A SYSTOLIC ALGORITHM FOR INTEGER GCD COMPUTATION

Revision (U) Carnegie-Mellon Univ Pittsburgh PA Dept of
Computer Science R P Brent et al. Apr 84 CMU-CS-84-135

UNCLASSIFIED N00014-80-C-0236

END

Date Printed 9 84

Systolic

GCD
A systolic algorithm for integer GCD computation

R.P. Brent
Department of Computer Science and
Centre for Mathematical Analysis
Australian National University
GPO Box 4, Canberra, ACT 2601
Australia

H.T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
USA

A systolic algorithm for integer GCD computation

R.P. Brent
Department of Computer Science and
Centre for Mathematical Analysis
Australian National University
GPO Box 4, Canberra, ACT 2601
Australia

H.T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
USA

December 1982 (revised April 1984)
Abstract

We show 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 with connections to its nearest neighbours.

Key Words and Phrases

Systolic algorithm, systolic array, VLSI, greatest common divisor, special-purpose hardware, rational arithmetic, Euclidean algorithm, linear-time computation, symbolic and algebraic computation.
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 \([4, 8]\), etc. For serial computation the classical Euclidean algorithm and its variants have been thoroughly analysed and their behaviour is well understood \([1, 11, 14]\). 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 \([2, 5, 6, 17]\). 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 with VLSI technology \([10, 13]\), we do not permit the most general "unbounded parallelism" model of \([2]\). 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. Such "systolic algorithms" have been surveyed in \([7, 12]\). 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 such as the "PSC" chip designed at Carnegie-Mellon University \([9]\).
This paper is a companion paper to [6], where the easier problem of polynomial GCD computation (over a finite field) is considered. It is recommended that the reader unfamiliar with systolic algorithms should read [6] or Sections 1-2 of [7] before reading Sections 4-6 of this paper.

In Section 2 we consider various serial algorithms for integer GCD computation. The classical Euclidean algorithm [11] and the binary Euclidean algorithm [3, 8, 15] 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 in Section 6. Finally, a possible VLSI implementation is discussed in Section 7.

2. The classical and binary Euclidean algorithms

Assume that \( a \) and \( b \) are positive. The classical Euclidean algorithm [11] may be written as:

\[
\text{while } b \neq 0 \text{ do } \begin{pmatrix} a \\ b \end{pmatrix} := \begin{pmatrix} b \\ a \mod b \end{pmatrix} \; ; \; \text{GCD} := a .
\]

This is simple, but not attractive for a systolic implementation because the inner loop includes the division operation "\( a \mod b \)" which takes time \( \Omega(n) \). The "binary" Euclidean algorithm [3, 11, 15] avoids divisions, so is superficially more attractive. The binary Euclidean algorithm may be written as:
\{\text{assume } a, b \text{ odd}\}
\begin{align*}
t &:= |a-b|; \\
&\text{while } t \neq 0 \text{ do} \\
&\quad \begin{aligned}
&\text{repeat } t := t \div 2 \text{ until odd }(t); \\
&\quad \text{if } a \geq b \text{ then } a := t \text{ else } b := t; \\
&\quad t := |a-b| \\
&\end{aligned} \\
&\text{end}; \\
&\text{GCD} := a.
\end{align*}

\begin{figure}
\text{Figure 1: The binary Euclidean Algorithm } B_1 \text{ (usual version)}
\end{figure}

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

\begin{align*}
\{\text{assume } a \text{ odd, } b \neq 0\}
&\text{while } b \neq 0 \text{ do} \\
&\quad \begin{aligned}
&\quad \text{while even }(b) \text{ do } b := b \div 2; \\
&\quad \text{if } a \geq b \text{ then } a := b \text{ else } b := a \\
&\end{aligned} \\
&\text{end}; \\
&\text{GCD} := a.
\end{align*}

\begin{figure}
\text{Figure 2: The binary Euclidean Algorithm } B_2 \text{ (alternative version)}
\end{figure}

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

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 \( b \) .
To prove convergence of algorithms $B_1$ and $B_2$, it is sufficient to note that $a + b$ decreases by a factor $\frac{3}{4}$ 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

We attempt to modify the binary Euclidean algorithm so that operations 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, $|a| < 2^n$, $|b| < 2^n$}

$\alpha := n$; $\{|a| < 2^\alpha\}$

$\beta := n$; $\{|b| < 2^\beta\}$

while $b \neq 0$ do

begin

while even ($b$) do

begin

$b := b \text{ div } 2$; $\beta := \beta - 1$

end;

$\{\text{odd}(b), |b| < 2^\beta\}$

LO : if $\alpha \geq \beta$ then

begin

$a \leftrightarrow b$; $\alpha \leftrightarrow \beta$

end;

$\{\text{odd}(a), \text{odd}(b), |a| < 2^\alpha, |b| < 2^\beta, a \leq b\}$

L1 : if even ((a+b) $\text{ div } 2$) then $b := (a+b) \text{ div } 2$

else $b := (a-b) \text{ div } 2$

end;

GCD := $|a|$.

Figure 3 : Algorithm $B_3$: a first attempt
The idea of algorithm $B_3$ is to maintain integers $\alpha$ and $\beta$ such that $|\alpha| \leq 2^\alpha$, $|\beta| \leq 2^\beta$, and replace the test "$\alpha \geq \beta$" in algorithm $B_2$ by the weaker but more easily implemented test "$\alpha > \beta$". (For details of how this test is implemented see Section 4 below; at present we merely note that $\alpha$ and $\beta$ are number with $O(\log n)$ bits whereas $\alpha$ and $\beta$ 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 $\alpha \leq \beta$ does not imply $\alpha \leq b$, so the variables $b$ and $a$ may become negative, then $|\alpha| \leq 2^\alpha$, $|\beta| \leq 2^\beta$ and $\alpha \leq \beta$ before L2 does not imply $|\beta| \leq 2^\beta$ after L2. Changing the test "$\alpha \geq \beta$" 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

$$|\beta| \leq \frac{2^\alpha + 2^\beta}{2} \leq 2^\beta$$

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 \( a \) and \( b \) depend only on the two least significant bits of their 2's complement binary representations.

Observe that the two variables \( a \) and \( b \) in algorithm \( B_3 \) are unnecessary: only their difference \( \delta = a - b \) 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 \( a \) and \( b \) are included only to show the relationship to Algorithm \( B_3 \).

\[
\begin{align*}
\text{assume } a, b \text{ not both zero} \\
\{a := n; b := n\} \\
\delta := 0; \{\delta = a - b, |a| \leq 2^a, |b| \leq 2^b\} \\
m := 1; \\
\text{L0: while even (a) and even (b) do} \\
\quad \text{begin} \\
\quad \quad a := a \div 2; b := b \div 2; \{a := a - 1, b := b - 1\} \\
\quad \quad m := 2m \\
\quad \text{end;} \\
\quad \text{if even (a) then } a \leftrightarrow b; \\
\quad \text{(now a odd, } |a| \leq 2^a, |b| \leq 2^b) \\
\text{L1: while } b \neq 0 \text{ do} \\
\quad \text{begin} \\
\quad \quad \text{L2: while even (b) do} \\
\quad \quad \quad \text{begin} \\
\quad \quad \quad \quad b := b \div 2; \delta := \delta + 1 \{\beta := \beta - 1\} \\
\quad \quad \quad \text{end;} \\
\quad \quad \quad \text{if } \delta \geq 0 \text{ then} \\
\quad \quad \quad \quad \text{begin} \\
\quad \quad \quad \quad \quad a \leftrightarrow b; \delta := -\delta \{a \leftrightarrow b\} \\
\quad \quad \quad \quad \text{end;} \\
\quad \quad \quad \text{L3: if even } ((b+a) \div 2) \text{ then } b := (b+a) \div 2 \\
\quad \quad \quad \quad \text{else } b := (b-a) \div 2 \\
\quad \quad \text{end;} \\
\quad \text{G := m*a. } \{G = \pm \text{GCD}\}
\end{align*}
\]

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 \( \delta \) in algorithm PM. The only operations required are "\( \delta := \delta + 1 \)" , "\( \delta := -\delta \)" , and "if \( \delta \geq 0 \) ...". Rather than represent \( S \) in binary, we choose a "sign and magnitude unary" representation, i.e. keep sign \( (\delta) \) and \( |\delta| \) separate, and represent \( \epsilon = |\delta| \) in unary as the distance between 1-bits in two streams of bits. With this representation all required operations on \( \delta \) 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 \( \text{ain} \) and \( \text{bin} \) for the bits in the 2's complement binary representation of the numbers \( a \) and \( b \) (least significant bit first), \( \text{startin} \) to indicate the least significant bit of \( a \), and three additional streams \( \text{startoddin} \), \( \text{epsin} \) and \( \text{negin} \) which should be all zero on input to the leftmost cell. \( \text{startoddin} \) is used to indicate the least significant 1-bit of \( a \) and \( b \) (so the distance between 1 bits in the \( \text{startin} \) and \( \text{startoddin} \) streams is \( \log_2 m \), where \( m \) is the highest power of 2 in \( \text{GCD} (a,b) \), as in Figure 4). \( \text{epsin} \) and \( \text{negin} \) are used to represent

![Diagram of systolic cell](image.png)
\( c = |\delta| \) and sign \((-\delta)\) respectively \((\epsilon\) is represented as the distance between 1-bits in the \(\text{epsin}\) and \(\text{startoddin}\) streams).

The cell has twelve internal state bits: one for each of the six inputs, and six additional bits (\(\text{wait}\), \(\text{shift}\), \(\text{carry}\), \(\text{swap}\), \(\text{eps2}\) and \(\text{minus}\)). The \(\text{wait}\) bit is on if the cell is waiting for the first nonzero bit in the binary representations of \(a\) and \(b\). The \(\text{shift}\) 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 \(\text{carry}\) bit has the obvious meaning for a cell performing the serial binary operation \(a + b\) or \(a - b\). The \(\text{swap}\) bit signifies that \(a\) and \(b\) are to be interchanged (and the sign of \(\delta\) reversed); to save cells the interchange may be combined with a shift. \(\text{eps2}\) is used to save a bit on the \(\text{epsin}\) stream (so the operations \(\epsilon := \epsilon - 1\) and \(\epsilon := \epsilon + 1\) can be implemented by shifting the \(\text{epsin}\) stream either faster or slower than the "normal" speed of one cell per two cycles). Finally, \(\text{minus}\) 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 required

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

**Theorem 1**

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

**Proof**

Consider the systolic implementation of Algorithm PM. The statements

```
"a := \text{div } 2; \ b := b \text{ div } 2; \ {a := a-1; \ b := b-1}
```

are implemented by one systolic cell. Similarly for the statements

```
"b := b \text{ div } 2; \ \delta := \delta + 1 \ \{b := b-1}
```

and the statement

```
"L3: if even ((a+b) \text{ div } 2) then b := (a+b) \text{ div } 2
else b := (a-b) \text{ 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 \ \delta \geq 0 \ then
begin
a <-> b;
\delta := -\delta \ \{a <-> b\}
end"
```
may be disregarded because they are implemented by cells which have been counted already.

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

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

\[
\begin{align*}
    p &= p_1 + p_2 + p_3 \\
    &\leq p_1 + 2p_2 + 1 \\
    &\leq 2(2p_1 + p_2) + 1 \\
    &\leq 4n - 2(a+\beta) + 1 \\
    &\leq 4n + 1,
\end{align*}
\]

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

Before proving a stronger result, we need some Lemmas.

**Lemma 1**

Let \( \mu = \frac{1+\sqrt{17}}{8} = 0.6404 \) 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) \[ 2^{-(k+m)} + (1+\mu^{-1})(1-4^{-m})/(3.2^{k+1}) \leq \mu^{2^{k+m+1}} \]

(v) \[ 2^{-k} \leq \mu^{2^{k+m}} \]

(vi) \[ 2^{-(k+m)} + \mu^{-1}(1-4^{-m})/(3.2^k) \leq \mu^{2^{k+m-1}} \]

(vii) \[ \mu^{-1/2^{k+m}} + (1-4^{-m})/(3.2^k) \leq \mu^{2^{k+m-2}} \]

Proof

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

\[ \mu^3 = \frac{13+5\sqrt{17}}{128} > \frac{13+5.4}{128} > \frac{1}{4} \]

so \[ \mu > 2^{-2/3} \]

Note that \( k < \mu \), 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 \( \mu^{k-m} \). With \( k = m \), (iii) - (vii) reduce to (iii)' - (vii)'

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

(iv)' \[ 2^{-2m} + (1+\mu^{-1})(1-4^{-m})/(3.2^{m+1}) \leq \mu^{3m+1} \]

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

(vi)' \[ 2^{-2m} + \mu^{-1}(1-4^{-m})/(3.2^m) \leq \mu^{3m-1} \]

(vii)' \[ 2^{-2m} + \mu(1-4^{-m})/(3.2^m) \leq \mu^{3m/2} \]
We observe that (v)' follows from (ii), (vi)' follows from (iii)' (since
\[ 2^{-2m} \leq 2^{-(m+1)} \text{ and } (1-4^{-m})/(3.2^m) < 2^{-(m+1)} \text{, and (vii)'} \]
follows from (iv)' (since \( \mu \leq \frac{1+\mu^{-1}}{2} \) and \( \mu^2 \leq \mu^2 \)). Hence it is sufficient to prove (iii)' and (iv)'. When \( m = 1 \), (iii)' and (iv)' reduce to equalities

(iii)'' \[ (1+\mu^{-1}) = \mu \text{ and} \]
(iv)'' \[ 1/4 + (1+\mu^{-1})/16 = \mu^2 \]

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

\[ 2^{-(m+1)} + (1+\mu^{-1})/2^{m+3} \leq \mu^{(3m+1)/2} \]

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

\[ 2^{-2m} + (1+\mu^{-1})(1-4^{-m})/3.2^{m+1} \leq 2^{-(m+1)} + (1+\mu^{-1})/2^{m+3}, \]

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

**Lemma 2**

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

I: \{ \( \delta = 0 \), \( |a| \leq K/\mu \), \( |b| \leq K \), \( p = 0 \)\}

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

H: \{ \( \delta = 0 \Rightarrow |a| \leq K\mu^{p/2-1} \), \( |b| \leq K\mu^{p/2} \)\}

holds at all points indicated by \{H\} in Figure 6.
{L}
repeat
L1: {H}
  if \( \delta \geq 0 \) then
  L2: begin \{J_1\}
    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 \( a \neq 0 \) and even \( a \) do
      L3: begin
        \( a := a \div 2; \)
        \( p := p+1; \)
        \( \delta := \delta-1 \)
      end
    end
  end
else \( \delta < 0 \)
L5: begin \{J_2\}
  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 \( b \neq 0 \) and even \( b \) do
    L6: begin
      \( b := b \div 2; \)
      \( p := p+1; \)
      \( \delta := \delta+1 \)
    end
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 \( p \geq 0 \). If \( p = 0 \) we have \( H \) true at label \( L1 \), by the initial condition \( I \).

Note that \( \delta \) 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 \( \delta \neq 0 \).

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
\[
|a_2| \leq K \cdot \mu \cdot p_2^{2 - 1} \quad \text{and} \quad |b_2| \leq K \cdot \mu \cdot p_2^{2 - 2}.
\]
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,
\[
|a_0| \leq K \cdot \mu \cdot p_0^{2 - 1} \quad \text{and} \quad |b_0| \leq K \cdot \mu \cdot p_0^{2 - 2}.
\]

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 \( k \geq 1 \), \( (\delta', a, b, p) = (\delta_1, a_1, b_1, p_1) \), where \( \delta_1 = -k \), \( p_1 = p_0 + p_1 + 1 \), \( a_1 = (a_0 + b_0)/2^{k+1} \), and \( b_1 = b_0 \). Then block \( L5 \) is executed some number \( m \geq 1 \)
times until L7 is reached with δ = 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

$$b_2 = b^{'-1}.$$

where

$$b^{'-1} = b^{'-1} + a_i$$

and

$$b^{'-1} = (b^{'-1} + a_i) / 2$$

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

It follows that

$$|a_2| \leq |a_0|/2^{k+1} + |b_0|/2^{k+1}$$

$$\leq K \mu^{p_0/2} (u^{-1} + 1)/2^{k+1}$$

by Ind. Hypothesis

$$\leq K \mu^{p_0 + 2(k + m - 1)}/2$$

by Lemma 1 (iii)

$$= K \mu^{p_2/2 - 1}$$

$$\leq K \mu^{\sum_{i=1}^{m} (k_i + 1)}$$

and

$$|b_2| \leq |b_1|/2^{k+1}$$

$$+ |a_1|(1/2^m + 1/2^{m+1+k-1} + \ldots + 1/2^{k+1})$$

$$\leq |b_0|/2^{k+m}$$

$$+ |a_1|(1/4 + 1/4^2 + \ldots + 1/4^m)$$

$$\leq |b_0|/2^{k+m} + |a_1|(1 - 4^{-m})/3$$

$$\leq K \mu^{p_0/2} [1/2^{k+m} + (u^{-1} + 1)(1 - 4^{-m})/3.2^{k+1}]$$

by Ind. Hypothesis
\[(p_0 + 2k + m + 1)/2 \leq K \mu \]
\[p_2/2 \leq K \mu \]

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

Now consider case (ii) \((a_0 \text{ even})\). After reaching \(L_4\) with \(p = p_0\) and \(\delta = 0\), block \(L_3\) is executed some positive number \(k\) times more, then \(\delta = -k < 0\) and block \(L_5\) 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 \(|a_2| \leq K \mu p_2/2^{l-1}\) and \(|b_2| \leq K \mu p_2/2\)

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 \(L_5\) (with \((a, b, p) = (a_0, b_0, p_0)\) say) rather than in block \(L_2\). This can only be the case if \(b_0\) is odd. Then, block \(L_2\) 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

\[a_1 = (a_0 + b_0)/2^{k+1}, \quad b_1 = b_0,\]
\[p_1 = p_0 + k + 1,\]
\[a_2 = b_2,\]
\[|b_2| \leq |b_1|/2^{k+m} + |a_1|/(1-4^{-m})/3, \quad \text{and}\]
\[p_2 = p_1 + k + m.\]

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

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

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

$J_1 : |b| \leq K \mu p/2$

and

$J_2 : |a| \leq K \mu p/2-1$

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

**Proof**

Consider $J_1$ ($J_2$ is similar). Clearly $\delta \geq 0$. 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, $|a'| \leq K \mu p'/2-1$ and $|b'| \leq K \mu p'/2$. 
If \( \delta = 0 \) (so \( p = p' \)) we are finished. If \( \delta > 0 \) then block L6 must have been executed \( \delta \) times since \( p = p' \), so
\[
b = b'/2^\delta, \quad p = p' + \delta.
\]
Thus
\[
|b| \leq K \frac{u(p-\delta)}{2} 2^{-\delta}
\]
\[
\leq K \frac{u^{p/2}}{2} \quad \text{as} \quad u^{-1/2} < 2.
\]

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
\[
c = 2/\log_2 \left( \frac{\sqrt{17} - 1}{2} \right) = 3.1105.
\]

**Proof**

The result is trivial if \( a \) or \( b \) is zero, so suppose both \( a \) and \( b \) are nonzero.

Let \( 2^k \) (\( k \geq 0 \)) be the largest power of two which divides both \( a \) and \( b \). Then the loop LO of Algorithm PM reduces \( (a,b) \) to \( (a/2^k, b/2^k) \) and its systolic implementation uses \( k \) cells. At this point \( \delta = 0, |a| \leq 2^{n-k}, |b| \leq 2^{n-k}, \) and at least one of \( a \) and \( b \) is odd. Subject to this condition, it is easy to see that the "while" loop LI of Algorithm PM is equivalent to Algorithm P (Figure 6) modulo interchanges of \( a \) and \( b \) and corresponding sign reversals of \( \delta \).
Furthermore, the variable $p$ of Algorithm P counts the number of cells required in the systolic implementation (Appendix A) of loop $L_1$ 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,

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

Thus

$$p/2-1 \leq (n-k)/\log_2(1/\mu).$$

The total number of cells required is

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

and the right hand side attains its maximum (over $k \geq 0$) 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**

$$3n-5 + (n \mod 2)$$ of the cells defined in Appendix A are necessary to compute $\pm \gcd(a,b)$ for certain $n$-bit positive integers $a, b$. 
Proof

The result is trivial if \( n \leq 2 \), so suppose \( n \geq 3 \). If \( n \) is even, take \( a = 3.2^{n-2} + 1 \), \( b = 3.2^{n-2} - 1 \). It is easily verified that \( 3n-5 \) cells are necessary and sufficient to compute \( \gcd(a,b) \). Similarly, if \( n \) is odd, take \( a = 3.2^{n-2} - 1 \), \( b = 3.2^{n-2} + 1 \). 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 \( 2 \leq n \leq 18 \), the number of cells (as in Appendix A) required to compute \( \gcd(a,b) \) 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.
<table>
<thead>
<tr>
<th>n (number of bits)</th>
<th>lower bound (Theorem 3)</th>
<th>cells required in worst case</th>
<th>upper bound (Theorem 2)</th>
<th>example of worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>1 3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>7 5</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>15 13</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>11</td>
<td>16</td>
<td>17 23</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>15</td>
<td>19</td>
<td>57 47</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>18</td>
<td>22</td>
<td>33 125</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>20</td>
<td>25</td>
<td>119 213</td>
</tr>
<tr>
<td>9</td>
<td>23</td>
<td>23</td>
<td>28</td>
<td>319 349</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>26</td>
<td>32</td>
<td>647 693</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
<td>29</td>
<td>35</td>
<td>1535 1537</td>
</tr>
<tr>
<td>12</td>
<td>31</td>
<td>33</td>
<td>38</td>
<td>3847 3829</td>
</tr>
<tr>
<td>13</td>
<td>35</td>
<td>35</td>
<td>41</td>
<td>6143 6145</td>
</tr>
<tr>
<td>14</td>
<td>37</td>
<td>38</td>
<td>44</td>
<td>10257 13651</td>
</tr>
<tr>
<td>15</td>
<td>41</td>
<td>41</td>
<td>47</td>
<td>24575 24577</td>
</tr>
<tr>
<td>16</td>
<td>43</td>
<td>45</td>
<td>50</td>
<td>64229 61519</td>
</tr>
<tr>
<td>17</td>
<td>47</td>
<td>47</td>
<td>53</td>
<td>98303 98305</td>
</tr>
<tr>
<td>18</td>
<td>49</td>
<td>50</td>
<td>56</td>
<td>185487 210061</td>
</tr>
</tbody>
</table>

Table 1: Number of cells required for n-bit inputs
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 [13]. 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) [13]. From [10] 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 \( \lambda \) is as in Mead and Conway [13].

We assume that Boolean variables \( a, b, \ldots \) are implemented by clocked registers in a standard manner [13]. It would in principle be possible to implement the GCD cell as a single large PLA with \( n_I = n_O = 18 \), \( p = 85 \). 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

"if $E_1$ then $B_1$
else if $E_2$ then $B_2$
...
else $B_5$"

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

<table>
<thead>
<tr>
<th>PLA</th>
<th>$n_i$</th>
<th>$n_o$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$P_1$</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>9</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$P_5$</td>
<td>9</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

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

7. Conclusion

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.

8. Postscript (April 1984)

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 [13] and $\lambda = 2.5$ micron. For the sake of variety and to reduce power consumption, our implementation used "blue-green function blocks" (Plate 7(b) of [13]) instead of PLAs. To minimize routing problems, we used only two function blocks instead of the six suggested in Section 7. One function block, the "control" function block, computes the functions $E_1$, $\neg E_1, E_2$, $\neg E_1 \neg E_2, \neg E_3, \neg E_4$ (see Section 7) 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 function block. To simplify testing we used static rather than dynamic registers. The total size of the cell is $688\lambda$ by $1022\lambda$. 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. The layout is illustrated in Figure 7. Preliminary testing indicates that the prototype performs as expected.
Figure 7: Prototype systolic integer GCD cell
For applications we usually want to compute the extended GCD. It is possible to do this using a straightforward extension of the systolic cell defined in Appendix A: see Bojanczyk and Brent [16].

Recently Purdy [18] has suggested a different way to compute integer GCDs in linear time. However, although Purdy's algorithm is linear on average, its worst-case behaviour is quadratic. Consequently, our systolic algorithm, which is linear even in the worst case (by Theorem 1), seems preferable.
References


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)

\[
\begin{align*}
aout & := a; \quad a := \text{ain}; \quad \{\text{standard transfers}\} \\
bout & := b; \quad b := \text{bin}; \\
\text{startout} & := \text{start}; \quad \text{start} := \text{startin}; \\
\text{startoddout} & := \text{startodd}; \quad \text{startodd} := \text{startoddin}; \\
\text{epsout} & := \text{eps2}; \quad \text{eps2} := \text{eps}; \quad \text{eps} := \text{epsin}; \quad \{\text{delay here}\} \\
\text{negout} & := \text{neg}; \\
\text{wait} & := (\text{wait or start}) \text{ and not startodd}; \quad \{\text{wait for nonzero bit}\}
\end{align*}
\]

if startodd or (wait and (a or b)) then
begin
\[
\begin{align*}
\text{eps} & := \text{eps or wait}; \\
\text{eps2} & := 0; \quad \{0 \equiv \text{false}, \ 1 \equiv \text{true}\} \\
\text{neg} & := \text{negin and not wait}; \\
\text{startodd} & := 1; \\
\text{wait} & := 0; \quad \{\text{end of waiting for a nonzero bit}\} \\
\text{swap} & := \text{not a}; \\
\text{shift} & := \text{not (a and b)}
\end{align*}
\]
end
else if wait then \text{epsout} := \text{eps2} \quad \{\text{normal speed}\}
else if shift then \{\text{shift b faster than a, may also swap}\}
begin
\[
\begin{align*}
\text{aout} & := (\text{bout and swap}) \text{ or (aout and not swap)}; \quad \{\text{normal speed}\} \\
\text{bout} & := (a \text{ and swap}) \text{ or (b and not swap)}; \quad \{\text{fast speed}\} \\
\text{epsout} & := (\text{eps and neg}) \text{ or (epsout and not neg)}; \\
\text{neg} & := \text{negin and not (eps and startoddout)}; \quad \{\delta \text{ may become zero}\} \\
\text{negout} & := \text{neg}
\end{align*}
\]
end
else if startoddout then
begin
\[
\begin{align*}
\text{epsout} & := \text{eps2}; \quad \{\text{normal speed}\} \\
\text{swap} & := \text{not neg}; \\
\text{neg} & := \text{negin or not eps2}; \quad \{\delta := -|\delta|\} \\
\text{negout} & := \text{neg}; \\
\text{aout} & := \text{aout or swap}; \quad \{\text{swap implies b}\} \\
\text{bout} & := 0; \quad \{\text{and new b is even}\} \\
\text{carry} & := a \oplus b; \quad \{\text{may be borrow or carry}; \ominus \text{ is exclusive or}\} \\
\text{minus} & := \text{not carry} \quad \{1 \text{ iff we form } (b - a) \text{ div } 2\}
\end{align*}
\]
end
else \{not startoddout\}
begin
\[
\begin{align*}
\text{epsout} & := \text{eps2}; \quad \{\text{normal speed}\} \\
\text{aout} & := (\text{bout and swap}) \text{ or (aout and not swap)}; \quad \{\text{normal speed}\} \\
\text{bout} & := a \oplus b \ominus \text{carry}; \quad \{\text{fast speed}\} \\
\text{carry} & := \text{majority (b, carry, a \oplus minus)} \quad \{\text{majority true if 2 or 3 of its arguments are true}\}
\end{align*}
\]
end.
\[ \text{cells as defined in Appendix A are sufficient to compute } \gcd(a, b) \text{ for any } n\text{-bit integers } a \text{ and } b, \text{ where} \]
\[ c = \frac{\sqrt{17} - 1}{2} \approx 3.1105. \]

\textbf{Proof}

The result is trivial if \( a \) or \( b \) is zero, so suppose both \( a \) and \( b \) are nonzero.

Let \( 2^k (k \geq 0) \) be the largest power of two which divides both \( a \) and \( b \). Then the loop \( L0 \) of Algorithm PM reduces \( (a, b) \) to \( \left( \frac{a}{2^k}, \frac{b}{2^k} \right) \) and its systolic implementation uses \( k \) cells. At this point
\[ \delta = 0, \quad |a| \leq 2^{n-k}, \quad |b| \leq 2^{n-k}, \quad \text{and at least one of } a \text{ and } b \text{ is odd.} \]
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 \( \delta \).