Comput. Math. Applic. Vol. 14, No. 4, pp. 233–238, 1987
Printed in Great Britain.

0097-4943/87 $3.00 + 0.00

A SYSTOLIC ALGORITHM FOR EXTENDED GCD COMPUTATION†

A. W. BOJANCZYK‡ and R. P. BRENT
Centre for Mathematical Analysis, The Australian National University, GPO Box, Canberra, ACT 2601, Australia

(*Received* 28 *March* 1986)

Communicated by E. Y. Rodin

1. INTRODUCTION

For given positive integers a, b the extended GCD problem is to find integers u, v, g such that

$$ua + vb = g, \tag{1}$$

where g is the greatest common divisor of a and b. There are many number-theoretic applications of the extended GCD, for example in error correcting codes [1] and integer factorization [2, 3]. A common case [3] is that g is known to be 1 and we want to compute a multiplicative inverse modulo b, i.e. compute u such that $ua = 1 \pmod{b}$.

We describe an efficient algorithm which requires $O(n^2)$ bit operations (where a and b may be represented in n bits) but is easily pipelined. Thus, using $O(n)$ pipeline stages or $O(n)$ systolic cells, the result can be computed in time $O(n)$. Our algorithm is described in Section 2. In Section 3 we discuss an alternative algorithm proposed by Purdy [4]. An algorithm for reducing rational numbers to standard form is given in Section 4, and some conclusions are stated in Section 5.

2. ALGORITHM PM

In this section we describe Algorithm EPM, which is an extension of Algorithm PM of Brent and Kung [5–7]. Algorithm PM is related to the binary Euclidean algorithm [8, 9] but is more easily pipelined or implemented on a systolic array [8, 9].

Given two odd integers a_0 and b_0, Algorithm PM applies a sequence of GCD-preserving transformations

$$\begin{bmatrix} a_k \\ b_k \end{bmatrix} = \mathrm{T}_k \begin{bmatrix} a_{k-1} \\ b_{k-1} \end{bmatrix}, \quad k = 1,2,\ldots,$$

until, for sufficiently large m, $b_m = 0$ and $|a_m| = g = \mathrm{GCD}\,(a_0, b_0)$. The transformations T_k may be represented by 2×2 matrices of the form

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1/2 \end{bmatrix}, \quad \mathbf{M}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{M}_3 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{M}_4 = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix}.$$

Algorithm EPM proceeds in the same manner as Algorithm PM, but maintains a matrix

$$\mathbf{U}_k = \begin{bmatrix} \mu_k & \lambda_k \\ \gamma_k & \eta_k \end{bmatrix}$$

†A preliminary version of this paper was presented at the *9th Australian Computer Science Conference*, Canberra (Jan. 1986).
‡Present address: Department of Computer Science, Washington University, St Louis, MO 63130, U.S.A.

of integers such that

$$\mathbf{U}_k \begin{bmatrix} a_0 \\ b_0 \end{bmatrix} = \begin{bmatrix} a_k \\ b_k \end{bmatrix}. \tag{2}$$

Thus, after m transformations

$$\mu_m a_0 + \lambda_m b_0 = a_m = \pm \mathrm{GCD}(a_0, b_0).$$

Clearly, we start with

$$\mathbf{U}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Before the kth transformation, we have (by induction on k)

$$\mathbf{U}_{k-1} \begin{bmatrix} a_0 \\ b_0 \end{bmatrix} = \begin{bmatrix} a_{k-1} \\ b_{k-1} \end{bmatrix}$$

and GCD $(a_{k-1}, b_{k-1}) = \mathrm{GCD}(a_0, b_0)$. The transformation T_k is determined as in Algorithm PM, and

$$\begin{bmatrix} a_k \\ b_k \end{bmatrix} = \mathrm{T}_k \begin{bmatrix} a_{k-1} \\ b_{k-1} \end{bmatrix}.$$

Thus, we can take $\mathbf{U}_k = \mathrm{T}_k \mathbf{U}_{k-1}$ to maintain equation (2). However, this is only possible if $\mathrm{T}_k \mathbf{U}_{k-1}$ is an integer matrix. The only case which can give non-integral entries in $\mathrm{T}_k \mathbf{U}_{k-1}$ is $\mathrm{T}_k = \mathbf{M}_1$. Since a_0 and b_0 are odd, this can only arise if a_{k-1} is odd and b_{k-1} is even (since $b_k = b_{k-1}/2$ is an integer). Hence, we define

$$\mathbf{U}_k = \begin{cases} \begin{bmatrix} \mu_{k-1} & \lambda_{k-1} \\ (\gamma_{k-1} + b_0)/2 & (\eta_{k-1} - a_0)/2 \end{bmatrix}, & \text{if } \mathrm{T}_k = \mathbf{M}_1 \text{ and } \gamma_{k-1} \text{ is odd,} \\ \mathrm{T}_k \mathbf{U}_{k-1}, & \text{otherwise,} \end{cases}$$

and it is easy to verify that $\mathbf{U}_k$ is an integer matrix satisfying expression (2).

To ensure convergence in $O(n)$ steps, where n is number of bits required to represent a_0 and b_0 in binary, we define α_k and β_k as in Algorithm PM. That is, we define $\alpha_0 = \beta_0 = n$ and maintain α_k and β_k so that the conditions $|a_k| \leqslant 2^{\alpha_k}$ and $|b_k| \leqslant 2^{\beta_k}$ are satisfied. By considering the effect of the transformation T_k, it is sufficient to define

$$(\alpha_k, \beta_k) = \begin{cases} (\alpha_{k-1}, \beta_k - 1), & \text{if } \mathrm{T}_k = \mathbf{M}_1, \\ (\beta_{k-1}, \alpha_{k-1}), & \text{if } \mathrm{T}_k = \mathbf{M}_2, \\ [\alpha_{k-1}, \max(\alpha_{k-1}, \beta_{k-1}) + 1], & \text{otherwise.} \end{cases}$$

We now define Algorithm EPM formally by a Pascal-like procedure in which the subscript k is omitted. For simplicity we assume that a_0 and b_0 are odd; otherwise we may divide by their common factor 2^j, giving a_0' and b_0' say, and if a_0' or b_0' is even we replace it by $a_0' + b_0'$.

```
begin {Algorithm EPM, assuming a0, b0 odd}
[a]  := [a0];
[b]     [b0]

[μ  λ]  := [1  0];
[γ  η]     [0  1]

α := n; β := n;
repeat
  while even(b) do {transformation M1}
    begin
      b := b div 2; β := β − 1;
```

if odd(γ) **then** $\begin{bmatrix}\gamma\\ \eta\end{bmatrix} := \begin{bmatrix}(\gamma + b_0) \text{ div } 2\\ (\eta - a_0) \text{ div } 2\end{bmatrix}$

else $\begin{bmatrix}\gamma\\ \eta\end{bmatrix} := \begin{bmatrix}\gamma \text{ div } 2\\ \eta \text{ div } 2\end{bmatrix}$

end;
if $\alpha \geqslant \beta$ **then** {transformation $\mathbf{M}_2$}
begin

$$\begin{bmatrix}a\\ b\end{bmatrix} := \begin{bmatrix}b\\ a\end{bmatrix}; \quad \begin{bmatrix}\alpha\\ \beta\end{bmatrix} := \begin{bmatrix}\beta\\ \alpha\end{bmatrix}; \quad \begin{bmatrix}\mu & \lambda\\ \gamma & \eta\end{bmatrix} := \begin{bmatrix}\gamma & \eta\\ \mu & \lambda\end{bmatrix}$$

end;
if $((a + b) \bmod 4) = 0$ **then** {transformation $\mathbf{M}_3$}

$$\begin{bmatrix}b\\ \gamma\\ \eta\end{bmatrix} := \begin{bmatrix}b\\ \gamma\\ \eta\end{bmatrix} + \begin{bmatrix}a\\ \mu\\ \lambda\end{bmatrix}$$

else {transformation $\mathbf{M}_4$}

$$\begin{bmatrix}b\\ \gamma\\ \eta\end{bmatrix} := \begin{bmatrix}b\\ \gamma\\ \eta\end{bmatrix} - \begin{bmatrix}a\\ \mu\\ \lambda\end{bmatrix};$$

$\beta := \beta + 1$ {$\beta \geqslant \alpha$ here}
until $b = 0$ {now GCD$(a_0, b_0) = \text{abs}(a)$, $\mu * a_0 + \lambda * b_0 = a$}
end.

Algorithm EPM

As in Algorithm PM, the variables α and β may be replaced by their difference $\delta = \alpha - \beta$. It is clear that the arithmetic operations required by Algorithm EPM can be pipelined if a_0 and b_0 are given in two's complement binary, least-significant bits first. (For the method by which operations on δ and testing the termination criterion may be pipelined, see Ref. [5, 6].)

Initially $\alpha + \beta = 2n$, and each iteration of the "repeat" loop (excepting the first) reduces $\alpha + \beta$, because there are at least two transformations $\mathbf{M}_1$ followed by zero or one of $\mathbf{M}_2$ and one of $\mathbf{M}_3$ or $\mathbf{M}_4$. Thus, $m = 8n + O(1)$ transformations suffice to reduce β to -1, and hence b to 0. In fact, each transformation $\mathbf{M}_2$ can be combined with the immediately following $\mathbf{M}_3$ or $\mathbf{M}_4$, reducing m to $6n + O(1)$, and from numerical evidence we conjecture that $m = 4n + O(1)$ transformations are sufficient.

On termination of Algorithm PM it is possible that $\max(|\mu|, |\lambda|) \geqslant \max(|a_0|, |b_0|)$. However, it is easy to prove by induction that

$$\max(|\mu|, |\lambda|) < (3/2)\max(|a_0|, |b_0|),$$

so at most one reduction of the form

$$\begin{bmatrix}\mu\\ \lambda\end{bmatrix} := \begin{bmatrix}\mu\\ \lambda\end{bmatrix} \pm \begin{bmatrix}-b_0\\ a_0\end{bmatrix}$$

will give $\max(|\mu|, \lambda|) < \max(|a_0|, |b_0|)$ while maintaining the relation $\mu a_0 + \lambda b_0 = \pm \text{GCD}(a_0, b_0)$.

3. PURDY'S ALGORITHM

A different method for computing the extended GCD of two n-bit integers a_0 and b_0 has been proposed by Purdy [4], and will be outlined here. For simplicity, we may assume that the GCD of a_0 and b_0 is odd.

Purdy's algorithm has two stages, a "forward sweep" and a "backward sweep". In the forward sweep the GCD of a_0 and b_0 is computed by applying a finite sequence of GCD preserving

transformations $(T_1, \ldots, T_M)$ and a record is kept indicating what transformation was used in each step.

The transformations are defined as follows:

$$(a_{i+1}, b_{i+1}) = T_{i+1}(a_i, b_i) = \begin{cases} (a_i/2, b_i) & \text{if } a_i \text{ is even } (1^0) \\ (a_i, b_i/2) & \text{if } b_i \text{ is even } (2^0) \\ [(a_i + b_i)/2, (a_i - b_i)/2], & \text{otherwise } (3^0). \end{cases}$$

The forward sweep terminates when one of the numbers a_i or b_i becomes zero, the other being $\pm\text{GCD}\,(a_0, b_0)$. The type of the transformation T_i used in step i is recorded in the $(M+1)$-vector **C**, i.e. $\mathbf{C}[i] = 1$, 2 or 3 if T_i is of type 1^0, 2^0 or 3^0, respectively. In addition, $\mathbf{C}[M+1] = 1$ if $a_M = 0$ and $\mathbf{C}[M+1] = 2$ if $b_M = 0$.

Coefficients u_0 and v_0, such that

$$a_0 u_0 - b_0 v_0 = g,$$

are computed in the backward sweep. This is achieved by recovering the sequence $\{(a_i, b_i)\}$, $i = M, \ldots, 1$, and computing another sequence $\{(u_i, v_i)\}$ such that for each i

$$a_i u_i - b_i v_i = g.$$

The number of steps in both sweeps is the same but the backward sweep is computationally more expensive. The algorithm for the backward sweep is given below.

Backward sweep

```
{Given the vector C and g (which is odd) compute u0 and v0 such that a0*u0 - b0*v0 = g.}
{Starting values for a[M], b[M], u[M], v[M]}
if C[M+1] = 1 then
  begin
    a[M] := 0;  b[M] := g;
    u[M] := 0;  v[M] := -1;
  end
else
  begin
    a[M] := g;  b[M] := 0;
    u[M] := 1;  v[M] := 0;
  end;
{Main loop}
for k := M down to 2 do
  begin
    case C[k] of
1:  begin
      a[k-1] := 2 * a[k];  b[k-1] := b[k];
      if odd(u[k]) then
      begin
        u[k-1] := (u[k] + b[k]) div 2;  v[k-1] := v[k] + a[k];
      end else
      begin
        u[k-1] := u[k] div 2;  v[k-1] := v[k];
      end;
    end;
2:  begin
      b[k-1] := 2 * b[k];  a[k-1] := a[k];
      if odd(v[k]) then
```

```
      begin
        v[k-1] := (v[k] + a[k]) div 2;   u[k-1] := u[k] + b[k];
      end else
      begin
        v[k-1] := v[k] div 2;
        u[k-1] := u[k];
      end;
    end;
3:  begin
      a[k-1] := a[k] + b[k];   b[k] := a[k] - b[k];
      if odd(u[k] - v[k]) then
      begin
        u[k-1] := (u[k] - v[k] + b[k-1]) div 2;
        v[k-1] := (-u[k] - v[k] + a[k-1]) div 2;
      end else
      begin
        u[k-1] := (u[k] - v[k]) div 2;   v[k-1] := (-u[k] - v[k]) div 2;
      end;
    end;
  end; {end of loop}
```

Purdy's algorithm has average running time approximately $K \cdot n$ for some positive constant K, see Theorem 2 in Ref. [4]. However, the worst case behavior is quadratic in n. To see this take as input to the algorithm $a = 2^{n-1} + 2^{n-2} + 1$, $b = 2^{n-1} - 2^{n-2} + 1$. Note that after one step $a = 2^{n-1} + 1$ and $b = 2^{n-2}$; after n more steps the pattern is repeated: $a = 2^{n-2} + 1$ and $b = 2^{n-3}$. Thus the forward sweep will need $n(n+1)/2 + 1$ transformations.

We have tested both algorithms for all pairs of odd integers in the range from 1 to 2^{15}. Our tests have shown that, on average, Purdy's algorithm requires twice as many transformations to compute the extended GCD. Also, the average number of additions in Purdy's algorithm is twice as large as for algorithm EPM.

The only advantage we have observed for Purdy's algorithm is that the computed coefficients u_0 and v_0 do not exceed, in absolute value, the greater of $|a_0|$ and $|b_0|$. In fact the following is true.

Lemma

If $\{(a_i, b_i)\}$ and $\{(u_i, v_i)\}$, $i = M, \ldots, 1$, are the sequences computed in the backward sweep of Purdy's algorithm then for $i = M, \ldots, 1$

$$\max(|u_i|, |v_i|) \leqslant \max(|a_i|, |b_i|)$$

Proof: The proof is by induction and can be found in Ref. [12]. ■

Remark

Our tests suggested that the ratio

$$\max(|u_0|, |v_0|)/\max(|a_0|, |b_0|)$$

does not exceed 2/3 except in trivial cases, but we were unable to prove this. ■

4. REDUCTION OF RATIONALS TO STANDARD FORM

When performing rational arithmetic it is usually desirable to reduce a result a_0/b_0 to a_0'/b_0', where $a_0' = a_0/\text{GCD}\ (a_0, b_0)$ and $b_0' = b_0/\text{GCD}\ (a_0, b_0)$. An algorithm similar to Algorithm EPM allows this reduction to be performed during the computation of GCD (a_0, b_0), without any division. We simply maintain

$$\mathbf{U}_k = \begin{bmatrix} \mu_k & \lambda_k \\ \gamma_k & \eta_k \end{bmatrix}, \text{ such that } \mathbf{U}_k \begin{bmatrix} a_k \\ b_k \end{bmatrix} = \begin{bmatrix} a_0 \\ b_0 \end{bmatrix},$$

by taking $\mathbf{U}_0 = I$ and $\mathbf{U}_k = \mathbf{U}_{k-1}\mathrm{T}_k^{-1}$ for $k \geqslant 1$. Then,

$$\begin{bmatrix} a_0 \\ b_0 \end{bmatrix} = a_m \begin{bmatrix} \mu_m \\ \gamma_m \end{bmatrix}, \quad |a_m| = \mathrm{GCD}(a_0, b_0)$$

so $a_0' = \pm\mu_m$ and $b_0' = \pm\gamma_m$, where m is the number of transformations required for convergence (i.e., $b_m = 0$).

5. CONCLUSIONS

We have shown that the Algorithm PM of Brent and Kung [5–7] can be extended to solve the problem (1). The resulting Algorithm EPM requires $O(n^2)$ bit operations and can be pipelined, so with $O(n)$ pipeline stages or $O(n)$ systolic cells it requires time $O(n)$. A similar algorithm may be used to reduce rational numbers to standard form without divisions.

The extended GCD algorithm of Purdy [4] appears to be inferior to Algorithm EPM in the average case, and it is certainly inferior in the worst case, as it may occasionally require of order n^2 stages, and hence of order n^3 bit operations.

Asymptotically faster parallel GCD algorithms are known Ref. [11], but Algorithms PM and EPM appear to be the fastest known algorithms which are practically useful and could readily be implemented in hardware.

REFERENCES

1. R. P. Brent and H. T. Kung, Systolic VLSI arrays for polynomial GCD computation. *IEEE Trans. Comput.* **C-33,** 731–736 (1984).
2. R. P. Brent, An improved Monte Carlo factorization algorithm. *BIT* **20,** 176–184 (1980).
3. R. P. Brent, Some integer factorization algorithms using elliptic curves. *Proc. 9th Australian Computer Science Conf.* (Ed. G. W. Gerrity), Canberra, pp. 149–163 (1986).
4. G. B. Purdy G. B., A carry-free algorithm for finding the greatest common divisor of two integers, *Comput. Math. Applic.* **9,** 311–316 (1983).
5. R. P. Brent and H. T. Kung, Asystolic VLSI arrays for linear-time GCD computation. In *VLSI '83* (Edited by F. Anceau and E. J. Aas). Elsevier/North Holland, New York. 145–154 (1983).
6. R. P. Brent and H. T. Kung, A systolic algorithm for integer GCD computation. In *ARITH-7, Proc. 7th Symp. on Computer Arithmetic*, Illinois (June 1985).
7. R. P. Brent, H. T. Kung and F. T. Luk, Some linear-time algorithms for systolic arrays. In *Information Processing '83* (Edited by R. Mason). North Holland, Amsterdam, pp. 865–876 (1983).
8. R. P. Brent, Analysis of the binary Euclidean algorithm. In *New Directions and Recent Results in Algorithms and Complexity* (Edited by J. F. Traub). Academic Press, New York, pp. 321–355. (1976).
9. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, 2nd edn. Addison-Wesley, Reading, Mass. (1981).
10. H. T. Kung, Why systolic architectures? *IEEE Comput.* **15** (1), 37–46 (1982).
11. A. Borodin, J. von zur Gathen and J. Hopcroft, Fast parallel matrix and GCD computations. In *Proc. 23rd A. Symp. on the Foundations of Computer Science*, IEEE Computer Society, pp. 65–77 (1982).
12. A. Bojanczyk and R. P. Brent, A systolic algorithm for extended GCD computation. Technical Report CMA-29-85, Centre for Mathematical Analysis, The Australian National Univ., Canberra (1985).