FPGA Implementation of an Extended Binary GCD Algorithm for Systolic Reduction of Rational Numbers

Bogdan Mătăsaru and Tudor Jebelean

RISC-Linz, A–4040 Linz, Austria
email: bmatasar@risc.uni-linz.ac.at

Abstract. We present the FPGA implementation of an extension of the binary *plus–minus* systolic algorithm which computes the GCD (greatest common divisor) and also the normal form of a rational number, without using division. A sample array for 8 bit operands consumes 83.4% of an Atmel 40K10 chip and operates at 25 MHz.

1 Introduction

Arbitrary precision (or exact) arithmetic is necessary in various computer algebra applications (e. g. solving of systems of polynomial equations) and it may consume most of the computing time [2].

Reduction of rational numbers occurs very often in exact arithmetic: sooner or later, the reduction of the result is required in order to avoid an unacceptable increase in size of its numerator and denominator. The usual approach is to use the GCD (greatest common divisor) once (or twice on shorter operands [3]) and some divisions (or better exact divisions [4, 8]). Alternatively, one can use the extended GCD algorithm [7], but this will probably be less efficient for sequential implementations.

In this paper we present the systolic implementation of an extension of the binary GCD algorithm [10], more precisely of the plus-minus algorithm [1] as improved in [5]. Extensions of the original binary algorithm have been considered by Gosper [7], and also in [11]. Our algorithm has a different flavor [9] because it is designed for systolic implementation.

Since it avoids the division, and it is also based on simple operations like shifts and additions/subtractions, our device may be interesting even for classical implementations on some sequential architectures. In this paper we demonstrate the usefulness of this approach on a systolic architecture, by designing a systolic version of the algorithm and by implementing it on an Atmel FPGA (field programmable gate array) circuit. The systolic algorithm is an extension of the systolic GCD presented in [5] which avoids global broadcasting by pipelining the "command" through the array. This in turn has the disadvantage that *wait* states have to be introduced, because of the two-way flow of the information. Our algorithm takes advantage of these *wait* states: the computations required by the

extended algorithm are done instead of wait. Hence, the speed is the same as the one of the previous algorithm, but additionally we obtain the reduced fraction.

The circuit is presented as a systolic array, enjoying the properties of uniform design (many identical cells) and local communications (only between adjacent cells). The input of the operands is done in parallel manner, while the output can be organized serially or in parallel, depending on the application.

The sample implementation of an array for 8 bit operands on the Atmel FPGA part 40K10 consists of 968 elementary macros and consumes after layout 83.4% of the area. The longest delay is 40 ns, thus operation at 25 MHz is possible, which rivals the speed of the best software implementations on sequential machines for words of 32 bits, but it will probably gain considerably in speed in a special device for more words.

2 The algorithm

The input of the binary (plus-minus) GCD algorithm consists of 2 n-bit operands a and b. The operations executed on the operands are additions, subtractions and shifts and they are decided by inspecting the least significant two bits of each operand.

Let us denote by a_k, respectively b_k, the values of the operands at step k. The extended GCD algorithm computes also the sequences of cofactors of u_k, v_k, t_k, and w_k such that at each step k:

$$u_k \cdot a + v_k \cdot b = a_k \cdot 2^k, \tag{1}$$

$$t_k \cdot a + w_k \cdot b = b_k \cdot 2^k. \tag{2}$$

The algorithm ends when $b_k = 0$. Then, a_k equals the GCD and $t_k \cdot a + w_k \cdot b = 0$, hence $a/b = -w_k/t_k$.

The sequence of cofactors is defined recursively starting from the initial values: $u_0 = 1, v_0 = 0, t_0 = 0, w_0 = 1$. (We denote by $x[1], x[0]$ the least significant bits of an operand x.)

Shift both: ($a_k[0] = 0$, $b_k[0] = 0$): the cofactors remain unchanged.
Interchange and shift b: ($a_k[0] = 0$, $b_k[0] = 1$):

$$u_{k+1} := 2 \cdot t_k \ , \ v_{k+1} := 2 \cdot w_k,$$
$$t_{k+1} := u_k \ , \ w_{k+1:} = v_k.$$

Shift b: ($a_k[0] = 1$, $b_k[0] = 0$):

$$u_{k+1} := 2 \cdot u_k \ , \ v_{k+1} := 2 \cdot v_k,$$
$$t_{k+1} := t_k \ , \ w_{k+1} := w_k.$$

Plus-minus: ($a_k[0] = 1$, $b_k[0] = 1$), **plus** if $a_k[1] \neq b_k[1]$, otherwise **minus**:

$$u_{k+1} := 2 \cdot t_k \ , \ v_{k+1} := 2 \cdot w_k$$
$$t_{k+1} := u \pm t_k \ , \ w_{k+1} := v_k \pm w_k$$

One can easily verify that the relations (1) and (2) are preserved at each step, and one can prove that t_k and w_k remain relatively prime and at least one of them is not null. Thus, when b_k becomes null, $-w_k/t_k$ is the reduced form of the initial fraction a/b. (For the proofs and a more detailed description of the algorithm see [9].)

3 The systolic array

The systolic array is organized as in [5]: the operands are fed in parallel, one digit of each in each processor, the rightmost processor (P_0) corresponds to the least-significant bit – see Fig. 1. An array of $N+1$ processors will accommodate operands up to N bits long. (The leftmost processor and the ones above the significant bits of the operands will contain the sign bit). All the intermediate values are kept in complement representation: therefore additions/subtractions can be performed without knowing the actual sign of the operands.

Each processor communicates only with its neighbors: the commands (states) and the carries propagate right-to-left, while the intermediate values of the operands propagate left-to-right. All the processors except P_0 are identical. P_0 computes at each step a command code (depending on $a[1], a[0], b[1], b[0]$) which is then propagated to the other processors and controls their operations.

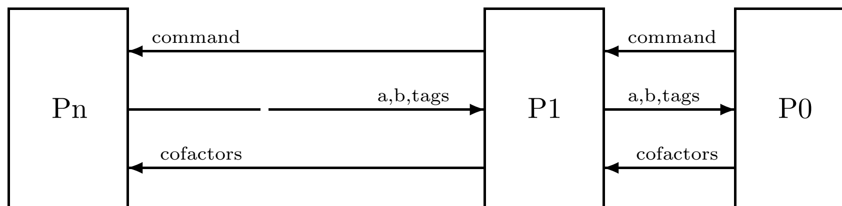

Fig. 1. The systolic array for the extended binary algorithm.

Each processor has a memory of 16 one-bit registers. 8 registers are necessary for the *plus-minus* GCD array and they have the same meaning as in [5]: s_1, s_2, s_3 contain the command code (or *state*), a, b are bits of the operands, ta, tb (*tags*) indicate their significant bits, and sa stores the sign of a.

Additionally, 8 registers are used by the extended algorithm: u, v, t, w keep the values of the cofactors, ct and cw are carries needed for the plus-minus operations, and u', v' keep intermediate values needed for left-shifts.

In the next tables we describe the operation of the processors. For a value x on a processor, $x_{[+]}$ and $x_{[-]}$ will denote the value x on the left, respectively right neighboring processor. (The rightmost processor receives copies its own values as [+] values.) By $< c, r >:=$*expr*, we specify that, after the evaluation of the expression *expr*, r will contain the result restricted to the register's size and c will contain the carry.

The operation of the processor P_0:

Algorithm P_0

begin

  if $ta_{[+]} = 1$ then $sa := a_{[+]}$; [find sign of A]

  else $sa := sa_{[+]}$;

  if $s \neq \mathbf{w}$ then [update the cofactors on the wait step]

    switch s

      case **C:**

        $(u, t, u') := (0, u, t)$**;**

        $(v, w, v') := (0, v, w)$**;**

      case **S:**

        $(u, u') := (0, u)$**;**

        $(v, v') := (0, v)$**;**

      case **P,p:**

        $(u, u', < ct, t >) := (0, t, u + t)$**;**

        $(v, v', < cw, w >) := (0, w, v + w)$**;**

      case **M,m:**

        $(u, u', < ct, t >) := (0, t, u - t)$**;**

        $(v, v', < cw, w >) := (0, w, v - w)$**;**

    $s := \mathbf{w}$;

  else [find the appropriate active state]

    if $a = 0 \wedge b = 0$ then [shift both A and B]

      $s := \mathbf{B}$;

      $(a, b, ta, tb) := (a_{[+]}, b_{[+]}, ta_{[+]}, tb_{[+]})$;

    if $a = 0 \wedge b = 1$ then [interchange A and B and shift B]

      $s := \mathbf{C}$;

      $(a, b, ta, tb) := (b, a_{[+]}, tb, ta_{[+]})$;

    if $a = 1 \wedge b = 0$ then [shift B]

      $s := \mathbf{S}$;

      $(b, tb) := (b_{[+]}, tb_{[+]})$;

    if $a = 1 \wedge b = 1$ then [plus or minus]

      if $a_{[+]} = b_{[+]}$ then $s := \mathbf{m}$; [minus]

      else $s := \mathbf{P}$; [plus]

      $b := 0$;

end

The rest of the processors ($P_1, \ldots, P_N$) operate like this:

```
Algorithm P_i
begin
    if ta[+] = 1 then sa := a[+];                                [find sign of A]
    else sa := sa[+];
    switch s[-]                                            [right state considered]
        case w:                              [update the cofactors on the wait step]
            switch s
                case C:
                    (u, t, u') := (u'[-], u, t);
                    (v, w, v') := (v'[-], v, w);
                case S:
                    (u, u') := (u'[-], u);
                    (v, v') := (v'[-], v);
                case P,p:
                    (u, u', < ct, t >) := (u'[-], t, u + t + ct[-]);
                    (v, v', < cw, w >) := (v'[-], w, v + w + cw[-]);
                case M,m:
                    (u, u', < ct, t >) := (u'[-], t, u - t - ct[-]);
                    (v, v', < cw, w >) := (v'[-], w, v - w - cw[-]);
        case B:                                              [shift both A and B]
            s := s[-];                                  [state is propagated leftwards]
            (a, b, ta, tb) := (a[+], b[+], ta[+], tb[+]);
        case C:                                     [interchange A and B and shift B]
            s := s[-];                                  [state is propagated leftwards]
            (a, b, ta, tb) := (b, a[+], tb, ta[+]);
        case S:                                                         [shift B]
            s := s[-];                                  [state is propagated leftwards]
            (b, tb) := (b[+], tb[+]);
        case P,p:                                                          [plus]
            s := s[-];                                  [state is propagated leftwards]
            (a, ta) := (b, tb);                                       [set a to old b]
            < s_2, b >:= a[+] + b[+] + s_2;                      [set b to shifted sum]
            tb := (ta ∧ tb)                          [tag the minimal correct position]
        case M,m:                                                         [minus]
            (a, ta) := (b, tb);                                       [set a to old b]
            < s_2, b >:= a[+] - b[+] - s_2;               [set b to shifted difference]
            tb := (ta ∧ tb)                          [tag the minimal correct position]
end
```

The **w**ait state was introduced in the systolic GCD algorithm in order to eliminate the global broadcasting of the operation code. In this state the pro-

cessor was not used. In the extended algorithm we replace this wait time by the computation of the cofactors, therefore increasing the efficiency of the circuit.

Note that the data used to build the cofactors flows from right to left, either as a carry in the plus/minus steps, or by shifting in the shift steps. That is, the partial value of a cofactor kept in the registers of the processor P_k depends only on the values kept on the processors $P_0, \ldots, P_k$. Moreover, the cofactors are less or equal to the correspondent input operands, so their length is less or equal to n. Therefore, the number of processors needed for the GCD computation is sufficient also for obtaining the reduced form of the rational number.

The termination of the systolic GCD algorithm is detected by P_0 when the value of b is 0 and the tag of b is 1. Sometimes, the extended algorithm requires several additional steps to finish the propagation of the "command" to the most significant bits of the cofactors. However, this is not a problem if the result is retrieved in a LSF way because after b_k becomes 0 the circuit pipelines only **B** operations which do not change the values of the cofactors t_k and w_k.

4 FPGA Experiments

We implemented the array on an ATMEL FPGA using the Atmel IDS environment from Cadence Systems Inc. and Workview Office from Viewlogic Systems Inc. This represents an extension of the GCD implementation reported in in [6].

For a circuit containing 9 processors (accomodates operands up to 8 bits), the `netlist` phase reports a number of 968 elementary blocks, which is smaller than the area used by the GCD together with division presented in [6]. Some of the intermediate values between the GCD algorithms and the two exact divisions are not needed anymore; this simplifies the function that updates the operands. The function that computes the cofactors requires 31 elementary gates per processor. Thus, the circuit for computing the extended gcd is also simpler than a circuit which computes the gcd followed by two exact divisions.

The longest delay path passes through 13 ports and it is determined by the original GCD array - the computation of the cofactors does not increase the computing time.

The automatic layout was successful on a Atmel AT40K10 chip, and consumes 384 logical cells (83.4% of the total). The longest delay through our the circuit is 40 ns – corresponding to a speed of 25 MHz. This gives an estimated time of 0.005 ms for computing the reduced fraction of 32 bit operands, which rivals the best current sequential processor (on an Ultra SPARC 60 machine with the GMP library we have got an average time of 0.006 ms). However, a special device constructed on the bases of this implementation for longer operands (e. g. 4 to 10 words) will gain in speed by a considerable factor, because the computing time of the systolic array increases linearly with the length of the operands, while the sequential algorithms have quadratic complexity.

5 Conclusions and further work

We demonstrated the usefulness of the extended plus-minus GCD algorithm for the reduction of rational fractions by an FPGA implementation. Because it avoids the divisions altogether, this approach leads to a smaller implementation (as number of gates/cells), while keeping the speed of the original algorithm. Based on this sample implementation one can easily construct a device for rational arithmetic for long operands, because the circuit is uniform and can be extended by simple tiling.

Further possible improvements of this implementation include: addition of a bus for pipelining the input operands (this would allow pipelined preprocessing for steps **shift both** and **interchange** and will lead to significant simplification of the processors in the array); as well as addition of a bus for pipelining of the output operands.

References

1. R. P. Brent and H. T. Kung, *A systolic algorithm for integer GCD computation*, Procs. of the 7th Symp. on Computer Arithmetic (K. Hwang, ed.), IEEE Computer Society, June 1985, pp. 118–125.
2. B. Buchberger, *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory*, Recent trends in Multidimensional Systems (Dordrecht-Boston-Lancaster) (Bose and Reidel, eds.), D. Reidel Publishing Company, 1985, pp. 184–232.
3. P. Henrici, *A subroutine for computations with rational numbers*, Journal of the ACM **3** (1956), 6–9.
4. T. Jebelean, *An algorithm for exact division*, Journal of Symbolic Computation **15** (1993), no. 2, 169–180.
5. ———, *Systolic normalization of rational numbers*, ASAP'93: International Conference on Application–Specific Array Processors (Venice, Italy) (L. Dadda and B. Wah, eds.), IEEE Computer Society Press, October 1993, pp. 502–513.
6. ———, *Rational arithmetic using FPGA*, More FPGAs (W. Luk and W. Moore, eds.), Abingdon EE&CS Books, Oxford, 1994, Proceedings of FPLA'93: International Workshop on Field Programmable Logic and Applications, Oxford, UK, September 1993, pp. 262–273.
7. D. E. Knuth, *The art of computer programming*, vol. 2, Addison-Wesley, 1981.
8. W. Krandick and T. Jebelean, *Bidirectional exact integer division*, Journal of Symbolic Computation **21** (1996), 441–455.
9. B. Matasar, *An extension of the binary GCD algorithm for systolic parallelization*, Tech. Report 99–47, RISC–Linz, 1999, `http://www.risc.uni-linz.ac.at/library`.
10. J. Stein, *Computational problems associated with Racah algebra*, J. Comp. Phys. **1** (1967), 397–405.
11. K. Weber, *The accelerated integer GCD algorithm*, ACM Trans. on Math. Software **21** (1995), no. 1, 111–122.