VLSI '83
F. Anceau and E.J. Aas (eds.)
Elsevier Science Publishers B.V. (North-Holland)

SYSTOLIC VLSI ARRAYS FOR LINEAR-TIME GCD COMPUTATION

R.P. Brent
Department of Computer Science
Australian National University
Canberra, A.C.T. 2601
Australia

H.T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
U.S.A.

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 hardware solutions to both the GCD problem and the extended GCD problem. The systolic arrays have been implemented on the CMU programmable systolic chip (PSC) to demonstrate its application to the decoder implementation for Reed-Solomon codes. The integer GCD problem is also considered, and it is shown that a linear systolic array of O(n) cells can compute the GCD of two n-bit integers in time O(n).

1. INTRODUCTION

The polynomial GCD problem is to compute a greatest common divisor of any two non-zero polynomials. The problem is fundamental to algebraic and symbolic computations (see, e.g. [7]) , and to the decoder implementation for a variety of error-correcting codes (see, e.g. [10, 11]. Many 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 hardware implementation, these previously known algorithms are all too irregular or/and too complex to be useful. For example the Euclidean algorithm involves a sequence of complicated divisions of polynomials whose sizes can be only determined during the computation. This paper describes some simple and regular systolic structures that can lead to efficient hardware 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 $m + n$. Figure 1-1 illustrates that the systolic array inputs the coefficients of the given polynomials, $\Sigma_{i=0}^{n} a_i x^i$ and $\Sigma_{i=0}^{m} b_i x^i$, 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 as we shall see later 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 entering 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.

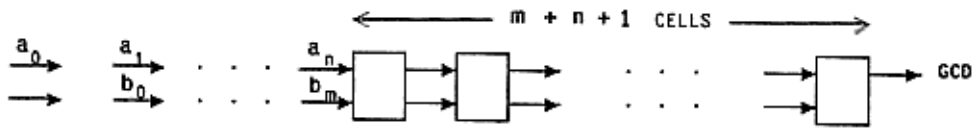


Figure 1-1: Systolic GCD Array.

Like many other systolic designs (see, e.g. [9]), the systolic GCD arrays of this paper are suitable for VLSI implementation and can achieve high throughputs. 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 PSC chip [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 Sections 5 and 7).

The integer GCD problem is analogous to the polynomial GCD problem, but is more complicated because of the problem of carries. Nevertheless, in Section 6 we outline how systolic arrays can be used to solve the integer GCD problem.

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. Hopefully 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 in either PASCAL or LISP, and thoroughly tested by simulation.

2. 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 $\bar{A}$ and $\bar{B}$, with the property that a GCD of A and B is also a GCD of $\bar{A}$ and $\bar{B}$, 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 [3] we point out that with straightforward modifications our designs will work over any unique factorization domain and thus require no divisions. Throughout the paper polynomials are represented by upper-case letters and their coefficients by corresponding lower-case letters. In the following we define two GCD-preserving transformations, R_A and R_B. Let

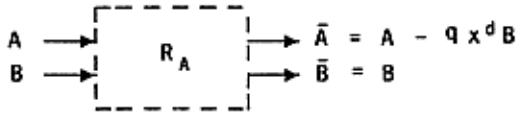
$$A = a_i x^i + \ldots + a_1 x + a_0$$

and

$$B = b_j x^j + \ldots + b_1 x + b_0$$

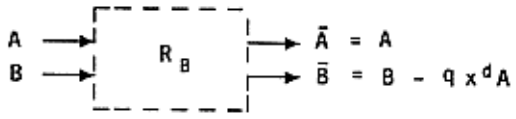
be the two polynomials to be transformed, where $a_i \neq 0$ and $b_j \neq 0$. Depending on the value of i-j, one of the following two transformations is applied to A and B.

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



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

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



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

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

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

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

$$\deg \bar{B} \leq \deg B - 1.$$

For notational convenience, we assume that the degree of the zero polynomial is -1. Thus, if both A and B are polynomials of degree zero, i.e., they are nonzero constants in the underlying field F, then transformation R_A reduces A to the zero polynomial, which has degree -1.

3. 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. Then as the preceding section shows, we can 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 $(T_1, T_2, \ldots, T_k)$ 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", $T_1, T_2, \ldots, T_k$, in the left-to-right direction, and get transformed at each stage accordingly; when they 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 Figure 3-1.

Suppose that, for each $i = 1, \ldots, k$, transformation T_i reduces the degree of A_0 or B_0 by δ_i, where δ_i is some positive integer. Then we call δ_i the *reduction value* of T_i. Since the total degree of A_0 and B_0 at the beginning of the GCD computation

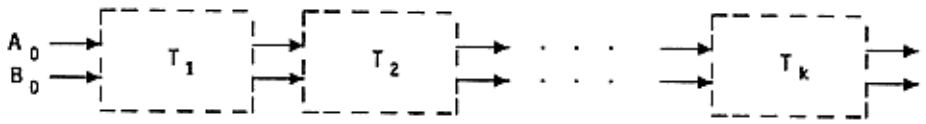


Figure 3-1: Function of the transformation sequence.

is n + m, the sum of reduction values over $i = 1, \ldots, k$ obeys the important relation: $\sum_{i=1}^{k} \delta_i \leq n + m + 1$.

4. SYSTOLIC GCD ARRAY

In this section we specify a systolic array of n + m + 1 cells that can compute a GCD of any two nonzero polynomials A_0 and B_0 of degrees no more than n and m, respectively.

Consider the transformation sequence $(T_1,T_2,\ldots,T_k)$ for the GCD computation for A_0 and B_0. We shall show that for each $i = 1,\ldots,k$, 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 as a result it can compute a GCD of A_0 and B_0. This is illustrated in Figure 4-1.

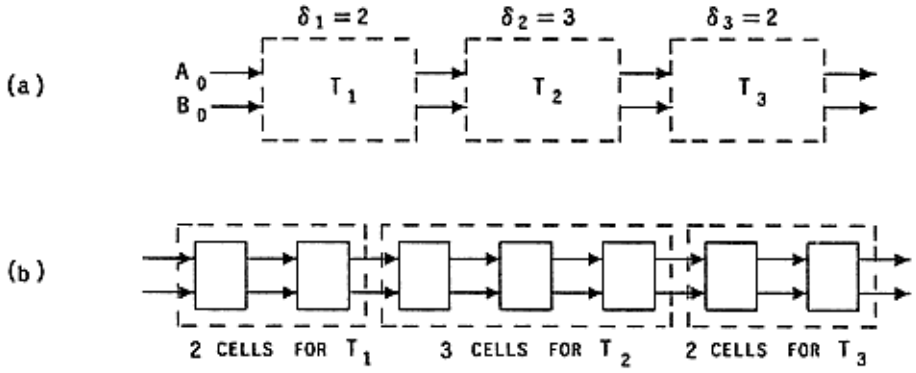
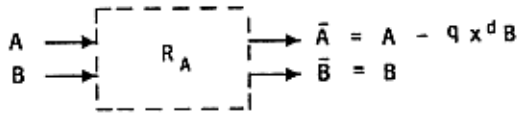


Figure 4-1: (a) Transformation sequence, and (b) its realization by three concatenated systolic subarrays.

4.1 The basic idea (for realizing a single transformation)

Let T be any transformation in the transformation sequence $(T_1,T_2,\ldots,T_k)$, and δ its reduction value. We illustrate how a subarray with δ cells can realize T, assuming that by some other methods (see, for example, Section 4.2 below) we know which one of R_A and R_B transformation T is.

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



where

$$A = a_i x^i + \ldots + a_1 x + a_0, \quad (a_i \neq 0),$$
$$B = b_j x^j + \ldots + b_1 x + b_0, \quad (b_j \neq 0),$$
$$q = a_i/b_j,$$

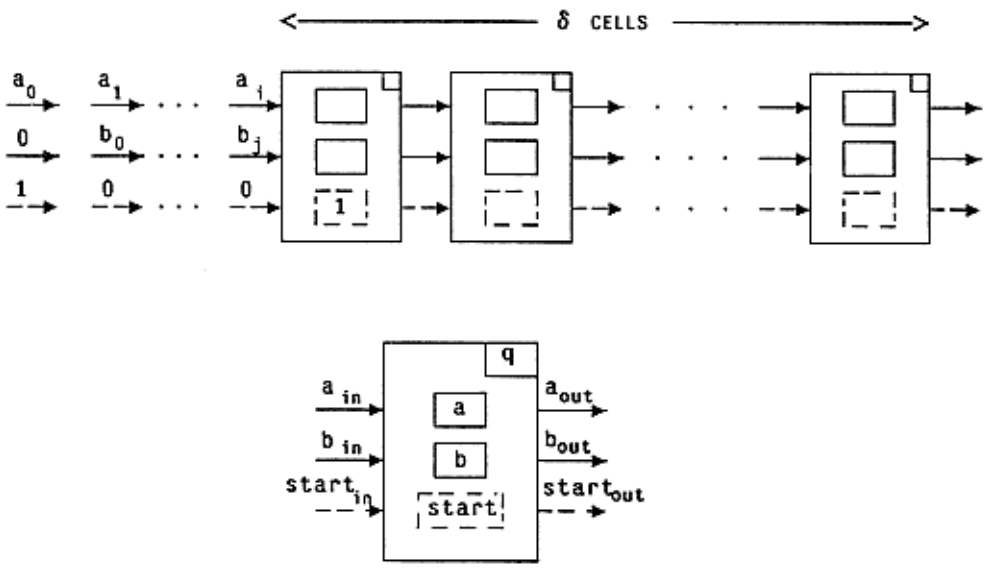
and

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

Note that either $\bar{A} = 0$, or

$$\bar{A} = \bar{a}_{i-\delta} x^{i-\delta} + \ldots + \bar{a}_1 x + \bar{a}_0,$$

where $\bar{a}_{i-\delta} \neq 0$. The systolic subarray, together with the operations performed by each of its cells at every cycle, for realizing T is described in Figure 4-2.




```
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
```

Figure 4-2: Systolic subarray and its cell definition for realizing a transformation R_A.

Terms in A and B move through the subarray in a serial manner, 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. Figure 4-2 assumes that i = j + 1 to illustrate the point that when i > j as many as i - j zeros 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 Figure 4-2 (and other Figures below) 1-bit wide systolic control paths and the associated latches 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 $\bar{A}$ in subsequent cycles. The q's computed by other cells, however, are always zeros, since terms of $\bar{A}$ that have degrees higher than $n - \delta$ vanish. 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 $\bar{a}_{i-\delta}$ of $\bar{A}$ will depart from the right-most cell at the same cycle as b_j, the nonzero leading term of $\bar{B}$. Thus $\bar{a}_{i-\delta}$ and b_j are ready to enter another subarray of cells to the right for realizing whatever transformation follows T.

There is no need to keep track of the δ value in the systolic subarray. If $\bar{A}$ is nonzero, the realization of the transformation following T starts automatically at the first cell that sees a nonzero input, i.e., $\bar{a}_{i-\delta}$, appearing at its input line denoted by a_{in}. If $\bar{A}$ 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 to be

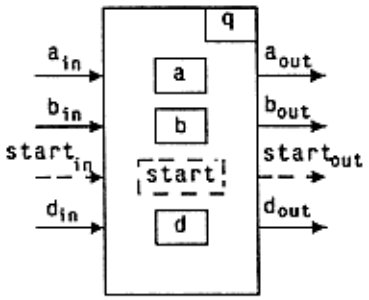
output from the right-most cell, and they will form terms in the GCD that we wish to compute.

4.2 Design using difference of degrees

We have seen that a systolic subarray with its cells defined by Figure 4-2 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. Then by the definition in Section 2, T is R_A if $d \geq 0$, otherwise T is R_B. The cell design in Figure 4-3 keeps track of the value of d, and consequently is able to determine on-the-fly




```
 initial: {wait for the beginning of a GCD computation}
     begin
     aout := a; bout := b; dout := d; startout := start;
     if start then
         begin
         if (ain = 0) or ((bin <> 0) and (din >= 0)) then
             begin
             state := reduceA;
             if bin = 0 then q := 0 else q := ain/bin; a := 0;
             b := bin; d := din - 1
             end
         else
             begin
             state := reduceB; q := bin/ain; b := 0;
             a := ain; d := din + 1
             end
         end;
     start := startin
     end
 reduceA: {transform A and shift a's faster than b's}
     begin
     dout := d; startout := start;
     if startin then state := initial;
     aout := ain - q*bin; bout := b; b := bin;
     start := startin; d := din
     end;
 reduceB: {transform B and shift b's faster than a's}
     begin
     dout := d; startout := start;
     if startin then state := initial;
     aout := a; a := ain; bout := bin - q*ain;
     start := startin; d := din
     end;
```

Figure 4-3: Cell definition for the design using difference of degrees.

which transformation to perform. To cope with the increased complexity in the cell definition, we specify the cell in terms of a finite state machine. There are a total 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 Figure 4-2. Suppose that $d = i - j \geq 0$ and $b_j \neq 0$. Marching to the right 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 $d \geq 0$). When $\bar{a}_{i-\delta}(\neq 0)$ 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 GCDs for multiple pairs of polynomials simultaneously, as they are being pumped through the array.

We assume that none of the given pairs of polynomials have x as their common factor, so their GCDs have nonzero constant terms. (A common factor x of two polynomials can be factored out from the polynomials before the computation). With this assumption, one can easily tell what the GCD is from the output emerging from the right-most end of the array. More precisely, the constant term of the GCD is the last nonzero term coming out 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 on the same output line. When it is inconvenient to assume that the GCDs have nonzero constant terms, one can either keep the degrees explicitly (instead of just their difference) or have a "stop" bit to indicate where a_0 and b_0 are.

5. SYSTOLIC ARRAY FOR THE EXTENDED GCD PROBLEM

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

$$UA_0 + VB_0 = GCD(A_0,B_0).$$

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 $GCD(A_0,B_0)$. We call this new problem the *extended GCD problem*. It is important for many applications including the decoder implementation for a variety of error-correcting codes. For example, finding the error location polynomial for Reed-Solomon decoding 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 $k = 16$ [11]. This is the most difficult step in the decoder implementation, since all other steps basically involve only polynomial evaluations, which can be efficiently carried out in hardware using Horner's rule [8].

The extended GCD problem is usually solved by the so-called extended Euclidean algorithm (see, e.g., [1]). Based on a similar principle, we notice that U, V and W can be computed by the same transformation sequence as for the GCD computation for A_0 and B_0. Observe that the following equations hold when $A = A_0$, $B = B_0$, $L = 1$, $M = 0$, $R = 0$ and $S = 1$:

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

Replacing the first equation with a difference gives:

$$(L - qx^d M)A_0 + (R - qx^d S)B_0 = A - qx^d B,$$
$$MA_0 + SB_0 = B.$$

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

$$A := A - qx^d B,\ L := L - qx^d M,\ R := R - qx^d S\ . \qquad (2)$$

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

$$B := B - qx^d A,\ M := M - qx^d L,\ S := S - qx^d R\ . \qquad (3)$$

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 $GCD(A_0,B_0)$ 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.

From the discussion above, we see that each cell of the systolic array should 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 method does not work, however, because degrees of L and R are increased by the transformation, whereas that of A is decreased. It is therefore important 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 means 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 [3].

Many variants of the designs presented above are mentioned in [3]. For example, we can avoid explicitly using the difference of degrees, we can avoid divisions, and we can insist that the GCD always emerges on the aout line of the right-most cell.

6. A SYSTOLIC ARRAY FOR INTEGER GCD COMPUTATION

Consider now the problem of computing the greatest common divisor GCD (a, b) of two positive integers a and b, assumed to be given in the usual binary representation. Our aim is to compute GCD (a, b) in time O(n) on a linear systolic array, where n is the number of bits required to represent a and b (say $a < 2^n$, $b < 2^n$). The significant difference between integer and polynomial GCD computations is that carries have to be propagated in the former, but not in the latter.

The classical Euclidean algorithm [7] may be written as:

$$\text{while } b \neq 0 \text{ do } \begin{pmatrix} a \\ b \end{pmatrix} := \begin{pmatrix} b \\ a \bmod b \end{pmatrix} \text{ ; GCD := a.}$$

This is simple, but not attractive for a systolic implementation because the division in the inner loop takes time $\Omega(n)$. More attractive is the less familiar "binary" Euclidean algorithm [2, 7] which uses only additions, shifts and comparisons:

```
{assume a, b odd for simplicity}
t := |a - b|;
while t ≠ 0 do
    begin
    repeat t := t div 2 until odd(t);
    if a ≥ b then a := t else b := t;
    t := |a - b|
    end;
GCD := a.
```

However, if we try to implement the binary Euclidean algorithm on a systolic array we encounter a serious difficulty: the test "if a ≥ b ..." may require knowledge of all bits of a and b, so again the inner loop takes time Ω(n) in the worst case.

6.1 Algorithm PM

Algorithm PM (for "plus-minus"), like the classical and binary Euclidean algorithms, finds the GCD of two n-bit integers a and b in O(n) iterations, but we shall see that it can be implemented in a pipelined fashion (least significant bits first) on a systolic array. Before defining Algorithm PM we consider the "precursor" algorithm defined in Figure 7-1. Using the assertions contained in braces, it is easy to prove that the algorithm terminates in at most 2n + 1 iterations (since α + β strictly decreases at each iteration of the repeat block except

```
{assume a odd and b ≠ 0, |a| ≤ 2^n, |b| ≤ 2^n}
α := n;  β := n;
    repeat
    while even(b) do begin b := b div 2;  β := β - 1 end;
    {now b odd, |b| ≤ 2^β}
    if α ≥ β then begin swap (a, b); swap (α, β) end; {"swap" has obvious
                                                       meaning}
    {now α ≤ β, |a| ≤ 2^α, |b| ≤ 2^β, a odd, b odd}
    if ((a + b) mod 4) = 0 then b := (a + b) div 2 else b := (a - b) div 2;
    {now b even, |b| ≤ 2^β}
    until b = 0;
GCD := |a|.
```

Figure 6-1: Precursor to Algorithm PM

possibly the first). Moreover, since all transformations on a and b are GCD-preserving, the GCD is computed correctly.

It is not necessary to maintain α and β: all we need is their difference δ = α - β (analogous to the difference of degrees in Section 4.2). This observation leads to Algorithm PM, which is defined in Figure 6-2.

```
{assume a odd, b ≠ 0}
δ := 0;
    repeat
    while even(b) do begin b := b div 2; δ := δ + 1 end;
    if δ ≥ 0 then begin swap (a, b);  δ := -δ end;
    if ((a + b) mod 4) = 0 then b := (a + b) div 2 else b := (a - b) div 2
    until b = 0;
GCD := |a|.
```

Figure 6-2: Algorithm PM

6.2 Implementation of Algorithm PM on a systolic array

For implementation on a systolic array, algorithm PM has a great advantage over the classical or binary Euclidean algorithms: the tests in the inner loop involve only the two least-significant bits of a and b. Hence, a cell can perform these tests before the high-order bits of a and b reach it via cells to its left. (The termination criterion "until b = 0" is not a problem.)

We have to consider implementation of operations on δ in algorithm PM. The only operations required are "δ := δ + 1", "δ := -δ", and "if δ ≥ 0 ...". Rather than represent δ in binary, we choose a "sign and magnitude unary" representation, i.e. keep sign (δ) and |δ| separate, and represent ε = |δ| in unary as the distance between 1-bits and two streams of bits. With this representation all required operations on δ can be pipelined.

After some optimisations and modifications to permit even inputs, we obtain a systolic integer GCD cell with six input ports (each one bit wide), six output ports, and twelve internal state bits. If at least 3.1106n such cells are connected together, they can compute $\pm$ GCD (a, b) for any n-bit inputs a and b. Further details, including the cell definition, may be found in [4, 5].

7. IMPLEMENTATION OF A REED-SOLOMON DECODER

The PSC chip [6] has been fabricated in nMOS and is functionally working at a 4MHz clock rate. We estimate that Reed-Solomon decoding [10, 11] can be performed with a throughput of 8 million bits per second using a linear array of 112 PSCs (assuming that each codeword has 224 information bytes and 32 check bytes, so errors involving no more than 16 symbols can be corrected). The fastest existing decoder known to us uses about 500 chips but achieves a throughput of no more than one million bits per second. The better performance of our design is largely due to our use of systolic arrays for the extended GCD problem as described in Section 5.

ACKNOWLEDGEMENT

The research 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 (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

REFERENCES

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, Mass., 1974).

[2] Brent, R.P., Analysis of the binary Euclidean algorithm, in: Traub, J.F. (ed.), New Directions and Recent Results in Algorithms and Complexity (Academic Press, New York, 1976).

[3] Brent, R.P., and Kung, H.T., Systolic VLSI Arrays for Polynomial GCD Computation, Report CMU-CS-82-118, Dept. of Comp. Sc., Carnegie-Mellon Univ. (May 1982).

[4] Brent, R.P. and Kung, H.T., A Systolic VLSI Array for Integer GCD Computation, Report TR-CS-82-11, Dept. of Comp. Sc., Australian National Univ. (Dec. 1982).

[5] Brent, R.P., Kung, H.T. and Luk, F.T., Some linear-time algorithms for systolic arrays, in: Mason, R.E.A. (ed), Information Processing 83 (North-Holland, Amsterdam, 1983), to appear.

[6] Fisher, A.L., Kung, H.T., Monier, L.M., Walker, H. and Dohi, Y., Design of the PSC: A programmable systolic chip, Proc. Third Caltech Conference on VLSI, Computer Science Press (March 1983), 287-302.

[7] Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 2nd edition (Addison-Wesley, Reading, Mass., 1981).

[8] Kung, H.T., Special-purpose devices for signal and image processing: an opportunity in VLSI, Proc. of the SPIE, Vol. 241, Real-Time Signal Processing III, pp 76-84, The Society of Photo-Optical Instrumentation Engineers (July 1980).

[9] Kung, H.T., Why systolic architectures, IEEE Computer 15, 1 (Jan. 1982), 37-46.

[10] MacWilliams, F.J. and Sloane, N.J., The Theory of Error Correcting Codes, (North-Holland, Amsterdam, 1977).

[11] McEliece, R.J., Encyclopedia of Mathematics and Its Applications, Volume 2: The Theory of Information and Coding (Addison-Wesley, Reading, Mass, 1977).