1

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

Designing Systolic Arrays for Integer GCD Computation

Tudor JEBELEAN

(August 3, 1994)

RISC-Linz Report Series No. 94-47

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

Published in: Proceedings, ASAP'94 (Application Specific Array Processors), IEEE Computer Society.

Designing Systolic Arrays for Integer GCD Computation

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

Abstract

We improve the classical result of Brent and Kung by a factor of 12 in area consumption, while maintaining the same average running time. Global broadcasting is eliminated using a novel technique which is more efficient then Leisersons semisystolic-to-systolic transformation and can be also applied to other arithmetic algorithms. Experiments using field programmable gate arrays demonstrate the possibility of speeding-up long integer arithmetic by two orders of magnitude by implementing this algorithm in dedicated hardware.

1: Introduction

This paper reports an advance within an on-going project for speeding-up *long-integer arithmetic* by using dedicated hardware [15]. Long-integer arithmetic is a key component of *computer algebra systems*, which have witnessed a spectacular growing of interest during the last decade [1, 3]. Also known as *symbolic computation*, the new area tends to replace or cooperate with classical (floating point based) numeric computation in more and more applications [7, 6]. However, due to the fact that arbitrary precision arithmetic has to be used in these systems, they suffer from a relative slowness, which is largely due to the long integer arithmetic, which consumes in some applications more than 99% of the computing time [4, 21]. The computation of the greatest common divisor (GCD) is particularly expensive in this context [13]. In fact GCD of long integers is also important in other areas - as for instance in *cryptography*.

Corresponding to its importance, GCD computation has received much attention in the literature - see [18] for a survey of classical sequential algorithms of Euclid, Lehmer [19] and *binary* [25]. Recent results on practically fast computation of long integer GCD are [14, 11, 24, 26]. All the above algorithms have $O(n^2)$ time-complexity in the length of the inputs [8], and even if algorithms with lower theoretical complexity exist - best is [23] with $O(n \log^2 n \log\log n)$ - these are apparently not efficient for the range of problems encountered in practice. Due to carry propagation, the Euclidean algorithm is not suitable for [systolic] parallelization, but the binary GCD algorithm seems more appropriate. A suggestion to this effect was given in [22], but effective parallelization was only achieved by [2], using a modified version of the binary algorithm - the *plus-minus* algorithm. A relative improvement of their scheme is suggested in [27] - but this in turn uses global broadcasting. Several VLSI implementation of the above ideas are described in [10]. A novel systolic implementation of the *plus-minus* algorithm is described in [16] - and this is in fact the precursor of the algorithm we will develop in this paper. All these systolic algorithms have time and area complexity $O(n)$. Sublinear parallel algorithms are described in [17, 5, 24], but these are apparently unfeasible due to large area/memory consumption and to the practical limitations of the parallel model which they use (PRAM).

*Supported by Austrian FWF project *Arbitrary Precision Arithmetic on Parallel Architectures*.

We present here a significant practical improvement of the Brent and Kung systolic GCD algorithm: a reduction of the area consumption by a factor of 12, while keeping the average running time at the same level. Namely, while Brent-Kung algorithm uses $4n$ processors with 24 [1-bit] registers each, our scheme uses $n/2$ processors with 16 bits each. This leads to a 1/12 reduction in the number of registers. The area reduction is in fact even bigger, since the transition function of the new design is handling a smaller number of variables. The *average running time* is roughly the same: $4n$ steps. For the old algorithm this is the running time for *any* input, because the operands are piped *through* an array of $4n$ processors. The running time of new algorithm is *dependent* on the actual input, and ranges from n (best case) to $6n$ (worst case), with an [experimentally determined] average of $4n$. It is important to note that the new algorithm will solve smaller problems in shorter time - which is essential for applications like variable-precision arithmetic, in contrast to the old algorithm whose running time is *fixed*, thus limiting its usage to fixed-size applications like cryptography. Finally, we note that the new algorithm also delivers the *sign* of the result, a problem which is not solved by the old algorithm, and which might require another n in space and time requirements.

The technique which is used for the elimination of global broadcasting is more efficient for arithmetic problems than the systolic conversion lemma of Leiserson [20]. Leisersons technique requires the triplication of the sets of registers containing the states of the processors. Also, in the case of arithmetic algorithms this technique will add unnecessary buffering and processing of the carries. Our technique only adds the registers needed to buffer the signals which are [initially] globally broadcasted.

Experiments carried out using field programmable logic from Atmel (was Concurrent Logic) show that moderate-sized chip could accommodate 16-bits operands. Namely, an 8-bit design consumes 313 [Atmel] gates and 92 registers, using 1500 cells after layout and routing (2210 equivalent gates). A device for longer operands can be realized by simply tiling several such chips in a row. The circuit can be operated at 15 MHz, that means that the speed of solving a 32-bit problem will be in the same range as that of a fast workstation, but because the timing of the systolic array increases only linearly, for longer operands a speed-up [roughly] equal to the number of words could be achieved (e. g. 10 times for 10-word operands). Due to the absence of global broadcasting, the timing will not be affected by the size of the circuit.

2: The basic algorithm

We start from a slightly simplified version of the *plus-minus* algorithm introduced by [2]. The algorithm is described in fig. 1. A, B are the input integers and a_0, a_1, b_0, b_1 denote the least significant bits of A, B.

In the context of our application (exact rational arithmetic), there is no need to recover the right-shifted value of the GCD. Rather, we save in A_s, B_s the values of the operands after the elimination of the common trailing binary zeroes, and we use these values together with the pseudo-GCD for normalization.

This algorithm is terminating because the operands A, B decrease during computation. First let us note that the steps [shift both] and [interchange and shift] may occur only finitely many times. After that, only steps of type [shift B] and [plus-minus] may occur. Suppose $A, B \leq 2^n$ at a certain moment. After a finite number of shifts, the situation $a_0 = 1 \texttt{ and } b_0 = 1$ 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

```
[GCD of A, B]
while B ≠ 0
   if a_0 = 0 and b_0 = 0 [shift both]
      (A, B) ← (A/2, B/2)
      (A_s, B_s) ← (A, B)
   if a_0 = 0 and b_0 = 1 [interchange and shift]
      (A, B) ← (B, A/2)
   if a_0 = 1 and b_0 = 0 [shift B]
      (A, B) ← (A, B/2)
   if a_0 = 1 and b_0 = 1 [plus-minus]
      if a_1 = b_1
         (A, B) ← (B, (A − B)/2)
      else
         (A, B) ← (B, (A + B)/2)
return A [the pseudo-GCD], A_s, B_s;
```

Figure 1. A variant of the plus-minus GCD algorithm.

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, and then the loop terminates.

The above algorithm can be implemented relatively easy on a systolic array using global broadcasting of the actions to be performed (we shall call them *instructions*). Namely, each processor will contain one bit of each operand, and the plus-minus operations are performed in complement arithmetic (one or both of the operands may become negative during the computation). The carries / borrows do not have to be rippled through the array, since at each step only the least-significant digits of the operands are necessary in order to continue the computation (that is, to determine the next operation). Instead of completely rippling the carries at each step, they will be computed and piped towards left at each systolic step. In other words, the operands A, B are represented in a redundant notation: by a string of values and by a string of carries. This complicates the arithmetic operations and also creates a problem in detecting the termination (test $B = 0$).

3: Elimination of global broadcasting

Some systolic algorithms (so called *semisystolic*) use global broadcasting for sending some data to all the cells in the same time. This technique is acceptable only on some architectures (for instance some SIMD machines provide an efficient mechanism for such operations), or under certain restrictions (for instance when the array is relatively small).

However, in the context of our problem the use of global broadcasting is not appropriate. We are trying to construct circuits of very large size - possibly accommodating tens of words - and we want our circuit to work well for inputs of *various* sizes. The use of global broadcasting will introduce a delay *per cycle* which depends on the length of the array - and in this case can be significantly big. This is true especially for programmable logic - where global signals are not simple wires, but buses which have a significant propagation delay. Also, global signals tend to decrease the efficiency of area utilization in programmable logic, because they consume many buses.

Therefore, we are concerned in this paper with designing a systolic circuit which uses *only*

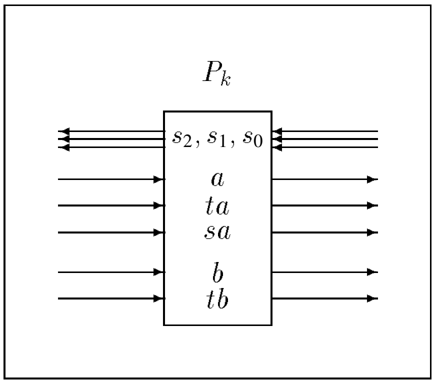

Figure 2. Structure of one systolic cell for GCD computation.

neighbor-to-neighbor communication.

If we want to eliminate the global broadcasting from the systolic algorithm hinted in the previous section, we can use the systolic conversion lemma of [20]. This will increase the number of the registers by a factor of 3, and will also introduce an unnecessary buffering and handling of the carries / borrows. Indeed, if the instructions 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 above. Rather, it is much simpler to ripple the carries in the classical fashion, together with the instruction signal. In fact, the information about the carry / borrow can be embedded in the instruction, leading to the following set of ‘‘statements’’:

- **w**: wait (no operation);
- **B**: shift both operands;
- **C**: interchange and shift B;
- **S**: shift B;
- **P**: add, carry 1;
- **p**: add, carry 0;
- **M**: subtract, borrow 1;
- **m**: subtract, borrow 0;

which can be encoded using only 3 bits.

The cells of the resulting array have the structure depicted in Fig.2, each cell having 8-bits: 3 for the instruction and 5 for the data. Besides the bits of the operands a, b, we use additional ‘‘tagging’’ bits ta, tb, which indicate the lengths of the operands: the sign-bits are tagged. Also, sa is used in order to determine the *sign* of the result. The detailed description of this systolic array is given in [16].

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 cell 0 (the rightmost) decides upon the instruction to be performed, it sends it to cell 1, and it has to wait one step for the result of the left neighbor to came back. This waiting instructions propagate then to all the array. As a result, the *efficiency* of the array is halved, since half of the processors are idle at each step. The situation is depicted in Fig. 3 (white cells are waiting).

We cannot eliminate the slow-down, but we can in turn limit the decrease of efficiency, by halving the number of processors. Namely, we can ‘‘pack’’ pairs of adjacent cells into one (this technique is known as ‘‘active-passive clustering - see e. g. [9]), and use only one implementation

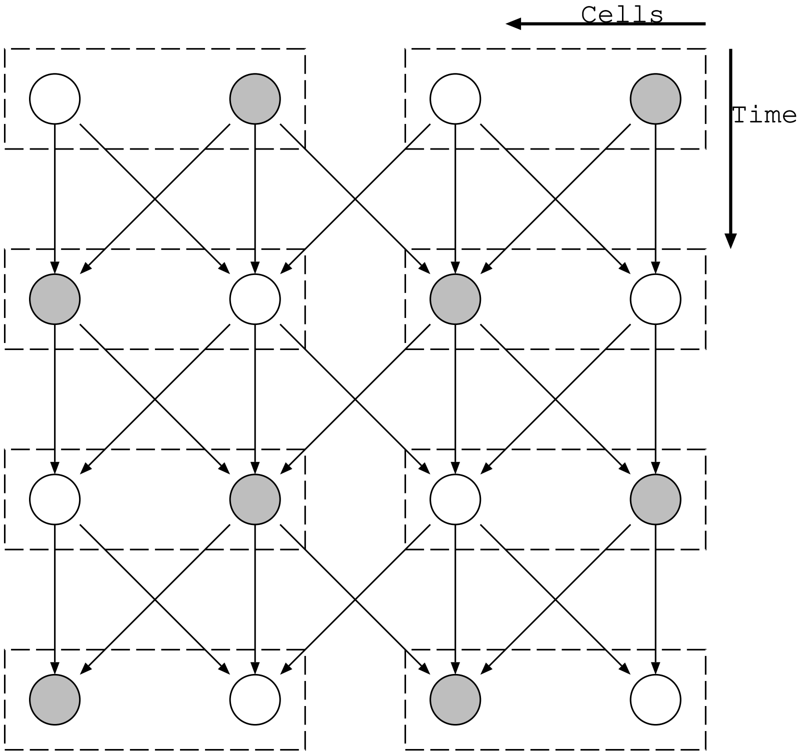

Figure 3. Active vs. waiting cells in the non-optimized algorithm.

of the transition function, which will be multiplexed alternatively between the two sets of data corresponding to the two initial cells. That means we will have $n/2$ cells, each with 16 registers. The structure of the new cells is depicted in Fig. 4. The multiplexors have to be driven by the same flip-flop signal, which selects the right-hand side entry at steps 0, 2, 4, . . . and left-hand side entry at steps 1, 3, 5, ...

The picture presents the theoretical situation in which the transition function actually depends on all the bits of the old state and of the adjacent cells states. For 8-bit states this will result 40 multiplexors. This is not the case in our algorithm, however - the transition function uses only 12 bits of input. By carefully analyzing the data flow one can realize the design with only 26 multiplexors (20 for *data* and 6 for *instructions*). In comparison to the design in [16] the transition function is also slightly simplified because there is no need to handle the wait instruction anymore.

It is noticeable that the technique presented above 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 rightward 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 [12].

4: Practical experiments

We have implemented a simulation of the systolic GCD algorithm. The number of steps for 1000 random inputs of 32 bits (GCD is small) was:

- average: 134.8 (4.2 n)

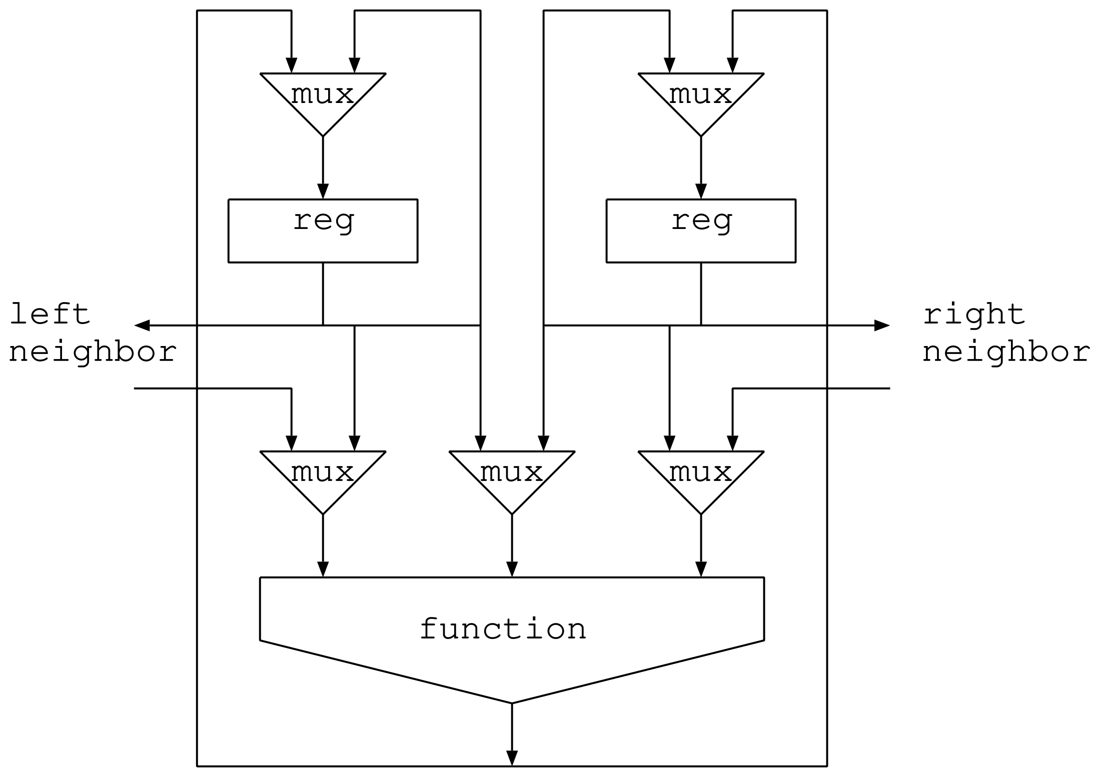

Figure 4. Structure of the new cell.

- maximal: 181
- minimal: 105

For 1000 random inputs of 32 bits with random GCD's of 16 bits the figures are:

- average: 90.5
- maximal: 117
- minimal: 65

Also, we designed a hardware implementation of the GCD algorithm using the FPGA (Field Programmable Gate Array) development system from Amtel (was Concurrent Logic, or CLi), which uses CAD software from Viewlogic.

The circuit implementing the GCD computation for 8-bit operands uses 305 cells (313 gates and 92 registers). which means that a circuit for 32-bit operands will need about 1,300 cells. After automatic layout, the 8-bit circuit consumed about a half (1,576) of a 6005 Cli programmable chip. The area reduction in comparison with the design in [16] (which uses wait states) is about 30%.

The circuit can be operated at 15 MHz (using CLi-like components), which gives an average of $135 * 64ns = 8.6\mu s$ for normalizing one-word operands.

If one computes the GCD of two 32-bit words on a fast RISC processor by the Euclidean algorithm, one needs an average of 16 steps of 8 to 12 clocks. On a processor operating at 20 to 25 MHz this will give a timing which is in the same range as the one of the hardware systolic array.

However, for longer operands a significant speed-up could be achieved. The array will be able to compute the GCD of 10-word (of 32 bits) operands in about $135 * 10 * 64ns = 0.086ms$, while only GCD computation takes $26ms$ on a DECstation 5000/200 using the Lehmer-Euclid algorithm (see [13] for timings of sequential GCD computation). Hence, a significant speed-up could be achieved.

References

[1] A. G. Akritas. *Elements of Computer Algebra with Applications*. John Wiley, 1989.

[2] 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.

[3] M. Bronstein, editor. *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, Kiev, Ukraine, July 1993. ACM Press.

[4] B. Buchberger and T. Jebelean. Parallel rational arithmetic for Computer Algebra Systems: Motivating experiments. In *3rd Scientific Workshop of the Austrian Center for Parallel Computation*, Weinberg, Austria, March 1992. Report ACPC/TR 93-3, February 1993.

[5] B. Chor and O. Goldreich. An improved parallel algorithm for integer GCD. *Algorithmica*, 5:1--10, 1990.

[6] A. M. Cohen and L. J. van Gastel, editors. *SCAFI'92: Studies in Computer Algebra for Industry II*, Amsterdam, 1992. Report Series of the Computer Algebra Netherlands Expertise Center.

[7] G. Cohen, T. Mora, and O. Moreno, editors. *AAECC-10: 10th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, Puerto Rico, May 1993. Springer Verlag. LNCS 673.

[8] G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3:1--10, 1974.

[9] P. Dewilde and E. Deprette. Architectural synthesis of large, nearly regular algorithms. *Ann. Telecom.*, 46(1-2):49 -- 59, 1991.

[10] A Guyot. OCAPI: Architecture of a VLSI coprocessor for the GCD and extended GCD of large numbers. In *ARITH-10: 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, June 1991. IEEE Computer Society Press.

[11] 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.

[12] T. Jebelean. An Algorithm for Exact Division. *Journal of Symbolic Computation*, 15(2):169--180, February 1993.

[13] T. Jebelean. Comparing Several GCD Algorithms. In *ARITH-11: IEEE Symposium on Computer Arithmetic*, pages 180--185, Windsor, Canada, June 1993.

[14] T. Jebelean. Improving the multiprecision Euclidean algorithm. In *DISCO'93: International Symposium on Design and Implementation of Symbolic Computation Systems, Gmunden, Austria (LNCS 722)*, pages 45--58. Springer Verlag, September 1993.

[15] T. Jebelean. Systolic multiprecision arithmetic using FPGA. In W. Luk and W. Moore, editors, *FPLA'93: International Workshop on Field Programmable Logic and Applications, Oxford, UK*, September 1993. To appear.

[16] T. Jebelean. Systolic normalization of rational numbers. In L. Dadda and B. Wah, editors, *ASAP'93: International Conference on Application-Specific Array Processors*, pages 502--513, Venice, Italy, October 1993. IEEE Computer Society Press.

[17] 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.

[18] D. E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, 2 edition, 1981.

[19] D. H. Lehmer. Euclid's algorithm for large numbers. *Am. Math. Mon.*, 45:227--233, 1938.

[20] C. E. Leiserson. *Area-efficient VLSI computation*. PhD thesis, Carnegie-Mellon University, 1982.

[21] W. Neun and H. Melenk. Very large Grobner basis calculations. In Zippel, editor, *Computer algebra and parallelism. Proceedings of the second International Workshop on Parallel Algebraic Computation*, pages 89--100, Ithaca, May 1990. LNCS 584, Springer Verlag.

[22] G. B. Purdy. A carry-free algorithm for finding the greatest common divisor of two integers. *Computers & Mathematics with Applications*, 9:311--316, 1983.

[23] A. Schonhage. Schnelle Berechung von Kettenbruchentwicklugen. *Acta Informatica*, 1:139--144, 1971.

[24] J. Sorenson. Two fast GCD algorithms. Submitted to *J. of Algorithms*, 1993.

[25] J. Stein. Computational problems associated with Racah algebra. *J. Comp. Phys.*, 1:397--405, 1967.

[26] Ken Weber. The accelerated integer GCD algorithm. Technical report, Kent State University, 1993. To appear in ACM Trans. on Math. Software.

[27] D. Y. Y. Yun and C. N. Zhang. A fast carry-free algorithm and hardware design for extended integer GCD computation. In *ACM SYMSAC'86*, pages 82--84. ACM, 1986.