

Time domain algorithms and architectures for Reed–Solomon decoding

S. Choomchuay
B. Arambepola

Indexing terms: Error correction, Reed–Solomon decoding, VLSI

Abstract: Reed–Solomon decoders can be designed using time or frequency domain algorithms. The main advantage of the time domain method is its regular computational structure, an attractive feature for a flexible VLSI implementation. The main disadvantage of this is the relatively high computation count, hence it is important to find means of reducing the computational complexity of the time domain method. Several modifications to the time domain Reed–Solomon decoding algorithm are described for reducing its computational complexity by about 60%, as well as regular VLSI architectures for implementing the time domain algorithm. The paper concludes with a discussion of the advantages and disadvantages of the time domain decoding technique.

1 Introduction

Error control coding has become an essential means of ensuring data integrity in a variety of applications including satellite and mobile communications and the storage of data in magnetic and optical media. Among the various codes available for correcting multiple errors, the Reed–Solomon code has emerged as one of the most important codes. The notation $RS(N, k)$ is commonly used to denote a Reed–Solomon code of block size N symbols with each block containing k information symbols. Each symbol is an n bit word, which is usually interpreted as an element in $GF[2^n]$. This can correct for t errors, where $t = (N - k)/2$. Encoding of a Reed–Solomon code is usually simpler than decoding. In this paper we consider algorithms for decoding which lead to regular VLSI architectures comprising arrays of identical and locally connected cells.

Reed–Solomon decoding can be carried out directly in the domain in which the data is received, or by the transforming the data into another domain using a Galois field discrete Fourier transform (DFT). The last-mentioned method, now known as frequency domain decoding, has been widely employed in the design of

Reed–Solomon decoders. This traditionally begins with syndrome computation. Syndromes constitute $2t$ consecutive frequency components of the data block. The key equation is then solved to obtain the error locator polynomial. The results are then inverse transformed using the polynomial evaluation (which is essentially an IDFT) or using a fast IDFT algorithm following recursive extension of the frequency domain error sequence [1].

The time domain algorithm has been developed by Blahut [2] by taking the inverse DFT of all the sequences and operators of the Berlekamp–Massey algorithm [3]. This operates directly on the received data and generates the error sequence in the domain in which the data is received. Hence it eliminates the need for transform and inverse transform operators such as syndrome computation and Chien search. As a result, the complete algorithm can be implemented by the repeated application of a single operation, which is of importance for VLSI implementations.

However, the main drawback of the time domain algorithm is its high computation count. This is brought about by the fact that time domain key equation solving algorithm has to operate on the complete data sequence of length N , while the frequency domain algorithm needs to work only on the syndrome sequence of length $N - k$. Hence, to make the time domain algorithm attractive to practical applications it is important to reduce the computation count. This is the first objective of this paper. We show that it is possible to reduce the multiplication count of the time domain algorithm given by Blahut [2] by about 60%. The second objective is to provide regular array architectures for implementing this algorithm using VLSI technology.

A time domain architecture for Reed–Solomon decoding presented in Reference 4 has employed the recursive extension approach for error evaluation. The method presented in our paper has several advantages over such technique. Firstly, the algorithm described here has much fewer computations. Secondly, our algorithm can be implemented as a single array architecture, while that of Reference 4 needs one array for obtaining the error locator and another for recursive extension. Thirdly, the algorithm given here can be very easily applied to truncated codes, simply by truncating all the sequences in the algorithm or the architecture. It is not possible to do this for the architecture given in Reference 4.

© IEE, 1993

Paper 94101 (E5, E8), first received 9th October 1991 and in final revised form 15th October 1992

S. Choomchuay is with the Department of Electrical Engineering, Imperial College of Science, Technology and Medicine, Exhibition Road, London SW7 2BT, United Kingdom

B. Arambepola is with the GEC-Marconi Research Centre, West Hanningfield Road, Great Baddow, Chelmsford, Essex, CM2 8HN, United Kingdom

S. Choomchuay is grateful to the British Council for providing the financial support to carry out this research, and to Peter Cheung of Imperial College for encouragement and advice.

2 Frequency domain algorithms

The main steps of the frequency domain decoding algorithm are shown in Fig. 1. Let $N - k = 2t$. The algorithm begins by computing $2t$ syndromes S_1 to S_{2t} of the

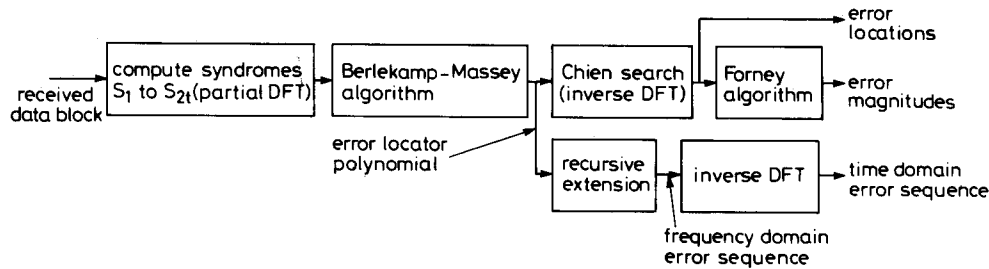


Fig. 1 Frequency domain decoding algorithm (two approaches shown)

received data block $\{v_i, i = 0, 1, \dots, N - 1\}$. These $2t$ syndromes constitute a cyclically contiguous block of $2t$ frequency components in the N -point DFT of $\{v_i\}$. Hence, syndrome computation is equivalent to computing a partial DFT. The N -point DFT and inverse DFT are defined in eqn. 1 by using an N th primitive root of unity α as the Galois field as the kernel.

$$V_k = \sum_{i=0}^{N-1} v_i \alpha^{ik} \quad \text{for } k = 0, 1, \dots, N - 1 \quad (1a)$$

$$v_i = \sum_{k=0}^{N-1} V_k \alpha^{-ik} \quad \text{for } i = 0, 1, \dots, N - 1 \quad (1b)$$

The syndromes S_1 to S_{2t} are the DFT components V_k , $k = t_0, t_0 + 1, \dots, t_0 + 2t - 1$, where k is taken modulo N . The value t_0 is defined by the generator polynomial used in the encoder. These $2t$ syndromes are used to obtain the locations and magnitudes of the t or fewer errors. This can be carried out using the Euclid algorithm, or the Berlekamp-Massey algorithm which we choose for our application.

The Berlekamp-Massey algorithm gives the tap weights of the shortest linear feedback shift register which can recursively generate all $2t$ syndromes. The polynomial with these tap weights as coefficients is known as the error locator polynomial $\Lambda(x)$. The roots of this polynomial define the location of the errors, i.e. if α^{-i} is a root of $\Lambda(x)$ then the i th position of the N -point time domain sequence contains an error. Alternatively, the coefficient sequence of $\Lambda(x)$ can be inverse transformed after padding with zero-valued samples up to length N . The zero-valued samples of the resulting IDFT denote the error locations.

The improvement of the Berlekamp-Massey procedure over the conventional algorithm (described in Reference 5) is given by eqn. 2 in Reference 6. In such modifications, a scalar term K is introduced and about t multiplications can be saved in each iteration. Moreover, postponing of inversion of Δ_r (Δ_r^{-1} is not used in the iteration in which it is computed) allows the inversion to be carried out leisurely, in parallel with polynomial updating. Fast inversion hardware, which is infrequently used, can be avoided.

Since Reed-Solomon is a nonbinary code error magnitudes are also needed for correcting errors. One way of obtaining the error magnitudes is by using the recursive extension procedure shown in Fig. 1. The Berlekamp-Massey algorithm gives the tap weights of a feedback shift register for generating the $2t$ syndromes. If this register is clocked $(N - 2t)$ cycles after initialising it with any contiguous syndrome block we obtain the discrete

Fourier transform of the error vector. Taking the inverse transform of this gives the time domain error vector. Hence error locations and magnitudes are computed together using this procedure, as illustrated in Fig. 1.

Although the shift register extension method is conceptually simple, it can be expensive in terms of computation.

The other method of error evaluation is known as the Forney algorithm. In this method the error magnitude at location i is

$$e_i = \frac{\Omega(x)}{x^{2-t_0} \Lambda'(x)} \quad \text{at } x = \alpha^{-i} \quad (2a)$$

where

$$\Omega(x) = (S_1 x + S_1 x^2 + \dots + S_{2t} x^{2t}) \Lambda(x) \bmod x^{2t} \quad (2b)$$

Instead of computing error magnitudes outside the iteration process, the algorithm given in Reference 6 can be written to provide both the error locations and magnitudes as follows:

Initialise: $\Lambda^0(x) = \Gamma^0(x) = \Omega^0(x) = 1$;

$$Z^0(x) = \Xi^0(x) = M^0(x) = 0;$$

$$L_0 = 0;$$

$$K_0 = 1;$$

$$\Delta_r = \sum_{j=0}^r \Lambda_j^{(r-1)} S_{r-j} \quad (3a)$$

$$L_r = \delta_r (r - L_{r-1}) + (1 - \delta_r) L_{r-1}$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r) K_{r-1}$$

$$\begin{bmatrix} \Lambda^r(x) \\ \Gamma^r(x) \\ Z^r(x) \\ \Xi^r(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} x & 0 & 0 \\ \delta_r & (1 - \delta_r) x & 0 & 0 \\ 0 & -\Delta_r K_{r-1} x^{(2-t_0)} & 1 & -\Delta_r K_{r-1} x \\ 0 & (1 - \delta_r) x^{(2-t_0)} & \delta_r & (1 - \delta_r) x \end{bmatrix} \times \begin{bmatrix} \Lambda^{r-1}(x) \\ \Gamma^{r-1}(x) \\ Z^{r-1}(x) \\ \Xi^{r-1}(x) \end{bmatrix} \quad (3b)$$

$$\begin{bmatrix} \Omega^r(x) \\ M^r(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} x \\ \delta_r & (1 - \delta_r) x \end{bmatrix} \begin{bmatrix} \Omega^{r-1}(x) \\ M^{r-1}(x) \end{bmatrix} \quad (3c)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$. Then

$$e_i = \frac{\Omega^{2t}(x)}{Z^{2t}(x)} \quad \text{where } x = \alpha^{-i} \text{ is a root of } \Lambda^{2t}(x) \quad (3d)$$

The algorithm in eqn. 3 is a modification of the matrix formulation given by Blahut [2]. First, Blahut's algo-

rithm has been modified to minimise multiplication count and postpone the inversion of Δ_r , as noted. Hence eqn. 3 has about 37% fewer multiplications than Blahut's method. This can be seen by comparing the matrices in eqns. 3b and 3c with the corresponding matrices in the frequency domain decoding algorithm given in Reference 2 (page 153). The above matrices have four nontrivial entries while those in Reference 2 have seven. Each nontrivial entry corresponds to t multiplications per iteration. Additionally, t multiplications are needed per iteration for computing Δ_r . Hence the algorithm given has $5t$ multiplications per iteration, while the corresponding algorithm given in Reference 2 has $8t$ multiplications per iteration. Since both algorithms have $2t$ iterations, this algorithm has about 37% fewer multiplications.

Secondly, Blahut's algorithm has been modified to enable the selection of any $2t$ cyclically contiguous block of frequency components as the syndromes. The parameter t_0 , which can be any integer in the range 0 to $N - 1$, defines the beginning of the syndrome set. Although the choice of t_0 has no effect on the performance of the Reed-Solomon code, certain values of t_0 lead to simpler encoder realisation. An example of this is found in the bit-serial encoder design given by Berlekamp in Reference 7. Also, it is necessary for the decoder design to conform to international standards. The telemetry standards given by CCSDS and ESA in References 8 and 9 specify syndrome locations not beginning from the first frequency component.

3 Time domain algorithms

In the RS decoding procedure given in Section 2, the Berlekamp-Massey algorithm is applied to determine the error locator polynomial after transforming the data into the frequency domain using a Galois field DFT. The time domain algorithm is developed as in Blahut [2] by taking the inverse DFT of all the sequences and operators required by the frequency domain Berlekamp-Massey algorithm. This algorithm operates directly on the received data and generates the error sequence in the domain in which the data is received, as depicted in Fig. 2.

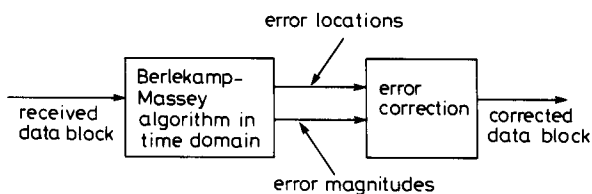


Fig. 2 Time domain decoding

3.1 Time domain version of eqn. 3 algorithm

The time domain algorithm is obtained by inverse transforming all the signal sequences of the frequency domain algorithm described in Section 2. Consider inverse-transforming the sequences of the algorithm in eqn. 3. The received data vector $\{v_i\}$ can be used as the inverse transform of the syndrome vector $\{S_k\}$. The sequence $\{\lambda_i\}$ is defined as the inverse transform of the coefficients $\{\Lambda_k\}$ of the error locator polynomial. The locations at which $\lambda_i = 0$ contain errors.

Eqn. 3 shows that Δ_r is the r th component of the convolution between sequences $\{S_k\}$ and $\{\Lambda_k\}$. This convolution maps into a set of pointwise multiplications between sequences $\{v_i\}$ and $\{\lambda_i\}$ in the time domain. Since S_1 is

the DFT component t_0 of $\{v_i\}$, Δ_r is the $(r + t_0 - 1)$ th DFT component of the product sequence $\{v_i \lambda_i\}$, as in eqn. 4a.

The multiplication of a polynomial by a power of x , say x^d , corresponds to a right shift of d of the sequence of polynomial coefficients. Fourier transform theory states that a frequency shift of d samples corresponds to multiplying the i th sample of the time domain sequence by α^{-id} . Using these two properties of the DFT, the frequency domain decoding algorithm of eqn. 3 can be mapped into an equivalent time domain algorithm, referred to here as algorithm A.

Algorithm A:

$$\text{Initialise: } \lambda_i^0 = \gamma_i^0 = \omega_i^0 = 1;$$

$$\zeta_i^0 = \xi_i^0 = \mu_i^0 = 0;$$

$$L_0 = 0;$$

$$K_0 = 1;$$

For $i = 0, 1, \dots, N - 1$

$$\Delta_r = \sum_{i=0}^{N-1} v_i \lambda_i^{r-1} \alpha^{i(r+t_0-1)} \quad (4a)$$

$$L_r = \delta_r (r - L_{r-1}) + (1 - \delta_r) L_{r-1}$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r) K_{r-1}$$

$$\begin{bmatrix} \lambda_i^r \\ \gamma_i^r \\ \zeta_i^r \\ \xi_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i} & 0 & 0 \\ \delta_r & (1 - \delta_r) \alpha^{-i} & 0 & 0 \\ 0 & -\Delta_r K_{r-1} \alpha^{-i(2-t_0)} & 1 & -\Delta_r K_{r-1} \alpha^{-i} \\ 0 & (1 - \delta_r) \alpha^{-i(2-t_0)} & \delta_r & (1 - \delta_r) \alpha^{-i} \end{bmatrix} \times \begin{bmatrix} \lambda_i^{r-1} \\ \gamma_i^{r-1} \\ \zeta_i^{r-1} \\ \xi_i^{r-1} \end{bmatrix} \quad (4b)$$

$$\begin{bmatrix} \omega_i^r \\ \mu_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i} \\ \delta_r & (1 - \delta_r) \alpha^{-i} \end{bmatrix} \begin{bmatrix} \omega_i^{r-1} \\ \mu_i^{r-1} \end{bmatrix} \quad (4c)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$. Then

$$e_i = \frac{\omega_i^{2t}}{\zeta_i^{2t}} \quad \text{wherever } \lambda_i = 0, e_i = 0 \text{ elsewhere} \quad (4d)$$

The key difference between the time domain and frequency domain algorithms is the fact that the syndrome computation and polynomial evaluation stages are not present in the time domain algorithm. Hence the time domain algorithm involves the repeated application of a single matrix operation.

Both time and frequency domain error location algorithms require $2t$ iterations. However, the frequency domain algorithm deals with sequences of length t while all the sequences used in the time domain formulation are of length N .

3.2 Improved time domain algorithms

Every nontrivial entry (i.e. an entry which is neither 0 nor 1) in the matrices of eqn. 4 results in N multiplications per each of the $2t$ iterations. Hence the first step in the optimisation procedure is concerned with minimising the number of nontrivial entries in the matrices. One such reduction compared to Blahut's original algorithm has

already been described in eqn. 3 of Section 2. Methods of achieving further reduction are outlined in this Section.

In addition to minimising nontrivial entries in the matrices, it is beneficial to make all the nontrivial entries identical. First, this reduces the computations needed to generate these matrices; secondly, it allows the number of Galois field multiplications needed to carry out the matrix-vector products to be reduced.

We first modify the frequency domain algorithm given by eqn. 2 of Reference 6 into the form given by frequency domain algorithm B.

Algorithm B:

Initialise: $\Lambda^0(x) = \Gamma^0(x) = 1$;

$$L_0 = 0;$$

$$K_0 = u_0 = 1;$$

$$\Delta_r = \sum_{j=0}^r \Lambda_j^{-1} S_{r-j} \quad (5a)$$

$$L_r = \delta_r(r - L_{r-1}) + (1 - \delta_r)L_{r-1} \quad (5b)$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r)K_{r-1} \quad (5c)$$

$$u_r = \delta_r + (1 - \delta_r)(u_{r-1} + 1) \quad (5d)$$

$$\begin{bmatrix} \Lambda^r(x) \\ \Gamma^r(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} x^{u_r} \\ \delta_r & (1 - \delta_r) \end{bmatrix} \begin{bmatrix} \Lambda^{r-1}(x) \\ \Gamma^{r-1}(x) \end{bmatrix} \quad (5e)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$.

There are two differences between algorithm B and the algorithm in eqn. 2 of Reference 6. First, in algorithm B the matrix entry $(1 - \delta_r)$ is not multiplied by x . Second, an additional scalar parameter u_r has been used in algorithm B. This is updated in accordance with eqn. 5d in each iteration. Note that the addition in eqn. 5d is an ordinary integer addition, and not an addition in the Galois field $GF[2^n]$. The parameter u_r is referred to here as the shift-index accumulation parameter for reasons which will become apparent.

In the Reference 6 algorithm, when δ_r is zero the polynomial $\Gamma^r(x)$ is formed by multiplying $\Gamma^{r-1}(x)$ by x . Hence if there are k successive iterations in which $\delta_r = 0$, then $\Gamma^{r-1}(x)$ gets multiplied by x^k . In algorithm B, the polynomial $\Gamma^{r-1}(x)$ is not multiplied by x ; instead, the number of successive times $\delta_r = 0$ is held in the shift-index accumulation parameter u_r . Then in obtaining polynomial $\Lambda^r(x)$ we use $\Gamma^{r-1}(x)$ multiplied by x^{u_r} .

It is clear that polynomial $\Lambda^r(x)$ in the algorithm B is the same as the polynomial denoted by the same symbol in Reference 6, for all r from 0 to $2t$. Hence both algorithms yield the same error locator polynomial. However, the auxiliary polynomial $\Gamma^r(x)$ used in algorithm B is different from the corresponding polynomial in Reference 6 in the iterations in which $\delta_r = 0$. This is irrelevant as the auxiliary polynomial is used simply as an aid for obtaining the error locator polynomial.

It is also clear that algorithm B does not have any significant computational advantages over the algorithm obtained using eqn. 2 of Reference 6. If there is a minor advantage, it is in the fact that no polynomial shift operations (multiplications by x) are needed in obtaining $\Gamma^r(x)$. However, it will soon be seen that the new version of the frequency domain Berlekamp Massey algorithm of eqn. 5 leads to a simpler time domain algorithm.

First expand algorithm B to include all polynomials needed for error location and error evaluation using eqn.

2a. Let two polynomials be defined by eqn. 6:

$$Z^r(x) = x^{2-t_0} \frac{d\Lambda^r(x)}{dx} \quad (6a)$$

$$\Xi^r(x) = x^{2-t_0} \frac{d\gamma^r(x)}{dx} \quad (6b)$$

One then obtains the algorithm C given by eqn. 7.

Algorithm C:

Initialise: $\Lambda^0(x) = \Gamma^0(x) = \Omega^0(x) = 1$;

$$Z^0(x) = \Xi^0(x) = M^0(x) = 0;$$

$$L_0 = 0;$$

$$K_0 = u_0 = 1;$$

$$\Delta_r = \sum_{j=0}^r \Lambda_j^{-1} S_{r-j} \quad (7a)$$

$$L_r = \delta_r(r - L_{r-1}) + (1 - \delta_r)L_{r-1}$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r)K_{r-1}$$

$$\begin{bmatrix} \Lambda^r(x) \\ \Gamma^r(x) \\ Z^r(x) \\ \Xi^r(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} x^{u_r} & 0 & 0 \\ \delta_r & (1 - \delta_r) & 0 & 0 \\ 0 & -\Delta_r K_{r-1} u_r x^{u_r+1-t_0} & 1 & -\Delta_r K_{r-1} x^{u_r} \\ 0 & 0 & \delta_r & (1 - \delta_r) \end{bmatrix}$$

$$\times \begin{bmatrix} \Lambda^{r-1}(x) \\ \Gamma^{r-1}(x) \\ Z^{r-1}(x) \\ \Xi^{r-1}(x) \end{bmatrix} \quad (7b)$$

$$\begin{bmatrix} \Omega^r(x) \\ M^r(x) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} x^{u_r} \\ \delta_r & (1 - \delta_r) \end{bmatrix} \begin{bmatrix} \Omega^{r-1}(x) \\ M^{r-1}(x) \end{bmatrix} \quad (7c)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$. If $\delta_r = 0$, $u_r = u_{r-1} + 1$; otherwise $u_r = 1$. Then

$$e_i = \frac{\Omega^{2t}(x)}{Z^{2t}(x)} \quad \text{where } x = \alpha^{-i} \text{ is a root of } \Lambda^{2t}(x) \quad (7d)$$

The number of nontrivial entries in the matrices of algorithm C is the same as that in eqn. 3. Therefore the number of computations in algorithm C is the same as that in eqn. 3. Hence frequency domain algorithm C has no significant advantage over the frequency domain algorithm given by eqn. 3. However, algorithm C leads to a much simpler time domain algorithm for the following reason.

Section 3.1 noted that multiplication of a polynomial by x in the frequency domain is equivalent to multiplying the samples of the time domain sequence by successive powers of α^{-1} . Although powers of x in the rows of the matrices of the frequency domain algorithm do not count as nontrivial multiplications, these do map into nontrivial multiplications in the time domain algorithm. Hence the motivation for developing the frequency domain algorithm C has been to minimise the number of the occurrences of powers of x in the matrices.

The time domain algorithm, obtained by taking the IDFT of algorithm C, is given as algorithm D. Rows 2 and 4 of the 4×4 matrix of eqn. 7b and row 2 of eqn. 7c are mapped into identical rows in the matrices of eqns. 8b and 8c. These rows do not contain any power of α , as in the case of algorithm A.

Algorithm D:

$$\text{Initialise: } \lambda_i^0 = \gamma_i^0 = \omega_i^0 = 1;$$

$$\zeta_i^0 = \xi_i^0 = \mu_i^0 = 0;$$

$$L_0 = 0;$$

$$K_0 = u_0 = 1;$$

for $i = 0, 1, \dots, N - 1$

$$\Delta_r = \sum_{i=0}^{N-1} v_i \lambda_i^{r-1} \alpha^{i(r+t_0-1)} \quad (8a)$$

$$L_r = \delta_r(r - L_{r-1}) + (1 - \delta_r)L_{r-1}$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r)K_{r-1}$$

$$\begin{bmatrix} \lambda_i^r \\ \gamma_i^r \\ \zeta_i^r \\ \xi_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i u_r} & 0 & 0 \\ \delta_r & (1 - \delta_r) & 0 & 0 \\ 0 & -\Delta_r K_{r-1} u_r \alpha^{-i(u_r+1-t_0)} & 1 & -\Delta_r K_{r-1} \alpha^{-i u_r} \\ 0 & 0 & \delta_r & (1 - \delta_r) \end{bmatrix} \times \begin{bmatrix} \lambda_i^{r-1} \\ \gamma_i^{r-1} \\ \zeta_i^{r-1} \\ \xi_i^{r-1} \end{bmatrix} \quad (8b)$$

$$\begin{bmatrix} \omega_i^r \\ \mu_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i u_r} \\ \delta_r & (1 - \delta_r) \end{bmatrix} \begin{bmatrix} \omega_i^{r-1} \\ \mu_i^{r-1} \end{bmatrix} \quad (8c)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$. If $\delta_r = 0$, $u_r = u_{r-1} + 1$; otherwise $u_r = 1$. Then

$$e_i = \frac{\omega_i^{2t}}{\zeta_i^{2t}} \quad \text{where } \lambda_i = 0 \quad (8d)$$

Computational requirements of algorithm D are studied in Section 3.4. A method of reducing the storage requirement of the time domain algorithm is discussed subsequently. First note that algorithm D requires seven data sequences, each of length N , to be stored. For large values of N this will have an impact on the hardware complexity. One method of reducing the storage requirement is by computing some of the sequences outside the array structure.

In algorithm E, iterations are carried out only for sequences λ_i , γ_i , ω_i and μ_i . The sequence ζ_i is needed only at error locations to evaluate error magnitudes, i.e. only at the locations where $\lambda_i = 0$. This is evaluated after the $2t$ time domain iterations are completed. (The proof of the algorithm is given in Section 7.)

Algorithm E:

$$\text{Initialise: } \lambda_i^0 = \gamma_i^0 = \omega_i^0 = 1;$$

$$\mu_i^0 = 0;$$

$$L_0 = 0;$$

$$K_0 = u_0 = 1;$$

For $i = 0, 1, \dots, N - 1$

$$\Delta_r = \sum_{i=0}^{N-1} v_i \lambda_i^{r-1} \alpha^{i(r+t_0-1)} \quad (9a)$$

$$L_r = \delta_r(r - L_{r-1}) + (1 - \delta_r)L_{r-1}$$

$$K_r = \delta_r \Delta_r^{-1} + (1 - \delta_r)K_{r-1}$$

$$\begin{bmatrix} \lambda_i^r \\ \gamma_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i u_r} \\ \delta_r & (1 - \delta_r) \end{bmatrix} \begin{bmatrix} \lambda_i^{r-1} \\ \gamma_i^{r-1} \end{bmatrix} \quad (9b)$$

$$\begin{bmatrix} \omega_i^r \\ \mu_i^r \end{bmatrix} = \begin{bmatrix} 1 & -\Delta_r K_{r-1} \alpha^{-i u_r} \\ \delta_r & (1 - \delta_r) \end{bmatrix} \begin{bmatrix} \omega_i^{r-1} \\ \mu_i^{r-1} \end{bmatrix} \quad (9c)$$

for $r = 1, 2, \dots, 2t$. Here $\delta_r = 1$ if both $\Delta_r \neq 0$ and $2L_{r-1} \leq r - 1$; otherwise $\delta_r = 0$. If $\delta_r = 0$, $u_r = u_{r-1} + 1$; otherwise $u_r = 1$.

$$\zeta_i = \alpha^{-i(t_0-1)} \sum_{k=0}^{N-1} h_k \lambda_{\langle i-k \rangle_N} \quad \text{where } \lambda_i = 0 \quad (9d)$$

$\langle \cdot \rangle_N$ denotes modulo N , and $h_k = \alpha^{-k} + \alpha^{-3k} + \dots + \alpha^{-k(N-2)}$. Then

$$e_i = \frac{\omega_i^{2t}}{\zeta_i}$$

Algorithm E does not have any computational advantages over algorithm D, however, it reduces the storage requirement in a fully pipelined implementation as it has two fewer sequences to store than algorithm D.

3.3 Truncated Reed-Solomon codes

Many practical applications of error control coding involves the use of truncated or shortened Reed-Solomon codes. Let $RS(N, k)$ be a cyclic Reed-Solomon code of block length N containing k message symbols. Then the truncated code $RS(N_t, k - N + N_t)$ of block length N_t is obtained by dropping $N - N_t$ message symbol positions from the code word.

For decoding we conceptually insert zero-valued samples for all the symbols dropped from the code during truncation. Zero-valued symbols do not contribute to any of the computations in algorithm D. Hence algorithm D can be used for truncated codes simply by truncating all the sequences of the algorithm to a length of N_t . For example, if the dropped symbols correspond to $i = N$ to $N_t - 1$, the time domain decoding algorithm for the truncated code will be algorithm D with the lower and upper limits of i set to 0 and $N_t - 1$.

Hence the time domain decoding algorithm for truncated codes is obtained simply by truncating all the sequences of algorithm D from length N to length N_t . Hence the computational count and storage requirements are reduced by a factor (N_t/N) . Furthermore, it is a very simple task to adapt a RS decoder designed for cyclic code for a truncated code of any length.

3.4 Computational requirements

Algorithm D is applicable to any value of t_0 , where α^{-t_0} is the first root of the generator polynomial used in the decoder. Hence there is a term containing t_0 in the 4×4 matrix in eqn. 8b. This term can be simplified if t_0 is made equal to 1. Fortunately, t_0 can be made equal to 1 in the decoder even though the actual value of t_0 used in the encoder may be different from 1.

To illustrate this by example, let $t_0 = 5$. Then the $2t$ syndromes are the frequency components 5 to $5 + 2t - 1$ of input data sequence $\{v_i\}$. If each sample v_i is multiplied by α^{4i} then the DFT of $\{v_i\}$ gets cyclically left-shifted by four sample spaces. Then the syndrome of the modified sequence will occupy frequency locations 1 to $2t$, i.e. effectively $t_0 = 1$ for the decoder. After the sequence has been decoded and corrected, the original spectral positions are restored by multiplication by α^{-4i} .

If t_0 is set to unity, all nontrivial entries of the matrices of eqn. 8 are identical. (The second term of the first row

of the 4×4 matrix does contain an additional multiplicative factor u_r . This takes the values 0 or 1 since in $GF[2^m]$ this is taken modulo 2.) Benefits of making all matrix entries equal has been listed at the beginning of Section 3.1. The total number of nontrivial entries in both matrices is four. Each nontrivial entry corresponds to N multiplications per each iteration. Note that the matrices of the corresponding time domain algorithm given by Blahut in page 154 of Reference 2 have 10 nontrivial entries.

Consider the number of computations required by algorithm D with t_0 chosen as unity. The value of the four identical nontrivial entries in the matrices is $\Delta_r K_{r-1} \alpha^{-i u_r}$. Here Δ_r , K_{r-1} and u_r are all scalars, not sequences. Also these remain unchanged for one whole iteration. Hence it takes approximately N multiplications to compute $\Delta_r K_{r-1} \alpha^{-i u_r}$ for all N values of i .

After obtaining the matrix coefficients as described it is expected to take $4N$ multiplications to update the vectors in one iteration, i.e. N multiplications per nontrivial entry. This multiplication count can be reduced to $3N$ by making use of the fact that the two nontrivial entries in row 3 of the 4×4 matrix are identical.

In addition, $2N$ multiplications are needed per iteration to obtain Δ_r . Hence the total number of multiplications per iteration is $6N$. The complete algorithm, comprising $2t$ iterations, contains $12Nt$ multiplications.

Algorithm D is now compared with the corresponding time domain algorithm given in page 154 of Reference 2. The matrices of Reference 2 contain 10 nontrivial entries, not all of which are identical. It can be shown that $2N$ multiplications per iteration are needed to form the matrix. Following this, $10N$ multiplications are needed to carry out the matrix-vector multiplications of one iteration. Hence the total number of multiplications per iteration (including $2N$ multiplications for obtaining Δ_r) is $14N$. For the $2t$ iterations $28Nt$ multiplications are needed. Hence the algorithm D has 57% fewer multiplications than the corresponding algorithm in Reference 2.

It is also necessary to compare the algorithm D with the time domain recursive extension algorithm in page 153 of Reference 2 derived from the frequency domain recursive extension algorithm mentioned in Section 2 (Fig. 1). As noted by Blahut, it is not possible to inverse transform the entire frequency domain recursive extension into the time domain in a single step. Each of the $N - 2t$ recursive extension operations has to be inverse transformed separately into the time domain.

From Reference 2, each time domain recursive extension stage takes $4N$ multiplications. There are $N - 2t$ recursive extensions. In addition, it takes $14Nt$ multiplications to obtain the time domain error locator sequence using the algorithm of Reference 2. (This number can be reduced to $8Nt$ using the methods described in this paper). This leads to a total multiplication-count of $4N^2 + 6Nt$. Since $t < N/2$, algorithm D has fewer multiplications than the recursive extension method. For practically useful codes, algorithm D has significantly fewer multiplications than the time domain recursive extension algorithm. Furthermore, time domain recursive extension cannot be implemented using a single array architecture. Two arrays are needed, one for obtaining the error location sequence and the other for recursive extension.

Algorithm D can easily be used for truncated codes with proportionate savings in computing time and storage. It is not possible to achieve such savings in decoding truncated codes using the time domain recursive extension algorithm.

A time domain architecture has recently been presented for decoding Reed-Solomon codes in Reference 4 based on the recursive extension algorithm. For the reasons already given, the algorithm and the architectures described in this paper are more efficient than those described in Reference 4.

4 Architectures

Pipelined and a parallel architectures for implementing the time domain Reed-Solomon decoding algorithms consist of regular and locally connected arrays of identical processing modules. Hence these can be categorised as systolic arrays. The size of the array can be tailored for the required throughput.

4.1 Pipelined architecture

This is based on algorithm D, which has $2t$ iterations. Fig. 3 shows a pipelined array of $2t$ identical processing modules for implementing this algorithm. Each module carries out one iteration of algorithm D. A separate correction block, although shown in Fig. 3, is not actually needed. There are unused multipliers in module $2t$, since auxiliary sequences need not be updated here. This unused circuitry can be used for error correction.

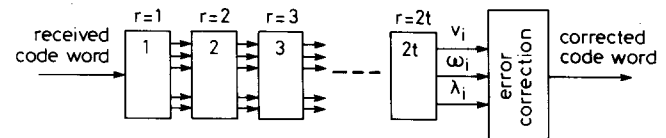


Fig. 3 Pipelined architecture

A functional block diagram of a processing module is given in Fig. 4. All input vectors are entered from the left, sample by sample. Each iteration begins with the evaluation of the discrepancy Δ_r , defined by eqn. 8a. This requires $2N$ multiplications. If two multipliers are used, evaluating Δ_r takes N cycles. Since vector updating follows Δ_r computation, all the input vectors are delayed by N clock periods, as denoted by $D(N)$ in Fig. 4. These delays may be thought of as shift registers, but can be implemented as first-in first-out memories to save silicon area.

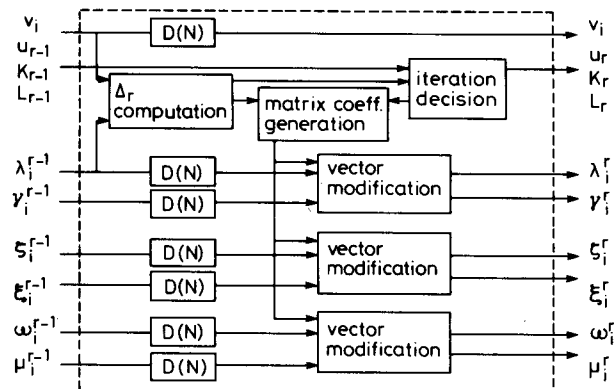


Fig. 4 Functional block diagram of processing module

A circuit diagram of the processing module is given in Fig. 5. The number of multipliers required has been minimised. The circuit for generating the inverse of Δ_r is not

shown. As described in Section 2, the inverse of Δ_r is not required in the iteration in which Δ_r is computed and the inversion can be carried out while the vectors are being updated. Hence there are N clock cycles for obtaining the inverse. This allows the use of simple and low-speed inversion circuit, as follows.

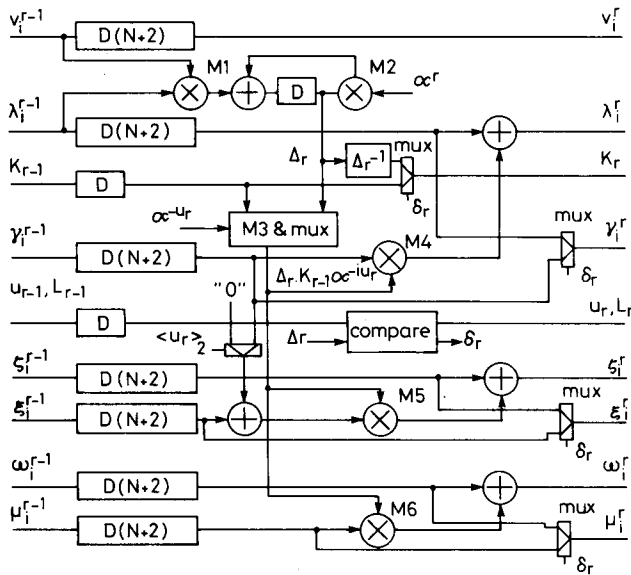


Fig. 5 Detailed design of processing module

Consider two linear feedback shift registers, each wired to multiply by a primitive element of the field in every clock period. Load one shift-register with Δ_r and the other with 1 and keep clocking these until the number in the first register becomes equal to unity. Then the second shift register contains the inverse of Δ_r . This approach is guaranteed to produce an inverse in N clock periods, since the size of the multiplicative group of the Galois field is N .

One drawback of the pipelined architecture is the large storage requirement. Seven delay lines, each of length N , have to be provided for each cell. This storage requirement can be reduced using algorithm E. In this algorithm only five vectors are used in the iterations. Hence the number of delay lines is reduced to five. The parallel architecture also minimises the storage requirement.

4.2 Parallel architecture

The preceding Section considered a pipelined array of $2t$ processing modules, with one module allocated to each iteration. The dual form of this is a fully parallel configuration. Here a processing module is not allocated for each iteration. Instead, all the processing modules together implement each iteration. In the fully parallel configuration there are N cells, as shown in Fig. 6 (detailed of a cell not shown).

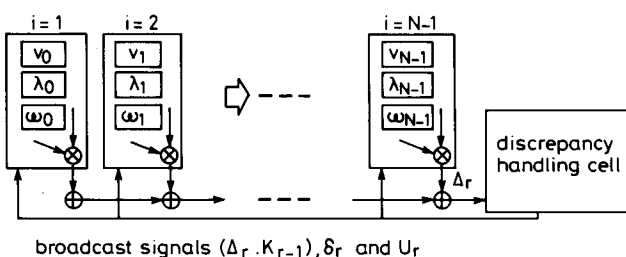


Fig. 6 Parallel architecture

In the architecture of Fig. 6, all N cells implement one iteration of algorithm D at any given time, say one clock cycle. The discrepancy handling cell computes Δ_r^{-1} and updates parameters L_r , K_r , u_r , and δ_r . Then the throughput of the array is $2t$ clock cycles per data block. However, owing to the global XOR operation, the clock period here is greater than that of the pipelined configuration discussed in Section 4.1. The storage requirement of this architecture is significantly less than that of the pipelined architecture. (Note that the pipelined architecture has seven delay lines of length N in each cell.)

In a practical design it is likely that a single multiplier will be used in each cell of Fig. 6 (time multiplexed between all multiplications). All these detailed design decisions, as well as the choice between parallel and pipelined architectures, must be made depending on the parameters (e.g. throughput, block length) specified for the decoder.

4.3 Truncated arrays

The fully pipelined and parallel architectures described may not be practical for many decoding applications. The array sizes need truncating to save hardware, which will result in a proportionate reduction of throughput.

Both architectures can be truncated. For example, in the parallel architecture $N/2$ cells can be used instead of N cells and each cell made to carry out the functions of two cells. This takes twice as much time and hence the throughput is halved. However, note that the storage hardware remains unchanged. It is possible to truncate the parallel array further.

The pipelined architecture too can be truncated. An array consisting of only t cells can be used and the data recycled twice through the array, as shown in Fig. 7. Throughput and hardware are halved. It is possible to truncate the array further.

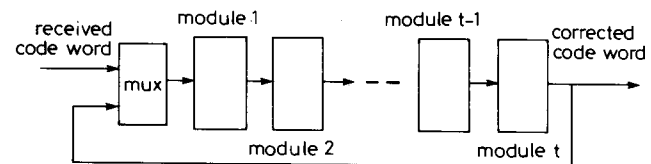


Fig. 7 Pipelined architecture employing t processing cells

In the extreme case, the pipelined architecture can be truncated to a single cell. The vectors of length N have to be cycled through this cell $2t$ times to complete the decoding. Hence throughput of this is $2Nt$ clock cycles per block. The parallel architecture can also be reduced to one cell. As the storage of the parallel architecture remains unchanged when the arithmetic hardware is halved, both the parallel and pipeline architectures converge into one architecture for the single cell case.

5 Conclusions

We have described methods for reducing the computational count in the time domain algorithm for Reed-Solomon decoding which is particularly suited for implementation using VLSI technology. One algorithm (D) has about 60% fewer multiplications than the time domain algorithm given by Blahut in Reference 2.

Since the time domain algorithm involves the repeated application of a single operation, only one cell design is needed. A number of these can be cascaded to meet the required throughput and error-correcting capability. Alternatively, given a fixed array size, it is possible to trade the throughput for error correcting capability or

vice versa. All the registers in the time domain design take the form of delay lines. Data is processed 'on the fly', hence the control complexity of the time domain design is low.

The time domain architecture described by Shayan *et al.* is based on a recursive extension technique. This has more computations than the algorithm given here. Furthermore, it cannot be realised using a single array architecture like the one described in this paper. It is also difficult to adapt it with computational and storage savings for truncated Reed-Solomon codes; whereas the algorithm considered in this paper is ideal for truncated codes.

Even after the optimisation discussed, the time domain algorithm still has two to four times more multiplications than the most efficient frequency domain algorithm, which ratio decreases with increasing t/N . However, it is often impossible to keep all the multipliers of a frequency domain architecture active at all times, since the computations are distributed unevenly between three sequential stages [1]. This problem does not arise with the time domain algorithm since there is only one stage of computation. Hence the ratio of time domain to frequency domain multiplication hardware is closer to two for many practical applications.

For very high speed implementations requiring many multipliers, the number of additional multipliers required by the time domain algorithm is quite large. Furthermore, at high speeds, syndrome computation and Chien-search stages of the frequency domain algorithm can be implemented using fixed-coefficient multipliers which are simpler than the general purpose multipliers required by the time domain algorithm. Hence the time domain algorithm does not seem to be suitable for such applications.

For moderate throughputs, the number of multipliers required by both algorithms are not very high and therefore the additional multiplication hardware required by the time domain algorithm is also not very significant. In such situations, the most appropriate algorithm and architecture should be determined by other factors such as control complexity, smoothness in data control flow, cascability and flexibility. The time domain technique is superior to the frequency domain method in all these aspects. Hence, although the time domain algorithm/architecture is not the most appropriate one for all appli-

cations, there is a reasonable class of applications for which the time domain approach is very suitable.

All the algorithms described have been verified using computer simulations.

6 References

- 1 ARAMBEPOLA, B., and CHOOMCHUAY, S.: 'Algorithm and architecture for Reed-Solomon codes', *GEC J. Res. Inc Marconi Rev.* 1992, 9, No. (3), pp. 172-184
- 2 BLAHUT, R.E.: 'A universal Reed-Solomon decoder', *IBM J. Res. Dev.*, 1984, 28, (2), pp. 150-158
- 3 BERLEKAMP, E.R.: 'Algebraic coding theory' (McGraw-Hill, New York, 1968)
- 4 SHAYAN, Y.R., LE-NGOC, T., and BHARGAVA, V.K.: 'A Versatile time-domain Reed-Solomon decoder', *IEEE Trans.*, 1990, SAC-8, pp. 1535-1542
- 5 BLAHUT, R.E.: 'Theory and practice of error control codes' (Addison-Wesley, Reading, MA, 1983)
- 6 ARAMBEPOLA, B., and CHOOMCHUAY, S.: 'Array architecture for Reed-Solomon decoding'. Proceedings of international symposium on *Circuits & systems*, Singapore, 1991, pp. 2963-2966
- 7 BERLEKAMP, E.R.: 'Bit-serial Reed-Solomon encoder', *IEEE Trans.*, 1982, IT-28, pp. 869-874
- 8 'Telemetry channel coding recommendation'. Consultative Committee for Space Data Systems, 101.0B-2, Issue 2, 1987
- 9 'Telemetry channel coding standard'. ESA PSS-040103, issue 1, 1989

7 Appendix

The sequence $\{\zeta_i\}$ is the N -point inverse DFT of the coefficient sequence of the polynomial $x^{2-t_0}\Lambda'(x)$. Consider the expression for $\Lambda'(x)$ in $GF[2^n]$ given by eqn. 10

$$\Lambda'(x) = \Lambda_1 + \Lambda_3 x^2 + \dots + \Lambda_{N-1} x^{N-2} \quad (10)$$

The coefficient sequence of $\Lambda'(x)$ can be obtained by the pointwise multiplication of the sequence $\{\Lambda_0, \Lambda_1, \dots, \Lambda_{N-1}\}$ with $\{0, 1, 0, 1, \dots\}$, followed by a left shift. The coefficient sequence of $x^{2-t_0}\Lambda'(x)$ is obtained by pointwise multiplying the above two sequences and then right shifting the resulting sequence by $1 - t_0$ places.

The inverse DFT of the sequence $\{0, 1, 0, 1, \dots\}$ is given by the sequence $\{h_i\}$ in algorithm E. The pointwise multiplication maps into a cyclic convolution between $\{\lambda_i\}$ and $\{h_i\}$. The right shift by $1 - t_0$ places maps into multiplications by $\alpha^{-i(t_0-1)}$ in the time domain as given by eqn. 9c. This concludes the proof.