A Systolic Algorithm for Integer GCD Computation

R. P. Brent

Centre for Mathematical Analysis
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.

ABSTRACT

It is shown that the greatest common divisor of two n-bit integers (given in the usual binary representation) can be computed in time $O(n)$ on a linear systolic array of $O(n)$ identical cells.

1. INTRODUCTION

We consider the problem of computing the greatest common divisor GCD (a, b) of two positive integers a and b. This problem arises when performing exact rational arithmetic during symbolic and algebraic computations, in factorisation[1, 2], etc. For serial computation the classical Euclidean algorithm and its variants have been thoroughly analysed and their behaviour is well understood[3, 4, 5]. In this paper we consider parallel algorithms which could readily be implemented in hardware. We assume that a and b are represented in binary in the usual way.

The integer GCD problem is superficially similar to the problem of computing a GCD of two polynomials[6, 7, 8]. The significant difference between integer and polynomial GCD computations is that carries have to be propagated in the former, but not in the latter.

Since our aim is to obtain parallel algorithms which can readily be implemented in hardware, we do not permit the most general "unbounded parallelism" model[6]. Instead, we restrict our attention to algorithms which can be implemented on a linear array of $O(n)$ "systolic processors" or "cells" with nearest-neighbour communication [9, 10]. In Sections 4-6 we show that GCD (a, b) can be computed in time $O(n)$ on a linear array of $O(n)$ cells, where each cell is a finite-state machine which could be implemented using a special-purpose VLSI chip or a microprogrammable chip[11].

In a companion paper[8] the easier problem of polynomial GCD computation (over a finite filed) is considered. It is recommended that the reader unfamiliar with systolic algorithms should read that paper before reading Sections 4-6 of this paper.

In Section 2 we consider various serial algorithms for integer GCD computation. The classical Euclidean algorithm[4] and the binary Euclidean algorithm[2, 12, 13] do not appear to lead to good parallel algorithms, but a new algorithm (PM), which may be regarded as a variant of the binary Euclidean algorithm and is described in Section 3, does lead to a good parallel algorithm. In Section 4 we consider the systolic implementation of algorithm PM, and in Section 5 we give an upper bound on the number of systolic cells required. A lower bound and some empirical results are given Section 6. Finally, a possible VLSI implementation is discussed in Section 7.

*H. T. Kung was sponsored in part by the Office of Naval Research under Contract N00014-80-C-0236, NR 048-659.

2. THE CLASSICAL AND BINARY EUCLIDEAN ALGORITHMS

Assume that a and b are positive. The classical Euclidean algorithm[4] may be written as:

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

This is simple, but not attractive for a systolic implementation because the inner loop includes the division operation "$a \bmod b$" which takes time $\Omega(n)$. The "binary" Euclidean algorithm[4, 12, 13] avoids divisions, so is superficially more attractive. The binary Euclidean algorithm may be written as:

```
{assume a, b odd}
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 .
```

Figure 1. The binary Euclidean Algorithm B_1 (usual version).

An alternative which avoids the use of an auxiliary variable t and the absolute value function is given in Figure 2.

```
{assume a odd, b ≠ 0}
while b ≠ 0 do
    begin
    while even (b) do b := b div 2;
    if a ≥ b then a <-> b;
    b := b-a
    end;
GCD := a .
```

Figure 2. The binary Euclidean Algorithm B_2 (alternative version).

It is easy to verify that Algorithm B_2 with initial $(a, b) = (\min(\bar{a}, \bar{b}), |\bar{a} - \bar{b}|)$ generates the same sequence as Algorithm B_1 with initial $(a, b) = (\bar{a}, \bar{b})$. Conversely, Algorithm B_1 with initial $(a, b) = (\bar{a} + \bar{b}, \bar{a})$ gives the same sequence as Algorithms B_2 with initial $(a, b) = (\bar{a}, \bar{b})$. Hence, the two algorithms are essentially equivalent.

To prove convergence of algorithms B_1 and B_2, it is sufficient to note that $a + b$ decreases by a factor of $3/4$ at each iteration, because the larger of a and b is replaced by $|a - b|/2^k$ for some $k \geq 1$.

3. ALGORITHM PM

Note that the inner loops of algorithms B_1 and B_2 involve the comparison "if $a \geq b$", and in the worst case this takes time $\Omega(n)$ because the result depends on all bits in the binary representations of a and b.

We attempt to modify the binary Euclidean algorithm so that opera-

tions in the inner loop depend only on the low-order bits of a and b (and hence the inner loop can be pipelined on a systolic array). Our first attempt is algorithm B_3.

```
{assume a odd, |a| ≤ 2^n, |b| ≤ 2^n}
α := n;          {|a| ≤ 2^α}
β := n;          {|b| ≤ 2^β}
while b ≠ 0 do
    begin
    while even (b) do
        begin
        b := b div 2; β := β - 1
        end;
    {odd(b), |b| ≤ 2^β}
L0 : if α ≥ β then
        begin
        a <-> b;  α <-> β
        end;
    {odd(a), odd(b), |a| ≤ 2^α, |b| ≤ 2^β,   α ≤ β}
    L1 : if even ((a+b) div 2) then b := (a+b) div 2
                               else b := (a-b) div 2
    end;
GCD := |a| .
```

Figure 3. Algorithm B_3: a first attempt.

The idea of algorithm B_3 is to maintain integers α and β such that $|a| \le 2^\alpha, |b| \le 2^\beta$, and replace the test "$a \ge b$" in algorithm B_2 by the weaker but more easily implemented test "$\alpha \ge \beta$". (For details of how this test is implemented see Section 4 below; at present we merely note that α and β are number with $O(\log n)$ bits whereas a and b have $O(n)$ bits.)

It is tempting to replace statement L1 by the simpler

"L2: $b := b - a$".

However, the algorithm would then fail to converge for certain initial data, e.g., $a = 1, b = 3$. The problem is that $\alpha \le \beta$ does not imply $a \le b$, so the variables b and a may become negative, then $|a| \le 2^\alpha, |b| \le 2^\beta$ and $\alpha \le \beta$ before L2 does not imply $|b| \le 2^\beta$ after L2. Changing the test "$\alpha \ge \beta$" to "$\alpha > \beta$" at statement L0 does not help (try $a = 3, b = 1$). We do have convergence of the algorithm given in Figure 3 because

$$|b| \le \frac{2^\alpha + 2^\beta}{2} \le 2^\beta$$

after statement L1, and since b is even after L1, $\alpha + \beta$ decreases by each time (except possibly the first) that the inner loop is executed. Hence the inner loop can be executed at most $2n + 1$ times.

Another alternative is to replace statement L1 by

"L3: $b := |b - a|$".

With this modification the algorithm converges, but the inner loop is difficult to pipeline because of the absolute value function occurring at statement L3. As given in Figure 3 the inner loop is easy to pipeline because the operations on α and β depend only on the two least significant bits of their 2's complement binary representations.

Observe that the two variables α and β in algorithm B_3 are unnecessary: only their difference $\delta = \alpha - \beta$ is required. Using this fact and allowing for the possibility of a being even, we get the algorithm PM (for "plus-minus"). The comments involving α and β are included only to show the relationship to Algorithm B_3.

```
    {assume a, b not both zero}
    {α := n; β := n}
     δ:= 0; {δ = α - β, |a| ≤ 2^α, |b| ≤ 2^β}
    m := 1;
L0: while even (a) and even (b) do
        begin
        a := a div 2; b := b div 2; {α := α -1,  β := β -1}
        m := 2*m
        end;
    if even (a) then a <-> b;
    {now a odd, |a| ≤ 2^α, |b| ≤ 2^β}
L1: while b ≠ 0 do
        begin
L2:     while even (b) do
            begin
            b := b div 2; δ := δ +1 {β := β -1}
            end;
        if  δ ≥ 0 then
            begin
            a <-> b;  δ := -δ  {α <-> β}
            end;
L3:     if even ((b+a) div 2) then b := (b+a) div 2
                  else b := (b-a) div 2
        end;
    G := m*a.  {G = ± GCD}
```

Figure 4. Algorithm PM.

4. SYSTOLIC IMPLEMENTATION OF ALGORITHM PM

Consider now the implementation of algorithm PM on a systolic array. We assume that a and b are represented as (2's complement) binary integers which enter the array least-significant bits first. The initial "while" loop L0 and the final multiplication by m are easily implemented: essentially the array just transmits unchanged the low-order zero bits of a and b.

There is no need for the array to check the termination criterion: once b becomes zero the systolic cells to the right will merely implement the "while" loop L2 ($b := b$ div 2; $\delta := \delta + 1$) and transmit a to the right.

In the inner loop L1 the essential tests involving a and b depend only on the low-order 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.

It only remains to consider the implementation of the operations on δ in algorithm PM. The only operations required are "$\delta := \delta + 1$", "$\delta := -\delta$", and "if $\delta \ge 0 \cdots$ then". Rather than represent δ in binary, we choose a "sign and magnitude unary" representation, i.e., keep sign (δ) and $|\delta|$ separate, and represent $\varepsilon = |\delta|$ in unary as the distance between 1-bits in two streams of bits. With this representation all required opeartions on δ can be pipelined. For example, the operation "$\delta := \delta + 1$" merely involves shifting one bit stream relative to the other and possibly complementing the sign bit.

These considerations lead, after some straightforward optimizations, to the systolic cell illustrated in Figure 5. The cell has six input streams (each one bit wide), and six output streams which are connected to the corresponding input streams of the cell to the right. The input streams are `ain` and `bin` for the bits in the 2's complement binary representation of the numbers a and b (least significant bit first), `startin` to indicate the least significant bit of a, and three additional streams

`startoddin`, `epsin` and `negin` which should be all zero on input to the leftmost cell. `startoddin` is used to indicate the least significant 1-bit of a and b (so the distance between 1 bits in the `startin` and `startoddin` streams is $\log_2 m$, where m is the highest power of 2 in GCD (a,b), as in Figure 4). `epsin` and `negin` are used to represent $\varepsilon = |\delta|$ and sign $(-\delta)$ respectively (ε is represented as the distance between 1-bits in the `epsin` and `startoddin` streams).

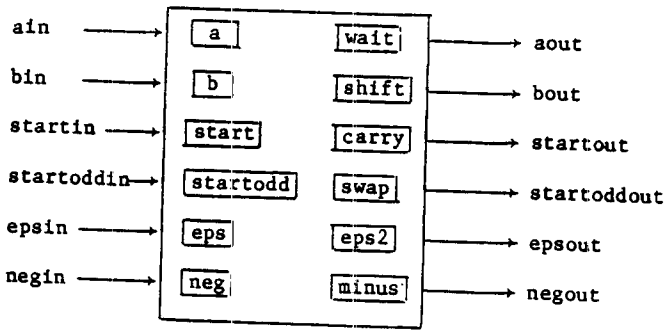

Figure 5. Systolic cell for integer GCD computation.

The cell has twelve internal state bits: one for each of the six inputs, and six additional bits (`wait`, `shift`, `carry`, `swap`, `eps2` and `minus`). The `wait` bit is on if the cell is waiting for the first nonzero bit in the binary representations of a and b. The `shift` bit signifies that the cell is shifting the bits of b right faster than those of a (i.e. implementing "$b := b$ div 2". The `carry` bit has the obvious meaning for a cell performing the serial binary operation $a + b$ or $a - b$. The `swap` bit signifies that a and b are to be interchanged (and the sign of δ reversed); to save cells the interchange may be combined with a shift. `eps2` is used to save a bit on the `epsin` stream (so the operations $\varepsilon := \varepsilon - 1$ and $\varepsilon := \varepsilon + 1$ can be implemented by shifting the `epsin` stream either faster or slower than the "normal" speed of one cell per two cycles). Finally, `minus` indicates whether subtraction or addition is to be performed at statement L3. A definition of the systolic cell is given in Appendix A.

5. UPPER BOUNDS ON THE NUMBER OF SYSTOLIC CELLS

In this section we show that the systolic array described in Section 4 will correctly compute $\pm$GCD (a, b) for n-bit integers a and b, provided that the number of cells in the array is at least $\lceil 3.1105n \rceil$. First we prove a weaker result.

Theorem 1: $4n$ cells as defined in Appendix A are sufficient to compute $\pm$GCD (a, b) for any n-bit integers a and b.

Proof: Consider the systolic implementation of Algorithm PM. The statements

"$a := a$ div 2; $b := b$ div 2; $\{\alpha := \alpha - 1; \beta := \beta - 1\}$"

are implemented by one systolic cell. Similarly for the statements

"$b := b$ div 2; $\delta := \delta + 1$; $\{\beta := \beta - 1\}$"

and the statement

"L3: if even $((a + b)$ div $2)$
then $b := (a + b)$ div 2
else $b := (a - b)$ div 2."

Suppose these statements are executed p_1, p_2 and p_3 times (respectively) before b becomes zero. This requires $p = p_1 + p_2 + p_3$ cells. Other statements such as

```
"if  δ ≥ 0 then
     begin
     a <-> b;
     δ := -δ   {α <-> β}
     end"
```

may be disregarded because they are implemented by cells which have been counted already.

At statement L1, if $b \neq 0$ we have $\delta = \alpha - \beta \leq 0, 0 \neq |a| \leq 2^\alpha$, $|b| \leq 2^\beta$. Hence $\beta \geq \alpha \geq 0$, so $\alpha + \beta \geq 0$.

Now $2n - (\alpha + \beta) = 2p_1 + p_2$ and $p_2 \geq p_3 - 1$ (since b is even after execution of statement L3). Thus

$$p = p_1 + p_2 + p_3 \leq p_1 + 2p_2 + 1 \leq 2(2p_1 + p_2) + 1$$
$$\leq 4n - 2(\alpha + \beta) + 1 \leq 4n + 1,$$

i.e., $4n + 1$ cells suffice. However, the last cell implementing statement L3 merely sets b to zero (at this point $b = \pm a$) and does not change a, so $4n$ cells would suffice to compute $\pm$GCD. This completes the proof of Theorem 1.

Before proving a stronger result, we need some Lemmas.

Lemma 1: Let $\mu = \dfrac{1 + \sqrt{17}}{8} \approx 0.6404$ and $1 \leq m \leq k$. Then

(i) $1 + \mu = 4\mu^2$

(ii) $\mu > 2^{-2/3}$

(iii) $(1 + \mu^{-1})/2^{k+1} \leq \mu^{(2k+m-1)/2}$

(iv) $2^{-(k+m)} + (1 + \mu^{-1})(1 - 4^{-m})/(3 \cdot 2^{k+1}) \leq \mu^{(2k+m+1)/2}$

(v) $2^{-k} \leq \mu^{(2k+m)/2}$

(vi) $2^{-(k+m)} + \mu^{-1}(1 - 4^{-m})/(3 \cdot 2^k) \leq \mu^{(2k+m-1)/2}$

(vii) $\mu^{-1}/2^{k+m} + (1 - 4^{-m})/(3 \cdot 2^k) \leq \mu^{(2k+m-2)/2}$

Proof: (i) is easy. For (ii), we have

$$\mu^3 = \frac{13 + 5\sqrt{17}}{128} > \frac{13 + 5.4}{128} > \tfrac{1}{4},$$

so

$$\mu > 2^{-2/3}.$$

Note that $1/2 < \mu$, so $2^{m-k} \leq \mu^{k-m}$. Thus, it is sufficient to prove (iii) - (vi) for $k = m$; the inequalities for $k > m$ then follow by multiplying the left sides by 2^{m-k} and the right sides by μ^{k-m}. With $k = m$, (iii) - (vii) reduce to (iii)' - (vii)'.

(iii)' $(1 + \mu^{-1})/2^{m+1} \leq \mu^{(3m-1)/2}$

(iv)' $2^{-2m} + (1 + \mu^{-1})(1 - 4^{-m})/(3 \cdot 2^{m+1}) \leq \mu^{\frac{3m+1}{2}}$

(v)' $2^{-m} \leq \mu^{3m/2}$

(vi)' $2^{-2m} + \mu^{-1}(1 - 4^{-m})/(3 \cdot 2^m) \leq \mu^{\frac{3m-1}{2}}$

(vii)' $2^{-2m} + \mu(1 - 4^{-m})/(3 \cdot 2^m) \leq \mu^{3m/2}$

We observe that (v)' follows from (ii), (vi)' follows from (iii)' (since $2^{-2m} \le 2^{-(m+1)}$ and $(1-4^{-m})/(3\cdot 2^m) < 2^{-(m+1)}$), and (vii)' follows from (iv)' (since $\mu \le (1+\mu^{-1})/2$) and $\mu^{(3m+1)/2} \le \mu^{3m/2}$). Hence it is sufficient to prove (iii)' and (iv)'. When $m = 1$, (iii)' and (iv)' reduce to equalities

(iii)" $(1 + \mu^{-1}) = \mu$, and

(iv)" $1/4 + (1 + \mu^{-1})/16 = \mu^2$.

For $m > 1$, we multiply the left side of (iii)" by 2^{1-m} and the right side by $\mu^{3(m-1)/2}$, giving (iii)' ($1/2 < \mu^{3/2}$ by (ii)). Similarly, from (iv)" we get

$$2^{-(m+1)} + (1 + \mu^{-1})/2^{m+3} \le \mu^{(3m+1)/2}$$

but it is easy to show (using $1 - 4^{1-m} \le 2(1 - 2^{1-m})$ and $\mu^{-1} \le 2$) that

$$2^{-2m} + (1 + \mu^{-1})(1 - 4^{-m})/3\cdot 2^{m+1} \le 2^{-(m+1)} + (1 + \mu^{-1})/2^{m+3},$$

so (iv)' follows. This completes the proof of Lemma 1.

Lemma 2: Consider Algorithm P defined in Figure 6. Provided the initial condition

$$\text{I: } \{\delta = 0, |a| \le k/\mu, |b| \le K, p = 0\}$$

holds, where $\mu = (1 + \sqrt{17})/8$ as in Lemma 1, then the condition

$$\text{H: } \{\delta = 0 => |a| \le K\mu^{p/2-1}, |b| \le K\mu^{p/2}\}$$

holds at all points indicated by {H} in Figure 6.

```
        {I}
        repeat
L1:     {H}
        if  δ ≥ 0 then
L2:         begin {J1}
            if odd (a) then
                begin
                if even ((a+b) div 2) then a := (a+b) div 2
                                      else a := (a-b) div 2;
                p := p+1
                end;
            while a ≠ 0 and even (a) do
L3:             begin
                a := a div 2;
                p := p+1;
                δ := δ-1
L4:             {H}
                end
            end
        else { δ < 0}
L5:         begin {J2}
            if odd (b) then
                begin
                if even ((a+b)div 2) then b := (b+a)div 2
                                     else b := (b-a)div 2;
                p := p+1
                end;
            while b ≠ 0 and even (b) do
L6:             begin
                b := b div 2;
                p := p+1;
                δ := δ+1
L7:             {H}
                end
            end
        until (a=0) or (b=0).
```

Figure 6. Algorithm P

Proof: Note that p increases monotonically during execution of algorithm P. The proof of Lemma 2 is by induction on $p \ge 0$. If $p = 0$ we have H true at label L1, by the initial condition I.

Note that δ decreases monotonically during execution of the block L2, then increases monotonically during execution of the block L5, then decreases monotonically during the execution of the block L2, etc. Also, H is trivially true if $\delta \ne 0$.

Consider an execution of block L5 such that $\delta = 0$ at statement L7. Let the values of (a, b, p) at this point be (a_2, b_2, p_2).

We shall show that H holds, i.e., that

$$|a_2| \le K\mu^{p_2/2-1} \text{ and } |b_0| \le K\mu^{p_2/2}.$$

First, suppose the previous occurrence of $\delta = 0$ occurred in block L2, say when the value of (a, b, p) at L4 was (a_0, b_0, p_0), where $p_0 < p_2$. By the inductive hypotheses,

$$|a_0| \le K\mu^{p_0/2-1} \text{ and } |b_0| \le K\mu^{p_0/2}.$$

There are two cases:

(i) a_0 odd, and

(ii) a_0 even.

Consider case (i) first. After reaching L4 with $p = p_0$, block L2 is executed once more. Thus, at L5 we have, for some $k \ge 1$, $(\delta, a, b, p) = (\delta_1, a_1, b_1, p_1)$, where $\delta_1 = -k$, $p_1 = p_0 + k + 1$, $a_1 = (a_0 \pm b_0)/2^{k+1}$, and $b_1 = b_0$. Then block L5 is executed some number $m \ge 1$ times until L7 is reached with $\delta = 0$. Thus, there is a sequence of positive integers $k_1 \ldots k_n$ such that

$$\sum_{i=1}^{m} k_i = k,\ p_2 = p_1 + k + m,\ a_2 = a_1,$$

and

$$b_2 = b'_m,$$

where

$$b'_0 = b_1 \text{ and } b'_i = (b'_{i-1} \pm a_i)/2^{k_i+1}$$

for $i = 1, \ldots, m$.

It follows that

$$|a_2| \le |a_0|/2^{k+1} + |b_0|/2^{k+1}$$
$$\le K\mu^{p_0/2}(\mu^{-1}+1)/2^{k+1} \quad \text{by Ind. Hypothesis}$$
$$\le K\mu^{(p_0+2k+m-1)/2} \quad \text{by Lemma 1 (iii)}$$
$$= K\mu^{p_2/2-1}$$

$$|b_2| \le |b_1|/2^{\sum_1^m (k_i+1)}$$
$$+ |a_1|(1/2^{k_m+1} + 1/2^{k_m+1+k_{m-1}+1} + \ldots + 1/2^{\sum_1^m (k_i+1)})$$
$$\le |b_0|/2^{k+m}$$
$$+ |a_1|(1/4+1/4^2+\ldots+1/4^m)$$
$$\le |b_0|/2^{k+m} + |a_1|(1-4^{-m})/3$$
$$\le K\mu^{p_0/2}[1/2^{k+m} + (\mu^{-1}+1)(1-4^{-m})/3.2^{k+1}]$$
$$\text{by Ind. Hypothesis}$$
$$\le K\mu^{(p_0+2k+m+1)/2} \quad \text{by Lemma 1 (iv)}$$
$$\le K\mu^{p_2/2}.$$

Thus, the proof for case (i) is complete.

Now consider case (ii) (a_0 even). After reaching L4 with $p = p_0$ and $\delta = 0$, block L3 is executed some positive number k times more then $\delta = -k < 0$ and block L5 is executed. Thus we have a_1, b_1, a_2, b_2, etc. as above except that

$$a_1 = a_0/2^k, \; p_1 = p_0 + k.$$

Hence $|a_2| \le K\mu^{p_2/2-1}$ and $|b_2| \le K\mu^{p_2/2}$ follows from the inductive hypothesis (for $p = p_0$) and Lemma 1 ((v) and (vi)).

Now, suppose that the previous occurrence of $\delta = 0$ occurred in block L5 (with $(a, b, p) = (a_0, b_0, p_0)$ say) rather than in block L2. This can only be the case if b_0 is odd. Then, block L2 is executed once, after which $(\delta, a, b, p) = (-k, a_1, b_1, p_1)$ say, and for some $k_1, \ldots, k_m$ we have

$$a_1 = (a_0 \pm b_0)/2^{k+1}, \; b_1 = b_0,$$

$$p_1 = p_0 + k + 1,$$

$$a_2 = b_2,$$

$$|b_2| \le |b_1|/2^{k+m} + |a_1|(1-4^{-m})/3, \text{ and}$$

$$p_2 = p_1 + k + m.$$

The result follows as in case (i) above.

Finally, we should consider an occurrence of $\delta = 0$ at statement L4 rather than at L7, with the previous occurrence of $\delta = 0$ at L7, and show that H holds. The proof of this is similar to case (ii) above, using Lemma 1 (vii), and hence is omitted.

This completes the proof of Lemma 2, by induction on p.

Lemma 3: Consider Algorithm P defined in Figure 6. Provided the initial condition

$$\text{I: } \{\delta = 0, |a| \le K/\mu, |b| \le K, p = 0\}$$

holds, where $\mu = (1 + \sqrt{17})/8$, then the conditions

$$\text{J}_1\text{: } |b| \le K\mu^{p/2}$$

and

$$\text{J}_2\text{: } |a| \le K\mu^{p/2-1}$$

hold whenever execution reaches the points indicated by $\{J_1\}$ and $\{J_2\}$, respectively.

Proof: Consider J_1 (J_2 is similar). Clearly $\delta \ge 0$. If $p = 0$ the result follows from I, so suppose $p > 0$. Then L4 and L7 must have been executed with $\delta = 0$. At the most recent such execution suppose the value of (a, b, p) was (a', b', p'). By Lemma 2, $|a'| \le K\mu^{p'/2-1}$ and $|b'| \le K\mu^{p'/2}$.

If $\delta = 0$ (so $p = p'$) we are finished. If $\delta > 0$ then block L6 must have been executed δ times since $p = p'$, so

$$b = b'/2^\delta, \; p = p' + \delta.$$

Thus $|b| \le K\mu^{(p-\delta)/2}.2^{-\delta} \le K\mu^{p/2}$ as $\mu^{-1/2} < 2$.

We are now ready to prove the main result of this section:

Theorem 2: $\lceil cn \rceil$ cells as defined in Appendix A are sufficient to compute $\pm$GCD(a, b) for any n-bit integers a and b, where

$$c = 2/\log_2(\frac{\sqrt{17}-1}{2}) \approx 3.1105.$$

Proof: The result is trivial if a or b is zero, so suppose both a and b are nonzero.

Let 2^k ($k \ge 0$) be the largest power of two which divides both a and b. Then the loop L0 of Algorithm PM reduces (a, b) to $(a/2^k, b/2^k)$ and its systolic implementation uses k cells. At this point $\delta = 0$, $|a| \le 2^{n-k}$, $|b| \le 2^{n-k}$, and at least one of a and b is odd. Subject to this condition, it is easy to see that the "while" loop L1 of Algorithm PM is equivalent to Algorithm P (Figure 6) modulo interchanges of a and b and corresponding sign reversals of δ. Furthermore, the variable p of Algorithm P counts the number of cells required in the systolic implementation (Appendix A) of loop L1 of Algorithm PM.

Algorithm P terminates when $a = 0$ or $b = 0$; immediately before this occurs we have $|a| = |b| = |G|$ at $\{J_1\}$ or $\{J_2\}$, where G is the GCD being computed. By Lemma 3,

$$1 \le |G| \le 2^{n-k}\mu^{p/2-1}.$$

Thus $p/2 - 1 \le (n-k)/\log_2(1/\mu)$. The total number of cells required is

$$p + k \le 2(n-k)/\log_2(1/\mu) + k + 1$$

and the right hand side attains its maximum (over $k \ge 0$) of $2n/\log_2(1/\mu) + 1$ when $k = 0$. Thus, since $\log_2(1/\mu)$ is not rational, the result follows.

6. A LOWER BOUND ON THE NUMBER OF SYSTOLIC CELLS

The following theorem shows that the constant c in Theorem 2 can not be reduced below 3.

Theorem 3: $3n - 5 + (n \bmod 2)$ of the cells defined in Appendix A are necessary to compute $\pm\mathrm{GCD}(a, b)$ for certain n-bit positive integers a, b.

Proof: The result is trivial if $n \le 2$, so suppose $n \ge 3$. If n is even, take $a = 3\cdot 2^{n-2} + 1$, $b = 3\cdot 2^{n-2} - 1$. It is easily verified that $3n-5$ cells are necessary and sufficient to compute $\pm\mathrm{GCD}(a, b)$. Similarly, if n is odd, take $a = 3\cdot 2^{n-2} - 1$, $b = 3\cdot 2^{n-2} + 1$. It is easily verified that $3n-4$ cells are necessary and sufficient in this case.

The best possible value of the constant c in Theorems 2 and 3 is unknown, but we conjecture that it is 3. The following table gives, for $2 \le n \le 18$, the number of cells (as in Appendix A) required to compute $\pm\mathrm{GCD}(a, b)$ for all positive n-bit integers (a, b), and an example (with minimal $\max(a, b)$) for which the worst case applies. For comparison the bounds of Theorems 2 and 3 are also given in the table.

7. A POSSIBLE VLSI IMPLEMENTATION

The GCD cell defined in Appendix A is a finite-state machine whose state is determined by the 24 Boolean variables $a, b, \ldots$. Consequently, it has 2^{24} states. In this section we outline one possible way in which the GCD cell could be implemented on a single chip using current nMOS technology. We do not claim that this is the best way to implement the GCD cell, but it does have the virtue of simplicity.

A set of Boolean functions can be implemented by a programmed logic array (PLA). The area of the PLA is approximately

$$A = 64(p+7)(2n_I + n_O + 6)\lambda^2,$$

where

n_I = number of input variables (or their complements),
n_O = number of outputs,
p = number of distinct product terms (i.e. conjuncts)

when the functions are written in disjunctive normal form (DNF), and λ is as in Mead and Conway[14].

We assume that Boolean variables $a, b, \ldots$ are implemented by clocked registers in a standard manner[14]. It would in principle be possible to implement the GCD cell as a single large PLA with $n_I = n_O = 18, p \approx 85$. However it is possible to split the GCD cell into several components which can each be implemented by a PLA of moderate size. The initial block of assignment statements "`aout := a; ...; negout := neg`" is easy to implement, as is the statement "`wait := (wait or start) and not startodd`".

The remainder of the cell definition has the form

n (number of bits)	lower bound (Theorem 3)	cells required in worst case	upper bound (Theorem 2)	example of worst case a	example of worst case b
2	1	3	7	1	3
3	5	6	10	7	5
4	7	10	13	15	13
5	11	11	16	17	23
6	13	15	19	57	47
7	17	18	22	33	125
8	19	20	25	119	213
9	23	23	28	319	349
10	25	26	32	647	693
11	29	29	35	1535	1537
12	31	33	38	3847	3829
13	35	35	41	6143	6145
14	37	38	44	10257	13651
15	41	41	47	24575	24577
16	43	45	50	64229	61519
17	47	47	53	98303	98305
18	49	50	56	185487	210061

Table 1: Number of cells required for n-bit inputs.

"if E_1 then B_1

else if E_2 then B_2

...

else B_5"

where $E_1, \ldots, E_4$ are Boolean expressions in the 6 variables `startodd`, `wait`, `a`, `b`, `shift`, `startoddout`, and $B_1, \ldots, B_5$ are blocks of assignment statements. We could implement each of $B_1, \ldots, B_5$ by "slave" PLAs $P_1, \ldots, P_5$ of moderate size. These PLAs compute in parallel and their outputs are selected by a "master" PLA which computes the 5 (mutually exclusive) functions E_1, $\bar{E}_1 \cdot E_2$, $\bar{E}_1 \cdot \bar{E}_2 \cdot E_3$, $\bar{E}_1 \cdot \bar{E}_2 \cdot \bar{E}_3 \cdot E_4$, $\bar{E}_1 \cdot \bar{E}_2 \cdot \bar{E}_3 \cdot \bar{E}_4$. The number of inputs ($n_I$), outputs ($n_O$) and product terms ($p$) for the PLAs $M, P_1, \ldots, P_5$ are given in Table 2.

PLA	n_I	n_O	p
M	6	5	7
P_1	5	7	6
P_2	1	1	1
P_3	9	5	8
P_4	5	3	9
P_5	9	4	12

Table 2: Parameters of PLAs for GCD cell implementation.

The total area of the PLAs in Table 2 is about 20000 λ^2. By way of comparison, the "brute-force" approach (using a single PLA) has area about 350000 λ^2. These estimates neglect I/O pads, routing between the PLAs, etc. With current technology the area of a chip can easily be 10^7 λ^2, so it should be possible to implement many GCD cells on a single chip.

A prototype systolic integer GCD cell was implemented on a multiproject chip (coordinated by the CSIRO VLSI Program, Adelaide, Australia) in November 1983, using nMOS technology with Mead and Conway design rules and λ = 2.5 micron. For the sake of variety and to reduce power consumption, our implementation used "blue-green function blocks" (Plate 7(b) of Mead and Conway[14]) instead of PLAs. To minimize routing problems, we used only two function blocks instead of the six suggested above. One function block, the "control" function block, computes the functions E_1, $\bar{E}_1 \cdot E_2, \ldots, \bar{E}_1 \cdot \bar{E}_2 \cdot \bar{E}_3 \cdot E_4$ and some other Boolean functions of its inputs, e.g. $a \oplus b$. The other function block, the "main" function block, computes all other required Boolean functions. To implement the finite state machine, 18 outputs of the main function block are fed back (through clocked registers) to the control functions block. To simplify testing we used static rather than dynamic registers. The total size of the cell is 688λ by 1022λ. About 40 percent of the area is occupied by the two function blocks, the remainder being used for I/O pads, registers, clock drivers, etc.

8. CONCLUDING REMARKS

We have shown that the greatest common divisor of two n-bit integers (given in the usual binary representation) can be computed in time $O(n)$ on a linear array of $O(n)$ identical systolic cells, each of which is a finite-state machine which could be implemented on (part of) a VLSI chip. Thus, special-purpose hardware for integer GCD computation could easily be built. Since GCD computation is the most time-consuming operation when rational arithmetic is performed, such hardware could be worthwhile for applications involving exact rational arithmetic, e.g., symbolic computation.

Recently Purdy[15] has suggested a different way to compute integer GCDs. Although Purdy's algorithm is linear on average, its worst-case behavious is quadratic.

For applications we usually want to compute the extended GCD. It is possible to do this using a straightforward extension of the systeolic cell defined in Appendix A. This extension will appear in a paper by Bojanczyk and Brent.

References

1. Brent, R.P., "An Improved Monte Carlo Factorization Algorithm," *BIT*, Vol. 20, 1980, pp. 176-184.

2. Brent, R.P. and Pollard, J.M., "Factorization of the Eighth Fermat Number," *Math. Comp.*, Vol. 36, 1981, pp. 627-630.

3. Aho, A., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1975.

4. Knuth, D.E., *Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts, The Art of Computer Programming, Vol. 2, 1981, Second Edition

5. Schonhage A., "Schnelle Berechnung von Kettenbruchentwicklungen," *Acta Informatica*, Vol. 1, 1971, pp. 139-144.

6. Borodin, A., von zur Gathen, J. and Hopcroft, J., "Fast Parallel Matrix and GCD Computations," *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, November 1982, pp. 65-71.

7. Brent, R.P., Gustavson, F.G. and Yun, D.Y.Y., "Fast Solution of Toeplitz Systems of Equations and Computation of Pade Approximants," *Journal of Algorithms*, Vol. 1, No. 3, September 1980, pp. 259-295.

8. Brent, R.P. and Kung, H.T., "Systolic VLSI Arrays for Polynomial GCD Computation," *IEEE Trans. Computers*, Vol. C-33, No. 8, August 1984, pp. 731-736.

9. Brent, R.P. ,Kung, H.T. and Luk, F.T., "Some Linear-Time Algorithms for Systolic Arrays," *Proceedings of the IFIP 9th World Computer Congress*, Mason, R., ed., IFIP, September 1983, pp. 865-876.

10. Kung, H.T., "Why Systolic Architectures?," *Computer Magazine*, Vol. 15, No. 1, January 1982, pp. 37-46.

11. Fisher, A.L., Kung, H.T., Monier, L.M. and Dohi, Y., "The Architecture of a Programmable Systolic Chip," *Journal of VLSI and Computer Systems*, Vol. 1, No. 2, 1984, pp. 153-169, An earlier version appears in *Conference Proceedings of the 10th Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pp. 48-53.

12. Brent, R.P., "Analysis of the Binary Euclidean Algorithm," *New Directions and Recent Results in Algorithms and Complexity*, Traub, J.F., ed., Academic Press, 1976, pp. 321-355.

13. Stein, J., "Computational Problems Associated with Racah Algebra," *J. Comput. Phys.*, Vol. 1, 1967, pp. 397-405.

14. Mead, C.A. and Conway, L.A., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

15. Purdy, G.B., "A Carry-free Algorithm for Finding the Greatest Common Divisor of Two Integers," *Computers and Maths. with Appls*, Vol. 9, 1983, pp. 311-316.

APPENDIX A: Cell definition for systolic integer GCD computation

```
{The language used is Pascal with some trivial extensions and declarations
omitted.  See Figure 5 for I/O ports}

aout := a;                    a := ain;     {standard transfers}
bout := b;                    b := bin;
startout := start;            start := startin;
startoddout := startodd;      startodd := startoddin;
epsout := eps2; eps2 := eps;  eps := epsin; {delay here}
negout := neg;

wait := (wait or start) and not startodd; {wait for nonzero bit}

if startodd or (wait and (a or b)) then
  begin
  eps := eps or wait;
  eps2 := 0; {0 ≡ false,  1 ≡ true}
  neg := negin and not wait;
  startodd := 1;
  wait := 0; {end of waiting for a nonzero bit}
  swap := not a;
  shift := not (a and b)
  end
else if wait then epsout := eps2 {normal speed)
else if shift then {shift b faster than a, may also swap}
  begin
  aout := (bout and swap)  or  (aout and not swap); {normal speed}
  bout := (a and swap)  or  (b and not swap);       {fast speed}
  epsout := (eps and neg)  or  (epsout and not neg);
  neg := neg and not (eps and startoddout); {δ may become zero}
  negout := neg
  end
else if startoddout then
  begin
  epsout := eps2;              {normal speed}
  swap := not neg;
  neg := neg or not eps2;      {δ := -|δ|}
  negout := neg;
  aout := aout or swap;        {swap implies b}
  bout := 0;                   {and new  b  is even}
  carry := a ⊕ b;          {may be borrow or carry; ⊕ is exclusive or}
  minus := not carry       {1 iff we form (b - a) div 2}
  end
else {not startoddout}
  begin
  epsout := eps2;               {normal speed}
  aout := (bout and swap) or (aout and not swap); {normal speed}
  bout := a ⊕ b ⊕ carry;        {fast speed}
  carry := majority (b, carry, a ⊕ minus) {majority true if 2 or 3 of
                                           its arguments are true}
  end.
```