Systolic VLSI Arrays for Polynomial GCD Computation

RICHARD P. BRENT, SENIOR MEMBER, IEEE, AND H.T. KUNG

***Abstract*— The problem of finding a *greatest common divisor* (GCD) of any two nonzero polynomials is fundamental to algebraic and symbolic computations, as well as to the decoder implementation for a variety of error-correcting codes. This paper describes new systolic arrays that can lead to efficient VLSI solutions to both the GCD problem and the extended GCD problem.**

***Index Terms*— Algorithms, error-correcting codes, greatest common divisor, special-purpose hardware, systolic arrays, VLSI.**

I. Introduction

The polynomial GCD problem is to compute a *greatest common divisor* of any two nonzero polynomials. The problem is fundamental to algebraic and symbolic computations (see, e.g., [3], [7]), and to the decoder implementation for a variety of error-correcting codes (see, e.g., [12], [13]). Many sequential algorithms for solving the GCD problems are known in the literature. In fact, the Euclidean algorithm and its variants for solving the problem are among the most well-known and well-studied computer algorithms (see [1], [7]). However, for direct VLSI implementation, these previously known algorithms all seem to be too irregular and/or too complex to be useful. For example, the Euclidean algorithm involves a sequence of complicated polynomial divisions on polynomials whose size can only be determined during the computation. This paper describes some simple and regular systolic structures that can lead to efficient VLSI solutions to the GCD problem and some of its variants. For instance, we describe a systolic array of $m + n + 1$ cells that can find a GCD of any two polynomials of total degree no more than $m + n$. Fig. 1 illustrates that the systolic array inputs the coefficients of the given polynomials $\sum_{i=0}^{n} a_i x^i$ and $\sum_{i=0}^{m} b_i x^i$ at the left-most cell, and outputs the coefficients of their GCD at the right-most cell. More precisely, if a unit of time is the cell cycle time (which is basically the time to perform a division, or both a multiplication and an addition), then at $2(m + n + 1)$ time units after the a_n and b_m enter the left-most cell, the coefficients of the GCD will start coming out from the right-most cell at the rate of one coefficient every time unit. During computation all cells work in parallel and communicate with their nearest neighbors only.

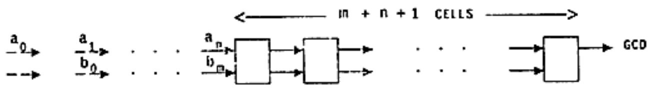


Fig. 1. Systolic GCD array.

Like many other systolic designs (see, e.g., [10], [11]), systolic GCD arrays in this paper are suitable for VLSI implementation and can achieve high throughputs with relatively low costs. These systolic GCD arrays were actually designed for the purpose of implementing the decoder for Reed–Solomon error-correcting codes (and BCH and Goppa codes, in general) with the CMU programmable systolic chip (PSC) that has been fabricated in nMOS and is functionally working [5], [6]. The most difficult step in the decoder implementation was to solve a version of the extended GCD problem; results of this paper helped solve this problem (see Section V).

It may not be easy to understand some of the more complicated systolic arrays of this paper. We shall start with the basic ideas and describe some simpler designs first. We hope informal arguments we give will convince the reader that our designs are correct. Formal correctness proofs for our designs would be very interesting, but they are beyond the scope of this paper. Nevertheless, every design mentioned in this paper has been coded, and thoroughly tested by simulation.

II. GCD-Preserving Transformations

All of the known algorithms for solving the GCD problem are based on the general technique of reducing the degrees of the two given polynomials by "GCD-preserving transformations." A *GCD-preserving transformation* transforms a pair of polynomials A and B into another pair $\overline{A}$ and $\overline{B}$, with the property that a GCD of A and B is also a GCD of $\overline{A}$ and B, and vice versa. When one of the two polynomials is reduced to the zero polynomial by a sequence of such transformations, the other polynomial will be a GCD of the original two polynomials.

Methods of this paper are also based on the general technique outlined above. We assume throughout that coefficients of the polynomials belong to a finite field F. This assumption is appropriate for applications to the decoder implementation for error-correcting codes; in the last section of the paper we point out that with straightforward modi-

Manuscript received August 4, 1982; revised December 12, 1983. This work was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659, in part by the National Science Foundation under Grant MCS 78-236-76, and in part by the Defense Advanced Research Projects Agency, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

R. P. Brent is with the Centre for Mathematical Analysis, The Australian National University, Canberra, Australia.
H. T. Kung is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

fications our designs will work over any unique factorization domain and thus require no divisions. In the following we define two GCD-preserving transformations R_A and R_B. Let

$$A = a_ix^i + \cdots + a_1x + a_0$$

and

$$B = b_jx^j + \cdots + b_1x + b_0$$

be the two polynomials[1] to be transformed, where $a_i \neq 0$ and $b_j \neq 0$. Depending on the value of $i - j$, one of the following two transformations is applied to A and B.

Transformation R_A (for the case when $i - j \geq 0$):

$$\begin{tikzcd} A \arrow[r] & \boxed{} \arrow[r] & \overline{A} = A - qx^dB \\ B \arrow[r] & \boxed{R_A} \arrow[r] & \overline{B} = B \end{tikzcd}$$

where $d = i - j$ and $q = a_i/b_j$.

Transformation R_B (for the case when $i - j < 0$):

$$\begin{tikzcd} A \arrow[r] & \boxed{} \arrow[r] & \overline{A} = A \\ B \arrow[r] & \boxed{R_B} \arrow[r] & \overline{B} = B - qx^dA \end{tikzcd}$$

where $d = j - i$ and $q = b_j/a_i$.

It is easy to check that both the transformations are GCD preserving. Furthermore, note that R_A reduces the degree of A by at least one, i.e.,

$$\deg \overline{A} \leq \deg A - 1\,,$$

and R_B reduces the degree of B by at least one, i.e.,

$$\deg \overline{B} \leq \deg B - 1\,.$$

For notational convenience, we assume that the degree of the zero polynomial is -1. Suppose that both A and B are polynomials of degree zero, i.e., they are nonzero constants in the underlying field F. Then by transformation R_A, A will be reduced to the zero polynomial, which has degree -1.

III. Transformation Sequence for the GCD Computation

Suppose that we want to compute a GCD of two given polynomials A_0 and B_0 of degrees n and m. We will apply a sequence of GCD-preserving transformations, each one being either R_A or R_B, to the two polynomials until one of them becomes the zero polynomial; at this point a GCD of A_0 and B_0 is the other (nonzero) polynomial. We call this sequence of transformations the *transformation sequence* for the GCD computation for A_0 and B_0, and denote it by $(T_1, T_2, \cdots, T_k)$ for some k, where T_i, $1 \leq i \leq k$, is either transformation R_A or R_B. Note that the transformation sequence is uniquely defined for given A_0 and B_0.

An instructive way to view the function of the transformation sequence is to imagine that polynomials A_0 and B_0 move through the "transformation stages," $T_1, T_2, \cdots, T_k$, in the left-to-right direction, and get transformed at each stage accordingly. When the polynomials move out from the last stage T_k, one of them will be the zero polynomial and the other the GCD of A_0 and B_0 that we want to compute. This view is illustrated in Fig. 2.

For each $i = 1, \cdots, k$, transformation T_i reduces the degree of one of its two input polynomials by some positive integer δ_i. We call δ_i the *reduction value* of T_i. Usually, δ_i is 1, but it could be greater than 1 at times. Since the total degree of A_0 and B_0 is $n + m$, the sum of reduction values over $i = 1, \cdots, k$ obeys the important relation $\sum_{i=1}^{k} \delta_i \leq n + m + 1$.

IV. Systolic GCD Array

In this section we describe a systolic array of $n + m + 1$ cells capable of computing a GCD of any two polynomials A_0 and B_0 of degrees no more than n and m, respectively.

Consider the transformation sequence $(T_1, T_2, \cdots, T_k)$ for the GCD computation for A_0 and B_0. We shall show that for each $i = 1, \cdots, k$, transformation T_i can be realized by a subarray of δ_i cells where δ_i is the reduction value of T_i. Since $\sum_{i=1}^{k} \delta_i \leq n + m + 1$, the systolic array with $n + m + 1$ cells can realize all the transformations, and therefore can compute a GCD of A_0 and B_0. This is illustrated in Fig. 3.

A. Basic Idea (for Realizing a Single Transformation Stage)

Let T be any transformation in the transformation sequence $(T_1, T_2, \cdots, T_k)$, and δ its reduction value. We illustrate how a subarray of δ cells can realize T, assuming that by some other methods (see, for example, Section IV-B) we know which one of R_A and R_B transformation T is.

Here we consider only the case when T is R_A; the case when T is R_B can be treated similarly. Without loss of generality, assume T is defined as follows:

$$\begin{tikzcd} A \arrow[r] & \boxed{} \arrow[r] & \overline{A} = A - qx^dB \\ B \arrow[r] & \boxed{R_A} \arrow[r] & \overline{B} = B \end{tikzcd}$$

where

$$A = a_ix^i + \cdots + a_1x + a_0, \qquad (a_i \neq 0)\,,$$
$$B = b_jx^j + \cdots + b_1x + b_0, \qquad (b_j \neq 0)\,,$$
$$q = a_i/b_j,$$

and

$$d = i - j \geq 0\,.$$

Note that either $\overline{A}$ is the zero polynomial, or

$$\overline{A} = \overline{a}_{i-\delta}x^{i-\delta} + \cdots + \overline{a}_1x + \overline{a}_0,$$

where $\overline{a}_{i-\delta} \neq 0$. The systolic subarray for realizing T, together with the operations performed by each of its cells at every cycle, is shown in Fig. 4.

Terms in A and B move through the subarray serially, high degree terms first. The nonzero leading terms of A and B are lined up so that they enter the left-most cell at the same cycle. Fig. 4 assumes that $i = j + 1$ to illustrate the point that for the case when $i > j$, for the input of B, as many as $i - j$ zeros

[1]Throughout the paper polynomials are represented by upper-case letters and their coefficients by corresponding lower-case letters.

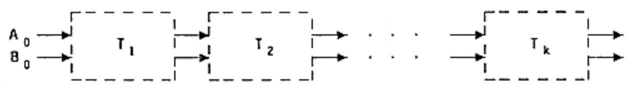


Fig. 2. Function of the transformation sequence.

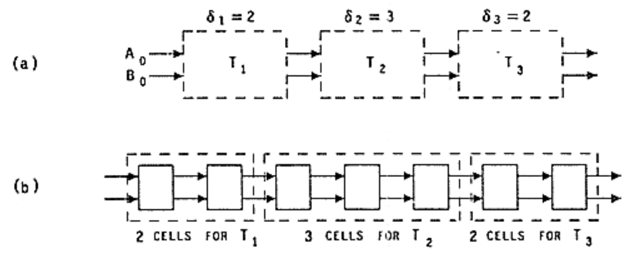
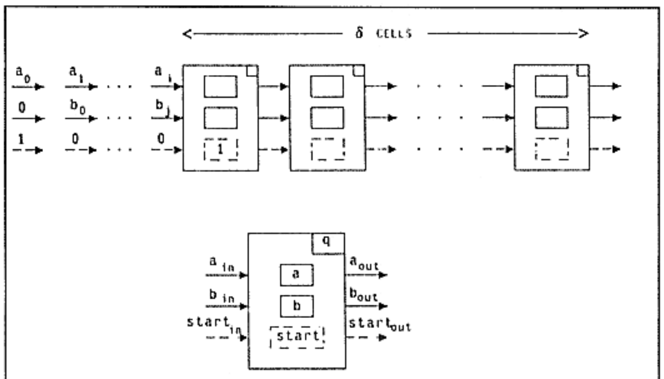


Fig. 3. (a) Transformation sequence, and (b) its realization by three concatenated systolic subarrays.



```
if start then
        begin
        q:= a_in/b_in;
        a_out := 0              {* pad in zeroes for vanishing terms in Ā *}
        end
else a_out := a_in - q*b_in;
b_out:= b; b := b_in;      {* it takes 2 cycles for each b to pass a cell *}
start_out := start; start := start_in
```

Fig. 4. Systolic subarray and its cell definition for realizing a transformation R_A.

should be added to the left of b_0. Besides the systolic data paths for a and b, there is another 1-bit wide *systolic control path*, denoted by **start**; a true value on this path signals to a cell the beginning of a new GCD computation in the following cycle. In Fig. 4 (and other figures in sequel) 1-bit wide systolic control paths and the associated latches in a cell are shown by dotted arrows and boxes.

It is easy to see that the left-most cell performs $q := a_i/b_j$ in the first cycle and computes terms of $\overline{A}$ in subsequent cycles. The q's computed by other cells, however, are always zeros since terms of $\overline{A}$ that have degrees higher than $i - \delta$ are zero terms. The only function of these cells is to shift the a's faster than the b's; notice that each b stays at each cell it passes for an extra cycle. Through these "shifting" cells the nonzero leading term $\overline{a}_{i-\delta}$ of $\overline{A}$ will depart from the right-most cell at the same cycle as b_j, the nonzero leading term of $\overline{B}$. Thus, $\overline{a}_{i-\delta}$ and b_j are ready to enter another subarray of cells to the right-hand side for realizing whatever transformation follows T. The important fact that outputs $\overline{A}$ and $\overline{B}$ are lined up is depicted in Fig. 5.

There is no need to keep track of the δ value in the systolic subarray. If $\overline{A}$ is nonzero, the realization of the trans-

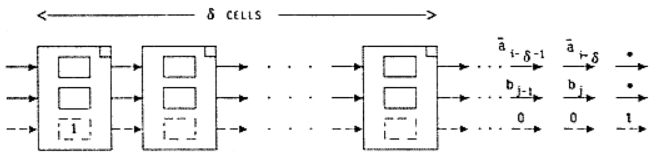


Fig. 5. Outputs of the systolic subarray of Fig. 4.

formation following T starts automatically at the first cell that sees a nonzero input, i.e., $\overline{a}_{i-\delta}$, appearing at its input line $\mathbf{a}_{\text{in}}$. If $\overline{A}$ is the zero polynomial, then T must be the last transformation T_k. In this case, the b's will continue being shifted to the right-hand side to be output from the right-most cell, and they will be the terms in the GCD that we wish to compute.

B. Design Using Difference of Degrees

We have seen that a systolic subarray with its cells defined by Fig. 4 can realize any transformation T, if it is known which one of the transformations R_A and R_B transformation T is. Let

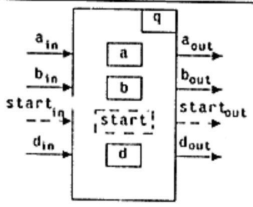
$$d = \deg A - \deg B$$

where A and B are the polynomials to be transformed by T. By the definition in Section II, T is R_A if $d \geq 0$; otherwise, T is R_B. The cell design in Fig. 6 keeps track of the value of d, and consequently is able to determine on-the-fly which transformation to perform. We specify the cell in terms of a finite state machine of three states. Operations performed by each cell during a cycle depend on the state that the cell is in. Initially, every cell is in state **initial**. Triggered by the **start** signal it will go to one of the other two states—**reduceA** or **reduceB**, and eventually return to state **initial**.

To illustrate what the code does, consider once more the systolic subarray in Fig. 4. Suppose that $d = i - j > 0$ and $b_j \neq 0$. Marching to the right-hand side together with b_j is the current value of d. Each cell upon receiving a true value from the systolic control path **start** will go to state **reduceA** (since $d > 0$). When $\overline{a}_{i-\delta} (\neq 0)$ and b_j are output from the right-most cell of the subarray, they will enter the cell to the right in the following cycle with state **reduceA** if $d \geq 0$ or **reduceB** if $d < 0$.

With $m + n + 1$ cells a systolic array based on this design can compute a GCD of any two polynomials of total degree less than $m + n + 1$. Moreover, immediately after the input of one pair of polynomials, a new pair of polynomials can enter the systolic array. That is, the systolic array can compute GCD's for multiple pairs of polynomials simultaneously, as they are being pumped through the array. Fig. 7 depicts this pipelined computation.

We assume that none of the given pairs of polynomials has x as its common factor, so their GCD's have nonzero constant terms. (Note that common factor x of two polynomials can be easily factored out from the polynomials before the computation.) With this assumption, one can deduce a GCD from the output emerging from the right-most end of the array in a straightforward way. More precisely, the constant term of the GCD is the last nonzero term coming out



```
initial:  {wait for the beginning of a GCD computation}

        begin
        a_out := a; b_out:= b; d_out := d; start_out := start;
        if start then
              begin
              if (a_in = 0) or ((b_in <> 0) and (d_in >= 0)) then
                    begin
                    state := reduceA;
                    if b_in = 0 then q := 0 else q:= a_in/b_in;
                    a := 0; b := b_in; d := d_in - 1
                    end
              else
                    begin
                    state := reduceB; q := b_in/a_in; b := 0;
                    a := a_in; d := d_in + 1
                    end
              end;
        start := start_in
        end;

reduceA:  {transform A and shift a's faster than b's}

        begin
        d_out := d; start_out := start;
        if start_in then state := initial;
        a_out := a_in - q*b_in;  b_out := b; b := b_in;
        start := start_in; d := d_in
        end;

reduceB:  {transform B and shift b's faster than a's}

        begin
        d_out := d; start_out := start;
        if start_in then state := initial;
        a_out := a; a := a_in;  b_out := b_in - q*a_in;
        start := start_in; d := d_in
        end;
```

Fig. 6. Cell definition for the design using difference of degrees.

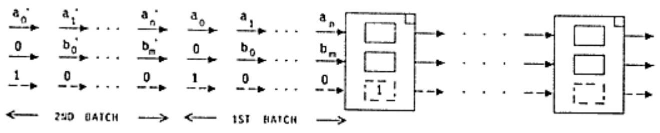

a_0' a_1' ... a_n' a_0 a_1 ... a_n
0 b_0' ... b_m' 0 b_0 ... b_m
1 0 ... 0 1 0 ... 0
2ND BATCH
1ST BATCH

Fig. 7. Multiple batches of polynomials entering the systolic array continuously.

from the array before output of the next batch of polynomials starts emerging, and the high-degree terms of the GCD are those terms that are output earlier at the same output line.

V. Systolic Array for the Extended GCD Problem

The GCD problem can be extended to find not only a greatest common divisor, $\mathrm{GCD}(A_0, B_0)$, of A_0 and B_0, but also polynomials U and V such that

$$UA_0 + VB_0 = \mathrm{GCD}(A_0, B_0).$$

More generally, for $n = \deg A_0 \geq \deg B_0$, we want to find polynomials U, V, and W such that

$$UA_0 + VB_0 = W$$

where $\deg V \leq n - k$, $\deg W < k$, and $k(\leq n)$ is some given integer greater than the degree of $\mathrm{GCD}(A_0, B_0)$. This new problem, called the *extended GCD problem*, is important for many applications including the decoder implementation for a variety of error-correcting codes. For example, finding the error location polynomial of Reed–Solomon decoding mentioned in Section I calls for solving the extended GCD problem in the general sense, with $A_0 = x^{32}, B_0$ being a given (syndrome) polynomial of degree 31, and $k = 16$ [13].

We show that U, V, and W can be computed by the same transformation sequence as for the GCD computation for A_0 and B_0. The following equations hold when $A = A_0, B = B_0$, $L = 1, M = 0, R = 0$, and $S = 1$:

$$\begin{cases} LA_0 + RB_0 = A\,, \\ MA_0 + SB_0 = B\,. \end{cases} \tag{1}$$

Replacing the first equation with a difference gives

$$\begin{cases} (L - qx^dM)A_0 + (R - qx^dS)B_0 = A - qx^dB\,, \\ MA_0 + SB_0 = B\,. \end{cases}$$

Thus, equations in (1) are invariant under the "extended" transformation R_A:

$$A := A - qx^dB, \qquad L := L - qx^dM, \qquad R := R - qx^dS\,. \tag{2}$$

Similarly, they are invariant under "extended" transformation R_B:

$$B := B - qx^dA, \qquad M := M - qx^dL, \qquad S := S - qx^dR\,. \tag{3}$$

Therefore, we can apply a sequence of these extended GCD-preserving transformations to $A(= A_0)$ and $B(= B_0)$ until A(or B) becomes $\mathrm{GCD}(A_0, B_0)$ or, for the general case, a polynomial W of degree less than k. At this time L and R (or M and S) will be the U and V that we want to compute.

Each cell of the systolic array will now perform the extended transformation R_A defined by (2) or R_B defined by (3). Consider the case when the extended transformation R_A is to be performed. At first glance, one might think that terms in L and R could be computed exactly the same way as those in A since they are defined by the same transformation [see (2)]. This scheme does not work, however, because degrees of L and R are increased by the transformation, while that of A is decreased. It is therefore necessary to "leave room," in front of the current leading terms of L and R to accommodate those higher degree terms to be acquired in the future. This implies that in the systolic array terms in L and R should travel more slowly than those in A. Based on this observation, a cell design for the extended GCD computation is given in Figs. 8 and 9. Codes involving terms in R and S are not shown in Fig. 9, as they are similar to those involving terms in L and M. To save pins in chip implementation, the current design can be trivially modified so that terms in L and M (or in R and S) will be computed in a separate pass.

Note that a cell has two internal registers for each of the **l**, **m**, **r**, and **s** data streams. Therefore, if the cell is in state **reduceA**, then a term in L or M will take twice as long to pass the cell as a term in A or B, respectively. Relatively speaking, if terms in M and S cross the cell at speed 1, then terms in L, R, and B cross at speed 2 and terms in A at speed 3. The correctness of the cell specification can be verified by

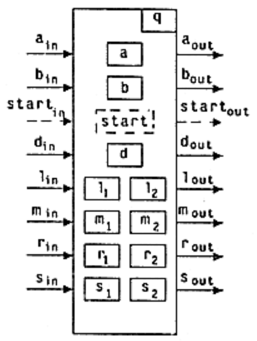


Fig. 8. Systolic cell for the extended GCD computation, a design allowing data moving at three different speeds simultaneously.

```
initial:  {wait for the beginning of a GCD computation}

        begin
        $a_{out}$ := a; $b_{out}$:= b; $d_{out}$ := d; $start_{out}$ := start;
        $l_{out}$ := $l_2$; $m_{out}$ := $m_2$;
        if start then
                begin
                if ($a_{in}$ = 0) or (($b_{in}$ <> 0) and ($d_{in}$ >= 0)) then
                        begin
                        state := reduceA;
                        if $b_{in}$ = 0 then q := 0 else q:= $a_{in}/b_{in}$;
                        a := 0; $l_1$ := 0; $l_2$ := $l_{in}$ - q*$m_{in}$;
                        b := $b_{in}$; $m_2$ := $m_1$; $m_1$ := $m_{in}$; d := $d_{in}$ - 1
                        end
                else
                        begin
                        state := reduceB; q := $b_{in}/a_{in}$; b := 0;
                        $m_1$ := 0; $m_2$ := $m_{in}$ - q*$l_{in}$;
                        a := $a_{in}$; $l_2$ := $l_1$; $l_1$ := $l_{in}$; d := $d_{in}$ + 1
                        end
                end;
        start := $start_{in}$
        end;

reduceA:  {transform A, shift a's faster than b's, and shift l's faster than m's}

        begin
        $d_{out}$ := d; $start_{out}$ := start;
        if $start_{in}$ then state := initial;
        $a_{out}$ := $a_{in}$ - q*$b_{in}$;
        $l_{out}$ := $l_2$; $l_2$ := $l_{in}$ - q *$m_{in}$; $b_{out}$ := b; b := $b_{in}$;
        $m_{out}$ := $m_2$; $m_2$ := $m_1$; $m_1$ := $m_{in}$;
        start := $start_{in}$; d := $d_{in}$
        end;

reduceB:  {transform B, shift b's faster than a's, and shift m's faster than l's}

        begin
        dout := d; $start_{out}$ := start;
        if $start_{in}$ then state := initial;
        $a_{out}$ := a; a := $a_{in}$; $l_{out}$ := $l_2$; $l_2$ := $l_1$; $l_1$ := $l_{in}$
        $b_{out}$ := $b_{in}$ - q*$a_{in}$;
        $m_{out}$ := $m_2$; $m_2$ := $m_{in}$ - q*$l_{in}$;
        start := $start_{in}$; d := $d_{in}$
        end;
```

Fig. 9. Cell definition for the systolic cell of Fig. 8 for the extended GCD computation (omitting statements involving **r**, **s**).

showing that the cell preserves the invariance of (1). Indeed, this invariance condition was used explicitly by the authors when designing the cell.

The systolic array described here for the extended GCD computation helps solve the difficult error-location problem in the decoder implementation of various error-correcting codes [12], [13]. All other portions of the decoder basically involve only polynomial evaluations, which we know can be efficiently carried out by systolic arrays using Horner's rule [8]. The CMU programmable systolic chip (PSC) mentioned earlier can implement systolic arrays for polynomial evaluation and for the extended GCD computation.

Suppose that in the Reed–Solomon code each codeword consists of 224 information bytes and 32 check bytes, so errors involving no more than 16 symbols can be corrected. We estimate that by using a linear array of 112 PSC's the Reed–Solomon decoding can be performed in a throughput of 8 million bits per second [5]. (The encoding is much easier; it requires only about 30 PSC chips to achieve the same throughput.) The fastest existing Reed–Solomon decoder with the same characteristics that we are aware of uses about 500 chips but achieves a throughput of no more than 1 million bits per second. The performance of our design is largely due to the new systolic arrays for the extended GCD problem reported in this section.

VI. Variants and Extensions

By keeping track of the beginning and end of each polynomial during the computation, it is possible to have designs without explicitly using difference of degrees of polynomials (and hence, no upper bound on this difference need be known when the cells are designed). Also, by interchanging data on the output lines $\mathbf{a}_{out}$ and $\mathbf{b}_{out}$, we can ensure that the output GCD always emerges on a fixed output line, say the $\mathbf{a}_{out}$ output line, as depicted in Fig. 1. These modifications lead to systolic algorithms whose VLSI implementations do not require pins for d_{in} and d_{out}. In one such algorithm [4], d is represented in unary as the distance between 1 bit on the **start** systolic control path and 1 bit on another systolic control path.

For some implementations division could be expensive. Fortunately, division can be totally avoided. More precisely, we can modify designs of this paper to deal with polynomials over any unique factorization domain rather than a field. Suppose that we are given polynomials $A = a_ix^i + \cdots + a_1x + a_0$ and $B = b_jx^j + \cdots + b_1x + b_0$, where $a_i \neq 0$ and $b_j \neq 0$, and that we want to reduce the degree of A. Instead of performing transformation R_A,

$$\overline{A} := A - q \cdot x^dB$$

where $q = a_i/b_j$ and $d = i - j$, we perform

$$\overline{A} := \alpha \cdot A - \beta \cdot x^dB$$

where $\alpha = b_j$ and $\beta = a_i$. This leads to designs requiring no division. The cell corresponding to Fig. 4 will now store values of α and β rather than q when in state **initial**, and perform two multiplications during each cycle when in state **reduceA** or **reduceB**.

All designs presented in this paper assume that high degree terms enter a systolic array first. Actually, a set of dual designs exists where low degree terms enter a systolic array first. This can be convenient in applications where GCD computations are linked to other computations which operate on polynomials with lower degrees terms first. A related result concerns systolic arrays for computing *integer* GCD's which (must) operate on low-order bits first. The systolic integer GCD algorithm will find the GCD of two n-bit binary integers with at most $4n$ systolic cells, which are just simple finite-state machines operating on single bits [2], [4].

Finally, we note that the systolic GCD array described in Section IV-B can be modified to perform polynomial division. The modified array simply performs the beginning portion of the Euclidean algorithm until variable d changes its sign for the first time. With this scheme, terms in the dividend, divisor, and remainder all move, but terms in the quotient stay, one term in each cell. This is in contrast to another systolic design for polynomial division [9], where terms in the divisor stay whereas terms in the dividend, quotient, and remainder all move.

REFERENCES

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1975.

[2] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for linear-time GCD computation," in *VLSI '83*, F. Anceau and E. J. Aas, Eds. Amsterdam: North-Holland, Aug. 1983, pp. 145–154.

[3] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun, "Fast solution of Toeplitz systems of equations and computation of Pade approximants," *J. Algorith.*, vol. 1, no. 3, pp. 259–295, Sept. 1980.

[4] R. P. Brent, H. T. Kung, and F. T. Luk, "Some linear-time algorithms for systolic arrays," in *Proc. IFIP 9th World Comput. Cong.*, Sept. 1983, pp. 865–876.

[5] A. L. Fisher, H. T. Kung, L. M. Monier, and Y. Dohi, "Architecture of the PSC: A programmable systolic chip," in *Proc. 10th Annu. Int. Symp. on Comput. Arch.*, June 1983, pp. 48–53.

[6] A. L. Fisher, H. T. Kung, L. M. Monier, H. Walker, and Y. Dohi, "Design of the PSC: A programmable systolic chip,"in *Proc. 3rd Caltech Conf. on Very Large Scale Integration*, R. Bryant, Ed. Rockville, MD: Computer Science Press, Mar. 1983, pp. 287–302.

[7] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Reading, MA: Addison-Wesley, 1981, 2nd ed.

[8] H. T. Kung, "Special-purpose devices for signal and image processing: An opportunity in VLSI," in *Proc. SPIE, Vol. 241, Real-Time Signal Processing III*, Soc. Photo-Opt. Instrument. Eng., July 1980, pp. 76–84.

[9] ——, "Use of VLSI in algebraic computation: Some suggestions," in *Proc. 1981 ACM Symp. on Symbolic and Algebraic Computation*, P. S. Wang, Ed., Aug. 1981, pp. 218–222.

[10] ——, "Why systolic architectures?" *Computer*, vol. 15, pp. 37–46, Jan. 1982.

[11] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proc. 1978*, I. S. Duff and G. W. Stewart, Eds. Soc. Indus. Appl. Math., 1979, pp. 256–282; see also, ——, in C. A. Mead and L. A. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 1980, pp. 37–46, Sect. 8.3.

[12] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes.* Amsterdam: North-Holland, 1977.

[13] R. J. McEliece, *Encyclopedia of Mathematics and Its Applications. Volume 2: The Theory of Information and Coding.* Reading, MA: Addison-Wesley, 1977.

Richard P. Brent (M'72–SM'83) was born in Melbourne, Australia, on April 20, 1946. He received the B.Sc. (hons) degree in mathematics from Monash University, Australia, in 1968, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, in 1970 and 1971, respectively.

From 1971 to 1972 he was employed in the Department of Mathematical Sciences, IBM T. J. Watson Research Center, Yorktown Heights, NY. Since 1972 he has been at the Australian National University, Canberra, Australia, where he is currently Professor of Computer Science. His research interests include VLSI design, computer arithmetic, analysis of algorithms, and computational complexity.

H. T. Kung received the Ph.D. degree from Carnegie-Mellon University, Pittsburgh, PA.

He joined the faculty of Carnegie-Mellon University in 1974, and is now a Professor of Computer Science. He leads a research team in the design and implementation of high-performance VLSI systems. In 1981 he was a fulltime Architecture Consultant to ESL, Inc., a subsidiary of TRW. He is interested in paradigms of mapping algorithms and systems directly onto VLSI.

Dr. Kung is Guggenheim Fellow (1983–84), and has served on program committees for numerous VLSI and computer science conferences.