An Extension of the Binary GCD Algorithm for Systolic Parallelization

Bogdan Mătăsaru

RISC-Linz, A–4040 Linz, Austria

Abstract. We present a new algorithm that extends the binary algorithm, and computes the GCD (greatest common divisor) and also the normal form of a rational number, without using division. We present also a systolic variant of this algorithm that works in a least-significant-first manner and without global broadcasting.

1 Introduction

The problem of reducing rational numbers occurs very often in exact rational 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.

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 [2]) and some divisions (or better exact divisions [3, 6]). Alternatively, one can use the extended GCD algorithm [5], but this will probably be less efficient for sequential implementations.

In this paper we introduce an extension to the binary GCD algorithm [7], more precisely of the plus-minus algorithm [1] as improved in [4]. The binary algorithm computes 2 sequences of operands with decreasing sizes. When one of the operands becomes 0, the other will be the GCD (without the powers of 2). We build also the sequences of cofactors corresponding to the operands. When the algorithms ends, the cofactors of the null operand give us the reduced fraction. Extensions of the original binary algorithm have been considered by Penk [5], and also in [8]. Although these algorithms compute the proper cofactors for the GCD of the operands, they are unsuitable for our goal. For example, in the algorithm described by Penk in the end the cofactors can be both 0. Due to its simplicity, our algorithm can be also very easily modified for a systolic approach. Tests made on single precision operands show that this algorithm is faster than the combination Euclidean GCD/integer division on machines were the integer division is much slower than the addition (see Fig 1). However, a real improvement is gained in the systolic version.

	Euclid + 2 divs	Binary + 2 divs	Extended binary
Solaris/SUN Sparc	0.00300 ms	0.00150 ms	0.00120 ms
Irix 6.5/SGI Octane	0.00180 ms	0.00130 ms	0.00130 ms
Linux/PC	0.00220 ms	0.00180 ms	0.00220 ms
Ultrix/DEC	0.02875 ms	0.01504 ms	0.01914 ms

Fig. 1. Simple precision GCD

2 The algorithm

The input of the binary (plus-minus) 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. We denote by $x[1], x[0]$ the least significant bits of an operand x. The extended binary algorithm computes also the sequences of cofactors u_k, v_k, t_k, and w_k such that at each step k:

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

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

Algorithm **Extended Binary**
Input: $a \in \mathbb{Z}, b \in \mathbb{Z}$;
Output: $a' \in \mathbb{Z}, b' \in \mathbb{Z}, a'/b' = a/b, \gcd(a', b') = 1$;
`begin`
  $k := 0; a_0 := a; b_0 := b$;
  $u_0 := 1; v_0 := 0; t_0 := 0; w_0 := 1$;
  `while` $b_k \neq 0$ `do`
    `if` $a_k[0] = 0 \wedge b_k[0] = 0$ `then` [shift both]
      $a_{k+1} := \frac{a_k}{2}; b_{k+1} := \frac{b_k}{2}$;
      $u_{k+1} := u_k; v_{k+1} := v_k$;
      $t_{k+1} := t_k; w_{k+1} := w_k$;
    `if` $a_k[0] = 0 \wedge b_k[0] = 1$ `then` [interchange and shift b]
      $a_{k+1} := b_k; b_{k+1} := \frac{a_k}{2}$;
      $u_{k+1} := 2 \cdot t_k; v_{k+1} := 2 \cdot w_k$;
      $t_{k+1} := u_k; w_{k+1} := v_k$;
    `if` $a_k[0] = 1 \wedge b_k[0] = 0$ `then` [shift b]
      $b_{k+1} := \frac{b_k}{2}$;
      $u_{k+1} := 2 \cdot u_k; v_{k+1} := 2 \cdot v_k$;
      $t_{k+1} := t_k; w_{k+1} := w_k$;
    `if` $a_k[0] = 1 \wedge b_k[0] = 1$ `then`
      `if` $a_k[1] = b_k[1]$ `then` [minus]

```
            a_{k+1} := b_k; b_{k+1} := (a_k - b_k)/2;
            u_{k+1} := 2 · t_k; v_{k+1} := 2 · w_k;
            t_{k+1} := u_k - t_k; w_{k+1} := v_k - w_k;
        else                                                    [plus]
            a_{k+1} := b_k; b_{k+1} := (a_k + b_k)/2;
            u_{k+1} := 2 · t_k; v_{k+1} := 2 · w_k;
            t_{k+1} := u_k + t_k; w_{k+1} := v_k + w_k;
    k := k + 1;
  a' := -w_k; b' := t_k;
end
```

The algorithm ends when $b_k = 0$. Then, $t_k \cdot a + w_k \cdot b = 0$. Later we will prove that w_k and t_k are relatively primes, hence $a/b = -w_k/t_k$.

The sequence of cofactors is defined recursively starting from the initial values:

$$u_0 = 1, v_0 = 0, t_0 = 0, w_0 = 1.$$

Thus, we preserve the relations (1),(2) for $k = 0$.

In order to analyze the transition from the step k to the step $k+1$, let us treat the cases independently:

shift both: $(a_k[0] = 0,\ b_k[0] = 0)$:

the cofactors remain unchanged.

$u_{k+1} \cdot a + v_{k+1} \cdot b = u_k \cdot a + v_k \cdot b = a_k \cdot 2^k = (2 \cdot a_{k+1}) \cdot 2^k = a_{k+1} \cdot 2^{k+1}$

and

$t_{k+1} \cdot a + w_{k+1} \cdot b = t_k \cdot a + w_k \cdot b = b_k \cdot 2^k = (2 \cdot b_{k+1}) \cdot 2^k = b_{k+1} \cdot 2^{k+1}$.

interchange and shift b: $(a_k[0] = 0,\ b_k[0] = 1)$:

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

Since $a_{k+1} = b_k$ and $b_{k+1} = \frac{a_k}{2}$, we have

$u_{k+1} \cdot a + v_{k+1} \cdot b = 2 \cdot t_k \cdot a + 2 \cdot w_k \cdot b = 2 \cdot (t_k \cdot a + w_k \cdot b) = 2 \cdot (2^k \cdot b_k) = 2^{k+1} \cdot b_k = 2^{k+1} \cdot a_{k+1}$

and

$t_{k+1} \cdot a + w_{k+1} \cdot b = u_k \cdot a + v_k \cdot b = 2^k \cdot a_k = 2^k \cdot (2 \cdot b_{k+1}) = 2^{k+1} \cdot b_{k+1}$.

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

The transition is given by

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

Taking into account the fact that $a_{k+1} = a_k$ and $b_{k+1} = \frac{b_k}{2}$, we have

$u_{k+1} \cdot a + v_{k+1} \cdot b = 2 \cdot u_k \cdot a + 2 \cdot v_k \cdot b = 2 \cdot (u_k \cdot a + v_k \cdot b) = 2 \cdot (2^k \cdot a_k) = 2^{k+1} \cdot a_k = 2^{k+1} \cdot a_{k+1}$

and
$t_{k+1} \cdot a + w_{k+1} \cdot b = u_k \cdot a + v_k \cdot b = 2^k \cdot b_k = 2^k \cdot (2 \cdot b_{k+1}) = 2^{k+1} \cdot b_{k+1}$.

plus/minus: $(a_k[0] = 1,\, b_k[0] = 1)$:

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

Again, taking into account the fact that $a_{k+1} = b_k$ and $b_{k+1} = \frac{a_k \pm b_k}{2}$, we have
$u_{k+1} \cdot a + v_{k+1} \cdot b = 2 \cdot t_k \cdot a + 2 \cdot w_k \cdot b = 2 \cdot (t_k \cdot a + w_k \cdot b) = 2 \cdot (2^k \cdot b_k) = 2^{k+1} \cdot a_{k+1}$
and
$t_{k+1} \cdot a + w_{k+1} \cdot b = (u_k \pm t_k) \cdot a + (v_k \pm w_k) \cdot b = (u_k \cdot a + v_k \cdot b) \pm (t_k \cdot a + w_k \cdot b) = 2^k \cdot a_k \pm 2^k \cdot b_k = 2^k \cdot (a_k \pm b_k) = 2^{k+1} \cdot b_{k+1}$.

Hence, we proved that the relations (1), (2) are preserved by the transition from step k to the step $k+1$. Since they hold also for $k = 0$, they hold for every $k \geq 0$.

Lemma 1. t_k and w_k are relatively prime, for all $k \geq 0$.

Proof. We build the sequence of determinants $(D_k)_{k \geq 0}$ with $D_k = \begin{vmatrix} u_k & v_k \\ t_k & w_k \end{vmatrix}$.

If we moved from the step k to the step $k+1$ using the rule [**shift** b], the value of the determinant doubles ($D_{k+1} = 2 \cdot D_k$). If we used the rule [**plus**], [**minus**] or [**interchange and shift** b], the determinant doubles and changes the sign ($D_{k+1} = -2 \cdot D_k$). If the rule was [**shift both**], the determinant remains unchanged. Thus, $D_k = \pm 2^p$ and the only common divisors of t_k and w_k can be 1 or powers of 2.

By induction, one can prove that 2 is not a common divisor of t_k and w_k (at least one of them is odd), so they are relatively prime.

The initial values of t_0, w_0 are $0, 1$ respectively, so 2 is not a common divisor. Starting with $k = 1$, u_k and v_k are both even.

The rule [**shift both**] can be applied only in the beginning, and it preserves the values of t_{k+1} and w_{k+1} to 0 respectively 1.

The rule [**interchange and shift** b] can be applied at most once after 0 or more [**shift both**] steps, so $u_{k+1} = 0, v_{k+1} = 2, t_{k+1} = 1, w_{k+1} = 0$ (u_{k+1} and v_{k+1} are odd; 2 is not a common divisor of t_{k+1} and w_{k+1}).

If we used the rule [**shift** b], $t_{k+1} = t_k, w_{k+1} = w_k$, so 2 is not a common divisor of t_{k+1} and w_{k+1}.

If we used the rule [**plus**] or [**minus**] after a [**shift both**] step, we have $t_{k+1} = 1, w_{k+1} = \pm 1$; otherwise at least one of t_{k+1}, w_{k+1} is the sum/difference of an even and an odd number.

Moreover, t_{k+1} and w_{k+1} cannot be both null, and for a n such that $b_n = 0$ we have
$$\frac{a}{b} = -\frac{w_n}{t_n},$$

and $-\frac{w_n}{t_n}$ represents the canonical form of the fraction, up to the signs of the numerator and the denominator.

3 The systolic array

The systolic array is organized as in [4]: 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. 2. 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.

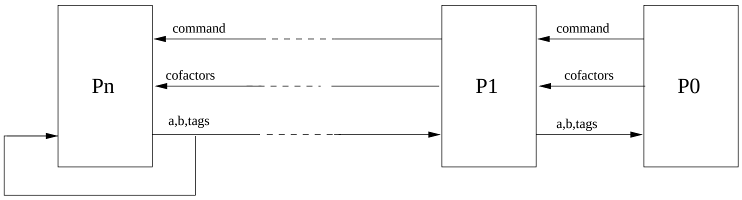

Fig. 2. The systolic array for the extended binary algorithm.

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 [4]: 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.

4 The systolic algorithm

In the next figures 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 ≠ 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 := w;
    else                                     [find the appropriate active state]
        if a = 0 ∧ b = 0 then                               [shift both A and B]
            s := B;
            (a, b, ta, tb) := (a[+], b[+], ta[+], tb[+]);
        if a = 0 ∧ b = 1 then                [interchange A and B and shift B]
            s := C;
            (a, b, ta, tb) := (b, a[+], tb, ta[+]);
        if a = 1 ∧ b = 0 then                                        [shift B]
            s := S;
            (b, tb) := (b[+], tb[+]);
        if a = 1 ∧ b = 1 then                                  [plus or minus]
            if a[+] = b[+] then s := m;                                  [minus]
            else s := 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]

      $tb := (ta \wedge tb)$ [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]

      $tb := (ta \wedge tb)$ [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 processor 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 $P_0, \dots, P_k$. 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 **S** operations which do not change the values of the cofactors t_k and w_k.

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. P. Henrici, *A subroutine for computations with rational numbers*, Journal of the ACM **3** (1956), 6–9.
3. T. Jebelean, *An algorithm for exact division*, Journal of Symbolic Computation **15** (1993), no. 2, 169–180.
4. ———, *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.
5. D. E. Knuth, *The art of computer programming*, vol. 2, Addison-Wesley, 1981.
6. W. Krandick and T. Jebelean, *Bidirectional exact integer division*, Journal of Symbolic Computation **21** (1996), 441–455.
7. J. Stein, *Computational problems associated with Racah algebra*, J. Comp. Phys. **1** (1967), 397–405.
8. K. Weber, *The accelerated integer GCD algorithm*, ACM Trans. on Math. Software **21** (1995), no. 1, 111–122.