Algorithm-Based Low-Power/High-Speed Reed–Solomon Decoder Design

Arun Raghupathy, *Member, IEEE,* and K. J. R. Liu, *Senior Member, IEEE*

***Abstract*—With the spread of Reed–Solomon (RS) codes to portable wireless applications, low-power RS decoder design has become important. This paper discusses how the Berlekamp Massey Decoding algorithm can be modified and mapped to obtain a low-power architecture. In addition, architecture level modifications that speed-up the syndrome and error computations are proposed. Then the VLSI architecture and design of the proposed low-power/high-speed decoder is presented. The proposed design is compared with a normal design that does not use these algorithm/architecture modifications. The power reduction when compared to the normal design is estimated. The results indicate a power reduction of about 40% or a speed-up of 1.34.**

***Index Terms*—Berlekamp Massey algorithm, Channel coding, decoding, error-correction coding, Forney's method, high-speed integrated circuits, low power systems, parallel algorithms, parallel architectures, Reed–Solomon codes, very-large-scale integration, .**

I. Introduction

ERROR-CONTROL codes are used widely in communication systems to combat channel noise. These codes protect data from errors by introducing redundancy selectively in the transmitted data. Error-control codes are also used in storage systems to protect the data from errors that are introduced during the process of reading the data.

Error-control codes can be classified into convolutional and block codes. Reed–Solomon (RS) codes are linear block codes. Primitive RS codes are also cyclic. RS codes belong to the class of nonbinary Bose–Chaudhuri–Hocquenheim (BCH) codes. RS codes are among the most widely used block codes because they are capable of correcting burst errors as well as random errors. In addition, efficient decoding algorithms have been developed for RS codes.

A concatenated coding scheme consisting of a convolutional inner code (i.e., applied last, removed first) and an RS outer code (i.e., applied first, removed last) has been accepted as a standard for space communications [1]. In audio compact discs [2], a pair of cross-interleaved RS codes are used to protect against errors that occur due to imperfections in the read process. RS codes are used in the U.S Cellular Digital Packet Data (CDPD) service [3] to protect user data. RS codes are being considered for use in xDSL (Digital Subscriber Line) services to protect the data from impulse noise [4]. RS codes are good candidates for use in wireless communication systems as a part of a concatenated coding system, along with convolutional codes [5], [6].

An (n, k) primitive RS code defined in the Galois field $\mathrm{GF}(q = 2^m)$ has code words of length $n = 2^m - 1$, where m is a positive integer and k is the number of information symbols in the codeword. This RS code has minimum distance $d_{\min} = n - k + 1$ and has $n - k$ redundant symbols. The generator polynomial $g(x)$ of the code is $g(x) = \prod_{i=0}^{n-k-1}(x - \alpha^{b+i})$, where α is a primitive element in $\mathrm{GF}(2^m)$ and b is an integer constant. This implies that these $n - k$ consecutive powers of α are roots of every codeword polynomial. This property has been used to develop many efficient decoding algorithms for RS decoding.

Let $e = [e_0 \; e_1 \; \cdots \; e_{n-1}]$ denote the error vector. Note that $e_i = 0$ if no error occured at position i. Also, $e_i \neq 0$ denotes the actual value of the error introduced by the channel at position i. Assume further that ν errors have occured at positions $l = i_1, i_2, \ldots, i_\nu$. The symbols v_i of the possibly corrupted word received from the channel can be written in terms of the codeword symbols c_i and the error symbols e_i as $v_i = c_i + e_i$ for $i = 0, 1, \ldots, (n-1)$. Then the decoding problem is to find the error values e_l and the error locations $l = i_1, i_2, \ldots, i_\nu$. The received polynomial $v(x)$ can be formed from the received symbols as $v(x) = v_0 + v_1 x \cdots + v_{n-1}x^{n-1}$. The error locator polynomial $\Lambda(x)$ is defined as $\Lambda(x) = \prod_{l=1}^{\nu}(1 - x\alpha^{i_l})$. The syndromes are defined as $S_j = \sum_{i=0}^{n-1} \alpha^{i(b+j)} v_i = v(\alpha^{b+j})$ for $j = 0, 1, \ldots, (2t-1)$ and written in polynomial form as $S(x) = \sum_{i=0}^{2t-1} S_j x^j$. The *key equation* can then be written as

$$\Lambda(x)S(x) = \Gamma(x) \bmod x^{2t} \tag{1}$$

where $\Gamma(x)$ is the error magnitude polynomial with $\deg(\Gamma(x)) < t$. The solution of this equation plays a pivotal role in the decoding process.

Below, we briefly review and summarize the various RS decoding algorithms. The RS decoding algorithms can be classified as shown in Fig. 1. On the left side of Fig. 1, all the algorithms that involve syndrome computation are shown. We will discuss these algorithms first. Once the syndrome has been computed, the next step is to solve the key equation to obtain the error locator polynomial. Peterson [7] proposed a solution to this problem which was later improved and extended by Gorenstein and Zierler [8]. Their approach was to write a set of equations involving the unknown error locator polynomial coefficients and the known syndromes, and then, solve the system of equations for the

Manuscript received October 1998; revised July 2000. This work is supported in part by the NSF NYI Award MIP9457397 and the ONR Grant N00014-93-10566. This paper was recommended by Associate Editor Y. Leblebici.

A. Raghupathy was with the Electrical Engineering Department and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA. He is now with Qualcomm, Inc., San Diego, CA 92121 USA.

K. J. R. Liu is with the Electrical Engineering Department and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA.

Publisher Item Identifier S 1057-7130(00)09921-3.

1057–7130/00$10.00 © 2000 IEEE

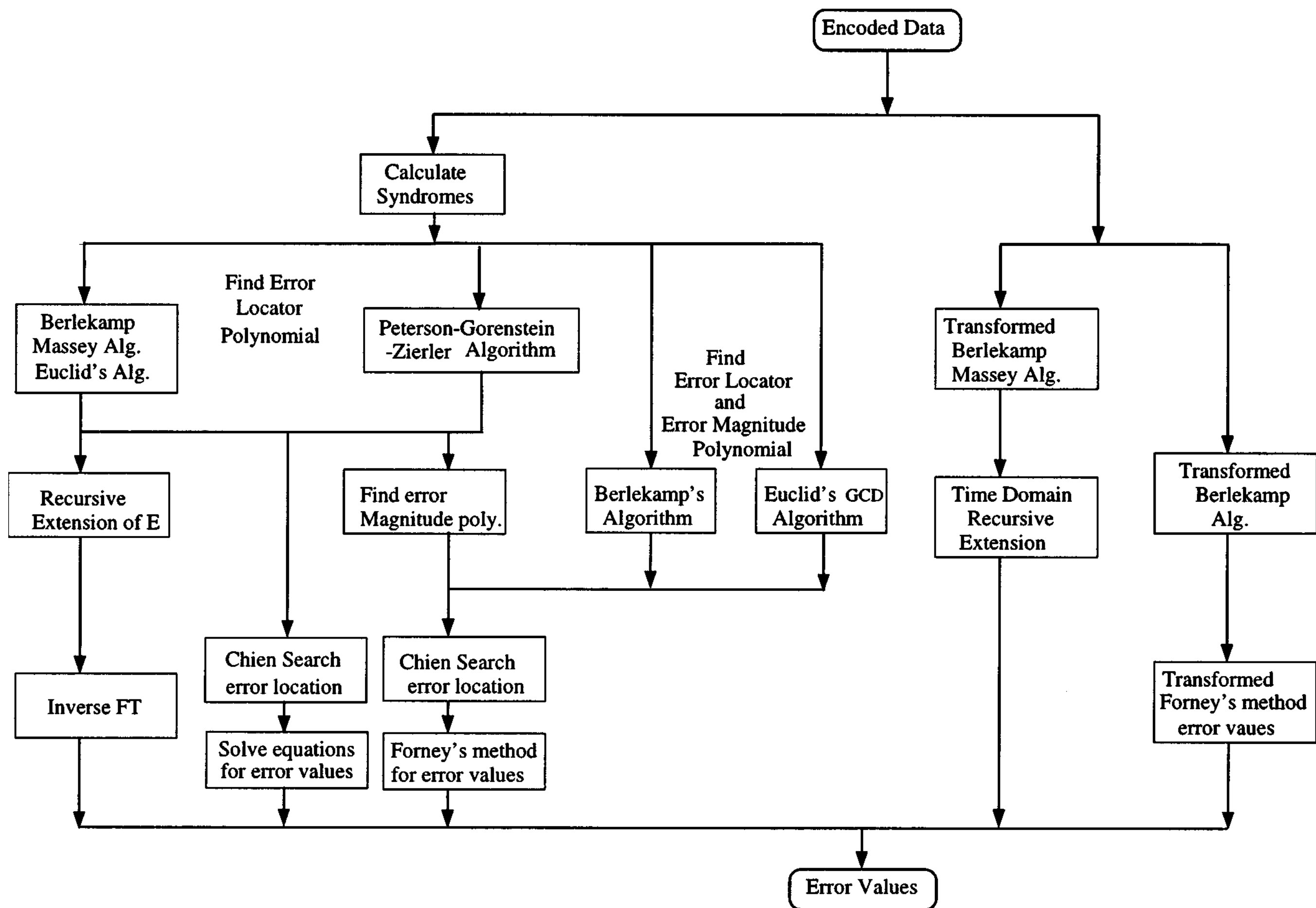

Fig. 1. Overview of RS decoding techniques.

error locator using direct matrix inversion techniques. These matrix inversions become computationally inefficient for large ν. It can be mentioned that for small ν (for example in compact disc systems), closed form expressions for the error locations and error values in terms of the syndromes can be obtained. An efficient technique for finding the error locator was first proposed by Berlekamp [9]. Massey [10] interpreted this algorithm in terms of linear feedback shift register (LFSR) synthesis (this algorithm is referred to in the literature as the Berlekamp Massey algorithm). Sugiyama [11], [12] recognized that the key equation could also be solved by applying Euclid's greatest common divisor (GCD) algorithm to obtain the error locator polynomial efficiently. Once the error locator polynomial has been obtained, the error values can be computed in the time or frequency domains. In the frequency domain, the error transform is found by recursively extending [13], [12] the syndromes and the error vector is then found by applying an inverse Fourier transform (FT). In the time domain, the roots of the error-locator polynomial are found by a Chien search [14]. Then, the error values may be found either by solving the linear set of equations directly or by using a technique called Forney's method [15]. Forney's method also requires computation of $\Gamma(x)$. Berlekamp's algorithm can be used to compute $\Gamma(x)$ by using a parallel set of iterations. $\Gamma(x)$ can also be obtained from Euclid's algorithm by maintaining some additional information [12] during the iterative process. Alternately, $\Gamma(x)$ can be directly computed using the key equation after $\Lambda(x)$ has been computed.

Decoder implementations that use the syndrome are generally based on one of the above algorithms. Liu [16] proposed a RS decoder design that used Massey's shift register synthesis, followed by recursive extension of the error transform and, finally, an inverse FT to get the error vector. Shao *et al.* [17] and Demassieux *et al.* [18] based their decoder designs on Euclid's algorithm to find the error locator polynomial. Then the error transform was recursively extended and, finally, an inverse FT was performed to get the error values. Shao and Reed [19], Tong [20], Whitaker *et al.* [1], and Berlekamp *et al.* [21, Ch. 10] used Euclid's algorithm in their decoders to find the error locator and the error magnitude polynomials. Then, Chien search was used in conjunction with Forney's algorithm to calculate the error values. The implementation of Euclid's GCD algorithm was generally based on the systolic array proposed by Brent and Kung [22]. The computational complexity of the syndrome based approach that uses either the Berlekamp algorithm or Euclid's method followed by the Chien search is $O(nt)$.

Another approach [23], [12] shown on the right side of Fig. 1, avoids the computation of the syndrome. Here, the algorithm (either the Berlekamp–Massey or the Berlekamp algorithm) is transformed so that all its variables are in the time domain. This is achieved by taking the Inverse FT of all sequences in

the original algorithm. Hence, this algorithm is sometimes referred to as transform decoding without transforms. Shayan *et al.* [24] designed a versatile decoder based on this technique that can decode any RS code over $\mathrm{GF}(2^5)$. The decoder was based on a transformed version of the Berlekamp Massey algorithm and time domain recursive extension. The disadvantage is that the computational complexity is $O(n^2)$. Therefore, this technique can be used only for small blocklengths. Choomchuay and Arambepola [25] proposed an architecture based on a reorganized form of the computation in [23] that reduces the computational complexity. However, the storage requirements are $O(n)$. Hsu and Wang [26] proposed some modifications to [25] that reduced the storage requirements to $O(t^2)$. While the order of the computational complexity in [25] and [26] is reduced to $O(nt)$, the actual complexity remains larger than for the syndrome-based Berlekamp algorithm.

In Section II, we summarize our approach to low-power/high-speed RS decoding. In Section III, we discuss how the Berlekamp algorithm can be modified to enable a low-power VLSI architecture/design. We first discuss modifications to the errors-only Berlekamp algorithm in Section III-A. We later extend these modifications to the errors-and-erasures Berlekamp algorithm in Section III-B. In Section IV, we discuss the VLSI architecture of an RS decoder including, specifically, architecture level techniques for the syndrome and error computations. In addition, we compare architecture level power estimates for the normal and modified designs. We also specify under what conditions power reduction can be expected. In Section V, we discuss the VLSI chip design of the normal and modified decoders. We also discuss the synthesis and layout results which show that the power consumption can be reduced by 40%. Finally, in Section VI, we present concluding remarks.

II. The Algorithm/Architecture-Based Approach

Until recently, the two key parameters in VLSI design were area and speed. Power consumption was a consideration only in order to reduce packaging and cooling costs for the chip. With the proliferation of portable devices, power consumption has become a primary design parameter. This is because power consumption determines the battery lifetime in portable systems. Algorithm/architecture-level transformations can potentially have the greatest impact. This is because, at the algorithm level, maximum design flexibility is available. In addition, device- and circuit-level techniques can be applied independent of these algorithm/architecture-level techniques to obtain further power savings. We therefore focus on algorithm/architecture-level techniques to obtain low-power RS decoding.

Various algorithm/architecture-level transformations [27]–[31] have been proposed in the literature. The transformations proposed include algebraic transformations [28] (such as associativity, distributivity, etc.), loop-unrolling/look ahead transformations [32] that try to break the recursive loops, retiming [27], folding/unfolding [33] transformations, and strength reduction [34]. These transformations can be used to obtain area-efficient high-speed or low-power designs [31], [35]–[37], depending on the design goal. One of the approaches to reduce the power consumption is to use these transformations to expose parallelism and enable pipelining in the computation.

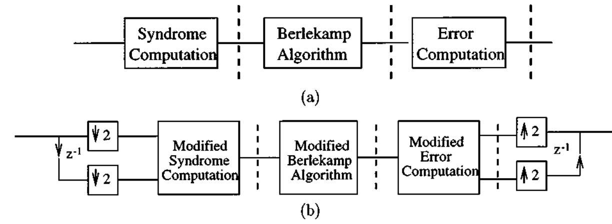

Fig. 2. Block diagram for (a) normal decoder and (b) modified decoder.

Over the last decade, we have seen continued scaling down of device feature size. This has improved the performance of VLSI designs in terms of speed. Also, algorithms of increased complexity can be implemented on a single chip. In addition, the area required to implement a given algorithm has shrunk dramatically because of the increased transistor density. It has been suggested in [30] that low-power operation can be obtained by modifying the algorithm to enable a parallel implementation. In other words, we utilize the increase in available transistor count cleverly to obtain power reduction. The idea can be explained briefly as follows. The power dissipation in a well-designed digital CMOS circuit can be modeled as [38]

$$P \approx \alpha \cdot C_{\text{eff}} \cdot V_{dd}^2 \cdot f_{\text{clk}} \tag{2}$$

where

- α average probability that the total node capacitance is switched (also referred to as the activity factor);
- C_{eff} effective load capacitance;
- V_{dd} supply voltage;
- f_{clk} operating clock frequency.

Similarly, the delay of the CMOS device can be approximated [30] as

$$T_D \approx \frac{C_L \times V_{\text{dd}}}{I} = \frac{C_L \times V_{\text{dd}}}{\epsilon (V_{\text{dd}} - V_t)^2} \tag{3}$$

where

- C_L capacitance along the critical path;
- ϵ a device parameter;
- V_t threshold voltage of the transistor.

Fig. 2 shows the normal and modified designs. For the moment, let us assume that such a formulation is possible. We will show later how transformations can be applied to obtain such a formulation. The modified design can operate at a slower speed while maintaining throughput. If, for example, the critical path delays for the normal and modified designs are T and T', respectively, at a supply voltage of V. Then, the supply voltage of the modified design can be reduced to V' to reduce power (where V' is chosen such that $T'' = 2T$). The ratio of power consumption in the modified design P_M to the normal design P_N can be written based on (2) as

$$\frac{P_M}{P_N} = \frac{C_M}{C_N} \left(\frac{V'}{V}\right)^2 \frac{f/2}{f} \tag{4}$$

where C_M and C_N are the effective capacitances of the normal and modified designs.

In this paper, we consider an RS decoding algorithm [39] that starts with the computation of the syndrome. Then, the error locator and error magnitude polynomials are computed using Berlekamp's algorithm. Finally, the error locations are found using Chien's search, and the error values are computed using Forney's algorithm. The syndrome computation involves evaluating the received polynomial $v(x)$ at $x = \alpha^{b+j}; j = 0, 1, \ldots, n-k-1$. The Chien search involves locating the errors by evaluating $\Lambda(x)$ at $x = \alpha^{-i}; 0 \leq i \leq n-1$ and comparing the result with zero. The Forney's algorithm computes the error values by evaluating $\Lambda(x)$ and $\Gamma(x)$ at the error locations. Architecture-level techniques can be used to create additional parallelism in the syndrome and error-value computations as we will show later. The computation of the error locator and error magnitude polynomial is more involved and creating additional parallelism is more difficult. We will discuss algorithm level techniques that modify the Berlekamp algorithm to achieve low-power operation. It should be pointed out that the Berlekamp algorithm and Euclid's algorithm have the same computational complexity. We choose to use the Berlekamp algorithm because it possesses some properties that enable the application of a look-ahead [32] like transformation.

III. Low-Power Berlekamp Algorithm

As mentioned briefly in the previous section, the error-correction problem involves finding the location of the errors and their corresponding values. In particular, we can correct $\lfloor d_{\min} - 1/2 \rfloor$ errors using a code that has a minimum distance $d_{\min}$. The location of erasures are known to the decoder. Correcting erasures, therefore, involves only finding the erasure values. In general, any pattern of ρ erasures and ν errors can be corrected provided $2\nu + \rho <= d_{\min} - 1$. The Berlekamp algorithm can be used to solve both problems.

We first consider techniques to modify the Berlekamp algorithm for low-power operation in the context of error correction, in Section III-A. Then, we extend our approach to solve the errors-and-erasures correction problem in Section III-B. We assume that the number of redundant symbols is given by $n - k = 2t$ so that we can correct up to t error patterns.

A. Errors-Only Decoding

The original Berlekamp algorithm is reproduced here. When expressed in this form, we can observe some of its properties that will aid in the development of our algorithm modifications. Note that the Berlekamp algorithm (Algorithm 1, described below) computes both the error locator polynomial $\Lambda(x)$ and the error magnitude polynomial $\Gamma(x)$ in parallel (see steps 5a and 5b of Algorithm 1).

Berlekamp Algorithm: Algorithm 1

0. Initialize: $\Lambda^{(0)}(x) = 1, B^{(0)}(x) = 1, \Gamma^{(0)}(x) = 0, A^{(0)}(x) = x^{-1}, L_0 = 0$
1. **for** $r = 1$ **to** $2t$
2. $\Delta_r = \sum_{j=0}^{L_{r-1}} \Lambda_j^{(r-1)} S_{r-1-j}$
3. **if** $\Delta_r \neq 0$ **then** $b1 = 0$ **else** $b1 = 1$
4. **if** $2L_{r-1} <= (r-1)$ **then** $b2 = 0$ **else** $b2 = 1$

TABLE I
Updating $\Lambda(x), B(x)$ and L for the BM Algorithm

Condition	$M_r(x)$	$f_r(L_{r-1})$
$\overline{b1}\ \overline{b2}$	$\begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1} & 0 \end{bmatrix}$	$r - L_{r-1}$
$b1 + b2$	$\begin{bmatrix} 1 & -\Delta_r x \\ 0 & x \end{bmatrix}$	L_{r-1}

5a.
$$\begin{bmatrix} \Lambda^{(r)}(x) \\ B^{(r)}(x) \end{bmatrix} = M_r(x) \begin{bmatrix} \Lambda^{(r-1)}(x) \\ B^{(r-1)}(x) \end{bmatrix} \text{ (refer to Table I)}$$

5b.
$$\begin{bmatrix} \Gamma^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} = M_r(x) \begin{bmatrix} \Gamma^{(r-1)}(x) \\ A^{(r-1)}(x) \end{bmatrix} \text{ (refer to Table I)}$$

6. $L_r = f_r(L_{r-1})$ (refer to Table I)
7. **end for**
8. Output: $\Lambda(x) = \Lambda^{(2t)}(x), \Gamma(x) = \Gamma^{(2t)}(x)$.

We want to look at updating $(\Lambda^{(2k-2)}(x), B^{(2k-2)}(x))$ to $(\Lambda^{(2k)}(x), B^{(2k)}(x))$ without going through $(\Lambda^{(2k-1)}(x), B^{(2k-1)}(x))$. In this way, we want to halve the number of iterations. Of course, this modified iteration will be more complicated than a single iteration of the original algorithm. In general, the modified iteration takes time $T' \geq T$. In order to get an improved algorithm in terms of speed/power, it is enough if $T' \leq 2T$. In particular, the modified algorithm must expose additional parallelism so that the above holds. Ideally, we would like to have T' as close to T as possible in order to maximize the improvement in speed or power. We observe that the update matrix for $(\Gamma^{(r)}(x), A^{(r)}(x))$ is the same as that for $(\Lambda^{(r)}(x), B^{(r)}(x))$ (see Steps 5a and 5b in Algorithm 1). This implies that any transformation that we derive for $(\Lambda(x), B(x))$ will apply to the polynomial pair $(\Gamma(x), A(x))$. We note therefore that the critical variables to consider when modifying the Berlekamp algorithm are $\Lambda(x)$, $B(x)$, and L.

We make several observations about the variables involved in the Berlekamp algorithm that will enable the transformation to be performed efficiently. Note that $L_r > L_{r-1}$ only if $\Delta_r \neq 0$ and $2L_{r-1} \leq r-1$. This implies that L_r can increase only once in any two iterations. We can prove this property by contradiction. Let us assume that $L_{r+1} > L_r > L_{r-1}$. This implies that $\Delta_r \neq 0, 2L_{r-1} \leq r-1, L_r = r - L_{r-1}$. Also, $\Delta_{r+1} \neq 0, 2L_r \leq (r+1) - 1$. Using $L_r = r - L_{r-1}$ in $2L_r \leq (r+1) - 1$, we get $2L_{r-1} > r-1$. Since we cannot have $2L_{r-1} > r-1$ and $2L_{r-1} \leq r-1$, we have obtained a contradiction that completes the proof. Also, observe that $B^{(r)}(x)$ is updated only when L_r increases. Otherwise, $B^{(r)}(x)$ is just shifted. Note that the pair $(\Lambda^{(r)}(x), L_r)$ defines a linear feedback shift register [12] of minimum size that generates the sequence $S_0, S_1, \ldots, S_{r-1}$. In other words

$$S_i = -\sum_{j=1}^{L_r} \Lambda_j^{(r)} S_{i-j} \tag{5}$$

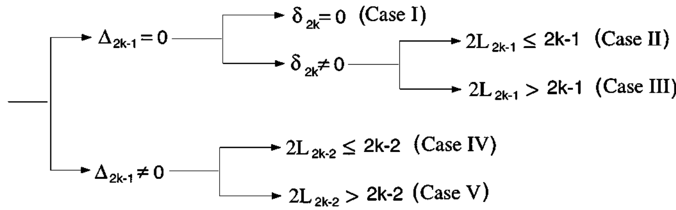

Fig. 3. Different updating cases when odd and even iterations are considered together.

for $i = L_r, L_r+1, \ldots, r-1$.

Let Δ_r be the discrepancy at iteration r. In what follows, we refer to $r = 2k-1$ as the odd iteration and $r = 2k$ as the even iteration. We want to design the modified algorithm so that the critical variables at the end of the even iteration of the original algorithm match the variables after the kth iteration of the modified algorithm. Define a new variable $\delta_{2k} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-1-j}$ as the discrepancy predicted for the even iteration. Note from Algorithm 1 that $\delta_{2k} = \Delta_{2k}$ if $\Delta_{2k-1} = 0$ (since, in this case $\Lambda^{(2k-2)}(x) = \Lambda^{(2k-1)}(x)$). We will derive the modified algorithm by starting with $(\Lambda^{(2k-2)}(x), B^{(2k-2)}(x))$ and L_{2k-2} and considering the effect of two consecutive iterations of the original algorithm.

Various cases that need to be considered are shown in Fig. 3. For cases I, II, and III, since $\Delta_{2k-1} = 0$, we have for the odd iteration, $\Lambda^{(2k-2)}(x) = \Lambda^{(2k-1)}(x)$, $L_{2k-1} = L_{2k-2}$ and $B^{(2k-1)}(x) = xB^{(2k-2)}(x)$. This implies that $\delta_{2k} = \Delta_{2k}$. For case I, since $\Delta_{2k} = \delta_{2k} = 0$, we also get $\Lambda^{(2k)}(x) = \Lambda^{(2k-1)}(x), B^{(2k)}(x) = xB^{(2k-1)}(x)$ and $L_{2k} = L_{2k-1}$ for the even iteration.

For case II, we have $\delta_{2k} \neq 0$ ($\Delta_{2k} = \delta_{2k}$) and $2L_{2k-1} \leq 2k-1$. This implies that $\Lambda^{(2k)}(x) = \Lambda^{(2k-1)}(x) - \Delta_{2k} x B^{(2k-1)}(x) = \Lambda^{(2k-2)}(x) - \delta_{2k} x^2 B^{(2k-2)}(x)$, $B^{(2k)}(x) = \Delta_{2k}^{-1} \Lambda^{(2k-1)}(x) = \Delta_{2k}^{-1} \Lambda^{(2k-2)}(x)$ and $L_{2k} = 2k - L_{2k-1} = 2k - L_{2k-2}$.

For case III, we have $\delta_{2k} \neq 0$ ($\Delta_{2k} = \delta_{2k}$) and $2L_{2k-1} > 2k-1$ so that $\Lambda^{(2k)}(x) = \Lambda^{(2k-1)}(x) - \Delta_{2k} x B^{(2k-1)}(x) = \Lambda^{(2k-2)}(x) - \delta_{2k} x^2 B^{(2k-2)}(x)$, $B^{(2k)}(x) = xB^{(2k-1)}(x) = x^2 B^{(2k-2)}(x)$ and $L_{2k} = L_{2k-1} = L_{2k-2}$. Therefore, the final updates for cases I, II and III can be written as shown in Rows 1, 2, and 3, respectively, of Table II.

For cases IV and V, $\Delta_{2k} \neq 0$ so that $\delta_{2k} \neq \Delta_{2k}$. For both these cases, in the odd iteration we get $\Lambda^{(2k-1)}(x) = \Lambda^{(2k-2)}(x) - \Delta_{2k-1} x B^{(2k-2)}(x)$. For case IV, since $2L_{2k-2} > 2k-2$, for the odd iteration we get $B^{(2k-1)}(x) = \Delta_{2k-1}^{-1} \Lambda^{(2k-2)}(x)$ and $L_{2k-1} = 2k-1-L_{2k-2}$. On the other hand, for case V, for the odd iteration we get $B^{(2k-1)}(x) = xB^{(2k-2)}(x)$ and $L_{2k-1} = L_{2k-2}$.

We will use the idea of Massey's synthesis (as explained in [12]) to find an update from $\Lambda^{(2k-1)}(x)$ to $\Lambda^{(2k)}(x)$ for cases IV and V using the syndromes and variables that can be computed at $r = 2k-2$. We first consider case IV. In this case, it is impossible for $2L_{2k-1} \leq 2k-1$ (as explained before). Therefore, $B^{(2k)}(x) = xB^{(2k-1)}(x) = \Delta_{2k-1}^{-1} x \Lambda^{(2k-2)}(x)$ and $L_{2k} = L_{2k-1} = 2k-1-L_{2k-2}$. The update equation for $\Lambda^{(2k-1)}(x)$ can be written as $\Lambda^{(2k)}(x) = \Lambda^{(2k-1)}(x) - \Delta_{2k} x B^{(2k-1)}(x)$. The problem with this update is that in the modified algorithm there is no direct way to compute $\Delta_{2k} = \sum_{j=0}^{L_{2k-1}} \Lambda_j^{(2k-1)} S_{2k-1-j}$ (since we want to avoid computing $\Lambda^{(2k-1)}(x)$). We approach the problem by writing the update in terms of $\Lambda^{(2k-2)}(x)$ and $B^{(2k-2)}(x)$ as

TABLE II
UPDATING $\Lambda(x), B(x)$ AND L FOR THE BM ALGORITHM WHERE $G(x) = 1 - \Delta_{2k-1}^{-1}\beta_{2k-1}x$ AND $G1(x) = -\beta_{2k-1}x^2 - \Delta_{2k-1}x$

Condition	$M_k(x)$	$f_k(L_{2k-2})$	$g_k(\alpha_{2k-2})$
$b1\ b0$	$\begin{bmatrix} 1 & 0 \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}
$b1\ \overline{b0}\ \overline{b2}$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ \delta_{2k}^{-1} & 0 \end{bmatrix}$	$2k - L_{2k-2}$	$\delta_{2k}^{-1}\eta_{2k}$
$b1\ \overline{b0}\ b2$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}
$\overline{b1}\ \overline{b2}$	$\begin{bmatrix} G(x) & -\Delta_{2k-1}x \\ x\Delta_{2k-1}^{-1} & 0 \end{bmatrix}$	$2k-1-L_{2k-2}$	$\Delta_{2k-1}^{-1}\delta_{2k}$
$\overline{b1}\ b2$	$\begin{bmatrix} 1 & G1(x) \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}

$$\begin{aligned}\Lambda^{(2k)}(x) &= \Lambda^{(2k-1)}(x) - \Delta_{2k} x B^{(2k-1)}(x) \\ &= \left(\Lambda^{(2k-2)}(x) - \Delta_{2k-1} x B^{(2k-2)}(x)\right) \\ &\quad + U_1 x \Lambda^{(2k-2)}(x) \end{aligned} \tag{6}$$

where $U_1 = -\Delta_{2k}\Delta_{2k-1}^{-1}$ in an unknown. We know from the original algorithm that $B^{(2k-2)}(x)$ can be written as $B^{(2k-2)}(x) = \Delta_m^{-1} x^{2k-2-m} \Lambda^{(m-1)}(x)$, where $m < 2k-1$ is the last iteration for which $L_m > L_{m-1}$. We will use the property that $(\Lambda^{(2k)}(x), L_{2k})$ must represent the minimum length shift register for the sequence $S_0, S_1, \ldots, S_{2k-1}$. This implies that $\sum_{j=0}^{L_{2k}} \Lambda_j^{(2k)} S_{2k-1-j} = 0$. Based on this identity, we can solve for U_1 as follows:

$$\begin{aligned}&\sum_{j=0}^{L_{2k}} \Lambda_j^{(2k)} S_{2k-1-j} \\ &\quad = \sum_{j=0}^{L_{2k}} \Lambda_j^{(2k-2)} S_{2k-1-j} - \Delta_{2k-1}\Delta_m^{-1} \sum_{j=0}^{L_{2k}} \Lambda_j^{(m-1)} S_{m-j} \\ &\qquad + U_1 \sum_{j=0}^{L_{2k}} \Lambda_j^{(2k-2)} S_{2k-2-j} \\ &\quad = \delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m + U_1\Delta_{2k-1} = 0 \end{aligned} \tag{7}$$

where $\gamma_m = \sum_{j=0}^{L_m} \Lambda_j^{(m-1)} S_{m-j}$. This implies $U_1 = -\Delta_{2k-1}^{-1}(\delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m)$. The update for $\Lambda(x)$ can be written as $\Lambda^{(2k)}(x) = \Lambda^{(2k-2)}(x) - \Delta_{2k-1} x B^{(2k-2)}(x) - \Delta_{2k-1}^{-1}(\delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m) x \Lambda^{(2k-2)}(x)$. Note that we need to maintain the variable $\Delta_m^{-1}\gamma_m$ from one iteration to the other in the modified algorithm. Observe that the term $(\Delta_m^{-1}\gamma_m)$ needs to be updated only at iteration m' for $L_{m'} > L_{m'-1}$. We will discuss the details of this update after we discuss case V.

For case V, we have $2L_{2k-2} > 2k-2 \Rightarrow L_{2k-1} = L_{2k-2} \Rightarrow 2L_{2k-1} > 2k-1$ (the last inequality follows because both $2L_{2k-2}$ and $2k-2$ are even). This means that $B^{(2k)} = xB^{(2k-1)} = x^2 B^{(2k-2)}$ and $L_{2k} = L_{2k-2}$. To obtain the update equation for $\Lambda^{(2k)}(x)$, we proceed as in

case IV. The update equation for $\Lambda^{(2k-1)}(x)$ can be written as $\Lambda^{(2k)}(x) = \Lambda^{(2k-1)}(x) - \Delta_{2k} x B^{(2k-1)}(x)$. We can rewrite this equation as $\Lambda^{(2k)}(x) = (\Lambda^{(2k-2)}(x) - \Delta_{2k-1} x B^{(2k-2)}(x)) + U_2 x^2 B^{(2k-2)}(x)$, where $U_2 = -\Delta_{2k}\Delta_{2k-1}^{-1}$ in an unknown. Again, we use $B^{(2k-2)}(x) = \Delta_m^{-1} x^{2k-2-m} \Lambda^{(m-1)}(x)$ along with the requirement that $\sum_{j=0}^{L_{2k}} \Lambda_j^{(2k)} S_{2k-1-j} = 0$ to obtain

$$\begin{aligned}\sum_{j=0}^{L_{2k}} \Lambda_j^{(2k)} S_{2k-1-j} &= \sum_{j=0}^{L_{2k}} \Lambda_j^{(2k-2)} S_{2k-1-j} \\ &\quad - \Delta_{2k-1}\Delta_m^{-1} \sum_{j=0}^{L_{2k}} \Lambda_j^{(m-1)} S_{m-j} \\ &\quad + U_2 \Delta_m^{-1} \sum_{j=0}^{L_{2k}} \Lambda_j^{(m-1)} S_{m-1-j} \\ &= \delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m + U_2\Delta_m^{-1}\Delta_m = 0. \quad (8)\end{aligned}$$

This implies $U_2 = -(\delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m)$. The update for $\Lambda(x)$ can be written as $\Lambda^{(2k)}(x) = \Lambda^{(2k-2)}(x) - \Delta_{2k-1} x B^{(2k-2)}(x) - (\delta_{2k} - \Delta_{2k-1}\Delta_m^{-1}\gamma_m) x^2 B^{(2k-2)}(x)$. Again, we observe that the term $(\Delta_m^{-1}\gamma_m)$ appears. We define a variable α_{2k} that holds at the end of iteration k the value $\Delta_m^{-1}\gamma_m$, where m is the last iteration of the original algorithm at which $L_m > L_{m-1}$. Therefore, α_{2k} needs to be updated at iteration k if L changes in either the even or odd iteration (i.e., in cases II and IV). In case IV, L changes in the odd iteration. This means that $L_{2k} = 2k - 1 - L_{2k-2}$ and $m = 2k - 1$. The new value of Δ_m can be written as $\sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-2)} S_{2k-2-j} = \Delta_{2k-1}$. Similarly, the new value of γ_m can be written as $\sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-2)} S_{2k-1-j} = \delta_{2k}$. In case II, L changes in the even iteration. This means that $m = 2k$. The new value of Δ_m can be written as $\sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-1)} S_{2k-1-j} = \sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-2)} S_{2k-1-j} = \delta_{2k-1}$. Similarly, the new value of γ_m can be written as $\sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-1)} S_{2k-j} = \sum_{j=0}^{L_{2k-2}} \Lambda_j^{(2k-2)} S_{2k-j} = \eta_{2k}$. Note that we need to compute η_{2k} (which can be interpreted as a 2-step predictor for the discrepancy) at each iteration of the modified algorithm.

We also note that in order to differentiate between the five cases as shown in Fig. 3, we do not need to test both L_{2k-2} and L_{2k-1} (i.e., we do not need to check both $2L_{2k-2} \le 2k-2$ and $2L_{2k-1} \le 2k-1$). In cases I, II and III, $L_{2k-1} = L_{2k-2}$ and this makes the two tests equivalent. This is because $2L_{2k-1} \le 2k-1 \Rightarrow 2L_{2k-2} \le 2k-1 \Rightarrow 2L_{2k-2} < 2k-1 \Rightarrow 2L_{2k-2} \le 2k-2$ (since $2L_{2k-2}$ is even, $2L_{2k-2}$ cannot equal $2k-1$). Also, $2L_{2k-2} \le 2k-2 \Rightarrow 2L_{2k-2} = 2L_{2k-1} \le 2k-2 \Rightarrow 2L_{2k-1} < 2k-1$. In cases IV and V, we need to test only $2L_{2k-2} \le 2k-2$. This suggests that we can do away with the test $2L_{2k-1} \le 2k-1$ and perform only the test $2L_{2k-2} \le 2k-2$.

All of the above is summarized in Algorithm 2 and Table II. Notice that the number of iterations has been reduced to t. We will exploit this fact to obtain a high-speed/low-power implementation.

The only difference between the Berlekamp Massey synthesis and the Berlekamp algorithm is that the former computes only $\Lambda(x)$, while the latter computes $\Lambda(x)$ and $\Gamma(x)$ in parallel (see steps 5a and 5b of Algorithm 1).

Up to this point, we have discussed the computation of $\Lambda(x)$ without referring to $\Gamma(x)$. In Step 5b, $(\Gamma(x), A(x))$ are updated with the same matrix $M_r(x)$ (see step 5b of Algorithm 1). Therefore, we can apply the modifications discussed above to step 5b of Algorithm 1 as well to obtain step 9b of Algorithm 2 (described below).

The modified Berlekamp algorithm can be compactly written as follows:

Low-Power Berlekamp Algorithm: Algorithm 2

0. Initialize $\Lambda^{(0)}(x) = 1, B^{(0)}(x) = 1, \Gamma^{(0)}(x) = 0, A^{(0)}(x) = x^{-1}$, $L_0 = 0$, $\alpha_0 = S_0$
1. **for** $k = 1$ **to** t
2. $\Delta_{2k-1} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-2-j}$
3. $\delta_{2k} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-1-j}$
4. $\eta_{2k} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-j}$
5. $\beta_{2k-1} = \delta_{2k} - \Delta_{2k-1}\alpha_{2k-2}$
6. **if** $\delta_{2k} \ne 0$ **then** $b0 = 0$ **else** $b0 = 1$ **fi**
7. **if** $\Delta_{2k-1} \ne 0$ **then** $b1 = 0$ **else** $b1 = 1$ **fi**
8. **if** $2L_{2k-2} <= (2k-2)$ **then** $b2 = 0$ **else** $b2 = 1$ **fi**

9a.
$$\begin{bmatrix}\Lambda^{(2k)}(x)\\ B^{(2k)}(x)\end{bmatrix} = M_k(x)\begin{bmatrix}\Lambda^{(2k-2)}(x)\\ B^{(2k-2)}(x)\end{bmatrix} \text{ (refer to Table II)}$$

9b.
$$\begin{bmatrix}\Gamma^{(2k)}(x)\\ A^{(2k)}(x)\end{bmatrix} = M_k(x)\begin{bmatrix}\Gamma^{(2k-2)}(x)\\ A^{(2k-2)}(x)\end{bmatrix} \text{ (refer to Table II)}$$

10. $L_{2k} = f_k(L_{2k-2})$ (refer Table II)
11. $\alpha_{2k} = g_k(\alpha_{2k-2})$ (refer to Table II)
12. **end for**
13. Output: $\Lambda(x) = \Lambda^{(2t)}(x), \Gamma(x) = \Gamma^{(2t)}(x)$.

B. *Extension to Errors-and-Erasures Decoding*

In this section, we further develop the algorithm modifications for situations in which we want to correct a combination of errors and erasures that are present in the received word. An erasure at a particular position in the received word indicates that the received symbol at that position is incorrect. Erasure information is usually available when possible errors are detected in other parts of the system (for example, by the demodulator or by the inner decoder in a concatenated coding scheme). In error correction, both the error values and error locations need to be found. In erasure correction, on the other hand, only the erasure values are to be found (erasure locations are known at the input to the decoder). Let us assume that there are ν errors and ρ erasures in the received word. In general, ν errors and ρ erasures can be corrected if $2\nu + \rho \le d_{\min} - 1$. Let $v_i, i = 0, 1, \ldots, n-1$ denote the received vector, e_i the error values and f_i the erasure values. Also, let $i_l, l = 1, 2, \ldots, \nu$ denote the error locations and $j_l,\ l = 1, 2, \ldots, \rho$ denote the erasure locations. Note that the

TABLE III
UPDATING $\Lambda(x), B(x)$ AND L FOR THE BM ALGORITHM

Condition	$M_r(x)$	$M1_r(x)$	$f_r(L_{r-1})$
$\overline{b1}\ \overline{b2}\ b3$	$\begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1} & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1} & 0 \end{bmatrix}$	$r - L_{r-1} + \rho$
$(b1 + b2)b3$	$\begin{bmatrix} 1 & -\Delta_r x \\ 0 & x \end{bmatrix}$	$\begin{bmatrix} 1 & -\Delta_r x \\ \Delta_r^{-1} & 0 \end{bmatrix}$	L_{r-1}
$\overline{b3}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & -\Delta_r x \\ 0 & x \end{bmatrix}$	L_{r-1}

error values and the erasure values are nonzero only at the error locations and erasure locations, respectively. Define v'_i such that

$$v'_i = \begin{cases} 0 & \text{if } i \text{ is an erasure location} \\ v_i & \text{otherwise} \end{cases}$$

The syndromes S_j can be computed as

$$S_j = \sum_{j=0}^{n-1} v'_i \alpha^{ij}, j = 0, 1, \ldots, 2t-1. \tag{9}$$

Define the syndrome polynomial as $\bar{S}(x) = \sum_{j=0}^{2t-1} S_j x^j$ and the erasure polynomial using the known erasure locations as $\Psi(x) = \prod_{l=1}^{\rho}(1 - x\alpha^{j_l})$. The key equation can be written as

$$\Lambda(x)\Psi(x)\bar{S}(x) = \bar{\Gamma}(x) \bmod x^{2t} \tag{10}$$

where $\bar{\Gamma}(x)$ is the error magnitude polynomial with $\deg(\bar{\Gamma}(x)) < \rho + \nu$. Equivalently, we can write

$$\bar{\Lambda}(x)\bar{S}(x) = \bar{\Gamma}(x) \bmod x^{2t} \tag{11}$$

where $\bar{\Lambda}(x) = \Lambda(x)\Psi(x)$. It has been shown (e.g., see [23], [12], [21, Ch. 5]) that if we initialize the Berlekamp Massey algorithm as $\Lambda^{(\rho)}(x) = B^{(\rho)}(x) = \Psi(x)$ and use $\bar{S}(x)$ to compute Δ_r then at the end of $2t - \rho$ iterations, we get $\Lambda^{(2t)}(x) = \bar{\Lambda}(x)$. In addition, if we want to compute $\bar{\Gamma}(x)$ in parallel we need to find an appropriate initialization and update for $\Gamma^{(\rho)}(x)$ and $A^{(\rho)}(x)$. We can show that if we define

$$\bar{\Gamma}^{(r)}(x) = \bar{S}(x)\bar{\Lambda}^{(r)}(x) \bmod x^{\rho+r} \tag{12}$$

and

$$\bar{A}^{(r)}(x) = \bar{S}(x)\bar{B}^{(r)}(x) - x^{\rho+r-1} \bmod x^{\rho+r} \tag{13}$$

then similar to the proof in [12], $(\bar{\Gamma}^{(r)}(x), \bar{A}^{(r)}(x))$ can be updated as

$$\begin{bmatrix} \bar{\Gamma}^{(r)}(x) \\ \bar{A}^{(r)}(x) \end{bmatrix} = M_r(x) \begin{bmatrix} \bar{\Gamma}^{(r-1)}(x) \\ \bar{A}^{(r-1)}(x) \end{bmatrix} r = \rho, \rho+1, \ldots, 2t \tag{14}$$

with the matrix $M_r(x)$ defined as in Table I. This means that if we can initialize at $r = \rho$

$$\bar{\Gamma}^{(\rho)}(x) = \Psi(x)\bar{S}(x) \bmod x^{\rho} \tag{15}$$

and

$$\bar{A}^{(\rho)}(x) = \Psi(x)\bar{S}(x) - x^{\rho-1} \bmod x^{\rho} \tag{16}$$

then we can obtain at $r = 2t, \bar{\Gamma}^{(2t)}(x) = \Psi(x)\Lambda(x)\bar{S}(x) \bmod x^{2t}$. We propose to use iterations $r = 1, 2, \ldots, \rho$ to compute this initialization. If we assume that $\Psi(x)$ is pre-calculated along with the syndromes, then $\Psi(x)$ and $\bar{S}(x)$ are available at the start of the algorithm at $r = 1$. We initialize $\Lambda^{(0)}(x) = B^{(0)}(x) = \Psi(x)$. For iterations $r = 1, 2, \ldots, \rho$, we define the update matrix so that $\Lambda^{(r)}(x)$ and $B^{(r)}(x)$ are unchanged. Also, we set $A^{(r)}(x) = x^{r-1}$, $r = 1, 2, \ldots, \rho$. We notice that $\Delta_r = \Gamma_{r-1}^{(\rho)}$, $r = 1, 2, \ldots, \rho$ and that $\Delta_r = -\Delta_r$ over GF(2^m). This means that we can compute one coefficient of $\Gamma^{(\rho)}(x)$ in each iteration. In the rth iteration, we can obtain $\Gamma^{(r)}(x) = \sum_{j=0}^{r-1} \Gamma_j^{(\rho)} x^j$, $r = 1, 2, \ldots, \rho$ by using the update matrix $M1(x)$ (Row 3 of Table III). Also, at the end of the ρth iteration, we set $A^{(\rho)}(x) = \Gamma^{(\rho)}(x) - x^{\rho-1}$. This completes the initialization according to (15) and (16). The complete algorithm is given in Algorithm 3.

Berlekamp Algorithm Errors and Erasures: Algorithm 3

0. Initialize: $\Lambda^{(0)}(x) = \Psi(x), B^{(0)}(x) = \Psi(x), \Gamma^{(0)} = 0, A^{(0)} = x^{-1}, L_0 = \rho$
1. **for** $r = 1$ **to** $2t$
2. $\Delta_r = \sum_{j=0}^{L_{r-1}} \Lambda_j^{(r-1)} S_{r-1-j}$
3. **if** $\Delta_r \neq 0$ **then** $b1 = 0$ **else** $b1 = 1$
4. **if** $2L_{r-1} <= (r + \rho - 1)$ **then** $b2 = 0$ **else** $b2 = 1$
5. **if** $r <= \rho$ **then** $b3 = 0$ **else** $b3 = 1$

6a.

$$\begin{bmatrix} \Lambda^{(r)}(x) \\ B^{(r)}(x) \end{bmatrix} = M_r(x) \begin{bmatrix} \Lambda^{(r-1)}(x) \\ B^{(r-1)}(x) \end{bmatrix} \text{ (refer Table III)}$$

6b.

$$\begin{bmatrix} \Gamma^{(r)}(x) \\ A^{(r)}(x) \end{bmatrix} = M1_r(x) \begin{bmatrix} \Gamma^{(r-1)}(x) \\ A^{(r-1)}(x) \end{bmatrix} \text{ (refer Table III)}$$

7. $L_r = f_r(L_{r-1})$ (refer Table III)
8. **if** $r = \rho$ **then** $A^{(r)}(x) = \Gamma^{(r)}(x) - x^{r-1}$
9. **end for**
10. Output:$\bar{\Lambda}(x) = \Lambda^{(2t)}(x), \bar{\Gamma}(x) = \Gamma^{(2t)}(x)$.

In order that we be able to modify Algorithm 3 (in the same way that Algorithm 1 was modified to obtain Algorithm 2), we need ρ to be even. This will allow us to reduce the first ρ iterations in Algorithm 3 into $\rho/2$ iterations. We have originally assumed that the number of redundant symbols $n - k$ is even. Under this assumption we will show how to manipulate the erasures so that their number is always even (without changing the error and erasure correcting capability of the decoder). Let us consider a situation in which the number of errors and erasures are within the error-correction capability of the code (i.e., $2\nu + \rho \leq 2t$). If in addition, the number of erasures ρ in a codeword is odd, then $2\nu + \rho < 2t$. For the sake of argument, let us assume that we can convert one of the erasures into an error, we still have $2(\nu + 1) + \rho - 1 \leq 2t$ (the codeword remains correctable). Alternately, we can add one more erasure to the codeword. Again, we have $2\nu + (\rho + 1) \leq 2t$. In both these cases, we have made the effective number of erasures even. Note that if the input codeword is such that $2\nu + \rho > 2t$, the above procedure

TABLE IV
UPDATING $\Lambda(x), B(x)$ AND L FOR THE BM ALGORITHM WHERE $G(x) = 1 - \Delta_{2k-1}^{-1}\beta_{2k-1}x$ AND $G1(x) = -\beta_{2k-1}x^2 - \Delta_{2k-1}x$

Condition	$M_k(x)$	$M1_k(x)$	$f_k(L_{2k-2})$	$g_k(\alpha_{2k-2})$
$b1\ b0\ b3$	$\begin{bmatrix} 1 & 0 \\ 0 & x^2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}
$b1\ \overline{b0}\ \overline{b2}\ b3$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ \delta_{2k}^{-1} & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ \delta_{2k}^{-1} & 0 \end{bmatrix}$	$2k - L_{2k-2} + \rho$	$\delta_{2k}^{-1}\eta_{2k}$
$b1\ \overline{b0}\ b2\ b3$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ 0 & x^2 \end{bmatrix}$	$\begin{bmatrix} 1 & -\delta_{2k}x^2 \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}
$\overline{b1}\ \overline{b2}\ b3$	$\begin{bmatrix} G(x) & -\Delta_{2k-1}x \\ x\Delta_{2k-1}^{-1} & 0 \end{bmatrix}$	$\begin{bmatrix} G(x) & -\Delta_{2k-1}x \\ x\Delta_{2k-1}^{-1} & 0 \end{bmatrix}$	$2k - 1 - L_{2k-2} + \rho$	$\Delta_{2k-1}^{-1}\delta_{2k}$
$\overline{b1}\ b2\ b3$	$\begin{bmatrix} 1 & G1(x) \\ 0 & x^2 \end{bmatrix}$	$\begin{bmatrix} 1 & G1(x) \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}
$\overline{b3}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & G1(x) \\ 0 & x^2 \end{bmatrix}$	L_{2k-2}	α_{2k-2}

can introduce additional errors/erasures in uncorrectable codewords. We can detect such a case during the decoding process and flag it so that no changes are made to the codeword as it passes through the decoder.

We note that for iterations $k > \rho/2$ in Algorithm 3, the modifications are the same as when we derived Algorithm 2. These modifications are shown in the first five rows of Table IV. In order to compute $\Gamma^{(\rho)}(x)$ and $A^{(\rho)}(x)$ in the first ρ iterations, we define the update matrix so that $A^{(2k)}(x) = x^{2k-1}; k = 1, 2, \ldots, \rho/2$. In addition, we recognize that we can obtain coefficients of the $\Gamma(x)$ polynomial as $\Delta_{2k-1} = \Gamma_{2k-2}^{(\rho)}$ and $\delta_{2k} = \Gamma_{2k-1}^{(\rho)},\ k = 1, 2, \ldots, \rho/2$. In each iteration, we compute 2 coefficients of the polynomial $\Gamma^{(\rho)}(x)$. Additionally, $\alpha_{2k} = 0; k = 1, 2, \ldots, \rho/2$ which ensures that $\beta_{2k-1} = \delta_{2k}$. Therefore, Row 6 of Table IV performs the correct update of the polynomials $\Gamma(x)$ and $A(x)$. At iteration $k = \rho/2$, we set $A^{(\rho)}(x) = \Gamma^{(\rho)}(x) - x^{\rho-1}$ and ensure that α_ρ is initialized correctly for the iterations that compute the error locator polynomial (i.e., for $k > \rho/2$).

Low-Power Berlekamp Algorithm- Errors and Erasures: Algorithm 4

0a. Initialize $\Lambda^{(0)}(x) = 1, B^{(0)}(x) = 1, \Gamma^{(0)}(x) = 0, A^{(0)}(x) = x^{-1}, L_0 = 0$
0b. **if** $\rho = 0$**then** $\alpha_0 = S_0$ **else** $\alpha_0 = 0$ **fi**
1. **for** $k = 1$ **to** t
2. $\Delta_{2k-1} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-2-j}$
3. $\delta_{2k} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-1-j}$
4. $\eta_{2k} = \sum_{j=0}^{L_{k-1}} \Lambda_j^{(2k-2)} S_{2k-j}$
5. $\beta_{2k-1} = \delta_{2k} - \Delta_{2k-1}\alpha_{2k-2}$
6. **if** $\delta_{2k} \neq 0$ **then** $b0 = 0$ **else** $b0 = 1$ **fi**
7. **if** $\Delta_{2k-1} \neq 0$ **then** $b1 = 0$ **else** $b1 = 1$ **fi**
8. **if** $2L_{2k-2} <= (2k-2)$ **then** $b2 = 0$ **else** $b2 = 1$ **fi**
9. **if** $2k \leq \rho$ **then** $b3 = 0$ **else** $b3 = 1$ **fi**

10a.

$$\begin{bmatrix} \Lambda^{(2k)}(x) \\ B^{(2k)}(x) \end{bmatrix} = M_k(x) \begin{bmatrix} \Lambda^{(2k-2)}(x) \\ B^{(2k-2)}(x) \end{bmatrix} \text{ (refer Table II)}$$

10b.

$$\begin{bmatrix} \Gamma^{(2k)}(x) \\ A^{(2k)}(x) \end{bmatrix} = M1_k(x) \begin{bmatrix} \Gamma^{(2k-2)}(x) \\ A^{(2k-2)}(x) \end{bmatrix} \text{ (refer Table II)}$$

11. $L_{2k} = f_k(L_{2k-2})$ (refer Table II)
12. $\alpha_{2k} = g_k(\alpha_{2k-2})$ (refer Table II)
13. **if** $2k = \rho$ **then** $A^{(r)}(x) = \Gamma^{(r)}(x) - x^{r-1}, \alpha_{2k} = \eta_{2k}$ **fi**
14. **end for**
15. Output: $\Lambda(x) = \Lambda^{(2t)}(x), \Gamma(x) = \Gamma^{(2t)}(x)$.

We note that these modified algorithms can be used to decode shortened and punctured RS codes without any further change. This is because we can conceptually treat all the dropped symbols as erased positions [2] and use the same decoding algorithms. Also, the errors-and-erasures decoder algorithm can be used for encoding [2]. In this case, the k message symbols along with $n-k$ erased symbols are fed into the decoder. The erasures are "corrected" by the decoder so that the output of the decoder is the n symbol codeword corresponding to the k message symbols.

IV. VLSI ARCHITECTURE

In the previous section, we discussed algorithm-level transformations for the Berlekamp algorithm. A variety of architectures can be developed based on these algorithms. In this section, we discuss how this algorithm transformation translates into one specific VLSI architecture for the errors-only decoding problem. A similar architecture can also be developed for the errors-and-erasures case. In addition, we also discuss architecture level transformations for the syndrome and error computations in errors-only decoding.

A. Architecture for Syndrome Computation

The syndrome computation is given by

$$S_j = \sum_{i=0}^{n-1} \alpha^{i(b+j)} v_i; \quad j = 0, 1, \ldots, (2t-1). \tag{17}$$

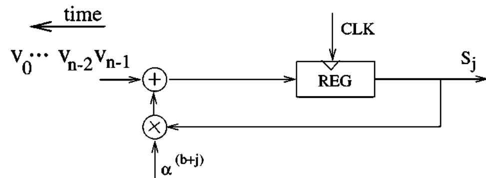

Fig. 4. An architecture for the normal syndrome computation.

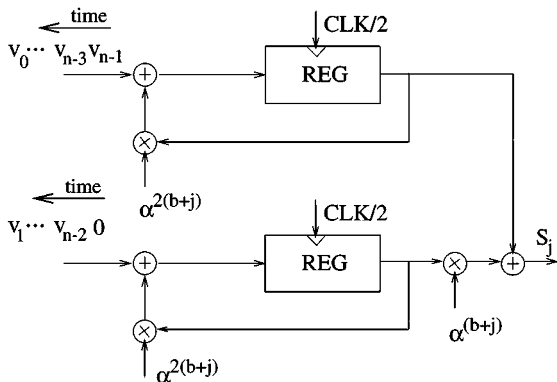

Fig. 5. An architecture for the modified syndrome computation.

This equation can be computed using Horner's rule as

$$S_j = ((\cdots(v_{n-1}\alpha^{b+j} + v_{n-2})\alpha^{b+j} + \cdots)\alpha^{b+j} + v_0. \quad (18)$$

An architecture that computes S_j is shown in Fig. 4. Let us assume that the register is cleared before we start the computation and that the sequence $v_{n-1}, v_{n-2}, \ldots, v_0$ is fed into the input in that order. At clock cycle l, the previous accumulated value is multiplied with $\alpha^{(b+j)}$ and added to v_{n-l}. At the end of n clock cycles, we obtain S_j in the register. Using $2t$ such cells, we can compute $S_j; j = 0, 1, \ldots, 2t-1$ in parallel.

In order to enable a parallel computation which can be completed in approximately $n/2$ clock cycles, we can rewrite (17) as

$$S_j = \sum_{l=0}^{\lfloor n/2 \rfloor} \alpha^{2l(b+j)} v_{2l} + \alpha^{b+j} \sum_{l=0}^{\lfloor n/2 \rfloor - 1} \alpha^{2l(b+j)} v_{2l+1} \quad (19)$$

when n is odd. The limits of both the sub-summations will change to $n/2-1$ when n is even. Note from (19) that the first and second sub-summations consist of even and odd symbols, respectively. Both these sub-summations can be computed using Horner's rule (similar to (18)). An architecture for this computation is shown in Fig. 5. At the end of $\lfloor n/2 \rfloor + 1$ clocks (when n is odd), we can obtain the two sub-summations. Finally, the odd sub-summation result is multiplied by α^{b+j} and added to the even sub-summation. This means that the modified architecture can complete the computation in approximately half the number of clock cycles as the normal syndrome computation.

B. Architecture for Error Computation

The error value computation based on Forney's method can be written as

$$e_i = \begin{cases} 0, & \text{if } \Lambda(\alpha^{-i}) \neq 0 \\ -\frac{\alpha^i \Gamma(\alpha^{-i})}{\alpha^{bi}\Lambda'(\alpha^{-i})}, & \text{if } \Lambda(\alpha^{-i}) = 0. \end{cases} \quad (20)$$

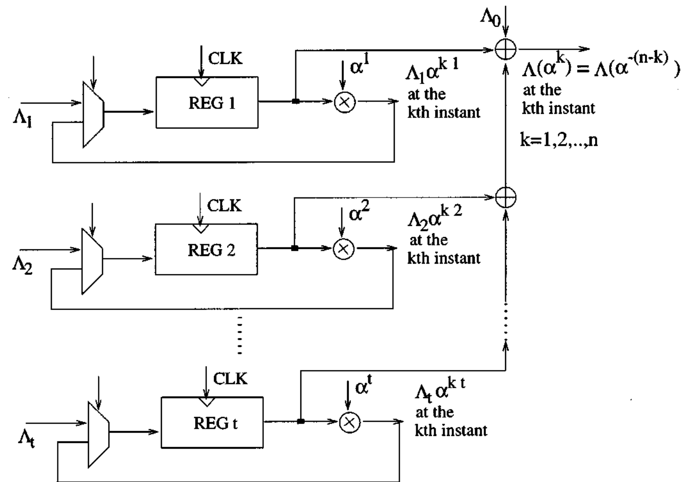

Fig. 6. An architecture for the normal error computation.

Note that we need to compute $\Lambda(\alpha^{-i}), \Lambda'(\alpha^{-i})$ and $\Gamma(\alpha^{-i})$ for $i = 0, 1, \ldots, n-1$ in order to compute all the e_i. In any field of characteristic 2

$$\Lambda'(x) = \sum_{k=0}^{t} k\Lambda_k x^{k-1} = \sum_{k:\text{odd}} \Lambda_k x^{k-1} \quad (21)$$

and therefore, $(\alpha^{-i})\Lambda'(\alpha^{-i})$ can be obtained as part of the computation for $\Lambda(\alpha^{-i})$. This is because $\Lambda(x) = \Lambda_{\text{even}}(x^2) + x\Lambda_{\text{odd}}(x^2)$, where $\Lambda_{\text{even}}(x)$ and $\Lambda_{\text{odd}}(x)$ are the polynomials formed by the even coefficients and the odd coefficients of $\Lambda(x)$, respectively. In addition, the circuit computes $\alpha^{-i}\Gamma(\alpha^{-i})$ (this avoids the need for the initialization of $A(x) = x^{-1}$ in Algorithm 2). We can write the error computation in these terms as

$$e_i = \begin{cases} 0 & \text{if } \Lambda(\alpha^{-i}) \neq 0 \\ -\frac{\alpha^{-i}\Gamma(\alpha^{-i})}{\alpha^{(b-1)i}\alpha^{-i}\Lambda'(\alpha^{-i})} & \text{if } \Lambda(\alpha^{-i}) = 0 \end{cases} \quad (22)$$

An architecture for the evaluation of $\Lambda(x)$ at α^{-i} is shown in Fig. 6. Initially, the coefficients of the $\Lambda(x)$ polynomial are loaded into the registers. After the initialization is complete, the multiplexer passes the value that is fed back. The value in the ith register is multiplied by α^i, so that the ith register holds $\Lambda_i\alpha^{ki}$ at the kth clock cycle. The contents of the registers are added up to obtain $\Lambda(\alpha^{-(n-k)})$ for $k = 1, 2, \ldots, n$. A similar architecture can be used to evaluate $\Gamma(\alpha^{-(n-k)})$. This implies that we can evaluate e_{n-k} at the kth cycle. In other words, in n cycles we can compute all the required error values.

If a parallel architecture is used to compute e_{2i} and e_{2i+1}, then all the required error values can be computed in $\lfloor n/2 \rfloor + 1$ cycles. This modified architecture is illustrated in Fig. 7. In this case, the outputs of the registers $1, 3, \ldots, 2t-3$ are used to compute $\Lambda(\alpha^{-(n-2k)})$ and the outputs of the registers $2, 4, \ldots, 2t-2$ compute $\Lambda(\alpha^{-(n-(2k+1))})$ for $k = 0, 1, \ldots, \lfloor n/2 \rfloor$ when n is odd.

C. Architecture for the Berlekamp Algorithm

The original Berlekamp and its modified version have been discussed in detail in Section III. In this section, we discuss architectures for both Algorithm 1 and Algorithm 2. An architecture for performing the computation described

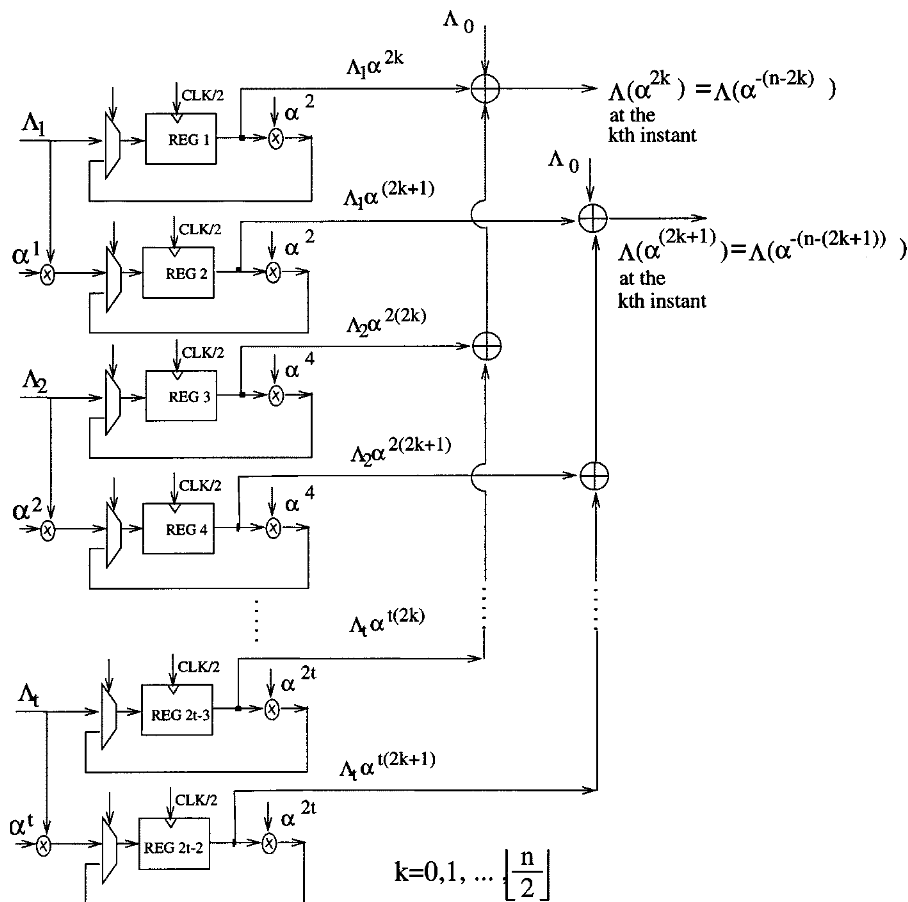

Fig. 7. An architecture for the modified error computation.

in Algorithm 1 is shown in Fig. 8. The registers shown in the diagram hold the polynomials $S(x), \Lambda(x), \Gamma(x), B(x)$ and $A(x)$. The exact manner in which they hold the various coefficients will become clear shortly. The r register holds the algorithm iteration counter and mirrors the variable r in Algorithm 1. At the start of the rth iteration, register Λ holds $(\Lambda_0^{(r-1)}, \Lambda_1^{(r-1)}, \ldots, \Lambda_t^{(r-1)})$ in positions $p[0], p[1], \ldots, p[t]$, respectively. In the next $t+1$ clocks, $\Lambda_l^{(r)};\ l = 0, 1, \ldots, t$ are computed in sequence and shifted into position $p[t]$. The l register holds the serial iteration counter that increments at every clock and synchronizes the various serial operations. The l counter has a range from $l = 0, 1, \ldots, t+2$ (i.e., l is a modulo $t+3$ counter) and r in incremented when $l = t+2$. Δ_{r+1} is accumulated serially at the same time that $\Lambda^{(r-1)}(x)$ is serially updated to $\Lambda^r(x)$. In particular, the partially accumulated result $\sum_{l'=0}^{l-1} \Lambda_{l'}^{(r)} S_{r-l'}, l = 1, 2, \ldots, t+1$ is available when the serial iteration counter has the value l. The contents and updates of register S are defined in order to enable the computation of Δ_{r+1} in iteration r when $\Lambda^{(r-1)}(x)$ is updated to $\Lambda^{(r)}(x)$. Register S is initialized so that it holds $S_2, S_3, \ldots, S_{2t-1}, S_0, S_1$ in positions $p[0], p[1], \ldots, p[2t-1]$, respectively. For $l = 1, 2, \ldots, (t+1)$, the positions $p[t-1], p[t], \ldots, p[2t-1]$ are right rotated by one position each time and the postions $p[0], \ldots, p[t-2]$ are kept unchanged, while Δ_{r+1} is computed. When $l = t+2$, positions $p[0], p[1], \ldots, p[2t-1]$ of register S are left rotated by one position so that $p[2t-1]$ holds $S_{r+1 \bmod 2t}$.

At the beginning of iteration r, Δ_r and L_{r-1} are available. Note that Δ_1 is initialized to S_0. These variables are used to compute the variables in the update matrix $M_r(x)$ as well as other decision variables such as $b1$ and $b2$. Since the update matrix $M_r(x)$ contains x in its second column, we conclude that a shifted version of $B(x)$ (i.e., $xB(x)$) is also required. In order to enable an update for $B(x)$ in a serial manner, a new position $p[-1]$ is introduced. Register B holds in positions $p[-1], p[0], \ldots, p[t]$ the values $0, B_0^{(r-1)}, B_1^{(r-1)}, \ldots, B_t^{(r-1)}$ at the start of iteration r. The results are shifted into register B at position $p[t]$. Depending on whether the shifted version of $B(x)$ is required or not, the contents of either $p[-1]$ or $p[0]$ is used in the update. $\Gamma(x)$ and $A(x)$ are updated in a similar manner since the update matrix $M_r(x)$ is the same as that for $\Lambda(x)$ and $B(x)$.

A simliar architecture can be developed corresponding to the modified algorithm (i.e., Algorithm 2). This architecture is shown in Fig. 9. As before, the inner products for $\Delta_{2k+1}, \delta_{2k+2}$ and η_{2k+2} are computed as $\Lambda^{(2k-2)}(x)$ is being updated to $\Lambda^{(2k)}(x)$. The algorithm iteration counter k in this case counts from $k = 1, 2, \ldots, t$. In this case, the serial iteration counter increments from 0 to $t+3$ (i.e., l is a modulo $t+4$ counter). We note the presence of x in the first column of the update matrix $M_k(x)$. This indicates that a shifted version of $\Lambda^{(2k-2)}(x)$ is required. This means that we need to add one position $p[-1]$ to the Λ register (just as we did for the B register in the architecture for the original algorithm). The second column

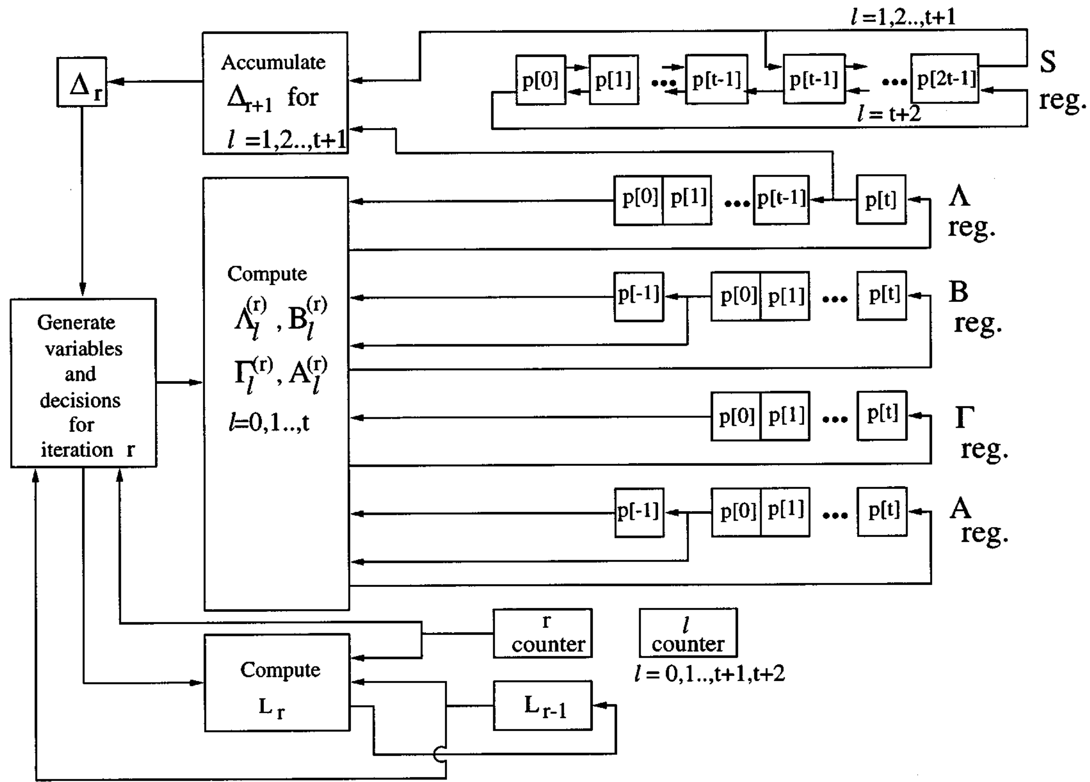

Fig. 8. An architecture for the Original Berlekamp algorithm.

of $M_k(x)$ contains both x and x^2. This suggests that we need two shifted versions of $B^{(2k-2)}(x)$ (i.e., $xB^{(2k-2)}(x)$ and $x^2B^{(2k-2)}(x)$). We add two positions $p[-1]$ and $p[-2]$ to the B register. In particular, at the start of the kth iteration the Λ register holds $(0, \Lambda_0^{(2k-2)}, \Lambda_1^{(2k-2)}, \ldots, \Lambda_t^{(2k-2)})$ in positions $p[-1], p[0], p[1], \ldots, p[t]$ respectively. As the serial iteration counter l increments from 0 to $t, \Lambda_l^{(2k)}$ are computed in sequence. The B register at the start of kth iteration holds $0, 0, B_0^{(2k-2)}, B_1^{(2k-2)}, \ldots, B_t^{(2k-2)}$ in positions $p[-2], p[-1], p[0], p[1], \ldots, p[t]$ respectively. $B(x), \Gamma(x)$ and $A(x)$ are updated in a similar manner, while l increments from 0 to t. $\Delta_{2k+1}, \delta_{2k+2}$ and η_{2k+2} are accumulated, and while l increments from 1 to $t+1$. The register S must be organized to provide the appropriate data for the $\Delta_{2k+1}, \delta_{2k+2}$ and η_{2k+2} accumulators. At the beginning of iteration 1, register S holds $(S_3, S_4, \ldots, S_{2t-1} S_0, S_1, S_2)$ in positions $p[0], p[1], \ldots, p[2t-1]$ and (S_3, S_4) in positions $p[2t], p[2t+1]$, respectively. For $l = 1, 2, \ldots, t+1$, positions $p[0], \ldots, p[t-2]$ are kept unchanged, while positions $p[t-1], p[t], \ldots, p[2t-1]$ are rotated and positions $p[2t], p[2t+1]$ are right shifted using the value out of $p[2t-1]$. When $l = t+2$ or $t+3$, positions $p[0], \ldots, p[2t-1]$ are left rotated so that $p[2t-1]$ holds $S_{2k+2 \bmod 2t}$. When $l = t+3, p[2t]$ and $p[2t+1]$ are loaded with the values being shifted into $p[0]$ and $p[1]$, respectively. This ensures appropriate operation of the architecture in Fig. 9.

In summary, Fig. 8 describes an architecture for the original errors-only algorithm (i.e., Algorithm 1), while Fig. 9 describes an architecture for the modified errors-only algorithm (i.e., Algorithm 2).

D. Extension of Architecture to Handle Errors and Erasures

When erasures are present in addition to errors, the symbols of the received word that are erasures are flagged. We refer to these flags as erasure indicators $\mathrm{er}_i; i = 0, 1, \ldots, n-1$. The binary variable er_i indicates the presence or absence of an erasure (i.e., $\mathrm{er}_i = 1$ indicates the presence of an erasure and $\mathrm{er}_i = 0$ the absence of an erasure at location i).

We know from Section III-B that the erasure locator polynomial needs to be computed to provide the appropriate initialization for Algorithm 3. Fig. 10 shows an architecture that computes the erasure locator polynomial from the erasure indicators. This computation is performed in parallel with the computation of the syndrome (see Fig. 4. Note that the only modification required in the syndrome computation is that when $\mathrm{er}_i = 1, v_i$ is set to zero. The registers Ψ_i correspond to the coefficients of the erasure locator polynomial $\Psi(x)$. Register Ψ_0 is initialized to oneand the other registers Ψ_i's are set to zero. If $\mathrm{er}_{n-1-l} = 1$ the erasure locator coefficients need to be updated based on

$$\begin{aligned}(\Psi_0 + \Psi_1 x + \cdots + \Psi_\rho x^\rho)(1 + x\alpha^{n-1-l}) \bmod x^{\rho+1} \\ = \Psi_0 + (\Psi_1 + \Psi_0\alpha^{n-1-l})x + (\Psi_2 + \Psi_1\alpha^{n-1-l})x^2 \\ + \cdots + (\Psi_\rho + \Psi_{\rho-1}\alpha^{n-1-l})x^\rho. \qquad (23)\end{aligned}$$

Therefore, the updated coefficients are given by $\Psi_{k,\mathrm{new}} = (\Psi_k + \Psi_{k-1}\alpha^{n-1-l})$. Note that $\alpha^{n-1-l}; l = 0, 1, \ldots, n-1$ is generated and stored in the register shown on the right side of Fig. 10. Depending on whether $\mathrm{er}_{n-1-l} = 1$ or $\mathrm{er}_{n-1-l} = 0$ the coefficients are modified or left unmodified as indicated by the multiplexer circuitry. After n clocks, we get the coefficients

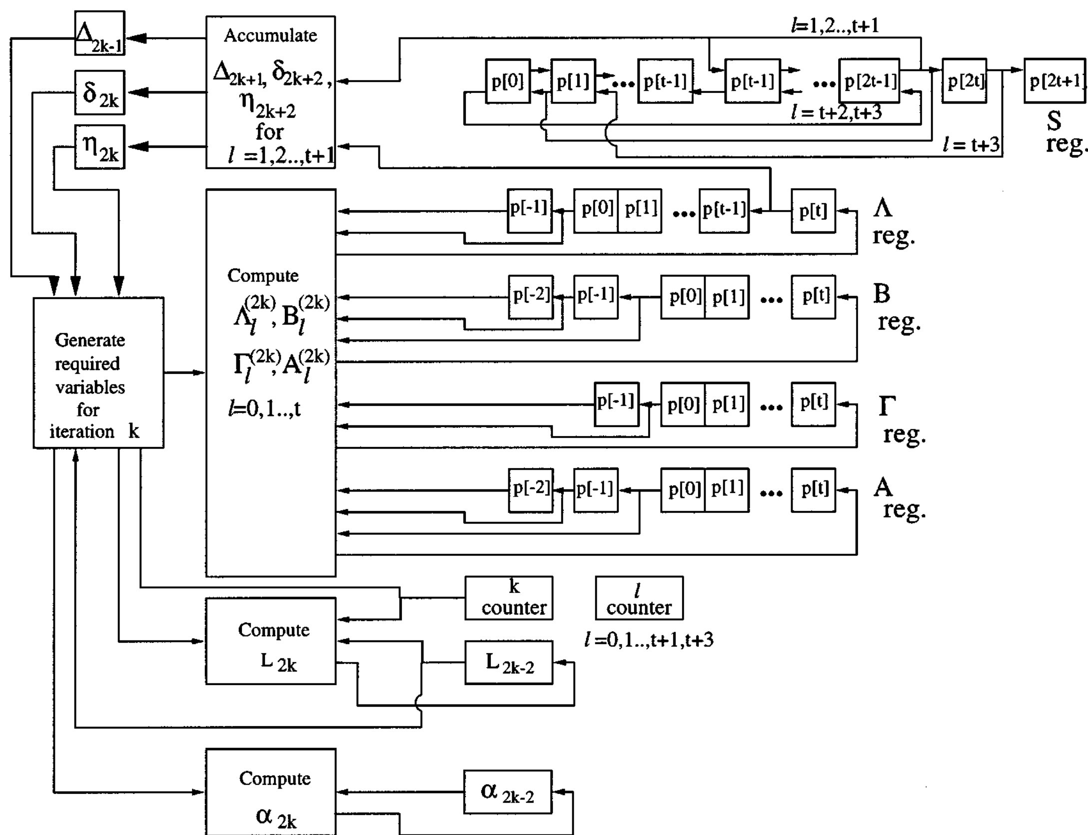

Fig. 9. An architecture for the modified Berlekamp algorithm.

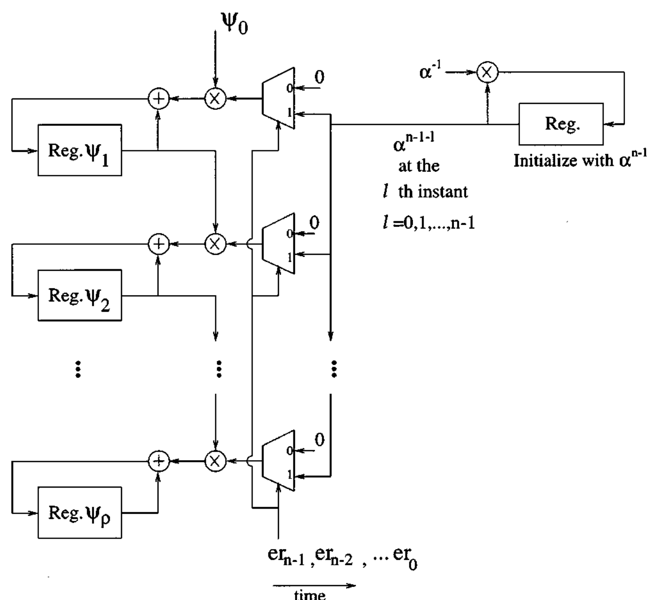

Fig. 10. An architecture for the normal erasure locator computation.

of the erasure locator polynomial $\Psi(x)$. An erasure counter is also maintained to count the number of errors in the codeword.

An architecture for the modified erasure locator computation is shown in Fig. 11. This allows the erasure locator computation to be completed in $\lfloor n/2 \rfloor + 1$ clocks so that this computation can be done in parallel with the modified syndrome computation. We assume that, again, register Ψ_0 is initialized to one and the other registers Ψ_i's are set to zero. Since we have two erasure indicators that are input at one time, we have four cases corresponding to the four binary combinations of er_{n-1-2l} and $\text{er}_{n-1-(2l-1)}$. Case A corresponds to the combination $\text{er}_{n-1-2l}\text{er}_{n-1-(2l-1)} = 00$. In this case, the registers Ψ_i need not be changed. Case B corresponds to the combination $\text{er}_{n-1-2l} = 0$ and $\text{er}_{n-1-(2l-1)} = 1$. In this case, we need to modify the registers based on

$$\begin{aligned}(\Psi_0 + \Psi_1 x + \cdots + \Psi_\rho x^\rho)\left(1 + x\alpha^{n-1-(2l-1)}\right) \bmod x^{\rho+1} \\ = \Psi_0 + \left(\Psi_1 + \Psi_0\alpha^{n-1-(2l-1)}\right)x \\ + \left(\Psi_2 + \Psi_1\alpha^{n-1-(2l-1)}\right)x^2 + \cdots \\ + \left(\Psi_\rho + \Psi_{\rho-1}\alpha^{n-1-(2l-1)}\right)x^\rho. \end{aligned} \tag{24}$$

In case C, which corresponds to $\text{er}_{n-1-2l} = 1$ and $\text{er}_{n-1-(2l-1)} = 0$, we need to update based on

$$\begin{aligned}(\Psi_0 + \Psi_1 x + \cdots + \Psi_\rho x^\rho)(1 + x\alpha^{n-1-2l}) \bmod x^{\rho+1} \\ = \Psi_0 + (\Psi_1 + \Psi_0\alpha^{n-1-2l})x + (\Psi_2 + \Psi_1\alpha^{n-1-2l})x^2 \\ + \cdots + (\Psi_\rho + \Psi_{\rho-1}\alpha^{n-1-2l})x^\rho. \end{aligned} \tag{25}$$

For case C, the update for the coefficient $\Psi_i; i > 0$ is given by $\Psi_{i,\text{new}} = \Psi_i + \Psi_{i-1}(\alpha^{n-1-2l} + \alpha^{n-1-(2l-1)}) + \Psi_{i-2}\alpha^{2(n-1)-(4l-1)}$. In case D, which corresponds to

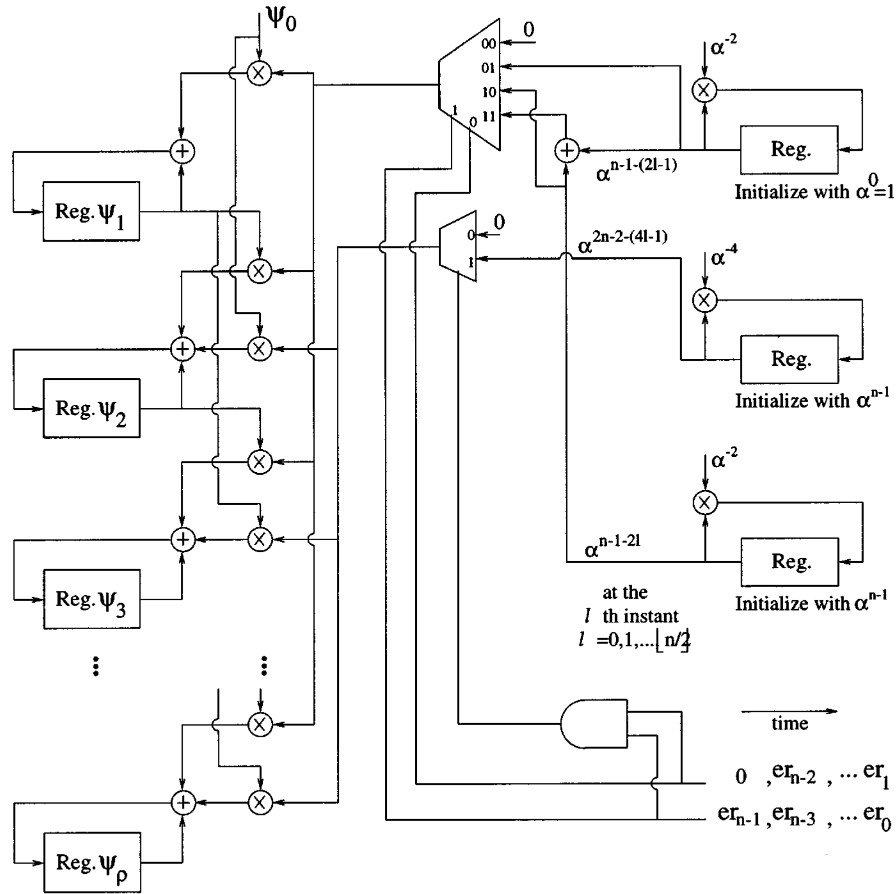

Fig. 11. An architecture for the modified erasure locator computation.

$\mathrm{er}_{n-1-2l}er_{n-1-(2l-1)} = 11$, we need to update the registers based on

$$\begin{aligned}&(\Psi_0 + \Psi_1 x + \cdots + \Psi_\rho x^\rho)(1 + x\alpha^{n-1-2l}) \\ &\quad \times \left(1 + x\alpha^{n-1-(2l-1)}\right) \bmod x^{\rho+1} \\ &= \Psi_0 + \left[\Psi_1 + \Psi_0\left(\alpha^{n-1-2l} + \alpha^{n-1-(2l-1)}\right)\right] x \\ &\quad + \left[\Psi_2 + \Psi_1\left(\alpha^{n-1-2l} + \alpha^{n-1-(2l-1)}\right)\right. \\ &\quad \left. + \Psi_0 \alpha^{2(n-1)-(4l-1)}\right] x^2 \\ &\quad + \cdots + \left(\Psi_\rho + \Psi_{\rho-1}\left(\alpha^{n-1-2l} + \alpha^{n-1-(2l-1)}\right)\right. \\ &\quad \left. + \Psi_\rho \alpha^{2(n-1)-(4l-1)}\right) x^\rho. \end{aligned} \tag{26}$$

For case D, the update for the coefficient $\Psi_i; i > 1$ is given by $\Psi_{i,\mathrm{new}} = \Psi_i + \Psi_{i-1}(\alpha^{n-1-2l} + \alpha^{n-1-(2l-1)}) + \Psi_{i-2}\alpha^{2(n-1)-(4l-1)}$. The architecture corresponding to this update is shown in Fig. 11. As we mentioned earlier, we can obtain the coefficients of Ψ_i after $\lfloor n/2 \rfloor + 1$ clocks.

Algorithm 4 works only when ρ is even. We explained in Section III-B how the number of erasures can be made even without loss in performance (within the code's error-correcting capability) as long as the $n-k$ is even. The computation of the erasure locator polynomial has to take care of this. We have to pre-modify some of the erasure indicators (in particular, er_1 and er_0). If, at the start of iteration $l = \lfloor n/2 \rfloor$, the erasure counter is odd and $\mathrm{er}_1 = \mathrm{er}_0$, then we set $\mathrm{er}_1 = 0$ and $\mathrm{er}_0 = 1$. On the other hand, if the erasure counter is even and $\mathrm{er}_1 \neq \mathrm{er}_0$, then we set $\mathrm{er}_1 = 1$ and $\mathrm{er}_0 = 1$. In all other cases, the original values of er_1 and er_0 are retained. This ensures that the erasure locator polynomial always has an even degree.

E. Architecture Level Power Estimates

In Sections IV-A through IV-C, we have described VLSI architectures for the normal and modified algorithm. The original syndrome computation takes approximately n clocks, while the modified version takes only approximately $n/2$. Similarly, the error computation takes n clocks, while the modified version takes approximately $n/2$ clocks. The original Berlekemp architecture takes $2t(t+3)$ clocks, while the modified version takes $t(t+4)$. Let us assume that each of the modules are implemented as state machines with pipelining between the modules. This indicates that our algorithm and architectural modifications can be put into the framework, as shown in Fig. 2. Fig. 12(a) and (b) show the pipeline stages and the number of clock cycles used by each module of the original architecture and modified architecture, respectively. We also observe that the architectures described in the previous sub-sections are independent of t and the critical paths are independent of t. We will investigate under

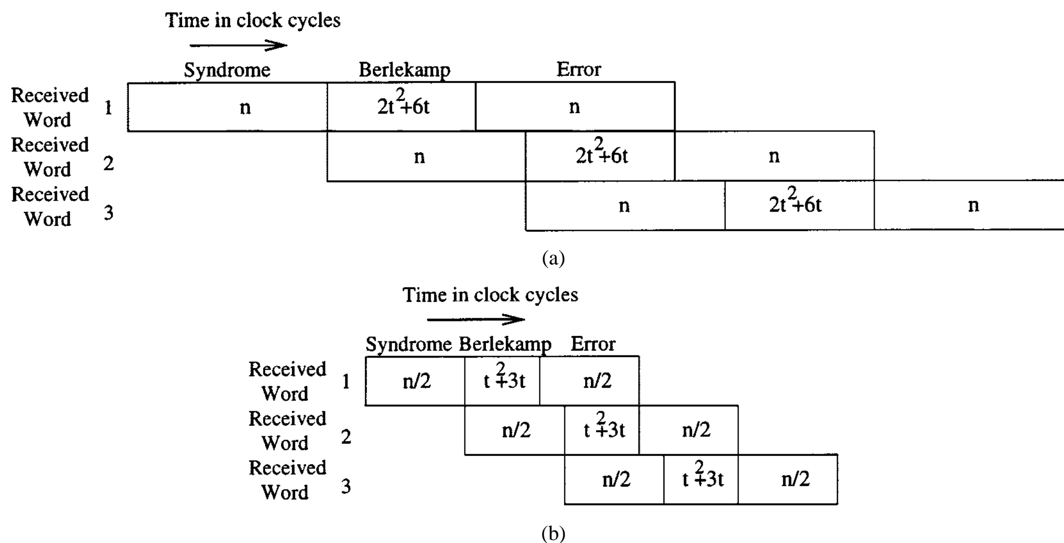

Fig. 12. Timing of the Pipeline Stages.

what conditions we can expect a low-power implementation.

Let us assume that the normal circuit can work at a maximum clock speed of f and the modified one at a maximum clock speed of f'. We can operate the modified design in two modes: high speed and low power. In the high-speed mode, the modified design is operated at the same supply voltage as the normal design. The throughput will be $2 \times f'/f$ times the throughput of the normal design. So a speedup can be expected as long as $f' > f/2$. In the low-power mode, the throughput is maintained constant and the voltage of the modified design is scaled down. This means that the clock rate of the low-power design can be slowed down to $f/2$. If the critical paths of the normal and modified designs at the maximum voltage V are $T = 1/f$ and $T' = 1/f'$, respectively, then the voltage can be reduced to V' (at this voltage the critical path delay in the modified design becomes $T'' = 2T$). The ratio of capacitances can be estimated as the ratio of active areas (A) of the designs. Therefore, the ratio of the power consumption in the modified design to the normal design can be written based on (2) as

$$\frac{P_M}{P_N} = \frac{A_M}{A_N}\left(\frac{V'}{V}\right)^2 \frac{f/2}{f} \tag{27}$$

We can obtain accurate estimates for T, T', A_M and A_N by actually synthesizing a VLSI layout of both these designs and performing SPICE simulations of the critical paths. We also note that at the same supply voltage, our modified design has a smaller latency when compared with the normal design. In particular, at the voltage V, the latency of the modifed design is $T'/2T$ times the latency of the normal design. This can be important in applications in which decoding delay is critical.

Note that the two decoders shown in Fig. 2 generate the error word as output. This means that for both designs we need to maintain the received word in a buffer and finally add the error word to the received word to get the corrected word. Note that if we want the exact same input and output data interface for

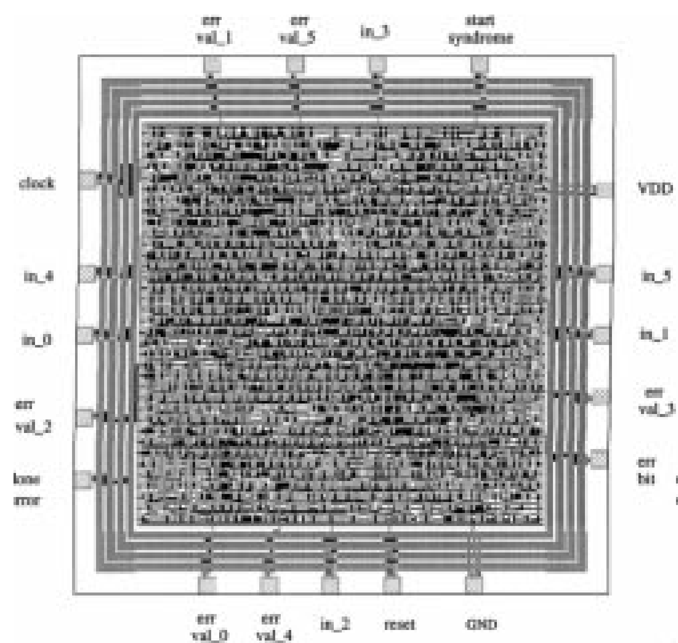

Fig. 13. VLSI layout of the original design.

both designs, then we need to generate the odd and output sequences on-chip at the input of the modified decoder. Also, we need to multiplex the two error values into a single sequence at the output of the modified decoder. In particular, at the input, we need just two m bit registers and an m bit 2 to 1 multiplixer (working at a clock rate f) to direct the odd and even inputs into the correct registers. Similarly, at the output, we need two m bit registers and an m bit 2 to 1 multiplexer (working at a clock rate f) that chooses the appropriate register to direct to output. We note that the additional circuit complexity required for this is very small. In our designs, we assume that the modified design accepts two inputs in parallel and generates two outputs in parallel. In the next section, we will describe some of the considerations involved in the VLSI design of the normal and modified architectures.

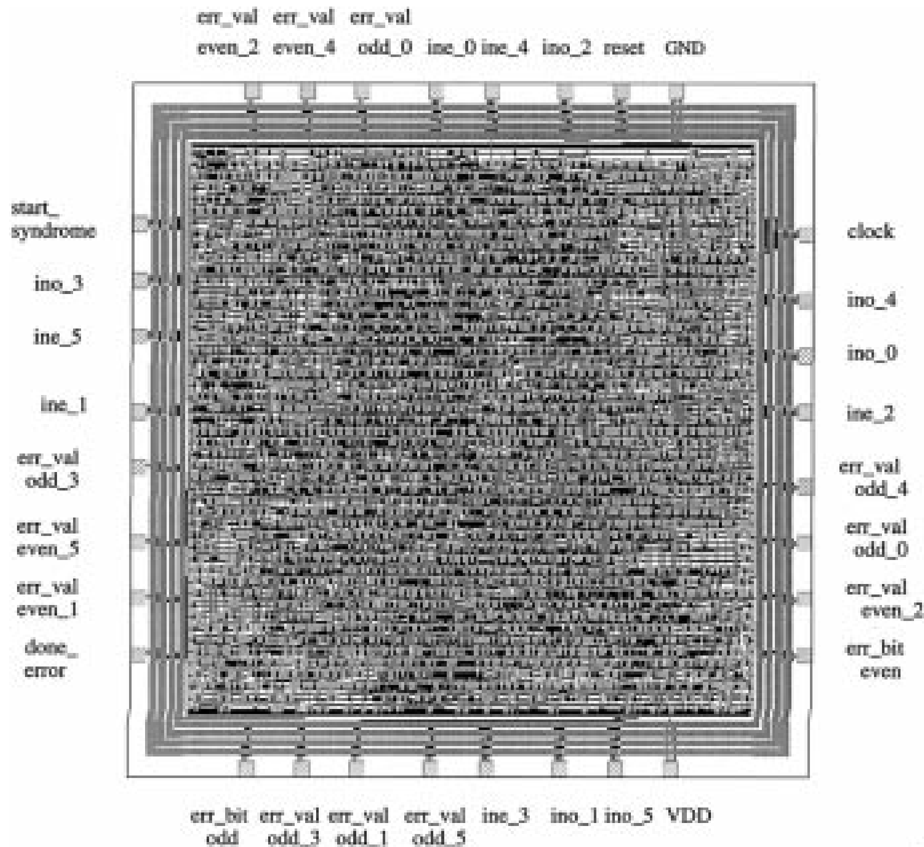

Fig. 14. VLSI layout of the modified design.

TABLE V
SUMMARY OF THE NORMAL AND MODIFIED DESIGNS

Parameter	Normal	Modified
No. of Transistors	26070	50639
Active Area	$3.52mm \times 3.52mm$	$5.05mm \times 5.05mm$
SPICE Critical Path Timing, $V = 5.0$v	$T_{norm} = 5.2ns$	$T_{mod} = 7.45ns$
SPICE Critical Path Timing, $V' = 3.6$v	N/A	$T'_{mod} = 2 \times 5.2ns$
Symbol Rate at $V = 5$v	192Msymbols/s	260Msymbols/s

V. VLSI DESIGN

Based on the normal and modified architectures developed in the previous section, we designed two separate decoders for a $(63, 57)$ RS code that can correct up to three errors. A binary representation for $\mathrm{GF}(64)$ was chosen to minimize the complexity (in terms of the number of transistors) of the GF multiplier and GF inverter. Elements of $\mathrm{GF}(64)$ were written in terms of a concatenation of two $\mathrm{GF}(8)$ elements (see [21, Ch. 10]. Based on this representation, the multiplication over $\mathrm{GF}(64)$ can be written in terms of multiplications over $\mathrm{GF}(8)$. The inverse over $\mathrm{GF}(64)$ can again be written in terms of multiplications over $\mathrm{GF}(64)$ and $\mathrm{GF}(8)$ as well as inversions over $\mathrm{GF}(8)$ (see [21, Ch. 10]. This allowed for an optimal choice of binary representation for $\mathrm{GF}(8)$ that leads to low complexity multipliers and inverters over $\mathrm{GF}(64)$. Multipliers and inverters were constructed as combinational circuits [21, Ch. 10], [40].

The complete algorithm was simulated in C and the test vectors were generated for the circuit. The architecture was then described using Verilog HDL and the functionality was verified. Then a timing driven synthesis was performed using Cadence's Synergy synthesis tool. A 0.8-μ CMOS standard cell library was used with the synthesis tool to obtain a flat gate level netlist. The functionality of the netlist was again verified against the C simulations. Finally, Cadence's Silicon Ensemble place and route tool was used to produce a channel-less layout. Routing was performed using three levels of metal. A SPICE netlist of the critical path transistors was extracted along with routing capacitances.

The above design process was completed for the normal and modified designs. The layout of the normal and modified designs are shown in Figs. 13 and 14, respectively. The corresponding areas are $A_N = 3.52$ mm $\times$ 3.52 mm and $A_M = 5.05$ mm $\times$ 5.05 mm. Table V summarizes the results. Fig. 15 shows how the critical path delays vary with voltage. The critical path delays were extracted from SPICE simulations. At a supply voltage of $V = 5$ V, the critical path delays of the normal and modified designs are $T = 5.2$ and $T' = 7.75$ ns. The voltage V' corresponds to the supply voltage at which the modified design is slowed down to $T'' = 2 \times 5.2$ ns. From Fig. 15, we obtain $V' = 3.6$ V for the modified design. This shows that the power consumption can be reduced by about 40%. Alternately, a speed-up by a factor of 1.34 can be obtained. In particular, the modified design in the high-speed mode can support a throughput of up to 192 Msymbols/s. Note that if a slower

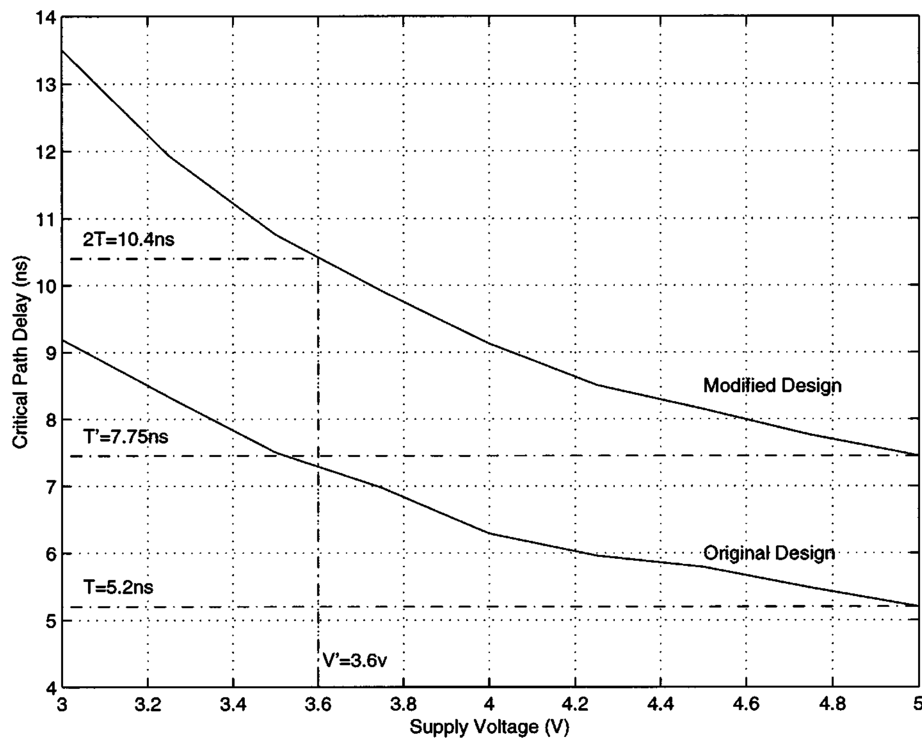

Fig. 15. Plot showing the variation of the critical paths of the normal and modified designs with supply voltage.

throughput is required, then either slower serial GF operators can be implemented or the GF operators may be re-used to do multiple operations [41] in both the normal and modified designs.

VI. Conclusion

In this paper, we have developed a low-power/high-speed Berlekamp algorithm that enables low-power/high-speed operations. We showed that similar modifications can be derived for both the errors-only decoding as well as the errors-and-erasures decoding. Our algorithm-level approaches expose additional parallelism that enable us to design a low-power RS decoder. Architecture level approaches were proposed for the syndrome and error computations. An architecture was proposed for the Berlekamp algorithm that takes advantage of the algorithm-level transformations. The impact of the algorithm and architecture level approach was evaluated by designing two decoders for a $(63, 57)$ RS code- one based on the normal algorithm and the other based on the low-power algorithm. Results showed that a speedup of 1.34 or power saving of 40% can be obtained. This validates our claim that algorithm level transformations, when intelligently applied, can have a strong impact on the power consumption of a design.

References

[1] S. Whitaker, J. Canaris, and K. Cameron, "Reed solomon VLSI codec for advanced television," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 1, pp. 230–236, June 1991.

[2] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[3] T. S. Rappaport, *Wireless Communications*. New York: IEEE Press, 1996.

[4] K. Maxwell, "Asymmetric digital subscriber line," *IEEE Commun. Mag.*, vol. 34, pp. 100–107, Oct. 1996.

[5] J. B. Cain and D. N. McGregor, "A recommended error control architecture for atm networks with wireless links," *IEEE J. Select. Areas Commun.*, vol. 15, pp. 16–27, Jan. 1997.

[6] R. D. Cideciyan and E. Eleftheriou, "Concatenated reed-solomon/convolutional coding scheme for data transmission in CDMA-based cellular systems," *IEEE Trans. Commun.*, vol. 45, pp. 1291–1303, Oct. 1997.

[7] W. W. Peterson, "Encoding and error-correction procedures for the Bose-Chaudhuri codes," *IRE Trans. Inform. Theory*, vol. IT-6, pp. 459–470, Sept. 1960.

[8] D. Gorenstein and N. Zierler, "A class of error-correcting codes in p^m symbols," *J. Soc. Ind. Applied Mathem.*, vol. 9, pp. 207–214, June 1961.

[9] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

[10] J. L. Massey, "Shift register synthesis and BCH coding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122–127, Jan. 1969.

[11] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A method for solving key equation for decoding goppa codes," *Inform. Control*, vol. 27, pp. 87–99, 1975.

[12] R. E. Blahut, *Algebraic Methods for Signal Processing and Error Control Coding*. New York: Springer-Verlag, 1992.

[13] ——, "Transform techniques for error control codes," *IBM J. Res. and Devel.*, vol. 23, pp. 299–315, May 1979.

[14] R. T. Chien, "Cyclic decoding procedure for the Bose-Chaudhuri-Hocquenghem codes," *IEEE Trans. Inform. Theory*, vol. IT-10, pp. 357–363, Oct. 1964.

[15] G. D. Forney Jr., "On decoding BCH codes," *IEEE Trans. Inform. Theory*, vol. IT-11, pp. 549–557, Oct. 1965.

[16] K. Y. Liu, "Architecture for VLSI design of Reed-Solomon decoders," *IEEE Trans. Comput.*, vol. C-33, pp. 178–189, Feb. 1984.

[17] H. M. Shao, T. K. Truong, L. J. Deutsch, J. H. Yuen, and I. S. Reed, "A VLSI design of a pipeline reed-solomon decoder," *IEEE Trans. Comput.*, vol. C-34, pp. 393–402, May 1985.

[18] N. Demassieux, F. Jutand, and M. Muller, "A 10 MHz (255, 233) Reed-Solomon decoder," in *Proc. IEEE 1988 Custom Integrated Circuits Conf.*, 1988, pp. 17.6.1–17.6.4.

[19] H. M. Shao and I. S. Reed, "On the VLSI design of a pipeline Reed-Solomon decoder using systolic arrays," *IEEE Trans. Comput.*, vol. 37, pp. 1273–1280, Oct. 1988.

[20] P. Tong, "A 40 MHz encoder-decoder chip generated by a reed-solution code compiler," in *Proc. Custom Integrated Circuits Conf.*, Boston, MA, May 1990, pp. 13.5.1–13.5.4.

[21] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Applications*. New York: IEEE Press, 1994.
[22] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for polynomial GCD computation," *IEEE Trans. Comput.*, vol. C-33, pp. 731–736, Aug. 1984.
[23] R. E. Blahut, "A universal reed-solomon decoder," *IBM J. Res. and Devel.*, vol. 28, pp. 150–158, Mar. 1984.
[24] Y. R. Shayan, T. Le-Ngoc, and V. K. Bhargava, "A versatile time domain Reed-Solomon decoder," *IEEE J. Select. Areas Commun.*, vol. 8, pp. 1535–1542, Oct. 1990.
[25] S. Choomchuay and B. Arambepola, "Time domain algorithms and architectures for reed-solomon decoding," *Proc. Inst. Elect. Eng. I, Commun., Speech and Vis.*, vol. 140, pp. 189–196, June 1993.
[26] J.-M. Hsu and C.-L. Wang, "An area-efficient pipelined VLSI architecture for decoding of Reed-Solomon codes based on a time-domain algorithm," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 7, pp. 864–871, Dec. 1997.
[27] C. Leiserson and J. Saxe, "Optimizing synchronous systems," *J. VLSI and Comput. Syst.*, vol. 1, no. 1, pp. 41–67, 1983.
[28] M. Potkonjak and J. Rabaey, "Optimizing resource utilization using transformations," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 277–292, Mar. 1994.
[29] K. K. Parhi, "High-level algorithm and arhitecture transformations for DSP synthesis," *J. VLSI Signal Processing*, vol. 9, pp. 121–143, Jan. 1995.
[30] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proc. IEEE*, vol. 83, pp. 498–523, Apr. 1995.
[31] K. Liu, A.-Y. Wu, A. Raghupathy, and J. Chen, "Algorithm-based low-power and high-performance multimedia signal processing," *Proc IEEE*, vol. 86, pp. 1155–1202, June 1998.
[32] K. K. Parhi and D. Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters—Part I: Pipeling using scattered look-ahead and decomposition," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1099–1117, July 1989.
[33] K. K. Parhi, C. Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectores," *IEEE J. Solid-State Circuits*, vol. 27, pp. 29–43, Jan. 1992.
[34] N. Shanbhag and M. Goel, "Low-power adaptive filter architectures and their application to 51.84 mb/s atmlan," *Signal Processing*, vol. 45, pp. 1276–1290, May 1997.
[35] A.-Y. Wu, K. Liu, Z. Zhang, K. Nakajima, A. Raghupathy, and S.-C. Liu, "Algorithm-based low power DSP design: Methodology and verification," in *VLSI Signal Processing VIII*, T. Nishitani and K. Parhi, Eds. New York: IEEE Press, 1995, pp. 277–286.
[36] A.-Y. Wu, K. J. R. Liu, Z. Zhang, K. Nakajima, and A. Raghupathy, "Low-power design methodology for DSP systems using multirate approach," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 1996, pp. 292–295.
[37] A. Raghupathy, "Low Power and High Speed Algorithms and VLSI Architectures for Error Control Coding and Adaptive Video Scaling," Ph.D. dissertation, Univ. of Maryland, College Park, MD, 1998.
[38] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, MA: Addison-Wesley, 1993.
[39] A. Raghupathy and K. J. R. Liu, "Low power/high speed design of a Reed Solomon decoder," in *Proc. IEEE Int. Symp. Circuits and Systems*, June 1997.
[40] T. C. Bartee and D. I. Schneider, "Computation with finite fields," *Inform. Control*, vol. 6, pp. 79–98, Mar. 1963.
[41] Y. Im and O.-S. Kwon, "An advanced vlsi architecture of rs decoders for advanced tv," in *Proc. IEEE Int. Conf. Communications*, vol. 3, June 1997, pp. 1346–1350.

Arun Raghupathy (S'95–M'99) received the B.Tech. degree in electronics and communications engineering from the Indian Institute of Technology, Madras, India, in 1993, the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland at College Park in 1995 and 1998, respectively. His Ph.D. research focused on the development of techniques that enable the implementation of low-power high-performance VLSI signal processing systems.

Currently he is a Senior Engineer in the ASIC Department, Qualcomm, Inc., San Diego, CA, where he is involved in the development of modem ASICs for third-generation (3G) CDMA systems.

K. J. R. Liu (S'86–M'90–SM'93) received the B.S. degree from National Taiwan University in 1983 and the Ph.D. degree from the University of California, Los Angeles, in 1990, both in electrical engineering.

Since 1990, he has been with Electrical and Computer Engineering Department and Institute for Systems Research, University of Maryland at College Park, where he is a Professor. During his sabbatical leave in 1996-1997, he was Visiting Associate Professor at Stanford University, Stanford, CA. His research interests span broad aspects of signal processing, image/video processing, and communications, in which he has published over 200 papers.

Dr. Liu was an Associate Editor of IEEE TRANSACTIONS ON SIGNAL PROCESSING, a Guest Editor of special issues on Multimedia Signal Processing of the PROCEEDINGS OF THE IEEE, a Guest Editor of a Special Issue on Signal Processing for Wireless Communications of IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, and a Guest Editor of special issue on Multimedia Communications over Networks of IEEESIGNAL PROCESSING MAGAZINE. He currently serves as the Chair of Multimedia Signal Processing Technical Committee of IEEE Signal Processing Society, and a Guest Editor of a Special Issue on Multimedia Over IP of IEEE TRANSACTIONS ON MULTIMEDIA, an Editor of the *Journal of VLSI Signal Processing Systems*, and the Series Editor of Marcel Dekker series on signal processing and communications. He is the recipient of numerous awards, some of which include the 1994 National Science Foundation Young Investigator Award, the IEEE Signal Processing Society's 1993 Senior Award (Best Paper Award), the IEEE Benelux Joint Chapter on Vehicular Technology and Communications 1999 Award (Best Paper Award from IEEE VTC'99, Amsterdam), the 1994 George Corcoran Award for outstanding contributions to electrical engineering education and the 1995-1996 Outstanding Systems Engineering Faculty Award in recognition of outstanding contributions in interdisciplinary research, both from the University of Maryland, College Park.