Systolic Normalization of Rational Numbers*

Tudor Jebelean
RISC-LINZ
—
Research Institute for Symbolic Computation
Johannes Kepler University, A4040 Linz, Austria (Europe)
Tel: +43 (7236) 3231-64, -20 Fax: +43 (7236) 3231-30
e-mail: tjebelea@risc.uni-linz.ac.at
—

October 7, 1996

Abstract

We present a systolic algorithm for normalization of rational numbers which is scalable to arbitrary length operands, and we discuss the possibility of implementing it within a rational arithmetic coprocessor for the use of computer algebra systems. A preliminary estimation shows that the performance for 32 bit operands is comparable to that of a fast RISC processor, but for long operands (e. g. 100 words) a significant speed-up (100 times) can be achieved.

Introduction

An emerging field in scientific computation is *computer algebra*, which is concerned with computations with symbolic expressions using arbitrary precision arithmetic. However, a serious drawback of the existing computer algebra systems is the relative slowness of arbitrary precision arithmetic, which is currently implemented in software. For instance, the experiments presented in [Buchberger and Jebelean, 1992] show that in Gröbner Bases computation (needed for solving systems of multivariate polynomial equations), long rational arithmetic consumes 86% up to 99.7% of the computing time, when the length of the input coefficients grows from 5 to 50 decimal digits, while the total computation time grows by a factor of 40. Also [Neun and Melenk, 1990] mention that 80% of the computation time is spent with bignums.

The purpose of our research is the implementation of long rational arithmetic in a "coprocessing" unit, based on systolic parallelization of these operations. We present in this paper a systolic algorithm for normalization of arbitrary length rational numbers, that is, computation of the GCD and of the reduced fraction. This algorithm is based on the "plus-minus" scheme introduced in [Brent and Kung, 1983]. In contrast with their algorithm and others (e. g. [Yun and Zhang, 1986]), the algorithm we present here has several new features:

- A mobile tag marks the sign-bit of the operands during GCD computation. This allows easier treatment of operands having different lengths, as well as easier detection of termination and of the GCD sign.

*Acknowledgements:
Austrian Forschungsförderungsfonds, project S5302-PHY (Parallel Symbolic Computation);
Austrian Ministry for Science and Research, project 613.523/3-27a/89 (Gröbner Bases) and doctoral scholarship;
POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846);

- The algorithm does not use global broadcasting, but only neighbor to neighbor communication. This allows the scaling of the device for arbitrary length operands without increase of the per–step computing time.
- The reduced fraction is computed using a systolic algorithm for *exact division* (introduced in [Jebelean, 1993d]) which is also fully pipelined and has time/space requirements proportional to the length of the quotient.

We discuss the possibility of implementing this algorithm in hardware, by using programmable logic arrays. A preliminary estimation shows that the performance for 32 bit operands will be comparable to that of a fast RISC processor, but for long operands (up to 100 words) a speed-up of one to two orders of magnitude can be achieved (75 times for 50 words, 125 times for 100 words of 32 bits).

The normalization algorithm can be aggregated with multiplication and addition schemes in order to obtain a coprocessor for long integer and long rational arithmetic, which would significantly increase the performance of the computer algebra systems.

1 Systolic GCD computation

Parallelization of the classical Euclidean algorithm for integer GCD computation is difficult because of the carry propagation. Some of the difficulty is removed in the *binary* algorithm introduced in [Stein, 1967]. [Purdy, 1983] also designs a carry–free algorithm, whose worst-case time-complexity is still quadratic. In [Brent and Kung, 1983] and [Brent and Kung, 1985] an improvement of the binary algorithm (the *PlusMinus* algorithm) was introduced and parallelized in a systolic fashion, demonstrating that GCD computation in $O(n)$ steps is possible. [Yun and Zhang, 1986] suggest some further improvements of this last algorithm. The generalizations of the binary algorithm presented in [Jebelean, 1993a] and in [Sorenson, 1993] are also suitable for systolic parallelization, this time at the "word" level (i. e., digits in a high radix).

GCD algorithms on other parallel computing models (CRCW PRAM) were also studied ([Kannan *et al.*, 1987], [Adleman and Kompella, 1988], [Chor and Goldreich, 1990]), which have sublinear complexity. However, we choose here the systolic approach, due to several characteristics of the problem we want to solve and of environment we use:

- The pipelining properties of the systolic algorithms allow efficient aggregation of several processing units. This is important when dealing with rational arithmetic, were several integer operations have to be chained. We use here a variant of Brent & Kung algorithm ([Brent and Kung, 1983]), which works *least-significant digits first* (LSF), that is the same as Atrubins cellular algorithm for *multiplication* ([Atrubin, 1965]), and the same as the systolic *exact division* algorithm from [Jebelean, 1993d], based on [Jebelean, 1993b]. As for *addition* and *subtraction*, these are easy to realize in a LSF pipelined manner.
- In a fast communicating environment (as hardware implementation offers), there is little or no parallelization overhead in the systolic manner, while the overhead introduced by the asymptotically faster methods usually prevents the efficient implementation for small problems.
- The $O(n)$ speed–up given by systolic algorithms already leads to a significant improvement of the overall system. In fact, for multiple precision arithmetic performed on a "coprocessor" this is the best we can hope for, since the time needed to fetch the operands and send back the result to the host is already $O(n)$.
- The non-systolic algorithms lead to $O(n^2)$ area when implemented in hardware, which is not feasible for the range of operands we are interested in (up to 100 words of 32 bits). In contrast, systolic algorithms require only $O(n)$ hardware area.
- For the hardware implementation, the systolic approach offers the possibility of simple and regular design, which leads to a robust solution.

It is useful to give a short description of the *PlusMinus* algorithm and of the main idea for its systolic implementation.

The theoretical basis of the algorithm are the relations:

$$GCD(a,b) = GCD(a, a+b) = GCD(a, a-b).$$

$$\text{If } a \text{ and } b \text{ are } \textit{even}, \text{ then:} \quad GCD(a,b) = 2 * GCD(a/2, b/2).$$

$$\text{If } a \text{ is } \textit{odd} \text{ and } b \text{ is } \textit{even}, \text{ then:} \quad GCD(a,b) = GCD(a, b/2).$$

Suppose now that a_1, a_0 and b_1, b_0 are the least-significant pairs of bits of the operands A and B. Then the following situations are possible:

- Both a_0 and b_0 are null. Then A and B are right-shifted with one binary position, and the new operands are examined again. The number of these initial shifts is stored for the end of the computation, when the corresponding power of 2 will be included in the result.
- Only one of a_0, b_0 is null. Then the even operand is shifted.
- Both a_0 and b_0 are nonzero. Then a_1 and b_1 are inspected. If they are equal, then $C = |A - B|$ is computed, otherwise $C = A + B$, and the process is reiterated with C and the minimum from A, B. Note that C computed in this way is always divisible by 4. Therefore $C/4$ will be at least one bit shorter than $\max(A, B)$. This ensures the termination of the algorithm.

The algorithm terminates when one of the operands becomes zero, and then the other operand will be the "pseudo-GCD", which is to be multiplied with the appropriate power of two in order to give the correct GCD.

The *PlusMinus* algorithm is suitable for systolic parallelization because the main decisions about the procedure are taken using only the least-significant digits of the operands. If each processor contains one pair of digits, then the rightmost processor can make these decisions and can "propagate" the needed actions to the other processors. However, because the carries propagate leftward, there is no need to wait until the full result of an operation is ready, but the next step may start as soon as the least-significant digits of this result are computed.

However, there are some inconveniences in the fact that computation proceeds in this way. The values of the operands cannot be compared, hence subtraction may produce negative results. This means one has to adapt the algorithm for computation with negative numbers, by using complement arithmetic. Also, the result may be negative, hence a way must be found to detect the sign and to handle it in subsequent computations. Finally, the fact that one of the operands becomes zero is less obvious when the computations are pipelined. In the next section we deal with these problems.

2 Adapting systolic PlusMinus algorithm

We present here the modifications of Brent & Kung systolic GCD algorithm which make it suitable for our application. The original algorithm (see [Brent and Kung, 1983]) works by pipelineing the n-digits operands through an array of $3n$ processors. In contrast, we choose a solution in which the operands are kept steady in an array of n processors, and they are shifted at different speeds according to the number of divisions by 2 which have to be performed. This solution is better suited to variable-length operands, because in this way one can ensure that the number of steps is proportional to the variable length of the inputs.

The modified algorithm presented in [Yun and Zhang, 1986] also has "steady" operands, but in turn it uses global broadcasting: at each step, the decision of the rightmost processor (the operation to be performed) is broadcasted to all the other processors, such that each operation is done synchronously using redundant representation, and only the carries are propagated in a pipelined fashion. In contrast, in our algorithm there is no global broadcasting, but only neighbor-to-neighbor communications are used. This increases the number of steps by a factor of two: a "wait" state has to be introduced between each two "active" states, in order to ensure that the correct values are ready at each step. On the other hand, absence of global communications makes the hardware implementation easy scalable to big lengths of the inputs, as it is required by our application. Also, this makes the algorithm suitable for implementation on asynchronous systolic arrays, which might be needed for very long operands in order to eliminate the problem of clock-skew. Note also that the problem of detecting the termination (one operand null) becomes difficult for very long operands, in the case of global communication: a global AND has to be performed at each step.

In order to handle the signs and detect the termination, we introduce two mobile *tags* which indicate the sign-bit of each operand. Each time when the length of one operand is reduced (by subtraction or by shifting), the corresponding tag will move accordingly to the right. When the tag reaches the least-significant bit, then we know the corresponding operand is null.

The tags allow easy adaptation of the algorithm to variable length operands. By setting the tag one bit higher than the most significant bit of each operand, the algorithm will compute only as much as it is needed, even if the hardware implementation is physically made to accommodate longer operands.

Note that the sign of an operand is given by the "tagged" bit (*plus* for *zero*, *minus* for *one*). Therefore, the use of mobile tags also leads to a way of detecting the sign of the resulting GCD: the value of the "tagged" bit is all the time propagated to the right, such that when the computation is ready the information about the sign is available in the rightmost processor. If we would like to have the positive value of the GCD at the end of the computation, we could replicate some of the computations such that the complement of the current GCD is updated all the time. Using the sign information one could than choose the correct value at the end. However, this is not needed for our application, because the next coarse step of the computation (exact division of the two initial operands by the GCD), can be easily adapted two use either of the positive or negative values. Also because of this, it is not needed to keep track of the initial common shifts and to left-shift the pseudo-GCD at the end. Rather, we save the values of the operands after the common shifts and we divide them by this pseudo-GCD.

A more detailed description of the algorithms follows.

$N+1$ processors are used, where $N \geq n$ (the length of the operands). The bits of the operands A and B are held in the processors in an one-to-one fashion: the leftmost processor (denoted P_0) holds the least-significant bits, the n-th processor (P_{n-1}) holds the most significant bits, and the rest hold zeroes. Processor P_0 has a particular behavior: it will take the decisions about the computation and it will send them to the left. The other processors just apply the operations as they are received from the right. The leftmost processor P_N acts similarly to the others, with the only exception that it will not receive values from its left, but instead it will use its own old values. In fact, in this way P_N simulates an infinite chain of processors holding identical values.

Each processor has 8 one-bit registers:

- a and b hold the bits of the operands. These are initialized as described above. At the end of the computation a will hold the bits of the pseudo-GCD and b will be zero.
- ta and tb hold the tags of A and B. Initially the tags up to $n-1$ are zero and the tags from n on are one.
- sa holds the sign of A (zero for $+$, one for $-$), initially null.
- $s = (s_2, s_1, s_0)$ holds the "state" information, i. e. the code of the operation to be performed. There are 8 states:
  - **w**: wait;
  - **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.

The codification into (s_2, s_1, s_0) is done such that s_2 equals the carry/borrow.

Initially all the states are **w**ait.

For describing the operations of the processors, we also use $r_{[+]}$ ($r_{[-]}$) to denote the corresponding values on the left (right) neighboring processor.

The operation of processor $P0$ looks as follows:

```
if  ta[+] = 1 [find sign of A]
    then  sa ← a[+]
    else  sa ← sa[+]
if s ≠ w
    then [wait if previous state was active]
        s ← w
    else [find the appropriate active state]
        if a = 0 and b = 0
            then [shift both A and B]
                s ← B
                a = a[+];    ta ← ta[+]
                b = b[+];    tb ← tb[+]
        if a = 0 and b = 1
            then [interchange A and B and shift B]
                s ← C
                a ← b;       ta = tb
                b ← a[+];    tb ← ta[+]
        if a = 1 and b = 0
            then [shift B]
                s ← S
                b ← b[+];    tb ← tb[+]
        if a = 1 and b = 1
            then [plus or minus]
                if  a[+] = b[+]
                    then  [minus]
                        s ← m  ;   b ← 0
                    else  [plus]
                        s ← P  ;   b ← 0
```

The rest of the processors operate like this:

```
if  ta[+] = 1 [find sign of A]
    then  sa ← a[+]
    else  sa ← sa[+]
s ← s[-] [state is propagated leftward]
switch s[-] [right state is considered]
    case w  : [no modifications]
```

case B : [shift both A and B]
$a = a_{[+]}$; $ta \leftarrow ta_{[+]}$
$b = b_{[+]}$; $tb \leftarrow tb_{[+]}$
case C : [interchange A and B and shift B]
$a \leftarrow b$; $ta = tb$
$b \leftarrow a_{[+]}$; $tb \leftarrow ta_{[+]}$
case S : [shift B]
$b \leftarrow b_{[+]}$; $tb \leftarrow tb_{[+]}$
case P , p : [plus]
$a \leftarrow b$; $ta \leftarrow tb$ [set a to old b]
$(s_2, b) \leftarrow a_{[+]} + b_{[+]} + s_2$ [set b to shifted sum]
$tb \leftarrow (ta \ \texttt{AND} \ tb)$ [tag the minimal correct position]
case M , m : [plus]
$a \leftarrow b$; $ta \leftarrow tb$ [set a to old b]
$(s_2, b) \leftarrow a_{[+]} - b_{[+]} - s_2$ [set b to shifted difference]
$tb \leftarrow (ta \ \texttt{AND} \ tb)$ [tag the minimal correct position]

The *plus-minus* operations require some explanation: A gets the old value of B, while B gets the sum/difference. This ensures that the pseudo-GCD will be obtained in A, because only B may become null. By using $a_{[+]}, b_{[+]}$ in the right-hand side of the expressions, the sum/difference is already shifted rightward. By setting s_2 to the carry/borrow, the new state will be modified to the correct value among **P** , **p** (**M** , **m**). Using logical AND for computing the tag of the result, this tag will be set on the minimal position which was previously tagged in both operands.

The algorithm terminates when in P_0 $b = 0$ and $tb = 1$. At that moment A contains the pseudo GCD G and sa of P_0 gives the sign of G. Indeed, suppose G is m bits long. The last operation has to be a subtraction, because B becomes null, hence B was holding G before this last subtraction, having the tag on position (at least) m. The tag of the null B begins to propagate rightward only after the subtraction signal reaches this position m. Note that at that moment all the correct bits of A (old B) are already in place. Since the rightmost m bits of B are zero, only operation **S** is issued, hence A is not modified. From position m, the tag of B will need $2m$ steps to reach P_0 and trigger termination. During this time, the sign of A too has time to reach P_0.

The worst-case running time (number of steps) of this algorithm is two times higher than the worse-case running time of the original *PlusMinus* algorithm (about 3.1 n according to [Brent and Kung, 1985]), hence it equals (roughly) 6 n. However, from a practical point of view it is the average time which is interesting. According to our experiments, the average number of steps for 1000 random pairs of 32 bits is 135 steps, hence about 4 n, and the average for pairs having GCD of 16 bits is 91 steps, hence about 3 n.

In order to continue the computation (find the normalized fraction), the values of the operands after the initial common right-shifts are needed. This is satisfied by saving a, b into two separate string of registers, each time the new state is **B** . In this way, the save-registers will finally contain the values of A, B after the last **B** operation.

3 Exact division

By *exact division* we understand the division without remainder, i. e. when it is known in advance that the remainder is null. [Jebelean, 1993b] presents a new algorithm for exact division in high-radix, which speeds-up this operation by taking advantage of this knowledge. Also, this algorithm is suitable for systolic parallelization, because it works "right-to-left". A systolic exact division for binary-radix is contained in the division-with-remainder algorithm presented in [Purdy and Purdy, 1987], but this one uses global broadcasting. In [Jebelean, 1993d] we presented several LSF systolic algorithms for exact division, in high-radix, some of them without global broadcasting. We will use here one of these algorithms, particularized for the binary-radix case and slightly modified in order to fit to the present problem. This algorithm accepts the inputs into the arrays, as they are provided by the previous pseudo-GCD algorithm, and produces the result (exact quotient) in a pipelined fashion, LSF, in the rightmost processor. The number of steps equals the length of the quotient.

We present here the outline of the algorithm. The basic idea of *exact division* is the following: Let A, G be such that $G \mid A$ and G is odd. We want to find the exact quotient $X = A/G$. Let a_0, g_0, x_0 be the least-significant binary digits of A, G, X. Then from $a_0 = x_0 * g_0$ and $g_0 = 1$ one gets $x_0 = a_0$.

Now if we denote by X' the shifted X:

$$X = x_0 + 2 * X',$$

then one has:

$$(A - G * x_0)/2 = G * X',$$

hence the next digit of X can be computed by applying the same scheme to shifted $(A - G * x_0)$ and G.

The systolic algorithm (see Fig.1) starts with the digits of A and X in the processors $P_0, \ldots, P_m$, where $m+1$ is the length of the quotient. Processor P_0 has a distinct behavior: at step 0 it will output $x_0 = a_0$, and at step $k > 0$ it will output $x_k = a_{[+]} - g_1 * x_{k-1}$, where $a_{[+]}$ is the (updated) bit of A supplied by P_1 and g_1 is the one-before least-significant digit of G. The borrow obtained at each step is subtracted from x_k at the next step.

The other processors pipe A and G rightward, and the bits x_k of the result X leftward. Each bit g_k of G is moved until it reaches $P_{k/2}$, where it is stored (hence two bits of G are stored in each cell). Each bit a_k of A is updated by one or two subtractions when it meets the appropriate bits of X. Again the borrows resulting from subtraction are incorporated in the result at the next step.

For more details concerning this algorithm the reader is referred to [Jebelean, 1993d].

We present below the details of systolic exact division algorithm adapted to our problem: computation of the quotients $X = A/G$ and $Y = B/G$.

First, we eliminate step 0 of the original division algorithm. The values x_0 and y_0 will be output during the last step of the GCD computation, from the least-significant save-registers containing the shifted values of A and B. During this operation, the saved values of A and B will be transferred into the registers of the division array, shifted rightward with one position. The division will now start directly with step 1.

Another modification concerns the sign of G. The value of the sign $sg \leftarrow sa$ from P_0 of the GCD array is pipelined leftward in the division array and selects subtraction (addition) if the sign is positive (negative). In this way, the results will be correct even if G supplied by the GCD array is negative.

The following registers are used by each processor:

- a, b: dividends;
- g, g': divisor;
- sg: sign of divisor;
- x, y: results;
- $u = (u_1, u_0), v = (v_1, v_0)$: carries/borrows.

The operation of processor P_0 is as follows:

$(u_0, x) \leftarrow a_{[+]} \pm g * x - u_0$
$(v_0, y) \leftarrow b_{[+]} \pm g * x - v_0$
$\texttt{OUTPUT}(x, y)$

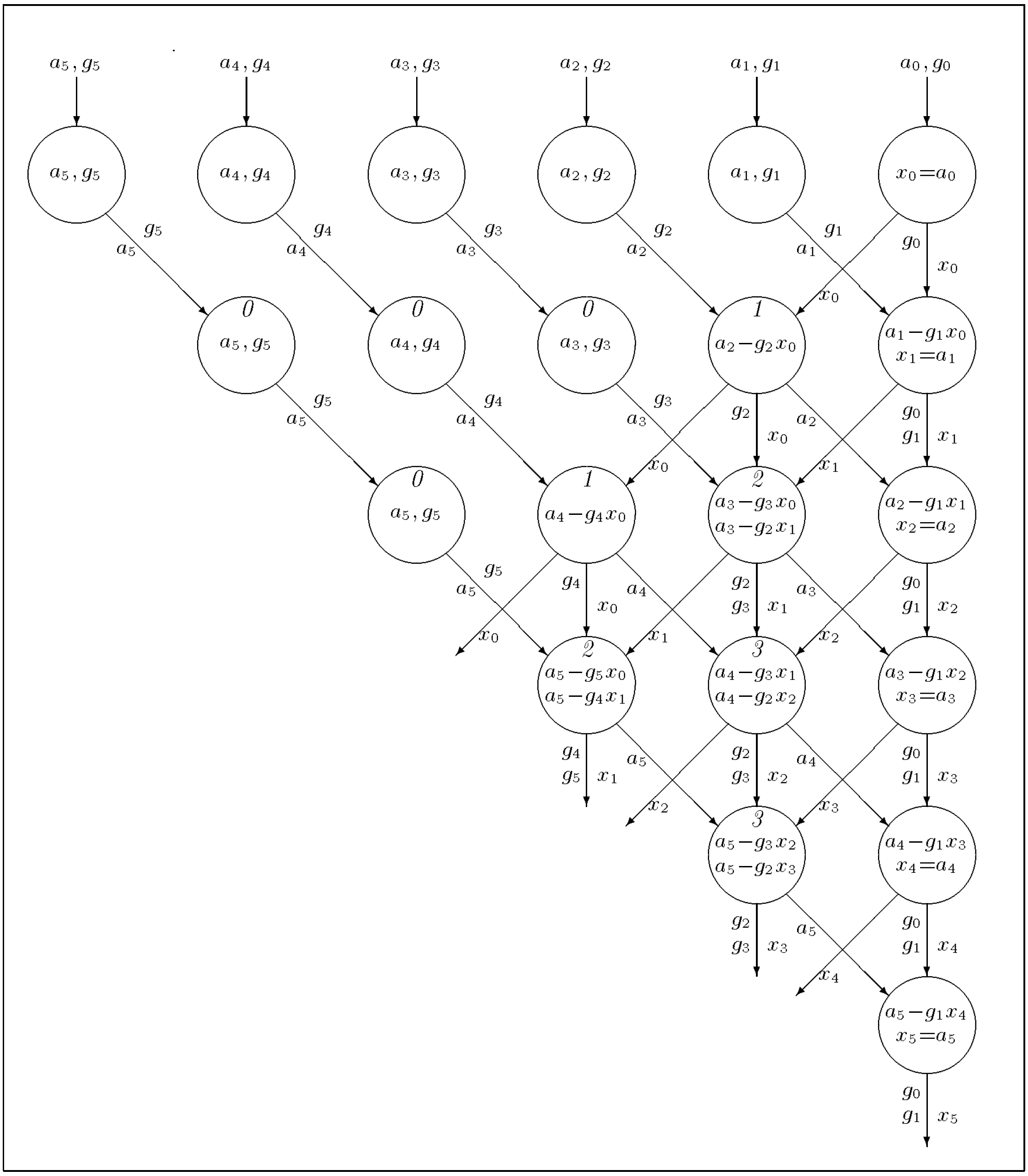

Figure 1: Data flow in systolic exact divison (borrows not shown).

The operation of processor P_1 differs from the others only in the way states are handled: its initial state is 1, the next step it becomes 2, and for the subsequent steps it will be 3 (unchanged).

The other processors start in state 0 and operate like this:

switch s [four operation modes]
  case 0: [only shift]
    $g \leftarrow g_{[+]}\,;\, a \leftarrow a_{[+]}\,;\, b \leftarrow b_{[+]}$
  case 1: [store one bit of G and add/subtract]
    $g \leftarrow g_{[+]}\,;\, x \leftarrow x_{[-]}\,;\, y \leftarrow y_{[-]}$
    $(u_0, a) \leftarrow a_{[+]} \pm g_{[+]} x_{[-]}$
    $(v_0, b) \leftarrow b_{[+]} \pm g_{[+]} y_{[-]}$
  case 2: [store another bit of G and add/subtract]
    $g' \leftarrow g_{[+]}\,;\, x \leftarrow x_{[-]}\,;\, y \leftarrow y_{[-]}$
    $(u_1, u_0, a) \leftarrow a_{[+]} \pm g_{[+]} x \pm g\ x_{[-]} - u_0$
    $(v_1, v_0, b) \leftarrow b_{[+]} \pm g_{[+]} y \pm g\ y_{[-]} - v_0$
  case 3: [add/subtract]
    $x \leftarrow x_{[-]}\,;\, y \leftarrow y_{[-]}$
    $(u_1, u_0, a) \leftarrow a_{[+]} \pm g'\ x \pm g\ x_{[-]} - (u_1, u_0)$
    $(v_1, v_0, b) \leftarrow b_{[+]} \pm g'\ y \pm g\ y_{[-]} - (v_1, v_0)$
$s \leftarrow s_{[-]}$ [state is propagated leftward]
$sg \leftarrow sg_{[-]}$ [and also the sign of G]

The termination detection must be handled separately. In order to signal termination after a number of steps equal to the length of the quotient, a counter CN will be used which is operated in the following way:

- Initially CN is set by the external environment to the bit-length of the biggest input.
- During GCD computation, the counter is decremented at each **B** ("shift both") step.
- At first "active" (non **wait**) step which is not **B**, the value of the counter is saved in an auxiliary set of registers. Note that this value equals the length of the shifted A and B.
- Subsequently, the counter is decremented at each **S** step, but is reset to the saved value at each non-**S** active step. Note that, at the end of GCD computation, the counter will be decremented with the number of consecutive **S** steps, which is the length of G. Hence, the counter is set now to the length of the quotient, or inferior by one to it.
- During the exact division, the counter is decremented at each step, and the termination is triggered when the counter reaches zero. Because step 0 of the exact division is done simultaneously with the last step of GCD computation, the ending time is correctly detected.

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)
- 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 Concurrent Logic, Inc. (CLi), which uses CAD software from Viewlogic.

The circuit implementing the transition function needs 32 elementary gates and has a depth of 6 gates. The delay (using CLi-like components) is 35 ns. The implementation of a full cell (including registers) uses 47 gates and registers (47 Cli cells – without routing cells). The GCD array of cells can be operated at 20 MHz. The division array would be probably operational at a similar speed, since the most complicated function (double addition/subtraction in state 3 can be implemented with a depth of 7 gates.

Hence, an array for computing the pseudo-GCD of 32 bits operands will require about 1,500 gates and registers, and will need an average of about $135 * 50ns = 0.007ms$. 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 pseudo-GCD of 50-word (of 32 bits) operands in about $4.2 * 50 * 32 * 50ns = 0.35ms$, while this takes $26ms$ on a DECstation 5000/200 using the Lehmer-Euclid algorithm (see [Jebelean, 1993c] for timings of sequential GCD computation). Hence, a speed-up of 75 times could be achieved. For 100 words operands, one would get a speed-up of $87/0.7 = 125$.

These, of course, are semi-theoretical projections. One would need to experiment with a real system in order to get the real figures, and one of the difficult problems which remains to be solved is the connection and synchronization of the coprocessor with the host. However, similar applications of FPGA devices for long integer multiplication in cryptography (see e. g. [Bertin *et al.*, 1989], [Shand *et al.*, 1990]) which were successful show that this is feasible.

References

[Adleman and Kompella, 1988] L. M. Adleman and K. Kompella. Using smoothness to achieve parallelism. In *20th Annual ACM Symposium on Theory of Computing*, pages 528–538, 1988.

[Atrubin, 1965] A. J. Atrubin. A one–dimensional iterative multiplier. *IEEE Trans. on Computers*, C-14:394–399, 1965.

[Bertin *et al.*, 1989] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In McCanny, McWhirter, and Swartzlander, editors, *Systolic array processors*. Prentice Hall, 1989.

[Brent and Kung, 1983] R. P. Brent and H. T. Kung. Systolic VLSI arrays for linear-time GCD computation. In Anceau and Aas, editors, *VLSI'83*, pages 145–154. Elsevier (North-Holland), 1983.

[Brent and Kung, 1985] 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.

[Buchberger and Jebelean, 1992] 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.

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

[Jebelean, 1993a] T. Jebelean. A Generalization of the Binary GCD Algorithm. In *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, Kiev, Ukraine, July 1993.

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

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

[Jebelean, 1993d] T. Jebelean. Systolic Algorithms for Exact Division. In *Workshop on Fine Grain and Massive Parallelism*, Dresden, Germany, April 1993.

[Kannan *et al.*, 1987] 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.

[Neun and Melenk, 1990] W. Neun and H. Melenk. Very large Gröbner 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.

[Purdy and Purdy, 1987] C. N. Purdy and G. B. Purdy. Integer division in linear time with bounded fan–in. *IEEE Trans. on Computers*, C-36(5):640–644, 1987.

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

[Shand *et al.*, 1990] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. Technical Report 1, Digital Research Laboratory, Paris, 1990.

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

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

[Yun and Zhang, 1986] 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.