An Algorithm for Exact Division *

Tudor Jebelean
RISC-LINZ

Research Institute for Symbolic Computation
Johannes Kepler University, A-4040 Linz, Austria (Europe)
Tel: +43 (7236) 3231-20 Fax: +43 (7236) 3231-30
e-mail: tjebelea@risc.uni-linz.ac.at

October 19, 1999

Abstract

Current computer algebra systems use the quotient-remainder algorithm for division of long integers even when it is known in advance that the remainder is zero. We propose an algorithm which computes the quotient of two long integers in this particular situation, starting from the least-significant digits of the operands. This algorithm is particularly efficient when the radix is a prime number or a power of 2.

The computing time of this new algorithm is smaller than the computing time of the classical division algorithm. If the length of the result is much smaller than the lengths of the inputs, then the speed-up may be quite significant, as it is confirmed by practical experiments.

Most importantly, however, the new algorithm is better suited for systolic parallelization in a "least-significant digit first" pipelined manner, and therefore it is suitable for aggregation with other systolic algorithms for the arithmetic of long integers and long rationals.

We also present applications of this algorithm in integer GCD computation and in division modulo a power of 2.

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

Introduction

In many basic computer algebra algorithms the division of long integers is performed in situations when it is known in advance that the remainder is zero. We shall call this *exact division*. Some examples are operations with rationals and computation of the primitive part of polynomials. Presently, such a division is done by applying the classical quotient–remainder algorithm (see [Knuth 81] section 4.3.1), and then just ignoring the remainder.

In this paper we present an algorithm for division, which speeds-up this operation by taking advantage of the fact that the division is exact. The algorithm is particularly efficient when the radix is a prime or a power of 2, because it uses the inverse modulo the radix of the least-significant digit of the dividend.

The basic idea of the algorithm was first introduced by [Pompeiu 59] (cited by [Davis 85]), as a method of solving the following mathematical puzzle: find the high order digits of the integer C and the quotient C/A, when sufficiently many low order digits of C are known, A is known, A divides C, and the least-significant digit of A is relatively prime to the radix 10. The algorithm for the binary case was first presented in [Purdy 87], together with its systolic implementation.

We generalize this method to arbitrary base and arbitrary divisor, thus turning it into a systematic algorithm for exact division, and we show how this algorithm can be efficiently used for the purpose of algebraic computations.

This algorithm is a good candidate for systolic implementation, because the computation uses the least-significant digits of the operands first (only part of the digits of the dividend are needed[1]), and gives the quotient also least-significant digits first. The "least-significant digits first" manner in which this new algorithm operates makes it suitable for pipelined aggregation with other multiprecision systolic algorithms (like multiplication – [Atrubin 65]), and this is very useful in constructing a systolic system for multiprecision rational arithmetic.

Such an attempt is worthwhile because the speed of multiprecision arithmetic has a decisive influence on the speed of computer algebra systems. By performing a series of measurements upon typical algebraic algorithms [Buchberger, Jebelean 92], we found out that most of the computation time is spent in integer/rational operations, and this proportion increases when the length of the input coefficients increases. Also, the absolute total time increases dramatically, exclusively on behalf of integer/rational operations. For example, in Gröbner Bases computation [Buchberger 85], when increasing the coefficient length from 1 to 10 decimal digits, the proportion of rational operations grows from 62% to 98%, and the total computation time grows by a factor of 25.

The theoretical time-complexity of the new algorithm is different from the complexity of the classical algorithm. If $m \geq n$ are the lengths of the input operands, then the number of digit multiplications is

$$T_{\text{new}} = \begin{cases} (m-n+1)(m-n+2)/2, & \text{if } m+1 \leq 2n, \\ mn - 3n(n-1)/2 & \text{otherwise,} \end{cases}$$

[1]The length of the operands needs to be known in order to exploit this advantage.

while

$$T_{\text{classic}} = n(m - n + 1)$$

(see [Collins 74]). In the first case (when the length of the result is small), the speed-up of $2n/(m-n+2)$ may be quite significant. Benchmarks using the GNU multiple precision arithmetic library [Granlund 89] and SACLIB Computer Algebra system [Collins et. al. 92] show a speed-up of almost 2 for $m = 2n$, and 3 for $m = 3n/2$ (see Fig.3).

The exact division algorithm can be used to devise a "least-significant digit first" GCD algorithm.

Finally, the new division algorithm can be used for inversion modulo a power of 2 much faster than the extended Euclidean algorithm (experiments using SACLIB indicate more than 20 times speed-up).

1 The new algorithm

The basic idea introduced by [Pompeiu 59] is to find the least-significant digit of the quotient by using only the least-significant digits of the operands. Let A, B, C be multiprecision positive integers expressed in radix β, such that

$$C = A * B$$

If a, b, c are the least-significant digits, then

$$c = (a * b)_{mod\beta}$$

If β is a prime number and $a \neq 0$, then one can find $a' = a^{-1}_{mod\beta}$, hence

$$b = (a' * c)_{mod\beta}$$

Thus one obtains the exact division algorithm for prime radix which is presented in Fig.1. Note that line [15] is a loop containing `Length`(A) simple precision multiplications.

One can see that only the least-significant K digits of C are actually used in the computation. Hence, a great deal of computation can be saved if in the inner loop only the interesting digits of the new C are computed. The improved algorithm is obtained from the previous one by replacing [15] by

[15'] $\quad C_{k+1} \leftarrow (C_k - b_k * A)_{mod\beta^{K-k}}/\beta$

Now the inner loop contains max(`Length`(A), $K-k$) multiplications.

The same algorithm can be applied to exact division of polynomials (A, B, C are univariate polynomials, a, b, c are the coefficients of the least-degree monomials). Here there is no problem with the primality of β or with the modular division. However, we see no advantage of this method over the classical left-to-right one, unless, for some reasons (parallelization, pipelining), the right-to-left

```
B ← EDIV(C_0, A) [B = C_0/A,  A|C_0, β is prime]
begin
   [Skip common trailing zeroes]
   while β|A
      A ← A/β
      C_0 ← C_0/β
   [Compute modular inverse]
   a' ← (LowestDigit(A))^{-1}_{mod β}
   B ← Empty
   K ← Length(C_0) − Length(A) + 1
   [Main loop]
   for k = 0 to K − 1 do
      b_k ← (a' * LowestDigit(C_k))_{mod β}
      SetNewHighestDigit(B, b_k)
      [Inner loop]
      C_{k+1} ← (C_k − b_k * A)/β
   return B
end
```

Figure 1: EDIV: the algorithm for exact division.

version of the algorithm is needed. It is interesting to remark that, in the case of exact division of polynomials, some computations can be saved if one computes only the needed coefficients of the new C, as in the scheme above. Note that the same improvement can be applied to the classical algorithm, and the speed-up is quite high when the degrees of A and C are close (see time complexity analysis below).

Coming back to the division of integers, let us suppose the radix is arbitrary. Then a is invertible only if β and a are relatively prime. Let be $P = p_1^{k_1} * \ldots * p_m^{k_m}$ the highest P such that P divides A and the primes $p_1, \ldots, p_m$ divide β. P can be determined in the following manner:

- Set $P \leftarrow \beta^k$ and $A \leftarrow A/\beta^k$, where k is the number of null least-significant digits of A.
- Repeatedly find $g = GCD(\beta, \texttt{LowestDigit}(A))$ and set $A \leftarrow A/g$, $P \leftarrow gP$, until g becomes 1.

Since P divides A, P also divides C, hence one can apply EDIV to C/P and A/P. This reduces the efficiency of the algorithm when P is big, since one has to apply the classical algorithm for these preliminary divisions.

However, in the implementation of a computer algebra system, the radix is usually[2] a power of 2. In order to adapt the algorithm to this case, we introduce

[2] Some interactive systems (like Maple V) use a power of 10 as radix. This eases the conversion

a right shifting of A until it becomes odd. Then a is odd, hence a and β are relatively prime. Since A divides C, C may be shifted by the same number of positions, hence the quotient does not change. The new form of the algorithm is obtained by replacing lines [3] – [5] by

```
[ 3']    while 2|A
[ 4']      A ← A/2
[ 5']      C ← C/2
```

Let us call the classical quotient–remainder algorithm IQR. (A detailed description of this algorithm is given in [Knuth 81], section 4.3.1 and the time-complexity analysis is given in [Collins 74].) The advantages of EDIV in comparison with IQR are:

- Easiest computation of the next digit of the quotient.

  In IQR, the next digit of the quotient is computed by a division of a two-digit integer by a one-digit integer. Also, a certain amount of guessing is involved.

  In EDIV, the next digit is obtained by a one-digit multiplication, from whose result only the least-significant digit is needed. (Note that the modular inversion is performed only once, at the beginning of the procedure.)

- Less digit multiplications are required. In Fig.2 we present a pictorial "estimation" of the speed up (shaded areas correspond to the number of digit multiplications). A detailed analysis follows below.

  If $m = \mathtt{Length}(C)$, $n = \mathtt{Length}(A)$, then IQR requires $n(m-n+1)$ multiplications (area of the shaded parallelogram).

  In EDIV, if $m+1 \geq 2n$, then $\max(n, K-k)$ multiplications are performed in line [15']. Thus, the total is

  $$((m-n+1)-(n-1))n + (n-1) + \ldots + 2 + 1 = mn - 3n(n-1)/2.$$

  This means $n(n-1)/2$ less multiplications, and the theoretical speed up is

  $$\frac{mn - n(n-1)}{mn - \frac{3}{2}n(n-1)} = 1 + \frac{n-1}{2m-3n+3}.$$

  Since $m+1 \geq 2n$ implies $2m-3n+3 \geq n+1$, the speed up in this case ranges between 1 and 2.

  If $m+1 \leq 2n$, then $K-k$ multiplications are performed in line [15']. Thus, the total number of multiplications is

  $$(m-n+1) + (m-n) + \ldots + 2 + 1 = (m-n+1)(m-n+2)/2.$$

of integers into user-readable format, but also reduces the efficiency of multiprecision computations. For instance, in long integer multiplication, the inner loop contains a digit multiplication whose result needs to be splitted into two digits. If the radix is not a power of 2, this splitting requires a division.

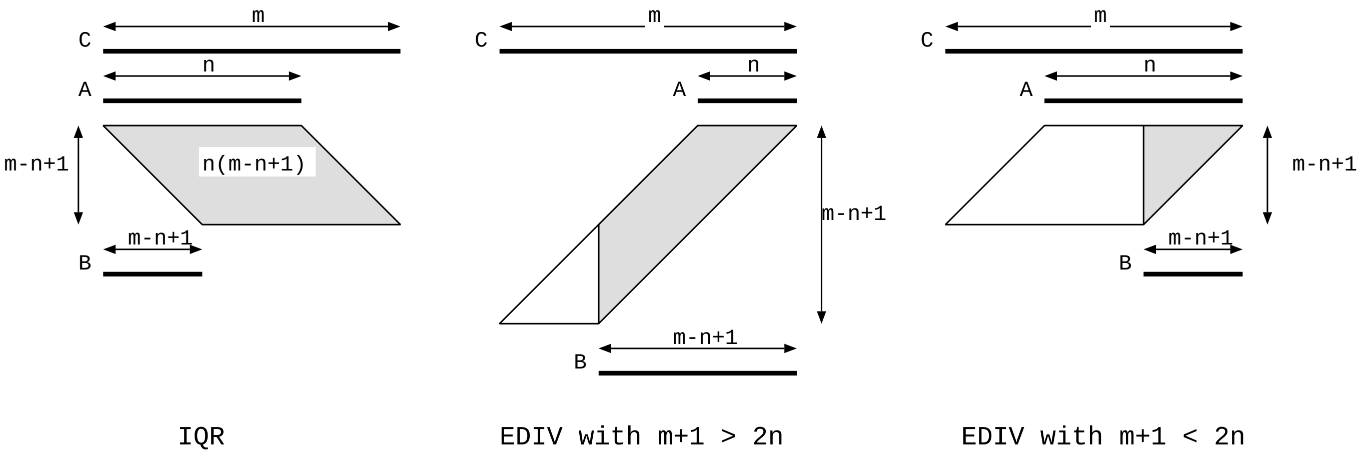

Figure 2: Evaluation of computing times

The theoretical speed up is

$$2\frac{n}{m-n+2} = 2\frac{\texttt{Length}(A)}{1+\texttt{Length}(B)},$$

which can be quite high when length of B is relatively small. (For instance, if $m = 34$ and $n = 30$, then the theoretical speed up is 10.)

In Fig.3 we presents benchmarks of the new algorithm using the GNU multiprecision arithmetic library [Granlund 91] ($\beta = 2^{32}$) and also the SACLIB Computer Algebra system [Collins e.a. 92] ($\beta = 2^{29}$), with GNU C optimizing compiler on a the Digital DECstation 5200 (MIPS RISC architecture). For large lengths of operands, the measured speed-ups agree well with the theoretical analysis, while for small lengths the speed up is less than expected. This is probably due to the fact that when lengths are small, the weight of multiplication steps within the total computation time is smaller. In the SACLIB implementation, the lengths of the two operands have to be computed by scanning the linked lists which store the long integers. This may also contribute decrease of speed up for small lengths. However, for longer integers the effect of this additional operation dissapears.

2 GCD computation

Like the quotient-remainder algorithm, EDIV can also be used to build a GCD algorithm, which will work "least-significant digits first".

In the first version of the algorithm EDIV (prime radix, full computation of C_{k+1}), suppose $A < C_0$ are arbitrary (condition $A \mid C_0$ is not assumed anymore), and the main loop is repeated as long as $C_k > 0$. Then the algorithm stops when for some j

$$C_{j+1} \leq 0 < C_j.$$

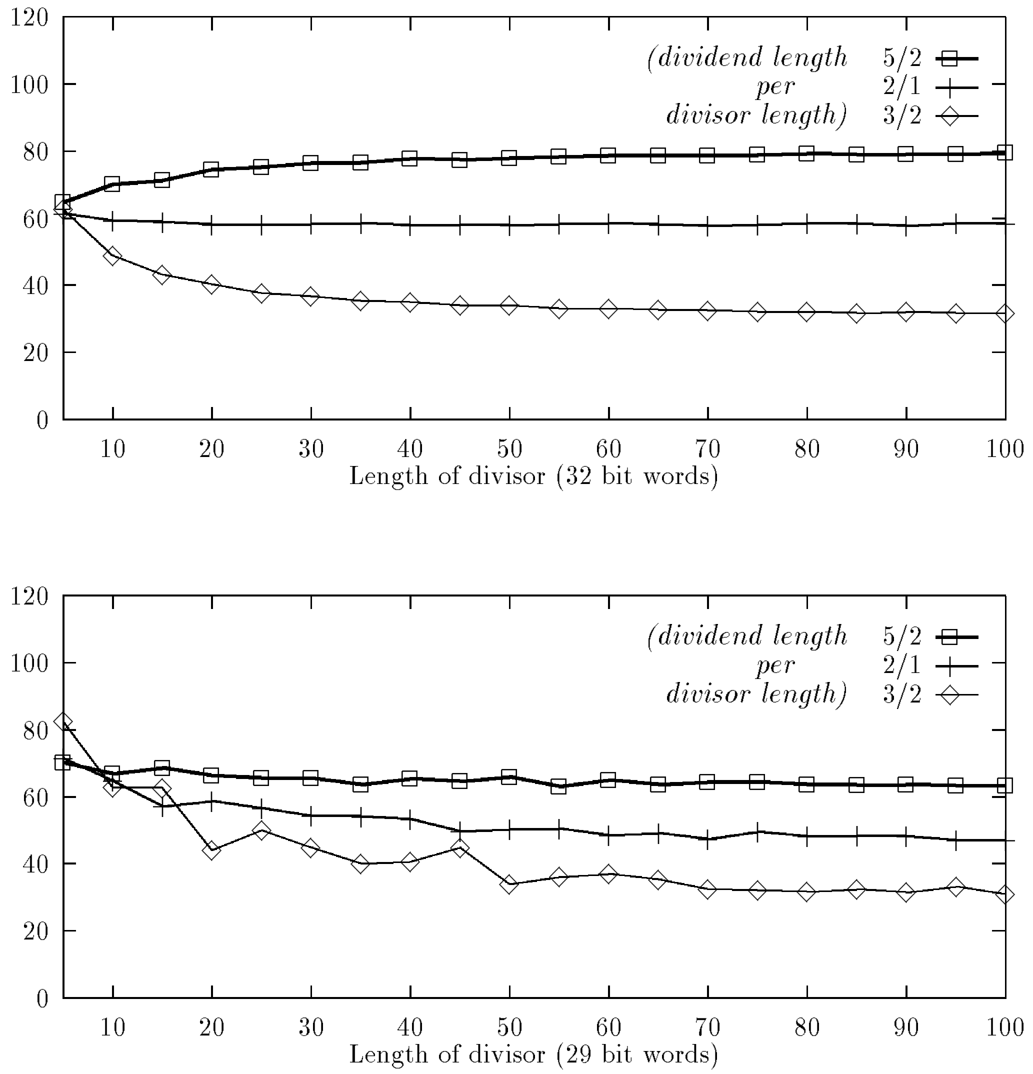

Figure 3: EDIV/IQR % comparative timings, under GNU (top) and SACLIB (bottom).

Then the following equivalent relations hold:

$$-C_{j+1} = (A * b_j - C_j)/\beta,$$
$$-C_{j+1} * \beta = A * b_j - C_j < A * b_j,$$
$$-C_{j+1} < A * \frac{b_j}{\beta} < A.$$

In particular

$$0 \leq -C_{j+1} < A. \tag{1}$$

Furthermore, one obtains inductively

$$C_1 * \beta = C_0 - b_0 * A,$$
$$C_2 * \beta^2 = C_1 * \beta - b_1 * A * \beta = C_0 - (b_1 * \beta + b_0) * A,$$
$$\cdots$$
$$C_{j+1} * \beta^{j+1} = C_0 - (b_j * \beta^j + \ldots + b_0) * A,$$

and so

$$C_0 = A * B + C_{j+1} * \beta^{j+1}.$$

Since the prime β divides neither C_0, nor A, one has

$$GCD(C_0, A) = GCD(A, -C_{j+1}). \tag{2}$$

The relations (1) and (2) allow us to construct the algorithm EDGCD (Fig.4) for GCD computation.

The algorithm terminates because at each iteration of the main loop, A is replaced with a non-negative strictly smaller value.

Analogous to the division algorithm, this new GCD algorithm can be also adapted to the case when the radix is a power of 2, by changing lines [4] – [7] into

[4'] **while** $2|A$ and $2|C$
[5'] $A \leftarrow A/2$
[6'] $C \leftarrow C/2$
[7'] $G \leftarrow G * 2$

and lines [11], [12] into

[11'] **while** $2|A$
[12'] $A \leftarrow A/2$

Experimentally, we noticed that the average number of iterations of the main loop is 30% smaller than the number of divisions required by the Euclidean algorithm. This is the same as the number of divisions required by the modified Euclidean algorithm ($(A, B) \leftarrow (B, \min(A_{\mathrm{mod}\ B}, B - A_{\mathrm{mod}\ B}))$, instead of

```
G ← EDGCD(C, A) [G = GCD(C_0, A), C ≥ A, β prime]
begin
   G ← 1
   [Skip common trailing zeroes]
   while β|A and β|C
      A ← A/β
      C ← C/β
      G ← G * β
   [Main loop]
   while A ≠ 0
      [Skip trailing zeroes of A]
      while β|A
         A ← A/β
      a' ← (LowestDigit(A))^{-1}_{modβ}
      while C > 0
         C ← (C − A * (a' * LowestDigit(C))_{modβ})/β
      (C, A) ← (A, −C)
   return G ← G * C
end
```

Figure 4: EDGCD: Exact division based GCD algorithm.

$(A, B) \leftarrow (B, A_{\mathrm{mod}\ B})$, see [Knuth 81], Section 4.5.3, Exercise 30), which is known to be optimal for GCD algorithms based on division [Kronecker 01]. Fig.5 shows the timings and the average number of iterations of the main loop needed in EDGCD (our algorithm) compared with the number of divisions required by the modified Euclidean algorithm. For each length, 1000 random pairs of integers were tested. The experiments were done using the GNU multiprecision library [Granlund 91] and GNU optimizing compiler on Digital DECstation 5200 (MIPS RISC architecture).

3 Division modulo 2^n

Let us consider the algorithm MODIV shown in Fig.6.

According to line [10]

$$C_{k+1} * \beta = (C_k - b_k * A)_{\mathrm{mod}\beta^{(N-k)}},$$

and by induction

$$C_K * \beta^N = (C_0 - B * A)_{\mathrm{mod}\beta^N},$$

where

$$B = b_{N-1}\beta^{N-1} + \ldots + b_0.$$

Therefore

$$B = (C_0/A)_{\mathrm{mod}\beta^N},$$

Length of operands	Timings in milliseconds			Number of steps		
	EDGCD	Modified Euclidean	Relative %	EDGCD	Modified Euclidean	Relative %
5	2.0	2.2	91.4	66	66	100.0
10	5.1	5.5	93.0	131	130	100.8
15	9.2	10.1	90.9	196	195	100.5
20	14.3	15.8	90.3	262	260	100.8
25	20.5	22.8	89.8	327	325	100.6
30	27.8	30.9	89.9	393	390	100.8
35	36.8	40.2	91.5	460	454	101.3
40	45.8	50.7	90.4	524	519	101.0
45	56.8	62.3	91.2	590	584	101.0
50	68.1	75.3	90.5	655	649	100.9
55	80.5	89.5	89.9	721	714	101.0
60	94.2	105.1	89.6	786	779	100.9
65	109.5	121.5	90.1	854	844	101.2
70	124.8	138.7	90.0	916	908	100.9
75	142.2	157.6	90.2	983	974	100.9
80	160.3	177.6	90.2	1047	1039	100.8
85	179.5	198.8	90.3	1113	1104	100.8
90	199.5	221.6	90.0	1180	1170	100.9
95	220.4	244.9	90.0	1243	1233	100.8
100	243.1	269.4	90.2	1313	1297	101.2

Figure 5: Benchmarks of EDGCD and modified Euclidean GCD.

$B \leftarrow MODIV(C_0, A, N)$ $[B = (C_0/A)_{\mathrm{mod}\beta^N},\ GCD(A,\beta) = 1]$
`begin`
  [Compute modular inverse]
  $a' \leftarrow (\mathtt{LowestDigit}(A))^{-1}_{mod\beta}$
  $B \leftarrow \mathtt{Empty}$
  [Main loop]
  `for` $k = 0$ `to` $N - 1$ `do`
    $b_k \leftarrow (a' * \mathtt{LowestDigit}(C_k))_{mod\beta}$
    $\mathtt{SetNewHighestDigit}(B, b_k)$
    [Inner loop]
    $C_{k+1} \leftarrow (C_k - b_k * A)_{\mathrm{mod}\beta^{N-k}}/\beta$
  `return` B
`end`

Figure 6: MODIV: the algorithm for modular division.

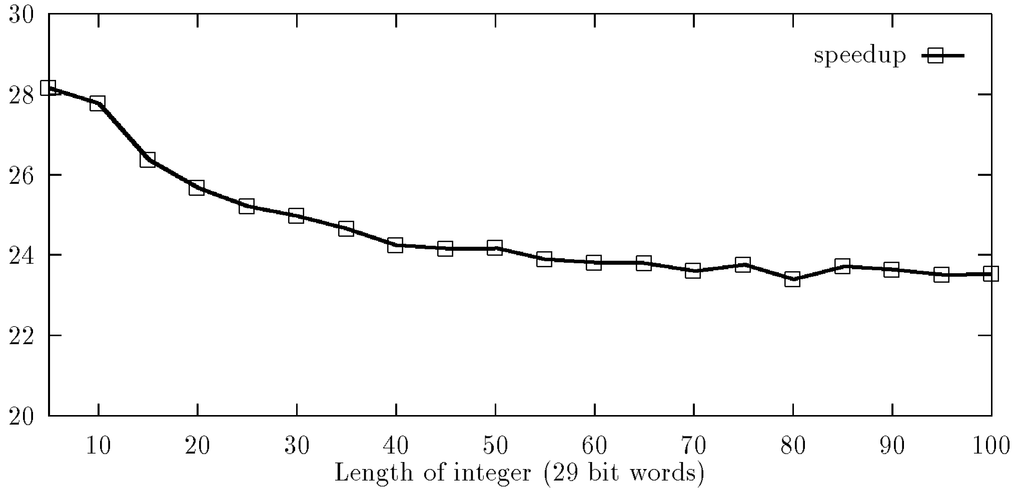

Figure 7: Speed up of modular inverse using SACLIB.

hence the algorithm MODIV performs modular division (in particular, modular inverse, by taking $C = 1$ and $N = \text{Length}(A)$).

This is more efficient then the extended GCD algorithm because, essentially, it is equivalent to a long division. Also, the modular quotient is computed directly, while the classical procedure needs an inversion followed by a multiplication.

Experiments on a DECstation 5200 using SACLIB indicate more than 20 times speed up in computing the inverse modulo a power of $\beta = 2^{29}$ (see Fig. 7). The speed up was measured in comparison with the extended Euclidean algorithm improved by [Lehmer 38] and [Collins 80].

For $\beta = 2$, one obtains an algorithm for division modulo 2^n which is simple enough to be suitable for hardware implementation (see Fig. 8). Here by $C[i], B[i]$ we mean "the i-th binary position", counted from right to left, and by `ShiftLeft` we mean shifting by 1 binary position.

Independently from our research, [Văcariu 92] observed that a very similar algorithm can be used for right to left computation of the period of the result of a fixed–point division.

For implementing the digit modular inversion needed in EDIV, we have used an interesting recursion formula which is derived in the sequel.

Let

$$A = a_1 * \beta + a_0, \quad C_0 = 1, \quad N = 2.$$

By applying MODIV "symbolically" one obtains

$$a' = (a_0)^{-1}_{\mathrm{mod}\beta},$$

$$b_0 = (a' * 1)_{\mathrm{mod}\beta} = a',$$

```
B ← MODIV2(C, A) [B = (C/A)_{mod 2^n}, A odd]
begin
  for i = 0 to (n − 1)
    if C[i] = 1 then
      C ← C − A
      B[i] ← 1
    else
      B[i] ← 0
    ShiftLeft(A)
  return B
end
```

Figure 8: MODIV2: binary algorithm for division modulo 2^n.

$$C_1 = (C_0 - b_0 * A)_{\mathrm{mod}\beta^2}/\beta = (1 - a' * A)_{\mathrm{mod}\beta^2}/\beta,$$

and note that $C_1 < \beta$, hence

$$b_1 = (C_1 * a')_{\mathrm{mod}\beta},$$

$$b_1 * \beta = (C_1 * \beta * a')_{\mathrm{mod}\beta^2} = ((1 - a' * A) * a')_{\mathrm{mod}\beta^2}.$$

Therefore, since

$$A^{-1}_{mod\beta^2} = b_1 * \beta + b_0,$$

one has

$$A^{-1}_{mod\beta^2} = (((1 - A * a') * a') + a')_{mod\beta^2}$$

$$\text{where} \quad a' = (a_0)^{-1}_{mod\beta}.$$

This relation holds whenever A and β are relatively prime (thus, whenever $A^{-1}_{\mathrm{mod}\beta^2}$ exists), and reduces the computation of the modular inverse of A to the computation of the modular inverse of the least-significant half of A.

For instance, if inversion modulo 2^{32} is needed, by applying two times the formula above, one reduces the problem to the computation of the modular inverse of an 8-bit number, which can be done by a simple look-up in a precomputed table. On a Digital DECstation 5200, the computing time necessary for inversion of 1,000,000 random numbers using this scheme was 3.8 seconds, vs. 49.5 seconds using the extended Euclidean algorithm (speed up by a factor of 13). By using a precomputed table of inverses of 16-bit numbers, the computing time was 2.6 seconds (speed up by a factor of 19).

4 Systolic computations

The exact division algorithm presented above is a good candidate for systolic implementation: at each iteration of the main loop, only the least-significant digits of C_k and A are required for computing the next digit b_k of the quotient.

After that, the least-significant digit of C_{k+1} can be computed and then b_{k+1}, while other processors can work in parallel for updating the rest of the digits of C_{k+1}, since these computations do not affect the value of b_{k+1}. Note that this parallelization scheme cannot be applied to the left–to–right classical division algorithm: updating the value of C must be completed before proceeding to the next step, since the value of the most-significant digit is likely to be affected by the computations upon the least-significant digits.

"Least-significant digit first" processing is also used in the most efficient systolic algorithm for long integer multiplication ([Atrubin 65], also [Knuth 81], Section 4.3.3 E), and can be easily implemented for addition/subtraction. Therefore, one can aggregate EDIV and the other multiprecision algorithms in a pipelined structure. This is very useful, for instance, in building a systolic device for rational arithmetic.

References

[Atrubin 65] A. J. Atrubin — *A one–dimensional iterative multiplier*, IEEE C-14 (1965) 394–399.

[Buchberger 85] B. Buchberger — *Gröbner Bases: An algorithmic method in polynomial ideal theory*, in N. K. Bose (ed.) — *Multidimensional Systems Theory*, D. Reidel Publishing Co., 1985.

[Buchberger, Jebelean 92] B. Buchberger, T. Jebelean — *Parallel rational arithmetic for Computer Algebra systems: Motivating experiments*, RISC–Linz Report 92-29, May 1992.

[Collins 74] G. E. Collins — *The computing time of the Euclidean algorithm*, SIAM J. Comput. 3 (1974), 1, 1 – 10.

[Collins 80] G. E. Collins — *Lecture notes on arithmetic algorithms*, Univ. of Wiscousin, 1980.

[Collins et. al. 92] G. E. Collins, H. Hong, M. Encarnation, J. Johnson, W. Krandick, A. Neubacher — *A SACLIB primer*, RISC–Linz Report 92-34, 1992.

[Davis 85] P. J. Davis — *Pompeiu's magic seven*, in P. J. Davis, W. G. Chain — *3.1416 and all that*(second edition), Birkhäuser, Boston 1985, pp. 14 – 19.

[Granlund 91] T. Granlund — *GNU MP: The GNU multiple precision arithmetic library*, Free Software Foundation, 1991.

[Knuth 81] D. E. Knuth — *The art of computer programming*, Vol. 2, Addison-Wesley 1981.

[Kronecker 01] L. Kronecker — *Vorlesungen über Zahlentheorie*, Vol. 1, Teubner, Leipzig, 1901, 118.

[Pompeiu 59] D. Pompeiu — *Oeuvre mathematique*, Academie de R. P. R., Bucarest, 1959.

[Purdy 87] C. N. Purdy, G. B. Purdy — *Integer division in linear time with bounded fan–in*, IEEE Trans. Computers C-36 (1987), 5, 640–644.

[Văcariu 92] C. T. Văcariu — *Method and symmetrical architecture circuit for performing the exact division through step by step approximation, in various ways and formats (in German)* — Patent Application 892/92, Österreichisches Patentamt – Vienna, May 1992.