in all of    */
/* the four bytes. Therefore lamwordj contains lamj lamj lamj lamj as    */
/* a packed quantity where 1<= j<=8                                      */
/*----------------------------------------------------------------------*/
lam_dword0 = lamptr[0];
lam_dword1 = lamptr[1];
lam0123    = _lo(lam_dword0);
lam4567    = _hi(lam_dword0);
lam8xxx    = _lo(lam_dword1);

lam0123    = _shrmb(lam4567,lam0123);
lam4567    = _shrmb(lam8xxx,lam4567);
lamword00  = _pack2(lam0123, lam0123);
lamword0   = _packl4(lamword00, lamword00);
lamword1   = _packh4(lamword00,lamword00);
lamword01  = _packh2(lam0123, lam0123);
lamword2   = _packl4(lamword01, lamword01);
lamword3   = _packh4(lamword01, lamword01);
```

```
    lamword02  = _pack2(lam4567, lam4567);
    lamword4   = _packl4(lamword02, lamword02);
    lamword5   = _packh4(lamword02, lamword02);
    lamword03  = _packh2(lam4567, lam4567);
    lamword6   = _packl4(lamword03, lamword03);
    lamword7   = _packh4(lamword03, lamword03);
    /*-----------------------------------------------------------------------*/
    /* Completely unroll the loop that does the initial state computation     */
    /* and perform the following:                                             */
    /*                                                                        */
    /* word0 = lam0*alp0   lam0*alp0^64  lam0*alp0^128  lam0*alp0^192         */
    /* word1 = lam1*alp0^2 lam1*alp0^128 lam1*alp0^256  lam0^alp0^384         */
    /* ...                                                                    */
    /* word7 = lam0^alp0^8 lam1*alp0^64^8 ...           lam0*alp0^192^8       */
    /*-----------------------------------------------------------------------*/
    state0123word0  = _gmpy4(alpha0123word0, lamword0);
    state0123word1  = _gmpy4(alpha0123word1, lamword1);
    state0123word2  = _gmpy4(alpha0123word2, lamword2);
    state0123word3  = _gmpy4(alpha0123word3, lamword3);
    state0123word4  = _gmpy4(alpha0123word4, lamword4);
    state0123word5  = _gmpy4(alpha0123word5, lamword5);
    state0123word6  = _gmpy4(alpha0123word6, lamword6);
    state0123word7  = _gmpy4(alpha0123word7, lamword7);
    /*-----------------------------------------------------------------------*/
    /* Two constants are needed in the loop namely FF to mask off lower byte  */
    /* and a packed set of 1's to do the computation of the four possible     */
    /* zeroes.                                                                */
    /*-----------------------------------------------------------------------*/
    constant0 = 0x000000FF;
    unityconstant = 0x01010101;
    /*-----------------------------------------------------------------------*/
    /* Twin pointers are used on the zeros aray to do away with the dep-      */
    /* endency on the pointer. If T == 8 then zerosptr2 = zeros + 7 else      */
    /* it is zeros + 3. Also the four state words 4,..7 are zeroed so that    */
    /* result0123word3 may percolate through as result0123word7 for the case  */
    /* of T = 4, permitting the same code to be used for T = 4 and 8.         */
    /*-----------------------------------------------------------------------*/
    zerosptr1 = zeros;
    zerosptr2 = zeros + 7;
    if (RS_T == 4) zerosptr2 = zeros + 3;
    if (RS_T == 4) state0123word4 = state0123word5 =
                   state0123word6 = state0123word7 = 0;
    /*-----------------------------------------------------------------------*/
    /* Save off original value of pointers to figure out the number of zeroes */
    /* found outside the loop.                                                */
    /*-----------------------------------------------------------------------*/
    zerosptr1orig = zerosptr1;
    zerosptr2orig = zerosptr2;

  /*-----------------------------------------------------------------------*/
  /* If T is equal to 1, then use the div_inv_table to find out the solution */
  /*-----------------------------------------------------------------------*/
    lam01 = (lam0123 &0x000000FF);
```

```
      if (RS_T == 1) zeros[0] = div_inv_table[lam01];
      if ( RS_T != 1)
      {

    /*----------------------------------------------------------------------*/
    /* If T is either 4 or 8, then sum up all the elemts in the state–words*/
    /* and update the state words and check for zeroes and store off the   */
    /* indices, if a zero has been found.                                  */

    /*----------------------------------------------------------------------*/
          for ( i = 1; i <=64; i++)
          {
               result0123word0 = unityconstant;
               result0123word0 = (result0123word0 ^ state0123word0);
               state0123word0  = _gmpy4(state0123word0, aword0);
               result0123word1 = (result0123word0 ^ state0123word1);
               state0123word1  = _gmpy4(state0123word1, aword1);
               result0123word2 = (result0123word1 ^ state0123word2);
               state0123word2  = _gmpy4(state0123word2, aword2);
               result0123word3 = (result0123word2 ^ state0123word3);
               state0123word3  = _gmpy4(state0123word3, aword3);
               result0123word4 = (result0123word3 ^ state0123word4);
               state0123word4  = _gmpy4(state0123word4, aword4);
               result0123word5 = (result0123word4 ^ state0123word5);
               state0123word5  = _gmpy4(state0123word5, aword5);
               result0123word6 = (result0123word5 ^ state0123word6);
               state0123word6  = _gmpy4(state0123word6, aword6);
               result0123word7 = (result0123word6 ^ state0123word7);
               state0123word7  = _gmpy4(state0123word7, aword7);
               result0 = result0123word7 & constant0;
               result1 = (result0123word7 >> 8) & constant0;
               result2 = (result0123word7 >> 16) & constant0;
               result3 = (result0123word7 >> 24);
               if (!result0)    *zerosptr1++ = i;
               if (!result1)    *zerosptr1++ = i + 64;
               if (!result2)    *zerosptr2–– = i + 128;
               if (!result3)    *zerosptr2–– = i + 192;
           }
       }
       /*-----------------------------------------------------------------------*/
       /* The nuber of zeroes found is the sum of elements1 and elements2.     */
       /* elements1 and 2 are in turn found out by deducting the incremented   */
       /* pointers from the original pointers. If T == 1 then the # of zeroes  */
       /* found ie. ptr is set to 1.                                           */
       /*-----------------------------------------------------------------------*/

       elements1 = zerosptr1 – zerosptr1orig;
       elements2 = zerosptr2orig – zerosptr2;
       ptr = elements1 + elements2;
       if (RS_T == 1) ptr = 1;
```

```
   /*-----------------------------------------------------------------*/
   /* If ptr is not equal to lam_degree then a fail code of 3 is set.     */
   /* If lam_degree is greater than T a fail code of 2 is set.          */
   /* In both cases the zeros array is filled with 1's. If ptr is the     */
   /* same as lam_deg find the actual zeroes from the indices of the      */
   /* zeroes through the use of the exponentiation tables.              */
   /*-----------------------------------------------------------------*/
  *fail = 0;
   limit = 1;
   if (ptr != lam_deg) *fail = 3;
   if (ptr != lam_deg)  limit = 0;
   if (lam_deg > RS_T) *fail = 2;
   if (lam_deg > RS_T)  limit = 0;
   if ( RS_T > 1)
   {
       for (i = 0; i < RS_T; i++)
       {
           zero_val = exp_table2[zeros[i]];
           if (!limit) zero_val = 1;
           zeros[i] = zero_val;
       }
   }
}
/* ======================================================================= */
/*  End of file:  ch_srch_i.c                                              */
/* ----------------------------------------------------------------------- */
/*           Copyright (c) 1999 Texas Instruments, Incorporated.          */
/*                          All Rights Reserved.                          */
/* ======================================================================= */
```

The piped loop kernel shown below can be obtained by using the assembly optimizer to write a serial assembly version of the loop, to obtain a 4 cycle loop in which 32 Galois field multiplies are performed at the rate of 8 multiplies/cycle.

```
L_1:                                                                ; Cycle 1
  [!A_res0]STB .D1T1 A_i0,            *A_zerosptr1++                ; zero
||      SHRU   .S2X  A_result0123w7,  16,             B_res23xx        ; res >> 16
||     AND    .S1   A_res123x,       A_constant0,     A_res1           ; low16
||     GMPY4  .M1   A_state0123w6,   A_aword6,        A_state0123w6    ; sw6*aw6
||     XOR    .L1   A_result0123w5,  A_state0123w6,   A_result0123w6   ; rw5^sw6
||     XOR    .D2   B_result0123w2,  B_state0123w3,   B_result0123w3   ; rw2^sw3
||     GMPY4  .M2   B_state0123w0,   B_aword0,        B_state0123w0    ; sw0*aw0
||     MV     .L2   B_unityconstant, B_result0123w0                    ; Init.res
L_2:                                                                ; Cycle 2
       ADD    .S1   A_i1,            1,             A_i1              ; i1++
||     ADD    .L1   A_i0,            1,             A_i0              ; i0++
||[!A_res1]STB .D1T1 A_i1,            *A_zerosptr1++                ; zero
||     AND    .D2   B_res23xx,       B_constant0,     B_res2           ; Low16
||     SHRU   .S2X  A_result0123w7,  24,              B_res3           ; res>>16
||     GMPY4  .M1   A_state0123w7,   A_aword7,        A_state0123w7    ; sw7*aw7
||     GMPY4  .M2   B_state0123w3,   B_aword3,        B_state0123w3    ; sw3*aw3
||     XOR    .L2   B_result0123w0,  B_state0123w0,   B_result0123w0   ; rw0^sw0
```

```
L_3:                                                                      ; Cycle 3
       ADD     .L2   B_i2,           1,               B_i2                ; i2++
||[!B_res2]STB  .D2T2 B_i2,          *B_zerosptr2––                       ; zero
||    BDEC    .S1   LOOP8,           A_iters                              ; Branch
||    XOR     .D1   A_result0123w6,  A_state0123w7,   A_result0123w7      ; rw6 ^ sw7
||    GMPY4   .M1   A_state0123w4,   A_aword4,        A_state0123w4       ; sw4 * aw4
||    XOR     .L1X  B_result0123w3,  A_state0123w4,   A_result0123w4      ; rw3 ^ sw4
||    GMPY4   .M2   B_state0123w1,   B_aword1,        B_state0123w1       ; sw1 * aw1
||    XOR     .S2   B_result0123w0,  B_state0123w1,   B_result0123w1      ; rw0 ^ sw1
L_4:                                                                      ; Cycle 4
      ADD     .S2   B_i3,             1,               B_i3               ; i3 ++
||[!B_res3]STB  .D2T2 B_i3,             *B_zerosptr2––                    ; zero
||    SHRU    .S1   A_result0123w7,   8,               A_res123x          ; res>>16
||    AND     .L1   A_result0123w7,   A_constant0,     A_res0             ; Low16
||    GMPY4   .M1   A_state0123w5,    A_aword5,        A_state0123w5      ; sw5 * aw5
||    XOR     .D1   A_result0123w4,   A_state0123w5,   A_result0123w5     ; rw4 ^ sw5
||    GMPY4   .M2   B_state0123w2,    B_aword2,        B_state0123w2      ; sw2 * aw2
||    XOR     .L2   B_result0123w1,   B_state0123w2,   B_result0123w2     ; rw1 ^ sw2
```

5.11 Forney Algorithm

This routine accepts inputs from the other three routines, namely syndrome, Chien search and Berleykamp Massey and computes the error magnitudes and performs the correction. The principal equation that needs to be evaluated is repeated once again in order to understand the various computational elements involved: $e_k = \frac{a^k \Omega(a^{-k})}{Lambda\ (a^{-k})}$. The Forney algorithm computes the first T values, of omega although in practice up-to 2T values can be computed using the syndrome. Sine at most T errors are to be corrected T values of Omega suffice. In addition the derivative of the lambda polynomial also needs to be computed, and both these polynomials need to be evaluated at each of the zeroes found using Chien search and plugged into the expression for e_k to solve for the error magnitudes at these k locations. With the knowledge of the k error magnitudes and the k error locations where 0 <= k <= T the errors can be corrected.

The definition of the omega polynomial is repeated here in order to understand how it is to be computed: $E(x).Lambda(x) = \Omega(x)(x^n - 1)$. In addition recall that E(x) is the Fourier transform of the error polynomial evaluated at the 2T roots, which are the syndromes $S_0, S_1, S_2, ..., S_{2T}$. In addition recall that because the field is finite, this corresponds to a circular convolution. The terms of Omega can be enumerated as shown below:

$$\Omega_0 = \lambda_0 \phi\ S_0, \Omega_1 = \lambda_1 \phi\ S_0 + \lambda_0 \phi\ S_{1,} ..., \Omega_T = \lambda_T \phi\ S_0 + \lambda_{T-1} \phi\ S_1 + ... + \lambda_0 \phi\ S_T$$

The next term to evaluate is the derivative of the lambda polynomial, which is merely the shifted version of the lambda polynomial zeroed out at the even locations because of the property of GF(2^m). The actual error magnitude evaluation can be viewed as the evaluation of the numerator and denominator. The numerator is the evaluation of the Omega polynomial at the k'th zero, and the denominator is the evaluation of the derivative of the Lambda polynomial obtained from Berleykamp-Massey at the kth zero. Each denominator term is also individually multiplied by the kth zero (α^{-k}) instead of multiplying the numerator by (α^k) . These concepts can be seen clearly by the following C code.

```
/* ========================================================================= */
/*  NAME                                                                     */
/*      Forney –– Forney Algorithm for Reed Solomon Decoder                  */
/*  USAGE                                                                    */
/*                                                                           */
/*      This routine has the following C prototype:                          */
/*                                                                           */
/*    void forney_cn(  const  unsigned char restrict   s[],                  */
/*                            unsigned char restrict   lambda[],             */
/*                                     int             lam_deg,              */
/*                            unsigned char restrict   zeros[],              */
/*                            unsigned char restrict   byte_i[],             */
/*                                     int             *fail,                */
/*                      const unsigned char *restrict  tables[],             */
/*                                     int             T,                    */
/*                                     int             RS_N,                 */
/*                            unsigned char restrict   scratch[] )           */
/*                                                                           */
/*   T: # of errors that code can correct                                    */
/*   s: array that contains 2T syndromes                                     */
/*   lambda: array of error locator polynomial of degree T + 1               */
/*           where lambda[0] = 1                                             */
/*   lam_deg:Degree of the lambda polynomial.                                */
/*   zeros:  Zeroes of the error locator polynomial found using Chien        */
/*           search.                                                         */
/*   byte_i: received code word of bytes                                     */
/*   fail:   pointer to store status of correction                           */
/*   tables: array of pointers that contain the foll: pointers               */
/*           tables[0] ––––> exp_table exponenentials of primitive element  */
/*           tables[1] ––––> log_table logarithms of the field elements     */
/*   For eg: exp_table[8] = 2^8 = 29   and inversely log[29] = 8             */
/*   for the case of (204,188,8) code for the default generator polynomial  */
/*   100011101 GF(256)                                                       */
/*   RS_N:   total number of bytes including parity bytes, 204 for the case */
/*           of (204,188,8) code.                                            */
/*   scratch: temporary scratch pad array of size 11 * T to fold temporary  */
/*            results.                                                       */
/*                                                                           */
/*   The forney routine accepts the syndromes array ”s” computed using the  */
/*   syndrome routine, ”lambda” error locator polynomial computed using the */
/*   ”Berley Kamp” or equivalent, ”zeros” array that contains the roots of  */
/*   the error locator polynomial, ”byte_i” the received code word and      */
/*   corrects the errors in the received codeword array.                    */

/*                                                                           */
/*  ASSUMPTIONS                                                              */
/*                                                                           */
/* a) s: array of syndromes of size 2T                                       */
/* b) lambda: error locator polynomial computed by Berley Kamp or equiv.     */
/* c) lam_deg: degree of error locator polynomial.                           */
/* d) zeros: zeroes of the error locator polynomial found by Chien search.  */
/* e) byte_i: received code word of size N                                   */
/* f) fail: location where fail status is stored                             */
/* g) None of these arrays overlap in memory.                              */
/* ----------------------------------------------------------------------- */
/*            Copyright (c) 2000 Texas Instruments, Incorporated.          */
/*                           All Rights Reserved.                          */
/* ======================================================================= */
void forney_cn(  const  unsigned char *restrict  s,
                        unsigned char *restrict  lambda,
                                 int             lam_deg,
                        unsigned char *restrict  zeros,
                        unsigned char *restrict  byte_i,
                                 int             *fail,
                  const unsigned char *restrict *restrict  tables,
                                 int             T,
                                 int             RS_N,
                        unsigned char *restrict  scratch )
{
    int i,j,k;
    /* omega[3T] + syn[4T] + den[T] + num[T] +poly[2T]*/
    unsigned char *omega = scratch;
    unsigned char *syn   = omega  +  3*T;
    unsigned char *numerator = syn + 4*T;
    unsigned char *denominator = numerator + T;
    unsigned char *poly  = denominator   +  T;
    unsigned char err_pos_i, err_mag_i, a, b;
    int  RS_T = T;
    int  RS_2T = 2*T;
    int  t0;

    const unsigned char *exp_table2 = tables[0];
    const unsigned char *div_inv_table = tables[1];
    const unsigned char *log_table = tables[2];

    /*---------------------------------------------------------------------*/
    /*   Create an array of size 4T and copy the 2T syndrome values from T  */
    /*   Compute the omega polynomial as the circular convolution of the    */
    /*   Lambda polynomial of Berlekamp–Massey and the syndrome, recursive– */
    /*   ly by Horner’s rule.                                               */
    /*---------------------------------------------------------------------*/
    for (i= 0; i< RS_T; i++) syn[i] = 0;
    for (i= RS_T; i < (RS_2T + RS_T); i++) syn[i] = s[i–RS_T];
    for (i= RS_2T + RS_T; i<(RS_2T + RS_2T); i++) syn[i] = 0;
    for (j = RS_2T – 1; j >= 0; j––)
    {
        omega[j]= 0;
        for (i = 0; i <= RS_T; i++)
        {
            t0 = _gmpy4(lambda[i],syn[RS_T+j–i]);
            omega[j]^= t0;
        }
     }
```

```
  /*------------------------------------------------------------------*/
  /* Now compute the derivative of the lambda poly and store the result */
  /* This is done by shifting the terms of the lambda polynomial by one */
  /* and by zeroing out the even terms.                                 */
  /*------------------------------------------------------------------*/
  for (i = RS_T; i< RS_2T; i++) poly[i] = 0;
  for (i = 0; i < RS_T;i++)
  {
    poly[i] = lambda[i];
    if (i%2 == 0) poly[i] = 0;
  }
 /*-------------------------------------------------------------------*/
 /* Now find the error magnitudes by evaluating the Omega polynomial    */
 /* and the derivative of the Lambda polynomial at the zeroes found     */
 /* using the Chien search routine. In addition instead of multiplying  */
 /* the numerator by alpha^k multiply the denominator by alpha^–k which */
 /* is contained in the zeroes array.                                   */
 /*-------------------------------------------------------------------*/
  for (i=0; i< RS_T; i++)
  {
    denominator[i] = 0;
    numerator[i] = 0;
  }

  for (j = RS_T – 1; j >= 0; j––)
  {
     for (i = 0; i< RS_T; i++)
     {
       t0  =  _gmpy4(zeros[i]] , log_table[denominator[i]);
       denominator[i] = t0;
       denominator[i]^= poly[1+j];
       t0  =  _gmpy4([zeros[i],numerator[i]);
       numerator[i] = t0;
       numerator[i]^= omega[j];
     }
   }
   for (i = 0; i < RS_T; i++)
   {
     t0 = _gmpy4(denominator[i], zeros[i]);
     denominator[i] = t0;
   }
   /*-----------------------------------------------------------------*/
   /* Use the div_inv table and divide the denominator and the zeroes  */
   /* array. Use the log_table to find k from alpha^k. Multiply the    */
   /* result of 1/denominator with the numerator to obtain the error   */
   /* magnitude. Deduct N – 1 from the error location to get the       */
   /* modified error location.  If fail code has not been set perform  */
   /* error correction.                                                */
   /*-----------------------------------------------------------------*/
   for (i = 0; i < RS_T; i++)
   {
       err_pos_i = 0;
       err_mag_i = 0;
```

```
          b = div_inv_table[denominator[i]];
          a = div_inv_table[zeros[i]];
          if (i < lam_deg)
          {
              t0 = _gmpy4(numerator[i], b);
              err_mag_i = t0;
              err_pos_i = log_table[a];

              if (!denominator[i])
              {
                  *fail = 4;
              }

              if (err_pos_i > RS_N – 1)
              {
                *fail = 5;
              }
          }
          k = RS_N–1 – err_pos_i; /* subtract the error from r(x) */
          if (*fail == 0)
          {
            byte_i[k] = byte_i[k] ^ err_mag_i;
          }
      }
}
/* ======================================================================== */
/*  End of file:  forney_c.c                                                */
/* ------------------------------------------------------------------------ */
/*            Copyright (c) 2000 Texas Instruments, Incorporated.           */
/*                           All Rights Reserved.                           */
/* ========================================================================*/
```

This code can be optimized to use the capabilities of the C6400 by optimizing each of the computational steps of the Forney algorithm. These steps are shown below:

1. Omega polynomial calculation:

   Each term of the lambda polynomial (byte) is read in and replicated in all four bytes to form a packed word. This packed word is then multiplied with a certain number of syndromes as shown below. For example λ_0 multiples the first T syndromes, while λ_1 multiplies the first T-1 syndromes, and so on till λ_7 multiplies only the first syndrome S_0. These partial terms are written out explicitly. Each column indicates how many multiplies need to be performed for a given λ_i . Consider the case of T = 8 errors:

   $\lambda_0 \otimes S_0, \lambda_0 \otimes S_1, \lambda_0 \otimes S_2, ..., \lambda_0 \otimes S_7,$

   $\lambda_1 \otimes S_0, \lambda_0 \otimes S_1, \lambda_0 \otimes S_2, ..., \lambda_0 \otimes S_6,$

   . . .

   $\lambda_7 \otimes S_0$

2. The zeroes found from Chien search are read into registers and the numerator and denominator computations are done using registers instead of reading and writing to memory. This results in 2 words for the numerator and 2 words for the denominator.

3. The loop that detects the error magnitudes reads the denominator and numerator by shifting the calculated denominator words one byte at a time, and switch between two words after four bytes of the first word have been consumed.

```
/* ======================================================================= */
/*  NAME                                                                   */
/*                                                                         */
/*      forney –– Forney Algorithm for Reed Solomon Decoder                */
/*                                                                         */
/*                                                                         */
/*  USAGE                                                                  */
/*                                                                         */
/*      This routine has the following C prototype:                        */
/*                                                                         */
/*    void forney_cn(  const  unsigned char restrict   s[],                */
/*                            unsigned char restrict   lambda[],           */
/*                                     int             lam_deg,            */
/*                            unsigned char restrict   zeros[],            */
/*                            unsigned char restrict   byte_i[],           */
/*                                     int             *fail,              */
/*                      const unsigned char *restrict  tables[],           */
/*                                     int             T,                  */
/*                                     int             RS_N,               */
/*                            unsigned char restrict   scratch[] )         */
/*                                                                         */
/*   T: # of errors that code can correct                                  */
/*   s: array that contains 2T syndromes                                   */
/*   lambda: array of error locator polynomial of degree T + 1             */
/*           where lambda[0] = 1                                           */
/*   lam_deg:Degree of the lambda polynomial.                              */
/*   zeros:  Zeroes of the error locator polynomial found using Chien      */
/*           search.                                                       */
/*   byte_i: received code word of bytes                                   */
/*   fail:   pointer to store status of correction                         */
/*   tables: array of pointers that contain the foll: pointers             */
/*           tables[0] ––––> exp_table exponenentials of primitive element */
/*           tables[1] ––––> log_table logarithms of the field elements    */
/*   For eg: exp_table[8] = 2^8 = 29   and inversely log[29] = 8           */
/*   for the case of (204,188,8) code for the default generator polynomial */
/*   100011101 GF(256)                                                     */
/*   RS_N:   total number of bytes including parity bytes, 204 for the case */
/*           of (204,188,8) code.                                          */
/*   scratch: temporary scratch pad array of size 11 * T to fold temporary */
/*            results.                                                     */
/*                                                                         */
/*   The forney routine accepts the syndromes array ”s” computed using the */
/*   syndrome routine, ”lambda” error locator polynomial computed using the */
/*   ”Berleykamp” or equivalent, ”zeros” array that contains the roots of  */
/*   the error locator polynomial, ”byte_i” the received code word and     */
/*   corrects the errors in the received codeword array.                   */
/*                                                                          */
/*                                                                          */
/*  ASSUMPTIONS                                                             */
/*                                                                          */
/* a) s: array of syndromes of size 2T                                      */
/* b) lambda: error locator polynomial computed by   or equiv.              */
/* c) lam_deg: degree of error locator polynomial.                          */
/* d) zeros: zeroes of the error locator polynomial found by Chien search.  */
/* e) byte_i: received code word of size N                                  */
/* f) fail: location where fail status is stored                            */
/* g) None of these arrays overlap in memory.                               */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/* ------------------------------------------------------------------------ */
/*            Copyright (c) 2000 Texas Instruments, Incorporated.           */
/*                           All Rights Reserved.                           */
/* ======================================================================== */
void forney_c(  const  unsigned char *restrict  s,
                       unsigned char *restrict  lambda,
                                int             lam_deg,
                       unsigned char *restrict  zeros,
                       unsigned char *restrict  byte_i,
                                int             *fail,
                 const unsigned char *restrict  *restrict tables,
                                int             T,
                                int             RS_N,
                       unsigned char *restrict  scratch )
{
    int i,j,k;
    /* omega[3T] + syn[4T] + den[T] + num[T] +poly[2T]*/
    unsigned char *omega = scratch;
    unsigned char *syn   = omega  +  3*T;
    unsigned char *numerator = syn + 4*T;
    unsigned char *denominator = numerator + T;
    unsigned char *poly  = denominator   +  T;
    unsigned char err_pos_i, err_mag_i, b;
    unsigned char constant = 0xFF;
    unsigned int lam_t20, lam_word0;
    unsigned int lam_t31, lam_word1;
    unsigned int lam_t64, lam_word2;
    unsigned int lam_t75, lam_word3;
    unsigned int lam_word4, lam_word00;
    unsigned int lam_word5, lam_word01;
    unsigned int lam_word6;
    unsigned int lam_word7;
    unsigned int *syn_ptr;
    unsigned int sword3210, sword7654;
    unsigned int statew0, statew1;
    unsigned int statew2, statew3;
    unsigned int statew4, statew5;
    unsigned int statew6, statew7;
    unsigned int statew8, statew9;
    unsigned int statew10, statew11;
    unsigned int *omega_ptr = (unsigned int *) (omega);
    unsigned int omega_word0, omega_word1;
    unsigned int *lamptr = (unsigned int *) (lambda);
    unsigned int *poly_ptr = (unsigned int *) (poly);
    unsigned int  poly_word0, poly_word1, poly_word2, poly_word3;
    int  RS_T = T;

    const unsigned char *div_inv_table = tables[1];
    const unsigned char *log_table = tables[2];
    double *zerosptr, zero_val;
    unsigned int zeroword0, zeroword1;
    unsigned int denword0,  denword1;
    unsigned int numword0, numword1;
    unsigned char *ptr_omega, *ptr_poly;
    unsigned int   polyval, polyword, omegaval, omegaword;
    unsigned int  denval, numval, zerval;
    unsigned int  denom_i, numer_i, zero_i;
    /*-------------------------------------------------------------*/
    /* Compute omega(x) = S(x)*lambda(x)     Note: the array s[] is */
    /* already in the form of the S(x) syndrome polynomial.         */
    /* omega[0] = lam0 * S0                                         */
    /* omega[1] = lam0 * S1 + lam1 * S0                             */
    /* omega[2] = lam0 * S2 + lam1 * S1 + lam2 * S0                 */
    /* omega[3] = lam0 * S3 + lam1 * S2 + lam2 * S1 + lam3 * S0     */
    /* omega[7] = lam0 * S7 + lam1 * S6 + lam2 * S5 + lam3 * S4     */
    /*          +lam4 * S3 + lam5 * S2 + lam6 * S1                 */
    /*-------------------------------------------------------------*/
    lam_word00 = *lamptr++                   ;  /*lam3210*/
    lam_word01 = *lamptr++                   ;  /*lam7654*/
    lam_t20 = _pack2(lam_word00,  lam_word00);  /* lam20 */
    lam_t31 = _packh2(lam_word00, lam_word00); /* lam31 */
    lam_t64 = _pack2(lam_word01,  lam_word01);  /* lam64 */
    lam_t75 = _packh2(lam_word01, lam_word01); /* lam75 */
    lam_word0 = _packl4(lam_t20, lam_t20)    ;
    lam_word1 = _packh4(lam_t20, lam_t20)    ;
    lam_word2 = _packl4(lam_t31, lam_t31)    ;
    lam_word3 = _packh4(lam_t31, lam_t31)    ;
    lam_word4 = _packl4(lam_t64, lam_t64)    ;
    lam_word5 = _packh4(lam_t64, lam_t64)    ;
    lam_word6 = _packl4(lam_t75, lam_t75)    ;
    lam_word7 = _packh4(lam_t75, lam_t75)    ;
    syn_ptr = (unsigned int *) (s)           ;
    sword3210   = *syn_ptr++                 ;
    sword7654   = *syn_ptr++                 ;
```

```
                                                 /*------------------------*/
statew0     = _gmpy4(lam_word0, sword3210);  /* l0s3 l0s2 l0s1 l0s0   */
statew1     = _gmpy4(lam_word0, sword7654);  /* l0s7 l0s6 l0s5 l0s4   */
sword7654   = _shlmb(sword3210, sword7654);  /* sword7654 = sword6543 */
sword3210   <<= 8;                           /* sword3210 = sword210x */

statew2     = _gmpy4(lam_word1, sword3210);  /* l1s2 l1s1 l1s0  l100  */
statew3     = _gmpy4(lam_word1, sword7654);  /* l1s6 l1s5 l1s4  l1s3  */
sword7654   = _shlmb(sword3210, sword7654);  /* sword7654 = sword5432 */
sword3210   <<= 8;                           /* sword3210 = sword10xx */
statew4     = _gmpy4(lam_word2, sword3210);  /* l2s1 l2s0 l200  l200  */
statew5     = _gmpy4(lam_word2, sword7654);  /* l2s5 l2s4 l2s3  l2s2  */
sword7654   = _shlmb(sword3210, sword7654);  /* sword7654 = sword5432 */
sword3210   <<= 8;                           /* sword3210 = sword10xx */
statew6     = _gmpy4(lam_word3, sword3210);  /* l3s0 l3s0 l300  l300  */
statew7     = _gmpy4(lam_word3, sword7654);  /* l3s4 l3s3 l3s2  l3s1  */
sword7654   = _shlmb(sword3210, sword7654);  /* sword7654 = sword4321 */
sword3210   <<= 8;                           /* sword3210 = sword0xxx */
statew8     = _gmpy4(lam_word4, sword7654);  /* l4s3 l4s2 l4s1  l4s0  */
sword7654   <<= 8;
statew9     = _gmpy4(lam_word5, sword7654);  /* l5s2  l5s1 l5s0  l500 */
sword7654   <<= 8;
statew10    = _gmpy4(lam_word6, sword7654);  /* l6s1 l6s0 l600  l600  */
sword7654   <<= 8;
statew11     = _gmpy4(lam_word7, sword7654);  /* l7s0 l700 l700 l700  */
/*-------------------------------------------------------------------------*/
/* Add partial results together to form the T terms of omega in 2     */
/* words and store them out.                                           */
/*-------------------------------------------------------------------------*/
omega_word0  = statew0 ^ statew2 ^ statew4 ^ statew6;
omega_word1  = statew1 ^ statew3 ^ statew5 ^ statew7 ^
               statew8 ^ statew9 ^ statew10 ^ statew11;
*omega_ptr++ = omega_word0;
*omega_ptr++ = omega_word1;
/*-------------------------------------------------------------------------*/
/* Now compute the derivative of the lambda poly and store the result */
/* The lambda polynomial’s odd terms are ANDED and stored to form     */
/* the derivative.                                                     */
/*-------------------------------------------------------------------------*/
poly_word3 = 0;
poly_word2 = 0;
poly_word0 = (lam_word00 & 0xFF00FF00);
poly_word1 = (lam_word01 & 0xFF00FF00);
*poly_ptr++ = poly_word0;
*poly_ptr++ = poly_word1;
*poly_ptr++ = poly_word2;
*poly_ptr++ = poly_word3;
```

```
/*------------------------------------------------------------*/
/* Now find the error magnitudes and error positions by compu- */
/* ting the numerator and denominator that contain T terms,    */
/* where T is assumed to be utmost 8 in 2 words.               */
/* Two words of numerator are numword0 and numword1            */
/* Two words of the denominator are denword0 and denword1      */
/*------------------------------------------------------------*/
zerosptr = (double *) (zeros);
zero_val = *zerosptr;
zeroword0 = _lo(zero_val);
zeroword1 = _hi(zero_val);
denword1 = denword0 = 0;
numword1 = numword0 = 0;
/*---------------------------------------------------------------*/
/* Omega polynomial and the ”poly” polynomial are read as bytes   */
/* in the next loop, starting with omega[8], poly[9] which is 0   */
/*---------------------------------------------------------------*/
ptr_omega = omega + 7;
ptr_poly  = poly  + 8;
_nassert(RS_T == 8);
/*------------------------------------------------------------------*/
/* Inner loop that iterates for each of the T zeroes is completely   */
/* unrolled and the loop iterates T times building up the result     */
/* of evaluating the omega polynomial and the Lambda’ polynomial     */
/* for all the eight zeroes recursively using Horner’s rule.         */
/* denword0 computes the denominator for the first four zeroes       */
/* denword1 computes the denominator for the second four zeroes      */
/* numword0 computes the numerator for the first four zeroes         */
/* numword1 computes the numerator for the second four zeroes        */
/*------------------------------------------------------------------*/

for (j = RS_T–1; j >= 0; j––)
{
     polyval=*ptr_poly ––;
     polyword=_pack2(polyval,polyval);
     polyword=_packl4(polyword,polyword);
     omegaval=*ptr_omega ––;
     omegaword=_pack2(omegaval,omegaval);
     omegaword=_packl4(omegaword,omegaword);
     denword0=_gmpy4(denword0,zeroword0);
     denword0=(denword0^polyword);
     numword0=_gmpy4(numword0,zeroword0);
     numword0=(numword0^omegaword);
     denword1=_gmpy4(denword1,zeroword1);
     denword1=(denword1^polyword);
     numword1=_gmpy4(numword1,zeroword1);
     numword1=(numword1^omegaword);
}
```

```
        /*-----------------------------------------------------------------*/
        /* Instead of scaling the numerator by alpha^k , the denominator is   */
        /* scaled by alpha^–k which is by definition the k th zero in the     */
        /* zeroes array. This is done by multiplying denword0 * zeroword0     */
        /* denword1 * zeroword1.                                              */
        /*-----------------------------------------------------------------*/
        denword0 = _gmpy4(denword0,zeroword0);
        denword1 = _gmpy4(denword1,zeroword1);
        /*-----------------------------------------------------------------*/
        /* Core loop that performs the division of each byte of the denomina– */
        /* tor with the numerator. j is a loop that switches from one         */
        /* denominator/numerator word to the next after all four bytes of     */
        /* the previous denominator/numerator word have been exhausted        */
        /* Use lookup table to perform division and then multiply numerator   */
        /* by denominator to get error magnitude. Also perform division to    */
        /* convert alpha^–k to alpha^k and then perform a logarithm table     */
        /* lookup to determine k the location. With the error magnitude and   */
        /* location known, the error can be corrected, by performing an XOR   */
        /* of the received data byte with the error magnitude.                */
        /*-----------------------------------------------------------------*/
        j = 4;
        denom_i = denword0;
        zero_i  = zeroword0;
        numer_i = numword0;
        for(i = 0;i < T; i++)
        {
            denval = (denom_i & constant);
            zerval = (zero_i  & constant);
            numval = (numer_i & constant);
            denom_i = (denom_i >> 8);
            zero_i  = (zero_i >> 8);
            numer_i = (numer_i >> 8);
            b = div_inv_table[denval];
            j––;
            if(!j) denom_i = denword1;
            if(!j) numer_i = numword1;
            if(!j) zero_i  = zeroword1;
            if(!j) j = 4;
            err_mag_i = _gmpy4( numval, b);
            err_pos_i = log_table[div_inv_table[zerval]];
            if (!denval)  *fail = 4;
            if (err_pos_i > RS_N–1)  *fail = 5;
            k = RS_N–1 – err_pos_i; /* subtract the error from r(x) */
            if (!*fail) byte_i[k] = ( byte_i[k]^err_mag_i );
       }
    }
```

```
/* ======================================================================== */
/*  End of file:  forney_i.c                                                */
/* ------------------------------------------------------------------------ */
/*            Copyright (c) 2000 Texas Instruments, Incorporated.           */
/*                           All Rights Reserved.
                                                                            */

/* ======================================================================== */
```

The assembly optimizer can be used to obtain optimal performance by writing a serial assembly version of the loop shown above to achieve the correction of up to eight errors in 150 cycles.

5.12 Decoder driver function

The following C code shows how the different pieces are called from C using a simple driver function. The driver function has an additional overhead of 82 cycles in the worst case when T = 8. If there are no errors then it has an overhead of 30 cycles. Thus, the overhead from the driver is about 6.3%. The C code for the driver is shown below:

```
/* ======================================================================== */
/*  NAME                                                                    */
/*                                                                          */
/*      rs_dec_driver-- Driver file for Reed Solomon Decoder                */
/*                                                                          */
/*  USAGE                                                                   */
/*                                                                          */
/*      This routine has the following C prototype:                         */
/*                                                                          */
/*  int rs_dec_n_k( unsigned char  byte_i[],                                */
/*                  unsigned char byte_o[],                                 */
/*                  int * fail_epg)                                         */
/*                                                                          */
/* byte_i: input array of encoded characters of length N                    */
/* byte_o: decoded array of data bytes of length N                          */
/* fail_epg:fail code                                                       */
/*                                                                          */
/* DESCRIPTION                                                              */
/*                                                                          */
/* The Reed Solmon decoder accepts an input array byte_i and corrects upto  */
/* T errors and returns the corrected array in byte_o.                      */
/*                                                                          */
/* Assumptions                                                              */
/*                                                                          */
/* a) The K bytes of byte_o contain valid data bytes                        */
/* b) The parity bytes are no longer valid in byte_o                        */
/* c) The tables shown in this file are required for valid decode           */
/* d) The following are the fail codes                                      */
/*                                                                          */
/* Fail code: 3  Number of zeroes found, is not equal to lam_deg degree  of */
/*            Lambda polynomial                                             */
/*                                                                          */
/* Fail code: 2 Degree of Lambda Polynomial greater than T the maximum      */
/*            number of errors that can be corrected                        */
/*                                                                          */
/* Fail code: 4 Denominator of Forney expression is 0                       */
/* Fail code: 5 Error position is past N – 1                                */
/*                                                                          */
/*--------------------------------------------------------------------------*/
/*            Copyright (c) 1999 Texas Instruments, Incorporated.           */
/*                           All Rights Reserved.                           */
/* ======================================================================== */
/*--------------------------------------------------------------------------*/
/* Arguments for decoder: N, K, T = (204,188,8)                             */
/*--------------------------------------------------------------------------*/
#define RS_N   204
#define RS_K   188
#define RS_T   8
#define RS_2T  RS_T * 2
/*--------------------------------------------------------------------------*/
/* Log–table of entries for the code words:            256 bytes            */
/* Exp_table of entries for the sum of two code words: 512 bytes            */
/* Div_inv_table of entries                          : 256 bytes            */
/* Five beta tables:                                    80 bytes            */
/* Syndrome array for 2T syndromes:                     16 bytes            */
/* Lambda array for T + 1 entries:                       9 bytes            */
/* alpha array of various powers of primitive element:  32 bytes            */
/* for Chien search                                                         */
/* state_asm:                                           36 bytes            */
/* zeros_asm:                                            9 bytes            */
/* scratch:                                             88 bytes            */
/* GF_inv:                                            1024 bytes            */
/* Total Data Memory:                                 2342 bytes            */
/*--------------------------------------------------------------------------*/


unsigned char log_table[256] = {
 255, 0, 1, 25, 2, 50, 26, 198, 3, 223, 51, 238, 27, 104, 199, 75,
 4, 100, 224, 14, 52, 141, 239, 129, 28, 193, 105, 248, 200, 8, 76, 113,
 5, 138, 101, 47, 225, 36,  15, 33,  53, 147, 142, 218, 240,  18, 130,  69,
 29, 181, 194, 125, 106, 39, 249, 185, 201, 154,  9, 120,  77, 228, 114, 166,
 6, 191, 139, 98, 102, 221, 48, 253, 226, 152,  37, 179,  16, 145,  34, 136,
 54, 208, 148, 206, 143, 150, 219, 189, 241, 210, 19, 92, 131,  56,  70,  64,
 30, 66, 182, 163, 195,  72, 126, 110, 107,  58,  40,  84, 250, 133, 186,  61,
 202, 94, 155, 159,  10,  21, 121, 43,  78, 212, 229, 172, 115, 243, 167,  87,
 7, 112, 192, 247, 140, 128,  99,  13, 103,  74, 222, 237,  49, 197, 254,  24,
 227, 165, 153, 119,  38, 184, 180, 124, 17,  68, 146, 217,  35,  32, 137, 46,
 55, 63, 209,  91, 149, 188, 207, 205, 144, 135, 151, 178, 220, 252, 190, 97,
 242, 86, 211, 171,  20,  42,  93, 158, 132,  60, 57,  83,  71, 109,  65, 162,
 31, 45,  67, 216, 183, 123, 164, 118, 196,  23,  73, 236, 127,  12, 111, 246,
108, 161,  59,  82,  41, 157,  85, 170, 251,  96, 134, 177, 187, 204,  62,  90,
203, 89,  95, 176, 156, 169, 160,  81,  11, 245,  22, 235, 122, 117,  44, 215,
 79,174, 213, 233, 230, 231, 173, 232, 116, 214, 244, 234, 168,  80,  88, 175
};
unsigned char exp_table2[512] = {
 1,  2,   4,   8,  16,  32,  64, 128,  29,  58, 116, 232, 205, 135,  19,  38,
 76, 152,  45,  90, 180, 117, 234, 201, 143,  3,   6,  12,  24,  48,  96, 192,
 157, 39,  78, 156,  37,  74, 148,  53, 106, 212, 181, 119, 238, 193, 159, 35,
 70, 140,  5,  10,  20,  40,  80, 160,  93, 186, 105, 210, 185, 111, 222, 161,
 95, 190, 97, 194, 153,  47,  94, 188, 101, 202, 137,  15,  30,  60, 120, 240,
 253, 231, 211, 187, 107, 214, 177, 127, 254, 225, 223, 163,  91, 182, 113, 226,
 217, 175,  67, 134,  17,  34,  68, 136,  13,  26,  52, 104, 208, 189, 103, 206,
 129,  31,  62, 124, 248, 237, 199, 147,  59, 118, 236, 197, 151,  51, 102, 204,
 133,  23,  46,  92, 184, 109, 218, 169,  79, 158,  33,  66, 132,  21,  42,  84,
 168,  77, 154,  41,  82, 164,  85, 170,  73, 146,  57, 114, 228, 213, 183, 115,
 230, 209, 191,  99, 198, 145,  63, 126, 252, 229, 215, 179, 123, 246, 241, 255,
 227, 219, 171,  75, 150,  49,  98, 196, 149,  55, 110, 220, 165,  87, 174,  65,
 130,  25,  50, 100, 200, 141,   7,  14,  28,  56, 112, 224, 221, 167,  83, 166,
 81, 162,  89, 178, 121, 242, 249, 239, 195, 155,  43,  86, 172,  69, 138,   9,
 18,  36,  72, 144,  61, 122, 244, 245, 247, 243, 251, 235, 203, 139,  11,  22,
 44,  88, 176, 125, 250, 233, 207, 131,  27,  54, 108, 216, 173,  71, 142,
 1,   2,   4,   8,  16,  32,  64, 128,  29,  58, 116, 232, 205, 135,  19,  38,
 76, 152,  45,  90, 180, 117, 234, 201, 143,   3,   6,  12,  24,  48,  96, 192,
 157,  39,  78, 156,  37,  74, 148,  53, 106, 212, 181, 119, 238, 193, 159,  35,
 70, 140,   5,  10,  20,  40,  80, 160,  93, 186, 105, 210, 185, 111, 222, 161,
 95, 190,  97, 194, 153,  47,  94, 188, 101, 202, 137,  15,  30,  60, 120, 240,
 253, 231, 211, 187, 107, 214, 177, 127, 254, 225, 223, 163,  91, 182, 113, 226,
 217, 175,  67, 134,  17,  34,  68, 136,  13,  26,  52, 104, 208, 189, 103, 206,
 129,  31,  62, 124, 248, 237, 199, 147,  59, 118, 236, 197, 151,  51, 102, 204,
 133,  23,  46,  92, 184, 109, 218, 169,  79, 158,  33,  66, 132,  21,  42,  84,
 168,  77, 154,  41,  82, 164,  85, 170,  73, 146,  57, 114, 228, 213, 183, 115,
 230, 209, 191,  99, 198, 145,  63, 126, 252, 229, 215, 179, 123, 246, 241, 255,
 227, 219, 171,  75, 150,  49,  98, 196, 149,  55, 110, 220, 165,  87, 174,  65,
 130,  25,  50, 100, 200, 141,   7,  14,  28,  56, 112, 224, 221, 167,  83, 166,
 81, 162,  89, 178, 121, 242, 249, 239, 195, 155,  43,  86, 172,  69, 138,   9,
 18,  36,  72, 144,  61, 122, 244, 245, 247, 243, 251, 235, 203, 139,  11,  22,
 44,  88, 176, 125, 250, 233, 207, 131,  27,  54, 108, 216, 173,  71, 142, 1, 2
 };
unsigned char div_inv_table[256] = {
  0,   1, 142, 244,  71, 167, 122, 186, 173, 157, 221, 152,  61, 170,  93, 150,
 216, 114, 192,  88, 224,  62,  76, 102, 144, 222,  85, 128, 160, 131,  75,  42,
 108, 237,  57,  81,  96,  86,  44, 138, 112, 208,  31,  74,  38, 139,  51, 110,
  72, 137, 111,  46, 164, 195,  64,  94,  80,  34, 207, 169, 171,  12,  21, 225,
  54,  95, 248, 213, 146,  78, 166,   4,  48, 136,  43,  30,  22, 103,  69, 147,
  56,  35, 104, 140, 129,  26,  37,  97,  19, 193, 203,  99, 151,  14,  55,  65,
  36,  87, 202,  91, 185, 196,  23,  77,  82, 141, 239, 179,  32, 236,  47,  50,
  40, 209,  17, 217, 233, 251, 218, 121, 219, 119,   6, 187, 132, 205, 254, 252,
  27,  84, 161,  29, 124, 204, 228, 176,  73,  49,  39,  45,  83, 105,   2, 245,
  24, 223,  68,  79, 155, 188,  15,  92,  11, 220, 189, 148, 172,   9, 199, 162,
  28, 130, 159, 198,  52, 194,  70,   5, 206,  59,  13,  60, 156,   8, 190, 183,
 135, 229, 238, 107, 235, 242, 191, 175, 197, 100,   7, 123, 149, 154, 174, 182,
  18,  89, 165,  53, 101, 184, 163, 158, 210, 247,  98,  90, 133, 125, 168,  58,
  41, 113, 200, 246, 249,  67, 215, 214,  16, 115, 118, 120, 153,  10,  25, 145,
  20,  63, 230, 240, 134, 177, 226, 241, 250, 116, 243, 180, 109,  33, 178, 106,
 227, 231, 181, 234,   3, 143, 211, 201,  66, 212, 232, 117, 127, 255, 126, 253
 };
```

```
/*---------------------------------------------------------------------------*/
/* Array of pointers to tables that is used by Forney                        */
/*---------------------------------------------------------------------------*/
const unsigned char *tables[3] = {exp_table2, div_inv_table, log_table};
/*---------------------------------------------------------------------------*/
/* Header files for Library components of the different algorithms          */
/*---------------------------------------------------------------------------*/
#include ”syn_accumulate_h.h”
#include ”bk_massey_h.h”
#include ”ch_srch_h.h”
#include ”forney_h.h”
/*---------------------------------------------------------------------------*/
/* Align the following arrays to a double word boundary and the GF_inv      */
/*  to a 1K boundary, because circular buffering is used.  This is key      */
/*  for the decoder to work correctly.                                       */
/*---------------------------------------------------------------------------*/
#pragma DATA_ALIGN(beta,           8);
#pragma DATA_ALIGN(beta_RS_N_4,    8);
#pragma DATA_ALIGN(beta_RS_N_2,    8);
#pragma DATA_ALIGN(beta_RS_3N_4,   8);
#pragma DATA_ALIGN(beta_RS_5N_4,   8);
#pragma DATA_ALIGN(syn_ASM,        8);
#pragma DATA_ALIGN(lambda_ASM,     16);
#pragma DATA_ALIGN(zeros_ASM,      8);
#pragma DATA_ALIGN(alpha,          8);
#pragma DATA_ALIGN(state_ASM,      8);
#pragma DATA_ALIGN(scratch,        8);
#pragma DATA_ALIGN(GF_inv,      1024);
/*---------------------------------------------------------------------------*/
/* beta array contains powers of the primitive elements, beta_RS_N_4        */
/* contains the N/4 power of the roots of the generator polynomial,         */
/* beta_RS_N_2 contains the N/2 power of the roots of the generator         */
/*  polynomial and so on. These tables are used by the syndrome accumulate  */
/*  routine, syn_acc.                                                        */
/*-------------------------------------------------------------------------- -*/
unsigned char beta[16] =       {  1,   2,    4,   8,  16,  32,  64, 128,
                                 29,  58,  116, 232, 205, 135,  19,  38
                               };
unsigned char beta_RS_N_4[16]= {  1, 10, 68, 146, 221, 1, 10, 68, 146,
                                221,  1, 10, 68, 146, 221, 1
                               };
unsigned char beta_RS_N_2[16]= {  1, 68, 221,  10, 146, 1,  68, 221, 10,
                                146,  1, 68, 221,  10, 146, 1
                               };
unsigned char beta_RS_3N_4[16]={  1, 146,  10, 221,   68,   1,  146, 10, 221,

                                 68,   1, 146,  10,  221,  68, 1
                               };
unsigned char beta_RS_5N_4[16]={  1,   1, 1,   1,   1,  1,   1,   1,  1,   1,
                                  1,   1, 1,   1,   1,  1
                               };
const unsigned char alpha[] = {
                                2,   4,  8,   16,  32,  64,  128, 29,
                              190, 46, 100, 32,  94,  169, 28,  116,
                              23,  8,  184, 64,  169, 58,  33,  205,
                              25,  92, 47,  128, 28,  33,  30,  19
                            };
unsigned int GF_inv[256] = {
  0x00000000, 0x01010101, 0x8e8e8e8e, 0xf4f4f4f4,
  0x47474747, 0xa7a7a7a7, 0x7a7a7a7a, 0xbabababa,
  0xadadadad, 0x9d9d9d9d, 0xdddddddd, 0x98989898,
  0x3d3d3d3d, 0xaaaaaaaa, 0x5d5d5d5d, 0x96969696,
  0xd8d8d8d8, 0x72727272, 0xc0c0c0c0, 0x58585858,
  0xe0e0e0e0, 0x3e3e3e3e, 0x4c4c4c4c, 0x66666666,
  0x90909090, 0xdededede, 0x55555555, 0x80808080,
  0xa0a0a0a0, 0x83838383, 0x4b4b4b4b, 0x2a2a2a2a,
  0x6c6c6c6c, 0xedededed, 0x39393939, 0x51515151,
  0x60606060, 0x56565656, 0x2c2c2c2c, 0x8a8a8a8a,
  0x70707070, 0xd0d0d0d0, 0x1f1f1f1f, 0x4a4a4a4a,
  0x26262626, 0x8b8b8b8b, 0x33333333, 0x6e6e6e6e,
  0x48484848, 0x89898989, 0x6f6f6f6f, 0x2e2e2e2e,
  0xa4a4a4a4, 0xc3c3c3c3, 0x40404040, 0x5e5e5e5e,
  0x50505050, 0x22222222, 0xcfcfcfcf, 0xa9a9a9a9,
  0xabababab, 0x0c0c0c0c, 0x15151515, 0xe1e1e1e1,
  0x36363636, 0x5f5f5f5f, 0xf8f8f8f8, 0xd5d5d5d5,
  0x92929292, 0x4e4e4e4e, 0xa6a6a6a6, 0x04040404,
  0x30303030, 0x88888888, 0x2b2b2b2b, 0x1e1e1e1e,
  0x16161616, 0x67676767, 0x45454545, 0x93939393,
  0x38383838, 0x23232323, 0x68686868, 0x8c8c8c8c,
  0x81818181, 0x1a1a1a1a, 0x25252525, 0x61616161,
  0x13131313, 0xc1c1c1c1, 0xcbcbcbcb, 0x63636363,
  0x97979797, 0x0e0e0e0e, 0x37373737, 0x41414141,
  0x24242424, 0x57575757, 0xcacacaca, 0x5b5b5b5b,
  0xb9b9b9b9, 0xc4c4c4c4, 0x17171717, 0x4d4d4d4d,
  0x52525252, 0x8d8d8d8d, 0xefefefef, 0xb3b3b3b3,
  0x20202020, 0xecececec, 0x2f2f2f2f, 0x32323232,
  0x28282828, 0xd1d1d1d1, 0x11111111, 0xd9d9d9d9,
  0xe9e9e9e9, 0xfbfbfbfb, 0xdadadada, 0x79797979,
  0xdbdbdbdb, 0x77777777, 0x06060606, 0xbbbbbbbb,
  0x84848484, 0xcdcdcdcd, 0xfefefefe, 0xfcfcfcfc,
  0x1b1b1b1b, 0x54545454, 0xa1a1a1a1, 0x1d1d1d1d,
  0x7c7c7c7c, 0xcccccccc, 0xe4e4e4e4, 0xb0b0b0b0,
  0x49494949, 0x31313131, 0x27272727, 0x2d2d2d2d,
  0x53535353, 0x69696969, 0x02020202, 0xf5f5f5f5,
  0x18181818, 0xdfdfdfdf, 0x44444444, 0x4f4f4f4f,
  0x9b9b9b9b, 0xbcbcbcbc, 0x0f0f0f0f, 0x5c5c5c5c,
  0x0b0b0b0b, 0xdcdcdcdc, 0xbdbdbdbd, 0x94949494,
  0xacacacac, 0x09090909, 0xc7c7c7c7, 0xa2a2a2a2,
  0x1c1c1c1c, 0x82828282, 0x9f9f9f9f, 0xc6c6c6c6,
  0x34343434, 0xc2c2c2c2, 0x46464646, 0x05050505,
  0xcececece, 0x3b3b3b3b, 0x0d0d0d0d, 0x3c3c3c3c,
  0x9c9c9c9c, 0x08080808, 0xbebebebe, 0xb7b7b7b7,
  0x87878787, 0xe5e5e5e5, 0xeeeeeeee, 0x6b6b6b6b,
  0xebebebeb, 0xf2f2f2f2, 0xbfbfbfbf, 0xafafafaf,
  0xc5c5c5c5, 0x64646464, 0x07070707, 0x7b7b7b7b,
  0x95959595, 0x9a9a9a9a, 0xaeaeaeae, 0xb6b6b6b6,
  0x12121212, 0x59595959, 0xa5a5a5a5, 0x35353535,
  0x65656565, 0xb8b8b8b8, 0xa3a3a3a3, 0x9e9e9e9e,
  0xd2d2d2d2, 0xf7f7f7f7, 0x62626262, 0x5a5a5a5a,
  0x85858585, 0x7d7d7d7d, 0xa8a8a8a8, 0x3a3a3a3a,
  0x29292929, 0x71717171, 0xc8c8c8c8, 0xf6f6f6f6,
  0xf9f9f9f9, 0x43434343, 0xd7d7d7d7, 0xd6d6d6d6,
  0x10101010, 0x73737373, 0x76767676, 0x78787878,
  0x99999999, 0x0a0a0a0a, 0x19191919, 0x91919191,
  0x14141414, 0x3f3f3f3f, 0xe6e6e6e6, 0xf0f0f0f0,
  0x86868686, 0xb1b1b1b1, 0xe2e2e2e2, 0xf1f1f1f1,
  0xfafafafa, 0x74747474, 0xf3f3f3f3, 0xb4b4b4b4,
  0x6d6d6d6d, 0x21212121, 0xb2b2b2b2, 0x6a6a6a6a,
  0xe3e3e3e3, 0xe7e7e7e7, 0xb5b5b5b5, 0xeaeaeaea,
  0x03030303, 0x8f8f8f8f, 0xd3d3d3d3, 0xc9c9c9c9,
  0x42424242, 0xd4d4d4d4, 0xe8e8e8e8, 0x75757575,
  0x7f7f7f7f, 0xffffffff, 0x7e7e7e7e, 0xfdfdfdfd
 };
/*---------------------------------------------------------------------------*/
/* syn_ASM: array in which syndromes are computed,                           */
/* lambda_ASM: array in which lambda values are computed                     */
/* zeros_ASM: array in which zeroes are computed                             */
/*---------------------------------------------------------------------------*/
unsigned char syn_ASM[RS_2T];
unsigned char lambda_ASM[RS_T+1];
unsigned char zeros_ASM[RS_T];
/*---------------------------------------------------------------------------*/
/* arrays for performing temporary calculations and holding                  */
/* state information.                                                        */
/*---------------------------------------------------------------------------*/
unsigned char scratch[11 * RS_T];
int           state_ASM[36];
/*---------------------------------------------------------------------------*/
/* Reed Solomon decoder accepts encoded data in array byte_i of size N,      */
/* returning decoded output array in byte_o. Fail codes are returned         */
/*  Fail codes are returned in fail_epg.                                     */
/*---------------------------------------------------------------------------*/
int rs_dec_n_k( unsigned char  byte_i[], unsigned char byte_o[], int *
fail_epg)
{

    int   lam_deg = 0;
    int   error   = 0;
```

```
/*---------------------------------------------------------------------------*/
/* Initialize syndrome array, lambda array, zeros, and scratch area         */
/* using memset                                                              */
/*---------------------------------------------------------------------------*/
    memset( syn_ASM,    0,  sizeof(syn_ASM));
    memset( lambda_ASM, 0,  sizeof(lambda_ASM));
    memset( zeros_ASM,  0,  sizeof(zeros_ASM));
    memset( scratch,    1,  sizeof(scratch));

/*---------------------------------------------------------------------------*/
/* Perform syndrome computation on the received code word. If there are     */
/* no errors*/"error" is set to zero, and this can be used to avoid         */
/* performing other steps when there are no errors.                         */

/*---------------------------------------------------------------------------*/
error = syn_asm( byte_i,         syn_ASM,       0,            beta,
                 beta_RS_N_4,    beta_RS_N_2,   beta_RS_3N_4, beta_RS_5N_4,
                 204,            16 );

/*---------------------------------------------------------------------------*/
/* Initialize fail_epg to zero, and assume that the data to be decoded
/* is vaid.                                                                  */
/*---------------------------------------------------------------------------*/
   *fail_epg = 0;
/*---------------------------------------------------------------------------*/
/* Perform Berley_Kamp Massey algorithm and detemine the Lambda polynomial  */
/*---------------------------------------------------------------------------*/
    bk_massey_asm(   syn_ASM,        GF_inv,        RS_T,         fail_epg,
                   &lam_deg,        lambda_ASM );
/*---------------------------------------------------------------------------*/
/* Perform Chien search algorithm to determine the zeroes of the error      */
/* locator polynomial. If fail code of 3 is set, then it implies that there */
/* are more than T errors. This can be used to exit the decoder, without    */
/* perfoming forney algorithm. However the Forney algorithm can still be    */
/* called without checkinh fail status because no correction of input       */
/* codeword is performed, if fail code is set.                              */
/*---------------------------------------------------------------------------*/

    ch_srch_asm(    lambda_ASM,     lam_deg,       zeros_ASM,    fail_epg,
                    alpha,          state_ASM,     RS_T,         exp_table2,
                    log_table,      div_inv_table);

/*---------------------------------------------------------------------------*/
/* If errors have occurred, call forney to perform correction, otherwise    */
/* exit. This is equivalent to checking if error of the syndome routine     */
/* is non_zero.                                                              */
/*---------------------------------------------------------------------------*/
    if (lam_deg)
    {
        forney_asm    (    syn_ASM,   lambda_ASM,     lam_deg,      zeros_ASM,
                           byte_i,     fail_epg,      tables,       RS_T,
                              RS_N,      scratch );
    }

/*------------------------------------------------------------------------*/
/* Return fail code to calling routine                                    */
/*------------------------------------------------------------------------*/
    return(*fail_epg);
}
```

5.13 Performance of the Reed Solomon Decoder

The following table indicates the cycles for the case of T = 8 and T = 4. There are additional optimizations that can be performed to yield a lower cycle count. In each section the cycle count obtained from the tools by using the compiler, assembly optimizer and by hand coding are mentioned to indicate that the tools can not only reach the performance levels of hand coding but also drastically reduce software development time.

1. Case of T = 8 errors for (204,188,8) code

Table 1. Performance of the Decoder for T = 8 (204,188,8) Code

Name of Module	C Code	Assembly Optimizer	Hand Optimized
Syndrome Accumulate	480 cycles	470 cycles	470 cycles
Chien Search	1110 cycles	326 cycles	318 cycles
Berlekamp-Massey	340 cycles	263 cycles	246 cycles
Forney	180 cycles	154 cycles	150 cycles
Driver Function	80 cycles	80 cycles	80 cycles
Reed Solomon Decoder	2180 cycles	1293 cycles	1268 cycles

The data shown is for the case of the (204,188,8) worst case running time, example when eight errors have occurred in the received data. If there had been no errors all syndromes would have been zero and the decoder can proceed to decode the next block without executing the other three algorithms. Similarly if there are more than eight errors then the Chien search sets the fail flag and no error correction is attempted.

The three columns indicate performance obtained from the existing C compiler at the time this document was being written. The compiler’s performance is likely to improve significantly in the immediate future. The performance data shown compares intrinsic C code, partitioned linear assembly code and hand optimized code. It can be seen that barring the Chien search algorithm, it is possible to obtain performance comparable to hand assembly code directly from C. The performance for the case of T = 8 can be optimized further to obtain a decoder under 1000 cycles. The code-size of each of these modules is also shown below. It can be seen that the code-size is about 3K bytes for the entire decoder.

Table 2. Code Size for (204,188,8) Decoder

Name of Module	C Code	Assembly Optimizer	Hand Optimized
Syndrome Accumulate	1084 bytes	1100 bytes	1128 bytes
Chien Search	792 bytes	920 bytes	872 bytes
Berlekamp-Massey	460 bytes	628 bytes	296 bytes
Forney	696 bytes	1036 bytes	792 bytes
Total Code size	3032 bytes	3684 bytes	3088 bytes

2. Case of T = 4 errors for (102,94,4) code

Table 3. Performance of the Decoder for T = 4 (102,94,4) Code

Name of the Module	C Code	Assembly Optimizer	Hand assembly
Syndrome Accumulate	240 cycles	235 cycles	237 cycles
Chien Search	1110 cycles	326 cycles	318 cycles
Berlekamp-Massey	170 cycles	139 cycles	126 cycles
Forney	90 cycles	77 cycles	75 cycles
Reed Solomon Decoder	1600 cycles	782 cycles	755 cycles

The Chien search code has not been modified for T = 4, and the same code is used, because the performance in this case is limited by the latency, and instead of writing separate code, the code is re-used. The performance for T = 4 case can be further optimized to be under 700 cycles. Since the same code is used for both T = 8 and T = 4, the code-size table remains the same.

6 References

1. Texas Instruments Inc. *The eXpressDSP Algorithm Standard (xDAIS): Rules and Guidelines*, SPRU352, September 1999.
2. *Error Control and Coding* by Costello and J. Linn Prentice Hall
3. *TMS320C6000 CPU and Instruction Set* SPRU189

7 Acknowledgements

The author would like to thank David Hoyle and Todd Wolf for their contributions to the C64x implementation of the Reed Solomon Decoder.

8 Conclusions

This application report demonstrates the implementation of a (204, 188, 8) Reed Solomon decoder on the TMS320C64x architecture. Techniques that yield highly parallel implementations of the individual pieces of the Petersen-Gorenstein-Zierler (PGZ) algorithm were presented. These techniques clearly demonstrate the benefits of having DSPs that provide support for Galois Field based operations. These algorithms are able to make maximum use of the Galois field multiplier bandwidth and the availability of eight distinct units to sustain parallel processing on the TMS320C64x architecture. It has been shown that the optimizing C compiler of the C6000 architecture is able to extract data and instruction level parallelism on the complicated algorithms involved in developing the Reed Solomon decoder, thus minimizing the software development time and effort.

At clock speeds of 600 – 800MHZ, the Reed Solomon decoder for a 6Mbytes/sec ADSL channel requires a processing load of 4.75 % – 6 %. This allows for a multichannel implementation of the Reed Solomon decoder on a C64x DSP.