High-Speed Architectures for Reed–Solomon Decoders

Dilip V. Sarwate, *Fellow, IEEE,* and Naresh R. Shanbhag, *Member, IEEE*

***Abstract*—New high-speed VLSI architectures for decoding Reed–Solomon codes with the Berlekamp–Massey algorithm are presented in this paper. The speed bottleneck in the Berlekamp–Massey algorithm is in the iterative computation of *discrepancies* followed by the updating of the error-locator polynomial. This bottleneck is eliminated via a series of algorithmic transformations that result in a fully systolic architecture in which a single array of processors computes both the error-locator and the error-evaluator polynomials. In contrast to conventional Berlekamp–Massey architectures in which the critical path passes through two multipliers and $1 + \lceil \log_2(t + 1) \rceil$ adders, the critical path in the proposed architecture passes through only one multiplier and one adder, which is comparable to the critical path in architectures based on the extended Euclidean algorithm. More interestingly, the proposed architecture requires approximately 25% fewer multipliers and a simpler control structure than the architectures based on the popular extended Euclidean algorithm. For block-interleaved Reed–Solomon codes, embedding the interleaver memory into the decoder results in a further reduction of the critical path delay to just one XOR gate and one multiplexer, leading to speed ups of as much as an order of magnitude over conventional architectures.**

***Index Terms*—Interleaved codes, Berlekamp–Massey algorithm, pipelined decoders, Reed–Solomon codes, systolic architectures.**

I. Introduction

REED–SOLOMON codes [1], [3] are employed in numerous communications systems such as those for deep space, digital subscriber loops, and wireless systems as well as in memory and data storage systems. Continual demand for ever higher data rates makes it necessary to devise very high-speed implementations of decoders for Reed–Solomon codes. Recently reported decoder implementations [5], [19] have quoted data rates of ranging from 144 Mb/s to 1.28 Gb/s. These high throughputs have been achieved by architectural innovations such as pipelining and parallel processing. A majority of the implementations [2], [8], [15], [19] employ an architecture based on the extended Euclidean (**eE**) algorithm for computing the greatest common divisor of two polynomials [3]. A key advantage of architectures based upon the **eE** algorithm is regularity. In addition, the critical path delay in these architectures is at best $T_{\rm mult} + T_{\rm add} + T_{\rm mux}$, where $T_{\rm mult}$, $T_{\rm add}$, and $T_{\rm mux}$ are the delays of the finite-field multiplier, adder, and 2×1 multiplexer respectively, and this is sufficiently small for most applications. In contrast, relatively few decoder implementations have employed architectures based on the Berlekamp–Massey (**BM**) algorithm [1], [3], [10], presumably because the architectures were found to be irregular and to have a longer critical path delay that was also dependent on the error-correcting capability of the code [5]. In this paper, we show that, in fact, it is possible to reformulate the **BM** algorithm to achieve *extremely* regular decoder architectures. Surprisingly, these new architectures cannot only operate at data rates comparable to architectures based on the **eE** algorithm, but they also have lower gate complexity and simpler control structures.

This paper begins with a brief tutorial overview of the encoding and decoding of Reed–Solomon codes in Section II. Conventional architectures for decoders based on the **BM** algorithm are described in Section III. In Section IV, we show that it is possible to algorithmically transform the **BM** algorithm so that a homogenous systolic array architecture for the decoder can be developed. Finally, in Section V, we describe a pipelined architecture for block-interleaved Reed–Solomon codes that achieves an order of magnitude reduction in the critical path delay over the architectures presented in Sections III and IV.

Manuscript received August 23, 1999; revised June 7, 2000. This work was supported in part by the National Science Foundation under Grant CCR 99-79381.

The authors are with the Coordinated Science Laboratory and the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2307 USA (e-mail: sarwate@comm.csl.uiuc.edu; shanbhag@uiuc.edu).

Publisher Item Identifier S 1063-8210(01)03895-1.

II. Reed–Solomon Codes

We provide a brief overview of the encoding and decoding of Reed–Solomon codes.

A. Encoding of Reed–Solomon Codes

Let $(d_{k-1}, d_{k-2}, \ldots, d_1, d_0)$ denote k m-bit *data symbols* (bytes) that are to be transmitted over a communication channel (or stored in memory). These bytes are regarded as elements of the finite field (also called Galois field) $\mathrm{GF}(2^m)$,[1] and encoded into a *codeword* $(c_{n-1}, c_{n-2}, \ldots, c_1, c_0)$ of $n > k$ bytes. These codeword symbols are transmitted over the communication channel (or stored in memory).

For Reed–Solomon codes over $\mathrm{GF}(2^m)$, $n = 2^m - 1$, k is odd, and the code can correct $t = (n - k)/2$ byte errors. The encoding process is best described in terms of the *data polynomial* $D(z) = d_{k-1}z^{k-1} + d_{k-2}z^{k-2} + \cdots + d_1 z + d_0$ being transformed into a *codeword polynomial* $C(z) = c_{n-1}z^{n-1} + c_{n-2}z^{n-2} + \cdots + c_1 z + c_0$. All codeword polynomials $C(z)$ are polynomial multiples of $G(z)$, the *generator polynomial* of the code, which is defined as

$$G(z) = \prod_{i=0}^{2t-1} (z - \alpha^{m_0+i}) \tag{1}$$

[1]Addition (and subtraction) in $\mathrm{GF}(2^m)$ is the bit-by-bit XOR of the bytes. The $2^m - 1$ nonzero elements of $\mathrm{GF}(2^m)$ can also be regarded as the powers, α^i, $0 \le i \le 2^m - 2$, of a *primitive element* α (where $\alpha^{2^m-1} = 1 = \alpha^0$) so that the product of field elements α^i and α^j is $\alpha^i \cdot \alpha^j = \alpha^{i+j} = \alpha^l$ where $l \equiv i + j \bmod (2^m - 1)$.

where m_0 is typically zero or one. However, other choices sometimes simplify the decoding process slightly. Since $2t$ consecutive powers $\alpha^{m_0}, \alpha^{m_0+1}, \dots, \alpha^{m_0+2t-1}$ of α are roots of $G(z)$, and $C(z)$ is a multiple of $G(z)$, it follows that

$$C(\alpha^{m_0+i}) = 0, \quad 0 \le i \le 2t-1 \tag{2}$$

for all codeword polynomials $C(z)$. In fact, an arbitrary polynomial of degree less than n is a codeword polynomial if and only if it satisfies (2).

A *systematic* encoding produces codewords that are comprised of data symbols followed by *parity-check symbols* and is obtained as follows. Let $Q(z)$ and $P(z)$ denote the quotient and remainder respectively when the polynomial $z^{n-k}D(z)$ of degree $n-1$ is divided by $G(z)$ of degree $2t = n-k$. Thus, $z^{n-k}D(z) = Q(z)G(z) + P(z)$ where $\deg P(z) < n-k$. Clearly, $Q(z)G(z) = z^{n-k}D(z) - P(z) = C(z)$ is a multiple of $G(z)$. Furthermore, since the lowest degree term in $z^{n-k}D(z)$ is $d_0 z^{n-k}$ while $P(z)$ is of degree at most $n-k-1$, it follows that the codeword is given by

$$\begin{aligned}&(c_{n-1}, c_{n-2}, \dots, c_1, c_0)\\ &\quad = (d_{k-1}, d_{k-2}, \dots, d_1, d_0,\\ &\qquad - p_{n-k-1}, -p_{n-k-2}, \dots, -p_1, -p_0)\end{aligned}$$

and consists of the data symbols followed by the parity-check symbols.

B. Decoding of Reed–Solomon Codes

Let $C(z)$ denote the transmitted codeword polynomial and let $R(z)$ denote the received word polynomial. The input to the decoder is $R(z)$, and it assumes that

$$R(z) = C(z) + E(z),$$

where, if $e > 0$ errors have occurred during transmission, the error polynomial $E(z)$ can be written as

$$E(z) = Y_1 z^{i_1} + Y_2 z^{i_2} + \cdots + Y_e z^{i_e}.$$

It is conventional to say that the *error values* $Y_1, Y_2, \dots, Y_e$ have occurred at the *error locations* $X_1 = \alpha^{i_1}, X_2 = \alpha^{i_2}, \dots, X_e = \alpha^{i_e}$. Note that the decoder does not know $E(z)$; in fact, it does not even know the value of e. The decoder's task is to determine $E(z)$ from its input $R(z)$, and thus correct the errors by subtracting off $E(z)$ from $R(z)$. If $e \le t$, then such a calculation is always possible, that is, t or fewer errors can always be corrected.

The decoder begins its task of error correction by computing the *syndrome values*

$$s_i = R(\alpha^{m_0+i}) = C(\alpha^{m_0+i}) + E(\alpha^{m_0+i}) = E(\alpha^{m+i}) \qquad 0 \le i \le 2t-1. \tag{3}$$

If all $2t$ syndrome values are zero, then $R(z)$ is a codeword and it is assumed that $C(z) = R(z)$, that is, no errors have occurred. Otherwise, the decoder knows that $e > 0$ and uses the *syndrome polynomial* $S(z)$, which is defined to be

$$S(z) = s_0 + s_1 z + \cdots + s_{2t-1} z^{2t-1}$$

to calculate the error values and error locations. Define the *error locator* polynomial $\Lambda(z)$ of degree e and the *error evaluator* polynomial $\Omega(z)$ of degree at most $e-1$ to be

$$\Lambda(z) = \prod_{j=1}^{e}(1 - X_j z) = 1 + \lambda_1 z + \lambda_2 z^2 + \cdots + \lambda_e z^e \tag{4}$$

$$\begin{aligned}\Omega(z) &= \sum_{i=1}^{e} Y_i X_i^{m_0} \prod_{j=1, j\ne i}^{e} (1 - X_j z)\\ &= \omega_0 + \omega_1 z + \omega_2 z^2 + \cdots + \omega_{e-1} z^{e-1}.\end{aligned} \tag{5}$$

These polynomials are related to $S(z)$ through the *key equation* [1], [3]:

$$\Lambda(z)S(z) \equiv \Omega(z) \bmod z^{2t}. \tag{6}$$

Solving the key equation to determine both $\Lambda(z)$ and $\Omega(z)$ from $S(z)$ is the hardest part of the decoding process. The **BM** algorithm (to be described in Section III) and the **eE** algorithm can be used to solve (6). If $e \le t$, these algorithms find $\Lambda(z)$ and $\Omega(z)$, but if $e > t$, then the algorithms almost always fail to find $\Lambda(z)$ and $\Omega(z)$. Fortunately, such failures are usually easily detected.

Once $\Lambda(z)$ and $\Omega(z)$ have been found, the decoder can find the error locations by checking whether $\Lambda(\alpha^{-j}) = 0$ for each j, $0 \le j \le n-1$. Usually, the decoder computes the value of $\Lambda(\alpha^{-j})$ just before the j-th received symbol r_j leaves the decoder circuit. This process is called a *Chien search* [1], [3]. If $\Lambda(\alpha^{-j}) = 0$, then α^j is one of the error locations (say X_i). In other words, r_j is in error, and needs to be corrected before it leaves the decoder. The decoder can calculate the error value Y_i to be subtracted from r_j via Forney's error value formula [3]

$$Y_i = -\frac{X_i^{-(m_0-1)}\Omega\left(X_i^{-1}\right)}{\Lambda'\left(X_i^{-1}\right)} = -\left.\frac{z^{m_0}\Omega(z)}{z\Lambda'(z)}\right|_{z=\alpha^{-j}} \tag{7}$$

where $\Lambda'(z) = \lambda_1 + 2\lambda_2 z + 3\lambda_3 z^2 + \cdots$ denotes the formal derivative of $\Lambda(z)$. Note that the formal derivative simplifies to $\Lambda'(z) = \lambda_1 + \lambda_3 z^2 + \cdots$ since we are considering codes over GF(2^m). Thus, $z\Lambda'(z) = \lambda_1 z + \lambda_3 z^3 + \cdots$, which is just the terms of odd degree in $\Lambda(z)$. Hence, the value of $z\Lambda'(z)$ at $z = \alpha^{-j}$ can be found during the evaluation of $\Lambda(z)$ at $z = \alpha^{-j}$ and does not require a separate computation. Note also that (7) can be simplified by choosing $m_0 = 0$.

C. Reed–Solomon Decoder Structure

In summary, a Reed–Solomon decoder consists of three blocks:

- the syndrome computation (**SC**) block;
- the key-equation solver (**KES**) block;
- the Chien search and error evaluator (**CSEE**) block.

These blocks usually operate in pipelined mode in which the three blocks are separately and simultaneously working on three successive received words. The **SC** block computes the syndromes via (3) usually as the received word is entering the decoder. The syndromes are passed to the **KES** block which solves (6) to determine the error locator and error evaluator polynomials. These polynomials are then passed to the **CSEE** block, which calculates the error locations and error values via (7) and corrects the errors as the received word is being read out of the decoder.

The throughput bottleneck in Reed–Solomon decoders is in the **KES** block which solves (6): in contrast, the **SC** and **CSEE** blocks are relatively straightforward to implement. Hence, in this paper we focus on developing high-speed architectures for the **KES** block. As mentioned earlier, the key equation (6) can be solved via the **eE** algorithm (see [19] and [17] for implementations), or via the **BM** algorithm (see [5] for implementations). In this paper, we develop high-speed architectures for a reformulated version of the **BM** algorithm because we believe that this reformulated algorithm can be used to achieve much higher speeds than can be achieved by other implementations of the **BM** and **eE** algorithms. Furthermore, as we shall show in Section IV-B4, these new architectures also have lower gate complexity and a simpler control structure than architectures based on the **eE** algorithm.

III. Existing Berlekamp–Massey (BM) Architectures

In this section, we give a brief description of different versions of the Berlekamp–Massey (**BM**) algorithm and then discuss a generic architecture, similar to that in the paper by Reed *et al.* [13] for implementation of the algorithm.

A. The Berlekamp–Massey Algorithm

The **BM** algorithm is an iterative procedure for solving (6). In the form originally proposed by Berlekamp [1], the algorithm begins with polynomials $\Lambda(0,z)=1$, $\Omega(0,z)=0$ and iteratively determines polynomials $\Lambda(r,z)$ and $\Omega(r,z)$ satisfying the polynomial congruence

$$\Lambda(r,z)S(z)\equiv\Omega(r,z) \bmod z^r$$

for $r=1,2,\ldots 2t$ and, thus, obtains a solution $\Lambda(2t,z)$ and $\Omega(2t,z)$ to the key equation (6). Two "scratch" polynomials $B(r,z)$ and $H(r,z)$ with initial values $B(0,z)=1$ and $H(0,z)=-1$ are used in the algorithm. For each successive value of r, the algorithm determines $\Lambda(r,z)$ and $B(r,z)$ from $\Lambda(r-1,z)$ and $B(r-1,z)$. Similarly, the algorithm determines $\Omega(r,z)$ and $H(r,z)$ from $\Omega(r-1,z)$ and $H(r-1,z)$. Since $S(z)$ has degree $2t-1$, and the other polynomials can have degrees as large as t, the algorithm needs to store roughly $6t$ field elements. If each iteration is completed in one clock cycle, then $2t$ clock cycles are needed to find the error-locator and error-evaluator polynomials.

In recent years, most researchers have used the formulation of the **BM** algorithm given by Blahut [3] in which only $\Lambda(r,z)$ and $B(r,z)$ are computed iteratively. Following the completion of the $2t$ iterations, the error-evaluator polynomial $\Omega(2t,z)$ is computed as the terms of degree $t-1$ or less in the polynomial product $\Lambda(2t,z)S(z)$. An implementation of this version thus needs to store only $4t$ field elements, but the computation of $\Omega(2t,z)$ requires an additional t clock cycles. Although this version of the **BM** algorithm trades off space against time, it also suffers from the same problem as the Berlekamp version, viz. during some of the iterations, it is necessary to divide each coefficient of $\Lambda(r,z)$ by a quantity δ_r. These divisions are most efficiently handled by first computing δ_r^{-1}, the inverse of δ_r, and then multiplying each coefficient of $\Lambda(r,z)$ by δ_r^{-1}. Unfortunately, regardless of whether this method is used or whether one constructs separate divider circuits for each coefficient of $\Lambda(r,z)$, these divisions, which occur inside an iterative loop, are more time consuming than multiplications. Obviously, if these divisions could be replaced by multiplications, the resulting circuit implementation would have a smaller critical path delay and higher clock speeds would be usable.[2] A less well-known version of the **BM** algorithm [4], [13], has precisely this property and has been recently employed in practice [13], [5]. We focus on this version of the **BM** algorithm in this paper.

The inversionless **BM** (**iBM**) algorithm is described by the pseudocode shown below. The **iBM** algorithm actually finds scalar multiples $\beta\cdot\Lambda(z)$ and $\beta\cdot\Omega(z)$ instead of the $\Lambda(z)$ and $\Omega(z)$ defined in (4) and (5). However, it is obvious that the Chien search will find the same error locations and it follows from (7) that the same error values are obtained. Hence, we continue to refer to the polynomials computed by the **iBM** algorithm as $\Lambda(z)$ and $\Omega(z)$. As a minor implementation detail, $\lambda_0=1$ in (4) and thus requires no latches for storage, but the **iBM** algorithm must store $\lambda_0=\beta$. Note also that $b_{-1}(r)$ which occurs in Steps **iBM.2** and **iBM.3** is a constant: it has value zero for all r.

```
The iBM Algorithm
 Initialization:
  $\lambda_0(0)=b_0(0)=1, \lambda_i(0)=b_i(0)=0$ for $i=1,2,\ldots,t$.  $k(0)=0$.  $\gamma(0)=1$.
 Input: $s_i,\ i=0,1,\ldots,2t-1$.
    for $r=0$ step 1 until $2t-1$ do
       begin
          Step iBM.1   $\delta(r)=s_r\cdot\lambda_0(r)+s_{r-1}\cdot\lambda_1(r)+\cdots+s_{r-t}\cdot\lambda_t(r)$
          Step iBM.2   $\lambda_i(r+1)=\gamma(r)\cdot\lambda_i(r)-\delta(r)b_{i-1}(r),\ (i=0,1,\ldots,t)$
          Step iBM.3   if $\delta(r)\neq 0$ and $k(r)\geq 0$
                         then
                           begin
                              $b_i(r+1)=\lambda_i(r),\quad (i=0,1,\ldots,t)$
                              $\gamma(r+1)=\delta(r)$
                              $k(r+1)=-k(r)-1$
                           end
                         else
                           begin
                              $b_i(r+1)=b_{i-1}(r),\ (i=0,1,\ldots,t)$
                              $\gamma(r+1)=\gamma(r)$
                              $k(r+1)=k(r)+1$
                           end
       end
    for $i=0$ step 1 until $t-1$ do
       Step iBM.4   $\omega_i(2t)=s_i\cdot\lambda_0(2t)+s_{i-1}\cdot\lambda_1(2t)+\cdots+s_0\cdot\lambda_i(2t)$
 Output: $\lambda_i(2t),\ i=0,1,\ldots,t.\quad \omega_i(2t),\ i=0,1,\ldots,t-1$.
```

For $r<t$, Step **iBM.1** includes terms $s_{-1}\cdot\lambda_{r+1}(r), s_{-2}\cdot\lambda_{r+2}(r),\ldots,s_{r-t}\cdot\lambda_t(r)$ involving unknown quantities $s_{-1},s_{-2},\ldots,s_{r-t}$. Fortunately, it is known [3] that $\deg\Lambda(r,z)\leq r$ so that $\lambda_{r+1}(r)=\lambda_{r+2}(r)=\cdots=\lambda_t(r)=0$

[2]The astute reader will have noticed that the Forney error value formula (7) also involves a division. Fortunately, these divisions can be *pipelined* because they are *feed-forward* computations. Similarly, the polynomial evaluations needed in the **CSEE** block (as well as those in the **SC** block) are feed-forward computations that can be pipelined. Unfortunately, the divisions in the **KES** block occur inside an iterative loop and, hence, pipelining the computation becomes difficult. Thus, as was noted in Section II, the throughput bottleneck is in the **KES** block.

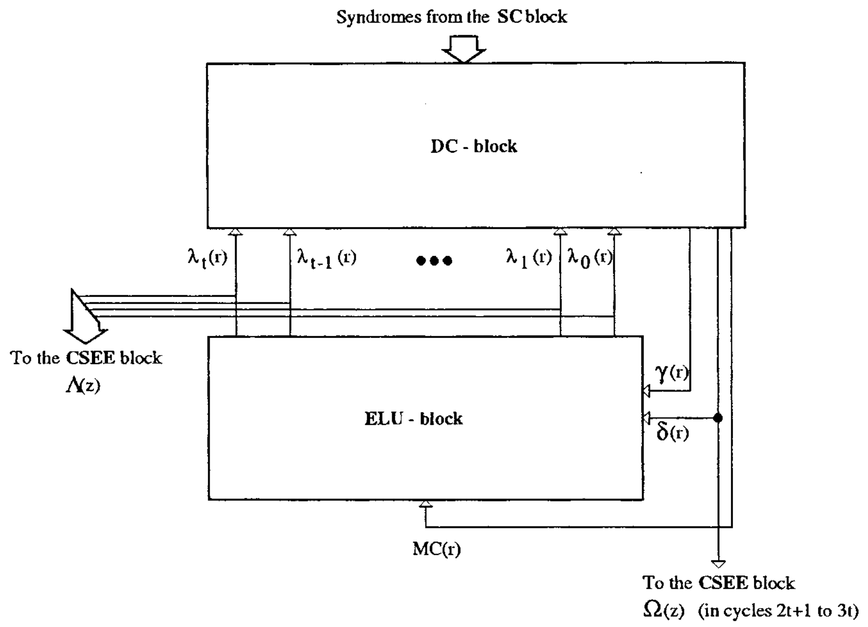


Fig. 1. The **iBM** architecture.

and therefore the unknown s_i do not affect the value of $\delta(r)$. Notice also the similarity between Steps **iBM.1** and **iBM.4**. These facts have been used to simplify the architecture that we describe next.

B. Architectures Based on the iBM Algorithm

Due to the similarity of Steps **iBM.1** and **iBM.4**, architectures based on the **iBM** algorithm need only two major computational structures as shown in Fig. 1.

- The *discrepancy computation* (**DC**) block for implementing Step **iBM.1**.
- The *error locator update* (**ELU**) block which implements Steps **iBM.2** and **iBM.3** in parallel.

The **DC** block contains latches for storing the syndromes s_i, the GF(2^m) arithmetic units for computing the discrepancy $\delta(r)$ and the control unit for the entire architecture. It is connected to the **ELU** block, which contains latches for storing for $\Lambda(r,z)$ and $B(r,z)$ as well as GF(2^m) arithmetic units for updating these polynomials, as shown in Fig. 1. During a clock cycle, the **DC** block computes the discrepancy $\delta(r)$ and passes this value together with $\gamma(r)$ and a control signal $\mathrm{MC}(r)$ to the **ELU** block which updates the polynomials during the same clock cycle. Since all GF(2^m) arithmetic operations are completed in one clock cycle, we assume that m-bit parallel arithmetic units are being employed. Architectures for such Galois field arithmetic units can be found in numerous references including [7] and will not be discussed here.

1) DC Block Architecture: The **DC** block architecture shown in Fig. 2 has $2t$ latches constituting the DS shift register that are initialized such that the latches $\mathrm{DS}_1, \mathrm{DS}_2, \ldots, \mathrm{DS}_{2t-1}, \mathrm{DS}_0$ contain the syndromes $s_1, s_2, \ldots, s_{2t-1}, s_0$, respectively. In each of the first $2t$ clock cycles, the $t+1$ multipliers compute the products in Step **iBM.1**. These are added in a binary adder tree of depth $\lceil \log_2(t+1) \rceil$ to produce the discrepancy $\delta(r)$. Thus, the delay in computing $\delta(r)$ is $T_\delta = T_{\mathrm{mult}} + \lceil \log_2(t+1) \rceil \cdot T_{\mathrm{add}}$.

A typical control unit such as the one illustrated in Fig. 2 has counters for the variables r and $k(r)$, and storage for $\gamma(r)$. Following the computation of $\delta(r)$, the control unit computes the OR of the m bits in $\delta(r)$ to determine whether $\delta(r)$ is nonzero. This requires $m-1$ two-input OR gates arranged in a binary tree of depth $\lceil \log_2 m \rceil$. If the counter for $k(r)$ is implemented in twos-complement representation, then $k(r) \geq 0$ if and only if the most significant bit in the counter is 0. The delay in generating signal $\mathrm{MC}(r)$ is thus $T_{\mathrm{MC}} = T_\delta + \lceil \log_2 m \rceil \cdot T_{\mathrm{or}} + T_{\mathrm{and}}$. Finally, once the $\mathrm{MC}(r)$ signal is available, the counter for $k(r)$ can be updated. Notice that a twos-complement arithmetic addition is needed if $k(r+1) = k(r)+1$. On the other hand, negation in two's-complement representation complements all the bits and then adds one and, hence, the update $k(r+1) = -k(r) - 1$ requires only the complementation of all the bits in the $k(r)$ counter. We note that it is possible to use *ring counters* for r and $k(r)$, in which case $k(r)$ is updated just T_{mux} seconds after the $\mathrm{MC}(r)$ signal has been computed.

Following the $2t$ clock cycles for the **BM** algorithm, the **DC** block computes the error-locator polynomial $\Omega(z)$ in the next t clock cycles. To achieve this, the $\mathrm{DS}_t, \mathrm{DS}_{t+1}, \ldots, \mathrm{DS}_{2t-1}$ latches are reset to zero during the $2t$th clock cycle, so that, at the beginning of the $(2t+1)$-th clock cycle, the contents of the DS register (see Fig. 2) are $s_1, s_2, \ldots, s_{t-1}, 0, 0, \ldots, 0, s_0$. Also, the outputs of the **ELU** block are frozen so that these do not change during the computation of $\Omega(z)$. From Step **iBM.4**, it follows that the "discrepancies" computed during the next t clock cycles are just the coefficients $\omega_0(2t), \omega_1(2t), \ldots, \omega_{t-1}(2t)$ of $\Omega(z)$. The architecture in Fig. 2 is an enhanced version of the one described in [13]. The latter uses a slightly different structure and different initialization of the DS register in the **DC** block, which requires more storage and makes it less adaptable to the subsequent computation of the error-locator polynomial.

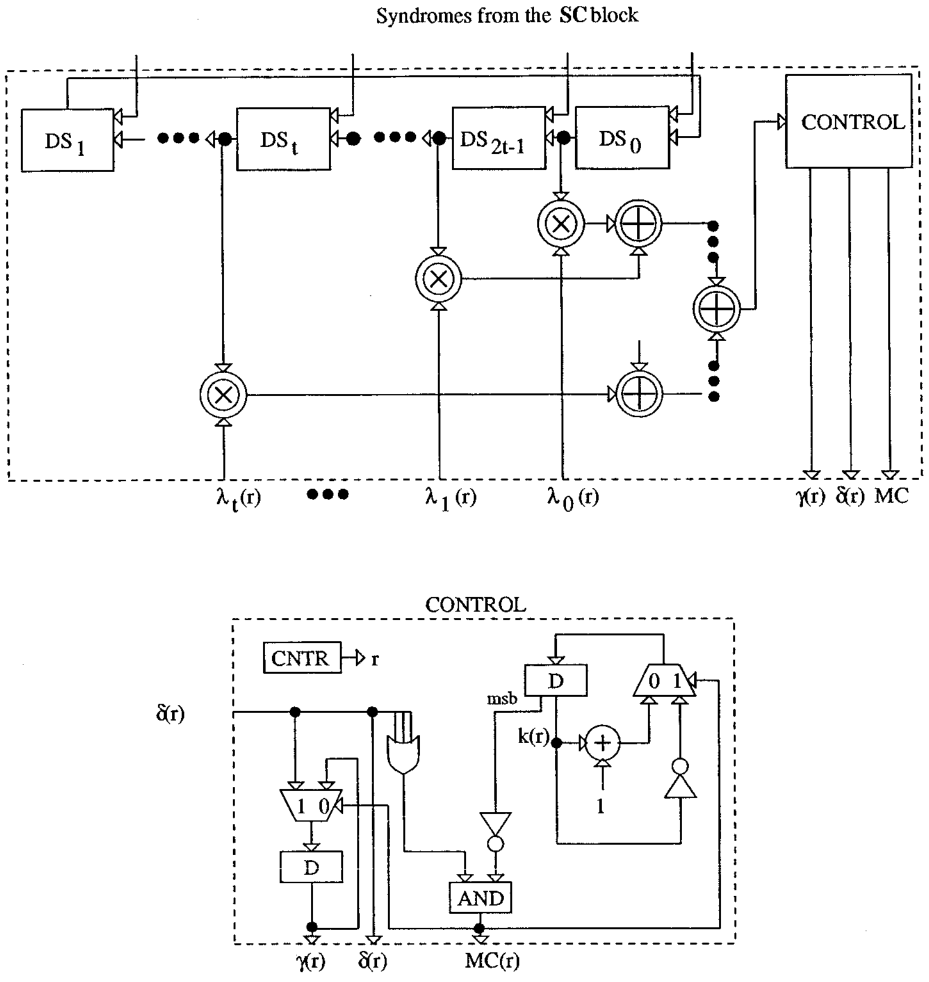


Fig. 2. The discrepancy computation (**DC**) block.

Note that the total hardware requirements of the **DC** block are $2t$ m-bit latches, $t+1$ multipliers, t adders, and miscellaneous other circuitry (counters, arithmetic adder or ring counter, OR gates, inverters and latches), in the control unit. From Fig. 2, the critical path delay of the **DC** block is

$$T_{\mathrm{DC}} = T_{\mathrm{mult}} + (1 + \lceil \log_2(t+1) \rceil) \cdot T_{\mathrm{add}} + \lceil \log_2 m \rceil \cdot T_{\mathrm{or}} + T_{\mathrm{and}}.$$

2) ELU Block Architecture: Following the computation of the discrepancy $\delta(r)$ and the $\mathrm{MC}(r)$ signal in the **DC** block, the polynomial coefficient updates of Steps **iBM.2** and **iBM.3** are performed simultaneously in the **ELU** block. The processor element **PE0** (hereinafter the **PE0** *processor*) that updates one coefficient of $\lambda(z)$ and $B(z)$ is illustrated in Fig. 3(a). The complete **ELU** architecture is shown in Fig. 3(b), where we see that signals $\delta(r)$, $\gamma(r)$, and $\mathrm{MC}(r)$ are broadcast to all the **PE0** processors. In addition, the latches in all the **PE0** processors are initialized to zero except for $\mathbf{PE0}_0$, which has its latches initialized to the element $1 \in \mathrm{GF}(2^m)$. Notice that $2t+2$ latches and multipliers, and $t+1$ adders and multiplexers are needed. The critical path delay of the **ELU** block is given by

$$T_{\mathrm{ELU}} = T_{\mathrm{mult}} + T_{\mathrm{add}}.$$

3) iBM Architecture: Ignoring the hardware used in the control section, the total hardware needed to implement the **iBM**

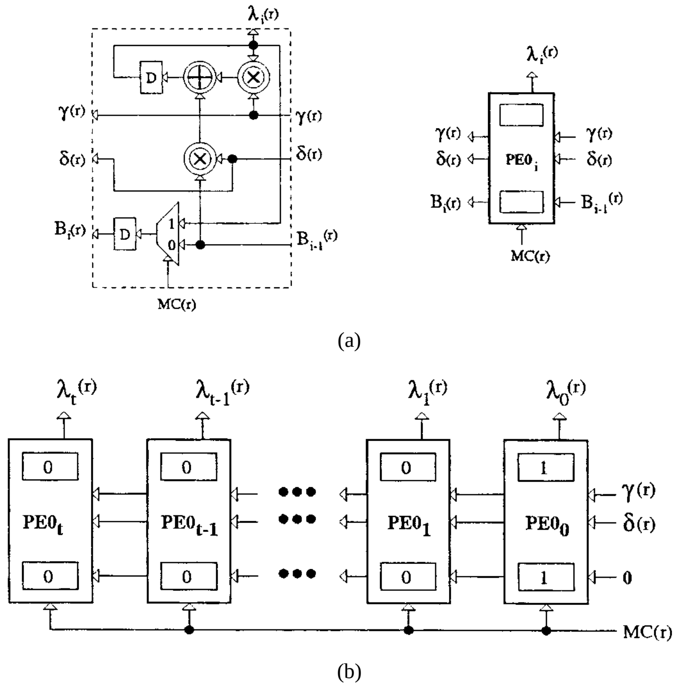


Fig. 3. The **ELU** block diagram. (a) The **PE0** processor. (b) The **ELU** architecture. The latches in $\mathbf{PE0}_0$ are initialized to $1 \in \mathrm{GF}(2^m)$ and those in other **PE0**s are initialized to zero.

algorithm is $4t+2$ latches, $3t+3$ multipliers, $2t+1$ adders, and $t+1$ multiplexers. The total time required to solve the key equation for one codeword is $3t$ clock cycles. Alternatively, if $\Omega(2t, z)$ is computed iteratively, the computations require only $2t$ clock cycles. However, since the computations required to update $\Omega(r, z)$ are the same as that of $\Lambda(r, z)$, a near-duplicate of the **ELU** block is needed.[3] This increases the hardware requirements to $6t+2$ latches, $5t+3$ multipliers, $3t+1$ adders, and $2t+1$ multiplexers. In either case, the critical path delay of the **iBM** architecture can be obtained from Figs. 1, 2, and 3 as

$$T_{\text{iBM}} = 2 \cdot T_{\text{mult}} + (1 + \lceil \log_2(t+1) \rceil) \cdot T_{\text{add}} \tag{8}$$
$$> 2 \cdot (T_{\text{mult}} + T_{\text{add}}) \tag{9}$$

which is the delay of the direct path that begins in the **DC** block starting from the DS_i latches, through a multiplier, an adder tree of height $\lceil \log_2(t+1) \rceil$ (generating the signal $\delta(r)$), feeding into the **ELU** block multiplier and adder before being latched. We have assumed that the indirect path taken by $\delta(r)$ through the control unit (generating signal $\text{MC}(r)$) feeding into the **ELU** block multiplexer is faster than the direct path, i.e., $T_{\text{mult}} > \lceil \log_2 m \rceil \cdot T_{\text{or}} + T_{\text{and}}$. This is a reasonable assumption in most technologies. Note that more than half of T_{iBM} is due to the delay in the **DC** block, and that this contribution increases logarithmically with the error correction capability. Thus, reducing the delay in the **DC** block is the key to achieving higher speeds.

In the next section, we describe algorithmic reformulations of the **iBM** algorithm that lead to a systolic architecture for the **DC** block and reduce its critical path delay to T_{ELU}.

IV. Proposed Reed–Solomon Decoder Architectures

The critical path in **iBM** architectures of the type described in Section III passes through *two* multipliers as well as the adder tree structure in the **DC** block. The multiplier units contribute significantly to the critical path delay and hence reduce the throughput achievable with the **iBM** architecture. In this section, we propose new decoder architectures that have a smaller critical path delay. These architectures are derived via algorithmic reformulation of the **iBM** algorithm. This reformulated **iBM** (**riBM**) algorithm computes the *next* discrepancy $\delta(r+1)$ at the same time that it is computing the *current* polynomial coefficient updates, that is, the $\lambda_i(r+1)$'s and the $b_i(r+1)$'s. This is possible because the reformulated discrepancy computation does not use the $\lambda_i(r+1)$'s explicitly. Furthermore, the discrepancy is computed in a block which has the *same* structure as the **ELU** block, so that both blocks have the same critical path delay $T_{\text{mult}} + T_{\text{add}}$.

A. Reformulation of the iBM Algorithm

1) Simultaneous Computation of Discrepancies and Updates: Viewing Steps **iBM.2** and **iBM.3** in terms of polynomials, we see that Step **iBM.2** computes

$$\Lambda(r+1, z) = \gamma(r) \cdot \Lambda(r, z) - z \cdot \delta(r) \cdot B(r, z) \tag{10}$$

while Step **iBM.3** sets $B(r+1, z)$ either to $\Lambda(r, z)$ or to $z \cdot B(r, z)$. Next, note that the discrepancy $\delta(r)$ computed in Step **iBM.1** is actually $\delta_r(r)$, the coefficient of z^r in the polynomial product

$$\begin{aligned}\Lambda(r, z) \cdot S(z) &= \Delta(r, z) \\ &= \delta_0(r) + \delta_1(r) \cdot z + \cdots + \delta_r(r) \cdot z^r + \cdots.\end{aligned} \tag{11}$$

Much faster implementations are possible if the decoder computes *all* the coefficients of $\Delta(r, z)$ (and of $\Theta(r, z) = B(r, z) \cdot S(z)$) even though only $\delta_r(r)$ is needed to compute $\Lambda(r+1, z)$ and to decide whether $B(r+1, z)$ is to be set to $\Lambda(r, z)$ or to $z \cdot B(r, z)$.

Suppose that at the beginning of a clock cycle, the decoder has available to it all the coefficients of $\Delta(r, z)$ and $\Theta(r, z)$ (and, of course, of $\Lambda(r, z)$ and $B(r, z)$ as well). Thus, $\delta(r) = \delta_r(r)$ is available at the beginning of the clock cycle, and the decoder can compute $\Lambda(r+1, z)$ and $B(r+1, z)$. Furthermore, it follows from (10) and (11) that

$$\begin{aligned}\Delta(r+1)(z) &= \Lambda(r+1, z) \cdot S(z) \\ &= [\gamma(r) \cdot \Lambda(r, z) - z \cdot \delta_r(r) \cdot B(r, z)] \cdot S(z) \\ &= \gamma(r) \cdot \Delta(r, z) - z \cdot \delta_r(r) \cdot \Theta(r, z)\end{aligned}$$

while $\Theta(r+1, z) = B(r+1, z) \cdot S(z)$ is set to either $\Delta(r, z) = \Lambda(r, z) \cdot S(z)$ or to $z \cdot \Theta(r, z) = z \cdot B(r, z) \cdot S(z)$. In short, $\Delta(r+1, z)$ and $\Theta(r+1, z)$ are computed in *exactly* the same manner as are $\Lambda(r+1, z)$ and $B(r+1, z)$. Furthermore, all four polynomial updates can be computed simultaneously, and all the polynomial coefficients as well as $\delta_{r+1}(r+1)$ are thus available at the beginning of the *next* clock cycle.

2) A New Error-Evaluator Polynomial: The **riBM** algorithm simultaneously updates four polynomials $\Lambda(r, z)$, $B(r, z)$, $\Delta(r, z)$, and $\Theta(r, z)$ with initial values $\Lambda(0, z) = B(0, z) = 1$ and $\Delta(0, z) = \Theta(0, z) = S(z)$. The $2t$ iterations thus produce the error-locator polynomial $\Lambda(2t, z)$ and also the polynomial $\Delta(2t, z)$. Note that since $\Omega(2t, z) \equiv \Lambda(2t, z) \cdot S(z) \bmod z^{2t}$ it follows from (11) that the low-order coefficients of $\Delta(2t, z)$ are just $\Omega(2t, z)$, that is, the $2t$ iterations compute *both* the error-locator polynomial $\Lambda(2t, z)$ and the error-evaluator polynomial $\Omega(2t, z)$—the additional t iterations of Step **iBM.4** are not needed. The high-order coefficients of $\Delta(2t, z)$ can also be used for error evaluation. Let $\Delta(2t, z) = \Omega(2t, z) + z^{2t} \cdot \Omega^{(h)}(z)$, where $\Omega^{(h)}(z)$ of degree at most $e-1$ contains the high-order terms. Since X_i^{-1} is a root of $\Lambda(2t, z)$, it follows from (11) that $\Delta(2t, X_i^{-1}) = \Omega(2t, X_i^{-1}) + X_i^{-2t}\Omega^{(h)}(X_i^{-1}) = 0$. Thus, (7) can be rewritten as

$$Y_i = \frac{X_i^{-(m_0+2t-1)}\Omega^{(h)}\left(X_i^{-1}\right)}{\Lambda'\left(X_i^{-1}\right)} = \left.\frac{z^{m_0+2t}\Omega^{(h)}(z)}{z\Lambda'(z)}\right|_{z=X_i^{-1}}. \tag{12}$$

We next show that this variation of the error evaluation formula has certain architectural advantages. Note that the choice $m_0 = -2t = n - 2t$ is preferable if (12) is to be used.

3) Further Reformulation: Since the updating of all four polynomials is identical, the discrepancies can be calculated using an **ELU** block like the one described in Section III. Unfortunately, for $r = 0, 1, \ldots, 2t-1$, the discrepancy $\delta_r(r)$ is computed in processor $\mathbf{PE0}_r$. Thus, multiplexers are needed to route the appropriate latch contents to the control unit and

[3]Since $\deg \Omega(2t, z) < t$, the array has only t **PE0** processors.

to the **ELU** block that computes $\Lambda(r+1,z)$ and $B(r+1,z)$. Additional reformulation of the **iBM** algorithm, as described next, eliminates these multiplexers. We use the fact that for any $i < r$, $\delta_i(r)$ and $\theta_i(r)$ cannot affect the value of any later discrepancy $\delta_{r+j}(r+j)$. Consequently, we need not store $\delta_i(r)$ and $\theta_i(r)$ for $i < r$. Thus, for $r = 0, 1, \ldots, 2t-1$, define $\hat{\delta}_i(r) = \delta_{i+r}(r)$ and $\hat{\theta}_i(r) = \theta_{i+r}(r)$ and the polynomials

$$\hat{\Delta}(r,z) = \sum_{i=0}^{2t-1} \hat{\delta}_i(r) z^i \quad \text{and} \quad \hat{\Theta}(r,z) = \sum_{i=0}^{2t-1} \hat{\theta}_i(r) z^i$$

with initial values $\hat{\Delta}(0,z) = \hat{\Theta}(0,z) = S(z)$. It follows that these polynomial coefficients are updated as

$$\begin{aligned}\hat{\delta}_i(r+1) &= \delta_{i+1+r}(r+1)\\ &= \gamma(r)\delta_{i+1+r}(r) - \delta_r(r)\theta_{i+r}(r)\\ &= \gamma(r)\hat{\delta}_{i+1}(r) - \hat{\delta}_0(r)\hat{\theta}_i(r)\end{aligned}$$

while $\hat{\theta}_i(r+1) = \theta_{i+1+r}(r+1)$ is set either to $\delta_{i+1+r}(r) = \hat{\delta}_{i+1}(r)$ or to $\theta_{i+r}(r) = \hat{\theta}_i(r)$. Note that the discrepancy $\delta_r(r) = \hat{\delta}_0(r)$ is always in a fixed (zero-th) position with this form of update. As a final comment, note this form of update ultimately produces $\hat{\Delta}(2t,z) = \delta_{2t}(2t) + \delta_{2t+1}(2t)z + \cdots = \Omega^{(h)}(2t,z)$ and, thus, (12) can be used for error evaluation in the **CSEE** block.

The **riBM** algorithm is described by the following pseudocode. Note that $b_{-1}(r) = \hat{\delta}_{2t}(r) = 0$ for all values of r, and these quantities do not need to be stored or updated.

The riBM Algorithm

```
Initialization:
  $\lambda_0(0) = b_0(0) = 1, \lambda_i(0) = b_i(0) = 0$ for $i = 1,2,\ldots,t$.  $k(0) = 0$.  $\gamma(0) = 1$.
Input: $s_i$, $i = 0,1,\ldots,2t-1$.
  $\hat{\delta}_i(0) = \hat{\theta}_i(0) = s_i$,  $(i = 0,\ldots,2t-1)$
  for $r = 0$ step 1 until $2t-1$ do
    begin
      Step riBM.1  $\lambda_i(r+1) = \gamma(r)\cdot\lambda_i(r) - \hat{\delta}_0(r)\cdot b_{i-1}(r), (i = 0,1,\ldots,t)$
                   $\hat{\delta}_i(r+1) = \gamma(r)\cdot\hat{\delta}_{i+1}(r) - \hat{\delta}_0(r)\cdot\hat{\theta}_i(r), (i = 0,\ldots,2t-1)$
      Step riBM.2  if $\hat{\delta}_0(r) \neq 0$ and $k(r) \geq 0$
                   then
                     begin
                       $b_i(r+1) = \lambda_i(r)$,  $(i = 0,1,\ldots,t)$
                       $\hat{\theta}_i(r+1) = \hat{\delta}_{i+1}(r)$,  $(i = 0,1,\ldots,2t-1)$
                       $\gamma(r+1) = \hat{\delta}_0(r)$
                       $k(r+1) = -k(r) - 1$
                     end
                   else
                     begin
                       $b_i(r+1) = b_{i-1}(r)$,  $(i = 0,1,\ldots,t)$
                       $\hat{\theta}_i(r+1) = \hat{\theta}_i(r)$,  $(i = 0,1,\ldots,2t-1)$;
                       $\gamma(r+1) = \gamma(r)$
                       $k(r+1) = k(r) + 1$
                     end
    end
Output: $\lambda_i(2t)$,  $(i = 0,1,\ldots,t)$;  $\omega_i^{(h)}(2t) = \hat{\delta}_i(2t)$,  $(i = 0,1,\ldots,t-1)$
```

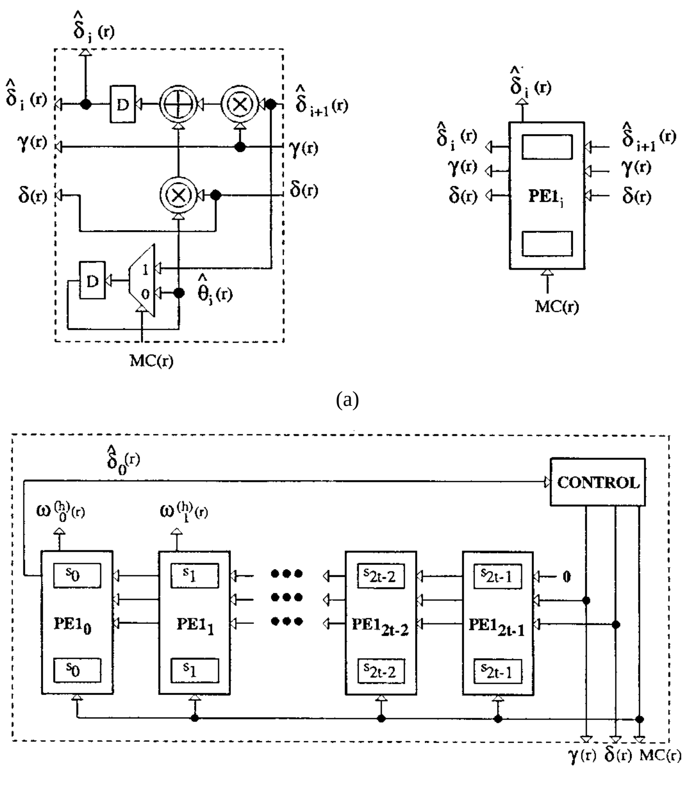


Fig. 4. The **rDC** block diagram. (a) The **PE1** processor. (b) The **rDC** architecture.

Next, we consider architectures that implement the **riBM** algorithm.

B. High-Speed Reed–Solomon Decoder Architectures

As in the **iBM** architecture described in Section III, the **riBM** architecture consists of a reformulated discrepancy computation (**rDC**) block connected to an **ELU** block.

1) The rDC Architecture: The **rDC** block uses the processor **PE1** shown in Fig. 4(a) and the **rDC** architecture shown in Fig. 4(b). Notice that processor **PE1** is very similar to processor **PE0** of Fig. 3(a). However, the contents of the upper latch "flow through" **PE1** while the contents of the lower latch "recirculate". In contrast, the lower latch contents "flow through" in processor **PE0** while the contents of the upper latch "recirculate". Obviously, the hardware complexity and the critical path delays of processors **PE0** and **PE1** are identical. Thus, assuming as before that $T_{\text{mult}} > \lceil \log_2 m \rceil \cdot T_{\text{or}} + T_{\text{and}}$, we get that $T_{\text{rDC}} = T_{\text{mult}} + T_{\text{add}}$. Note that the delay is independent of the error-correction capability t of the code.

The hardware requirements of the proposed architecture in Fig. 4 are $2t$ **PE1** processors, that is, $4t$ latches, $4t$ multipliers, $2t$ adders, and $2t$ multiplexers, in addition to the control unit, which is the same as that in Fig. 2.

2) The riBM Architecture: The overall **riBM** architecture is shown in Fig. 5. It uses the **rDC** block of Fig. 4 and the **ELU** block in Fig. 3. Note that the outputs of the **ELU** block do not feed back into the **rDC** block. Both blocks have the same critical path delay of $T_{\text{rDC}} = T_{\text{ELU}} = T_{\text{mult}} + T_{\text{add}}$ and since they

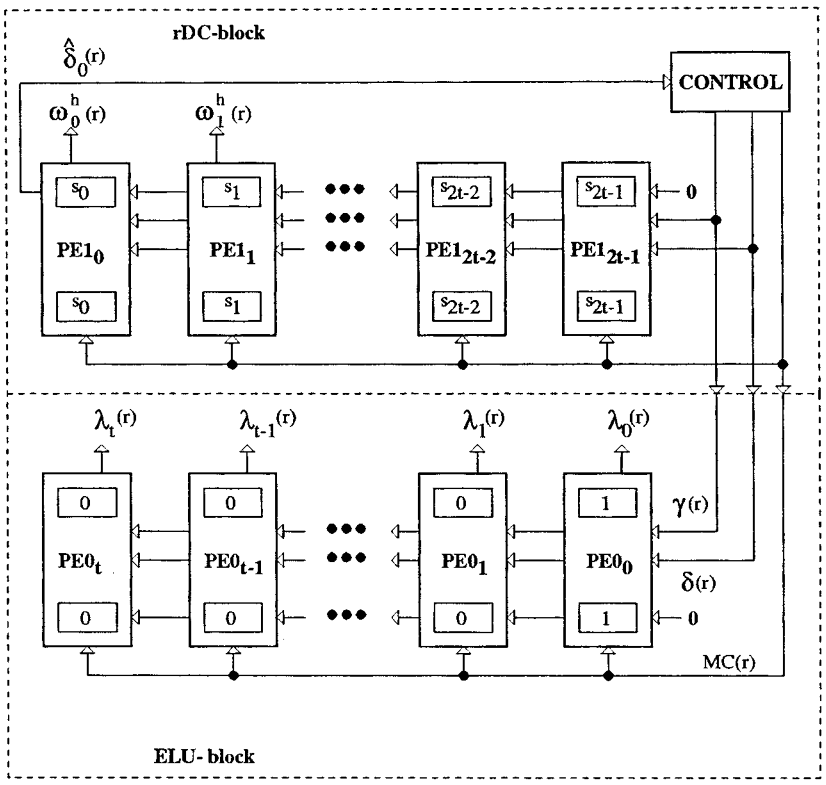


Fig. 5. The systolic **riBM** architecture.

operate in parallel, our proposed **riBM** architecture achieves the same critical path delay:

$$T_{\text{riBM}} = T_{\text{mult}} + T_{\text{add}}$$

which is less than half the delay $T_{\text{iBM}} = 2 \cdot T_{\text{mult}} + (1 + \lceil \log_2(t+1) \rceil) \cdot T_{\text{add}}$ of the enhanced **iBM** architecture.

As noted in the previous subsection, at the end of the $2t$-th iteration, the $\mathbf{PE1}_i$s, $i = 0, \ldots, t-1$ contain the coefficients of $\Omega^{(h)}(2t, z)$ which can be used for error evaluation. Thus, $2t$ clock cycles are used to determine both $\Lambda(z)$ and $\Omega^{(h)}(z)$ as needed in (12). Ignoring the control unit, the hardware requirement of this architecture is $3t+1$ processors, that is, $6t+2$ latches, $6t+2$ multipliers, $3t+1$ adders, and $3t+1$ multiplexers. This compares very favorably with the $6t+2$ latches, $5t+3$ multipliers, $3t+1$ adders, and $2t+1$ multiplexers needed to implement the enhanced **iBM** architecture of Section III in which both the error-locator and the error-evaluator polynomial are computed in $2t$ clock cycles. Using only $t-1$ additional multipliers and t additional multiplexers, we have reduced the critical path delay by more than 50%. Furthermore, the **riBM** architecture consists of two systolic arrays and is thus very regular.

3) The RiBM Architecture: We now show that it is possible to eliminate the **ELU** block entirely, and to implement the **BM** algorithm in an enhanced **rDC** block in which the array of $2t$ **PE1** processors has been lengthened into an array of $3t+1$ **PE1** processors as shown in Fig. 6. In this completely systolic architecture, a *single* array computes *both* $\Lambda(z)$ and $\Omega^{(h)}(z)$. Since the $t+1$ **PE0** processors eliminated from the **ELU** block re-appear as the $t+1$ additional **PE1** processors, the **RiBM** architecture has the same hardware complexity and critical path delay as the **riBM** architecture. However, its extremely regular structure is esthetically pleasing, and also offers some advantage in VLSI circuit layouts.

An array of **PE0** processors in the **riBM** architecture (see Fig. 5) carries out the same *polynomial* computation as an array of **PE1** processors in the **RiBM** architecture (see Fig. 6), but in the latter array, the polynomial coefficients shift left with each clock pulse. Thus, in the **RiBM** architecture, suppose that the initial loading of $\mathbf{PE1}_0, \mathbf{PE1}_1, \ldots, \mathbf{PE1}_{2t-1}$ is as in Fig. 4, while $\mathbf{PE1}_{2t}, \mathbf{PE1}_{2t+1}, \ldots, \mathbf{PE1}_{3t-1}$ are loaded with zeros, and the latches in $\mathbf{PE1}_{3t}$ are loaded with $1 \in \mathrm{GF}(2^m)$. Then, as the iterations proceed, the polynomials $\hat{\Delta}(r,z)$ and $\hat{\Theta}(r,z)$ are updated in the processors in the left-hand end of the array (effectively, $\Delta(r,z)$ and $\Theta(r,z)$ get updated and shifted leftwards). After $2t$ clock cycles, the coefficients of $\Omega^{(h)}(z)$ are in processors $\mathbf{PE1}_0$–$\mathbf{PE1}_{t-1}$. Next, note that $\mathbf{PE1}_{3t}$ contains $\Lambda(0,z)$ and $B(0,z)$, and as the iterations proceed, $\Lambda(r,z)$ and $B(r,z)$ shift leftwards through the processors in the right-hand end of the array, with $\lambda_i(r)$ and $b_i(r)$ being stored in processor $\mathbf{PE1}_{3t-r+i}$. After $2t$ clock cycles, processor $\mathbf{PE1}_{t+i}$ contains $\lambda_i(2t)$ and $b_i(2t)$ for $i = 0, 1, \ldots, t$. Thus, the same array is carrying out two separate computations. These computations do not interfere with one another. Polynomials $\Lambda(r,z)$ and $B(r,z)$ are stored in processors numbered $3t-r$ or higher. On the other hand, since $\deg \Delta(r,z) = \deg S(z) + \deg \Lambda(r,z)$, it follows that $\deg \hat{\Delta}(r,z) \le 2t-1-r+l(r)$ where $l(r) = (r-k(r))/2$ is known to be an upper bound on $\deg \Lambda(r,z)$. It is known [3] that $l(r)$ is a nondecreasing function of r and that it has maximum value $l(2t) = e$ if $e \le t$ errors have occurred. Hence, $2t-1-r+l(r) < 3t-r$ for all r, $0 \le r \le 2t$, and thus, as

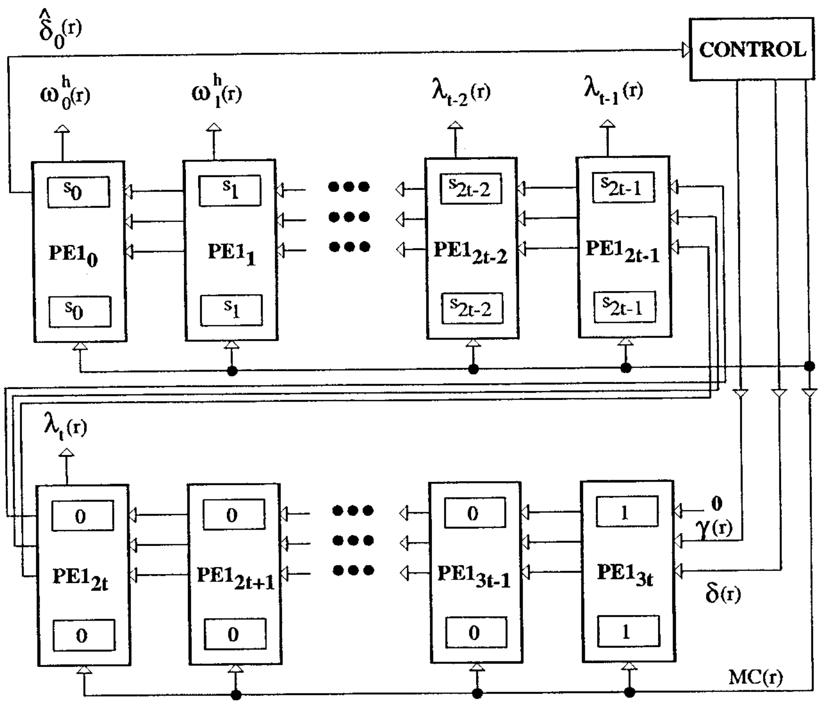


Fig. 6. The homogenous systolic **RiBM** architecture.

$\Lambda(r, z)$ and $B(r, z)$ shift leftwards, they do not over-write the coefficients of $\hat{\Delta}(r, z)$ and $\hat{\Theta}(r, z)$.

We denote the contents of the array in the **RiBM** architecture as polynomials $\tilde{\Delta}(r, z)$ and $\tilde{\Theta}(r, z)$ with initial values $\tilde{\Delta}(0, z) = \tilde{\Theta}(0, z) = S(z) + z^{3t}$. Then, the **RiBM** architecture implements the following pseudocode. Note that $\tilde{\delta}_{3t+1}(r) = 0$ for all values of r, and this quantity does not need to be stored or updated.

```
The RiBM Algorithm
 Initialization:
 $\tilde{\delta}_{3t}(0) = 1$;  $\tilde{\delta}_i(0) = 0$ for $i = 2t, 2t+1, \ldots, 3t-1$.  $k(0) = 0$.  $\gamma(0) = 1$.
 Input: $s_i$, $i = 0, 1, \ldots, 2t-1$.
   $\tilde{\delta}_i(0) = \tilde{\theta}_i(0) = s_i$,  $(i = 0, \ldots, 2t-1)$
   for $r = 0$ step 1 until $2t-1$ do
     begin
       Step RiBM.1 $\tilde{\delta}_i(r+1) = \gamma(r) \cdot \tilde{\delta}_{i+1}(r) - \tilde{\delta}_0(r) \cdot \tilde{\theta}_i(r)$,  $(i = 0, \ldots, 3t)$
       Step RiBM.2 if $\tilde{\delta}_0(r) \neq 0$ and $k(r) \geq 0$
                   then
                     begin
                       $\tilde{\theta}_i(r+1) = \tilde{\delta}_{i+1}(r)$,  $(i = 0, 1, \ldots, 3t)$
                       $\gamma(r+1) = \tilde{\delta}_0(r)$
                       $k(r+1) = -k(r) - 1$
                     end
                   else
                     begin
                       $\tilde{\theta}_i(r+1) = \tilde{\theta}_i(r)$, $(i = 0, 1, \ldots, 3t)$;
                       $\gamma(r+1) = \gamma(r)$
                       $k(r+1) = k(r) + 1$
                     end
     end
 Output: $\lambda_i(2t) = \tilde{\delta}_{t+i}(2t)$,  $(i = 0, 1, \ldots, t)$;  $\omega_i^{(h)}(2t) = \tilde{\delta}_i(2t)$,  $(i = 0, 1, \ldots, t-1)$.
```

4) Comparison of Architectures: Table I summarizes the complexity of the various architectures described so far. It can be seen that, in comparison to the conventional **iBM** architecture (Berlekamp's version), the proposed **riBM** and **RiBM** systolic architectures require $t - 1$ more multipliers and t more multiplexers. All three architectures require the same numbers of latches and adders and all three architectures require $2t$ cycles to solve the key equation for a t-error-correcting code. The **riBM** and **RiBM** architectures require considerably more gates than the conventional **iBM** architecture (Blahut's version), but also require only $2t$ clock cycles as compared to the $3t$ clock cycles required by the latter. Furthermore, since the critical path delay in the **riBM** and **RiBM** architectures is less than half the critical path delay in either of the **iBM** architectures, we conclude that the new architectures significantly reduce the total time required to solve the key equation (and thus achieve higher throughput) with only a modest increase in gate count. More important, the regularity and scalability of the **riBM** and **RiBM** architectures creates the potential for automatically generating regular layouts (via a core generator) with predictable delays for various values of t and m.

Comparison of the **riBM** and **RiBM** architectures with **eE** architectures is complicated by the fact that most recent implementations use *folded* architectures in which each processor element in the systolic array has only a few arithmetic units, and these units carry out all the needed computations via time-division multiplexing. For example, the hypersystolic **eE** architecture in [2] has $2t + 1$ processor elements each containing only one multiplier and adder. Since each iteration of the Euclidean algorithm requires four multiplications, the processors

TABLE I
COMPARISON OF HARDWARE COMPLEXITY AND PATH DELAYS

Architecture	Adders	Multipliers	Latches	Muxes	Clock cycles	Critical path delay
iBM (Blahut)	$2t+1$	$3t+3$	$4t+2$	$t+1$	$3t$	$> 2 \cdot (T_{\text{mult}} + T_{\text{add}})$
iBM (Berlekamp)	$3t+1$	$5t+3$	$6t+2$	$2t+1$	$2t$	$> 2 \cdot (T_{\text{mult}} + T_{\text{add}})$
riBM	$3t+1$	$6t+2$	$6t+2$	$3t+1$	$2t$	$T_{\text{mult}} + T_{\text{add}}$
RiBM	$3t+1$	$6t+2$	$6t+2$	$3t+1$	$2t$	$T_{\text{mult}} + T_{\text{add}}$
Euclidean [14]	$4t+2$	$8t+8$	$4t+4$	$8t+8$	$2t$	$T_{\text{mult}} + T_{\text{add}} + T_{\text{mux}}$
Euclidean [2](folded)	$2t+1$	$2t+1$	$10t+5$	$14t+7$	$12t$	$T_{\text{mult}} + T_{\text{add}} + T_{\text{mux}}$

of [2] need several multiplexers to route the various operands to the arithmetic units, and additional latches to store one addend until the other addend has been computed by the multiplier, etc. As a result, the architecture described in [2] requires not only many more latches and multiplexers, but also *many* more clock cycles than the **riBM** and **RiBM** architectures. Furthermore, the critical path delay is slightly larger because of the multiplexers in the various paths. On the other hand, finite-field multipliers themselves consist of large numbers of gates (possibly as many as $2m^2$, but fewer if logic minimization techniques are used) and, thus, a complete comparison of gate counts for the two architectures requires very specific details about the multipliers. Nonetheless, a rough comparison is that the **riBM** and **RiBM** architectures require three times as many gates as the hypersystolic **eE** architecture, but solve the key equation in one-sixth the time.

It is, of course, possible to implement the **eE** algorithm with more complex processor elements, as described by Shao et al. [14]. Here, the four multiplications in each processor are computed using four separate multipliers. The architecture described in [14] uses only $2t+1$ processors as compared to the $3t+1$ **PE0** or **PE1** processors needed in the **riBM** and **RiBM** architectures, but each processor in [14] has 4 multipliers, four multiplexers, and two adders. As a result, the **riBM** and **RiBM** architectures compare very favorably to the **eE** architecture of [14]—the new architectures achieve the same (actually slightly higher) throughput with much smaller complexity.

One final point to be made with respect to the comparison between the **riBM** and **RiBM** architectures and the **eE** architectures is that the controllers for the systolic arrays in the former are actually much simpler. In the **eE** architecture of [14], each processor also has a "control section" that uses an arithmetic adder, comparator, and two multiplexers. $2\lceil \log_2 t \rceil$ bits of arithmetic data are passed from processor to processor in the array, and these are used to generate multiplexer control signals in each processor. Similarly, the **eE** architecture of [2] has a separate control circuit for each processor. The delays in these control circuits are not accounted for in the critical path delays for the **eE** architectures that we have listed in Table I. In contrast, *all* the multiplexers in the **riBM** and **RiBM** architectures receive the same signal and the computations in these architectures is purely systolic in the sense that all processors carry out exactly the same computation in each cycle, with all the multiplexers set the same way in all the processors—there are *no* cell-specific control signals.

5) Preliminary Layout Results: Preliminary layout results from a core generator are shown in Fig. 7 for the **KES** block for a four-error-correcting Reed–Solomon code over $\mathrm{GF}(2^8)$. The processing element **PE1** is shown in Fig. 7(a) where the upper eight latches store the element $\tilde{\delta}_r$ while the lower eight latches store the element $\tilde{\theta}_r$. A complete **RiBM** architecture is shown in Fig. 7(b) where the 13 **PE1** processing elements are arrayed diagonally and the error locator and error evaluator polynomials output latches can be seen to be arrayed vertically. The critical path delay of the **RiBM** architecture as reported by the synthesis tool in SYNOPSYS was 2.13 ns in TSMC's 0.25 μm 3.3 V CMOS technology.

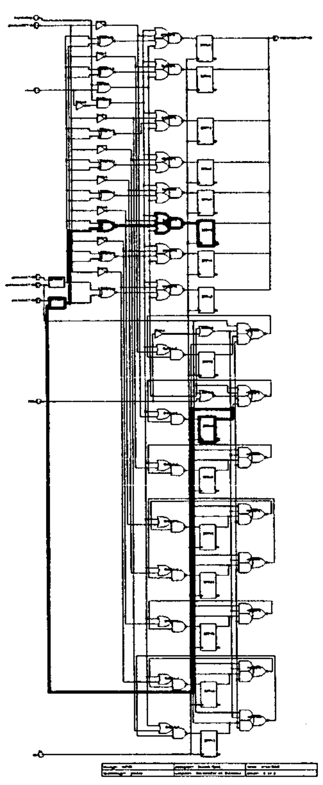
In the next section, we develop a pipelined architecture that further reduces the critical path delay by as much as an order of magnitude by using a *block interleaved* code.

V. PIPELINED REED–SOLOMON DECODERS

The iterations in the original **BM** algorithm were pipelined using the *look-ahead* transformation [12] by Liu *et al.* [9], and the same method can be applied to the **riBM** and **RiBM** algorithms. However, such pipelining requires complex overhead and control hardware. On the other hand, *pipeline interleaving* (also described in [12]) of a decoder for a *block-interleaved* Reed–Solomon code is a simple and efficient technique that can reduce the critical path delay in the decoder by an order of magnitude. We describe our results for only the **RiBM** architecture of Section IV, but the same techniques can also be applied to the **riBM** architecture as well as to the decoder architectures described in Section III.

A. Block-Interleaved Reed–Solomon Codes

1) Block Interleaving: Error-correcting codes for use on channels in which errors occur in bursts are often interleaved so that symbols from the same codeword are not transmitted consecutively. A burst of errors thus causes single errors in multiple codewords rather than multiple errors in a single codeword. The latter occurrence is undesirable since it can easily overwhelm the error-correcting capabilities of the code and cause a decoder failure or decoder error. Two types of interleavers, block interleavers and convolutional interleavers,


(a)

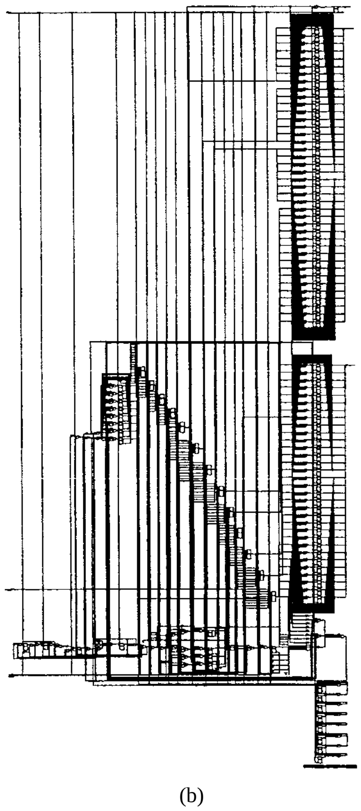

(b)

Fig. 7. The **RiBM** architecture synthesized in a 3.3 V, 0.25 μm CMOS technology. (a) The **PE1** processing element. (b) The **RiBM** architecture.

are commonly used (see, e.g., [16], [18]). We restrict our attention to block-interleaved codes.

Block-interleaving an (n,k) code to *depth* M results in an (nM,kM) interleaved code whose codewords $(c_{nM-1}, c_{nM-2}, \ldots, c_1, c_0)$ have the property that $(c_{(n-1)M+i}, c_{(n-2)M+i}, \ldots, c_{M+i}, c_i)$, $0 \le i \le M-1$, is a codeword in the (n,k) code. Equivalently, a codeword of the (nM,kM) code is a *multichannel data stream* in which each of the M channels carries a codeword of the (n,k) code.

2) Interleaving via Memory Arrays: The usual description (see, e.g., [16], [18]) of an encoder for the block-interleaved (nM,kM) code involves partitioning kM data symbols $(d_{kM-1}, d_{kM-2}, \ldots, d_1, d_0)$ into M blocks of k consecutive symbols, and encoding each block into a codeword of the (n,k) code. Next, these M codewords are stored row wise into an $M \times n$ memory array. The memory is then read out column wise to form the block-interleaved codeword. Notice that the block-interleaved codeword is systematic in the sense that the parity-check symbols follow the data symbols, but the Reed–Solomon encoding process described in Section II-A results in a block-interleaved codeword

$$\begin{aligned}&(c_{nM-1}, c_{nM-2}, \ldots, c_{(n-1)M}, c_{(n-1)M-1}, \ldots)\\ &\quad = (d_{kM-1}, d_{k(M-1)-1}, \ldots, d_{k-1}, d_{kM-2}, \ldots)\end{aligned}$$

in which the data symbols are *not* transmitted over the channel in the order in which they entered the encoder.[4] At the receiver, the interleaving process is reversed by storing the nM received symbols column-wise into an $M \times n$ memory array. The memory is then read out row wise to form M received words of length n that can be decoded by a decoder for the (n,k) code. The information symbols appear in the correct order in the deinterleaved stream and the decoder output is passed on to the destination.

3) Embedded Interleavers: An alternative form of block interleaving embeds the interleaver into the encoder, thereby transforming it into an encoder for the (nM,kM) code. For interleaved Reed-Solomon codes, the mathematical description of the encoding process is that the generator polynomial of the interleaved code is $G(z^M)$, where $G(z)$ denotes the generator polynomial of the (n,k) code as defined in (1), and the codeword is formed as described in Section II-A, i.e., with $D(z)$ now denoting the data polynomial $d_{kM-1}z^{kM-1} + d_{kM-2}z^{kM-2} + \cdots + d_1 z + d_0$ of degree $kM-1$, the polynomial $z^{(n-k)M}D(z)$ is divided by $G(z^M)$ to obtain the remainder $P(z)$ of degree $(n-k)M-1$. The transmitted codeword is $z^{(n-k)M}D(z) - P(z)$. In essence, the data stream $(d_{kM-1}, d_{kM-2}, \ldots, d_1, d_0)$ is treated as if it were a *multichannel data stream* and the stream in each channel is encoded with the (n,k) code. The output of the encoder is a codeword in the block-interleaved Reed–Solomon code (no separate interleaver is needed) and it has the property that the data symbols are transmitted over the channel in the order in which they entered the encoder.

The astute reader will have observed already that the encoder for the (nM,kM) code is just a *delay-scaled encoder* for the (n,k) code. The delay-scaling transformation of an architecture replaces every delay (latch) in the architecture with M delays, and re-times the architecture to account for the additional delays. The encoder treats its input as a multichannel data stream and produces a multichannel output data stream, that

[4]In fact, the data symbol ordering is that which is produced by interleaving the data stream in blocks of k symbols to depth M.

is, a block-interleaved Reed–Solomon codeword. Note also that while the interleaver array has been eliminated, the delay-scaled encoder uses M times as much memory as the conventional encoder.

Block-interleaved Reed–Solomon codewords produced by delay-scaled encoders contain the data symbols in the correct order. Thus, a *delay-scaled decoder* can be used to decode the received word of nM symbols, and the output of the decoder also will have the data symbols in the correct order. Note that a separate deinterleaver array is not needed at the receiver. However, the delay-scaled decoder uses M times as much memory as the conventional decoder. For example, delay-scaling the **PE1** processors in the **RiBM** architecture of Fig. 6 results in the delay-scaled processor **DPE1** shown in Fig. 8. Note that for $0 \leq i \leq 2t-1$, the top and bottom sets of M latches in $\mathbf{DPE1}_i$ are initialized with the syndrome set $\mathbf{S}_i^{0,M-1} = [s_{i,0}, s_{i,1}, \ldots, s_{i,M-1}]$, where $s_{i,j}$ is the ith syndrome of the jth codeword. For $2t \leq i \leq 3t-1$, the latches in $\mathbf{DPE1}_i$ are initialized to zero while the latches in $\mathbf{DPE1}_{3t}$ are initialized to $1 \in \mathrm{GF}(2^m)$. After $2tM$ clock cycles, processors $\mathbf{DPE1}_0$–$\mathbf{DPE1}_{t-1}$ contain the interleaved error-evaluator polynomials while processors $\mathbf{DPE1}_t$–$\mathbf{DPE1}_{2t}$ contain the interleaved error-locator polynomials.

We remark that delay-scaled decoders can also be used to decode block-interleaved Reed–Solomon codewords produced by memory array interleavers. However, the data symbols at the output of the decoder will still be interleaved and an $M \times k$ memory array is needed for deinterleaving the data symbols into their correct order. This array is smaller than the $M \times n$ array needed to deinterleave the M codewords prior to decoding with a conventional decoder, but the conventional decoder also uses less memory than the delay-scaled decoder.

Delay-scaling the encoder and decoder eliminates separate interleavers and deinterleaver and is thus a natural choice for generating and decoding block-interleaved Reed–Solomon codewords. However, a delay-scaled decoder has the same critical path delay as the original decoder, and hence cannot achieve higher throughput than the original decoder. On the other hand, the extra delays can be used to *pipeline* the computations in the critical path, and this leads to significant increases in the achievable throughput. We discuss this concept next.

B. Pipelined Delay-Scaled Decoders

The critical path delay in the **RiBM** architecture is mostly due to the finite-field multipliers in the processors. For the delay-scaled processors **DPE1** shown in Fig. 8, these multipliers can be pipelined and the critical path delay reduced significantly. We assume that $M \geq m$ and describe a pipelined finite-field multiplier with m stages.

1) A Pipelined Multiplier Architecture: While pipelining a multiplier, especially if it is a feedforward structure, is trivial, it is not so in this case. This is because for RS decoders the pipelining should be done in such a manner that the initial conditions in the pipelining latches are consistent with the syndrome values generated by the **SC** block. The design of finite-field multipliers depends on the choice of basis for the representation. Here, we consider only the standard polynomial basis in which

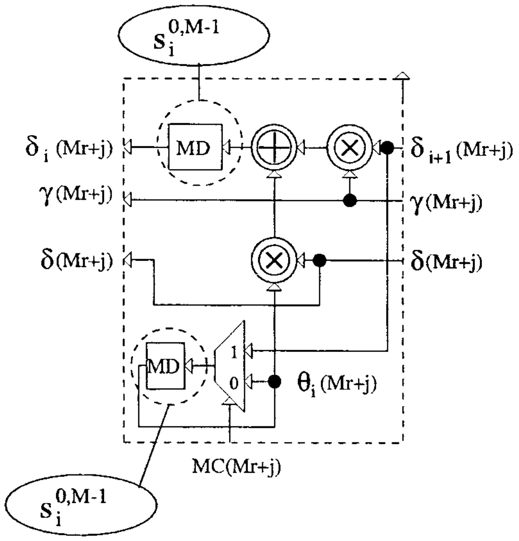


Fig. 8. Delay-scaled **DPE1** processor. Initial conditions in the latches are indicated in ovals. The delay-scaled **RiBM** architecture is obtained by replacing the **PE1** processors in Fig. 6 with **DPE1** processor and delay-scaling the control unit as well.

the m-bit byte $(x_{m-1}, x_{m-2}, \ldots, x_1, x_0)$ represents the Galois field element $X = x_{m-1}\alpha^{m-1} + x_{m-2}\alpha^{m-2} + \cdots + x_1\alpha + x_0$.

The pipelined multiplier architecture is based on writing the product of two $\mathrm{GF}(2^m)$ elements X and Y as

$$\begin{aligned} XY &= X(y_0 + y_1\alpha + y_2\alpha^2 + \cdots y_{m-1}\alpha^{m-1}) \\ &= Xy_0 + (X\alpha)y_1 + ((X\alpha)\alpha)y_2 + \cdots \\ &\quad + (\cdots((X\alpha)\alpha)\cdots\alpha)y_{m-1}. \end{aligned}$$

Let pp_i denote the sum of the first i terms in the sum above. The multiplier processing element $\mathbf{MPE}_i$ shown in Fig. 9(a) computes pp_{i+1} by adding either $X\alpha^i$ (if $y_i = 1$) or 0 (if $y_i = 0$) to pp_i. Simultaneously, $\mathbf{MPE}_i$ multiplies $X\alpha^i$ by α. Since α is a constant, this multiplication requires only XOR gates, and can be computed with a delay of only $T_{\mathrm{xor}} = T_{\mathrm{add}}$. On the other hand, the delay in computing pp_{i+1} is $T_{pp} = T_{\mathrm{mux}} + T_{\mathrm{add}}$. Thus, the critical path delay is an order of magnitude smaller than $T_{\mathrm{riBM}} = T_{\mathrm{mult}} + T_{\mathrm{add}}$, and tremendous speed gains can be achieved if the pipelined multiplier architecture is used in decoding a block-interleaved Reed–Solomon code. Practical considerations such as the delays due to pipelining latches, clock skew and jitter will prevent the fullest realization of the speed gains due to pipelining. Nevertheless, the pipelined multiplier structure in combination with the systolic architecture will provide significant gains over existing approaches.

The pipelined multiplier thus consists of m **MPE** processors connected as shown in Fig. 9(b) with inputs $pp_0 = 0, X$ and the y_i's. The initial conditions of the latches at the y input are zero, and therefore the initial conditions of the lower latches in the **MPE**s do not affect the circuit operation. The product XY appears in the upper latch of $\mathbf{MPE}_{m-1}$ after m clock cycles and each succeeding clock cycle thereafter computes a new product. Notice also that during the first m clock cycles, the initial contents of the upper latches of the **MPE**s appear in succession at the output of $\mathbf{MPE}_{m-1}$. This property is crucial to the proper operation of our proposed pipelined decoder.

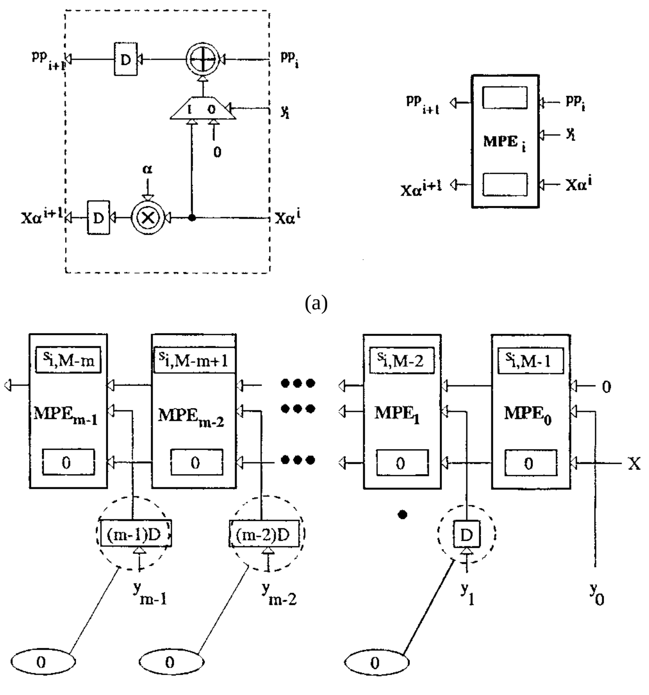


Fig. 9. The pipelined multiplier block diagram. (a) The multiplier processing element (**MPE**). (b) The multiplier architecture. Initial conditions of the latches at the y input are indicated in ovals.

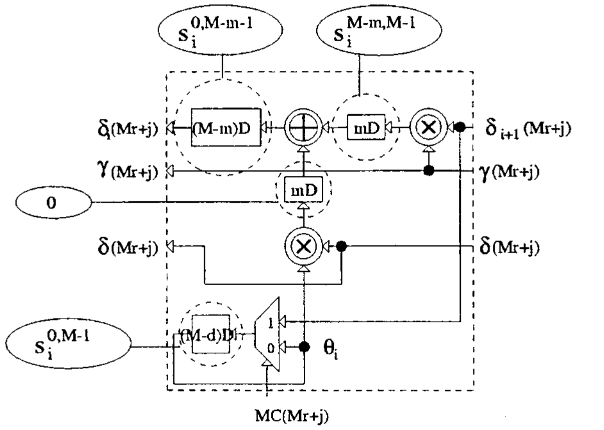


Fig. 10. Pipelined **PPE1** processor. Initial conditions in the latches are indicated in ovals. The pipelined **RiBM** architecture is obtained by replacing the **PE1** processors in Fig. 6 with **PPE1** processor and employing the pipelined delay-scaled controller.

2) The Pipelined Control Unit: If the pipelined multiplier architecture described above (and shown in Fig. 9) is used in the **DPE1** processors of Fig. 8, the critical path delay of **DPE1** is reduced from $T_{\text{mult}} + T_{\text{add}}$ to just $T_{\text{mux}} + T_{\text{add}}$. Thus, the control unit delay in computing $\text{MC}(r)$, which is inconsequential in the **RiBM** architecture (as well as in the **iBM** and **riBM** architectures, and the delay-scaled versions of all these), determines the largest delay in a pipelined **RiBM** architecture.

Fortunately, the computation of $\text{MC}(r)$ can also be pipelined in (say) $d = \lceil \log_2 m \rceil + 1$ stages. This can be done by noting that d delays from $\textbf{DPE1}_0$ in the M-delay scaled **RiBM** architecture (see Fig. 8) can be retimed to the outputs of the control unit and then subsequently employed to pipeline it. Note, however, that the d latches in $\textbf{DPE1}_0$ that are being retimed are initialized to $S_i^{0,d-1}$ at the begininng of every decoding cycle. Hence, the retimed latches in the control unit will need to be initialized to values that are a function of syndromes $S_i^{0,d-1}$. This is not a problem because these syndromes will be produced by the **SC** block in the beginning of each decoding cycle.

3) Pipelined Processors: If pipelined multiplier units as described above are used in a delay-scaled **DPE1** processor, and the control unit is pipelined as described above, then we get the pipelined $\textbf{PPE1}_i$ processor shown in Fig. 10 (and the pipelined **RiBM** (**pRiBM**) architecture also described in Fig. 10). The initial values stored in the latches are the same as were described earlier for the **DPE1** processors. Note that some of the latches that store the coefficients of $\tilde{\Delta}(r, z)$ are part of the latches in the pipelined multiplier. However, the initial values in the latches in the lower multiplier in Fig. 10 are zero. Thus, during the first m clock cycles $(s_{i,M-m}, s_{M-m+1}, \ldots, s_{i,M-1})$ flow through into the leftmost latches without any change.

From the above description, it should be obvious that the **pRiBM** architecture based on the **PPE1** processor of Fig. 10 has a critical path delay of

$$T_{\text{pRiBM}} = T_{\text{add}} + T_{\text{mux}} \ll T_{\text{RiBM}} = T_{\text{add}} + T_{\text{mult}}. \qquad (13)$$

Thus, the **pRiBM** architecture can be clocked at speeds that can be as much as an order of magnitude higher than those achievable with the unpipelined architectures presented in Sections III and IV.

C. Decoders for Noninterleaved Codes

The **pRiBM** architecture can decode a block-interleaved code at significantly faster rates than the **RiBM** architecture can decode a noninterleaved code. In fact, the difference is large enough that a designer who is asked to devise a decoder for noninterleaved codes should give serious thought to the following design strategy.

- Read in M successive received words into an block-interleaver memory array.
- Read out a block-interleaved received word into a decoder with the **pRiBM** architecture.
- Decode the block-interleaved word and read out the the data symbols into a block-deinterleaver memory array.
- Read out the deinterleaved data symbols from the deinterleaver array.

Obviously, similar decoder design strategies can be used in other situations as well. For example, to decode a convolutionally interleaved code, one can first deinterleave the received words, and then re-interleave them into block-interleaved format for decoding. Similarly, if a block-interleaved code has *very large* interleaving depth M, the **pRiBM** architecture may be too large to implement on a single chip. In such a case, one can deinterleave first and then reinterleave to a suitable depth. In fact, the "deinterleave and reinterleave" strategy can be used to construct a universal decoder around a single decoder chip with fixed interleaving depth.

VI. Concluding Remarks

We have shown that the application of algorithmic transformations to the Berlekamp–Massey algorithm result in the **riBM** and **RiBM** architectures whose critical path delay is less than half that of conventional architectures such as the **iBM** architecture. The **riBM** and **RiBM** architectures use systolic arrays of identical processor elements. For block-interleaved codes, the deinterleaver can be embedded in the decoder architecture via delay scaling. Furthermore, pipelining the multiplications in the delay-scaled architecture result in an order of magnitude reduction in the critical path delay. In fact, the high speeds at which the **pRiBM** architecture can operate makes it feasible to use it to decode *noninterleaved* codes by the simple stratagem of internally interleaving the received words, decoding the resulting interleaved word using the **pRiBM** architecture, and then de-interleaving the output.

Future work is being directed toward integrated circuit implementations of the proposed architectures and their incorporation into broadband communications systems such as those for very high-speed digital subscriber loops and wireless systems.

Acknowledgment

The authors would like to thank the reviewers for their constructive criticisms which has resulted in significant improvements in the manuscript.

References

[1] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968. (revised ed.—Laguna Hills, CA: Aegean Park, 1984).
[2] E. R. Berlekamp, G. Seroussi, and P. Tong, *Reed–Solomon Codes and Their Applications*, S. B. Wicker and V. K. Bhargava, Eds. Piscataway, NJ: IEEE Press, 1994. A hypersystolic Reed–Solomon decoder.
[3] R. E. Blahut, *Theory and Practice of Error-Control Codes*. Reading, MA: Addison-Wesley, 1983.
[4] H. O. Burton, "Inversionless decoding of binary BCH codes," *IEEE Trans. Inform. Theory*, vol. IT-17, pp. 464–466, Sept. 1971.
[5] H.-C. Chang and C. Shung, "A Reed-Solomon product code (RS-PC) decoder for DVD applications," in *Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1998, pp. 390–391.
[6] H.-C. Chang and C. B. Shung, "New serial architectures for the Berlekamp–Massey algorithm," *IEEE Trans. Commun.*, vol. 47, pp. 481–483, Apr. 1999.
[7] M. A. Hasan, V. K. Bhargava, and T. Le-Ngoc, *Reed–Solomon Codes and Their Applications*, S. B. Wicker and V. K. Bhargava, Eds. Piscataway, NJ: IEEE Press, 1994. Algorithms and architectures for a VLSI Reed–Solomon codec.
[8] S. Kwon and H. Shin, "An area-efficient VLSI architecture of a Reed–Solomon decoder/encoder for digital VCRs," *IEEE Trans. Consumer Electron.*, pp. 1019–1027, Nov. 1997.
[9] K. J. R. 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.
[10] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122–127, Mar. 1969.
[11] J. Nelson, A. Rahman, and E. McQuade, "Systolic architectures for decoding Reed-Solomon codes," in *Proc. Int. Conf. Application Specific Array Processors*, Princeton, NJ, Sept. 1990, pp. 67–77.
[12] K. K. Parhi and D. G. Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters—Parts I and II," *IEEE Trans. Acoust. Speech Signal Processing*, vol. 37, pp. 1099–1134, July 1989.
[13] I. S. Reed, M. T. Shih, and T. K. Truong, "VLSI design of inverse-free Berlekamp–Massey algorithm," *Proc. Inst. Elect. Eng.*, pt. E, vol. 138, pp. 295–298, Sept. 1991.
[14] 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. 393–403, May 1985.
[15] P. Tong, "A 40 MHz encoder-decoder chip generated by a Reed-Solomon code compiler," *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 13.5.1–13.5.4, May 1990.
[16] R. B. Wells, *Applied Coding and Information Theory for Engineers*. Upper Saddle River, NJ: Prentice-Hall, 1999.
[17] S. R. Whitaker, J. A. Canaris, and K. B. Cameron, "Reed–Solomon VLSI codec for advanced television," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 1, pp. 230–236, June 1991.
[18] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
[19] W. Wilhelm, "A new scalable VLSI architecture for Reed–Solomon decoders," *IEEE J. Solid-State Circuits*, vol. 34, pp. 388–396, Mar. 1999.

Dilip V. Sarwate (S'68–M'73–SM'78–F'90) received the Bachelor of Science degree in physics and mathematics from the University of Jabalpur, Jabalpur, India, in 1965, the Bachelor of Engineering degree in electrical communication engineering from the Indian Institute of Science, Bangalore, India, in 1968, and the Ph.D. degree in electrical engineering from Princeton University, Princeton, NJ, in 1973.

Since January 1973, he has been with the University of Illinois at Urbana-Champaign, where he is currently a Professor of Electrical and Computer Engineering and a Research Professor in the Coordinated Science Laboratory. His research interests are in the general areas of communication systems and information theory, with emphasis on multiuser communications, error-control coding, and signal design.

Dr. Sarwate has served as the Treasurer of the IEEE Information Theory Group, as an Associate Editor for Coding Theory of the IEEE Transactions on Information Theory, and as a member of the editorial board of IEEE Proceedings. He was a Co-Chairman of the 18th, 19th, 31st, and 32nd *Annual Allerton Conferences on Communication, Control, and Computing* held in 1980, 1981, 1993, and 1994, respectively. In 1985, he served as a Co-Chairman of the Army Research Office Workshop on Research Trends in Spread Spectrum Systems. He has also been a member of the program committees for the IEEE Symposia on Spread Spectrum Techniques and Their Applications (ISSSTA) and the 1998 and 2001 Conferences on Sequences and Their Applications (SETA), as well as of several advisory committees for international conferences.

Naresh R. Shanbhag (S'87–M'93) received the B.Tech. degree from the Indian Institute of Technology, New Delhi, India, in 1988, the M.S. degree from the Wright State University, Dayton, OH, in 1990, and the Ph.D. degree from the University of Minnesota, Minneapolis, in 1993, all in electrical engineering.

From July 1993 to August 1995, he worked at AT&T Bell Laboratories, Murray Hill, NJ, where he was responsible for the development of VLSI algorithms, architectures, and implementation of broad-band data communications transceivers. In particular, he was the lead chip architect for AT&T's 51.84 Mb/s transceiver chips over twisted-pair wiring for asynchronous transfer mode (ATM)-LAN and broad-band access chip sets. Since August 1995, he has been with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, where he is presently an Associate Professor and the Director of the Illinois Center for Integrated Microsystems. At the University of Illinois, he founded the VLSI Information Processing Systems (ViPS) Group (http://www.icims.csl.uiuc.edu/~shanbhag/vips), whose charter is to explore issues related to low-power, high-performance, and reliable integrated circuit implementations of broad-band communications and digital signal processing systems spanning the algorithmic, architectural, and circuit domains. He has published more than 90 journal articles/book chapters/conference publications in this area and holds three U.S. patents. He is also a coauthor of the research monograph *Pipelined Adaptive Digital Filters* (Norwell, MA: Kluwer, 1994).

Dr. Shanbhag received the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS Best Paper Award, the 1999 IEEE Leon K. Kirchmayer Best Paper Award, the 1999 Xerox Faculty Award, the National Science Foundation CAREER Award in 1996, and the 1994 Darlington Best Paper Award from the IEEE Circuits and Systems Society. Since July 1997, he has been a Distinguished Lecturer for the IEEE Circuits and Systems Society. From 1997 to 1999, he served as an Associate Editor for the IEEE TRANSACTION ON CIRCUITS AND SYSTEMS: PART II. He has also served on the technical program committee of various international conferences.