**RISC-Linz**
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe

# Systolic Algorithms for Long Integer GCD Computation

Tudor JEBELEAN

(August 19, 1994)

RISC-Linz Report Series No. 94-58



Supported by: National Science Foundation (FWF), project no. P10002

Published in: Proceedings of CONPAR 94, Linz, Austria, Sept. 1994.

# Systolic Algorithms for Long Integer GCD Computation

Tudor Jebelean
RISC-Linz, A-4040 Austria
tjebelea@risc.uni-linz.ac.at

**Abstract.** Starting from a slightly improved version of the bit-level Brent–Kung plus-minus GCD algorithm, we find a word-level generalization which is suitable for systolic parallelization on multiprocessor architectures. The resulting semi-systolic algorithm is then transformed into a purely systolic one, without loss of efficiency: two-fold time increase with two-fold space decrease. The technique used for eliminating global broadcasting is more suitable to least-significant digits first arithmetic algorithms than Leiserson conversion lemma. We also present experimental results on MasPar (SIMD architecture), which confirm the linear-time complexity of our algorithm.

## 1 Introduction

Computation of the Greatest Common Divisor of two integers is the problem for which the oldest nontrivial algorithm was invented, and this algorithm is still in use [10], although with some modifications which make it more efficient on conventional computers [12, 7]. An alternative to the old Euclidean algorithm is the *binary* algorithm of [17], whose recent generalization [4, 18] proves to be at least as efficient as the Lehmer-Euclid scheme [6, 16]. All these algorithms have quadratic complexity [3], but theoretically faster schemes as [15] do not seem to be efficient for practical problems.

*Parallelization* of the Euclidean GCD algorithm is difficult because of the carry propagation. Systolic parallelization [13, 11] of (a variant of) the binary algorithm was more successful. [14] gives an idea to this effect, but his algorithm is apparently still quadratic in the worst case. [1] introduce a systolic algorithm having linear time and space complexity. The basis for their algorithm is a modified version of the binary GCD algorithm: the *plus-minus* algorithm, which is suitable for parallelization because all the decisions concerning the computation use only the *least-significant* digits of the operands. Sublinear parallel algorithms are described in [9, 2, 16], but these are apparently unfeasible due to large area/memory consumption and to the practical limitations of the parallel model which they use (PRAM).

The present paper presents a semi-systolic and a purely systolic algorithm for integer GCD computation at word-level (or high-radix), as well as practical

* Supported by Austrian Forschungsförderungsfonds (FWF), project P10002-PHY.

experiments using MasPar computer (SIMD architecture). The novel algorithms solve the problems of sign detection and variable-length operands, which are not treated in the old algorithm. The word-level algorithm is based on a generalization of a slightly modified version of the *plus-minus* algorithm [1]. This generalization allows the design of a systolic algorithm in *high-radix* (for instance - at word level), which is essential for implementation on multiprocessor machines. The previously known algorithm of Brent and Kung is designed at *bit* level, which makes it suitable only for hardware implementation.

## 2 The bit-level plus-minus algorithm

We start from a slightly simplified version of the *plus-minus* algorithm introduced by [1]. In the description in figure 1, $A, B$ are the input integers and $a_0, a_1, b_0, b_1$ denote their least-significant bits. $A_s, B_s$ hold the shifted values of $A, B$ after the common least-significant null bits are removed. They can be used together with the pseudo-GCD given by the algorithm in order to find the normalized fraction $A/B$. The other algorithms in this paper will also return this pseudo-GCD, but we will not emphasize $A_s, B_s$ anymore. Simple modifications in each algorithm will provide the shifted values. Also, it is quite obvious how to modify these algorithms by counting the number of binary shifts at the beginning and then shifting back the pseudo-GCD at the end.

[ 0] `pmGCD`$(A, B)$ [plus-minus]
[ 1] `while` $B \neq 0$ `do`
[ 2]     `if` $a_0 = 0$ `and` $b_0 = 0$ `then` [shift both]
[ 3]         $(A, B) \leftarrow (A/2, B/2)$
[ 4]         $(A_s, B_s) \leftarrow (A, B)$
[ 5]     `if` $a_0 = 0$ `and` $b_0 = 1$ `then` [interchange and shift]
[ 6]         $(A, B) \leftarrow (B, A/2)$
[ 7]     `if` $a_0 = 1$ `and` $b_0 = 0$ `then` [shift $B$]
[ 8]         $(A, B) \leftarrow (A, B/2)$
[ 9]     `if` $a_0 = 1$ `and` $b_0 = 1$ `then` [plus-minus]
[10]         `if` $a_1 = b_1$ `then` $(A, B) \leftarrow (B, (A - B)/2)$
[11]                     `else` $(A, B) \leftarrow (B, (A + B)/2)$
[12] `return` $A$ [the pseudo-GCD], $A_s, B_s$;

**Fig. 1.** The plus-minus algorithm.

The algorithm **terminates** because the absolute values of the operands $A, B$ decrease during computation. First let us note that the step [ 3] (shift both) can only occur finitely many times, while the step [ 5] (interchange and shift) can occur at most once. After that, only steps of type [ 7] (shift $B$) and [ 9] (plus-minus) may occur. Suppose $|A|, |B| \leq 2^n$ at a certain moment. After a finite number of steps [ 7], the situation [ 9] will occur. $A$ is then replaced by $B$, hence again $|A| \leq 2^n$. $B$ is replaced by $(A \pm B)/2 \leq 2^n$, but because of the choice of the sign $\pm$ this number is also even, hence at the next step a "shift $B$" will be performed. $|B|$ becomes then at most $2^{n-1}$, hence we have now $|A| \leq 2^n$, $|B| \leq 2^{n-1}$. Repeating the reasoning above, one can see that after another finite succession of shifts and *one* plus-minus step, the situation becomes $|A|, |B| \leq 2^{n-1}$. Hence, **two** plus-minus steps together with some shifts will always bring down the length of the operands with at least one bit. It follows that after a finite number of steps one of the operands becomes null. However, only $B$ can become null (by line [11]), and then the loop [ 1] terminates.

**Correctness** follows from the fact that all the transformations of the pair $(A, B)$ are GCD-preserving – except line [ 3] which eliminates the common power of 2. After exiting the loop $|A| = GCD(A, 0) = GCD(A_s, B_s)$. Due to the subtraction in line [10], the operands might become negative during the computation. Therefore, the result might be negative (its absolute value is the pseudo-$GCD$). In order to retrieve the absolute value of the result, its sign has to be known – this is a problem which has to be carefully treated in the parallel algorithm. Complementing the value of the result in digit-parallel fashion is a constant-time operation, because the pseudo-$GCD$ is odd.

Experimentally, the number of steps for 1,000,000 random inputs having [exactly] 31 bits was: minimal 26, average 67.55 and maximal 100, that is $0.83n$, $2.17n$ and $3.22n$, respectively.

The main difference between the above algorithm and Brent-Kung plus-minus algorithm is that we do not use the [approximative] difference of bit-lengths of the operands ($\delta$ in [1]). This simplifies the algorithm and also makes it more suitable for systolic implementation. Contrary to the appearences, using $\delta$ is not necessary for insuring convergence of the algorithm. Moreover, and somewhat surprising, using $\delta$ does not make the algorithm faster. Running the Brent-Kung algorithm on the same data set as before (1,000,000 random numbers of 31 bits), we obtained 31 steps minimal, 77.64 average and 101 maximal, that is $n$, $2.50n$ and $3.25n$, respectively. This is an increase by 15% of the average running time.

## 3 The word-level algorithm

The bit level algorithm is suitable for parallelization on very fine-grained architectures – e. g. hardware implementation.

However, if a multiprocessor architecture is the target machine, simulating a bit-level algorithm on word-oriented processors will significantly decrease the efficiency of the implementation. We show in [4] how to generalize the plus-minus algorithm at word-level, thus making it suitable for operation in high radix.

We outline here the basic ideas of this generalization. The bit-level plus-minus algorithm consists of alternating *shift* operations with plus-minus operations. We shall call the later *inter-reductions.* Shift operations are applied as long as at least one of the operands has null trailing bits. When there are no more trailing zero bits, and *inter-reduction* is applied in order to obtain a new operand with trailing zero bits. The way of making the inter-reduction is the key of the algorithm, which we generalize. Within the bit-level algorithm, the trailing *pairs of bits* are inspected and depending on their values *two cofactors* are determined (these are (1, 1) – note that they are only *one-bit long*) and a *sign*, which is + or −. The linear combination of the operands using these two cofactors and the sign (that is $A + B$ or $A - B$) has the property that its trailing pair of bits is null.

The **generalization** of the inter-reduction operation works as follows: the trailing *pairs of words*[2] of the operands are examined. Depending on their values two cofactors are determined, say $(x, y)$, each one word long, and a sign, such that the linear combination $C = xA \pm yB$ has its last pair of words null. One sees that by shifting $C$ becomes one word shorter than the maximum of $A, B$, exactly in the same way in which in the bit-level algorithm the sum (difference) becomes one bit shorter after shifting. However, the word-level algorithm has the advantage that a reduction of one word is performed at each step.

The essential detail remains, however, to prove that the desired cofactors can always be found, and if so to find an efficient method to compute them. Again, we outline here the basics of the method, whose details can be found in [4] (recently [18] found a simpler correctness proof). Let us denote by $a_0, a_1, b_0, b_1$ the least-significant words of $A, B$, and by $w$ the bit-length of the word. We are looking for $x, y$ such that $x, |y| < 2^w$ and $xa + yb = 0 \bmod 2^{2w}$, where $a = a_1 2^w + a_0, b = b_1 2^w + b_0$. We further assume that $a, b$ are odd, because trailing binary zeroes are already removed by shifting. Then one can compute $c = ab^{-1} \bmod 2^{2w}$, and the target relation becomes $xc + y = 0 \bmod 2^{2w}$, which is $k2^{2w} - xc = y$ for some $k$. If $c < 2^w$, then $(x, y) = (1, -c)$ are already the desired cofactors, but this happens only in $1/2^w$ of the situations. In the other situations, one can apply the extended Euclidean algorithm [10] to $2^{2w}$ and $c$, and find a remainder sequence $\{a_i\}$ and a cosequence of cofactors $u_i, v_i$ (for $i = 0, 1, \ldots$) such that $u_i 2^{2w} - v_i c = a_i$, hence $(v_i, a_i)$ are good candidates for $(x, y)$. It remains to choose that $i$ for which $v_i, a_i$ are smaller than $2^w$. Almost surprisingly it turns out that such an $i$ always exist. This is due to the fact that $a_{i-1}|v_i| < 2^{2w}$ for all $i$ (for a proof of this relation see [4] or [18]). Hence, if we stop the extended Euclidean algorithm at the first iteration when $a_i$ becomes smaller than $2^w$, then $|v_i|$ is also smaller than $2^w$ (consequence of $a_{i-1} > 2^w$).

However, the transformation $(A, B) \leftarrow (B, xA \pm yB)$ has the unpleasant property that it is *not* GCD preserving. Due to this, the resulting pseudo-GCD $G'$ is in general a multiple of the true $\mathrm{GCD}(A, B)$: some spurious factors are introduced at each inter-reduction, which are the common divisors of $B$ and $x$. These factors are in general small, hence the correction $G = GCD(A, B, G')$ by using the classical Lehmer-Euclid algorithm at the end will not be expensive (experimentally we noted an average length of 0.5 bits per inter-reduction step for $w = 32$). In the sequential version the algorithm will still outrun all the other known GCD algorithms [6].

[2] By *word* in this paper we understand a succession of bits – typically 16 or 32 – but the algorithms are valid for any word-length.

Having in mind the parallel implementation, however, one needs to avoid the final correction, because Lehmer-Euclid algorithm is difficult to parallelize. We transform the previous algorithm by using two pairs of cofactors, such that the new inter-reduction operation is GCD preserving. The algorithm is presented in figure 2. The routine `cofactors` used in line [12] delivers *two* pairs of cofactors. One pair is $(x, y)$ found as exposed above, while the other pair $(x', y')$ is chosen from $(v_{i-1}, a_{i-1})$ and $(v_{i+1}, a_{i+1})$ (the previous and the next pair in the extended Euclidean algorithm), such that $(x', y')$ is minimal. For the second pair we cannot be sure that $x'$ $(y')$ is smaller than $2^w$, but the inter-reduction still shortens the length of the operands enough for the algorithm to be convergent. (Experimentally we detected that for $w = 16$ in more than 99.9% of the cases the reduction is at least 10 bits.) Also, now the resulting GCD will be correct, because the transformation $(A, B) \leftarrow (xA + yB, x'A + y'B)$ is GCD preserving – up to a power of 2. This follows in a straightforward manner from the properties of the extended Euclidean algorithm. However, in the (rare) cases when $c < 2^w$, the device with the extended Euclidean algorithm does not work: here we choose (0,1) as the second pair of cofactors ($B$ is left unchanged). The incidence of such cases is so low (experimentally, for $w = 16$, 20 out of 1,000,000 cases) that they do not influence the average speed of the algorithm, however one has to take them into account when designing the parallel algorithm.

[ 0] `wGCD`$(A, B)$ [word-level]
[ 1] `while` $(A > 2^w)$ `or` $(B > 2^w)$ `do` [main loop]
[ 2]     [shift]
[ 3]     `if` $A = 0$ `then return` $B$
[ 4]     `while even`$(A)$ `do` $A \leftarrow A/2$
[ 5]     `if` $B = 0$ `then return` $A$
[ 6]     `while even`$(B)$ `do` $B \leftarrow B/2$
[ 7]     [inter-reduction]
[ 8]     $(x, y, x', y') \leftarrow$ `cofactors`$(a, b)$
[ 9]     $(A, B) \leftarrow (xA + yB, x'A + y'B)$
[10] `return EuclidGCD`$(A, B)$;

**Fig. 2.** The word-level plus-minus algorithm.

If the GCD is multiple-precision (bigger than $2^w$), then one of the operands becomes null and the algorithm will terminate in line [ 3] or [ 5]. If the GCD is simple-precision (smaller than $2^w$), then it usually happens that both operands become simple-precision at a certain step. In this case is more economic to compute the GCD by a simple-precision algorithm, for instance the Euclidean one (line [10]). Since we have now simple-precision operands, this is a constant-time computation. As in the case of the bit-level algorithm, the sign of the result must be investigated in order to retrieve the positive value of the GCD.

## 4 A semi-systolic algorithm

We shall use here the term "semi-systolic" [13] with a slightly modified meaning. Namely, while the cells of our systolic array are finite-state machines with neighbor-to-neighbor communication, we will allow some of the data to be *broadcasted* to all the processors from the first processor (see figure 3). This model is realistic if we consider hardware implementations where the array is not very big (e. g. it fits on one chip), or if we consider multiprocessor implementation on SIMD architecture (e. g. MasPar).

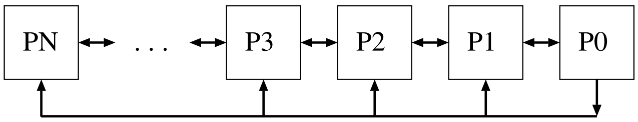


**Fig. 3.** Systolic array with global broadcasting.

The semi-systolic parallelization of the word-level plus-minus algorithm is done on an array of $N$ processors, where $N > n$ (the length of the operands). The operands are stored digit by digit in the processors, using the redundant representation in order to avoid carry rippling. Namely after each interreduction the carries will not be propagated along the array, but just added to the appropriate digits when a shift occurs.

The following variables occurring in the description of the algorithm have an instance on each processor:

- $a, b$ contain the digits of the operands $A, B$. In the processors corresponding to non-significant digits these are null.
- $c_a, c_b$ contain the carries which appear during computation. Because the carries are only partially absorbed at each step, together with $a, b$ they constitute the redundant representation of the operands. Each carry has to be represented on two words in order to avoid overflow.
- $t_a, t_b$ are *tags* which indicate the length of $A, B$. Initially the tags are set to 1 in the processors corresponding to the significant digits and 0 in rest. The tags are essential for detecting the termination situations and the sign of the result.

The other variables are handled only on processor P0 (which stores the least-significant digits of the operands) and are broadcasted to all the other processors when needed:

- $m$ indicates the number of bits for shifting.
- $x, y, x', y'$ are the cofactors used for inter-reduction.
- $I$ contains the code of the instruction to be executed.

[only on P0: find next instruction]
if $t_a^+ = t_b^+ = 0$ then Return(EclidGCD($a, b$)
else if $a = 0$ then
    if $t_a = 0$ then Return($B$)
        else $I \leftarrow$ ShiftWordA
else if even($a$) then $I \leftarrow$ ShiftBitsA(trailingBits($a$))
else if $b = 0$ then
    if $t_b = 0$ then Return($A$)
        else $I \leftarrow$ ShiftWordA
else if even($b$) then $I \leftarrow$ ShiftBitsB(trailingBits($b$))
else $I \leftarrow$ InterReduce(cofactors($a, b$))

**Fig. 4.** Finding the instruction on processor P0.

[instruction execution: on all processors]
case $I$ of [$I$ selects instruction]
    ShiftWordA: $(c_a, a) \leftarrow a^+ + c_a$; $t_a \leftarrow t_a^+$
    ShiftBitsA($m$): $a \leftarrow$ cat(low$_m(a^+)$, high$_m(a)$)
        $c_a \leftarrow$ cat(low$_m(c_a^+)$, high$_m(c_a)$)
    ShiftWordB: $(c_b, b) \leftarrow b^+ + c_b$; $t_b \leftarrow t_b^+$
    ShiftBitsB($m$): $b \leftarrow$ cat(low$_m(b^+)$, high$_m(b)$)
        $c_b \leftarrow$ cat(low$_m(c_b^+)$, high$_m(c_b)$)
    InterReduce($x, y, x', y'$):
        $(c_a, a) \leftarrow x(a + c_a^-) + y(b + c_b^-)$
        $(c_b, b) \leftarrow x'(a + c_a^-) + y'(b + c_b^-)$
        $t_a, t_b \leftarrow t_a^- \vee t_b^-$

**Fig. 5.** Execution of the instruction on all processors.

The processor P0 acts as a "driver" for the array, according to the program in figure 4, while the other processors will only execute the "routines" described in figure 5. Processor P0 also executes these routines. For the variables which have an instance on each processor we use the notation $v^-$ (resp. $v^+$) to designate the value of the variable on the right (resp. left) neighboring processor.

**ShiftWordA** and **ShiftWordB** move the operands one word rightward. Note that during this move one step of carry absorption is also performed. Also, the tag is moved right, which allows termination detection. **ShiftBitsA**($m$) and **ShiftBitsA**($m$) shift the operands right with $m$ trailing null bits. They use **cat**($u, v$) to concatenate two strings of bits and **high**$_m$ (**low**$_m$) to extract the higher (lower) bits of the respective words. The carries are also shifted and absorbed one step. `InterReduce`$(x, y, x', y')$ performs the two linear combinations.

Detection of the sign of the result can be done using the tags: if an operand is negative, then its value in the processor holding the first null tag is non-null.

The algorithm above can be easily implemented on word-oriented SIMD machines. In [8] we describe experiments using the MasPar architecture with various long integer operations – including GCD computation. The timings are presen-

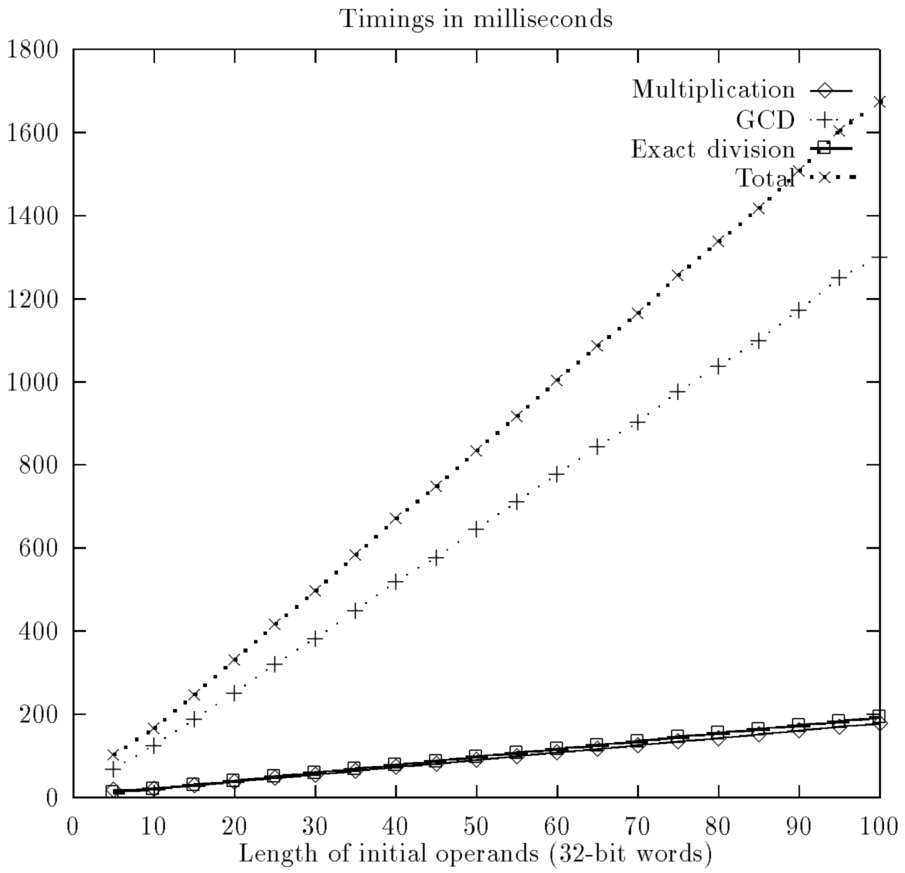


**Fig. 6.** Timings of MasPar implementation.

ted in figure 6, together with the timings for addition, multiplication, and exact division. These timings refer to the "rational reduction" operation encountered, for instance, in Gröbner bases computation: $A/B + X/Y * C/D$. (the result has to be normalized by exact division with the GCD). One can see that, on one hand, GCD computation is by far the most expensive operation, and on the other hand, that the running time depends linearly on the lengths of the inputs (15 to 300 words).

## 5 A purely systolic algorithm

In certain contexts the use of global broadcasting is not appropriate. For instance, global broadcasting could reduce the efficiency of the implementation on MIMD architectures having only local communications capabilities. Therefore, we will show in this section how to eliminate global broadcasting from the previous version of the systolic algorithm.

It is necessary to replace the broadcasting of $I$ with a pipelining of it along the array of processors. One way to do this is by using the systolic conversion lemma of [13]. This will increase the number of variables by a factor of 3, and will also introduce an unnecessary buffering and handling of the carries. Indeed, if the instructions $I$ have to be sent sequentially along the array, there is no point in keeping the complicated scheme of redundant arithmetic (or parallel carry propagation) described before. Rather, it is much simpler to ripple the carries in the classical fashion, together with the instruction signal. We emphasize the communications of a processor in figure 7, where one can see the *bidirectional* communication.

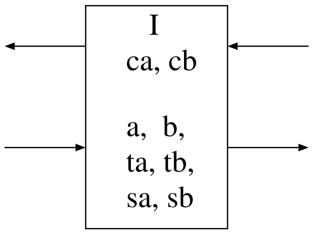


**Fig. 7.** Structure of one systolic processor for GCD computation.

Similarly to what happens in the systolic conversion lemma, the speed of the purely systolic algorithm is halved. This is due to the bidirectional communication. After P0 decides upon the instruction to be performed, it sends it to processor P1, and it has to wait one step for the result of the left neighbor to come back. Thus, for transforming the previous algorithm it is necessary to transform the instruction finding scheme in figure 4 by adding the following 3 lines between [ 0] and [ 1]:

[0.1] `if` $I \neq$ `Wait then` $I \leftarrow$ `Wait`
[0.2] `else`

The instruction execution is modified in several respects, the new routines are shown in fig. 8. The carries are now handled only by the inter-reduction routine: they will be pipelined leftward together with the `InterReduce` signal. Two new variables $s_a, s_b$ are introduced in order to detect the sign of the result. These variables get the values of the lowest-order non-significant digit of $A$ (resp. $B$), that is 0 iff $A$ (resp. $B$)is positive, whenever an $A$ and $B$ are modified (line [13]). The values of $s_a, s_b$ are shifted rightward at each word-shift (lines [ 4], [ 7]), such that when the signal $t$ indicating that an operand is null reaches P0, the signal $s$ indicating the sign of the other operand will also reach P0.

[ 0] [purely systolic version: instruction execution on all processors]
[ 1] `case` $I$ `of` [$I$ selects instruction]
[ 2] `Wait`: [do nothing – each two steps]
[ 3] `ShiftWordA`: $a \leftarrow a^+$; $t_a \leftarrow t_a^+$
[ 4] $s_a \leftarrow s_a^+$; $s_b \leftarrow s_b^+$
[ 5] `ShiftBitsA`$(m)$: $a \leftarrow$ `cat`$(\mathtt{low}_m(a^+), \mathtt{high}_m(a))$
[ 6] `ShiftWordB`: $b \leftarrow b^+$; $t_b \leftarrow t_b^+$
[ 7] $s_a \leftarrow s_a^+$; $s_b \leftarrow s_b^+$
[ 8] `ShiftBitsB`$(m)$: $b \leftarrow$ `cat`$(\mathtt{low}_m(b^+), \mathtt{high}_m(b))$
[ 9] `InterReduce`$(x, y, x', y')$:
[10] $(c_a, a) \leftarrow x(a + c_a^-) + y(b + c_b^-)$
[11] $(c_b, b) \leftarrow x'(a + c_a^-) + y'(b + c_b^-)$
[12] $t_a, t_b \leftarrow t_a^- \vee t_b^-$
[13] `if` $t_a = 0$ `then` $s_a \leftarrow a$; $s_b \leftarrow b$

**Fig. 8.** Execution of the instruction on all processors, purely systolic version.

The waiting instructions propagate to all the array. As a result, the *efficiency* of the array is halved, since half of the processors are idle at each step: at step 0 processors $0, 2, \ldots, 2k$ work, at step 1 processors $1, 3, \ldots, 2k+1$ work, and so on (see fig. 9). We cannot eliminate the slow-down, but we can in turn limit the decrease of efficiency, by halving the number of processors. Namely, we "pack" pairs of adjacent processors into one (see fig. 9): we have now $n/2$ processors, each with 2 instances of each variable. The computation is "multiplexed" alternatively between the two sets of data corresponding to the two initial processors. This also reduces the interprocessor communication, which will improve the performance on most architectures.

## 6 Conclusions

Until now, systolic algorithms for integer GCD computation have been developed only at *bit-level*, which makes them unsuitable for multiprocessor machines. We present a *word-level* semi-systolic algorithm for long integer GCD computation, based on a generalization of the plus-minus algorithm, as well as its performance on MasPar SIMD architecture. By using an "instruction-pipelining" technique we are able to transform this algorithm into a purely systolic one, which is suita-

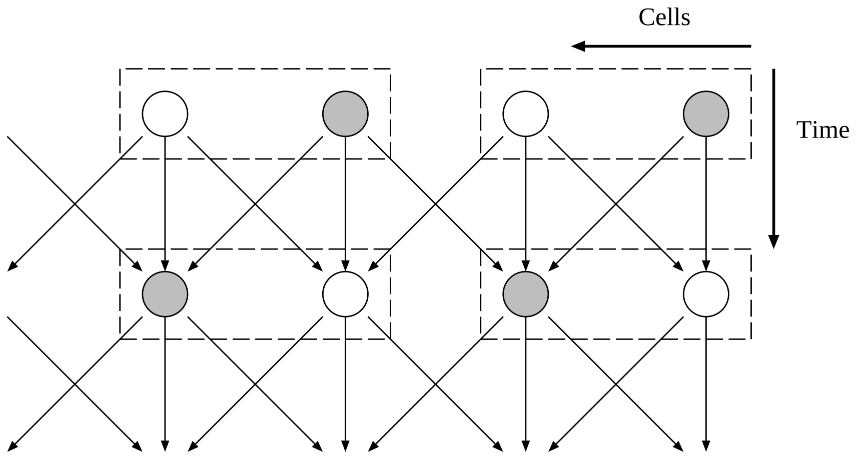


**Fig. 9.** Active-passive processors.

ble for implementation on distributed memory MIMD machines (e. g. transputer array).

It is noticeable that this technique can be used for the elimination of global broadcasting for other least-significant digits first arithmetic algorithms. Essentially it consists in:

- combining the global signal with the carry into a signal which is to be piped leftward through the array;
- reducing the number of functional units by packing two processors in one cell.

For instance, a good candidate for this technique is the *exact division* algorithm introduced in [5].

## References


1. R. P. Brent and H. T. Kung. A systolic algorithm for integer GCD computation. In K. Hwang, editor, *Procs. of the 7th Symp. on Computer Arithmetic*, pages 118–125. IEEE Computer Society, June 1985.
2. B. Chor and O. Goldreich. An improved parallel algorithm for integer GCD. *Algorithmica*, 5:1–10, 1990.
3. G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3:1–10, 1974.
4. T. Jebelean. A generalization of the binary GCD algorithm. In M. Bronstein, editor, *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, pages 111–116, Kiev, Ukraine, July 1993. ACM Press.
5. T. Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15(2):169–180, February 1993.

6. T. Jebelean. Comparing several GCD algorithms. In *ARITH-11: IEEE Symposium on Computer Arithmetic*, pages 180–185, Windsor, Canada, June 1993.
7. T. Jebelean. Improving the multiprecision Euclidean algorithm. In *DISCO'93: International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 45–58, Gmunden, Austria, September 1993. Springer Verlag LNCS 722.
8. Tudor Jebelean. *Systolic multiprecision arithmetic*. PhD thesis, RISC-Linz, April 1994.
9. R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM Journal on Computing*, 16:7–16, 1987.
10. D. E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, 2 edition, 1981.
11. H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982.
12. D. H. Lehmer. Euclid's algorithm for large numbers. *Am. Math. Mon.*, 45:227–233, 1938.
13. C. E. Leiserson. *Area-efficient VLSI computation*. PhD thesis, Carnegie-Mellon University, 1982.
14. G. B. Purdy. A carry-free algorithm for finding the greatest common divisor of two integers. *Computers & Mathematics with Applications*, 9:311–316, 1983.
15. A. Schönhage. Schnelle Berechung von Kettenbruchentwicklugen. *Acta Informatica*, 1:139–144, 1971.
16. J. Sorenson. Two fast GCD algorithms. *J. of Algorithms*, 16:110–144, 1994.
17. J. Stein. Computational problems associated with Racah algebra. *J. Comp. Phys.*, 1:397–405, 1967.
18. Ken Weber. The accelerated integer GCD algorithm. Technical report, Kent State University, 1993. To appear in ACM Trans. on Math. Software.

This article was processed using the LaTeX macro package with LLNCS style