Systolic Algorithms 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`

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. Several systolic implementations of this algorithm are presented, differing in the way inputs and output are pipelined. One of the algorithms is *on-line*: each digit of the quotient is obtained after the next two digits of the operands are fed into the array (least-significant digits first). Time and space complexity of the algorithms are linear in the length of the quotient. Systolic algorithms for exact division of polynomials are also presented.

Introduction

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.

This paper is part of a research aimed at speeding up computer algebra systems by systolic parallelization of long integer and long rational arithmetic. Systolic parallelization of long rational arithmetic in the most-significant digits first (MSF) pipelined manner was considered by [Trivedi, Ercegovac 77]. In particular, MSF systolic division was discussed by several authors ([Gex 71], [Kamal, Singh, Agrawal 75], [Preparata, Vuillemin 90]). However, the MSF methods require redundant representations, hence additional transformation operations have to be performed. Only [Purdy 87] presents a

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

least-significant digits first (LSF) exact division algorithm for binary radix, which uses global broadcasting of quotient digits. In fact, most of the approaches to this problem deal with the binary case, which limits the application of these algorithms to hardware implementation.

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.

[Jebelean 92] introduces a LSF algorithm for exact division based on the idea of [Pompeiu 59] (also in [Davis 85]) of using the modular inverse of the least-significant digit of the dividend.

We present here several variants of LSF systolic exact division algorithms which also work when the radix is an arbitrary power of 2, hence they can be also implemented on multiprocessor machines. We also show how to eliminate global broadcasting of quotient digits and how to construct an *on-line* algorithm, in which the result is obtained in a LSF pipelined fashion. The LSF manner in which this new algorithms operate makes them suitable for pipelined aggregation with other LSF systolic algorithms (like multiplication – [Atrubin 65], GCD – [Brent, Kung 83]), and with units for pipelined LSF addition/subtraction. This is very useful in constructing systolic systems for multiprecision rational arithmetic.

In the first section we present systolic algorithms for exact division of univariate polynomials, and in the second section we refine this algorithms in order to make them operant for multiprecision integers.

1 Exact division of polynomials

We work with univariate polynomials over a field. The basic arithmetic operations in the field are considered atomic.

The inputs to the exact division algorithm are:

$$C = c_0 + c_1 * x + c_2 * x^2 + \ldots + c_n * x^n,$$

$$A = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_m * x^m,$$

with

$$m \leq n \quad \text{and} \quad A|C.$$

We shall further suppose

$$a_0 \neq 0.$$

The cases when

$$a_0 = a_1 = \ldots = a_k = 0 \text{ for some } k$$

can be reduced to the previous case by some simple shift operations.

The output is:

$$B = b_0 + b_1 * x + b_2 * x^2 + \ldots + b_p * x^p,$$

such that:

$$C = A * B.$$

1.1 Sequential algorithm

The basic idea of the algorithm introduced in [Jebelean 92] is to compute the least-degree coefficient b_0 of B by using only the least-degree coefficients a_0 and c_0 of A and C:

$$b_0 = c_0/a_0.$$

After b_0 is obtained, the next coefficient b_1 can be computed by applying the same scheme to A and $(C - b_0 * A)/x$.

$B \leftarrow$ `SeqPoly`(C, A) [Sequential polynomial exact division]
[$n =$ `Degree`(C)]
[$m =$ `Degree`(A)]
[$p = n - m =$ `Degree`(C)]
[Main loop]
for $k = 0, 1, \ldots, p$ **do**
  $b_k \leftarrow c_k / a_0$
  [Inner loop]
  for $i = 1, 2, \ldots, p - k$ **do**
  $c_{k+i} \leftarrow c_{k+i} - b_k * a_i$

Figure 1: Sequential algorithm for polynomial exact division.

The sequential algorithm is presented in Fig.1. In order to simplify the presentation, we extend A by

$$a_{m+1} = a_{m+2} = \ldots = a_{n-m} = 0$$

in the case $m < n - m$.

1.2 Systolic algorithm: Version 0

The most obvious systolic algorithm can be obtained from `SeqPoly` by parallelizing the inner loop.

A number of $p + 1$ processors are used, processor P_k containing a_k and c_k $(0 \leq k \leq p)$. At step k, the processor P_k performs: $b_k \leftarrow c_k/a_0$, and then broadcasts the value of b_k to the processors $P_{k+1}, \ldots, P_p$, such that each of them can perform $c_{k+i} \leftarrow c_{k+i} - b_k * a_i$ during the next step. Since P_{k+i} needs a_i, A must be pipelined rightward through the array. After k steps, $P_1, \ldots, P_k$ become idle, just holding the values of $b_1, \ldots, b_k$. Step 0 is a little different: in order to have uniformity in the subsequent steps, we introduce a leftward shift of A.

Fig.2 shows the flow of computation $p = 5$. Each column represents one processor (the leftmost column is P_0). Each row represents one step of computation (the topmost column is step 0). There are several different cell behaviors ("states"), denoted by *1, 2, 3.* The idle steps (state *0*) are not displayed.

Fig.3 presents the formal description of the systolic algorithm. The description is made out of five declarations:

- `Local` enumerates the variables which have an instance on each processor. Typically, the local variable s contains the "state" information. Cells being in different states behave differently.
- `Global` enumerates the variables which are accessible to all processors. This declaration is missing if the algorithm does not use global broadcasting.
- `Input` describes the initial values of the local variables. The construction $v_{[k]}$ means "the value of variable v on processor P_k".
- `Evolution` describes one step of computation. This step is performed on each processor, repeatedly and synchronously. The construct $v_{[-]}$ ($v_{[+]}$) stands for the value of the variable v on the left (right) neighboring processor, at the end of the previous evolution step. Typically, an evolution step is a switch instruction over the state variable s, with a `case` for each possible value of s.
- `Output` describes the way in which the output of the algorithm is obtained from the systolic array.

The characteristics of the algorithm `SysPoly.0` are:

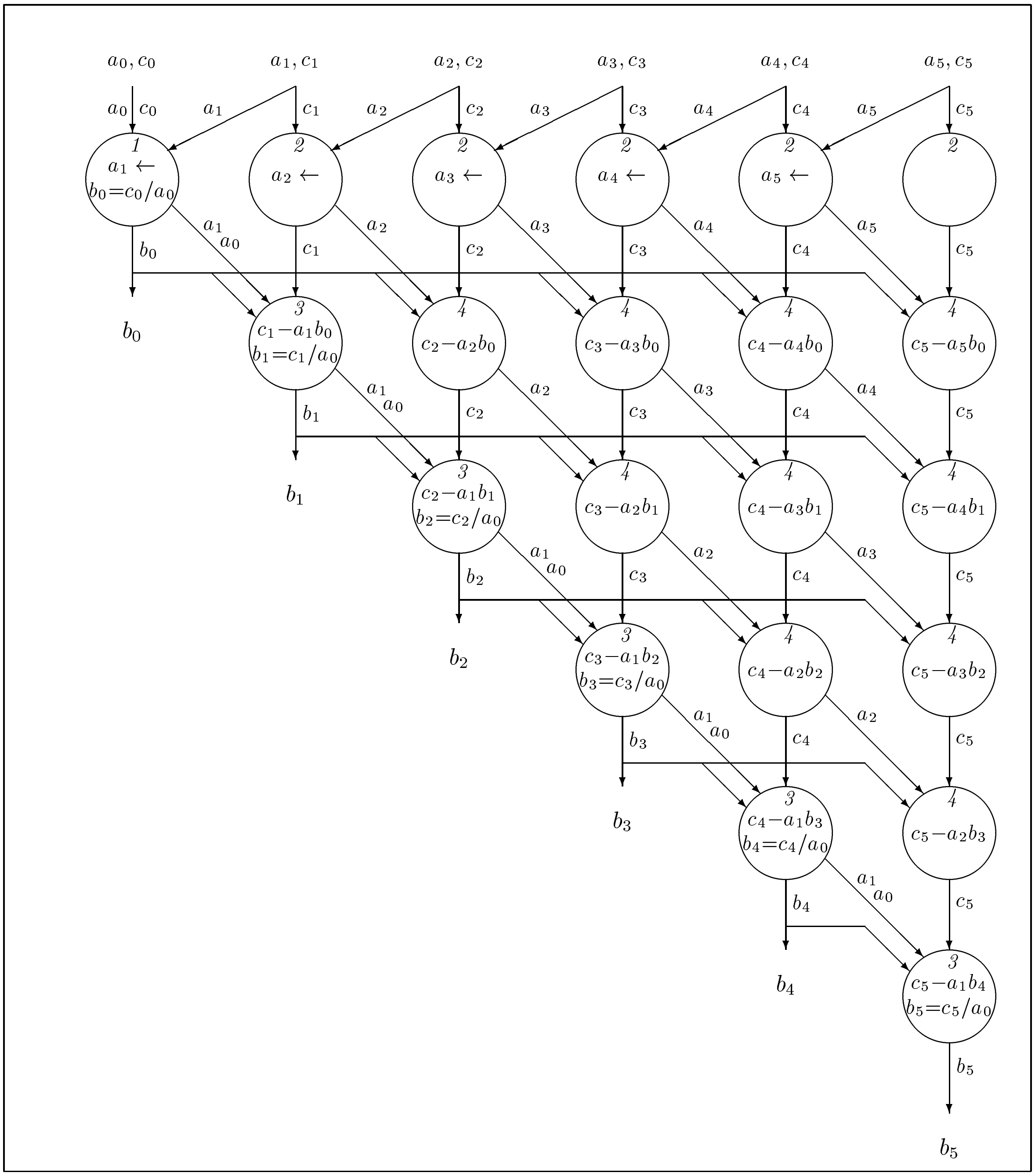

Figure 2: Data flow in version 0 of systolic polynomial exact division.

$B \leftarrow$ `SysPoly.0`(C, A) [Systolic polynomial exact division, version 0]

`Local`: a, a', c, s

`Global`: b_G

`Input`: $a_{[k]} = a_k, \quad 0 \leq k \leq \min(n, p) \quad [n = \texttt{Degree}(C)]$

$a_{[k]} = 0, \quad \min(n, p) < k \leq p \quad [m = \texttt{Degree}(A)]$

$c_{[k]} = c_k, \quad 0 \leq k \leq p \quad [p = n - m = \texttt{Degree}(B)]$

$s_{[0]} = 1$

$s_{[k]} = 2, \quad 1 \leq k \leq p$

`Evolution`: `case` $s = 1$: $a' \leftarrow a \quad [a_0]$

$a \leftarrow a_{[+]} \quad [a_1]$

$b_G \leftarrow c/a' \quad [b_0]$

$s \leftarrow 0$

`case` $s = 2$: $a \leftarrow a_{[+]}$

`if` $s_{[-]} = 2$ `then` $s \leftarrow 4$

`else` $s \leftarrow 3$

`case` $s = 3$: $a \leftarrow a_{[-]} \quad [a_1]$

$a' \leftarrow a'_{[-]} \quad [a_0]$

$b_G \leftarrow (c - a * b_G)/a' \quad [b_1, b_2, \ldots]$

$s \leftarrow 0$

`case` $s = 4$: $a \leftarrow a_{[-]}$

$c \leftarrow c - a * b_G$

$s \leftarrow s_{[-]}$

`case` $s = 0$: [no action]

`Output`: $b_k = b_{[k]}, \quad 0 \leq k \leq p$

Figure 3: Systolic algorithm for polynomial exact division: version 0.

- The number of steps and the number of processors is $p+1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p+1$ coefficients are needed). This is useful when some previous computation (for instance, GCD computation) leaves the two operands in the array.
- B is in the systolic array at the end of the computation. This is useful when B is needed in the array for some subsequent computation.
- Global broadcasting is used to send to all the cells the value of b_k. Thus, the algorithm can be implemented only on devices which allow such operation.
- During the computation, A moves rightward through the array, while C is kept steady.

1.3 Systolic algorithm: Version 1

By slightly changing the previous algorithm, we obtain a new scheme in which A is kept steady in the array, while the updated values of C are pipelined leftward. At each step, a new coefficient of B is obtained in the leftmost cell. Fig.4 shows the flow of computation for $p=5$.

The formal description is presented in Fig.5.

The characteristics of this algorithm are:

- The number of steps and the number of processors is $p+1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p+1$ coefficients are needed).
- B is obtained one coefficient at a time, LSF, in the leftmost processor. This is useful when B is needed in another unit for subsequent computation. This other unit can start working with the least-degree coefficients of B before the division is finished.
- Global broadcasting is used to send to all the cells the value of b_k.
- During the computation, C moves leftward through the array, while A is kept steady.

1.4 Systolic algorithm: Version 2

In order to eliminate the global broadcasting, the values of b_k must be pipelined rightward through the array.

Fig.6 depicts the flow of computation for $p=5$.

The description of the algorithm is presented in Fig.7.

The characteristics of this algorithm are:

- The number of steps and the number of processors is $p+1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p+1$ coefficients are needed).
- B is obtained one coefficient at a time, LSF, in the leftmost processor. Also, at the end of the computation, B is available in the array, but in the reversed order.
- No global broadcasting is needed.
- During the computation, C moves leftward through the array and B is pipelined rightward. Each coefficient a_k of A moves leftward until it reaches the processor $P_{k/2}$, where it is stored.

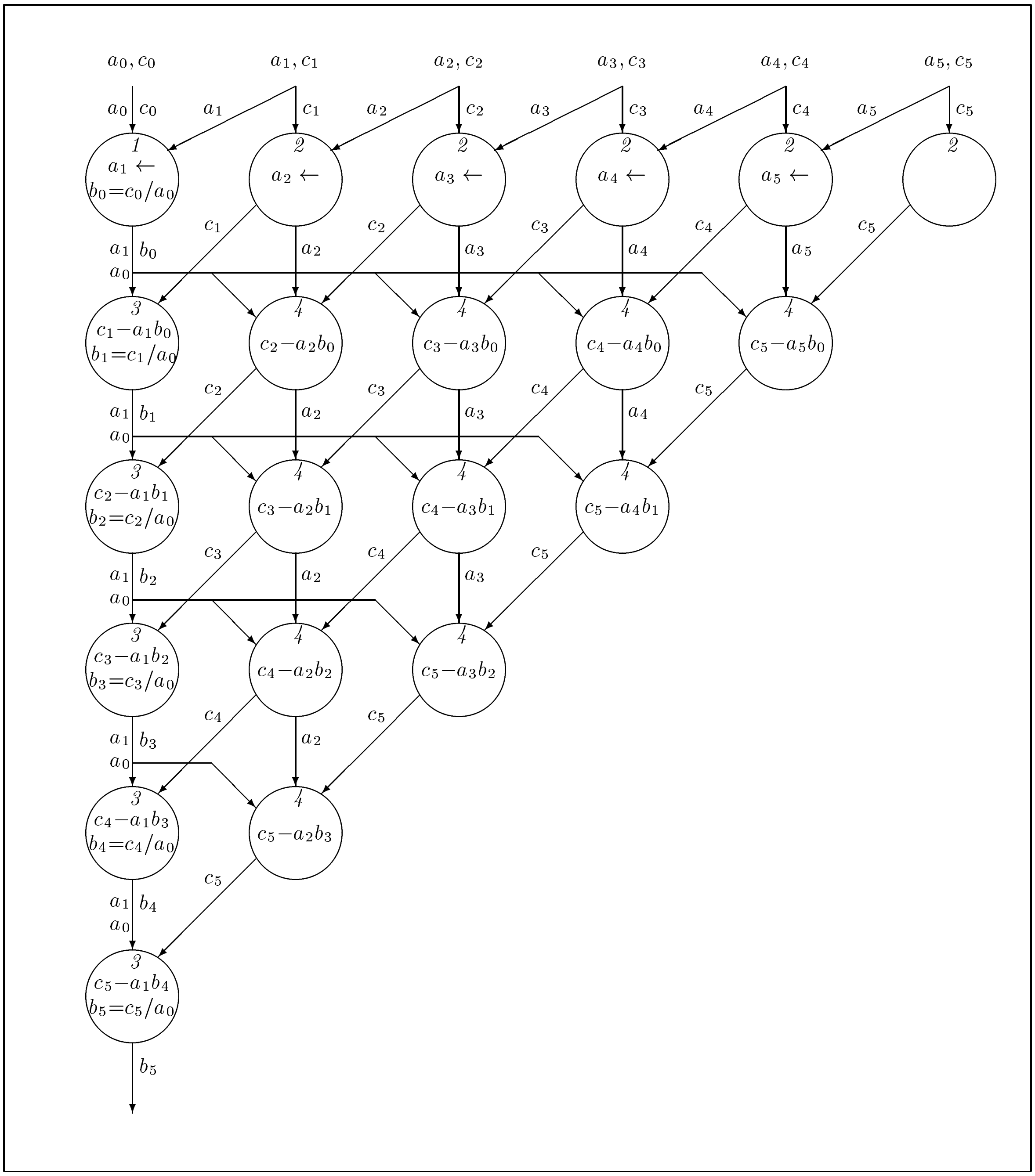

Figure 4: Data flow in version 1 of systolic polynomial exact divison.

$B \leftarrow$ `SysPoly.1`(C, A) [Systolic polynomial exact division, version 1]

`Local`: a, a', c, s

`Global`: b_G

$$
\begin{array}{llll}
\texttt{Input:} & a_{[k]} = a_k, & 0 \le k \le \min(n,p) & [n = \texttt{Degree}(C)] \\
& a_{[k]} = 0, & \min(n,p) < k \le p & [m = \texttt{Degree}(A)] \\
& c_{[k]} = c_k, & 0 \le k \le p & [p = n - m = \texttt{Degree}(B)] \\
& s_{[0]} = 1 & & \\
& s_{[k]} = 2, & 1 \le k \le p &
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Evolution: case } s = 1\text{:} & a' \leftarrow a & [a_0] \\
& a \leftarrow a_{[+]} & [a_1] \\
& b_G \leftarrow c/a' & [b_0] \\
& s \leftarrow 3 & \\
\texttt{case } s = 2\text{:} & a \leftarrow a_{[+]} & \\
& s \leftarrow 4 & \\
\texttt{case } s = 3\text{:} & b^G \leftarrow (c_{[+]} - a * b_G)/a' & \\
\texttt{case } s = 4\text{:} & c \leftarrow c_{[+]} - a * b_G &
\end{array}
$$

`Output`: $b_k = b_{[0]}$ at step k $(0 \le k \le p)$

Figure 5: Version 1 of the algorithm for polynomial exact division.

1.5 Systolic algorithm: Version 3

The last version of the systolic algorithm does not need the operands within the array at the beginning of computation. Rather, A and C are fed into the array one coefficient at a time, LSF. B is produced also LSF, in on-line fashion: coefficient b_k is produced immediately after a_k and c_k are fed into the array.

This algorithm is a modified version of the Atrubin's multiplication scheme [Atrubin 65]. Namely, the processors $P_1, P_2, \ldots$ compute the partial products of $A * B$ as in the multiplication algorithm, while processor P_0 subtracts the partial products from c_k and computes b_k.

The flow of computation for $p = 9$ is shown in Fig.8.

Before giving the formal description (Fig.9), let us emphasize the main ideas of this algorithm.

The local variable a stores the current values of a_k. At step k the value of a_k is available to cell 0 as $a_{[-1]}$. That means, cell 0 behaves like there would be another cell at its left, and asks for the value of a in this virtual left cell. The environment external to the systolic array provides this value (a_k at step k). If $m < p$, then 0 will be provided at steps $m + 1, \ldots, p$.

C is loaded in the same way, by using the local variable c.

The leftmost cell uses a_k and c_k in order to compute b_k. Then, like in Atrubin's algorithm, a_k and b_k are pipelined rightward through the array (variables a, b), until they reach the cell $P_{k/2}$ in which they are stored. Each two pairs are stored together: cell P_k stores $(a', b') = (a_{2k}, b_{2k})$ and $(a'', b'') = (a_{2k+1}, b_{2k+1})$.

On processors $P_1, P_2, \ldots$, the local variable c stores the partial products.

The characteristics of this algorithm are:

- The number of steps is $p + 1$ (length of the quotient), and the number of processors is $1 + \lfloor p/4 \rfloor$.
- A and C are loaded into the array during the computation, LSF, one coefficient at each step (only first $p + 1$ coefficients are needed).
- B is produced on-line, LSF, in the leftmost processor P_0.
- No global broadcasting is needed.

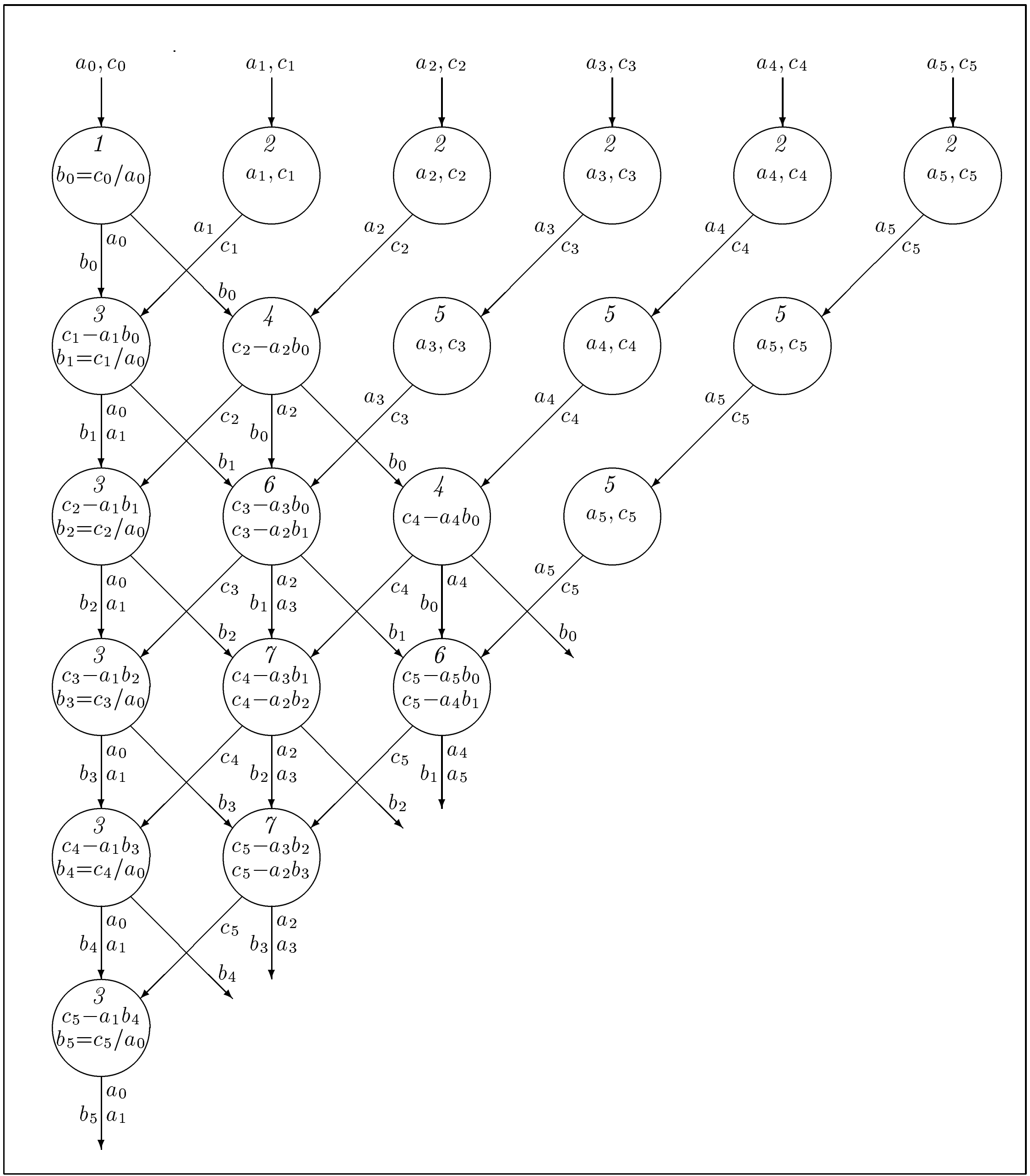

Figure 6: Data flow in version 2 of systolic polynomial exact divison.

$B \leftarrow \texttt{SysPoly.2}(C, A)$ [Systolic polynomial exact division, version 2]

$\texttt{Local:}\ a, a', b, c, s$

$$
\begin{array}{lllll}
\texttt{Input:} & a_{[k]} = a_k, & 0 \le k \le \min(n,p) & [n = \texttt{Degree}(C)] \\
& a_{[k]} = 0, & \min(n,p) < k \le p & [m = \texttt{Degree}(A)] \\
& c_{[k]} = c_k, & 0 \le k \le p & [p = n - m = \texttt{Degree}(B)] \\
& s_{[0]} = 1 & & \\
& s_{[k]} = 2, & 1 \le k \le p &
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Evolution:} & \texttt{case } s = 1\texttt{:} & a' \leftarrow a_{[+]} \quad [a_1] \\
& & b \leftarrow c/a \quad [b_0] \\
& & s \leftarrow 3 \\
& \texttt{case } s = 2\texttt{:} & \texttt{if } s_{[-]} = 2 \texttt{ then } s \leftarrow 5 \\
& & \qquad\qquad\quad \texttt{else } s \leftarrow 3 \\
& \texttt{case } s = 3\texttt{:} & b \leftarrow (c_{[+]} - a' * b)/a \\
& & [s \text{ remains 3 from now on}] \\
& \texttt{case } s = 4\texttt{:} & a \leftarrow a_{[+]} \quad [\text{store } a_{2k}] \\
& & b \leftarrow b_{[-]} \\
& & c \leftarrow c_{[+]} - a * b \\
& & s \leftarrow 6 \\
& \texttt{case } s = 5\texttt{:} & a \leftarrow a_{[+]} \\
& & c \leftarrow c_{[+]} \\
& & s \leftarrow s_{[-]} \\
& \texttt{case } s = 6\texttt{:} & a' \leftarrow a_{[+]} \quad [\text{store } a_{2k+1}] \\
& & c \leftarrow c_{[+]} - a' * b - a * b_{[-]} \\
& & b \leftarrow b_{[-]} \\
& & s \leftarrow 7 \\
& \texttt{case } s = 7\texttt{:} & c \leftarrow c_{[+]} - a' * b - a * b_{[-]} \\
& & b \leftarrow b_{[-]} \\
& & [s \text{ remains 7 from now on}]
\end{array}
$$

$$
\begin{array}{ll}
\texttt{Output:} & b_k = b_{[0]} \text{ at step } k, \quad 0 \le k \le p \\
& b_k = b_{[p-k]} \text{ after step } p, \quad 0 \le k \le p,\ 1 \le p
\end{array}
$$

Figure 7: Version 2 of the algorithm for polynomial exact division.

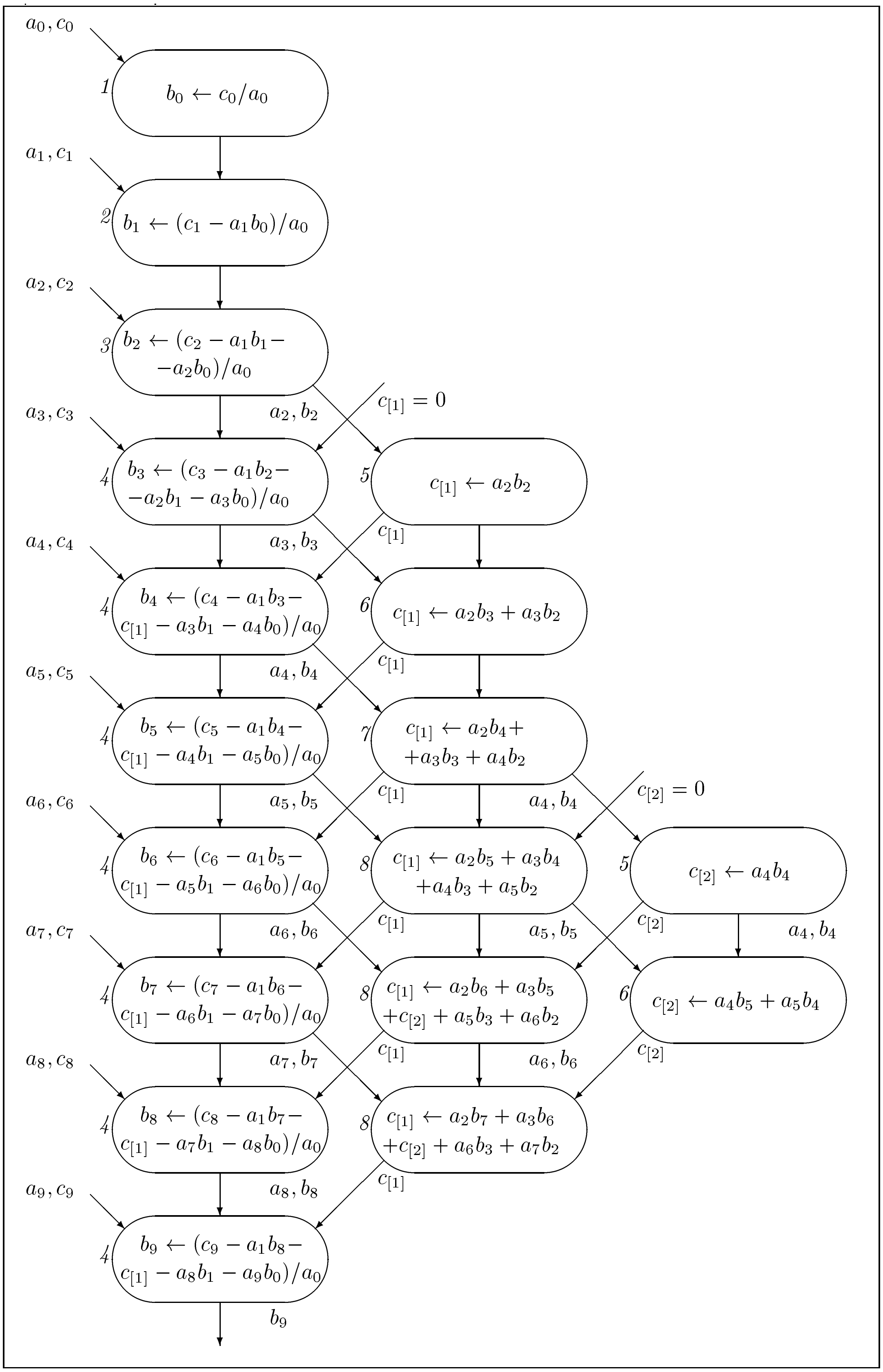

Figure 8: Data flow in version 3 of systolic polynomial exact division.

$B \leftarrow$ `SysPoly.3`(C, A) [Systolic polynomial exact division, version 3]

`Local`: $a, a', a'', b, b', b'', c, s$

`Input`: $a_{[-1]} = a_k$ at step $k \leq m$ [$n =$ `Degree`(C)]

$a_{[-1]} = 0$ at step $k > m$ [$m =$ `Degree`(A)]

$c_{[-1]} = c_k$ at step $k \leq p$ [$p = n - m =$ `Degree`(B)]

$c_{[k]} = 0, \quad 0 \leq k \leq p$

$s_{[0]} = 1$

$s_{[k]} = 0, \quad 1 \leq k \leq p$

`Evolution`: `case` $s = 0$: [only state change]

`if` $(s_{[-]} = 2$ or $s_{[-]} = 6)$

`then` $s \leftarrow 5$

[states 1 – 4 are specific to cell 0]

`case` $s = 1$: $a' \leftarrow a_{[-]}$ [a_0]

$b \leftarrow c_{[-]}/a')$ [b_0]

$b' \leftarrow b$ [b_0]

$s \leftarrow 2$

`case` $s = 2$: $a'' \leftarrow a_{[-]}$ [a_1]

$b \leftarrow (c_{[-]} - a''b)/a'$ [b_1]

$b'' \leftarrow b$ [b_1]

$s \leftarrow 3$

`case` $s = 3$: $a \leftarrow a_{[-]}$ [a_2]

$b \leftarrow (c_{[-]} - a''b'' - ab')/a'$ [b_2]

$s \leftarrow 4$

`case` $s = 4$: $b \leftarrow (c_{[-]} - a"b - c_{[+]} - a"b - a_{[-]}b')/a'$ [$b_3, b_4, \ldots$]

$a \leftarrow a_{[-]}$ [$a_3, a_4, \ldots$]

[s remains 4 from now on]

[states 5 – 8 are like in the multiplication algorithm]

`case` $s = 5$: $a' \leftarrow a_{[-]}$ [store a_{2k}]

$b' \leftarrow b_{[-]}$ [store b_{2k}]

$c \leftarrow a'b'$

$s \leftarrow 6$

`case` $s = 6$: $a'' \leftarrow a_{[-]}$ [store a_{2k+1}]

$b'' \leftarrow b_{[-]}$ [store b_{2k+1}]

$c \leftarrow a''b' + a'b''$

$s \leftarrow 7$

`case` $s = 7$: $a \leftarrow a_{[-]}$

$b \leftarrow b_{[-]}$

$c \leftarrow ab' + a''b'' + a'b$

$s \leftarrow 8$

`case` $s = 8$: $c \leftarrow a_{[-]}b' + ab" + c_{[+]} + a"b + a'b_{[-]}$

$a \leftarrow a_{[-]}$

$b \leftarrow b_{[-]}$

[s remains 8 from now on]

`Output`: $b_k = b_{[0]}$ at step k

Figure 9: Version 3 of the algorithm for polynomial exact division.

- During the computation, each pair of coefficients a_k, b_k of A and B moves rightward until it reaches the processor $P_{k/2}$, where it is stored. Also, the partial products of $A * B$ (computed by Atrubin's multiplication scheme) move leftward through the array, until they are consumed in the leftmost processor.

2 Exact division of long integers

The inputs to the exact division algorithm are:

$$C = c_0 + c_1 * \beta + c_2 * \beta^2 + \ldots + c_n * \beta^n,$$

$$A = a_0 + a_1 * \beta + a_2 * \beta^2 + \ldots + a_m * \beta^m,$$

where β is the radix ($\beta \geq 2$), $m \leq n$ and A divides C. We shall further suppose

$$a_0 \neq 0.$$

The cases when

$$a_0 = a_1 = \ldots = a_k = 0 \text{ for some } k$$

can be reduced to the previous case by some simple shift operations.

The output is:

$$B = b_0 + b_1 * \beta + b_2 * \beta^2 + \ldots + b_{n-m} * \beta^{n-m},$$

such that:

$$C = A * B.$$

2.1 Sequential algorithm

The basic idea of the algorithm introduced in [Jebelean 92] is to compute the least-significant digit b_0 of B by using only the least-significant digits a_0 and c_0 of A and C:

$$b_0 = (c_0 * (a_0)^{-1}_{\text{mod}\beta})_{\text{mod}\beta}. \tag{1}$$

After b_0 is obtained, the next coefficient b_1 can be computed by applying the same scheme to A and $(C - b_0 * A)/\beta$.

For (1) to be possible, a_0 and β must be relatively prime. This is true, for instance, if β is a prime number. However, since the radix used in the implementation of computer algebra systems is usually a power of 2, it is useful to note that this exact division scheme can be applied in this case also. For this, a preliminary right-shift of the operands is required, such that A (hence a_0) becomes odd. Since A divides C, shifting both of them with the same number of binary positions does not affect the value of the quotient.

Therefore, we shall consider from now on that a_0 and β are relatively prime.

The sequential algorithm is presented in Fig.10. Same as for polynomial algorithm, we extend A by

$$a_{m+1} = a_{m+2} = \ldots = a_{n-m} = 0$$

in the case $m < n - m$.

In contrast with the algorithm for polynomials (Fig.1), carry propagation is performed. The variable which holds the carry is y.

Also, double-digit arithmetic is used. That is, some assignments which occur in the description of the algorithms have the form

$$(h, l) \leftarrow \textit{expression}.$$

$B \leftarrow \mathtt{SeqInt}(C, A)$ [Sequential exact division of long integers]
[$n = \mathtt{Length}(C) - 1$]
[$m = \mathtt{Length}(A) - 1$]
[$p = n - m = \mathtt{Length}(B) - 1$]
$\overline{a} \leftarrow (a_0)^{-1}_{\mathrm{mod}\beta}$ [Compute inverse]
[Main loop]
$\mathtt{for}\ k = 0, 1, \ldots, p\ \mathtt{do}$
$b_k \leftarrow (\overline{a} * c_k)_{\mathrm{mod}\beta}$
$y = -\mathtt{High}(a_0 * b_k)$ [first carry]
[Inner loop]
$\mathtt{for}\ i = 1, 2, \ldots, p - k\ \mathtt{do}$
$(y, c_{k+i}) \leftarrow c_{k+i} - b_k * a_i + \mathrm{y}$

Figure 10: Sequential algorithm for long integer exact division.

This means that the result r of the *expression* is in the range $(-\beta^2, \beta^2)$, and h is assigned the most significant digit of r, while l is assigned the least significant digit of r. For instance, if $\beta = 10$, then

$$(h, l) \leftarrow 6 * 7 + 9$$

has the same effect as

$$h \leftarrow 5,$$
$$l \leftarrow 1.$$

For negative values, only the most-significant digit is signed, while the least-significant digit is always positive. Hence, if $r < 0$, then r_{high} and r_{low} are those unique integers such that:

$$-\beta < r_{\mathrm{high}} \leq 0,$$

$$0 \leq r_{\mathrm{low}} < \beta,$$

$$r = r_{\mathrm{high}} * \beta + r_{\mathrm{low}}.$$

For instance, if $\beta = 10$, then

$$(h, l) \leftarrow 9 - 6 * 7$$

has the same effect as

$$h \leftarrow -4,$$
$$l \leftarrow 7.$$

In fact, this is the way most digital computers handle this kind of operations (complement representation).

In the presentation of algorithms, we shall also use the function $\mathtt{High}$(*expression*) to select the most significant digit of the result of the *expression*.

These considerations lead to the sequential algorithm for exact division of long integers presented in Fig.10.

2.2 Systolic algorithm: Version 0

In contrast whith the polynomial algorithm, the inner loop of `SeqInt` is not suitable for parallelization: at each step, the carry y produced in the previous iteration is needed. In order to parallelize the algorithm, we introduce a vector Y for holding the carries: $y_0, y_1, \ldots, y_p$ store the "local" carries corresponding to $c_0, c_1, ..., c_p$. The parallel version of the algorithm is shown in Fig.11.

$B \leftarrow \texttt{ParInt}(C, A)$ [Parallel exact division of long integers]
[$n = \texttt{Length}(C) - 1$]
[$m = \texttt{Length}(A) - 1$]
[$p = n - m = \texttt{Length}(B) - 1$]
$\overline{a} \leftarrow (a_0)^{-1}_{\mathrm{mod}\beta}$ [Compute inverse]
$y_0 = 0$
[Main loop]
$\texttt{for } k = 0, 1, \ldots, p \texttt{ do}$
$b_k \leftarrow (\overline{a} * c_k)_{\mathrm{mod}\beta}$
$y_k = y_k - \texttt{High}(a_0 * b_k)$ [update first carry]
[Inner loop (parallel)]
$\texttt{for } i = 1, 2, \ldots, p - k \texttt{ do in parallel}$
$(y_{k+i}, c_{k+i}) \leftarrow c_{k+i} - b_k * a_i + y_{k+i-1}$

Figure 11: Parallel algorithm for long integer exact division.

In the parallel version the carries are not propagated at each step, but Y holds their values in order to be incorporated in C at the next iteration of the main loop. However, the value of c_{k+1} is computed by taking into account the two carries which affect it (cumulated in y_k). Hence, b_{k+1} will be given the correct value at the next iteration of the main loop.

The parallel version can be implemented on a systolic array, yielding the version 0 of systolic exact division of long integers (Fig.12). This is similar to `SysPoly.0` (Fig.3), except for carry handling.

A number of $p+1$ processors are used, processor P_k containing a_k, c_k and y_k ($0 \le k \le p$). At step k, the processor P_k performs: $b_k \leftarrow (\overline{a} * c_k)_{\mathrm{mod}\beta}$, and then broadcasts the value of b_k to the processors $P_{k+1}, \ldots, P_p$, such that each of them can perform $c_{k+i} \leftarrow c_{k+i} - b_k * a_i + y_{k+i-1}$ during the next step. Since P_{k+i} needs a_i, A must be pipelined rightward through the array. The variables a' and $\overline{a}$ hold the values of a_0 and $(a_0)^{-1}_{\mathrm{mod}\beta}$. At step k, processor P_k receives these values from its left neighbour and uses them for computing b_k. After k steps, $P_1, \ldots, P_k$ become idle, just holding the values of $b_1, \ldots, b_k$. Step 0 is a little different: in order to have uniformity in the subsequent steps, we introduce a leftward shift of A.

Fig.13 shows the data flow of when $\beta = 10$, $C = 45,704,126,916$, $A = 427,413$, $B = 106,932$ ($n = 10, m = 5, p = 5$). Note that only the least-significant 6 digits of C are stored in the processors, because the others are not needed in the computation.

The characteristics of this algorithm are:

- The number of steps and the number of processors is $p+1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p+1$ digits are needed). This is useful when some previous computation (for instance, GCD computation) leaves the two operands in the array.
- B is in the systolic array at the end of the computation. This is useful when B is needed in the array for some subsequent computation.
- Global broadcasting is used to send to all the cells the value of b_k. Thus, the algorithm can be implemented only on devices which allow such operation.
- During the computation, A moves rightward through the array, while C is kept steady. The carry vector Y also moves rightward through the array.

$B \leftarrow$ `SysInt.0`(C, A) [Systolic exact division of integers, version 0]

`Local`: $a, a', \overline{a}, c, y, s$

`Global`: b_G

`Input`: $a_{[k]} = a_k, \quad 0 \le k \le \min(n,p) \qquad [n = \texttt{Length}(C) - 1]$

$a_{[k]} = 0, \quad \min(n,p) < k \le p \qquad [m = \texttt{Length}(A) - 1]$

$c_{[k]} = c_k, \quad 0 \le k \le p \qquad [p = n - m = \texttt{Length}(B) - 1]$

$y_{[k]} = 0, \quad 0 \le k \le p$

$s_{[0]} = 1$

$s_{[k]} = 2, \quad 1 \le k \le p$

`Evolution`: `case` $s = 1$: $a' \leftarrow a \qquad [a_0]$

$\overline{a} \leftarrow a^{-1}_{\bmod \beta} \qquad [(a_0)^{-1}_{\bmod \beta}]$

$a \leftarrow a_{[+]} \qquad [a_1]$

$b_G \leftarrow (c * \overline{a})_{\bmod \beta} \qquad [b_0]$

$s \leftarrow 0$

`case` $s = 2$: $a \leftarrow a_{[+]}$

`if` $s_{[-]} = 2$ `then` $s \leftarrow 4$

`else` $s \leftarrow 3$

`case` $s = 3$: $a \leftarrow a_{[-]} \qquad [a_1]$

$a' \leftarrow a'_{[-]} \qquad [a_0]$

$\overline{a} \leftarrow \overline{a}_{[-]} \qquad [(a_0)^{-1}_{\bmod \beta}]$

$(y, c) \leftarrow c - a * b_G - \texttt{High}(a' * b_G) + y_{[-]}$

$b_G \leftarrow (c * \overline{a})_{\bmod \beta}$

$s \leftarrow 0$

`case` $s = 4$: $a \leftarrow a_{[-]}$

$(y, c) \leftarrow c - a * b_G + y_{[-]}$

$s \leftarrow s_{[-]}$

`case` $s = 0$: [no action]

`Output`: $b_k = b_{[k]} \quad (0 \le k \le p)$

Figure 12: Systolic algorithm for exact division of long integers: version 0.

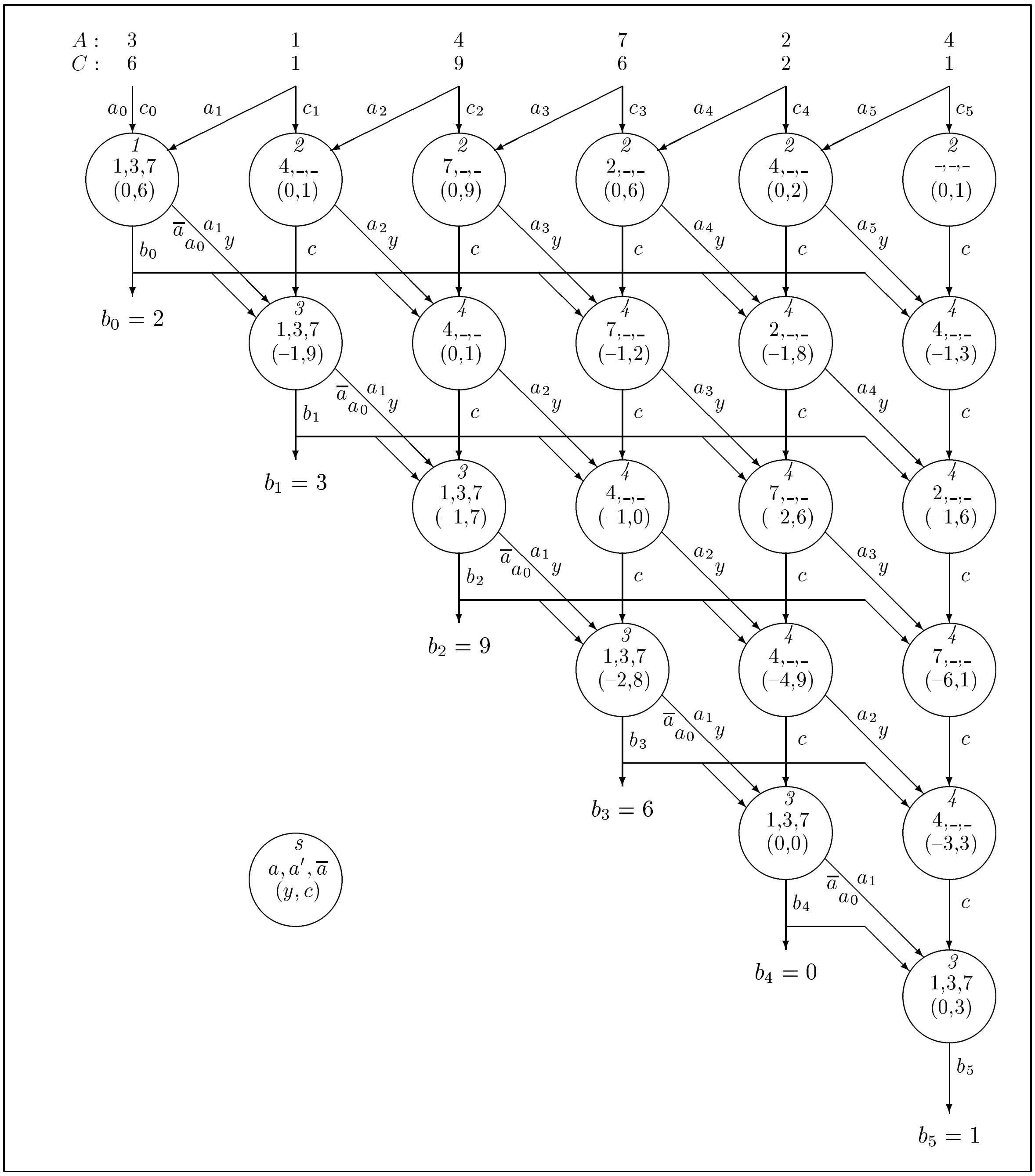

Figure 13: Data flow in version 0 of systolic exact divison of long integers.

2.3 Systolic algorithm: Version 1

By slightly changing the previous algorithm, we obtain a new scheme in which A and Y are kept steady in the array, while the updated values of C are pipelined leftward. At each step, a new digit of B is obtained in the leftmost cell.

The binary version of this algorithm was described by [Purdy, 87].

Fig.14 shows the data–flow for the same values of the inputs ($\beta = 10$, $C = 45,704,126,916$, $A = 427,413$, $B = 106,932$, $n = 10, m = 5, p = 5$).

The formal description is presented in Fig.15.

The characteristics of this algorithm are:

- The number of steps and the number of processors is $p + 1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p + 1$ digits are needed).
- B is obtained one digit at a time, LSF, in the leftmost processor. This is useful when B is needed in another unit for subsequent computation. This other unit can start to consume the least-significant digits of B before the division is finished.
- Global broadcasting is used to send to all the cells the value of b_k.
- During the computation, C moves leftward through the array, while A and Y are kept steady.

2.4 Systolic algorithm: Version 2

In order to eliminate the global broadcasting, the values of b_k must be pipelined rightward through the array.

Fig.16 depicts the flow of computation for the same example ($\beta = 10$, $C = 45,704,126,916$, $A = 427,413$, $B = 106,932$, $n = 10, m = 5, p = 5$).

The description of the algorithm is presented in Fig.17. Note that a supplementary bit is needed for storing each carry y_k. Indeed, initially

$$y_k = 0 < 2\beta$$

and by induction:

$$|c_k - a_i b_{j+1} - a_{i+1} b_j + y_k| \leq 3\beta + 2(\beta - 1)^2 < 2\beta^2.$$

The characteristics of this algorithm are:

- The number of steps and the number of processors is $p + 1$ (length of the quotient).
- A and C are in the systolic array at the beginning of computation (only first $p + 1$ digits are needed).
- B is obtained one digit at a time, LSF, in the leftmost processor. Also, at the end of the computation, B is available in the array, but in the reversed order.
- No global broadcasting is needed.
- During the computation, C moves leftward through the array, Y are kept steady, and B is pipelined rightward. Each digit a_k of A moves leftward until it reaches the processor $P_{k/2}$, where it is stored.

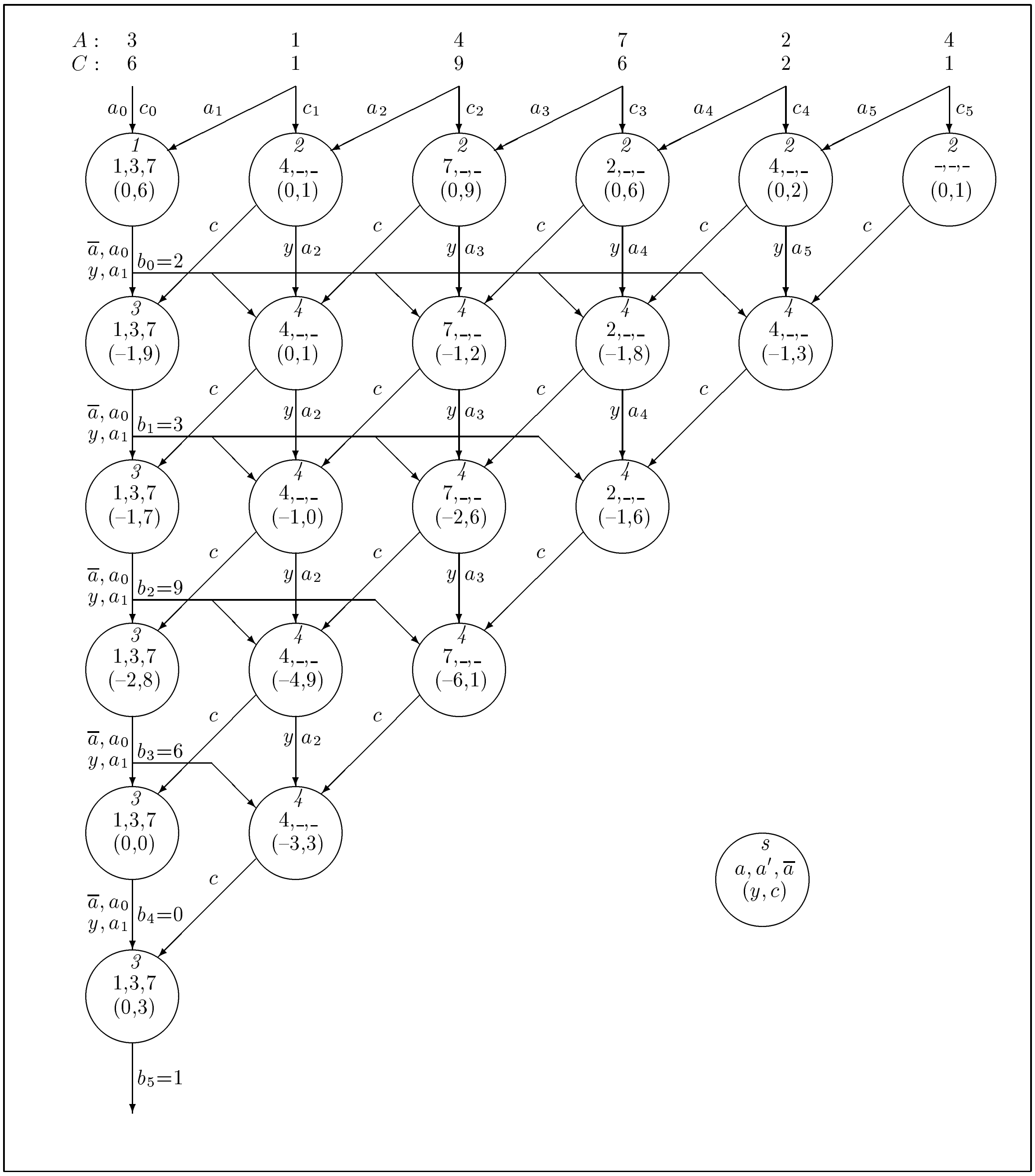

Figure 14: Data flow in version 1 of systolic exact divison of long integers.

$B \leftarrow$ `SysInt.1`(C, A) [Systolic exact division of integers, version 1]

`Local`: $a, a', \overline{a}, c, y, s$

`Global`: b_G

`Input`: $a_{[k]} = a_k, \quad 0 \le k \le \min(n,p) \qquad [n = \mathtt{Length}(C) - 1]$

$a_{[k]} = 0, \quad \min(n,p) < k \le p \qquad [m = \mathtt{Length}(A) - 1]$

$c_{[k]} = c_k, \quad 0 \le k \le p \qquad [p = n - m = \mathtt{Length}(B) - 1]$

$y_{[k]} = 0, \quad 0 \le k \le p$

$s_{[0]} = 1$

$s_{[k]} = 2, \quad 1 \le k \le p$

`Evolution`: `case` $s = 1$: $a' \leftarrow a \qquad [a_0]$

$\overline{a} \leftarrow a^{-1}_{\mathrm{mod}\beta} \qquad [(a_0)^{-1}_{\mathrm{mod}\beta}]$

$a \leftarrow a_{[+]} \qquad [a_1]$

$b_G \leftarrow (c * \overline{a})_{\mathrm{mod}\beta}$

$s \leftarrow 3$

`case` $s = 2$: $a \leftarrow a_{[+]}$

$s \leftarrow 4$

`case` $s = 3$: $(y, c) \leftarrow c_{[+]} - a * b_G - \mathtt{High}(a' * b_G) + y$

$b^G \leftarrow (c * \overline{a})_{\mathrm{mod}\beta}$

`case` $s = 4$: $(y, c) \leftarrow c_{[+]} - a * b_G + y$

`Output`: $b_k = b_{[0]}$ at step $k \quad (0 \le k \le p)$

Figure 15: Version 1 of the algorithm for exact division of integers.

2.5 Systolic algorithm: Version 3

The last version of the systolic algorithm does not need the operands within the array at the beginning of computation. Rather, A and C are fed into the array one digit at a time, LSF. Also, B is produced LSF in on-line fashion: digit b_k is produced immediately after a_k and c_k are fed into the array.

This algorithm (Fig.18, Fig.19) is a modified version of the Atrubin's multiplication scheme [Atrubin 65]. Namely, the processors $P_1, \ldots, P_p$ compute the partial products of $A * B$ as in the multiplication algorithm, while processor P_0 subtracts the partial products from c_k and computes b_k. The operations on c_k produce carries, which are stored in $Y = (y_0, y_1, \ldots)$. Two digits are needed to store each carry. At each step, the carries produced in the previous step are incorporated into c_k. This works correctly because the elements of C move leftward, while Y is kept steady.

Fig.18 shows the flow of computation for $\beta = 10$, $A = 2,590,427,413$, $B = 3,625,106,932$, $C = 9,390,576,371,709,126,916$ $(m = 18, n = 9, p = 9)$.

The description is displayed in Fig.19. The local variable a stores the current values of a_k. At step k the value of a_k is available to cell 0 as $a_{[-1]}$. That means, cell 0 behaves like there would be another cell at its left, and asks for the value of a in this virtual left cell. The environment external to the systolic array provides this value (a_k at step k). If $m < p$, then 0 will be provided at steps $m+1, \ldots, p$.

C is loaded in the same way, by using the local variable c.

The leftmost cell uses a_k and c_k in order to compute b_k. Then, like in Atrubin's algorithm, a_k and b_k are pipelined rightward through the array (variables a, b), until they reach the cell in which they are stored. Each two pairs are stored together: cell P_k stores $(a', b') = (a_{2k}, b_{2k})$ and $(a'', b'') = (a_{2k+1}, b_{2k+1})$.

The local variable $\overline{a}$ stores a_0^{-1} and it is used only in cell 0.

The local variable c stores the partial products in cells $P_1, \ldots, P_p$ and the difference between the current dividend digit and the partial product in cell P_0.

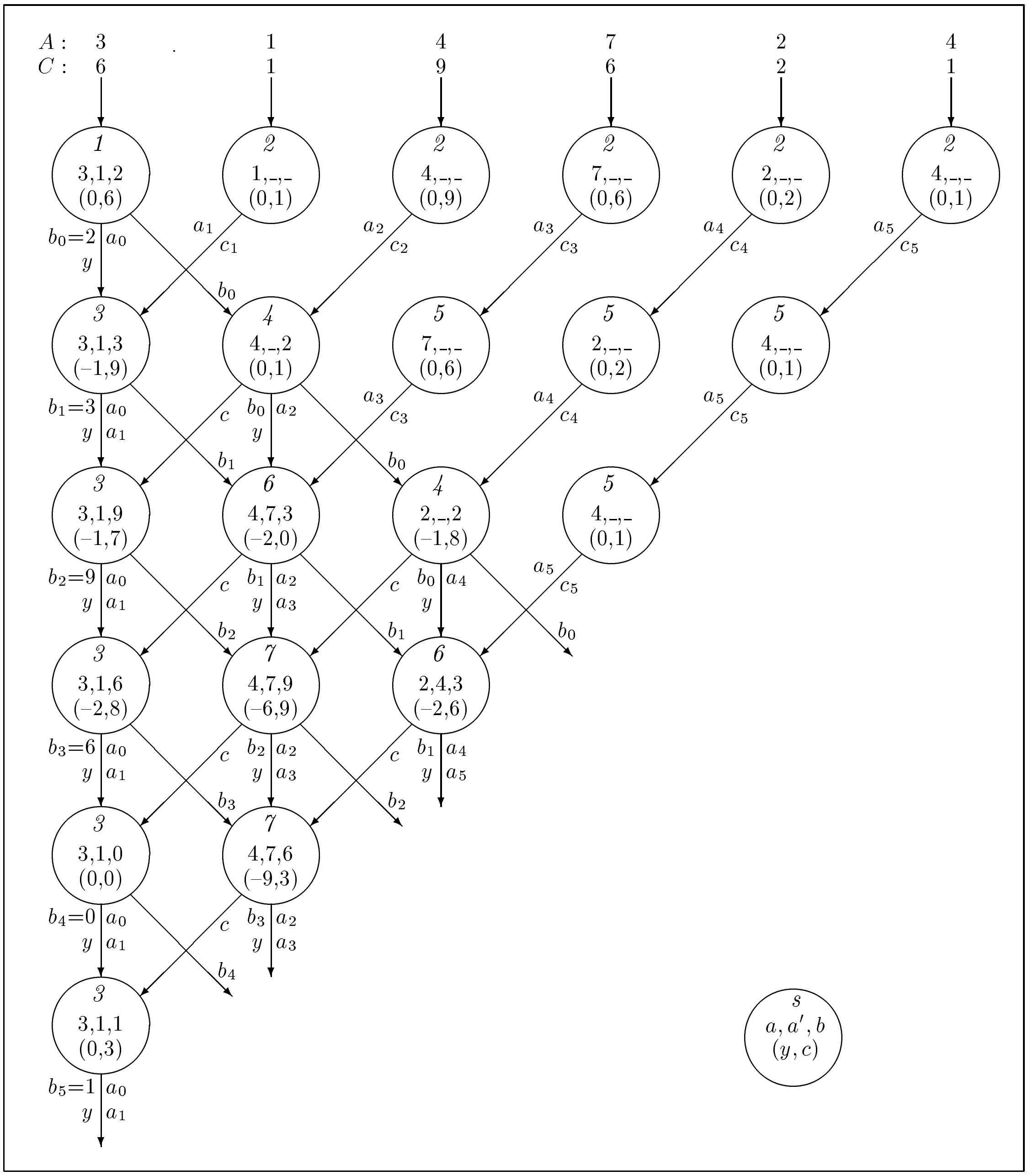

Figure 16: Data flow in version 2 of systolic exact divison of integers.

$B \leftarrow \texttt{SysInt.2}(C, A)$ [Systolic exact division of integers, version 2]

$\texttt{Local:}\ a, a', \overline{a}, b, c, y, s$

$$
\begin{array}{llll}
\texttt{Input:} & a_{[k]} = a_k, & 0 \le k \le \min(n,p) & [n = \texttt{Length}(C) - 1] \\
 & a_{[k]} = 0, & \min(n,p) < k \le p & [m = \texttt{Length}(A) - 1] \\
 & c_{[k]} = c_k, & 0 \le k \le p & [p = n - m = \texttt{Length}(B) - 1] \\
 & y_{[k]} = 0, & 0 \le k \le p & \\
 & s_{[0]} = 1 & & \\
 & s_{[k]} = 2, & 1 \le k \le p &
\end{array}
$$

$\texttt{Evolution:}$[states 1 and 2 are for the first step]

$$
\begin{array}{ll}
\texttt{case}\ s = 1: & \text{[processor 0]} \\
 & a' \leftarrow a_{[+]} \qquad [a_1] \\
 & \overline{a} \leftarrow a^{-1}_{\mathrm{mod}\beta} \qquad [(a_0)^{-1}_{\mathrm{mod}\beta}] \\
 & b \leftarrow (c * \overline{a})_{\mathrm{mod}\beta} \\
 & s \leftarrow 3 \\
\texttt{case}\ s = 2: & \text{[other processors are idle]} \\
 & \texttt{if}\ s_{[-]} = 2 \\
 & \quad \texttt{then}\ s \leftarrow 5 \\
 & \quad \texttt{else}\ s \leftarrow 4 \\
\text{[subsequent steps]} & \\
\texttt{case}\ s = 3: & \text{[processor 0]} \\
 & (y, c) \leftarrow c_{[+]} - a' * b - \texttt{High}(a * b) + y \\
 & b \leftarrow (c * \overline{a})_{\mathrm{mod}\beta} \\
 & [s \text{ remains 3 from now on}] \\
\texttt{case}\ s = 4: & \text{[first subtraction]} \\
 & a \leftarrow a_{[+]}\ [\text{store } a_{2k}] \\
 & b \leftarrow b_{[-]} \\
 & (y, c) \leftarrow c_{[+]} - a * b \\
 & s \leftarrow 6 \\
\texttt{case}\ s = 5: & [\text{just pipe } a \text{ and } c] \\
 & a \leftarrow a_{[+]} \\
 & c \leftarrow c_{[+]} \\
 & s \leftarrow s_{[-]} \\
\texttt{case}\ s = 6: & a' \leftarrow a_{[+]}\ [\text{store } a_{2k+1}] \\
 & (y, c) \leftarrow c_{[+]} - a * b - a' * b_{[-]} + y \\
 & b \leftarrow b_{[-]} \\
 & s \leftarrow 7 \\
\texttt{case}\ s = 7: & (y, c) \leftarrow c_{[+]} - a * b - a' * b_{[-]} + y \\
 & b \leftarrow b_{[-]} \\
 & [s \text{ remains 7 from now on}]
\end{array}
$$

$\texttt{Output:}\ b_k = b_{[0]}$ at step k $(0 \le k \le p)$

$\qquad b_k = b_{[p-k]}$ after step p $(0 \le k \le p,\ 1 \le p)$

Figure 17: Version 2 of the algorithm for exact division of integers.

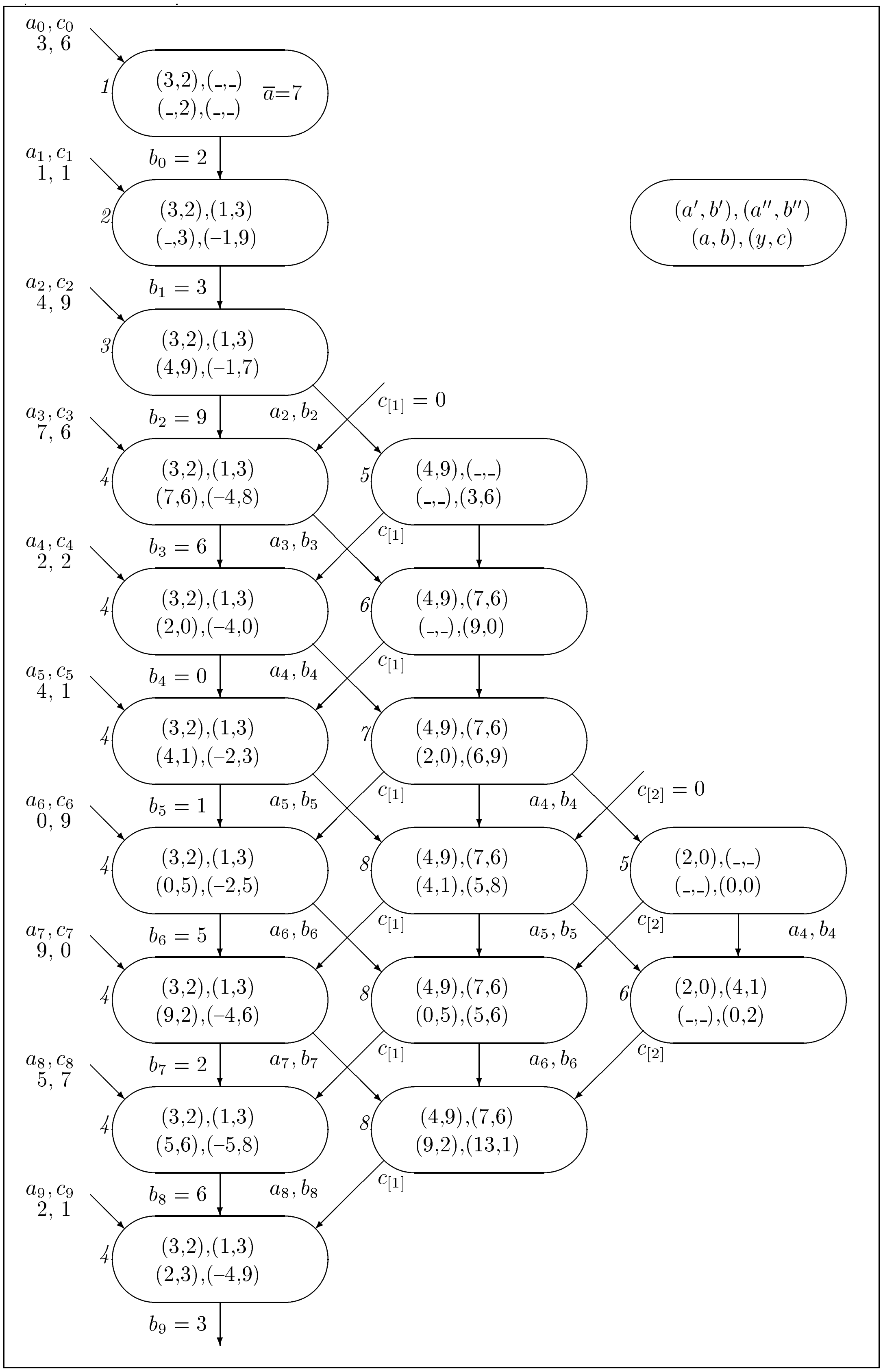

Figure 18: Data flow in version 3 of systolic exact division of long integers.

$B \leftarrow \texttt{SysPoly.3}(C, A)$ [Systolic exact division of integers, version 3]

$\texttt{Local}$: $a, a', a'', \overline{a}, b, b', b'', c, y, s$

$\texttt{Input}$: $a_{[-1]} = a_k$ at step $k \leq m$ $\quad$ $[n = \texttt{Length}(C) - 1]$
$a_{[-1]} = 0$ at step $k > m$ $\quad$ $[m = \texttt{Length}(A) - 1]$
$c_{[-1]} = c_k$ at step k $\quad$ $[p = n - m = \texttt{Length}(B) - 1]$
$c_{[k]} = 0, \quad 0 \leq k$
$s_{[0]} = 2$
$s_{[k]} = 0, \quad 1 \leq k$

$\texttt{Evolution}$: $\texttt{case}\ s = 0$: [only state change]
$\texttt{if}\ (s_{[-]} = 2$ or $s_{[-]} = 6)$
$\quad \texttt{then}\ s \leftarrow 5$

[states $1 - 4$ are specific to cell 0]

$\texttt{case}\ s = 1$: $a' \leftarrow a_{[-]}$ $\quad$ $[a_0]$
$\overline{a} \leftarrow (a')^{-1}_{\bmod \beta}$ $\quad$ $[(a_0)^{-1}_{\bmod \beta}]$
$b \leftarrow (c_{[-]}\overline{a})_{\bmod \beta}$ $\quad$ $[b_0]$
$b' \leftarrow b$ $\quad$ $[b_0]$
$s \leftarrow 2$

$\texttt{case}\ s = 2$: $a'' \leftarrow a_{[-]}$ $\quad$ $[a_1]$
$(y, c) \leftarrow c_{[-]} - \texttt{High}(a'b) - a''b$
$b \leftarrow (c\overline{a})_{\bmod \beta}$ $\quad$ $[b_1]$
$b'' \leftarrow b$ $\quad$ $[b_1]$
$s \leftarrow 3$

$\texttt{case}\ s = 3$: $a \leftarrow a_{[-]}$ $\quad$ $[a_2]$
$(y, c) \leftarrow c_{[-]} - \texttt{High}(a'b) - a''b'' - ab' + y$
$b \leftarrow (c\overline{a})_{\bmod \beta}$ $\quad$ $[b_2]$
$s \leftarrow 4$

$\texttt{case}\ s = 4$: $(y, c) \leftarrow c_{[-]} - \texttt{High}(a'b) - a''b - c[+] - ab'' - a_{[-]}b' + y$
$a \leftarrow a_{[-]}$ $\quad$ $[a_3, a_4, \ldots]$
$b \leftarrow (c\overline{a})_{\bmod \beta}$ $\quad$ $[b_3, b_4, \ldots]$
[s remains 4 from now on]

[states $5 - 8$ are like in the multiplication algorithm]

$\texttt{case}\ s = 5$: $a' \leftarrow a_{[-]}$ $\quad$ [store a_{2k}]
$b' \leftarrow b_{[-]}$ $\quad$ [store b_{2k}]
$(y, c) \leftarrow a'b'$
$s \leftarrow 6$

$\texttt{case}\ s = 6$: $a'' \leftarrow a_{[-]}$ $\quad$ [store a_{2k+1}]
$b'' \leftarrow b_{[-]}$ $\quad$ [store b_{2k+1}]
$(y, c) \leftarrow a''b' + a'b'' + y$
$s \leftarrow 7$

$\texttt{case}\ s = 7$: $a \leftarrow a_{[-]}$
$b \leftarrow b_{[-]}$
$(y, c) \leftarrow ab' + a''b'' + a'b + y$
$s \leftarrow 8$

$\texttt{case}\ s = 8$: $(y, c) \leftarrow a_{[-]}b' + ab'' + c_{[+]} + a''b + a'b_{[-]} + y$
$a \leftarrow a_{[-]}$
$b \leftarrow b_{[-]}$
[s remains 8 from now on]

$\texttt{Output}$: $b_k = b_{[0]}$ at step k

Figure 19: Version 3 of the algorithm for exact division of long integers.

The characteristics of this algorithm are:

- The number of steps is $p+1$ (length of the quotient), and the number of processors is $1 + \lfloor p/4 \rfloor$.
- A and C are loaded into the array during the computation, LSF, one digit at each step (only first $p+1$ digits are needed).
- B is produced on-line, LSF, in the leftmost processor P_0.
- No global broadcasting is needed.
- During the computation, each pair of digits a_k, b_k of A and B moves rightward until it reaches the processor $P_{k/2}$, where it is stored. C, containing the partial products of $A * B$ computed by Atrubin's multiplication scheme, moves leftward through the array, and is consumed in the leftmost processor. The carry vector Y is kept steady.

Conclusions

Four versions of systolic exact division were presented. Exact division of univariate polynomials is just a simplified variant of multiprecision exact division, because no carry handling has to be performed.

The multiprecision algorithms will work with any power of two as radix, hence they can be efficiently implemented on multiprocessor machines. This is to be contrasted with the usual approach of designing systolic arithmetic algorithms at the binary level. Also in contrast to the usual approach to systolic division, pipelineing is done least–significant digits first (least–degree coefficients first). In the case of multiprecision arithmetic, this approach avoids the redundant (signed–digit) representation.

Each version has its advantages and disadvantages over the other three, which makes it more or less suitable for a particular application:

- **Global broadcasting** is used by *Version 0* and *Version 1*, hence a basic rule of systolic computations is violated (local communications). However, this global broadcasting is used in a very limited way: only one value (the current quotient item) has to be broadcasted at each step. Also, one should note that SIMD machines usually provide an efficient global broadcasting mechanism (since the instructions have to be broadcasted anyway). The advantage of the first two versions is a more uniform loading of the processors.

  Version 2 and *Version 3* use only neighbor–to–neighbor communication, hence they are more suitable for implementation on linear–networked MIMD machines.

- **Pipelineing** of the input/output is done in different ways. In the first 3 versions, the two **inputs** are needed in the array at the beginning of computation. Hence, it is efficient to use one of these versions when some previous operation leaves the operands in the array (for instance: GCD computation). By contrast, the "on–line" *Version 3* is to be used when the operands are pipelined between different arithmetic units. This last algorithm is able to start consuming the inputs and producing the quotient before the previous operation is finished.

  The **output** is left within the array by *Version 0* and *Version 2*, and is pipelined outward through the first processor in *Version 1* and *Version 3*. Again, the choice of the right version depends on the use of the resulting quotient in subsequent computations.

References

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

[**Brent, Kung 83**] R. P. Brent, H. T. Kung — *Systolic VLSI arrays for linear-time GCD computation*, in V. Anceau, E. J. Aas – *VLSI'83*, Elsevier (North-Holland), 1983, 145 – 154.

[**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.

[**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.

[**Gex 71**] A. Gex — *Multiplier - divider cellular array*, Electronic Letters 7 (1971), 442–444.

[**Jebelean 92**] T. Jebelean — *An algorithm for exact division*, RISC–Linz Report 92-35, May 1992.

[**Kamal, Singh, Agrawal 75**] A. K. Kamal, H. Singh, D. P. Agrawal — *A generalized pipelined array*, IEEE Trans C-23 (1975), 533–536.

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

[**Preparata, Vuillemin 90**] F. P. Preparata, J. E. Vuillemin — *Practical cellular dividers*, IEEE Trans. C-39 (1990) 605-614.

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

[**Trivedi, Ercegovac 77**] K. S. Trivedi, M. D. Ercegovac — *On-line algorithms for division and multiplication*, IEEE Trans. C-26 (1977) 681–687.